

Massively Parallel Implementation
of Finite Element, Meshless and Isogeometric Analysis Methods
in Computational Mechanics

Alexander Karatarakis
PhD Dissertation
National Technical University of Athens (NTUA)



National Technical University of Athens (NTUA)
School of Civil Engineering
Institute of Structural Analysis and Antiseismic Research

**Massively Parallel Implementation of
Finite Element, Meshless and
Isogeometric Analysis Methods
in Computational Mechanics**

PhD Dissertation

by

Alexander Karatarakis

Advisor:

Professor Manolis Papadrakakis

May 2014

National Technical University of Athens
School of Civil Engineering
Institute of Structural Analysis and Antiseismic Research



**Massively Parallel Implementation of
Finite Element, Meshless and
Isogeometric Analysis Methods
in Computational Mechanics**

by Alexander Karatarakis

**Advisor:
Professor Manolis Papadrakakis**

Athens,
May 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Πολιτικών Μηχανικών
Εργαστήριο Στατικής και Αντισεισμικών Ερευνών



**Προσομοίωση κατασκευών με
πεπερασμένα στοιχεία, μη-πλεγματικές
μεθόδους και ισογεωμετρικές μεθόδους
σε περιβάλλον μαζικής πολυεπεξεργασίας**

από τον Αλέξανδρο Καραταράκη

**Επιβλέπων:
Καθηγητής Μανόλης Παπαδρακάκης**

Αθήνα,
Μάιος 2014

*Dedicated to my parents
Manolis and Anastasia,
and my brother
Aris*

© Copyright 2014
by Alexander Karatarakis
All Rights Reserved

PhD Examination Committee

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Manolis Papadrakakis
Professor
(Principal Advisor)
School of Civil Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Andreas Boudouvis
Professor
(Member of advisory committee)
School of Chemical Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Vissarion Papadopoulos
Assistant Professor
(Member of advisory committee)
School of Civil Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Kyriakos C. Giannakoglou
Professor
School of Mechanical Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Nectarios Koziris
Professor
School of Electrical and Computer Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Konstantinos V. Spiliopoulos
Associate Professor
School of Civil Engineering
National Technical University of Athens

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Nikos D. Lagaros
Assistant Professor
School of Civil Engineering
National Technical University of Athens

Abstract

The primary purpose of engineering analysis is to provide a numerical simulation of a physical phenomenon in a way that is accurate but also computationally feasible. The need to accurately simulate various physical processes in complex geometries is important, and has perplexed scientists for many years. The up-to-date simulation methods can accurately model the physical domain but often require high computational effort. A simulation needs to be performed within a reasonable time-frame with the given computational resources in order to be affordable in real-world applications. Thus, an important aspect in terms of feasibility is the efficient implementation of a simulation method that enables its application in large-scale problems. While the prevailing cost in traditional finite element analysis (FEA) is in the solution phase, the main drawback of meshless methods (MMs) and isogeometric analysis (IGA) when addressing real-world problems is the high cost for the formulation of the characteristic matrices. Therefore, in order to make them efficient in large-scale simulations, these methods require massively parallel algorithms not only for the solution phase but also for the assembly phase.

The aim of this work is to accelerate computationally expensive parts of simulation methods in a manner that is both efficient and scalable. Algorithms in the context of this aim are explored and implemented in this work. For the solution phase, domain decomposition is an attractive option as it splits the domain into several subdomains and allows their concurrent solution. As for the formulation phase, assembling the matrix by non-zero allows different parts of the matrix to be calculated in parallel. The calculations involved in a particular algorithm must be performed efficiently. Hence, calculations like matrix operations, which are omnipresent in the simulation, should be handled appropriately. Each matrix format has its own strengths and weaknesses and should be used for the task it is most suitable for. All of the above are combined with GPUs (graphics processing units), which have been attracting a lot of attention in recent years due to their remarkable performance features. GPU implementations are developed, for the solution phase in FEA and the formulation phase of MMs and IGA, which led to a great reduction of the time required for the simulation of a particular model.

The Ph.D. dissertation is organized as follows: Chapter 1 introduces the aims and objectives. Chapter 2 describes the three methods used in this work, i.e. FEM, MMs (focusing on element-free Galerkin methods - EFG) and IGA. The test examples that are used throughout this work are outlined here.

Chapter 3 is dedicated to domain decomposition solution methods. Chapter 4 presents graphics processing units (GPUs) and their characteristic properties, while Chapter 5 deals with the handling of matrices that are frequently encountered in simulation implementations. Chapter 6 contains the hybrid CPU-GPU implementation of the FETI domain decomposition method along with supporting numerical results. Relations between the basic entities (nodes, Gauss points, control points) are discussed in Chapter 7. Chapter 8 is dedicated to the formulation of the characteristic matrices of the simulation methods. Chapter 9 concludes with an overview of the present work, followed by the appendix, which includes supporting material, and bibliography.

Σύντομη Περίληψη

Η αριθμητική προσομοίωση κατασκευών και άλλων φορέων πρέπει να γίνεται με ένα τρόπο που προσφέρει ικανοποιητική ακρίβεια ενώ είναι υπολογιστικά εφικτός. Σε περιπτώσεις πολύπλοκης γεωμετρίας, η ακριβής προσομοίωση του φορέα είναι ένα από τα σημαντικότερα προβλήματα που αντιμετωπίζουν οι μηχανικοί. Οι σύγχρονες μέθοδοι προσομοίωσης μπορούν να προσφέρουν την επιθυμητή ακρίβεια αλλά πολλές φορές έχουν υψηλό υπολογιστικό κόστος. Για να είναι μια προσομοίωση εφαρμόσιμη σε πραγματικά προβλήματα, θα πρέπει να πραγματοποιείται σε λογικά υπολογιστικά χρονικά πλαίσια. Επομένως, ένας σημαντικός παράγοντας για την εφαρμογή των προσομοιώσεων στην πράξη είναι η αποδοτική υλοποίησή τους, η οποία θα επιτρέψει την εφαρμογή τους σε προβλήματα μεγάλης κλίμακας. Στις κλασικές μεθόδους πεπερασμένων στοιχείων (FEA) το μεγαλύτερο κόστος βρίσκεται στην επίλυση των αλγεβρικών εξισώσεων. Σε μη πλεγματικές μεθόδους (MMs) καθώς και στην ισογεωμετρική ανάλυση (IGA), το κόστος για την κατασκευή των χαρακτηριστικών μητρώων (π.χ. μητρώο στιβαρότητας) είναι ιδιαίτερα υψηλό. Επομένως, για να μπορούν αυτές οι μέθοδοι να αξιοποιηθούν σε προβλήματα μεγάλης κλίμακας, απαιτούνται τεχνικές μαζικής πολυεπεξεργασίας όχι μόνο για την επίλυση αλλά και για τη φάση κατασκευής των χαρακτηριστικών μητρώων, τα οποία απαιτούν αριθμητική ολοκλήρωση.

Ο σκοπός της παρούσας διατριβής είναι η επιτάχυνση των υπολογιστικά απαιτητικών φάσεων των μεθόδων αριθμητικής προσομοίωσης με βασικά κριτήρια την αποδοτικότητα και επεκτασιμότητα σε παράλληλο υπολογιστικό περιβάλλον. Για την επίλυση των εξισώσεων, οι μέθοδοι υποφορέων είναι ιδιαίτερα ελκυστικές καθώς χωρίζουν το φορέα σε πολλούς υποφορείς και επιτρέπουν την ταυτόχρονη επίλυσή τους. Όσον αφορά τη φάση κατασκευής των χαρακτηριστικών μητρώων, ο υπολογισμός με βάση τα μη μηδενικά στοιχεία του μητρώου επιτρέπει την παράλληλη υλοποίησή τους. Οι αριθμητικές πράξεις που πραγματοποιούνται κατά την εκτέλεση ενός αλγορίθμου πρέπει να γίνονται αποδοτικά. Επομένως, υπολογισμοί όπως πράξεις με μητρώα θέλουν ιδιαίτερη προσοχή. Κάθε τύπος μητρώου είναι κατάλληλος για διαφορετικές λειτουργίες και πρέπει να χρησιμοποιείται κατάλληλα. Όλα τα παραπάνω συνδυάζονται με τις κάρτες γραφικών (GPUs) οι οποίες έχουμε εξαιρετικές δυνατότητες για παράλληλους υπολογισμούς. Σε αυτή τη διατριβή υλοποιούνται κώδικες για κάρτες γραφικών για τη φάση επίλυσης στη μέθοδο των πεπερασμένων στοιχείων καθώς και τη φάση κατασκευής των χαρακτηριστικών μητρώων στις μη-πλεγματικές και στις ισογεωμετρικές μεθόδους με σκοπό τη σημαντική μείωση του χρόνου εκτέλεσης της

προσομοίωσης.

Η διατριβή οργανώνεται ως εξής: στο εισαγωγικό κεφάλαιο 1 παρουσιάζονται οι στόχοι και το αντικείμενο της διατριβής. Το κεφάλαιο 2 περιγράφει τις τρεις αριθμητικές μεθόδους που χρησιμοποιούνται, δηλαδή η μέθοδος πεπερασμένων στοιχείων, οι μη-πλεγματικές μέθοδοι καθώς και η μέθοδος ισογεωμετρικής ανάλυσης, ενώ παρουσιάζονται και τα παραδείγματα που θα χρησιμοποιηθούν και χρονομετρηθούν στα επόμενα κεφάλαια. Το κεφάλαιο 3 είναι αφιερωμένο στις μεθόδους υποφορέων. Το κεφάλαιο 4 παρουσιάζει τις κάρτες γραφικών και τα χαρακτηριστικά τους στοιχεία ενώ το κεφάλαιο 5 αναλύει το χειρισμό των μητρώων και τα διάφορα είδη που χρησιμοποιούνται. Το κεφάλαιο 6 περιλαμβάνει την υλοποίηση της μεθόδου υποφορέων FETI σε υβριδικό (CPU-GPU) περιβάλλον. Οι σχέσεις μεταξύ των βασικών οντοτήτων (κόμβοι, σημεία Gauss, σημεία ελέγχου) αναλύονται στο κεφάλαιο 7. Το κεφάλαιο 8 εστιάζει στη μορφοποίηση των χαρακτηριστικών μητρώων των μεθόδων προσομοίωσης. Το κεφάλαιο 9 κλείνει με μια ανακεφαλαίωση αυτής της διατριβής και ακολουθεί παράρτημα με υποστηρικτικό υλικό και βιβλιογραφία.

Εκτενής Περίληψη

1 Εισαγωγή

Η αριθμητική προσομοίωση κατασκευών και άλλων φορέων πρέπει να γίνεται με ένα τρόπο που προσφέρει ικανοποιητική ακρίβεια ενώ είναι υπολογιστικά εφικτός. Σε περιπτώσεις πολύπλοκης γεωμετρίας, η ακριβής προσομοίωση του φορέα είναι ένα από τα σημαντικότερα προβλήματα που αντιμετωπίζουν οι μηχανικοί. Οι σύγχρονες μέθοδοι προσομοίωσης μπορούν να προσφέρουν την επιθυμητή ακρίβεια αλλά πολλές φορές έχουν υψηλό υπολογιστικό κόστος. Για να είναι μια προσομοίωση εφαρμόσιμη σε πραγματικά προβλήματα, θα πρέπει να πραγματοποιείται σε λογικά υπολογιστικά χρονικά πλαίσια. Επομένως, ένας σημαντικός παράγοντας για την εφαρμογή των προσομοιώσεων στην πράξη είναι η αποδοτική υλοποίησή τους, η οποία θα επιτρέψει την εφαρμογή τους σε προβλήματα μεγάλης κλίμακας. Στις κλασικές μεθόδους πεπερασμένων στοιχείων (FEA) το μεγαλύτερο κόστος βρίσκεται στην επίλυση των αλγεβρικών εξισώσεων. Σε μη πλεγματικές μεθόδους (MMs) καθώς και στην ισογεωμετρική ανάλυση (IGA), το κόστος για την κατασκευή των χαρακτηριστικών μητρώων (π.χ. μητρώο στιβαρότητας) είναι ιδιαίτερα υψηλό. Επομένως, για να μπορούν αυτές οι μέθοδοι να αξιοποιηθούν σε προβλήματα μεγάλης κλίμακας, απαιτούνται τεχνικές μαζικής πολυεπεξεργασίας όχι μόνο για την επίλυση αλλά και για τη φάση κατασκευής των χαρακτηριστικών μητρώων, τα οποία απαιτούν αριθμητική ολοκλήρωση.

Ο σκοπός της παρούσας διατριβής είναι η επιτάχυνση των υπολογιστικά απαιτητικών φάσεων των μεθόδων αριθμητικής προσομοίωσης με βασικά κριτήρια την αποδοτικότητα και επεκτασιμότητα σε παράλληλο υπολογιστικό περιβάλλον. Για την επίλυση των εξισώσεων, οι μέθοδοι υποφορέων είναι ιδιαίτερα ελκυστικές καθώς χωρίζουν το φορέα σε πολλούς υποφορείς και επιτρέπουν την ταυτόχρονη επίλυσή τους. Όσον αφορά τη φάση κατασκευής των χαρακτηριστικών μητρώων, ο υπολογισμός με βάση τα μη μηδενικά στοιχεία του μητρώου επιτρέπει την παράλληλη υλοποίησή τους. Οι αριθμητικές πράξεις που πραγματοποιούνται κατά την εκτέλεση ενός αλγορίθμου πρέπει να γίνονται αποδοτικά. Επομένως, υπολογισμοί όπως πράξεις με μητρώα θέλουν ιδιαίτερη προσοχή. Κάθε τύπος μητρώου είναι κατάλληλος για διαφορετικές λειτουργίες και πρέπει να χρησιμοποιείται κατάλληλα. Όλα τα παραπάνω συνδυάζονται με τις κάρτες γραφικών (GPUs) οι οποίες έχουμε εξαιρετικές δυνατότητες για παράλληλους υπολογισμούς. Σε αυτή τη διατριβή υλοποιούνται κώδικες για κάρτες γραφικών για τη φάση επίλυσης στη μέθοδο των πεπερασμένων στοιχείων καθώς και τη φάση κατασκευής των χαρακτηριστικών μητρώων στις μη-πλεγματικές και στις ισογεωμετρικές μεθόδους με σκοπό τη σημαντική μείωση του χρόνου εκτέλεσης της

προσομοίωσης.

2 Μέθοδοι προσομοίωσης

Η ανάγκη ακριβούς προσομοίωσης διαφόρων φυσικών φαινομένων σε πολύπλοκες γεωμετρίες είναι σημαντική και έχει αποτελέσει αντικείμενο εντατικής έρευνας από επιστήμονες και μηχανικούς. Η πιο διαδεδομένη μέθοδος προσομοίωσης είναι η μέθοδος πεπερασμένων στοιχείων (FEM/FEA). Την τελευταία δεκαετία, δύο ακόμα μέθοδοι προσομοίωσης έχουν τραβήξει την προσοχή της επιστημονικής κοινότητας: οι μη-πλεγματικές/meshless μέθοδοι (MMs) και η ισογεωμετρική ανάλυση (IGA). Και οι δύο έχουν πλεονεκτήματα σε σχέση με τα πεπερασμένα στοιχεία, έχουν όμως και κάποιες αδυναμίες. Ένα από τα μειονεκτήματα και των δύο μεθόδων είναι ότι έχουν σημαντικά αυξημένο κόστος για τη μορφοποίηση των χαρακτηριστικών μητρώων.

2.1 Μη-πλεγματικές/Meshless μέθοδοι

Υπάρχουν πολλές αριθμητικές μέθοδοι προσομοίωσης για την επίλυση προβλημάτων μηχανικής και οι μέθοδοι που βασίζονται σε πλέγματα (δίκτυα) χρησιμοποιούνται ευρέως. Σε τέτοιες μεθόδους, όπως πεπερασμένα στοιχεία, πεπερασμένων διαφορών και πεπερασμένων όγκων, κάθε σημείο έχει σταθερό αριθμό προκαθορισμένων “γειτόνων”. Σε προσομοιώσεις όπου το υλικό προς προσομοίωση μπορεί να κινηθεί (όπως στην υπολογιστική ρευστοδυναμική) ή σε περιπτώσεις όπου το υλικό μπορεί να υποστεί μεγάλες παραμορφώσεις (όπως σε προσομοιώσεις πλαστικών υλικών), η συνδεσιμότητα του δικτύου είναι δύσκολο να διατηρηθεί χωρίς την εισαγωγή σφαλμάτων στην προσομοίωση. Αν το δίκτυο εκφυλιστεί, τα αποτελέσματα θα έχουν ανακρίβειες. Η κατάσταση μπορεί να βελτιωθεί με χρήση τεχνικών που αναδιαμορφώνουν το δίκτυο, αλλά ακόμα και σε αυτή την περίπτωση εισάγονται σφάλματα ενώ αυξάνεται και το υπολογιστικό κόστος. Επιπλέον, πολύπλοκες γεωμετρίες είναι δύσκολο να προσομοιωθούν με μεθόδους βασισμένες σε πλέγματα.

Οι μη-πλεγματικές μέθοδοι (MMs) είναι μια ιδιαίτερα ελκυστική οικογένεια μεθόδων λόγω του ότι είναι απλές, ακριβείς και δε χρειάζονται δίκτυο. Πιο συγκεκριμένα, δε χρειάζονται δίκτυο που να συνδέει τα χαρακτηριστικά σημεία του φορέα που προσομοιώνεται. Οι μη-πλεγματικές μέθοδοι επιτρέπουν την προσομοίωση προβλημάτων που θα ήταν δύσκολη σε άλλη περίπτωση. Προσφέρουν λύσεις με αυξημένη ακρίβεια, ενώ αποφεύγουν τους περιορισμούς και τις αδυναμίες που έχουν σχέση με το πλέγμα. Επιπλέον, μειώνεται ο χρόνος που απαιτείται για ανθρώπινες

επεμβάσεις σε σχέση με τις πλεγματικές μεθόδους που πολλές φορές τις χρειάζονται για τη δημιουργία χρήσιμων δικτύων. Οι μη-πλεγματικές μέθοδοι μπορούν εύκολα να χειριστούν τη δημιουργία/καταστροφή χαρακτηριστικών σημείων κατά τη διάρκεια της προσομοίωσης, μεγάλες παραμορφώσεις καθώς και ασυνέχειες που δεν είναι ευθυγραμμισμένες με τις πλευρές των στοιχείων. Η υψηλότερη ποιότητα της ανάλυσης συνοδεύεται όμως και από υψηλό υπολογιστικό κόστος για την κατασκευή των μητρώων (κυρίως) αλλά και την επίλυση των αλγεβρικών εξισώσεων. Συγκεκριμένα, τα χαρακτηριστικά μητρώα έχουν πιο εκτεταμένο εύρος ζώνης (bandwidth), είναι πιο πυκνά και η πολυπλοκότητα της μορφοποίησης είναι σημαντικά αυξημένη.

2.1.1 Παραδείγματα EFG

Τα παραδείγματα που αναλύονται είναι με τη μέθοδο element-free Galerkin (EFG). Σε σχέση με άλλους γεωμετρικούς φορείς με την ίδια πυκνότητα (όσον αφορά τον αριθμό των σημείων σε μια συγκεκριμένη περιοχή), οι επιλεγμένοι φορείς μεγιστοποιούν τον αριθμό των συσχετίσεων μεταξύ κόμβων-σημείων Gauss και μεταξύ κόμβων-κόμβων. Τα παραδείγματα είναι προβλήματα γραμμικής ελαστικότητας σε 2D (τετράγωνα) και 3D (κύβοι). Λεπτομέρειες για τα παραδείγματα δίνονται στον πίνακα 2.1. Σε όλες τις περιπτώσεις, το πεδίο επιρροής είναι ορθογωνικό με παράμετρο απόστασης 2.5.

EFG Example	Nodes	dof	Gauss points
2D-1	25,921	51,842	102,400
2D-2	76,125	152,250	300,304
2D-3	126,025	252,050	501,264
3D-1	9,261	27,783	64,000
3D-2	19,683	59,049	140,608
3D-3	35,937	107,811	262,144

Table 2.1: 2D και 3D παραδείγματα EFG

2.2 Ισογεωμετρική ανάλυση

Η ισογεωμετρική ανάλυση (Isogeometric Analysis - IGA) μπορεί να λύσει προβλήματα συνοριακών τιμών χρησιμοποιώντας τις ίδιες συναρτήσεις σχήματος που έχουν υιοθετηθεί από τη σχεδιαστική (CAD) κοινότητα για την περιγραφή της γεωμετρίας και για τη δημιουργία της αριθμητικής προσέγγισης της επίλυσης. Παρά την πολλά υποσχόμενη μεθοδολογία και πλεονεκτήματα σε σχέση με τα πεπερασμένα στοιχεία, ο υπολογισμός του μητρώου δυσκαμψίας,

μάζας και απόσβεσης είναι πιο δυσχερής. Λόγω της υψηλότερης συνέχειας μεταξύ στοιχείων, η μέθοδος παράγει πολύ περισσότερα στοιχεία από τη μέθοδο των πεπερασμένων στοιχείων για τον ίδιο αριθμό βαθμών ελευθερίας. Αυτό συνεπάγεται μεγαλύτερο αριθμό σημείων Gauss και κατ' επέκταση αύξηση του υπολογιστικού κόστους για την κατασκευή των χαρακτηριστικών μητρώων.

2.2.1 Παραδείγματα IGA

Τα παραδείγματα που εξετάζονται είναι τετράγωνα και κύβοι στον παραμετρικό χώρο ώστε να μεγιστοποιούν τον αριθμό των αλληλοσυσχετίσεων σε σχέση με άλλους ορθογωνικούς φορείς. Τα παραδείγματα είναι σε 2D και 3D γραμμική ελαστικότητα. Οι συναρτήσεις σχήματος που χρησιμοποιούνται είναι βασισμένες σε NURBS (non-uniform rational B-splines) και λεπτομέρειες για τα παραδείγματα δίνονται στον πίνακα 2.2. Το παράδειγμα P_{i-j} αντιστοιχεί σε τάξη $p=i$ της συνάρτησης βάσης, ενώ μεγαλύτερο j υποδηλώνει μεγαλύτερο παράδειγμα σε σύγκριση με τα παραδείγματα της ίδιας ομάδας.

IGA Example	p	n	Control points	dof	Elements	Gauss point per element	Gauss points	
2D	P2-1	2	225	50,625	101,250	49,729	9	447,561
	P2-2	2	500	250,000	500,000	248,004	9	2,232,036
	P2-3	2	633	400,689	801,378	398,161	9	3,583,449
	P3-1	3	225	50,625	101,250	49,284	16	788,544
	P3-2	3	320	102,400	204,800	100,489	16	1,607,824
	P3-3	3	388	150,544	301,088	148,225	16	2,371,600
	P4-1	4	160	25,600	51,200	24,336	25	608,400
	P4-2	4	225	50,625	101,250	48,841	25	1,221,025
	P4-3	4	275	75,625	151,250	73,441	25	1,836,025
3D	P2-1	2	19	6,859	20,577	4,913	27	132,651
	P2-2	2	26	17,576	52,728	13,824	27	373,248
	P2-3	2	33	35,937	107,811	29,791	27	804,357
	P3-1	3	19	6,859	20,577	4,096	64	262,144
	P3-2	3	21	9,261	27,783	5,832	64	373,248
	P3-3	3	26	17,576	52,728	12,167	64	778,688
	P4-1	4	15	3,375	10,125	1,331	125	166,375
	P4-2	4	17	4,913	14,739	2,197	125	274,625
	P4-3	4	19	6,859	20,577	3,375	125	421,875

Table 2.2: 2D και 3D παραδείγματα IGA

2.3 Παραδείγματα FEA

Η απόδοση της φάσης επίλυσης με πεπερασμένα στοιχεία παρουσιάζεται μέσω παραμετρικής ανάλυσης 3D προβλημάτων γραμμικής ελαστικότητας σε έναν κύβο. Ο φορέας είναι πλήρως δεσμευμένος στην κάτω επιφάνεια, και μερικώς δεσμευμένος κατά τις οριζόντιες διευθύνσεις στις πλαϊνές πλευρές ενώ η πάνω επιφάνεια δέχεται ισοκαταναμημένο φορτίο. Ο φορέας είναι διαχωρισμένος με 8-κομβικά εξαεδρικά στοιχεία. Το τελικό σύστημα έχει 1,058,610 βαθμούς ελευθερίας (β.ε). Η επίλυση γίνεται με μεθόδους υποφορέων, οπότε ο φορέας χωρίζεται σε υποφορείς, ο αριθμός των οποίων κυμαίνεται από 125 μέχρι 2744. Ο διαχωρισμός σε 125 υποφορείς φαίνεται στο σχήμα 2.1. Ο πίνακας 2.3 δείχνει τους β.ε του κάθε υποφορέα καθώς και τους β.ε. του συνοριακού προβλήματος για τους διάφορους διαχωρισμούς σε υποφορείς.

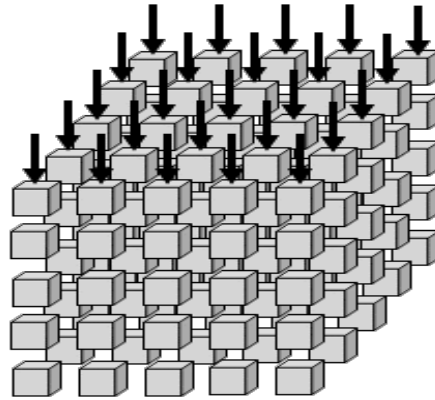


Fig. 2.1.: Διαχωρισμός του φορέα σε 125 υποφορείς

Number of subdomains	Subdomain dof	Interface dof
125	10,119	49,920
175	7,419	76,800
245	5,439	118,080
343	3,987	181,440
490	2,898	276,480
700	2,106	421,200
1000	1,530	641,520
1400	1,146	935,280
1960	858	1,363,440
2744	642	1,987,440

Table 2.3: Παραδείγματα FEA

3 Μέθοδοι υποφορέων

Οι μέθοδοι υποφορέων (domain decomposition methods - DDM) αποτελούν μια από τις βασικότερες κατηγορίες μεθόδων επίλυσης για πολλά προβλήματα προσομοίωσης στην εφαρμοσμένη μηχανική. Η κύρια συνοριακή μέθοδος (primal DDM) οδηγεί στην επίλυση του αρχικού συστήματος μέσω της επίλυσης του συνοριακού προβλήματος “κύριων” μεταβλητών (συνήθως μετατοπίσεις) μετά την απαλοιφή όλων των εσωτερικών βαθμών ελευθερίας (β.ε) των υποφορέων. Η δυϊκή συνοριακή μέθοδος (dual DDM) υπολογίζει τους πολλαπλασιαστές Lagrange που απαιτούνται για την επιβολή της συνέχειας μεταξύ των υποφορέων μετά την απαλοιφή όλων των β.ε των υποφορέων (και εσωτερικών αλλά και συνοριακών). Και οι δύο κατηγορίες, κύρια και δυϊκή, μελετούνται διεξοδικά και έχουν ενσωματωθεί σε εμπορικούς κώδικες.

Ο φορέας του σχήματος 3.1 χωρίζεται σε δύο υποφορείς. Οι συνοριακοί β.ε σημειώνονται με **b** (boundary). Οι υπόλοιποι β.ε θεωρούνται εσωτερικοί και συμβολίζονται με **i** (internal).

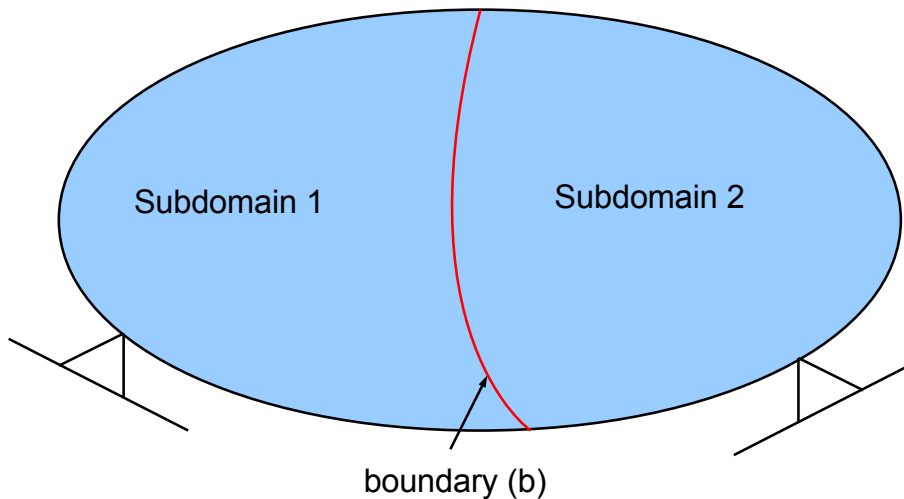


Fig. 3.1: Φορέας χωρισμένος σε δύο υποφορείς

3.1 Η κύρια συνοριακή μέθοδος

Η κύρια συνοριακή μέθοδος μειώνει τη διάσταση του συστήματος με χρήση στατικής συμπίκνωσης. Το αρχικό σύστημα είναι:

$$\mathbf{K} \mathbf{u} = \mathbf{f} \quad (3.1)$$

Αν οι εσωτερικοί β.ε του κάθε υποφορέα αριθμηθούν πρώτα, αφήνοντας τους συνοριακούς β.ε για το τέλος, τότε το σύστημα παίρνει τη μορφή:

$$\begin{bmatrix} \mathbf{K}_{ii}^{(1)} & \dots & 0 & \mathbf{K}_{ib}^{(1)} \\ & \mathbf{K}_{ii}^{(2)} & \dots & \mathbf{K}_{ib}^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \mathbf{K}_{ib}^{(n_s)} \\ \hline \mathbf{K}_{bi}^{(1)} & \mathbf{K}_{bi}^{(2)} & \dots & \mathbf{K}_{bi}^{(n_s)} & \mathbf{K}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{u}_i^{(1)} \\ \mathbf{u}_i^{(2)} \\ \vdots \\ \mathbf{u}_i^{(n_s)} \\ \mathbf{u}_b \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i^{(1)} \\ \mathbf{f}_i^{(2)} \\ \vdots \\ \mathbf{f}_i^{(n_s)} \\ \mathbf{f}_b \end{bmatrix} \quad (3.2)$$

$$\mathbf{u}_i^s = (\mathbf{K}_{ii}^s)^{-1} \mathbf{f}_i^s - (\mathbf{K}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \mathbf{u}_b \quad (3.3)$$

Το μητρώο δυσκαμψίας του συνόρου και το διάνυσμα των δυνάμεων είναι:

$$\mathbf{K}_{bb} = \sum_{s=1}^{n_s} \mathbf{K}_{bb}^s \quad \mathbf{f}_b = \sum_{s=1}^{n_s} \mathbf{f}_b^s \quad (3.4)$$

Στην κύρια συνοριακή μέθοδο, όλοι οι εσωτερικοί β.ε συμπεκνώνονται στους συνοριακούς β.ε. Η στατική συμπίκνωση δίνεται από την παρακάτω σχέση:

$$\underbrace{\left(\mathbf{K}_{bb} - \sum_{s=1}^{n_s} \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \right)}_{\hat{\mathbf{K}}_c \equiv \mathbf{S}} \mathbf{u}_b = \underbrace{\mathbf{f}_b - \sum_{s=1}^{n_s} \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{f}_i^s}_{\hat{\mathbf{f}}_c \equiv \hat{\mathbf{f}}_b} \quad (3.5)$$

$$\mathbf{S} \mathbf{u}_b = \hat{\mathbf{f}}_b \quad (3.6)$$

Το μητρώο \mathbf{S} αναφέρεται στη βιβλιογραφία ως “Schur complement”.

3.2 Η δυϊκή συνοριακή μέθοδος

Στη δυϊκή συνοριακή μέθοδο, μορφώνεται και επιλύεται το συνοριακό πρόβλημα υποφορέων στο οποίο άγνωστοι είναι οι πολλαπλασιαστές Lagrange οι οποίοι αντιπροσωπεύουν τις ενδοσυνοριακές δυνάμεις μεταξύ των υποφορέων. Η μέθοδος αναφέρεται συχνά στη βιβλιογραφία ως FETI (finite element tearing and interconnecting) και έχει αποδειχτεί ότι είναι πολύ αποδοτική για την επίλυση προβλημάτων μεγάλης κλίμακας σε παράλληλα συστήματα με κοινή και κατανεμημένη μνήμη.

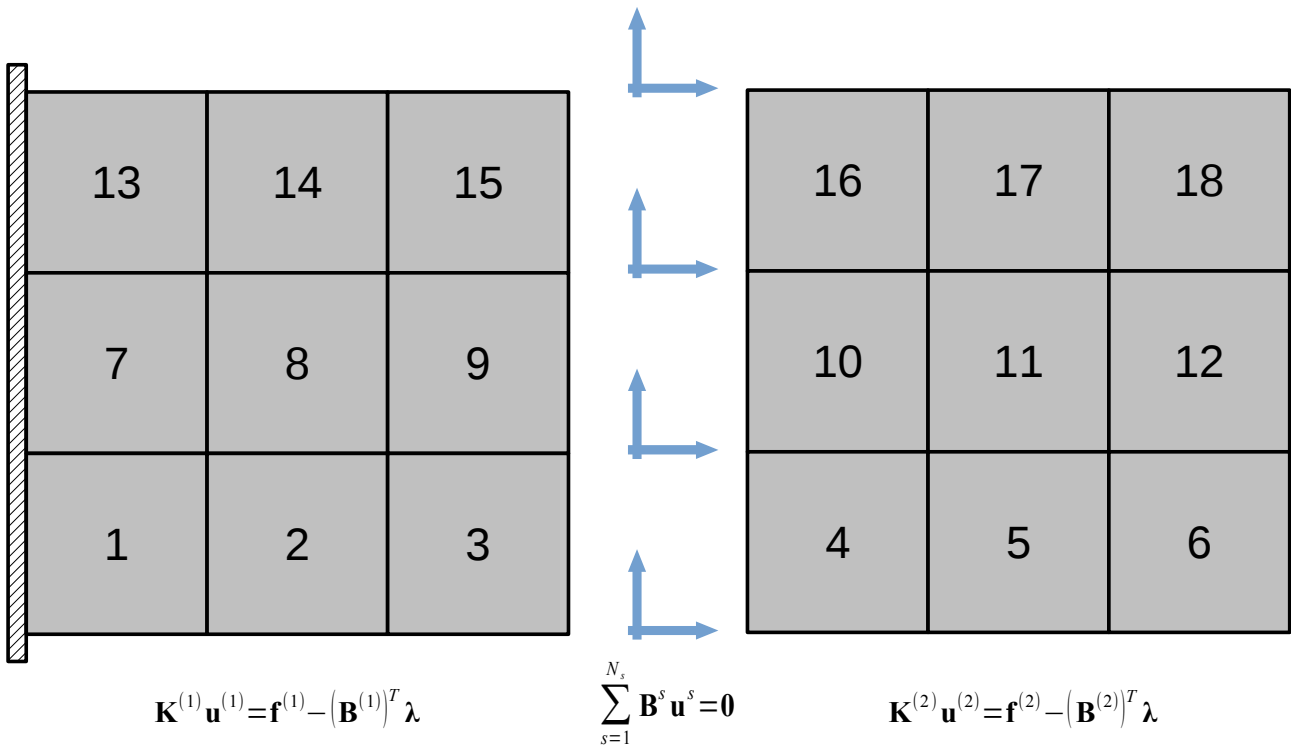


Fig. 3.2: Εξισώσεις υποφορέων και περιορισμοί

Οι εξισώσεις ισορροπίας του χωρισμένου φορέα είναι (γενική μορφή + παράδειγμα με 2 υποφορείς):

$$\mathbf{K}^g \mathbf{u}^g = \mathbf{f}^g \quad \begin{bmatrix} \mathbf{K}^{(1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}^{(2)} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \end{bmatrix} \quad (3.7)$$

Για τη διατήρηση της συνέχειας του φορέα, οι μετατοπίσεις των συνοριακών βαθμών πρέπει να είναι ίσες και για τους δύο υποφορείς.

$$\sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0} \qquad \mathbf{u}_{boundary}^{(1)} = \mathbf{u}_{boundary}^{(2)} \qquad (3.8)$$

όπου \mathbf{B} είναι ένα προσημασμένο Boolean μητρώο.

Το αρχικό καθολικό πρόβλημα, που περιγράφεται από την εξίσωση (3.1) μετατρέπεται στο πρόβλημα υποφορέων που περιγράφεται από τις παρακάτω σχέσεις:

$$\mathbf{K}^g \mathbf{u}^g = \mathbf{f}^g - \sum_{s=1}^{n_s} (\mathbf{B}^s)^T \boldsymbol{\lambda} \qquad (3.9)$$

$$\text{υπό τις συνθήκες: } \sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0}$$

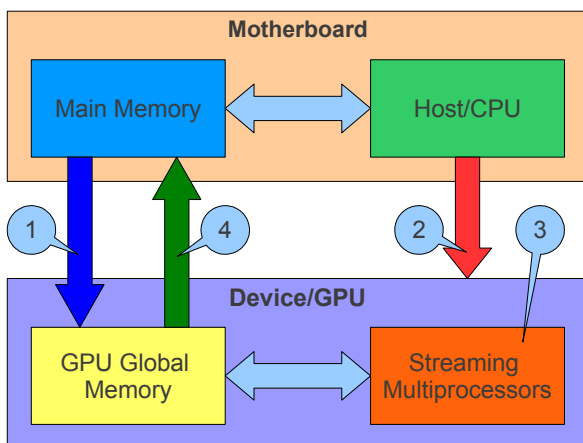
όπου $\boldsymbol{\lambda}$ είναι οι ενδοσυνοριακές δυνάμεις των υποφορέων όπως φαίνεται και στο σχήμα 3.2. Οι εξισώσεις ισορροπίας για κάθε υποφορέα s είναι:

$$\mathbf{K}^s \mathbf{u}^s = \mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda} \qquad (3.10)$$

4 Κάρτες γραφικών (Graphics Processing Units - GPUs)

Η υλοποίηση επιστημονικών εφαρμογών σε κάρτες γραφικών είναι ιδιαίτερα ενδιαφέρουσα λόγω του χαμηλού κόστους και των εξαιρετικών εγγενών χαρακτηριστικών τους. Λόγω των απαιτήσεων της βιομηχανίας παιχνιδιών, οι κάρτες γραφικών έχουν εξελιχθεί σημαντικά τα τελευταία χρόνια και παρουσιάζουν αξιοσημείωτη απόδοση. Αρχικά, οι αριθμητικές πράξεις έπρεπε να γίνουν έμμεσα, μέσω διαδικασιών που ήταν προορισμένες για γραφικά και με χρήση βιβλιοθηκών για γραφικά όπως OpenGL και DirectX. Οι υλοποιήσεις σε GPU διευκολύνθηκαν σημαντικά με την πρώτη έκδοση του CUDA-SDK, που οδήγησε σε γρήγορη ανάπτυξη του προγραμματισμού σε GPU καθώς και την εμφάνιση υπερυπελογιστών που τις αξιοποιούν, όπως φαίνεται στα Top 500 supercomputers. Σε αντίθεση με τις CPUs που έχουν στόχο να εκτελέσουν μια διεργασία πολύ γρήγορα, οι GPUs έχουν εγγενώς παράλληλη αρχιτεκτονική που εστιάζει στην εκτέλεση πολλών παράλληλων διεργασιών ταυτόχρονα.

Οι κάρτες γραφικών είναι παράλληλες συσκευές κατηγορίας SIMD (single instruction, multiple data), δηλαδή συσκευές με πολλά επεξεργαστικά στοιχεία που εκτελούν την ίδια διαδικασία σε διαφορετικά δεδομένα παράλληλα, εκμεταλλευόμενες έτσι την παραλληλία σε επίπεδο δεδομένων. Ο προγραμματισμός σε “γλώσσες” CUDA ή openCL περιλαμβάνει απλώς κάποιες επεκτάσεις στη C και άρα δεν απαιτεί εξειδικευμένη γνώση πάνω σε διεργασίες που αφορούν γραφικά. Στα πλαίσια των openCL και CUDA, η CPU αναφέρεται ως “host” ενώ η GPU αναφέρεται ως “device”. Η γενική ροή του προγραμματισμού σε GPU απεικονίζεται στο σχήμα 4.1.



1. Μεταφορά δεδομένων στη μνήμη της GPU
2. CPU: εντολές προς GPU
3. GPU: παράλληλη επεξεργασία
4. Μεταφορά αποτελεσμάτων στην κεντρική μνήμη

Fig. 4.1: Ροή επεξεργασίας με GPU

4.1 CPU vs GPU

Οι κάρτες γραφικών έχουν τη δυνατότητα να κάνουν πράξεις κινητής υποδιαστολής με διπλή ακρίβεια (floating-point operations, double precision) της τάξης των 1.5-3.0 TFLOPS, σε αντίθεση με τις CPUs που κυμαίνονται στα 150 GFLOPS. Παρά τη μεγάλη διαφορά σε αυτά τα νούμερα, οι GPUs δεν αντικαθιστούν τις CPUs. Κάθε τύπος επεξεργαστή είναι κατάλληλος για διαφορετικού τύπου διεργασίες οπότε είναι σημαντική η χρήση του κατάλληλου τύπου επεξεργαστή ανάλογα με τη διεργασία για την επίτευξη υψηλών επιδόσεων.

Οι δύο τύποι (CPUs, GPUs) έχουν σχεδιαστεί με τελείως διαφορετική λογική. Ο σχεδιασμός της CPU είναι “latency-oriented” και αποσκοπεί στο να βελτιώσει την εκτέλεση σειριακού κώδικα, δηλαδή να μειώσει το χρόνο εκτέλεσης μιας και μόνο διεργασίας. Από την άλλη, ο σχεδιασμός της GPU είναι “throughput-oriented” και επιτρέπει στις διεργασίες να γίνονται δυνητικά πιο αργά με αντάλλαγμα την εκτέλεση πάρα πολλών διεργασιών ταυτόχρονα. Οι δύο επεξεργαστές αξιοποιούνται, λοιπόν, για διαφορετικές διεργασίες και πρέπει να αντιμετωπίζονται ως συμπληρωματικοί συνεπεξεργαστές.

Οι GPUs αναπτύσσονται ταχύτατα και έτσι οι δυνατότητες τους αλλάζουν. Σε αυτή τη διατριβή, οι GPUs που χρησιμοποιούνται είναι οι: NVIDIA GeForce GTX 580 και NVIDIA GeForce GTX 680.

4.2 GPU Threads

Η GPU εφαρμόζουν τις ίδιες διεργασίες σε μεγάλο πλήθος δεδομένων. Οι διεργασίες αυτές ονομάζονται “kernels” και δημιουργούν ένα μεγάλο αριθμό από threads. Το thread είναι η μικρότερη μονάδα για επεξεργασία που μπορεί να προγραμματιστεί από το λειτουργικό σύστημα. Τα threads στις GPU θέλουν ελάχιστους κύκλους ρολογιού (clock cycles) για να δημιουργηθούν και να αξιοποιηθούν, σε αντίθεση με τις CPU που τα threads είναι γενικώς “ακριβά”. Μία CPU έχει χαμηλό αριθμό threads (π.χ. 4-12), ενώ μία GPU έχει χιλιάδες threads ή και περισσότερα. Σημειώνεται ότι αυτό δε σημαίνει ότι όλα τα threads εκτελούνται ταυτόχρονα, αλλά μεγάλος αριθμός threads επιτρέπει στη GPU να προγραμματίσει την εκτέλεση με το βέλτιστο τρόπο.

4.3 Οργάνωση των threads

Όλα τα threads που ορίζονται από ένα kernel αποτελούν το λεγόμενο “grid”. Τα threads ενός grid είναι οργανωμένα σε ομάδες που αναφέρονται γενικά ως “thread blocks” [CUDA] ή “thread group”

[openCL]. Το grid αποτελείται, λοιπόν, από μια ομάδα blocks (όλα ίδιου μεγέθους), και κάθε block αποτελείται από μια ομάδα από threads (σχήμα 4.3). Όσα threads βρίσκονται στο ίδιο block μπορούν να συνεργάζονται μεταξύ τους. Κάθε thread block είναι εντελώς ανεξάρτητο από τα υπόλοιπα, γεγονός που επιτρέπει στη GPU να τα εκτελέσει με οποιαδήποτε σειρά.

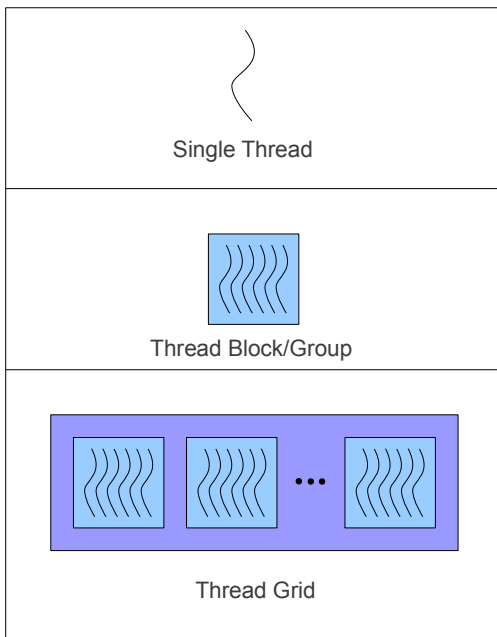


Fig. 4.3. Οργάνωση των threads

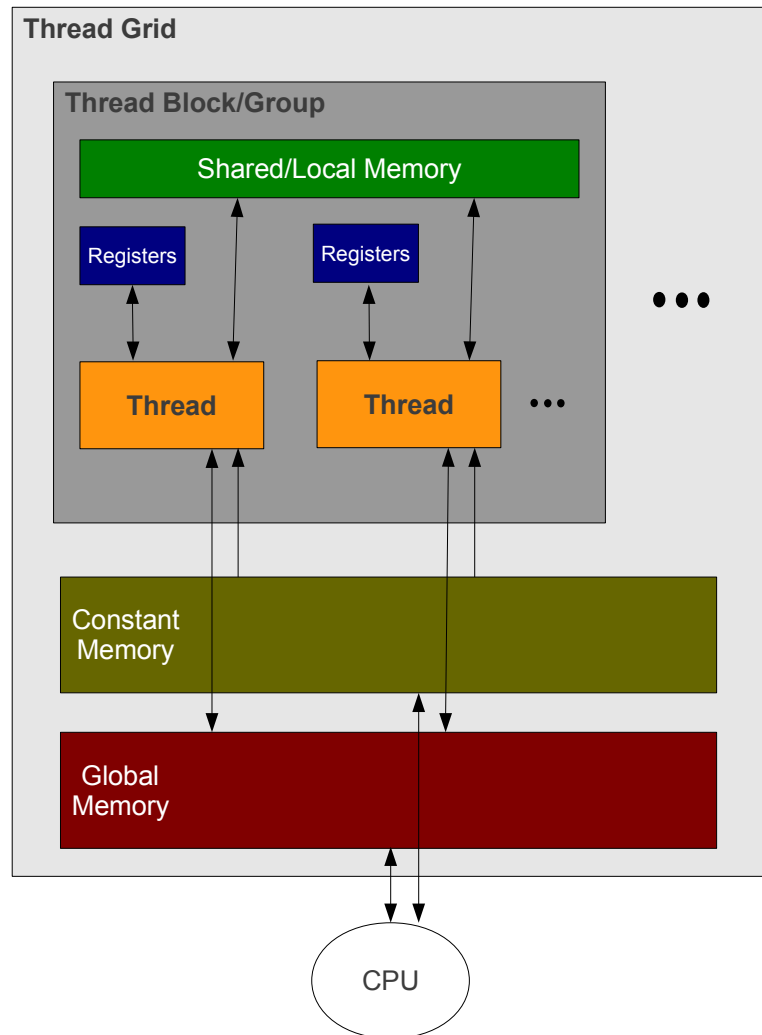


Fig. 4.2. Σχηματική απεικόνιση των μνημών της GPU

4.4 Μνήμες της GPU

Οι κάρτες γραφικών έχουν πολλών τύπων μνήμες οι οποίες πρέπει να αξιοποιηθούν από τους προγραμματιστές για την επίτευξη υψηλής απόδοσης. Το σχήμα 4.2 δείχνει μια απλοποιημένη αναπαράσταση των διαφόρων τύπων. Οι μνήμες διαφέρουν ως προς την ταχύτητα, το μέγεθος καθώς και με βάση το ποια threads έχουν πρόσβαση σε αυτές.

5 Χειρισμός μητρώων

Η αποθήκευση των μητρώων και οι μητρικές πράξεις επηρεάζουν σημαντικά την απόδοση μεγάλης κλίμακας προσομοιώσεων. Υπάρχουν πολλοί διαφορετικοί τρόποι αποθήκευσης, καθένας από τους οποίους έχει τα πλεονεκτήματα και μειονεκτήματά του. Για παράδειγμα, η πλήρης αποθήκευση αποτελεί τον πιο ευέλικτο και απλό τύπο αποθήκευσης, υποστηρίζοντας οποιαδήποτε πράξη/διαδικασία, αλλά απαιτεί την περισσότερη μνήμη για την αποθήκευσή του. Τα αραιά (sparse) μητρώα αποθηκεύουν τον ελάχιστον δυνατό αριθμό στοιχείων, αλλά έχουν υψηλό κόστος προσπέλασης και μπορούν να χρησιμοποιηθούν αποτελεσματικά μόνο για συγκεκριμένες πράξεις (διαφορετικές ανάλογα με τη sparse μορφή). Επομένως, η επιλογή της κατάλληλης μορφής αποθήκευσης για τις διαδικασίες στις οποίες συμμετέχει το μητρώο έχουν σημαντική επίδραση στην απόδοση.

5.1 Πλήρης αποθήκευση

5.1.1 Πλήρες μητρώο

Όταν ένα μητρώο με m γραμμές και n στήλες αποθηκεύεται σε πλήρη μορφή, αποθηκεύονται όλοι οι $m \times n$ όροι του μητρώου. Αν το μητρώο είναι τετραγωνικό με διάσταση n , αποθηκεύονται όλοι οι $n \times n$ όροι. Αυτός είναι ο πιο γενικός τύπος μητρώου, έχει γρήγορη προσπέλαση και υποστηρίζει όλες τις πράξεις. Όμως, για την αποθήκευσή του απαιτείται χώρος ανάλογος με $O(mn)$ ή $O(n^2)$. Επομένως, καλό είναι να χρησιμοποιείται μόνο για μικρά μητρώα και όταν το μητρώο είναι γεμάτο (ή σχεδόν γεμάτο) με μη-μηδενικούς όρους, όπως συμβαίνει για τα τοπικά μητρώα δυσκαμψίας των πεπερασμένων στοιχείων. Το πλήρες μητρώο μπορεί να αποθηκευτεί είτε κατά γραμμή (row-major) είτε κατά στήλη (column-major).

5.1.2 Συμμετρικό πλήρες μητρώο

Πολλά από τα μητρώα της ανάλυσης είναι συμμετρικά. Μπορούμε να εκμεταλλευτούμε αυτή την ιδιότητα για να ελαττώσουμε τη μνήμη αποθήκευσης κατά $\sim 50\%$. Τα συμμετρικά μητρώα είναι τετραγωνικά μητρώα για τα οποία $A_{ij} = A_{ji}$ για όλα τα i, j . Καθώς το κάτω τρίγωνο του μητρώου είναι ίσο με το άνω τρίγωνο, αρκεί να αποθηκευτεί ένα από αυτά. Το συμμετρικό πλήρες μητρώο υποδηλώνει ότι όλοι οι όροι του άνω ή κάτω τριγώνου αποθηκεύονται, σε αντίθεση με τις μορφές αποθήκευσης που αναφέρονται παρακάτω.

5.1.3 Διαγώνιο μητρώο

Το διαγώνιο μητρώο μπορεί να έχει μη-μηδενικά στοιχεία μόνο στη διαγώνιο – όλα τα άλλα στοιχεία θεωρούνται μηδέν. Το διαγώνιο μητρώο αποθηκεύει όλους τους A_{ii} όρους και απαιτεί μνήμη ανάλογη του $O(n)$. Παράδειγμα χρήσης αυτού του μητρώου είναι το μητρώο μάζας.

5.2 Αποθήκευση που λαμβάνει υπόψιν το εύρος ζώνης

Οι παραπάνω τύποι αποθήκευσης είναι “πλήρεις”, γιατί αποθηκεύουν ολόκληρο το μητρώο ή κάποια περιοχή του. Μητρώα με μικρό εύρος ζώνης μπορούν να αποθηκευτούν με πολύ πιο οικονομικό τρόπο. Το εύρος ζώνης είναι ο μικρότερος αριθμός συνεχόμενων διαγωνίων μέσα στις οποίες βρίσκονται όλα τα μη-μηδενικά στοιχεία. Το εύρος ημι-ζώνης περιλαμβάνει διαγωνίους μέχρι την κεντρική διαγώνιο.

Τα μητρώα (όπως πχ. το μητρώο δυσκαμψίας) που προκύπτουν από τις μεθόδους FEM/MMs/IGA ανήκουν σε αυτή την κατηγορία καθώς έχουν τους μη-μηδενικούς όρους τους σχετικά κοντά στη διαγώνιο. Η αρίθμηση επηρεάζει την απόσταση από τη διαγώνιο και υπάρχει πλήθος τεχνικών που αλλάζει την αρίθμηση για να μειώσει το εύρος ζώνης. Μείωση του εύρους ζώνης συνεπάγεται χαμηλότερο κόστος αποθήκευσης και πιο γρήγορες πράξεις.

5.2.1 Ταινιωτή αποθήκευση

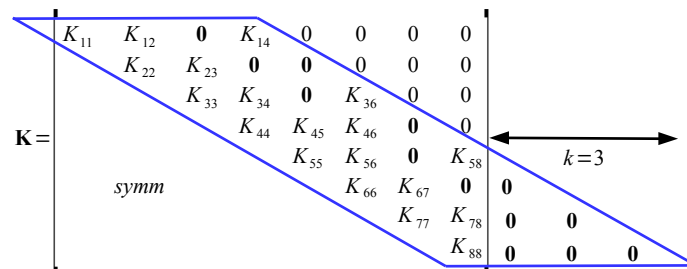


Fig. 5.1: Όροι που περιλαμβάνονται στην ταινιωτή αποθήκευση

Στο σχήμα 5.1, το εύρος ημιζώνης (χωρίς τη διαγώνιο) είναι 3. Επομένως, $3+1=4$ διαγώνιοι πρέπει να αποθηκευτούν. Σημειώνεται ότι περιλαμβάνονται μηδενικά στοιχεία στην ταινιωτή αποθήκευση, καθώς και συμπληρωματικά μηδενικά στοιχεία που επεκτείνουν τις διαγωνίους στο μέγεθος της κεντρικής διαγωνίου. Οι όροι εντός της μπλε γραμμής που αποθηκεύονται αποτελούν ένα πλήρες μητρώο διάστασης $order \times (k+1)$.

5.2.2 Αποθήκευση Skyline (οριογραμμής)

Η μορφή skyline αντιμετωπίζει το μητρώο κατά στήλη, ξεκινώντας από το διαγώνιο στοιχείο και ανεβαίνοντας μέχρι το τελευταίο μη-μηδενικό στοιχείο της στήλης (οτιδήποτε εκτός της οριογραμμής είναι μηδέν). Στο σχήμα 5.2, η μπλε γραμμή (οριογραμμή-skyline), περικλείει όλους τους όρους του μητρώου που θα αποθηκευτούν. Η μορφή skyline αποθηκεύει κάποια μηδενικά, σημαντικά λιγότερα όμως από την ταινιωτή μορφή, όπως για παράδειγμα τα μηδενικά σημειωμένα με στο σχήμα 5.2. Οι όροι αποθηκεύονται κατά στήλη σε ένα διάνυσμα, ξεκινώντας από το διαγώνιο στοιχείο της κάθε στήλης και ανεβαίνοντας, όπως φαίνεται στο σχήμα 5.3.

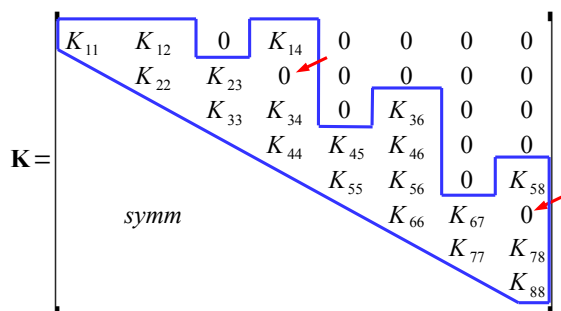


Fig. 5.2: Όροι που περιλαμβάνονται στην αποθήκευση skyline

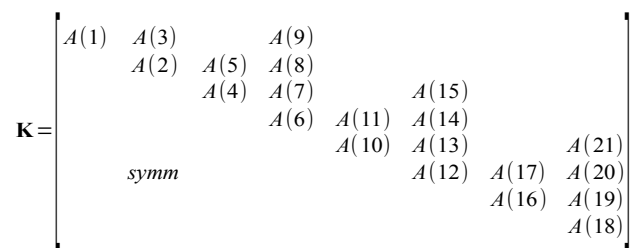


Fig. 5.3: Σειρά αποθήκευσης των όρων με την αποθήκευση skyline

Η μορφή skyline έχει επιπλέον και ένα βοηθητικό διάνυσμα. Αυτό το διάνυσμα περιλαμβάνει τους δείκτες των διαγώνιων στοιχείων του μητρώου.

$$\mathbf{diagIndexes} = [1 \ 2 \ 4 \ 6 \ 10 \ 12 \ 16 \ 18 \ 22] \quad (5.1)$$

5.2.3 Παραγοντοποίηση

Ένα από τα σημαντικότερα πλεονεκτήματα των μεθόδων αποθήκευσης που εκμεταλλεύονται το εύρος ζώνης σε σχέση με την πλήρη και τη sparse αποθήκευση είναι η παραγοντοποίηση. Σε ένα μητρώο με μικρό εύρος ζώνης, οι απαιτούμενες πράξεις είναι πολύ λιγότερες σε σχέση με τις πλήρεις μορφές, ενώ διατηρείται η δυνατότητα επί τόπου (in-place) παραγοντοποίησης, σε αντίθεση με τις sparse μορφές (σημείωση: αναφερόμαστε σε πλήρη παραγοντοποίηση).

5.3 Αραιή (Sparse) αποθήκευση

Για να έχει νόημα η αραιή αποθήκευση ενός μητρώου θα πρέπει τα μη μηδενικά στοιχεία είναι επαρκώς λίγα ώστε να έχει νόημα η ειδική αποθήκευσή τους για τη μείωση τόσο του χώρου όσο και των υπολογισμών που απαιτούνται στις πράξεις που συμμετέχει το μητρώο. Παρότι είναι επιθυμητό να αποθηκεύονται και να χρησιμοποιούνται μόνο οι μη-μηδενικοί όροι, στη γενική περίπτωση αυτό δεν εξασφαλίζει ούτε ότι η απαιτούμενη μνήμη θα είναι λιγότερη ούτε ότι ο υπολογιστικός φόρτος θα είναι μικρότερος. Αυτό συμβαίνει γιατί η αραιή αποθήκευση χρειάζεται περισσότερο χώρο ανά μη-μηδενικό στοιχείο από ότι σε πιο πυκνούς τύπους αποθήκευσης. Επιπλέον, οι πράξεις με αραιή αποθήκευση δεν είναι τόσο γρήγορη όσο με πιο πυκνούς τύπους, αλλά βέβαια οι πράξεις είναι (ή τουλάχιστον θα έπρεπε να είναι) σημαντικά λιγότερες. Για τους παραπάνω λόγους, μια συνθήκη για να είναι ένα μητρώο να είναι πρακτικά αραιό ώστε να δικαιολογεί τη χρήση των αραιών μεθόδων αποθήκευσης είναι να περιέχει $O(n)$ μη-μηδενικούς όρους.

Κάθε τύπος αραιής αποθήκευσης είναι κατάλληλος για πολύ συγκεκριμένη χρήση. Στα πλαίσια αυτής της διατριβής, υπάρχουν 2 βασικές κατηγορίες μορφών αραιής αποθήκευσης: μορφές που είναι κατάλληλες για πράξεις (π.χ. μητρώο επί διάνυσμα - sparse-matrix vector multiplication – SpMV) και μορφές που είναι κατάλληλες για σταδιακή κατασκευή του μητρώου.

5.3.1 Μορφές αραιής αποθήκευσης για τη φάση κατασκευής

Αυτές οι μορφές είναι κατάλληλες για τη σταδιακή κατασκευή ενός αραιού μητρώου. Το μητρώο φτιάχνεται με κάποια από αυτές και στη συνέχεια μετατρέπεται στις μορφές που είναι καλύτερες για μητρικές πράξεις. Υπάρχουν δύο υποκατηγορίες: μορφές που υποστηρίζουν γρήγορη ανάκτηση στοιχείων, οπότε επιτρέπουν την εύρεση και ανανέωση της τρέχουσας τιμής κάποιου όρου του μητρώου, καθώς και μορφές που δεν υποστηρίζουν γρήγορη ανάκτηση. Οι πρώτες είναι γενικά πιο πολύπλοκες στην υλοποίησή τους αλλά απαιτούνται στην περίπτωση που οι τιμές του μητρώου ανανεώνονται σταδιακά κατά τη φάση κατασκευής. Από την άλλη, αν υπολογίζεται η τελική τιμή του κάθε όρου του μητρώου πριν εισαχθεί στο μητρώο, τότε δε χρειάζονται ανακτήσεις και μπορούν να χρησιμοποιηθούν πιο απλές μορφές. Δύο τύποι εξετάζονται: Coordinate List (COO) και Dictionary of Keys (DOK).

5.3.1.1 Coordinate list (COO)

Η μορφή COO αποθηκεύει μία λίστα με τριπλέτες (γραμμή, στήλη, τιμή) για κάθε μη μηδενική τιμή. Ο πιο απλός τρόπος υλοποίησης είναι με χρήση τριών διανυσμάτων, ένα για τις γραμμές, ένα για τις στήλες και ένα για τις τιμές, όπου ο όρος i των διανυσμάτων αντιστοιχεί σε ένα όρο του μητρώου. Το μητρώο της σχ. (5.2) και διάστασης 4×5 δίνεται σε μορφή COO στη σχ. (5.3).

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ 0 & K_{22} & 0 & 0 & 0 \\ K_{31} & 0 & 0 & K_{34} & K_{35} \\ 0 & 0 & K_{43} & 0 & K_{45} \end{bmatrix} \quad (5.2)$$

$$\begin{aligned} rowIndexes &= [3 & 1 & 1 & 4 & 3 & 2 & 4 & 4] \\ columnIndexes &= [4 & 1 & 2 & 5 & 1 & 2 & 3 & 5] \\ values &= [K_{34} & K_{11} & K_{12} & K_{45} & K_{31} & K_{22} & K_{43} & K_{45}] \end{aligned} \quad (5.3)$$

Αυτή η μορφή αποθήκευσης είναι ιδανική όταν υπολογίζεται η τελική τιμή του κάθε όρου, αλλά δεν είναι κατάλληλη για συνεχή ανανέωση τιμών. Ένα ακόμα πλεονέκτημα αυτής της μορφής είναι ότι μετατρέπεται πολύ εύκολα στις μορφές CSR/CSC που είναι κατάλληλες για τη φάση επίλυσης.

5.3.1.2 Dictionary of Keys (DOK)

Σε αντίθεση με τη μορφή COO, η μορφή DOK επιτρέπει ανάκτηση και άρα προορίζεται για περιπτώσεις όπου οι τελικές τιμές του μητρώου δημιουργούνται σταδιακά εντός του αραιού μητρώου. Αυτό περιλαμβάνει εύρεση της τρέχουσας τιμής K_{ij} και κατάλληλης ανανέωσης της. Υπάρχουν πολλοί τρόποι για την υλοποίηση της μορφής DOK. Ένας τρόπος είναι να αντιστοιχίσουμε (γραμμή, στήλη) με τις τιμές του μητρώου με χρήση “hash-tables”, “binary search trees (BST)” ή αντίστοιχες δομές δεδομένων. Το πρώτο επιτρέπει ανάκτηση σε χρόνο $O(1)$, ενώ το δεύτερο σε χρόνο $O(\log NZ)$, όπου NZ είναι ο αριθμός των αποθηκευμένων στοιχείων, αλλά αποθηκεύει τις τιμές ταξινομημένες. Απεικόνιση της μορφής DOK δίνεται στη σχ. (5.4).

$$\begin{aligned} (1,1) \rightarrow K_{11} & \quad (1,2) \rightarrow K_{12} & (2,2) \rightarrow K_{22} & \quad (3,1) \rightarrow K_{31} \\ (3,4) \rightarrow K_{34} & \quad (3,5) \rightarrow K_{35} & (4,3) \rightarrow K_{43} & \quad (4,5) \rightarrow K_{45} \end{aligned} \quad (5.4)$$

5.3.2 Μορφές αραιής αποθήκευσης για αριθμητικές πράξεις

Το μητρώο προετοιμάζεται με μία από τις μορφές που αναφέρθηκαν παραπάνω, και στη συνέχεια μετατρέπεται στις μορφές Compressed Sparse Row (CSR) ή Compressed Sparse Column (CSC). Αυτοί οι δύο μορφές είναι κατάλληλες για αριθμητικές πράξεις, αλλά και πάλι πρέπει να επιλεγεί η βέλτιστη μορφή ανάλογα με τις πράξεις στις οποίες συμμετέχει το μητρώο κατά τη φάση επίλυσης.

5.3.2.1 Compressed Sparse Row (CSR)

Ξεκινώντας από τη μορφή COO, οι δείκτες της μορφής CSR είναι ταξινομημένοι κατά γραμμή και, εντός της ίδιας γραμμής, κατά στήλη. Σε ταξινομημένη μορφή πολλοί συνεχόμενοι δείκτες γραμμής θα είναι ίδιοι, οπότε στη συμπυκνωμένη (compressed) μορφή αποθηκεύουμε μόνο τη θέση στην οποία κάποιος συγκεκριμένος δείκτης εμφανίζεται για πρώτη φορά:

$$\begin{aligned} \text{rowIndexes} &= [1 \quad 3 \quad 4 \quad 7 \quad 9] \\ \text{columnIndexes} &= [1 \quad 2 \quad 2 \quad 1 \quad 4 \quad 5 \quad 3 \quad 5] \\ \text{values} &= [K_{11} \quad K_{12} \quad K_{22} \quad K_{31} \quad K_{34} \quad K_{35} \quad K_{43} \quad K_{45}] \end{aligned} \quad (5.5)$$

Η μορφή CSR είναι βολική για πράξεις που χρησιμοποιούν τα στοιχεία κατά γραμμή (όπως πολλαπλασιασμός μητρώου-διανύσματος), αλλά αργή για πράξεις που χρησιμοποιούν στοιχεία κατά στήλη.

5.3.2.2 Compressed Sparse Column (CSC)

Ξεκινώντας από τη μορφή COO, οι δείκτες της μορφής CSC είναι ταξινομημένοι κατά στήλη και, εντός της ίδιας στήλης, κατά γραμμή. Σε ταξινομημένη μορφή πολλοί συνεχόμενοι δείκτες στήλης θα είναι ίδιοι, οπότε στη συμπυκνωμένη (compressed) μορφή αποθηκεύουμε μόνο τη θέση στην οποία κάποιος συγκεκριμένος δείκτης εμφανίζεται για πρώτη φορά:

$$\begin{aligned} \text{rowIndexes} &= [1 \quad 3 \quad 1 \quad 2 \quad 4 \quad 3 \quad 3 \quad 4] \\ \text{columnIndexes} &= [1 \quad 3 \quad 5 \quad 6 \quad 7 \quad 9] \\ \text{values} &= [K_{11} \quad K_{12} \quad K_{22} \quad K_{31} \quad K_{34} \quad K_{35} \quad K_{43} \quad K_{45}] \end{aligned} \quad (5.6)$$

Η μορφή CSC είναι βολική για πράξεις που χρησιμοποιούν τα στοιχεία κατά στήλη (όπως πολλαπλασιασμός ανάστροφου μητρώου-διανύσματος), αλλά αργή για πράξεις που χρησιμοποιούν στοιχεία κατά γραμμή. Σημειώνεται ότι ο πολλαπλασιασμός μητρώου διανύσματος μπορεί να υλοποιηθεί αποδοτικά και με αυτή τη μορφή, αλλά η μορφή CSR ενδέχεται να είναι ταχύτερη.

6 Μέθοδοι υποφορέων σε υβριδική αρχιτεκτονική CPU-GPU

Η δυϊκή μέθοδος υποφορέων (FETI) υλοποιήθηκε σε υβριδικό CPU-GPU περιβάλλον με σκοπό την αξιοποίηση όλης της διαθέσιμης επεξεργαστικής ισχύς και διαθέσιμης μνήμης για την επίλυση ακόμα μεγαλύτερων προβλημάτων. Λόγω της ετερογένειας μεταξύ CPU και GPU καθώς και του διαφορετικού τρόπου προγραμματισμού τους, απαιτείται ειδικός χειρισμός σε αρκετά σημεία του αλγορίθμου της FETI για την επίτευξη της βέλτιστης απόδοσης. Ένα από τα βασικότερα θέματα που πρέπει να αντιμετωπιστούν είναι η μεγάλη διαφορά της απόδοσης μεταξύ CPU και GPU, που επηρεάζεται κυρίως από τις αριθμητικές πράξεις που εκτελούνται αλλά και άλλες παραμέτρους. Μάλιστα, η διαφορά στην απόδοση μεταξύ CPU και GPU δεν είναι η ίδια για τον υπολογισμό εσωτερικών γινομένων, για τον πολλαπλασιασμό μητρώου-διανύσματος ή για την άμεση επίλυση γραμμικών συστημάτων με παραγοντοποίηση Cholesky.

Εξετάστηκαν δύο διαφορετικές υλοποιήσεις για την επίλυση των τοπικών προβλημάτων σε επίπεδο υποφορέων. Η πρώτη χρησιμοποιεί άμεση επίλυση Cholesky ενώ η δεύτερη επαναληπτική επίλυση με PCG. Οι δύο αυτοί τρόποι, εκτός από την εντελώς διαφορετική υλοποίηση σε παράλληλο περιβάλλον, έχουν και διαφορετικές ανάγκες σε μνήμη, κάτι που επηρεάζει την κατανομή υποφορέων σε CPU και GPU.

6.1 Dynamic load-balancing

Η ετερογένεια των υποσυστημάτων αντιμετωπίστηκε με dynamic load balancing βασισμένο σε task queues. Πιο συγκεκριμένα, η CPU δημιουργεί μια ουρά (queue) από εργασίες που πρέπει να εκτελεστούν σε κάποιο βήμα του αλγορίθμου. Στην περίπτωση της άμεσης επίλυσης Cholesky, γίνεται παράλληλη παραγοντοποίηση στα μητρώα των υποφορέων, τα οποία είναι αποθηκευμένα σε μορφή skyline, και από τη CPU και από τη GPU. Δημιουργείται μια ουρά εργασιών για την εκτέλεση της παραγοντοποίησης και των εμπρός και πίσω αντικαταστάσεων. Η ουρά περιέχει τα κατάλληλα μητρώα των υποφορέων, και οι CPU/GPU τροφοδοτούνται με εργασίες με ένα ασύγχρονο τρόπο. Όταν κάποιος επεξεργαστής τελειώσει την τρέχουσα εργασία, παίρνει την επόμενη εργασία από την ουρά, όπως φαίνεται σχηματικά στο σχήμα 6.1. Κατά αυτό τον τρόπο, και η CPU και η GPU είναι συνεχώς απασχολημένες μέχρι η ουρά εργασιών να αδειάσει.

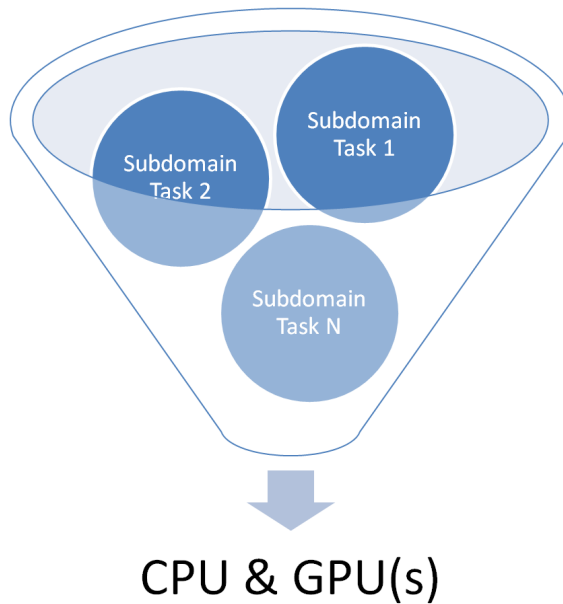


Fig. 6.1. Η ουρά εργασιών περιέχει αριθμητικούς υπολογισμούς που πρέπει να εκτελεστούν με τους διαθέσιμους πόρους.

6.2 Αριθμητικά αποτελέσματα

Έγινε παραμετρική μελέτη σε προβλήματα 3D γραμμικής ελαστικότητας ($E=39 \text{ MPa}$, $\nu=0.2$) σε κύβο. Ο φορέας είναι πλήρως δεσμευμένος στην κάτω επιφάνεια, και μερικώς δεσμευμένος κατά τις οριζόντιες διευθύνσεις στις πλαϊνές πλευρές και η πάνω επιφάνεια δέχεται ισοκατανεμημένο φορτίο. Ο φορέας είναι διαχωρισμένος με 8-κομβικά εξαεδρικά στοιχεία. Το τελικό σύστημα έχει 1,058,610 βαθμούς ελευθερίας (β.ε). Η επίλυση γίνεται με μεθόδους υποφορέων, οπότε ο φορέας χωρίζεται σε υποφορείς, ο αριθμός των οποίων κυμαίνεται από 125 μέχρι 2744.

Τα παραδείγματα εκτελέστηκαν με το παρακάτω hardware. CPU: Core i7-950 που έχει 4 πυρήνες (8 λογικούς πυρήνες) στα 3.06 GHz και 8MB cache. GPU: GeForce GTX580 με 512 πυρήνες CUDA και 1.5GB GDDR5 μνήμη. Όλες οι πράξεις κινητής υποδιαστολής είναι με διπλή ακρίβεια. Τα παραπάνω χαρακτηριστικά αρκούν για να εκτελεστούν όλοι οι υπολογισμοί στη μνήμη (δηλαδή δεν υπεισέρχεται μείωση της ταχύτητας λόγω ανεπαρκούς μνήμης και χρήσης του σκληρού δίσκου).

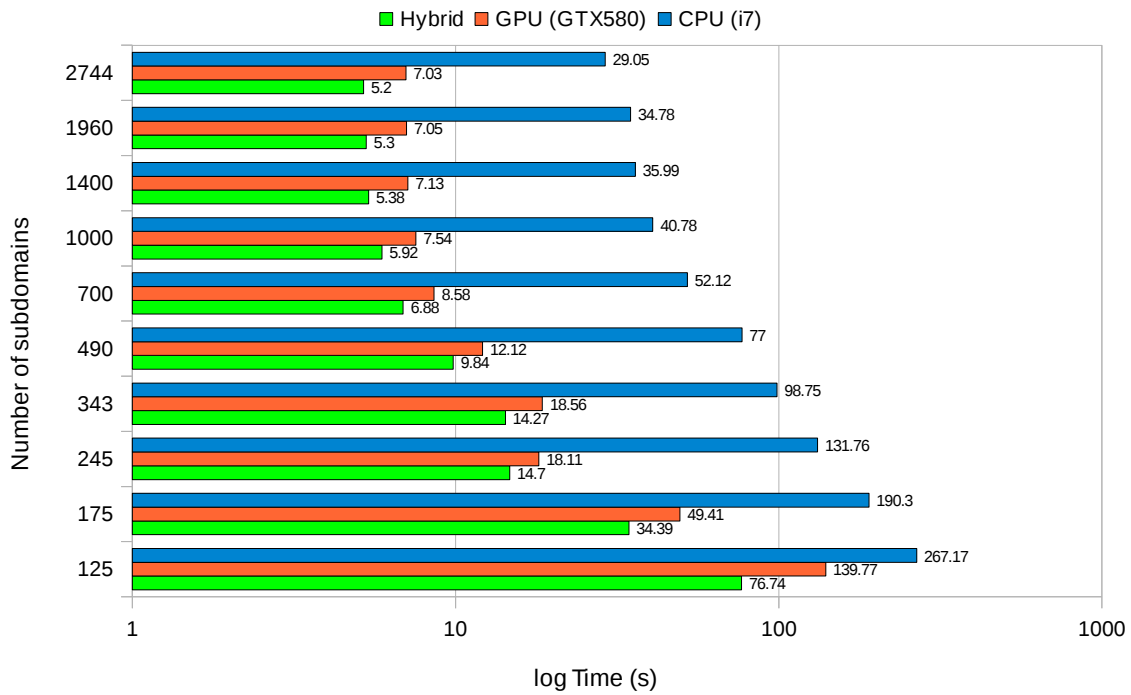


Fig. 6.2: Συνολικός χρόνος επίσης της FETI για τις περιπτώσεις: υβριδική, μόνο GPU (GTX580), μόνο CPU (i7); με άμεση επίλυση Cholesky για τα τοπικά προβλήματα υποφορέων

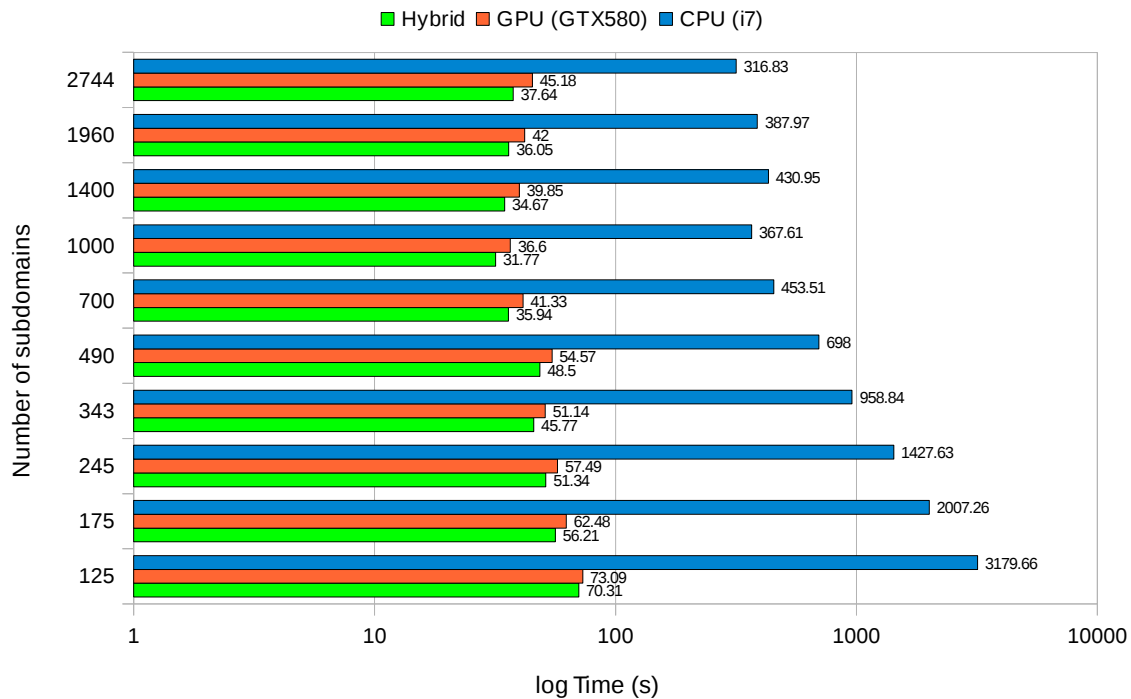


Fig. 6.3: Συνολικός χρόνος επίσης της FETI για τις περιπτώσεις: υβριδική, μόνο GPU (GTX580), μόνο CPU (i7); με επαναληπτική επίλυση PCG για τα τοπικά προβλήματα υποφορέων

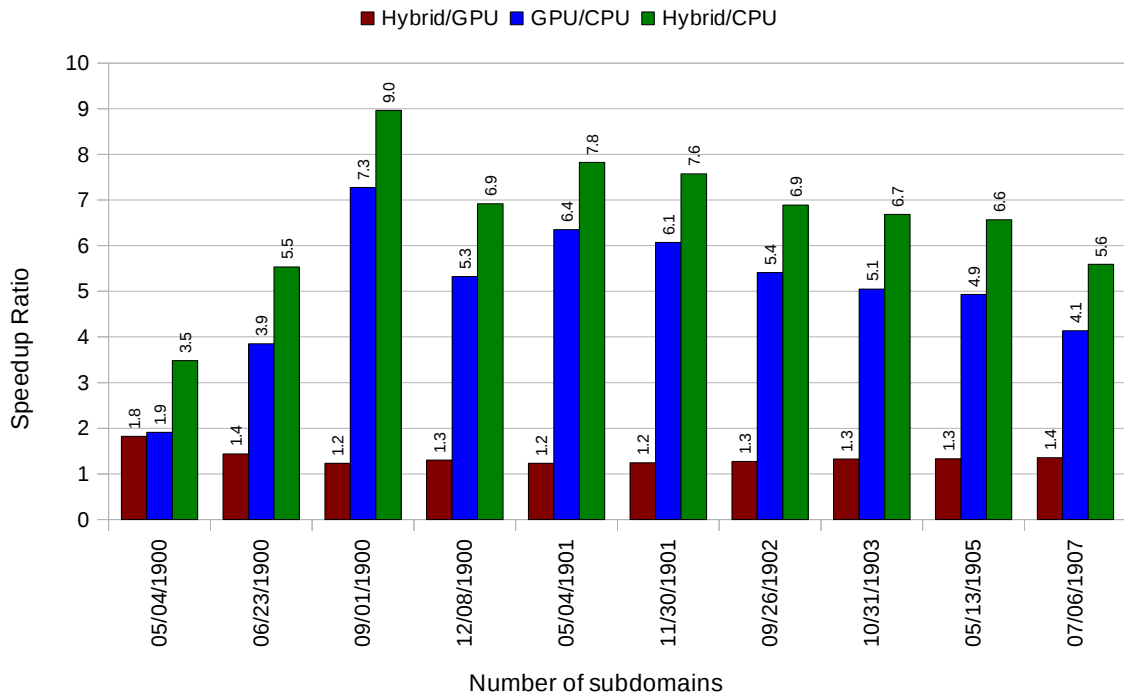


Fig. 6.4: Επιταχύνσεις για διαφορετικούς συνδυασμούς CPU (i7) και GPU (GTX 580) με άμεση επίλυση Cholesky για τα τοπικά προβλήματα υποφορέων

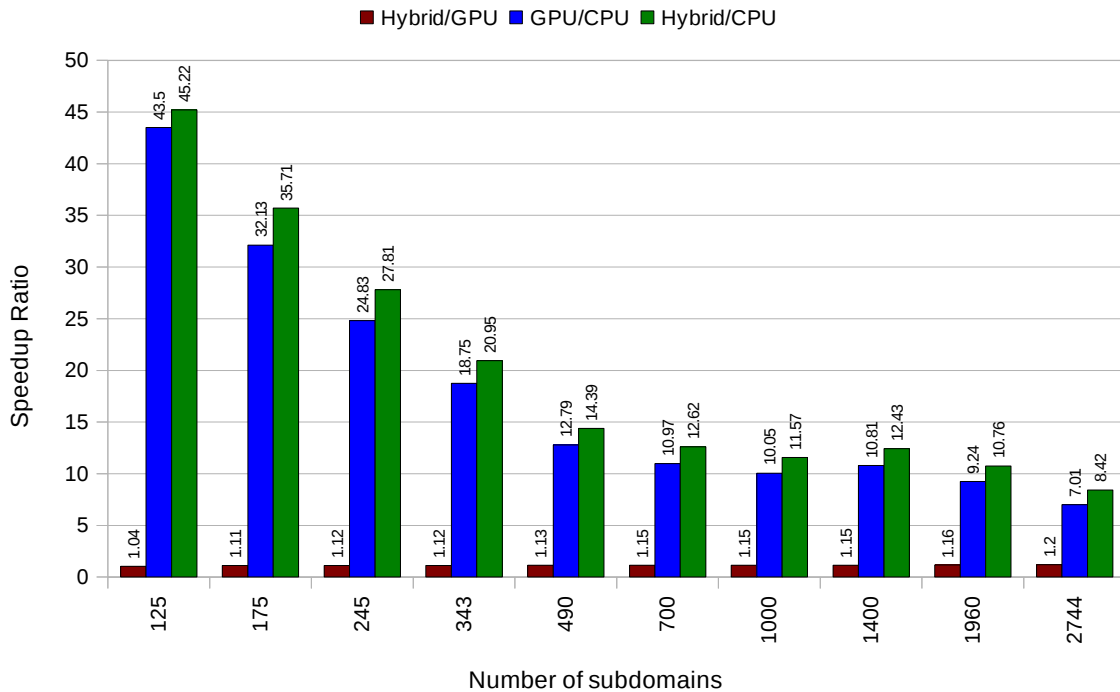


Fig. 6.5: Επιταχύνσεις για διαφορετικούς συνδυασμούς CPU (i7) και GPU (GTX 580) με επαναληπτική επίλυση PCG για τα τοπικά προβλήματα υποφορέων

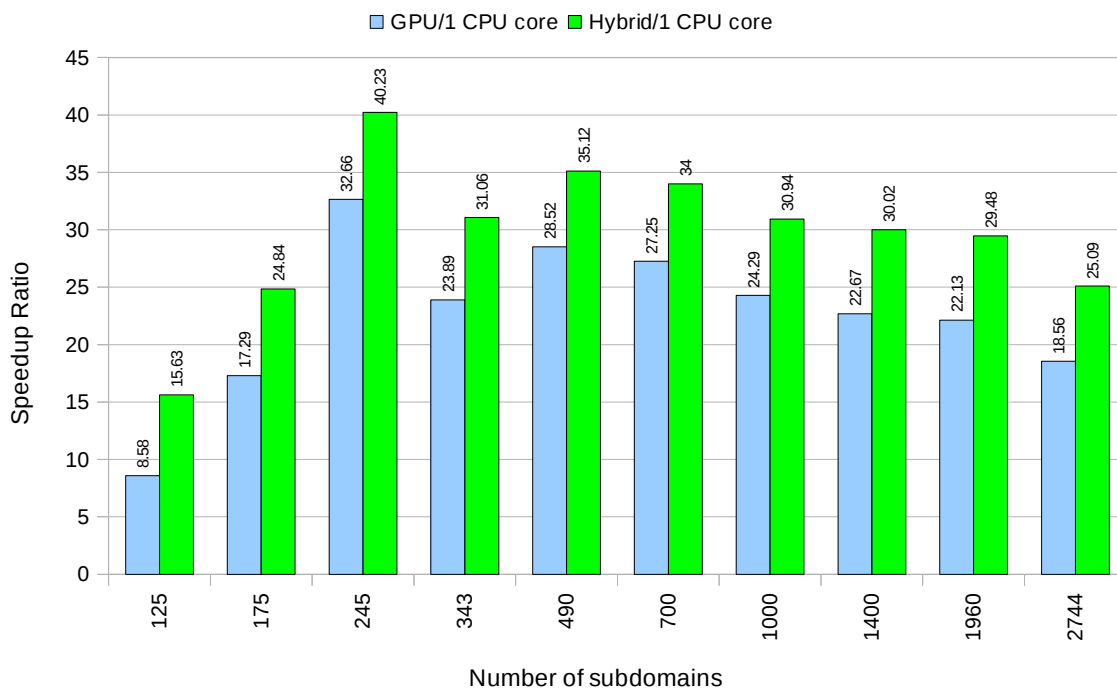


Fig. 6.6: Επιταχύνσεις ανά πυρήνα CPU για διαφορετικούς συνδυασμούς CPU (i7) και GPU (GTX 580) με άμεση επίλυση Cholesky για τα τοπικά προβλήματα υποφορέων

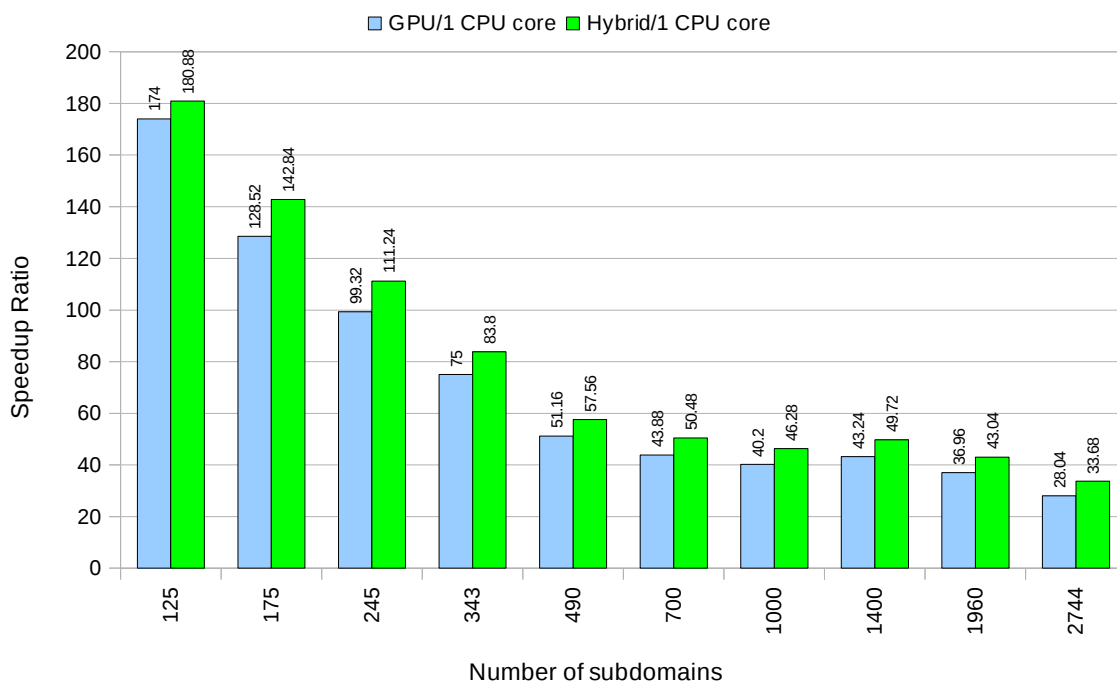


Fig. 6.7: Επιταχύνσεις ανά πυρήνα CPU για διαφορετικούς συνδυασμούς CPU (i7) και GPU (GTX 580) με επαναληπτική επίλυση PCG για τα τοπικά προβλήματα υποφορέων

7 Συσχέτιση μεταξύ των βασικών μονάδων της ολοκλήρωσης Gauss

Στη γενικότερη μορφή της ολοκλήρωσης Gauss, διακρίνουμε τις εξής δύο κατηγορίες: κομβικές μονάδες (N) και μονάδες Gauss (G). Στα πεπερασμένα στοιχεία και στις μη-πλεγματικές μεθόδους οι κομβικές μονάδες είναι οι κόμβοι ενώ στην ισογεωμετρική ανάλυση τα σημεία ελέγχου (control points). Οι μονάδες Gauss είναι είτε ξεχωριστά σημεία Gauss είτε ομάδες από σημεία Gauss που συνεισφέρουν στον υπολογισμό των συντελεστών που σχετίζονται με τις κομβικές μονάδες. Στα πεπερασμένα στοιχεία και την ισογεωμετρική ανάλυση, οι μονάδες Gauss είναι στοιχεία, τα οποία ομαδοποιούν πολλά σημεία Gauss, ενώ στις μη-πλεγματικές μεθόδους οι μονάδες Gauss είναι ανεξάρτητα σημεία Gauss. Η γενική αντιμετώπιση είναι χρήσιμη γιατί κάθε μέθοδος έχει διαφορετικές βασικές μονάδες (για παράδειγμα: FEA: κόμβοι + στοιχεία, IGA: σημεία ελέγχου + στοιχεία, MMs: κόμβοι + σημεία Gauss).

Με βάση τη γενική ορολογία, οι συσχετίσεις είναι:

- N-G αλληλοσυσχέτιση: αντιστοιχίζει κάθε κομβική μονάδα (N) με όλες τις μονάδες Gauss (G) που την επηρεάζουν
- G-N αλληλοσυσχέτιση: αντιστοιχίζει κάθε μονάδα Gauss (G) με όλες τις κομβικές μονάδες (N) που επηρεάζει
- Αλληλεπιδράσεις: αντιστοιχίζει κάθε κομβική μονάδα (N) με όλες τις άλλες κομβικές μονάδες (N) με τις οποίες αλληλεπιδρά
- Συνεργίες: αντιστοιχίζει κάθε ζεύγος (N-N) με τις μονάδες Gauss (G) που επηρεάζουν και τις δύο κομβικές μονάδες

7.1 Πεδίο επιρροής στις μεθόδους προσομοίωσης

Οι τρεις μέθοδοι που χρησιμοποιούνται σε αυτή τη διατριβή (FEA, MMs, IGA) διαφέρουν σημαντικά στο μέγεθος του πεδίου επιρροής αλλά και τον τρόπο συσχέτισης των βασικών μονάδων (κόμβοι, σημεία ελέγχου, σημεία Gauss, στοιχεία). Τα πεδία επιρροής καθορίζουν την αλληλοσυσχέτιση μεταξύ κόμβων/σημείων ελέγχου και στοιχείων/σημείων Gauss. Επιπλέον, καθορίζουν τις αλληλεπιδράσεις (ζεύγος κόμβων/σημείων ελέγχου). Ο γενικότερος ορισμός για ζεύγη αλληλεπίδρασης στην ολοκλήρωση Gauss είναι: δύο κομβικές μονάδες αλληλεπιδρούν, και άρα έχουν μη-μηδενικούς όρους στις αντίστοιχες θέσεις των χαρακτηριστικών μητρώων, αν και μόνο αν υπάρχει τουλάχιστον ένα σημείο Gauss που επηρεάζει και τις δύο κομβικές μονάδες.

7.1.1 Πεδίο επιρροής στις μη-πλεγματικές μεθόδους

Λόγω απουσίας στοιχείων, οι βασικές μονάδες στις μη-πλεγματικές μεθόδους (MMs) είναι οι κόμβοι και τα σημεία Gauss. Τα πεδία επιρροής είναι αρκετά μεγαλύτερα από τα αντίστοιχα στα πεπερασμένα στοιχεία (FEA), όπως φαίνεται στο σχήμα 7.1. Σημειώνεται ότι η σύγκριση γίνεται για ίσο αριθμό κόμβων και σημείων Gauss - οι δύο μέθοδοι δεν παρουσιάζουν την ίδια ακρίβεια σε αυτή την περίπτωση. Στα σχήματα, η ακτίνα του πεδίου επιρροής έχει ληφθεί ως 2,5 φορές η απόσταση μεταξύ δύο διαδοχικών κόμβων.

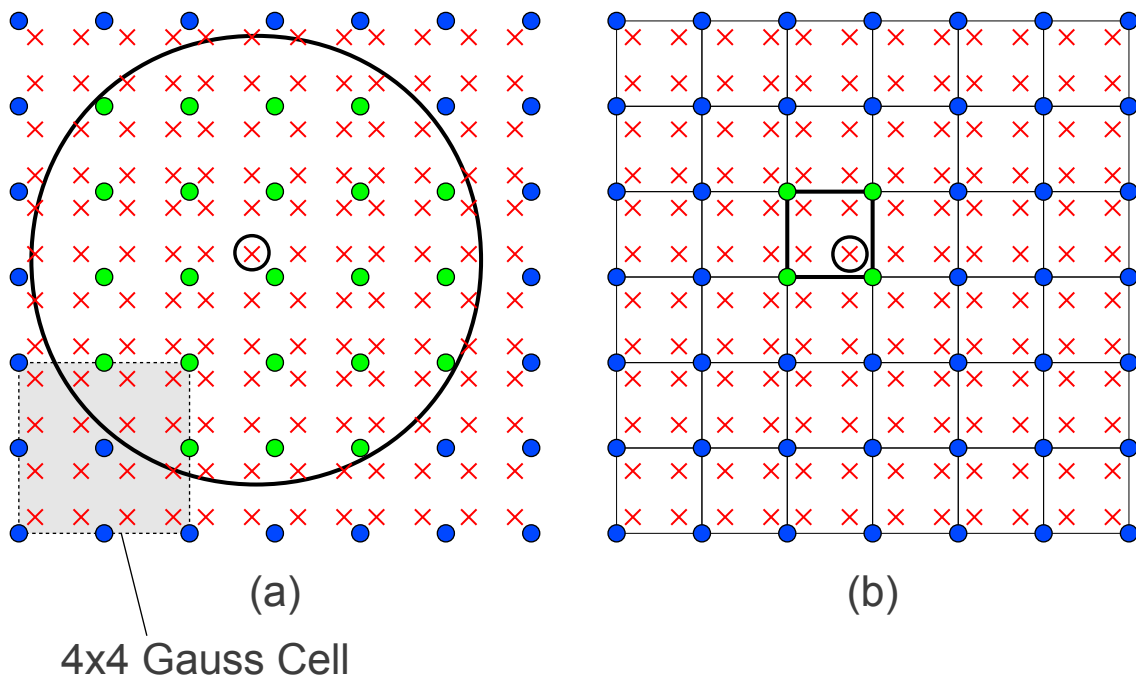


Fig. 7.1: Πεδίο επιρροής σημείου Gauss \otimes σε (a) MMs; (b) FEA, για τον ίδιο αριθμό κόμβων (●) και σημείων Gauss (×)

Στα πεπερασμένα στοιχεία, κάθε σημείο Gauss εμπλέκεται μόνο σε υπολογισμούς του στοιχείου στο οποίο ανήκει, δηλαδή για τη δημιουργία του τοπικού μητρώου δυσκαμψίας το οποίο στη συνέχεια προστίθεται στο καθολικό μητρώο δυσκαμψίας. Επιπλέον, οι συναρτήσεις σχήματος είναι προκαθορισμένες για κάθε τύπο στοιχείου και πρέπει να υπολογιστούν για κάθε συνδυασμό κόμβων και σημείων Gauss εντός του κάθε στοιχείου. Στις μη-πλεγματικές μεθόδους, οι συνεισφορές από τα σημεία Gauss προστίθενται απευθείας στο καθολικό μητρώο δυσκαμψίας ενώ οι συναρτήσεις σχήματος δεν είναι προκαθορισμένες και εκτείνονται σε μεγαλύτερο μέρος του φορέα, οπότε υπάρχουν πολύ περισσότεροι συνδυασμοί κόμβων και σημείων Gauss.

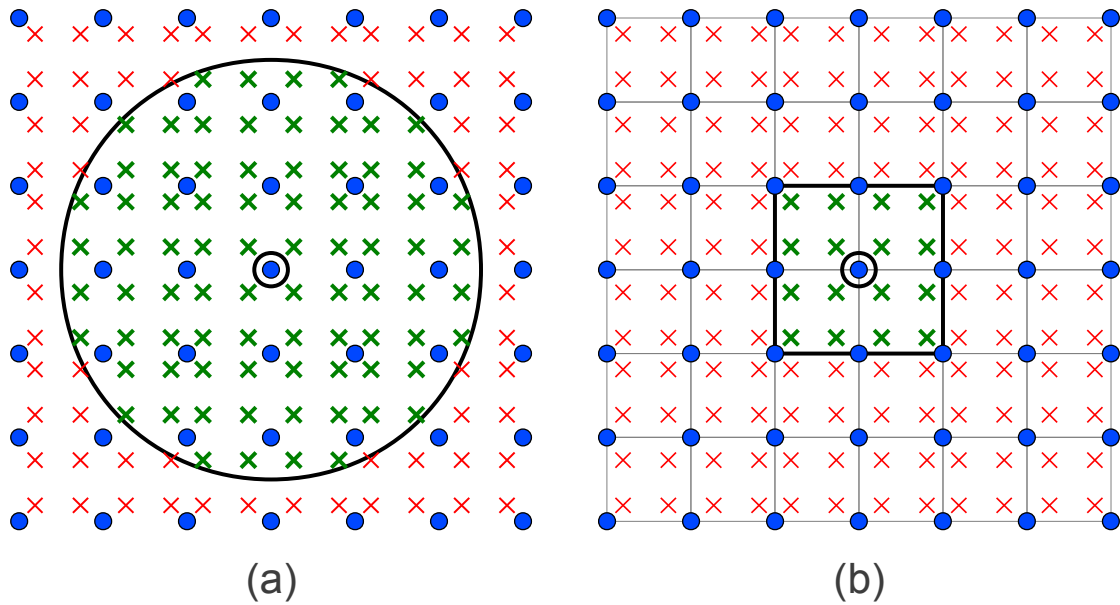


Fig. 7.2: Πεδίο επιρροής κόμβου \odot σε (a) MMs; (b) FEA, για τον ίδιο αριθμό κόμβων \bullet και σημείων Gauss \times

7.1.2 Πεδίο επιρροής στην ισογεωμετρική ανάλυση

Οι βασικές μονάδες στην ισογεωμετρική ανάλυση είναι τα σημεία ελέγχου και τα στοιχεία. Το σχήμα 7.3 συγκρίνει τις περιοχές επιρροής των σημείων ελέγχου/κόμβων (αντίστοιχα) στην ισογεωμετρική ανάλυση και στα πεπερασμένα στοιχεία, για διαφορετικά p . Σημειώνεται ότι η αλληλοσυσχέτιση είναι στην πραγματικότητα μεταξύ σημείων ελέγχου και σημείων Gauss. Τα στοιχεία αποτελούν μια βολική ομαδοποίηση ώστε οι αλληλοσυσχετίσεις να είναι ευκολότερες στο χειρισμό και την αποθήκευση.

Στα πεπερασμένα στοιχεία, κάθε σημείο Gauss εμπλέκεται μόνο σε υπολογισμούς για τους κόμβους εντός του στοιχείου στο οποίο περιέχεται. Οι συναρτήσεις σχήματος είναι προκαθορισμένες για κάθε τύπο στοιχείου και πρέπει να υπολογιστούν για κάθε συνδυασμό κόμβων και σημείων Gauss εντός του κάθε στοιχείου. Στην ισογεωμετρική ανάλυση, κάθε σημείο Gauss εμπλέκεται σε υπολογισμούς με σημεία ελέγχου και των γύρω περιοχών (7.4), ενώ οι συναρτήσεις σχήματος δεν είναι προκαθορισμένες και εκτείνονται σε μεγαλύτερο μέρος του φορέα, οπότε υπάρχουν πολύ περισσότεροι συνδυασμοί σημείων ελέγχου και σημείων Gauss. Για ισοδύναμα δίκτυα, το εύρος ζώνης είναι ίδιο στις δύο μεθόδους, αλλά η ισογεωμετρική ανάλυση έχει πολύ περισσότερες αλληλεπιδράσεις σημείων ελέγχου, άρα και πυκνότερα μητρώα δυσκαμψίας. Επιπλέον, ο υπολογισμός του κάθε μη-μηδενικού όρου είναι πιο δυσχερής καθώς τα σημεία ελέγχου επηρεάζονται από περισσότερα στοιχεία και κατ' επέκταση από σημαντικά περισσότερα σημεία Gauss.

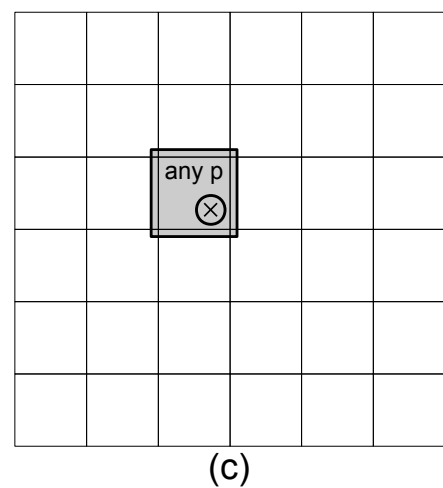
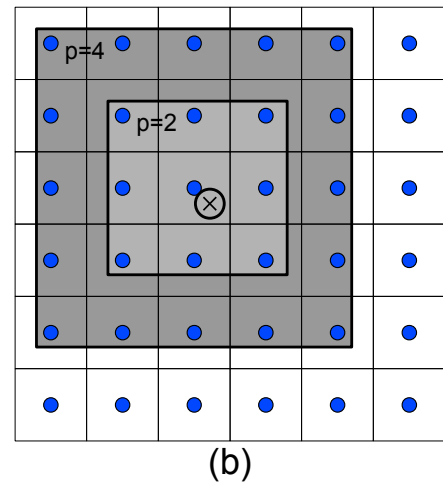
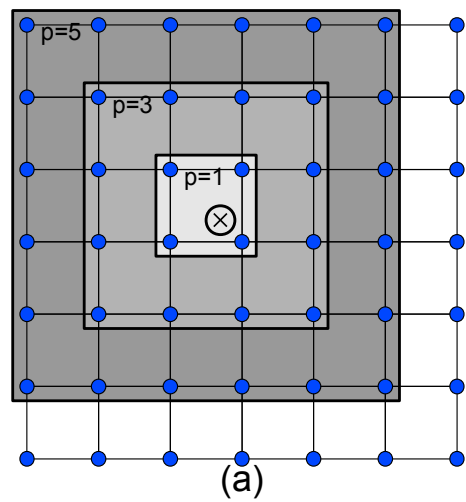
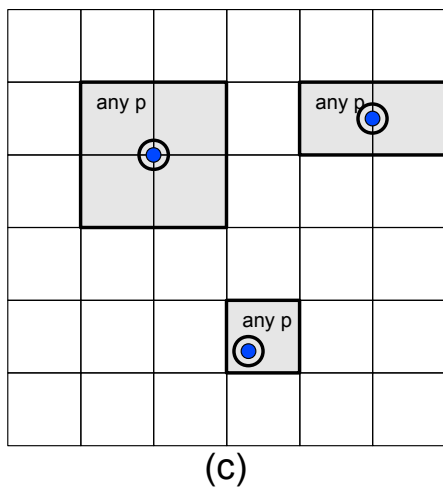
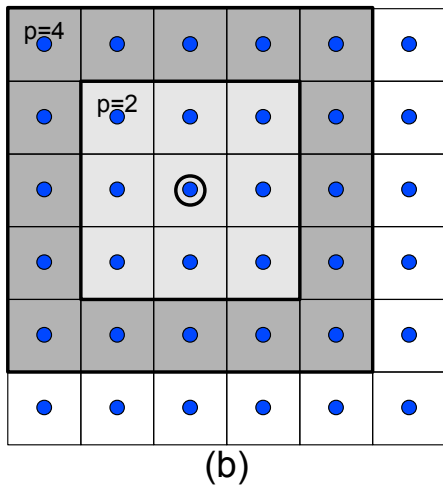
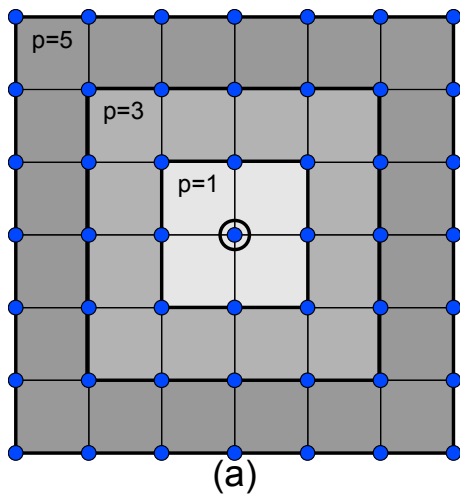


Fig. 7.3. Περιοχές επιρροής του σημείου ελέγχου \odot σε: (a) IGA (p μονός); (b) IGA (p ζυγός); (c) FEA. Οι οντότητες που επηρεάζονται είναι τα σημεία Gauss στις σημειωμένες περιοχές.

Fig. 7.4. Σημεία ελέγχου/κόμβοι που επηρεάζονται από σημείο Gauss \otimes σε: (a) IGA (p μονός); (b) IGA (p ζυγός); (c) FEA.

8 Μορφοποίηση των χαρακτηριστικών μητρώων

Οι δύο μέθοδοι που παρουσιάζονται είναι η κλασική μέθοδος με βάση τις συνεισφορές και η μέθοδος με βάση τις αλληλεπιδράσεις, η οποία είναι και κατάλληλη για παράλληλη επεξεργασία.

8.1 Μέθοδος μορφοποίησης με βάση τις συνεισφορές (contribution-wise, CW)

Η κατασκευή μέσω σταδιακής άθροισης των συνεισφορών είναι η τυπική μέθοδος για την μορφοποίηση των μητρώων που βασίζονται σε ολοκλήρωση Gauss. Το μητρώο αποτελείται από τις συνεισφορές όλων των σημείων Gauss:

$$\mathbf{K} = \sum_G w_G \mathbf{Q}_G \quad (8.1)$$

όπου w_G είναι ο συντελεστής βάρους του σημείου Gauss και \mathbf{Q}_G είναι ένα μητρώο που σχετίζεται με την εφαρμογή. Σε δομοστατικές εφαρμογές, $\mathbf{Q}_G = \mathbf{B}_G^T \mathbf{E} \mathbf{B}_G$ για FEA, MMs και IGA. Το μητρώο παραμορφώσεων \mathbf{B}_G υπολογίζεται στο εκάστοτε σημείο Gauss και \mathbf{E} είναι το καταστατικό μητρώο που περιγράφει τις ιδιότητες του υλικού. Δύο παραλλαγές της μεθόδου εξετάζονται. Η πρώτη είναι η πιο γενική και χειρίζεται τα σημεία Gauss ξεχωριστά, ενώ η δεύτερη είναι κατάλληλη για προσομοιώσεις βασισμένες σε στοιχεία και χειρίζεται ομάδες σημείων Gauss.

8.2 Μέθοδος μορφοποίησης με βάση τις αλληλεπιδράσεις (interaction-wise, IW)

Η μέθοδος που αναφέρθηκε παραπάνω υπολογίζει διαφορετικά τμήματα του αθροίσματος της σχ. (8.1) και σταδιακά τα αθροίζει για να προκύψει το τελικό αποτέλεσμα. Η IW μέθοδος υπολογίζει τις τελικές τιμές \mathbf{K}_{ij} και τις τοποθετεί στο μητρώο \mathbf{K} . Για κάθε συνδυασμό $i-j$, το \mathbf{K}_{ij} περιγράφει την αλληλεπίδραση μεταξύ δύο κόμβων (ή σημείων ελέγχου). Κάθε \mathbf{K}_{ij} διαμορφώνεται από τις συνεισφορές εκείνων των σημείων Gauss που επηρεάζουν και τους δύο κόμβους $i-j$:

$$\mathbf{K}_{ij} = \sum_{Sh.G} w_G \mathbf{Q}_{ij} = \sum_{Sh.G} w_G \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j \quad (8.2)$$

Δύο κόμβοι αλληλεπιδρούν και άρα έχουν μη-μηδενικό \mathbf{K}_{ij} αν υπάρχει τουλάχιστον ένα σημείο Gauss που επηρεάζει και τους δύο κόμβους. Αναφερόμαστε σε αυτά ως κοινά (shared) σημεία Gauss και είναι από τα βασικά συστατικά της IW μεθόδου. Και εδώ έχουμε δύο παραλλαγές, μία για χειρισμό των σημείων Gauss ξεχωριστά και μία για προσομοιώσεις με στοιχεία.

8.3 Παραλληλία στις μεθόδους μορφοποίησης

Το πιο σημαντικό πλεονέκτημα της μορφοποίησης με βάση τις αλληλεπιδράσεις (IW) είναι ότι είναι κατάλληλη για παράλληλη επεξεργασία. Στις μη-πλεγματικές μεθόδους και στην ισογεωμετρική ανάλυση, κάθε υπομητρώο \mathbf{K}_{ij} διαμορφώνεται από μεγάλο αριθμό συνεισφορών καθώς κάθε μονάδα Gauss επηρεάζει μεγάλο αριθμό κομβικών μονάδων. Η παραλληλοποίηση της μεθόδου μορφοποίησης με βάση τις συνεισφορές (CW) προϋποθέτει το λεγόμενο “scatter parallelism” που φαίνεται στο σχήμα 8.1 για δύο μονάδες Gauss C και D . Κάθε όρος του αθροίσματος μπορεί να υπολογιστεί ανεξάρτητα και παράλληλα, αλλά υπάρχει πρόβλημα όταν γίνεται η πρόσθεση γιατί πολλά threads πρέπει να γράψουν στις ίδιες θέσεις μνήμης. Το πρόβλημα μπορεί να παρακαμφθεί με κατάλληλο προγραμματισμό, όμως σε μαζικώς παράλληλα συστήματα, στα οποία χιλιάδες threads μπορεί να τρέχουν ταυτόχρονα, είναι πολύ επιβλαβές για την απόδοση γιατί όλες οι εγγραφές θα καταλήξουν να γίνονται σειριακά.

Στην IW μέθοδο, αντί να ανανεώνονται συνεχώς οι όροι του μητρώου, υπολογίζονται οι τελικές τιμές για κάθε υπομητρώο \mathbf{K}_{ij} και έπειτα τοποθετούνται στο καθολικό μητρώο. Για τον υπολογισμό του εκάστοτε \mathbf{K}_{ij} , πρέπει να αθροιστούν όλες οι συνεισφορές των μονάδων Gauss που ανήκουν στην τομή των πεδίων επιρροής των δύο κομβικών μονάδων. Επομένως, η IW μέθοδος χρησιμοποιεί το λεγόμενο “gather parallelism”, όπως φαίνεται στο σχήμα 8.2. Σε παράλληλη υλοποίηση, κάθε thread αναλαμβάνει ένα υπομητρώο \mathbf{K}_{ij} , συγκεντρώνει όλες τις συνεισφορές των μονάδων Gauss και γράφει σε θέσεις μνήμης που δεν γράφονται από κανένα άλλο thread.

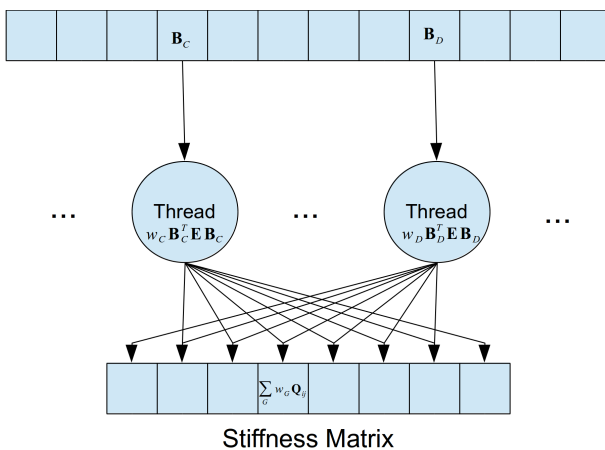


Fig. 8.1: Scatter parallelism στη μέθοδο μορφοποίησης με βάση τις συνεισφορές

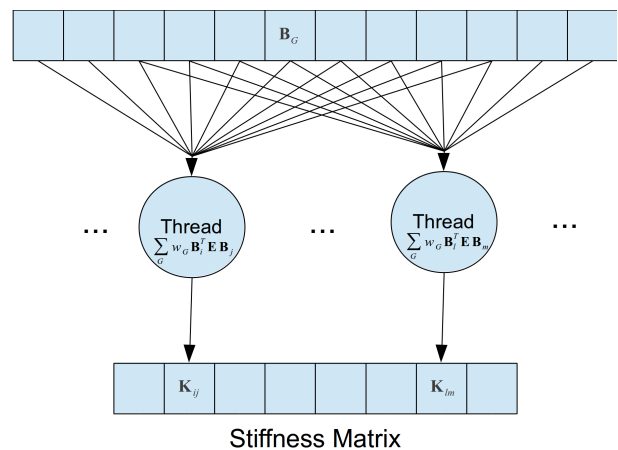


Fig. 8.2: Gather parallelism στη μέθοδο μορφοποίησης με βάση τις αλληλεπιδράσεις

8.4 Υλοποίηση της IW μεθόδου σε GPU

Η μέθοδος μορφοποίησης με βάση τις αλληλεπιδράσεις είναι κατάλληλη για εφαρμογή σε GPU. Οι υπολογισμοί χωρίζονται σε δύο kernels, καθένα από τους οποίους χρησιμοποιεί διαφορετικά επίπεδα παραλληλίας. Υπάρχουν διαφορετικές υλοποιήσεις και για τις δύο φάσεις ανάλογα με το αν η ανάλυση στηρίζεται σε ξεχωριστά σημεία Gauss ή σε στοιχεία. Οι παρακάτω υλοποιήσεις είναι γραμμένες σε openCL για περισσότερη ευελιξία.

8.4.1 Φάση 1 – Υπολογισμός των τιμών ολοκλήρωσης

Στην πρώτη φάση, υπολογίζονται οι τιμές ολοκλήρωσης για κάθε σημείο Gauss πάνω στις κομβικές μονάδες. Οι τιμές αυτές εκφράζουν κάτι διαφορετικό σε κάθε εφαρμογή. Για παράδειγμα, σε δομοστατικές εφαρμογές, υπολογίζονται οι τιμές των συναρτήσεων σχήματος για το μητρώο μάζας ή οι τιμές των παραγώγων των συναρτήσεων σχήματος για το μητρώο δυσκαμψίας. Υπάρχουν δύο επίπεδα παραλληλίας που εκμεταλλευόμαστε:

- Παραλλαγή ξεχωριστών σημείων Gauss: το κύριο επίπεδο παραλληλίας είναι τα σημεία Gauss και το δευτερεύων είναι οι επηρεαζόμενοι κόμβοι.
- Παραλλαγή στοιχείων: το κύριο επίπεδο παραλληλίας είναι τα στοιχεία και το δευτερεύων τα σημεία Gauss

8.4.2 Φάση 2 – Υπολογισμός των όρων του μητρώου

Στη δεύτερη και τελευταία φάση, υπολογίζονται τα υπομητρώα \mathbf{K}_{ij} . Για την αραιή (sparse) μορφή μητρώου που χρησιμοποιεί η IW μέθοδος (COO), αυτό συνεπάγεται απλώς τον υπολογισμό των δεικτών και των αντίστοιχων μη-μηδενικών τιμών για κάθε υπομητρώο. Μια απλή τεχνική για παράλληλη επεξεργασία είναι ο ταυτόχρονος υπολογισμός κάθε ζεύγους αλληλεπίδρασης. Λόγω των χαρακτηριστικών της GPU, όμως, επιλέγεται και ένα δεύτερο επίπεδο παραλληλίας, όπως αναφέρθηκε παραπάνω. Οι τιμές που προέκυψαν από τη φάση 1 αποτελούν μέρος των δεδομένων για τη φάση 2 και βρίσκονται ήδη στη GPU, εφόσον έχουν υπολογιστεί εκεί.

8.5 Αριθμητικά αποτελέσματα για τη μορφοποίηση του μητρώου δυσκαμψίας σε μη-πλεγματικές μεθόδους

Οι παραλλαγές των CW και IW μεθόδων που αφορούν προσομοιώσεις με ξεχωριστά σημεία Gauss υλοποιούνται και εφαρμόζονται για τον υπολογισμό του μητρώου δυσκαμψίας σε 2D και 3D παραδείγματα στατικής στη μη-πλεγματική μέθοδο προσομοίωσης element-free Galekrin (EFG). Η γεωμετρία των φορέων (τετράγωνα, κύβοι) μεγιστοποιεί τις αλληλοσυσχετίσεις και άρα το υπολογιστικό κόστος για τον εκάστοτε αριθμό κόμβων. Τα παραδείγματα εκτελέστηκαν με το παρακάτω hardware. CPU: Core i7-980X που έχει 6 πυρήνες (12 λογικούς πυρήνες) στα 3.33 GHz και 12MB cache. GPU: GeForce GTX680 με 1536 πυρήνες CUDA και 2GB GDDR5 μνήμη. Όλες οι πράξεις κινητής υποδιαστολής είναι με διπλή ακρίβεια.

EFG Example	CW Single Hash T.	CW List of Hash T.	CW Skyline	IW
2D-1	41	32	31	39
2D-2	41	32	29	32
2D-3	40	31	28	33
3D-1	76	58	26	44
3D-2	78	59	27	41
3D-3	74	56	25	41

Table 8.1: EFG: επιταχύνσεις GPU προς CPU για τη μορφοποίηση του μητρώου δυσκαμψίας

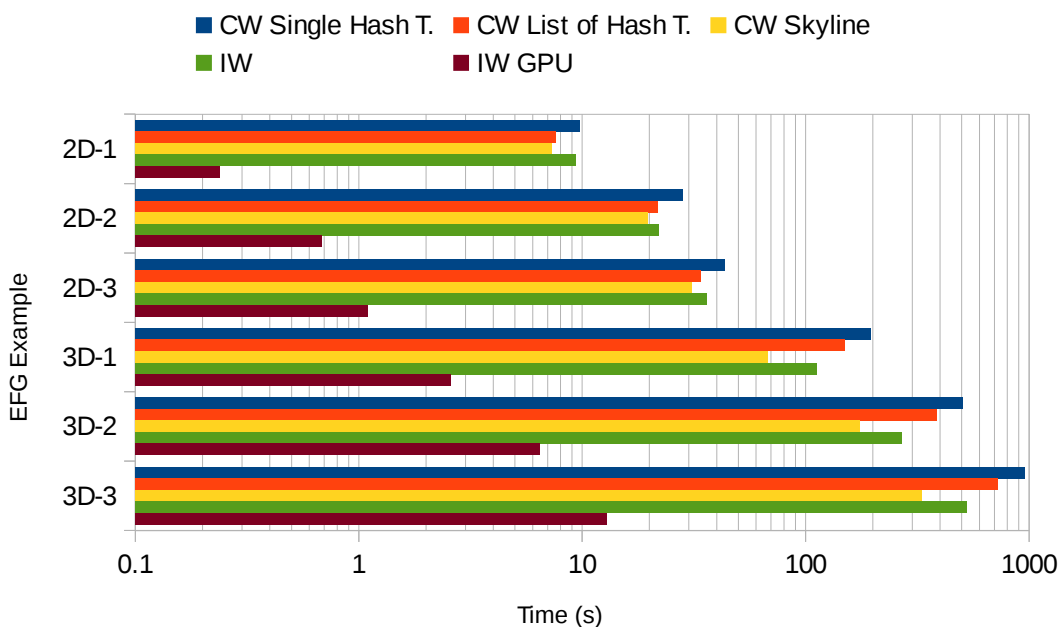


Fig. 8.3: EFG: χρόνος μορφοποίησης του μητρώου δυσκαμψίας με τις CW και IW μεθόδους (παραλλαγές για ξεχωριστά σημεία Gauss).

8.6 Αριθμητικά αποτελέσματα για τη μορφοποίηση του μητρώου δυσκαμψίας στην ισογεωμετρική ανάλυση

Οι παραλλαγές των CW και IW μεθόδων που αφορούν προσομοιώσεις με στοιχεία υλοποιούνται και εφαρμόζονται για τον υπολογισμό του μητρώου δυσκαμψίας σε 2D και 3D παραδείγματα στατικής στην ισογεωμετρική ανάλυση. Οι παράμετροι που χρησιμοποιήθηκαν μεγιστοποιούν τις αλληλοσυσχετίσεις και άρα το υπολογιστικό κόστος για τον εκάστοτε αριθμό σημείων ελέγχου. Τα παραδείγματα εκτελέστηκαν με το παρακάτω hardware. CPU: Core i7-980X που έχει 6 πυρήνες (12 λογικούς πυρήνες) στα 3.33 GHz και 12MB cache. GPU: GeForce GTX680 με 1536 πυρήνες CUDA και 2GB GDDR5 μνήμη. Όλες οι πράξεις κινητής υποδιαστολής είναι με διπλή ακρίβεια.

IGA Example	Non-coalesced		Coalesced	
	CW	IW	CW	IW
2D-P2-1	38	32	39	33
2D-P2-2	35	29	36	30
2D-P2-3	36	30	36	30
2D-P3-1	36	30	46	39
2D-P3-2	34	30	46	40
2D-P3-3	35	30	46	39
2D-P4-1	29	25	47	40
2D-P4-2	30	26	49	42
2D-P4-3	31	26	50	42
3D-P2-1	30	27	46	41
3D-P2-2	30	25	49	40
3D-P2-3	30	25	49	41
3D-P3-1	19	14	63	45
3D-P3-2	19	13	64	45
3D-P3-3	19	13	64	45
3D-P4-1	17	11	84	54
3D-P4-2	17	11	84	54
3D-P4-3	17	11	86	54

Table 8.2: IGA: επιταχύνσεις GPU προς CPU για τη μορφοποίηση του μητρώου δυσκαμψίας

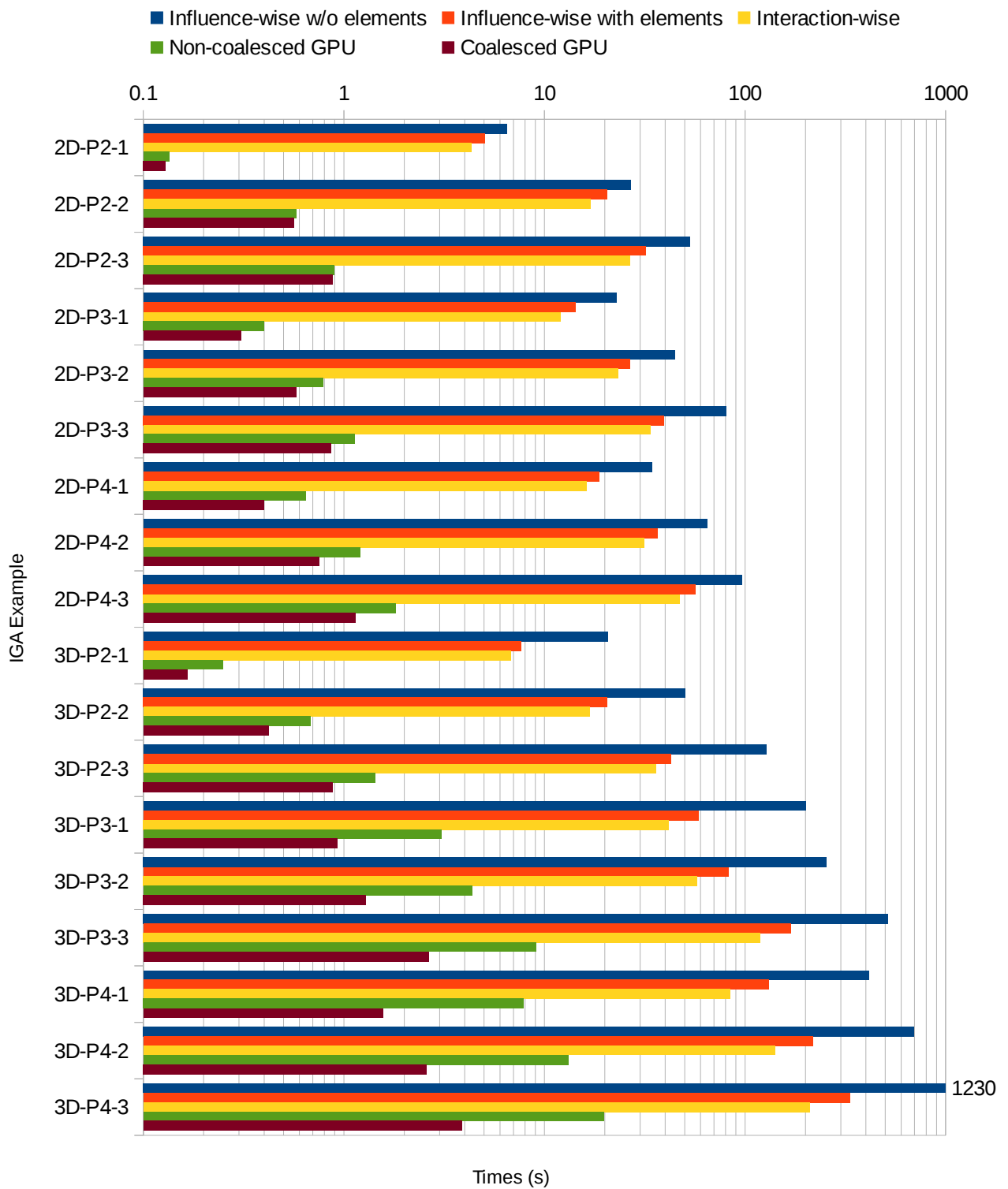


Fig. 8.4: IGA: χρόνος μορφοποίησης του μητρώου δυσκαμψίας με τις CW και IW μεθόδους (παρалаγές για στοιχεία, με εξαίρεση: "CW w/o elements").

Acknowledgements

First, I would like to thank my advisor, Professor Manolis Papadrakakis, for his scientific guidance starting from my graduate studies, throughout my post-graduate studies as well as throughout the years of academic research that lead to this PhD thesis.

For their valuable suggestions and comments, I would also like to express my deepest thanks to the other two members of the PhD advisory committee, namely Professor Andreas Boudouvis and Assistant Professor Vissarion Papadopoulos, as well as Assistant Professor Nikos Lagaros from my graduate and post-graduate advisory committee.

Special thanks to George Stavroulakis for directing me towards sound programming foundations at the start of my research. I would also like to thank Theofilos Manitaras for his support. Many thanks to my co-authors Panagiotis Metsis and Panagiotis Karakitsios.

Special thanks to all the people of the research team of Professor Papadrakakis for maintaining an enjoyable and cutting edge environment. Working with everyone has been a real pleasure.

Athens, May 2014

Alexander Karatarakis

Table of Contents

Abstract	xiii
Σύντομη Περίληψη	xv
Εκτενής Περίληψη	xvii
Acknowledgements	li
Acronyms and Abbreviations	lxxi
1 Introduction	1
1.1 Motivation.....	1
1.2 Aim and objectives.....	1
1.3 Organization and outline.....	2
2 Simulation methods	5
2.1 Meshless/Meshfree methods.....	5
2.1.1 Basic ingredients of element-free Galerkin methods.....	7
2.1.1.1 Basic approximations.....	8
2.1.1.2 Weight functions.....	8
2.1.1.3 Moving least squares (MLS) approximation.....	9
2.1.1.4 Galerkin weak form.....	11
2.1.1.5 Essential Boundary Conditions.....	13
2.1.2 EFG test examples.....	14
2.2 Isogeometric Analysis.....	15
2.2.1 Basic ingredients of isogeometric analysis methods.....	15
2.2.1.1 Non-Uniform Rational B-SPLines (NURBS).....	15
2.2.1.2 Stiffness matrix formulation.....	18
2.2.1.3 Quadrature rule.....	20
2.2.2 IGA Test examples.....	21
2.3 Finite element test examples.....	22
3 Domain decomposition methods	23
3.1 The primal domain decomposition implementation.....	23
3.1.1 Static condensation.....	25
3.1.1.1 LL decomposition.....	26
3.1.1.2 LDL decomposition.....	27
3.2 The dual domain decomposition implementation.....	28

3.2.1 FETI ingredients.....	28
3.2.2 Supported subdomains and supported degrees of freedom.....	36
3.2.3 Floating subdomains.....	37
3.2.4 Linear equations of the FETI interface problem.....	40
3.2.4.1 Matrix F.....	44
3.2.4.2 Vector d.....	44
3.2.4.3 Matrix G.....	45
3.2.4.4 Vector a.....	45
3.2.4.5 Vector e.....	45
3.2.5 Matrices of the boundary problem.....	46
3.2.5.1 Matrix F.....	46
3.2.5.2 Vector d.....	47
3.2.5.3 Matrix G and vector e.....	47
3.2.5.4 Vectors λ and a.....	47
3.2.6 Special cases.....	48
3.2.6.1 Boundary nodes with multiplicity >2	51
3.2.6.1.1 Minimum Constraints.....	52
3.2.6.1.2 Non-Redundant Constraints.....	53
3.2.6.1.3 Fully Redundant Constraints.....	54
3.2.6.2 Constrained boundary nodes.....	55
3.2.7 Solving the FETI interface problem.....	56
3.3 Preconditioners.....	60
3.3.1 General expression of preconditioners.....	61
3.3.2 Dirichlet preconditioner.....	62
3.3.3 Lumped preconditioner.....	63
3.3.4 Diagonal preconditioner.....	63
3.3.5 Preconditioner usage.....	64
3.4 Implementation considerations in the context of FETI.....	65
3.4.1 Matrix format.....	65
3.4.2 Variable type.....	65
3.4.3 Order of calculations.....	66
3.4.4 Boolean matrices.....	67

3.4.4.1 Type A: indexes only.....	68
3.4.4.2 Type B: signed indexes.....	69
3.4.4.3 Type C: Indexes and separate signs.....	70
3.4.4.4 Matrix-vector multiplication for compact B.....	70
3.4.4.4.1 Matrix-vector multiplication for Type A.....	73
3.4.4.4.2 Matrix-vector multiplication for Type B.....	74
3.4.4.4.3 Matrix-vector multiplication for Type C.....	75
3.4.4.5 Left multiplying vector with matrix for compact B.....	76
3.4.4.6 Local to Global Mapping.....	77
3.4.5 Matrix W.....	79
3.4.5.1 Matrix vector multiplication for compact W.....	81
4 Graphics Processing Units (GPUs).....	83
4.1 CPU vs GPU.....	84
4.2 CUDA and OpenCL.....	86
4.3 GPU Hardware.....	87
4.4 GPU Threads.....	90
4.5 Thread Organization.....	90
4.6 Warps and control divergence.....	91
4.7 Block size.....	93
4.8 GPU Memory.....	94
4.8.1 Global Memory.....	95
4.8.2 Constant Memory.....	97
4.8.3 Shared Memory [CUDA] or Local Memory [OpenCL].....	98
4.8.4 Registers.....	98
4.8.5 Other memories.....	98
4.8.6 Data transfer.....	99
4.9 Synchronization.....	99
4.10 Privatization.....	101
4.11 Atomic Operations.....	101
4.12 Reduction.....	103
4.13 Pinned Memory.....	106
4.14 GPU Task Parallelism.....	107

5 Handling of matrices.....	109
5.1 Dense Matrix.....	109
5.1.1 Row-major & column-major entry order.....	110
5.1.2 Implementations.....	110
5.1.2.1 Row-major, 2D array.....	111
5.1.2.2 Row-major, 1D array.....	112
5.1.2.3 Column-major, 2D array.....	113
5.1.2.4 Column-major, 1D array.....	114
5.2 Triangular Dense Matrix.....	115
5.2.1 Implementations.....	115
5.2.1.1 Lower triangular dense storage by row.....	116
5.2.1.2 Lower triangular dense storage by column.....	118
5.2.1.3 Upper triangular dense storage by row.....	120
5.2.1.4 Upper triangular dense storage by column.....	122
5.3 Symmetric Dense Matrix.....	123
5.3.1 Implementations.....	123
5.3.1.1 Lower Triangle by row or Upper Triangle by column.....	124
5.3.1.2 Lower Triangle by column or Upper Triangle by row.....	125
5.4 Diagonal Dense Matrix.....	127
5.5 Bandwidth-aware storage.....	128
5.5.1 Symmetric Banded Matrix.....	130
5.5.2 Symmetric Skyline Matrix.....	131
5.5.3 Factorization.....	135
5.5.4 Banded Factorization.....	136
5.5.5 Skyline Factorization.....	138
5.5.6 Numbering considerations.....	140
5.6 Sparse Matrix.....	145
5.6.1 Sparse Matrix Builders.....	145
5.6.1.1 Coordinate list (COO).....	146
5.6.1.2 Dictionary of Keys (DOK).....	147
5.6.2 Sparse Matrix formats for operations.....	149
5.6.2.1 Compressed Sparse Row (CSR).....	149

5.6.2.2 Compressed Sparse Column (CSC).....	150
5.6.2.3 Other sparse formats.....	151
5.7 Matrix multiplication.....	152
5.8 Order of calculations.....	154
5.9 Transpose.....	158
5.10 Matrices as a collection of vectors.....	159
6 Domain decomposition methods in hybrid CPU-GPU architectures.....	161
6.1 Introduction.....	161
6.2 Dual DDM (FETI) method.....	163
6.3 Hybrid CPU-GPU implementation.....	165
6.3.1 The Choleksy direct solver.....	165
6.3.2 The PCG iterative solver.....	168
6.3.3 The solution at the projection step.....	169
6.3.4 Dot products.....	169
6.3.5 Sparse matrix – vector multiplications.....	170
6.4 Dynamic load-balancing.....	171
6.4.1 Task Parallelism.....	171
6.5 Dynamic load-balancing implementation.....	172
6.6 Numerical results.....	174
6.7 Remarks.....	187
7 Relations between basic entities of Gauss quadrature.....	189
7.1 N-G correlations.....	190
7.2 G-N correlations.....	190
7.3 Interactions.....	191
7.4 Synergies.....	193
7.5 Domain of influence in the simulation methods.....	194
7.5.1 Domain of influence in FEA.....	194
7.5.2 Domain of influence in MMs.....	196
7.5.2.1 Comparison with FEA.....	198
7.5.2.2 Identification of correlations in MMs.....	201
7.5.2.3 Interactions and shared Gauss points in MMs.....	205
7.5.3 Domain of influence in IGA.....	209

7.5.3.1 Comparison with FEA.....	211
7.5.3.2 Interactions.....	214
7.5.3.3 Interaction comparison with FEA for equal number of freedom degrees.....	216
8 Formulation of the characteristic matrices.....	217
8.1 The contribution-wise (CW) method for assembling a matrix.....	217
8.1.1 Gauss point-wise variant of the CW method.....	217
8.1.2 Element-wise variant of the CW method.....	218
8.2 The interaction-wise (IW) method for assembling a matrix.....	219
8.2.1 IW variant with individual Gauss points.....	219
8.2.2 IW variant for element-driven applications.....	220
8.3 Scatter-gather parallelism of the matrix assembly methods.....	221
8.4 GPU implementation of the interaction-wise approach.....	223
8.4.1 Phase 1 – Calculation of quadrature values.....	223
8.4.1.1 Individual Gauss point variant.....	223
8.4.1.2 Element variant.....	225
8.4.2 Phase 2 – Calculation of matrix entries.....	226
8.4.2.1 Two-level individual Gauss point variant.....	227
8.4.2.2 Two-level element variant.....	229
8.5 Memory layout of quadrature values for coalesced access.....	230
8.6 Utilization of available hardware.....	232
8.7 GPU accelerated formulation of the EFG stiffness matrix.....	235
8.7.1 Computation of stiffness contribution for each Gauss point.....	235
8.7.1.1 Shape function derivative calculation.....	235
8.7.1.2 BTEB Calculation.....	237
8.7.2 Performance of the Gauss point-wise variant of the CW method.....	238
8.7.3 Performance of the interaction-wise approach.....	241
8.7.4 GPU implementation of the interaction-wise approach.....	242
8.7.4.1 Phase 1 – Calculation of shape function and derivative values.....	242
8.7.4.2 Phase 2 – Calculation of the global stiffness coefficients.....	243
8.7.5 Performance of the GPU implementation of the interaction-wise approach.....	244
8.7.6 Numerical results.....	244
8.8 GPU accelerated formulation of the IGA stiffness matrix.....	247

8.8.1 BTEB Calculation.....	247
8.8.2 Performance of the element-wise variant of the CW method.....	248
8.8.3 Performance of the interaction-wise approach.....	251
8.8.4 GPU implementation of the interaction-wise approach.....	252
8.8.4.1 Phase 1 – Calculation of shape function and derivative values.....	252
8.8.4.2 Phase 2 – Calculation of the global stiffness coefficients.....	253
8.8.5 Performance of the coalesced and non-coalesced GPU implementations of the interaction-wise approach.....	254
8.8.6 Numerical results.....	255
8.9 Remarks.....	258
9 Overview and concluding remarks.....	261
9.1 Future work.....	264
10 Appendix A: BEB calculations.....	265
10.1 The elasticity tensor in 3D problems.....	265
10.1.1 Anisotropic material.....	265
10.1.2 Orthotropic material.....	265
10.1.3 Isotropic material.....	266
10.2 The deformation matrix in 3D problems.....	267
10.3 Explicit calculation in 3D problems.....	268
10.3.1 Anisotropic material.....	269
10.3.2 Orthotropic material.....	270
10.3.3 Isotropic material.....	271
10.4 Total number of calculations required.....	274
10.5 The elasticity tensor in 2D problems.....	277
10.5.1 Anisotropic material.....	277
10.5.2 Orthotropic material.....	277
10.5.3 Isotropic material under Plane Stress.....	277
10.6 The deformation matrix in 2D problems.....	278
10.7 Explicit calculation in 2D problems.....	279
10.7.1 Anisotropic material.....	280
10.7.2 Orthotropic material.....	280
10.7.3 Isotropic material under Plane Stress.....	280

10.8 Total number of calculations required in 2D problems.....	282
11 References.....	285

List of Figures

Fig. 2.1: Weight functions.....	11
Fig. 2.2: C1 continuous quadratic basis derived from open uniform knot vector.....	16
Fig. 2.3. Partitioning of domain in 125 subdomains.....	22
Fig. 3.1: Example domain.....	29
Fig. 3.2: Numbering of nodes and degrees of freedom of the domain.....	29
Fig. 3.3: Domain teared into two subdomains.....	30
Fig. 3.4: Numbering of nodes and degrees of freedom of the teared domain.....	30
Fig. 3.5: Global numbering of nodes and degrees of freedom of the subdomains.....	31
Fig. 3.6: Equations of subdomains and constraints.....	35
Fig. 3.7: Close constraints.....	40
Fig. 3.8: Spaced-out constraints.....	40
Fig. 3.9: Example 2.....	48
Fig. 3.10: Numbering of nodes and degrees of freedom of the domain.....	48
Fig. 3.11: Domain teared into 4 subdomains.....	49
Fig. 3.12: Numbering of nodes and degrees of freedom of the teared domain.....	49
Fig. 3.13: Global numbering of nodes and degrees of freedom of the subdomains.....	50
Fig. 3.14: Special node cases.....	50
Fig. 3.15: Minimum constraints.....	52
Fig. 3.16: Non-redundant constraints.....	53
Fig. 3.17: Fully redundant constraints.....	54
Fig. 3.18: The PCPG algorithm.....	57
Fig. 3.19: Fully redundant constraints.....	72
Fig. 4.1. GPU processing flow paradigm.....	84
Fig. 4.2: SM vs SMX.....	87
Fig. 4.3: NVIDIA Fermi (GF100) GPU block diagram.....	88
Fig. 4.4: NVIDIA Kepler (GK110) GPU block diagram.....	88
Fig. 4.5. Thread Organization.....	91
Fig. 4.6: 2D thread grid with 2D thread blocks.....	91
Fig. 4.7. Visual Representation of GPU Memory Model and Scope.....	94
Fig. 4.8: DRAM burst example.....	95
Fig. 4.9: Fully coalesced memory access.....	95

Fig. 4.10: Non coalesced memory access.....	96
Fig. 4.11: Non coalesced strided memory access.....	96
Fig. 4.12: Non coalesced memory access due to misalignment.....	96
Fig. 4.13: A 4x4 matrix.....	97
Fig. 4.14: Barrier synchronization.....	100
Fig. 4.15. Parallel reduction tree.....	103
Fig. 4.16: Reduction pattern with thread divergence.....	104
Fig. 4.17: Good reduction pattern.....	105
Fig. 4.18: Data transfer between CPU and GPU.....	107
Fig. 4.19: Without task parallelism (dir. = direction).....	108
Fig. 4.20: With task parallelism (dir. = direction).....	108
Fig. 5.1: Row-major dense storage of a matrix.....	111
Fig. 5.2: Column-major dense storage of a matrix.....	113
Fig. 5.3: Storage by row of an lower triangular matrix.....	116
Fig. 5.4: Storage by column of an lower triangular matrix.....	118
Fig. 5.5: Storage by row of an upper triangular matrix.....	120
Fig. 5.6: Storage by column of an upper triangular matrix.....	122
Fig. 5.7: Symmetric storage of an symmetric matrix.....	124
Fig. 5.8: Symmetric storage of an symmetric matrix.....	125
Fig. 5.9: Storage of an diagonal matrix.....	127
Fig. 5.10: Entries stored with symmetric banded storage for a symmetric matrix.....	130
Fig. 5.11: Entries stored with symmetric skyline storage for an symmetric matrix.....	131
Fig. 5.12: Order of stored entries with symmetric skyline storage for an symmetric matrix.....	131
Fig. 5.13: The number of entries in the skyline storage as well as the extra entry needed in the array.	133
Fig. 5.14: The highest entries of columns and	134
Fig. 5.15: decomposition for symmetric matrices.....	135
Fig. 5.16: Entries involved in dot product between columns , (red line).....	136
Fig. 5.17: Entries involved in dot product between columns , (red line) for banded matrices.....	137
Fig. 5.18: decomposition for symmetric banded matrices.....	137
Fig. 5.19: Entries involved in dot product between columns , (red line) for a skyline matrix.....	138
Fig. 5.20: decomposition for symmetric banded matrices.....	139

Fig. 5.21: Good numbering of degrees of freedom. Grey = non-zero entries.....	140
Fig. 5.22: Bad numbering of degrees of freedom. Grey = non-zero entries.....	140
Fig. 5.23: Global numbering.....	141
Fig. 5.24: Local numbering.....	141
Fig. 5.25:Element numbering.....	141
Fig. 5.26: Numbering A. Grey = entries up to the highest non-zero entry of the column.....	143
Fig. 5.27: Numbering B. Grey = entries up to the highest non-zero entry of the column.....	143
Fig. 5.28: Bandwidth improvement through the (reverse) Cuthill-McKee algorithm.....	144
Fig. 5.29: Ordering of operations in linear algebra libraries.....	157
Fig. 6.1: The PCPG algorithm used for the solution of the interface problem.....	164
Fig. 6.2. Time in ms for factorizing a subdomain kernel. Horizontal axis represents the simultaneous factorizations computed at the GPU in parallel.....	167
Fig. 6.3: The PCG algorithm.....	168
Fig. 6.4. The task queue contains numerical computations to be performed by available resources	172
Fig. 6.5: Number of iterations for the PCPG solution of the interface problem with Cholesky and PCG subdomain solvers.....	174
Fig. 6.6: Computing time per subdomain for Cholesky factorization.....	176
Fig. 6.7: Computing time per subdomain for forward and backward substitutions.....	176
Fig. 6.8: Computing time per subdomain for PCG solution.....	177
Fig. 6.9: Computing time per subdomain for sparse matrix-vector multiplication.....	177
Fig. 6.10: Optimum subdomain distribution between CPU and GPU for the factorization and forward/backward substitutions of the Cholesky subdomain solver with the i7 and GTX 285 combination.....	178
Fig. 6.11: Optimum subdomain distribution between CPU and GPU for SpMV multiplications and PCG subdomain solver with the i7 and GTX 285 combination.....	178
Fig. 6.12: Optimum subdomain distribution between CPU and GPU for the factorization and forward/backward substitutions of the Cholesky subdomain solver with the i7 and GTX 580 combination.....	179
Fig. 6.13: Optimum subdomain distribution between CPU and GPU for SpMV multiplications and PCG subdomain solver with the i7 and GTX 580 combination.....	179
Fig. 6.14: Total solution time of FETI for the Hybrid, GPU (GTX285) only and CPU (i7) only cases	

with the Cholesky subdomain solver.....	181
Fig. 6.15: Total solution time of FETI for the Hybrid, GPU (GTX285) only and CPU (i7) only cases with the PCG subdomain solver.....	181
Fig. 6.16: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 285) with the Cholesky subdomain solver.....	182
Fig. 6.17: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 285) with the PCG subdomain solver.....	182
Fig. 6.18: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 285) with the Cholesky subdomain solver.....	183
Fig. 6.19: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 285) with the PCG subdomain solver.....	183
Fig. 6.20: Total solution time of FETI for the Hybrid, GPU (GTX580) only and CPU (i7) only cases with the Cholesky subdomain solver.....	184
Fig. 6.21: Total solution time of FETI for the Hybrid, GPU (GTX580) only and CPU (i7) only cases with the PCG subdomain solver.....	184
Fig. 6.22: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 580) with the Cholesky subdomain solver.....	185
Fig. 6.23: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 580) with the PCG subdomain solver.....	185
Fig. 6.24: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 580) with the Cholesky subdomain solver.....	186
Fig. 6.25: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 580) with the PCG subdomain solver.....	186
Fig. 7.1: N-G correlations: each nodal entity N is associated to all Gauss entities G influencing it.	190
Fig. 7.2: G-N correlations: each Gauss entity G is associated with all nodal entities N it influences.	190
Fig. 7.3: Nodal entity interactions.....	191
Fig. 7.4: Identifying interacting pairs for nodal entity	192
Fig. 7.5: Interacting pairs with shared Gauss entities.....	193
Fig. 7.6: FEA: domain of influence of node.....	194
Fig. 7.7: FEA: domain of infl. of Gauss p.....	194

Fig. 7.8: FEA: interacting nodes for $p = 2$	195
Fig. 7.9: Domain of influence of Gauss point in (a) MMs; (b) FEA, for the same number of nodes () and Gauss points ().....	196
Fig. 7.10: Domain of influence of node (a) MMs; (b) FEA, for the same number of nodes () and Gauss points ().....	197
Fig. 7.11: MMs: node interactions (nodes:, Gauss points).....	198
Fig. 7.12: Interacting nodes: (a) MMs; (b) FEA for the same number of nodes () and Gauss points ()	199
Fig. 7.13: Global search for correlations (nodes:, Gauss points).....	202
Fig. 7.14: Regioned search for correlations (nodes:, Gauss points).....	202
Fig. 7.15: Identifying the influencing Gauss points of node.....	203
Fig. 7.16: Interaction nodes with their shared Gauss points.....	205
Fig. 7.17: Identifying interacting node pairs for node	206
Fig. 7.18 Identifying interacting node pairs by considering Gauss points near the border of the domain of influence.....	207
Fig. 7.19: Region-wise search for interacting nodes. Only the shaded regions are inspected for shared Gauss points.....	208
Fig. 7.20. IGA 1D domains of influence of control points for various values of p	209
Fig. 7.21. Areas influencing control point in: (a) IGA (p odd); (b) IGA (p even); (c) FEA. The influencing entities are the Gauss points in the shaded areas.....	210
Fig. 7.22. Control points/nodes influenced by Gauss point in: (a) IGA (p odd); (b) IGA (p even); (c) FEA.....	210
Fig. 7.23. Visual comparison between (a)IGA p =even; (b) FEA p =2, for the same number of control points/nodes.....	212
Fig. 7.24. Visual comparison between (a)IGA p =odd; (b) FEA p =3, for the same number of control points/nodes.....	212
Fig. 7.25. Interacting control points/nodes for $p = 2$: (a) IGA; (b) FEA.....	214
Fig. 7.26. IGA Interacting control points for p =3.....	215
Fig. 8.1: Scatter parallelism in the contribution-wise approach.....	222
Fig. 8.2: Gather parallelism in the interaction-wise approach.....	222
Fig. 8.3: Thread organization in phase 1 for the individual Gauss point variant. A thread block/group is assigned to Gauss point G and each thread of the block/group is assigned to an influenced node of	

G.....	224
Fig. 8.4: Thread organization in phase 1 for the element variant. A thread block/group is assigned to element E and each thread of the block/group is assigned to a Gauss point of E. Each thread iterates over all influenced nodes	225
Fig. 8.5: Thread organization in phase 2 for the individual Gauss point variant showing the threads assigned to the shared Gauss points G of interacting pair	227
Fig. 8.6: Phase 2: concurrency within a block/group for the assembly phase in the GPU.....	228
Fig. 8.7. Thread organization in phase 2 for the element variant showing the threads assigned to the Gauss points G of the shared elements E of interacting pair	229
Fig. 8.8. Memory layout of quadrature values of an element “e”. The values are grouped together by Gauss point (G), and each group contains values for all influenced nodal entities (N). This leads to non-coalesced access pattern: consecutive threads access non-consecutive memory locations.....	231
Fig. 8.9. Memory layout of quadrature values of an element “e”. The values are grouped together by nodal entity (N), and each group contains values for all influencing Gauss points (G). This leads to coalesced access pattern: consecutive threads access consecutive memory locations.....	231
Fig. 8.10: Fully processing the nodes of the core also requires data from the halo surrounding it..	232
Fig. 8.11: Schematic representation of the processing of node pairs utilizing all available hardware.	234
Fig. 8.12: Phase 1 - Concurrency level for the calculation of shape function values in the GPU....	243
Fig. 8.13: Overview of the numerical results obtained for EFG with the contribution-wise (CW) and interaction-wise (IW) methods (individual Gauss point variants).....	245
Fig. 8.14. IW approach: Phase 1 - Concurrency within a block/group of threads for the calculation of shape function values in the GPU.....	253
Fig. 8.15: Overview of the numerical results obtained for IGA with the contribution-wise (CW) and interaction-wise (IW) methods (element variants, except “CW w/o elements”).....	256

List of Tables

Table 2.1: List of meshless methods.....	6
Table 2.2: 2D and 3D example details for EFG test examples.....	14
Table 2.3: 2D and 3D example details for IGA test examples.....	21
Table 2.4: Example details for FEA.....	22
Table 3.1: Numbering of interface nodes.....	32
Table 3.2: Storage formats for signed Boolean matrix.....	67
Table 4.1: CUDA vs OpenCL terminology.....	86
Table 4.2: Important GPU properties.....	89
Table 4.3: Specifications for GTX 580, GTX 680 and GTX Titan.....	89
Table 4.4: Block granularity considerations for Compute capability 2.0.....	93
Table 4.5: Memory scope and lifetime.....	94
Table 4.6: Desirable order of execution.....	101
Table 4.7: Undesirable order of execution.....	102
Table 5.1: Column height (diagonal exclusive) for each column for numbering A.....	142
Table 5.2: Column height (diagonal exclusive) for each column for numbering B.....	142
Table 5.3: Combinations of typical backing data structures for the DOK format. “Internal” refers to the data structures that stores rows/columns, whereas “external” is the data structure that stores references to them.....	148
Table 5.4: Matrix multiplication calculations for various types of matrices.....	152
Table 5.5: Multiplication from left to right.....	154
Table 5.6: Multiplication from right to left.....	155
Table 5.7: Example: multiplication from left to right.....	155
Table 5.8: Example: multiplication from right to left.....	155
Table 5.9: Multiplication from right to left with vector in the right.....	157
Table 5.10: Multiplication of 3 matrices and a vector.....	157
Table 6.1. Performance of PCG with and without re-orthogonalization.....	175
Table 7.1: Influences per node and Gauss point for EFG and FEA.....	198
Table 7.2: Total number of node-Gauss point correlations in EFG and FEA.....	199
Table 7.3: Number of interacting node pairs in EFG and FEA.....	200
Table 7.4: Total synergies for EFG and FEA.....	200
Table 7.5: Computing time required for all node-Gauss point correlations.....	204

Table 7.6: Computing time required for a naive identification of interacting nodes and their shared Gauss points.....	206
Table 7.7: Computing time for the identification of interacting nodes.....	206
Table 7.8: Computing time for the identification of interacting nodes by only inspecting Gauss points near the border.....	206
Table 7.9: Computing time to identify the shared Gauss points of an interacting node pair.....	208
Table 7.10: Computing time to identify the shared Gauss points of an interacting node pair with regioning.....	208
Table 7.11. Total elements and Gauss points in IGA and FEA, for $n = 121$ control points/nodes and different p , in 2D and 3D square and cubic domains.....	211
Table 7.12. Total elements in IGA and FEA with respect to p	211
Table 7.13. Correlations of nodes with elements and Gauss points for FEA.....	213
Table 7.14. Correlations of control points with elements and Gauss points for IGA.....	213
Table 7.15. Number of Gauss points influencing a control point/node with respect to p	214
Table 7.16. Interactions per control point/node in IGA and FEA.....	216
Table 7.17. Interactions per control point/node with respect to p	216
Table 8.1: Metrics of the EFG 2D and 3D elasticity problems.....	239
Table 8.2: Computing times for the formulation of the EFG stiffness matrix with the Gauss point-wise method for different sparse matrix builder implementations.....	239
Table 8.3: Computing times for the formulation of the EFG stiffness matrix when using sparse and skyline matrix formats in the Gauss point-wise approach.....	240
Table 8.4: Stored entries comparison of the EFG stiffness matrix when using sparse and skyline matrix formats.....	240
Table 8.5: Computing times for the formulation of the EFG stiffness matrix when using the interaction-wise variant with individual Gauss points.....	241
Table 8.6: Computing times for the formulation of the EFG stiffness matrix in the GPU implementation of the interaction-wise approach with individual Gauss points.....	244
Table 8.7: EFG: speedups obtained with the GPU implementation compared to the CPU implementations for the matrix formulation.....	245
Table 8.8: Best achieved elapsed time until the finish of the formulation of the stiffness matrix..	246
Table 8.9: Best projected elapsed time until the finish of the formulation of the stiffness matrix in a hybrid implementation.....	246

Table 8.10: Metrics of the IGA 2D and 3D elasticity problems.....	249
Table 8.11: Computing times for the formulation of the IGA stiffness matrix with the Gauss point-wise and element-wise approaches.....	249
Table 8.12. Single core CPU computing time for the formulation of the stiffness matrix in the element-wise (EW) approach with sparse and skyline storage.....	250
Table 8.13. Number of stored stiffness elements for skyline and sparse storage.....	250
Table 8.14: Computing times for the formulation of the IGA stiffness matrix when using the interaction-wise variant with elements.....	251
Table 8.15: Computing times for the formulation of the IGA stiffness matrix in the non-coalesced GPU implementation of the interaction- wise approach with elements.....	254
Table 8.16: Computing times for the formulation of the IGA stiffness matrix in the coalesced GPU implementation of the interaction-wise approach with elements.....	255
Table 8.17: IGA: Speedups obtained with the non-coalesced and coalesced GPU implementations compared to the CPU implementations.....	257
Table 10.1: Multiplications, additions and temporary variables required for a single pair of nodes and at a single Gauss point.....	274
Table 10.2: Multiplications required for calculations at a single Gauss point.....	275
Table 10.3: Multiplications required for calculations at a single Gauss point.....	276
Table 10.4: Multiplications, additions and temporary variables required for a single pair of nodes and at a single Gauss point.....	281
Table 10.5: Multiplications required for calculations at a single Gauss point.....	282
Table 10.6: Multiplications required for calculations at a single Gauss point (alternative way).....	283

Acronyms and Abbreviations

The following acronyms and abbreviations are used in the the dissertation. Acronyms are also spelled out the first time they appear in a section.

Acronym	Description
CPU	C entral P rocessing U nit
CUDA	C ompute U nified D evice A rchitecture
DDM	D omain D ecomposition M ethod
EFG	E lement F ree G alerkin
FEA / FEM	F inite E lement A nalysis / M ethod
FETI	F inite E lement T earing & I nterconnecting
GPU	G raphics P rocessing U nit
IGA	I sogeometric A nalysis
MMs	M eshless M ethods
SpMV	S parse M atrix- V ector (multiplication)

Abbreviation	Description	
	Latin	English
e.g.	exempli gratia	for example
et al.	et alii	and others
etc.	et cetera	and so forth
i.e.	id est	that is
p. or pp.		page

1 Introduction

1.1 Motivation

The primary purpose of engineering analysis is to provide numerical simulation of a physical phenomenon in a way that is accurate but also computationally feasible. The need to accurately simulate various physical processes in complex geometries is important, and has perplexed scientists for many years. The up-to-date simulation methods can accurately model the physical domain but often require high computational effort. A numerical simulation needs to be performed within a reasonable time-frame with the given computational resources in order to be affordable in real-world applications. Thus, an important aspect in terms of feasibility is the efficient implementation of a simulation methods that enables its application in large-scale problems.

While the prevailing cost in traditional finite element analysis (FEA) is in the solution phase, the main drawback of meshless methods (MMs) and isogeometric analysis (IGA) when addressing real-world problems is the high cost for the formulation of the characteristic matrices. Therefore, in order to make them efficient in large-scale simulations, these methods require massively parallel algorithms not only for the solution phase but also for the assembly of the characteristic matrices.

1.2 Aim and objectives

The aim of this work is to accelerate computationally expensive parts of the most popular numerical simulation methods in a manner that is both efficient and scalable. To that end, the main objectives were:

- To investigate algorithms that enable massive parallelism for all parts of the simulation phase.
 - Domain decomposition methods split the domain into several subdomains and allows their concurrent solution.
 - Assembling the matrix by non-zero allows different parts of the matrix to be calculated in parallel.

Often, the algorithms needed for an efficient serial implementation differ from those applicable to a massively parallel one.

- To make numerical operations involved in the algorithm as efficient as possible. Hence, operations like matrix calculations need to be performed with a format that carefully balances calculation time and space requirements.
- To explore details of Graphics Processing Units (GPUs). GPUs have a different programming model than CPUs and include a variety of different memories and special characteristics that need to be appropriately exploited in efficient implementations.
- To develop efficient CPU implementations for the initial and intermediate phases of a simulation. This mostly applies to MMs where the initialization phase is particularly expensive due to the absence of a grid.
- To develop GPU implementations for computationally expensive part of the simulation methods. In this work, the focus is on:
 - The solution phase in FEA
 - The formulation phase in MMs and IGA

1.3 Organization and outline

Chapter 2 describes the three numerical simulation methods used in this work, i.e. FEM, MMs (focusing on element free Galerkin methods - EFG) and IGA. The test examples that are used throughout this work are outlined in this chapter.

Chapter 3 is dedicated to domain decomposition solution methods. Domain decomposition methods allow the exploitation of the natural parallelism offered by the subdivision of the physical domains to a number of subdomains. The primal domain decomposition method is briefly described followed by an extensive presentation of the dual domain decomposition method (DDM/FETI), which is used in this work. The basic ingredients of FETI are presented, including floating subdomains which constitute a particular characteristic of the method. The solution of the linear equations of the FETI interface problem is discussed along with preconditioners and implementation considerations.

Chapter 4 presents graphics processing units (GPUs) and their characteristic properties. In a massively parallel context, GPUs are particularly interesting. This is due to their low cost, low

energy consumption and high performance. GPUs are parallel devices of the SIMD (single instruction, multiple data) classification and require a large number of threads to be effectively utilized (thousand, usually more). As a result, the principles of massively parallel programming directly apply to GPUs. GPU technology has matured considerably in the last years and is currently improving at a very fast pace. Chapter 4 presents details that need to be considered in any GPU implementation and are thus vital for the efficiency of the simulation methods considered in this work.

Chapter 5 deals with the handling of matrices that are frequently encountered in simulation implementations. Matrix storage and matrix operations are important performance factors for large-scale simulations. The choice of an appropriate format for the task at hand may significantly affect performance. Chapter 5 presents matrix formats that are commonly used in simulations along with appropriate storage schemes and implementation considerations. These include dense, banded, skyline formats as well as a variety of sparse formats.

Chapter 6 contains the hybrid CPU-GPU implementation of the FETI domain decomposition method along with supporting numerical results. The solution of the subdomain problems is tested with a direct Cholesky solver as well as with an iterative PCG solver, for different number of subdomains. An important strategy discussed in Chapter 6 is dynamic load-balancing. The dynamic load balancing with task parallelism and the parallel implementation of the sparse matrix-vector multiplications and dot products ensure that all components of the workstation are constantly busy with calculations resulting in full exploitation of their computing resources. The dynamic load balancing allows the efficient utilization of different CPUs and GPUs as well as any number of CPU cores or GPUs, while making sure that all components are used to their full capacity.

Relations between the basic entities (nodes, Gauss points, control points) are discussed in Chapter 7. Relations are presented in an abstract manner to cover all Gaussian quadrature-based methods before being applied to the specific simulation methods used in this work (FEA, MMs, IGA). Chapter 7 extensively deals with the domain of influence and its particular characteristics in each simulation method. The domain of influence is a fundamental factor that dictates the density and cost of the characteristic matrices of each method. A cost comparison of MMs and IGA with FEM is included to further highlight the differences and challenges between the methods. Generic techniques are presented for the identification of the relations. Furthermore, in the case of MMs where the identification is quite laborious, efficient algorithms are presented to improve the cost of

identification.

Chapter 8 is dedicated to the formulation of the characteristic matrices of the simulation methods. The techniques discussed in Chapter 8 are applicable to any simulation method whose characteristic matrices are based on Gaussian quadrature. Two primary formulation methods are presented: the standard contribution-wise (CW) method and the parallel-friendly interaction-wise (IW) method. Both methods have two variants, one that handles Gauss points explicitly and one that handles Gauss points as part of elements (or other groups). The CW approach is the typical approach for assembling matrices with Gauss quadrature. For simulation methods with a large number of contributions to the global matrix, a fine tuned matrix format for the assembly phase can greatly improve the performance properties of the CW approach. Dense or skyline formats feature fast indexing but use considerably more memory than sparse formats and thus can be prohibitive for large-scale simulations. Sparse matrix formats have the lowest memory cost but higher indexing cost, so sparse formats specifically tailored for the assembly phase are used in this work. The IW method has several advantages with respect to the CW approach. The most important one is its amenability to parallelism especially in massively parallel systems like the GPUs. Each interacting pair can be processed separately by any available processor in order to compute the corresponding submatrix.

GPU implementations are applied to the IW approach offering great speedups compared to CPU implementations. The interacting pairs keep the GPU constantly busy with calculations resulting in high hardware utilization. The IW approach offers great portability since it can be applied to any available hardware achieving even lower computing times when combined with many GPUs, hybrid CPU(s)/GPU(s) implementations and generally any available processing unit. The importance of this flexibility becomes apparent when considering contemporary and future developments like heterogeneous computing systems architecture.

2 Simulation methods

The need to accurately simulate various physical processes in complex geometries is important, and has been the subject of intensive research by scientists and engineers. The most widely used simulation method is the finite element method (FEM/FEA). In the last decade, two powerful simulation methods have also attracted the interest of the research community: meshless methods (MMs) and isogeometric analysis (IGA). MMs and IGA both have their own merits compared to FEM but they also have some shortcomings. One of the downsides of both methods is the significantly increased cost for the formulation of the characteristic matrices.

2.1 Meshless/Meshfree methods

Many numerical simulation schemes exist for solving engineering problems and mesh-based methods are widely used. In mesh-based methods, such as FEM, finite difference (FDM) and finite volume (FVM), each point has a fixed number of predefined neighbors. In simulations where the material being simulated can move around (as in computational fluid dynamics) or where large deformations of the material can occur (as in simulations of plastic materials), the connectivity of the mesh can be difficult to maintain without introducing error into the simulation [1]. If the mesh becomes degenerate, the simulation values may introduce inaccuracies. Remeshing can be used to remedy the situation but this can also introduce errors and affect the computational cost. Furthermore, complex geometries may require extensive meshing and involve difficulties when simulated with mesh-based methods [2].

Meshless or meshfree methods (MMs) are a particularly attractive family of methods that are receiving attention because they are relatively simple, accurate, and require no meshing. In particular, MMs are numerical simulation methods that do not require a mesh connecting the characteristic points of the simulation domain [1]. MMs can enable the simulation of otherwise difficult problems and can provide solutions with increased accuracy while also avoiding the need for mesh connectivity and consequently mesh-related weaknesses and limitations [3]. Manpower time is reduced when compared to mesh-based methods which sometimes require human assistance for the generation of useful meshes. MMs can also easily cope with creation/destruction of characteristic points during the simulation as well as large deformations and discontinuities that do not align with element edges. However, the higher quality of the analysis with MMs is accompanied with increased computational cost for the assembly of the matrices and the solution of the resulting

algebraic equations. In particular, the characteristic matrices have larger bandwidth, are more densely populated and their formulation complexity is substantially increased.

Boundary cloud method	BCM	
Boundary node method	BNM	
Diffuse element method	DEM	1992
Discrete Least Squares Meshless method	DLSM	2006
Discrete Vortex Method	DVM	
Dissipative particle dynamics	DPD	1992
Element-free Galerkin method	EFG	1994
Finite cloud method	FCM	
Finite Mass Method	FMM	2000
Finite pointset method	FPM	1998
Generalized finite difference method	GFDM	
Local Radial Basis Function Collocation Method	LRBFCM	
Material Point Method	MPM	
Meshfree local radial point interpolation method	RPIM	
Meshless local Petrov Galerkin	MLPG	
Method of Finite Spheres	MFS	
Method of fundamental solution	MFS	
Method of particular solution	MPS	
Moving particle finite element method	MPFEM	
Moving particle semi-implicit	MPS	
Natural element method	NEM	
Repeated Replacement Method	RRM	2012
Reproducing kernel particle method	RKPM	1995
Smoothed particle hydrodynamics	SPH	1977
Smoothed point interpolation method	S-PIM	2005

Table 2.1: List of meshless methods

There is a wide variety of MMs, as can be seen in Table 2.1. The more common techniques include kernel methods, element-free Galerkin, meshless Petrov-Galerkin, smooth-particle hydrodynamics, and radial basis functions. Each technique has particular traits and advantages for specific classes of problems. Related to MMs is the moving least squares and partition of unity methods as well as FEA methods that combine some meshless aspects, like extended FEM (XFEM). MMs are thoroughly analyzed in [4] with a particular focus on the most important techniques.

One of the first and most prominent meshless method is the element free Galerkin (EFG) method introduced by Belytschko et al. [5]. EFG requires only nodal data, no element connectivity is needed to construct the shape functions. However a global background cell structure is necessary for the numerical integration. Moreover, since the number of interactions between nodes and/or integration points is heavily increased, due to large domains of influence, the resulting matrices are more densely populated and the computational cost for the formulation and solution of the problem is much higher than in the conventional FEA [5].

To improve the computational efficiency of MMs, parallel implementations like the MPI parallel

paradigm has been used in large scale applications [6], [7] and several alternative methodologies have been proposed concerning the formulation of the problem. The smoothed FEA (SFEM) [8] couples FEM with meshless methods by incorporating a strain smoothing operation used in the mesh-free nodal integration method. The linear point interpolation method (PIM) [9] obtains the partial derivatives of shape functions effortlessly due to the local character of the radial basis functions. A coupled EFG/boundary element scheme [10], taking advantage of both the EFG and the boundary element method. Furthermore, solvers which perform an improved factorization of the stiffness matrix and use special algorithms for realizing the matrix-vector multiplication are proposed in [11], [12]. Divo and Kassab [13] presented a domain decomposition scheme on a meshless collocation method, where collocation expressions are used at each subdomain with artificial created interfaces. Wang et al. [9] presented a parallel reproducing kernel particle method (RKPM), using a particle overlapping scheme which significantly increases the number of shared particles and the time for communicating information between them. Recently, a novel approach for reducing the computational cost of EFG methods is proposed by employing domain decomposition techniques on the physical as well as on the algebraic domains [14]. In that work the solution of the resulting algebraic problems is performed with the dual domain decomposition FETI method with and without overlapping between the subdomains. The non-overlapping scheme has led to a significant decrease of the overall computational cost.

An extensive study on meshless methods can be found in [15].

2.1.1 Basic ingredients of element-free Galerkin methods

The EFG method, which is the method used in this work, is based on the diffuse elements method originated by Nayroles et al. [16]. The major features of the method are: (i) Moving least squares approximation for the construction of shape functions. (ii) Galerkin weak form with constraints to develop the discrete system of equations. (iii) Background cells to perform the numerical integration for assembling system matrices.

2.1.1.1 Basic approximations

Meshless approximations for a scalar function u in terms of the Lagrangian coordinates can be written as

$$u(\mathbf{x}, t) = \sum_{i \in S} \Phi_i(\mathbf{x}) u_i(t) \quad (2.1)$$

where $\Phi_i: \Omega \rightarrow \mathfrak{R}$ are the shape functions, u_i is the nodal values at particle i located at position \mathbf{x}_i , and S is the set of nodes for which $\Phi_i(\mathbf{x}) \neq 0$. The shape functions in eq. (2.1) are only approximants and not interpolants, since $u_i \neq u(\mathbf{x}_i)$. Therefore special techniques are needed to treat Dirichlet boundary conditions.

2.1.1.2 Weight functions

The shape functions Φ_i are obtained from the kernel functions, often called weight functions, which are denoted by $w_i: \Omega \rightarrow \mathfrak{R}$. In our study we use the cubic spline weight function:

$$w(r) = \begin{cases} \frac{2}{3} - 4r^2 + 4r^3 & , \quad r \leq \frac{1}{2} \\ \frac{4}{3} - 4r + 4r^2 - 4r^3 & , \quad \frac{1}{2} < r \leq 1 \\ 0 & , \quad r > 1 \end{cases} \quad (2.2)$$

$$r = \frac{\|\mathbf{x}_i - \mathbf{x}\|}{d_i} \quad (2.3)$$

where d_i is the support size of node i . The support size is a parameter which is crucial to solution accuracy, stability and computational cost, as it defines the bandwidth of the system matrices.

In 2D problems, circular and rectangular support types are commonly used:

$$\text{Circular} \quad w(\mathbf{x} - \mathbf{x}_i) = w\left(\frac{\|\mathbf{x}_i - \mathbf{x}\|}{d_i}\right) \quad (2.4)$$

$$\text{Rectangular} \quad w(\mathbf{x} - \mathbf{x}_i) = w\left(\frac{|\mathbf{x}_i - \mathbf{x}|}{d_i^x}\right) w\left(\frac{|\mathbf{y}_i - \mathbf{y}|}{d_i^y}\right) \quad (2.5)$$

2.1.1.3 Moving least squares (MLS) approximation

The approximation $u^h: \Omega \rightarrow \mathfrak{R}$ of the function $u: \Omega \rightarrow \mathfrak{R}$ is posed as a polynomial of order m with non-constant coefficients. The local approximation around a point $\bar{\mathbf{x}}: \Omega \rightarrow \mathfrak{R}$, evaluated at a point $\mathbf{x}: \Omega \rightarrow \mathfrak{R}$ is given by

$$u_L^h(\mathbf{x}, \bar{\mathbf{x}}) = \mathbf{p}^T(\mathbf{x}) \mathbf{a}(\bar{\mathbf{x}}) \quad (2.6)$$

where $\mathbf{p}(\mathbf{x})$ is a complete polynomial of order m and $\mathbf{a}(\bar{\mathbf{x}})$ contains non constant coefficients that depend on \mathbf{x} (hence the name ‘‘moving’’):

$$\mathbf{p}^T(\mathbf{x}) = [1 \quad x \quad x^2 \quad \dots \quad x^m] \quad (2.7)$$

$$\mathbf{a}^T(\bar{\mathbf{x}}) = [a_0(\mathbf{x}) \quad a_1(\mathbf{x}) \quad a_2(\mathbf{x}) \quad \dots \quad a_m(\mathbf{x})] \quad (2.8)$$

In two dimensional problems, the linear basis $\mathbf{p}(\mathbf{x})$ is given by

$$\mathbf{p}^T(\mathbf{x}) = [1 \quad x \quad y], \quad m=3 \quad (2.9)$$

and the quadratic basis by

$$\mathbf{p}^T(\mathbf{x}) = [1 \quad x \quad y \quad x^2 \quad y^2 \quad xy], \quad m=6 \quad (2.10)$$

The unknown parameters $a_j(\mathbf{x})$ are determined at any point \mathbf{x} , by minimizing a functional $J(\mathbf{x})$ defined by a weighted average over all nodes $i \in 1, \dots, n$:

$$J(\mathbf{x}) = \sum_{i=1}^n w(\mathbf{x} - \mathbf{x}_i) [u_L^h(\mathbf{x}_i, \mathbf{x}) - u_i]^2 = \sum_{i=1}^n w(\mathbf{x} - \mathbf{x}_i) [\mathbf{p}^T(\mathbf{x}_i) \mathbf{a}(\mathbf{x}) - u_i]^2 \quad (2.11)$$

where the parameters u_i are specified by the difference between the local approximation $u_L^h(\mathbf{x}, \bar{\mathbf{x}})$ and the value u_i , at node i , of the function u to be approximated, while n is the number of nodes in the neighborhood of \mathbf{x} where the weight function $w(\mathbf{x} - \mathbf{x}_i) \neq 0$. An extremum of $J(\mathbf{x})$ in eq.(2.11) with respect to the coefficients $a_j(\mathbf{x})$ can be obtained by setting the derivative of J with respect to $\mathbf{a}(\mathbf{x})$ equal to zero [17]. The following equations result:

$$\begin{aligned}
\sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) 2\mathbf{p}_1(\mathbf{x}_i) [\mathbf{p}^T(\mathbf{x}_i)\mathbf{a}(\mathbf{x})-u_i] &= 0 \\
\sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) 2\mathbf{p}_2(\mathbf{x}_i) [\mathbf{p}^T(\mathbf{x}_i)\mathbf{a}(\mathbf{x})-u_i] &= 0 \\
&\dots \\
\sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) 2\mathbf{p}_m(\mathbf{x}_i) [\mathbf{p}^T(\mathbf{x}_i)\mathbf{a}(\mathbf{x})-u_i] &= 0
\end{aligned}$$

After rearrangements the above equations become:

$$\sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) \mathbf{p}(\mathbf{x}_i) \mathbf{p}^T(\mathbf{x}_i) \mathbf{a}(\mathbf{x}) = \sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) \mathbf{p}(\mathbf{x}_i) u_i \quad (2.12)$$

A more compact form is:

$$\mathbf{A}(\mathbf{x}) \mathbf{a}(\mathbf{x}) = \mathbf{B}(\mathbf{x}) \mathbf{u} \quad (2.13)$$

where

$$\mathbf{A}(\mathbf{x}) = \sum_{i=1}^n w(\mathbf{x}-\mathbf{x}_i) \mathbf{p}(\mathbf{x}_i) \mathbf{p}^T(\mathbf{x}_i) \quad (2.14)$$

$$\mathbf{B}(\mathbf{x}) = w(\mathbf{x}-\mathbf{x}_1) \mathbf{p}(\mathbf{x}_1) w(\mathbf{x}-\mathbf{x}_2) \mathbf{p}(\mathbf{x}_2) \cdots w(\mathbf{x}-\mathbf{x}_n) \mathbf{p}(\mathbf{x}_n) \quad (2.15)$$

Solving $\mathbf{a}(\mathbf{x})$ from eq.(2.13) and substituting into eq.(2.6) the MLS approximants can be defined as:

$$u^h(\mathbf{x}) = \mathbf{p}^T(\mathbf{x}) [\mathbf{A}(\mathbf{x})^{-1} \mathbf{B}(\mathbf{x}) \mathbf{u}] \quad (2.16)$$

Recalling the form of the approximation defined in eq.(2.1)

$$u(\mathbf{x}, t) = \sum_{i \in S} \Phi_i(\mathbf{x}) u_i(t) \quad (2.17)$$

we can derive the MLS shape functions as:

$$\Phi(\mathbf{x}) = \mathbf{p}^T(\mathbf{x}) [\mathbf{A}(\mathbf{x})]^{-1} \mathbf{B}(\mathbf{x}) \quad (2.18)$$

The shape function Φ_i associated with node i at point \mathbf{x} is given by:

$$\Phi_i(\mathbf{x}) = \mathbf{p}^T(\mathbf{x}) [\mathbf{A}(\mathbf{x})]^{-1} w(\mathbf{x}-\mathbf{x}_i) \mathbf{p}(\mathbf{x}_i) \quad (2.19)$$

Matrix $\mathbf{A}(\mathbf{x})$ is the moment matrix of size $m \times m$. This matrix must be inverted whenever the MLS shape functions are to be evaluated. This fact is one of the major drawbacks of MLS-based MMs because of the computational cost involved and the possibility that this moment matrix is ill conditioned. Considering a linear basis in one dimension, the moment matrix becomes:

$$\mathbf{A}(\mathbf{x}) = w(\mathbf{x} - \mathbf{x}_1) \begin{bmatrix} 1 & x_1 \\ x_1 & x_1^2 \end{bmatrix} + w(\mathbf{x} - \mathbf{x}_2) \begin{bmatrix} 1 & x_2 \\ x_2 & x_2^2 \end{bmatrix} + \dots + w(\mathbf{x} - \mathbf{x}_n) \begin{bmatrix} 1 & x_n \\ x_n & x_n^2 \end{bmatrix} \quad (2.20)$$

If $n=1$, i.e. point \mathbf{x} is covered by only one nodal support while the basis is linear ($m=2$), matrix $\mathbf{A}(\mathbf{x})$ becomes singular and cannot be inverted. Therefore, it is necessary to have $n \geq m$. We should note also that if $n=m$, the nodes have to be arranged in different coordinate directions (not aligned) otherwise the matrix will still be singular.

For typical 2D and 3D problems the linear (eq. 2.9) or quadratic (eq. 2.10) basis is used.

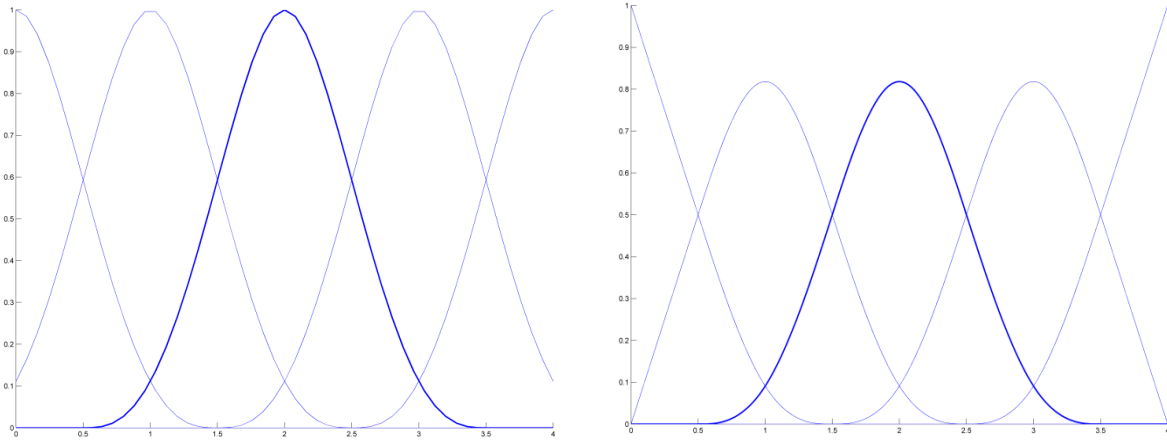


Fig. 2.1: Weight functions

2.1.1.4 Galerkin weak form

The trial and test functions are given by

$$u^h(\mathbf{x}) = \sum_{i=1}^N \Phi_i(\mathbf{x}) u_i \quad (2.21)$$

$$\delta u^h(\mathbf{x}) = \sum_{i=1}^N \Psi_i(\mathbf{x}) \delta u_i \quad (2.22)$$

Usually the same functions are used for the approximation of the test and trial functions ($\Phi_i = \Psi_i$).

The discrete equations are produced by considering a domain Ω , bounded by Γ which is partitioned in two sets Γ_u and Γ_t . Displacements are prescribed on Γ_u whereas tractions are prescribed on Γ_t . The weak form of linear elastostatics is to find \mathbf{u} in the trial space (contains C^0 functions), such that for all test functions $\delta \mathbf{u}$ in the test space (contains C^0 functions but vanishes on Γ_u) the following equation is satisfied:

$$\Pi(\mathbf{u}) = \min \quad (2.23)$$

$$\int_{\Omega} \varepsilon(\mathbf{u}) : \mathbf{C} : \varepsilon(\delta \mathbf{u}) d\Omega = \int_{\Gamma} \bar{\mathbf{t}} \cdot \delta \mathbf{u} d\Gamma + \int_{\Omega} \mathbf{b} \cdot \mathbf{v} d\Omega \quad (2.24)$$

Substitution of approximations for \mathbf{u} and $\delta \mathbf{u}$ into the above equations gives the discrete algebraic equations yields:

$$\mathbf{K} \mathbf{u} = \mathbf{f} \quad (2.25)$$

with

$$\mathbf{K}_{ij} = \int_{\Omega} \mathbf{B}_i^T \mathbf{C} \mathbf{B}_j d\Omega \quad (2.26)$$

$$\mathbf{f}_i = \int_{\Gamma_t} \Phi_i \bar{\mathbf{t}} d\Gamma + \int_{\Omega} \Phi_i \mathbf{b} d\Omega \quad (2.27)$$

In 2D problems matrix \mathbf{B} is given by:

$$\mathbf{B}_I = \begin{bmatrix} \Phi_{i,x} & 0 \\ 0 & \Phi_{i,y} \\ \Phi_{i,y} & \Phi_{i,x} \end{bmatrix} \quad (2.28)$$

and in 3D problems by:

$$\mathbf{B}_i = \begin{bmatrix} \Phi_{i,x} & 0 & 0 \\ 0 & \Phi_{i,y} & 0 \\ 0 & 0 & \Phi_{i,z} \\ 0 & \Phi_{i,z} & \Phi_{i,y} \\ \Phi_{i,z} & 0 & \Phi_{i,x} \\ \Phi_{i,y} & \Phi_{i,x} & 0 \end{bmatrix} \quad (2.29)$$

2.1.1.5 Essential Boundary Conditions

Due to the lack of the Kronecker delta property of shape functions, the essential boundary conditions cannot be imposed the same way as in FEM. Among the several available techniques (Lagrange multipliers, penalty factors, Niche's method) we use in our implementation penalty factors α . The functional that we have for our problem is

$$\bar{\Pi}(\mathbf{u}, \alpha) = \Pi(\mathbf{u}) + \frac{\alpha}{2} \int_{\Gamma} \mathbf{C}(\mathbf{u})^T \mathbf{C}(\mathbf{u}) d\Gamma \quad (2.30)$$

By applying the penalty factor method to elastostatics, the following weak form is obtained:

$$\int_{\Omega} \boldsymbol{\varepsilon}^T(\mathbf{u}) : \mathbf{C} : \boldsymbol{\varepsilon}(\mathbf{v}) d\Omega = \int_{\Gamma} \bar{\mathbf{t}} \cdot \mathbf{v} d\Gamma + \int_{\Omega} \mathbf{b} \cdot \mathbf{v} d\Omega + a \int_{\Gamma_u} \mathbf{u} \cdot \mathbf{v} d\Gamma - a \int_{\Gamma_u} \bar{\mathbf{u}} \cdot \mathbf{v} d\Gamma \quad (2.31)$$

which gives the equation $\mathbf{K} \mathbf{u} = \mathbf{f}$, where

$$\mathbf{K}_{ij} = \int_{\Omega} \mathbf{B}_i^T \mathbf{C} \mathbf{B}_j d\Omega - a \int_{\Gamma_u} \Phi_i \Phi_j d\Gamma \quad (2.32)$$

$$\mathbf{f}_i = \int_{\Gamma_t} \Phi_i \bar{\mathbf{t}} d\Gamma + \int_{\Omega} \Phi_i \mathbf{b} d\Omega - a \int_{\Gamma_u} \Phi_i \bar{\mathbf{u}} d\Gamma \quad (2.33)$$

The accuracy of the penalty method depends on the choice of the penalty parameter α . Although no additional unknowns are required, constraints are satisfied approximately and the accuracy of the solution is dependent on the value of α .

For the integration of eq. (2.26/2.32), virtual background cells are considered by dividing the problem domain into integration cells over which a Gaussian quadrature is performed:

$$\int_{\Omega} f(\mathbf{x}) d\Omega = \sum_j f(\xi_j) \omega_{\xi} \det J^{\xi}(\xi) \quad (2.34)$$

where ξ are the local coordinates and $\det J^{\xi}(\xi)$ is the determinant of the Jacobian.

2.1.2 EFG test examples

The examples analyzed in this study are squares (2D) or cubes (3D). Compared to other geometric domains with the same density (with respect to the number of points in a certain area), the selected domains maximize the number of node-Gauss point correlations and node-node interactions. The examples are 2D and 3D linear elasticity problems. The details for the examples are provided in Table 2.2. In all cases, the domain of influence is rectangular with influence range parameter 2.5.

EFG Example	Nodes	dof	Gauss points
2D-1	25,921	51,842	102,400
2D-2	76,125	152,250	300,304
2D-3	126,025	252,050	501,264
3D-1	9,261	27,783	64,000
3D-2	19,683	59,049	140,608
3D-3	35,937	107,811	262,144

Table 2.2: 2D and 3D example details for EFG test examples.

2.2 Isogeometric Analysis

Isogeometric analysis (IGA) was recently introduced by Hughes and co-workers [18] and since then it has attracted a lot of attention for solving boundary value problems as a result of using the same shape functions adopted from CAD community for describing the domain geometry and for building the numerical approximation of the solution.

Despite IGA's promising methodology and superior features [18]–[21] compared with finite element analysis (FEA), the computation of mass, stiffness and advection matrices is more laborious, which increases the cost of IGA in real-world applications. Due to its higher inter-element continuity, IGA produces quite more elements than FEA for the same number of degrees of freedom. This leads to an increase of the number of Gauss points and consequently of the computational cost for assembling the characteristic matrices. This drawback dramatically increases the computational cost in the multivariate domains, especially in 3D analysis.

It has been shown ([19], [20]) that standard element-wise Gauss rules are inefficient, because they do not take precise account of the preserved smoothness at the element boundaries in the case of higher-order NURBS and polynomial B-SPLines, and that the higher the inter-element regularity the fewer the required number of Gauss points per element. However, recently proposed integration rules, although optimal or nearly optimal in terms of the number of function evaluations, are either cumbersome to implement [19] or need special consideration to be given to the boundary elements [20]. In an effort to address the increased effort in the computation of IGA characteristic matrices, collocation methods have been introduced, requiring a minimum number of quadrature points [22], [23].

2.2.1 Basic ingredients of isogeometric analysis methods

2.2.1.1 Non-Uniform Rational B-SPLines (NURBS)

In IGA the exact geometry is always represented - even in the case of very coarse meshes - and thus there is no approximation in that regard. For the implementation of IGA three spaces should be defined: the physical space, the parameter space and the index space. For NURBS shape functions, the parameter space is very important as all calculations take place in this space, while the index space plays an auxiliary role. The input data is drawn from the physical space, which contains the Cartesian coordinates of the control points and their corresponding weights. The number of basis functions is equal to the number of degrees of freedom. The unknowns of the resulting algebraic

equations correspond to the displacements of the control points, while the knots are the boundaries of the corresponding isogeometric elements. In the case of uniform knot vector, knot spans have the same size in the parameter space while in the physical space they can have any size depending on the corresponding control points and shape functions. The discretized NURBS-model is subdivided into patches which are subdomains with the same material and geometry type and consist of a full tensor product grid of elements. In this respect, they are analogous to elements in FEA as the basis functions are interpolatory at its boundaries.

A knot vector is a non-decreasing set of coordinates in the parameter space, written as $\Xi = \{\xi_1, \xi_2, \dots, \xi_{n+p+1}\}$, where $\xi_i \in \mathbb{R}$ is the i^{th} knot, i is the knot index, $i=1,2,\dots,n+p+1$, p is the polynomial order and n is the number of basis functions used to construct the B-Spline curve. The knots partition the parameter space into elements. Element boundaries in the physical space are the projections of knot lines under the B-Spline mapping. Fig. 2.2 illustrates the quadratic C^1 continuous B-Spline basis functions, which are produced by the open uniform knot vector $\Xi = [0,0,0,1,2,3,4,5,6,7,8,9,9,9]$. Control points are shown as circles, while knots as rectangles. The interval $[0,9]$ is a single patch and consists of 9 elements and 11 control points, which correspond to 11 B-Spline basis functions.

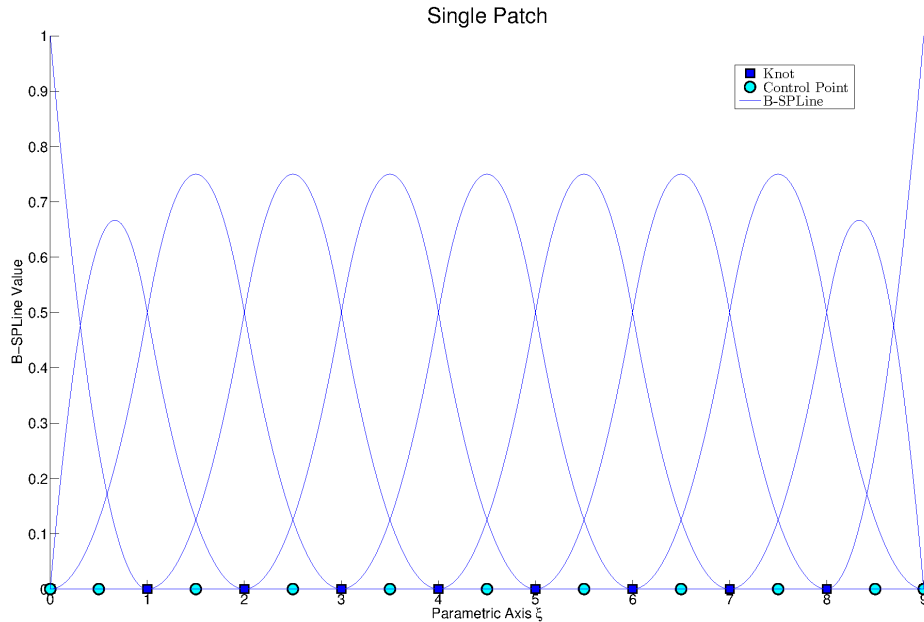


Fig. 2.2: C^1 continuous quadratic basis derived from open uniform knot vector $\Xi = [0,0,0,1,2,3,4,5,6,7,8,9,9,9]$

Given an open uniform knot vector $\Xi = \{\xi_1, \xi_2, \dots, \xi_{n+p+1}\}$, the B-Spline basis functions $N_i^p(\xi)$ are defined by the Cox-de Boor recursion formula:

$$N_i^0(\xi) = \begin{cases} 1, & \text{if } \xi_i \leq \xi < \xi_{i+1} \\ 0, & \text{otherwise} \end{cases} \quad (2.35)$$

$$N_i^p(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_i^{p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1}^{p-1}(\xi) \quad (2.36)$$

Due to their higher regularity between inter-element boundaries, they exhibit greater overlapping in comparison with the shape functions of FEA. Their basic feature is their tensor product nature. In the case of polynomial B-Splines, basis functions are used as shape functions, while in the case of NURBS, shape functions are produced from the following formulas:

$$\text{1D} \quad R_i^p(\xi) = \frac{N_i^p(\xi) W_i}{\sum_{i=1}^n N_i^p(\xi) W_i} \quad (2.37)$$

$$\text{2D} \quad R_{i,j}^{p,q}(\xi, \eta) = \frac{N_i^p(\xi) M_j^q(\eta) W_{i,j}}{\sum_{i=1}^n \sum_{j=1}^m N_i^p(\xi) M_j^q(\eta) W_{i,j}} \quad (2.38)$$

$$\text{3D} \quad R_{i,j,k}^{p,q,r}(\xi, \eta, \zeta) = \frac{N_i^p(\xi) M_j^q(\eta) L_k^r(\zeta) W_{i,j,k}}{\sum_{i=1}^n \sum_{j=1}^m \sum_{k=1}^l N_i^p(\xi) M_j^q(\eta) L_k^r(\zeta) W_{i,j,k}} \quad (2.39)$$

where W denotes weight factors with a full tensor product nature:

$$\text{2D} \quad W_{i,j} = W_i W_j \quad (2.40)$$

$$\text{3D} \quad W_{i,j,k} = W_i W_j W_k \quad (2.41)$$

The approximation of 1D displacement field in terms of control point variables can be written as

$$u(\xi) = \sum_{i=1}^n \{ R_i^p(\xi) u_{CPi} \} \quad (2.42)$$

where $R_i^p(\xi)$ are the shape functions, n is the number of basis functions or control points, p is the polynomial order and u_{CPi} is the displacement of control point i . The exact geometry is described by

$$X(\xi) = \sum_{i=1}^n \{R_i^p(\xi) X_{C_{Pi}}\} \quad (2.43)$$

where $X_{C_{Pi}}$ are the Cartesian coordinate(s) of the control point i .

There is a connection between polynomial basis order p , knot multiplicity m and continuity/regularity k , given by

$$k = p - m, \quad 1 \leq m \leq p + 1 \quad (2.44)$$

Regularity -1 indicates discontinuity and it appears for the extreme knots of a single patch. In this case, basis functions are interpolatory at these extreme knots. Regularity 0 resembles the case of finite elements and is the minimum continuity for interior knots with basis functions interpolatory at those knots. The case of maximum continuity is $p - 1$ and occurs when every interior knot is repeated only once.

In one-dimensional analysis with polynomial order p , multiplicity m and number of elements n^{el} , the corresponding number of control points n , which are directly linked to the number of degrees of freedom, is equal to

$$n = (p + 1)n^{el} - (k - 1)(n^{el} - 1) \quad (2.45)$$

The corresponding knot vector has $n + p + 1$ knot values. The external knots are repeated $p + 1$ times and the interior knots m times.

2.2.1.2 Stiffness matrix formulation

A given domain is represented with several NURBS-based isogeometric models, depending on its geometry features. Every NURBS-based model is decomposed into subdomains, called patches, according to the variance of its geometry and material. The more abrupt the geometry is, the more subdomains are considered. They can be assumed as macro-elements consisting of a tensor product mesh of elements and they are assembled in the same way as in finite elements. The arrays for the patches are constructed and assembled in element-by-element fashion by numerically integrating contributions over each element. In the parameter space, elements are rectangular.

The equilibrium equations applied to control points of the whole domain are expressed as

$$\mathbf{K} \mathbf{u} = \mathbf{f} \quad (2.46)$$

In order to formulate the domain/global stiffness matrix, the stiffness matrix of every patch $i=1, \dots, N_p$ has to be calculated:

$$\mathbf{K}^i = \int_V (\mathbf{B}^i)^T \mathbf{E}^i \mathbf{B}^i dV = \iiint_{\xi, \eta, \zeta} (\mathbf{B}^i)^T \mathbf{E}^i \mathbf{B}^i \det \mathbf{J}^i d\xi d\eta d\zeta \quad (2.47)$$

where \mathbf{E}^i , \mathbf{B}^i are the elasticity and deformation matrix of the patch i respectively.

The stiffness matrix formulation in 2D elasticity cases is presented below. For the 3D case, the formulation is analogous. Assuming n , m control points per parametric axis ξ , η respectively, the 2D control points are $N = nm$ (full tensor product) and the deformation matrix \mathbf{B} is given by:

$$\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \quad (2.48)$$

$(3 \times N) \quad (3 \times 4)(4 \times N)$

with

$$\mathbf{B}_1 = \frac{1}{\det \mathbf{J}(\xi)} \begin{bmatrix} J_{22} & -J_{12} & 0 & 0 \\ 0 & 0 & -J_{21} & J_{11} \\ -J_{21} & J_{11} & J_{22} & -J_{12} \end{bmatrix} \quad (2.49)$$

and

$$\mathbf{B}_2 = \begin{bmatrix} R_{1,\xi} & 0 & R_{2,\xi} & 0 & \dots & R_{N,\xi} & 0 \\ R_{1,\eta} & 0 & R_{2,\eta} & 0 & \dots & R_{N,\eta} & 0 \\ 0 & R_{1,\xi} & 0 & R_{2,\xi} & 0 & \dots & R_{N,\xi} \\ 0 & R_{1,\eta} & 0 & R_{2,\eta} & 0 & \dots & R_{N,\eta} \end{bmatrix} \quad (2.50)$$

The Jacobian matrix is local to patches rather than to elements and is given by

$$\mathbf{J}(\xi, \eta) = \underbrace{\begin{bmatrix} R_{1,\xi}(\xi, \eta) & R_{2,\xi}(\xi, \eta) & \dots & R_{N,\xi}(\xi, \eta) \\ R_{1,\eta}(\xi, \eta) & R_{2,\eta}(\xi, \eta) & \dots & R_{N,\eta}(\xi, \eta) \end{bmatrix}}_{(2 \times N)} \begin{bmatrix} X_{CP1} & Y_{CP1} \\ X_{CP2} & Y_{CP2} \\ \vdots & \vdots \\ X_{CPN} & Y_{CPN} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix}_{(2 \times 2)} \quad (2.51)$$

where $R_l(\xi, \eta)$ is the shape function that corresponds to the control point l , with Cartesian coordinates X_{CPl} , Y_{CPl} , and

$$R_{l,\xi}(\xi, \eta) = \frac{dR_l(\xi, \eta)}{d\xi}, \quad R_{l,\eta}(\xi, \eta) = \frac{dR_l(\xi, \eta)}{d\eta} \quad (2.52)$$

The derivatives in eq. (2.52) are obtained by applying the quotient rule to eq. (2.38):

$$\begin{aligned}\frac{dR_{i,j}^{p,q}(\xi, \eta)}{d\xi} &= \frac{\frac{dN_i^p(\xi)}{d\xi} M_j^q(\eta) W_{i,j} W(\xi, \eta) - N_i^p(\xi) M_j^q(\eta) W_{i,j} \frac{dW(\xi, \eta)}{d\xi}}{(W(\xi, \eta))^2} \\ \frac{dR_{i,j}^{p,q}(\xi, \eta)}{d\eta} &= \frac{N_i^p(\xi) \frac{dM_j^q(\eta)}{d\eta} W_{i,j} W(\xi, \eta) - N_i^p(\xi) M_j^q(\eta) W_{i,j} \frac{dW(\xi, \eta)}{d\eta}}{(W(\xi, \eta))^2}\end{aligned}\tag{2.53}$$

where

$$\begin{aligned}\frac{dW(\xi, \eta)}{d\xi} &= \sum_{i=1}^n \sum_{j=1}^m \left\{ \frac{dN_i^p(\xi)}{d\xi} M_j^q(\eta) W_{i,j} \right\} \\ \frac{dW(\xi, \eta)}{d\eta} &= \sum_{i=1}^n \sum_{j=1}^m \left\{ N_i^p(\xi) \frac{dM_j^q(\eta)}{d\eta} W_{i,j} \right\}\end{aligned}\tag{2.54}$$

2.2.1.3 Quadrature rule

The Gauss quadrature rule is applied to the non-zero knot spans as in FEA. However, the standard element-wise Gauss rule requires extensive function evaluations due to the increased support of the shape functions. According to [20], for the case of a one-dimensional function of order p , the optimal (minimum exact) number of Gauss points per element is equal to $(p+1)/2$ or $(p+2)/2$, for odd and even p , respectively. For the computation of the stiffness matrix in case of 1D elasticity, the integrand's order is equal to $q=2p-2$ and the optimal number of Gauss points per element is equal to $(q+2)/2=p$. In case of 2D and 3D elasticity, the integrand's order is equal to $q=2p$ and the optimal number of Gauss points per element is equal to $(q+2)/2=p+1$. The above rules are optimal for the case of minimum continuity. For higher continuity, new macro-element rules have been proposed [19], [20], which are more efficient but also more involved and difficult to implement.

2.2.2 IGA Test examples

The examples considered use squares or cubes in the parametric space to maximize the correlations for the same number of control points compared to other rectangular domains. The examples are 2D and 3D linear elasticity problems. The IGA variant used in this work utilizes shape functions based on non-uniform rational B-splines (NURBS) and the details for the examples considered are provided in Table 2.3. Example type P_{i-j} corresponds to order $p=i$ of the basis function of the group, while j denotes the relative size of the example within the same group.

IGA Example	p	n	Control points	dof	Elements	Gauss point per element	Gauss points	
2D	P2-1	2	225	50,625	101,250	49,729	9	447,561
	P2-2	2	500	250,000	500,000	248,004	9	2,232,036
	P2-3	2	633	400,689	801,378	398,161	9	3,583,449
	P3-1	3	225	50,625	101,250	49,284	16	788,544
	P3-2	3	320	102,400	204,800	100,489	16	1,607,824
	P3-3	3	388	150,544	301,088	148,225	16	2,371,600
	P4-1	4	160	25,600	51,200	24,336	25	608,400
	P4-2	4	225	50,625	101,250	48,841	25	1,221,025
	P4-3	4	275	75,625	151,250	73,441	25	1,836,025
3D	P2-1	2	19	6,859	20,577	4,913	27	132,651
	P2-2	2	26	17,576	52,728	13,824	27	373,248
	P2-3	2	33	35,937	107,811	29,791	27	804,357
	P3-1	3	19	6,859	20,577	4,096	64	262,144
	P3-2	3	21	9,261	27,783	5,832	64	373,248
	P3-3	3	26	17,576	52,728	12,167	64	778,688
	P4-1	4	15	3,375	10,125	1,331	125	166,375
	P4-2	4	17	4,913	14,739	2,197	125	274,625
	P4-3	4	19	6,859	20,577	3,375	125	421,875

Table 2.3: 2D and 3D example details for IGA test examples.

2.3 Finite element test examples

The performance of the solution phase of FEA implementations is demonstrated through a parametric study of 3D linear elasticity problems ($E=39 \text{ MPa}$, $\nu=0.2$) with a cubic geometry. The domain is fully restrained at the bottom surface, partially restrained on the horizontal directions of the side surfaces while the top surface is subjected to a distributed load, and are discretized with 8-node hexahedral finite elements, resulting in linear systems with 1,058,610 dof. Domain decomposition methods are utilized for the solution so the domain is split into a number of subdomains ranging from 125 to 2744. The subdivision in 125 subdomains is shown in Fig. 2.3. Table 2.4 shows the dof of each subdomain and of the corresponding interface problem for different number of subdomains.

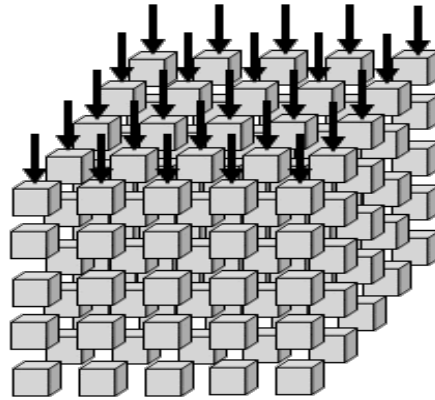


Fig. 2.3. Partitioning of domain in 125 subdomains

Number of subdomains	Subdomain dof	Interface dof
125	10,119	49,920
175	7,419	76,800
245	5,439	118,080
343	3,987	181,440
490	2,898	276,480
700	2,106	421,200
1000	1,530	641,520
1400	1,146	935,280
1960	858	1,363,440
2744	642	1,987,440

Table 2.4: Example details for FEA

3 Domain decomposition methods

Domain decomposition methods (DDM) constitute today an important category of methods for the solution of a variety of problems in simulation based applied science and engineering. The primal DDM (P-DDM) reach the solution by solving for the interface primal variables (usually the displacements) after eliminating the internal degrees of freedom (dof) of the subdomains, while the dual DDM (D-DDM) proceed with the computation of the Lagrange multipliers required to enforce compatibility between subdomains after elimination of all dof (internal and interface) of each subdomain. Both major categories, primal and dual, have gradually attracted the interest of a large number of researchers and have been incorporated into high-performance commercial software codes. The most important family of primal DDM is considered to be the balancing domain decomposition (BDD) method, introduced by Mandel [24], while the FETI method, introduced by Farhat and Roux [25], along with its variants, is considered a highly efficient dual DDM in both sequential as well as in parallel/distributed computing environment. A unified framework for formulating both primal and dual DDM is presented in [26], [27] for implicit static and dynamic computations and a new family of methods, namely the P-FETI methods, were proposed. Since their introduction, non-overlapping DDM have been widely used for solving large-scale problems in a number of fields in computational mechanics. Some recent indicative applications are in contact problems [28], porous media problems [29], heterogeneous problems [30], stochastic finite elements [31], [32], inequality-constrained quadratic programming [33], Navier-Stokes equations [34], Helmholtz problems [35], Galerkin least-squares methods [36].

3.1 The primal domain decomposition implementation

The primary domain decomposition implementation reduces the size of the solution system by utilizing static condensation (see Section 3.1.1). The initial domain system is:

$$\mathbf{K} \mathbf{u} = \mathbf{f} \tag{3.1}$$

If the internal degrees of freedom (dof) of each subdomain are numbered first and the boundary dof last, then the domain system can be written as:

$$\begin{bmatrix} \mathbf{K}_{ii}^{(1)} & \cdots & 0 & \mathbf{K}_{ib}^{(1)} \\ & \mathbf{K}_{ii}^{(2)} & \cdots & \mathbf{K}_{ib}^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \mathbf{K}_{ib}^{(n_s)} \\ \hline \mathbf{K}_{bi}^{(1)} & \mathbf{K}_{bi}^{(2)} & \cdots & \mathbf{K}_{bi}^{(n_s)} & \mathbf{K}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{u}_i^{(1)} \\ \mathbf{u}_i^{(2)} \\ \vdots \\ \mathbf{u}_i^{(n_s)} \\ \mathbf{u}_b \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i^{(1)} \\ \mathbf{f}_i^{(2)} \\ \vdots \\ \mathbf{f}_i^{(n_s)} \\ \mathbf{f}_b \end{bmatrix} \quad (3.2)$$

$$\mathbf{u}_i^s = (\mathbf{K}_{ii}^s)^{-1} \mathbf{f}_i^s - (\mathbf{K}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \mathbf{u}_b \quad (3.3)$$

The boundary stiffness submatrix and force subvector are:

$$\mathbf{K}_{bb} = \sum_{s=1}^{n_s} \mathbf{K}_{bb}^s \quad \mathbf{f}_b = \sum_{s=1}^{n_s} \mathbf{f}_b^s \quad (3.4)$$

In the primal subdomain implementation, all internal dof are condensed into the boundary dof. The static condensation is given by:

$$\underbrace{\left(\mathbf{K}_{bb} - \sum_{s=1}^{n_s} \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \right)}_{\hat{\mathbf{K}}_c \equiv \mathbf{S}} \mathbf{u}_b = \mathbf{f}_b - \underbrace{\sum_{s=1}^{n_s} \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{f}_i^s}_{\hat{\mathbf{f}}_c \equiv \hat{\mathbf{f}}_b} \quad (3.5)$$

$$\mathbf{S} \mathbf{u}_b = \hat{\mathbf{f}}_b \quad (3.6)$$

The matrix \mathbf{S} is commonly referenced as Schur complement.

$$\mathbf{S} = \sum_{s=1}^{n_s} \mathbf{S}^s = \sum_{s=1}^{n_s} (\mathbf{V}_b^s)^T \mathbf{S}^s \mathbf{V}_b^s \quad (3.7)$$

Each subdomain's contribution is given by:

$$\mathbf{S}^s = \mathbf{K}_{bb}^s - \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \quad (3.8)$$

Also, the right hand side vector is:

$$\hat{\mathbf{f}}_b = \mathbf{f}_b - \sum_{s=1}^{n_s} (\mathbf{V}_b^s)^T \mathbf{K}_{bi}^s (\mathbf{K}_{ii}^s)^{-1} \mathbf{f}_i^s \quad (3.9)$$

The Schur complement \mathbf{S} is dense so its formulation is computationally feasible in small problems or when the boundary degrees are few. In such case, eq. (Fig. 3.6) can even be solved directly. However, for large scale problems (especially 3D ones) with an extensive boundary, an iterative solution method is preferred because the formulation of the Schur complement \mathbf{S} can be entirely avoided.

3.1.1 Static condensation

The purpose of static condensation is to eliminate the the chosen e degrees of freedom and condense them in the rest of the degrees of freedom c . After condensation, the c degrees incorporate the effect of the e degrees. The condensed domain has only the c degrees of freedom but they have exactly the same behavior as the c degrees of the original domain meaning that any result on the condensed domain is valid for the original domain as well. After solving the condensed domain for the c degrees of freedom, a de-condensation is performed to calculate the e degrees.

$$\begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ec} \\ \mathbf{K}_{ce} & \mathbf{K}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{u}_e \\ \mathbf{u}_c \end{bmatrix} = \begin{bmatrix} \mathbf{f}_e \\ \mathbf{f}_c \end{bmatrix} \quad (3.10)$$

Solving the first equation of (3.10) for \mathbf{u}_e :

$$\begin{aligned} \mathbf{K}_{ee} \mathbf{u}_e + \mathbf{K}_{ec} \mathbf{u}_c &= \mathbf{f}_e \\ \mathbf{K}_{ee} \mathbf{u}_e &= \mathbf{f}_e - \mathbf{K}_{ec} \mathbf{u}_c \\ \mathbf{u}_e &= \mathbf{K}_{ee}^{-1} (\mathbf{f}_e - \mathbf{K}_{ec} \mathbf{u}_c) \\ \mathbf{u}_e &= \mathbf{K}_{ee}^{-1} \mathbf{f}_e - \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \mathbf{u}_c \end{aligned} \quad (3.11)$$

Substituting \mathbf{u}_e in the second equation of (3.10) and isolating \mathbf{u}_c :

$$\begin{aligned} \mathbf{K}_{ce} \mathbf{u}_e + \mathbf{K}_{cc} \mathbf{u}_c &= \mathbf{f}_c \\ \mathbf{K}_{ce} (\mathbf{K}_{ee}^{-1} \mathbf{f}_e - \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \mathbf{u}_c) + \mathbf{K}_{cc} \mathbf{u}_c &= \mathbf{f}_c \\ \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{f}_e - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \mathbf{u}_c + \mathbf{K}_{cc} \mathbf{u}_c &= \mathbf{f}_c \\ \underbrace{(\mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec})}_{\hat{\mathbf{K}}_c} \mathbf{u}_c &= \underbrace{\mathbf{f}_c - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{f}_e}_{\hat{\mathbf{f}}_c} \end{aligned} \quad (3.12)$$

$$\hat{\mathbf{K}}_c \mathbf{u}_c = \hat{\mathbf{f}}_c \quad (3.13)$$

where

$$\hat{\mathbf{K}}_c = \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \quad (3.14)$$

$$\hat{\mathbf{f}}_c = \mathbf{f}_c - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{f}_e \quad (3.15)$$

The condensed system refers to the c degrees only. However, while the stiffness matrix and force vector is modified for the condensed system, the \mathbf{u}_c vector has no modification which means that the solution of the condensed domain of eq. (3.13) is also the solution of the c degrees of freedom of the original domain. If the matrix is factorized, the condensed form is calculated directly from the

factorized matrix. This is shown for $\mathbf{L}\mathbf{L}^T$ and $\mathbf{L}\mathbf{D}\mathbf{L}^T$ decomposition in the next two sub-sections.

3.1.1.1 LL decomposition

$$\begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ec} \\ \mathbf{K}_{ce} & \mathbf{K}_{cc} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{ee} & \mathbf{0} \\ \mathbf{L}_{ce} & \mathbf{L}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{ee} & \mathbf{0} \\ \mathbf{L}_{ce} & \mathbf{L}_{cc} \end{bmatrix}^T \quad (3.16)$$

$$\begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ec} \\ \mathbf{K}_{ce} & \mathbf{K}_{cc} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{ee} & \mathbf{0} \\ \mathbf{L}_{ce} & \mathbf{L}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{ee}^T & \mathbf{L}_{cc}^T \\ \mathbf{0} & \mathbf{L}_{cc}^T \end{bmatrix}^T \quad (3.17)$$

Taking the equations of (3.17) separately:

$$\mathbf{K}_{ee} = \mathbf{L}_{ee} \mathbf{L}_{ee}^T \Rightarrow \mathbf{K}_{ee}^{-1} = \mathbf{L}_{ee}^{-T} \mathbf{L}_{ee}^{-1} \quad (3.18)$$

$$\mathbf{K}_{ce} = \mathbf{L}_{ce} \mathbf{L}_{ee}^T \Rightarrow \mathbf{L}_{ce} = \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \quad (3.19)$$

Substituting eq. (3.18) in $\hat{\mathbf{K}}_c$:

$$\begin{aligned} \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \\ \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{L}_{ee}^{-1} \mathbf{K}_{ec} \\ \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} (\mathbf{K}_{ce} \mathbf{L}_{ee}^{-T})^T \\ \text{due to (3.19)} \quad \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{L}_{ce} \mathbf{L}_{ce}^T \end{aligned} \quad (3.20)$$

Similarly, for $\hat{\mathbf{f}}_c$:

$$\begin{aligned} \hat{\mathbf{f}}_c &= \mathbf{f}_c - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{f}_e \\ \hat{\mathbf{f}}_c &= \mathbf{f}_c - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{L}_{ee}^{-1} \mathbf{f}_e \\ \text{due to (3.19)} \quad \hat{\mathbf{f}}_c &= \mathbf{f}_c - \mathbf{L}_{ce} \mathbf{L}_{ee}^{-1} \mathbf{f}_e \end{aligned} \quad (3.21)$$

for $j = N_{e+1}, \dots, N$

for $i = N_{e+1}, \dots, j$

$$\left(\hat{\mathbf{K}}_c \right)_{ij} = \left(\mathbf{K}_{cc} \right)_{ij} - \sum_{k=m}^{N_e} l_{ki} l_{kj}$$

end

end

3.1.1.2 LDL decomposition

$$\begin{bmatrix} \mathbf{K}_{ee} & \mathbf{K}_{ec} \\ \mathbf{K}_{ce} & \mathbf{K}_{cc} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{ee} & \\ & \mathbf{L}_{ce} \end{bmatrix} \begin{bmatrix} \mathbf{D}_{ee} & \\ & \mathbf{D}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{ee}^T & \mathbf{L}_{ec}^T \\ & \mathbf{L}_{cc}^T \end{bmatrix} \quad (3.22)$$

Taking the equations of (3.22) separately:

$$\mathbf{K}_{ee} = \mathbf{L}_{ee} \mathbf{D}_{ee} \mathbf{L}_{ee}^T \Rightarrow \mathbf{K}_{ee}^{-1} = \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1} \mathbf{L}_{ee}^{-1} \quad (3.23)$$

$$\mathbf{K}_{ce} = \mathbf{L}_{ce} \mathbf{D}_{ee} \mathbf{L}_{ee}^T \Rightarrow \mathbf{L}_{ce} = \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1} \quad (3.24)$$

Substituting eq. (3.23) in $\hat{\mathbf{K}}_c$:

$$\begin{aligned} \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{K}_{ec} \\ \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1} \mathbf{L}_{ee}^{-1} \mathbf{K}_{ec} \\ \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1} \underbrace{\mathbf{D}_{ee} \mathbf{D}_{ee}^{-1}}_{\mathbf{I}} \mathbf{L}_{ee}^{-1} \mathbf{K}_{ec} \\ \hat{\mathbf{K}}_c &= \mathbf{K}_{cc} - (\mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1}) \mathbf{D}_{ee} (\mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1})^T \end{aligned}$$

due to (3.24)

$$\hat{\mathbf{K}}_c = \mathbf{K}_{cc} - \mathbf{L}_{ce} \mathbf{D}_{ee} \mathbf{L}_{ce}^T \quad (3.25)$$

Similarly, for $\hat{\mathbf{f}}_c$:

$$\begin{aligned} \hat{\mathbf{f}}_c &= \mathbf{f}_c - \mathbf{K}_{ce} \mathbf{K}_{ee}^{-1} \mathbf{f}_e \\ \hat{\mathbf{f}}_c &= \mathbf{f}_c - \mathbf{K}_{ce} \mathbf{L}_{ee}^{-T} \mathbf{D}_{ee}^{-1} \mathbf{L}_{ee}^{-1} \mathbf{f}_e \end{aligned}$$

due to (3.24)

$$\hat{\mathbf{f}}_c = \mathbf{f}_c - \mathbf{L}_{ce} \mathbf{L}_{ee}^{-1} \mathbf{f}_e \quad (3.26)$$

(same as eq. 3.21).

for $j = N_{e+1}, \dots, N$

for $i = N_{e+1}, \dots, j$

$$(\hat{\mathbf{K}}_c)_{ij} = (K_{cc})_{ij} - \sum_{k=m}^{N_e} l_{ki} l_{kj} d_{kk}$$

end

end

3.2 The dual domain decomposition implementation

In the dual domain decomposition implementation we form and solve the subdomain interface problem in which the unknowns are Lagrange Multipliers that represent the boundary forces between different subdomains. This method is usually referenced as finite element tearing and interconnecting (FETI) and it was initially introduced in [25]. It has been proven to be particularly efficient for solving large scale problem in parallel clusters with distributed or shared memory [37]–[39]. Applications of the FETI method include a variety of fields in computational mechanics, like linear and non-linear finite element analysis, static and dynamic problems, stochastic finite elements as well as shape optimization and topology optimization problems [40]–[42].

3.2.1 FETI ingredients

The domain is teared in completely independent subdomains, which connect to each other with boundary forces that maintain the continuity of the domain. These forces, which are the unknowns of the interface problem, correspond to the Lagrange multipliers of the optimization problem subject to limitations that is formed on the subdomain boundary.

The solution of the resulting system of equations requires special handling due to the existence of zero elements along the diagonal. Hence, the system cannot be solved by a common iterative method like PCG, but with a modified iterative algorithm called projected preconditioned conjugate gradient (PCPG). Furthermore, FETI takes into account the presence of floating subdomains – subdomains whose stiffness matrix cannot be inversed because of inadequate support.

The domain of Fig. 3.1 comprises 18 quadrilateral finite elements and is supported on the left side. The domain node numbering is shown in Fig. 3.2. The domain's equilibrium equations are:

$$\mathbf{K} \mathbf{u} = \mathbf{f} \quad (3.27)$$

The domain is teared in two subdomains. The interface nodes are both on the left subdomain as well as the right one (Fig. 3.4).

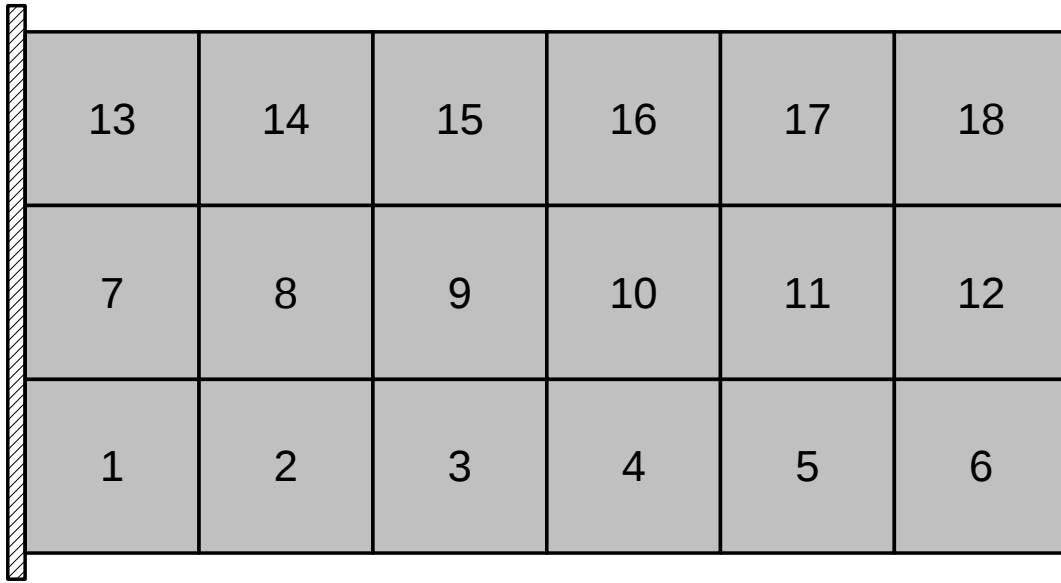


Fig. 3.1: Example domain

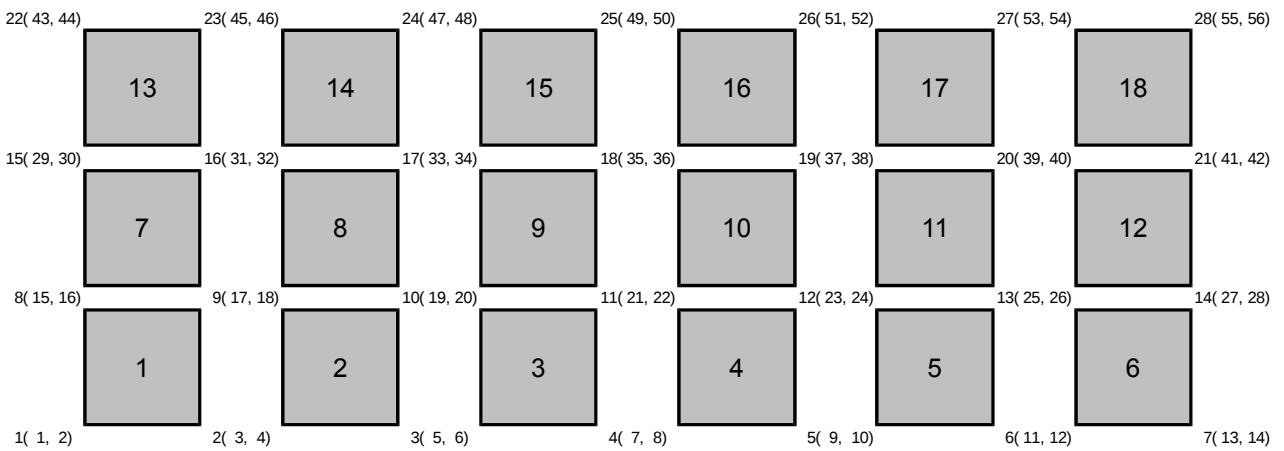


Fig. 3.2: Numbering of nodes and degrees of freedom of the domain

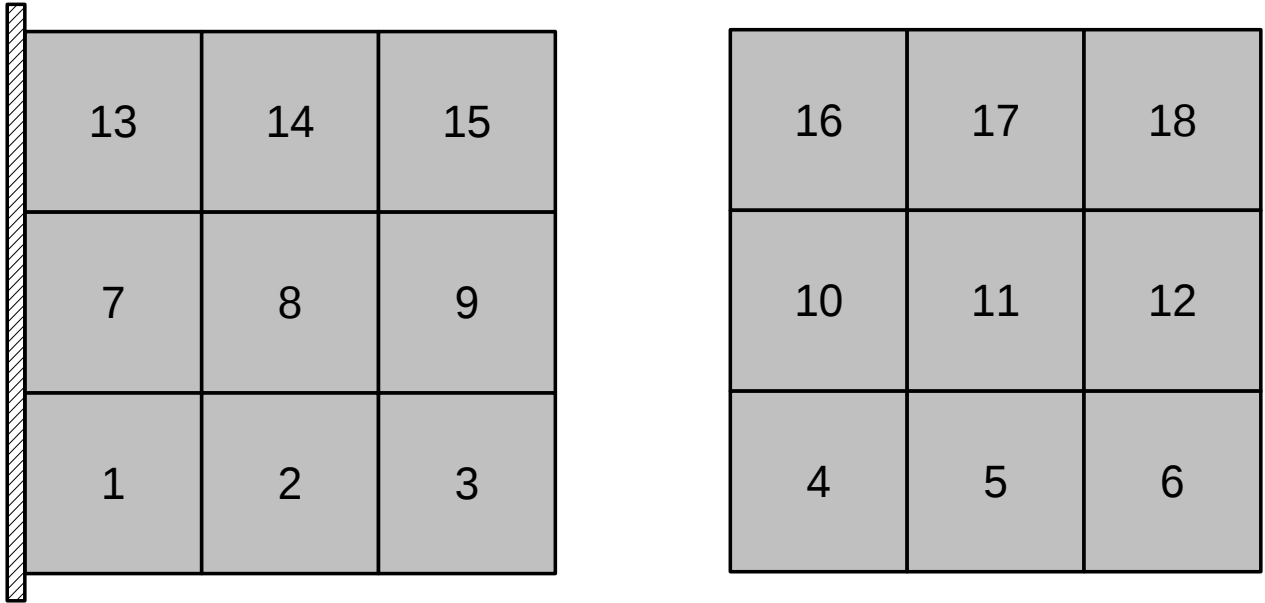


Fig. 3.3: Domain teared into two subdomains

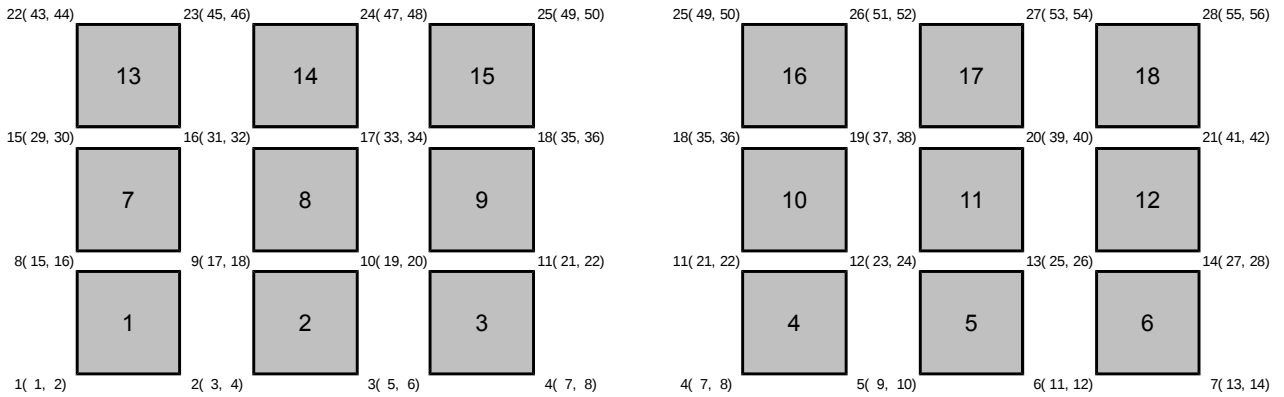


Fig. 3.4: Numbering of nodes and degrees of freedom of the teared domain

The equilibrium equation of the teared domain are:

$$\begin{bmatrix} \mathbf{K}^{(1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}^{(2)} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \end{bmatrix} \quad (3.28)$$

More generally, for any number of subdomains:

$$\mathbf{K}^{(TearedDomain)} \mathbf{u}^{(TearedDomain)} = \mathbf{f}^{(TearedDomain)} \quad (3.29)$$

where $\mathbf{K}^{(TearedDomain)}$ is a block diagonal matrix where each block is the stiffness matrix of a

subdomain and the vectors contain the vector of the subdomains in consecutive order. The matrix is block diagonal because the subdomains are independent so their interaction is zero. However, there are several constraints that must be introduced in the boundary of the subdomains in order to maintain continuity and make the problem equivalent to the original one. The actual equilibrium equations are derived by minimizing:

$$\Pi = \frac{1}{2} \left((\mathbf{u}^g)^T \mathbf{K}^g \mathbf{u}^g \right) - (\mathbf{f}^g)^T \mathbf{u}^g, \quad g \equiv \text{TearedDomain} \quad (3.30)$$

The boundary constraints are added to the equation through the use of Lagrange multipliers:

$$L(\mathbf{u}, \boldsymbol{\lambda}) = \frac{1}{2} \left((\mathbf{u}^g)^T \mathbf{K}^g \mathbf{u}^g \right) - (\mathbf{f}^g)^T \mathbf{u}^g + \boldsymbol{\lambda}^T \cdot [\text{constraints}], \quad g \equiv \text{TearedDomain} \quad (3.31)$$

In FETI, the Lagrange multipliers are subdomain boundary forces. Before moving on, the current numbering needs to be changed. The reason for this is that at the boundary the same nodes appear on all interconnected subdomains. Each node instance will be numbered separately, even through in essence it is the same node and all instances have all characteristics in common (e.g. coordinates, displacement etc). The number of instances of a node is called multiplicity of the node and the constraints must preserve the equality of these instances.

The teared domain has more degrees of freedom (dof) that the original one. Hereinafter, each instance of the node is treated as a separate node. The same applies to the dof of the node. The new numbering is shown in Fig. 3.5.

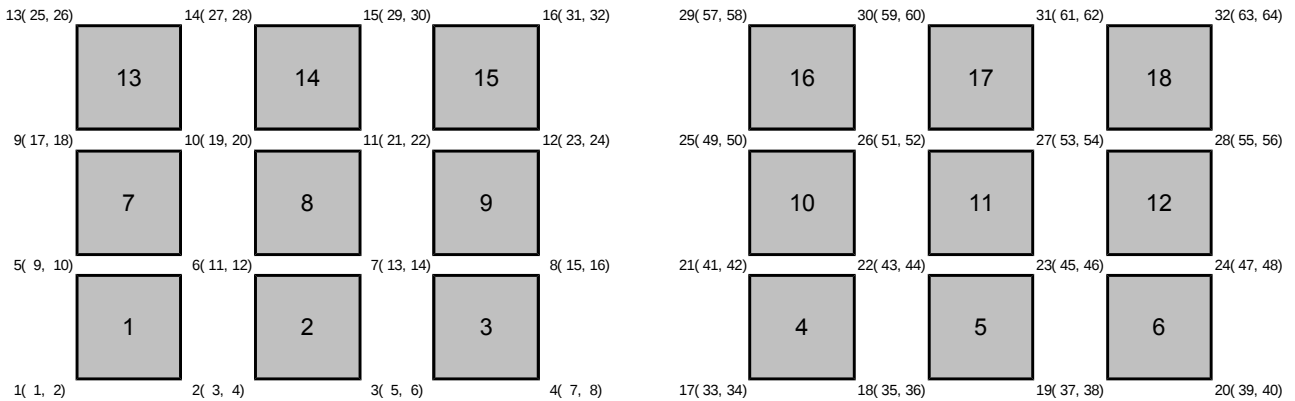


Fig. 3.5: Global numbering of nodes and degrees of freedom of the subdomains

In order to maintain domain continuity, boundary node displacements must be equal in both subdomains:

$$\mathbf{u}_{boundary}^{(1)} = \mathbf{u}_{boundary}^{(2)} \quad (3.32)$$

This simply states that the same node cannot have different displacement in its various instances. If this is not true, then the domain is not continuous. In our example, there are 4 boundary nodes, all of which have 2 instances, one in each subdomain. Hence, the boundary dof are 8 in total. The domain-scope and subdomain-scope numbering for the interface nodes is shown in Table 3.1. Each node has only one domain scope instance, but may have multiple subdomain-scope instances.

Domain scope	Subdomain scope	
4 (7,8)	4 (7,8)	17 (33,34)
11 (21,22)	8 (15,16)	21 (41,42)
18 (35,36)	12 (23,24)	25 (49,50)
25 (49,50)	16 (31,32)	29 (57,58)

Table 3.1: Numbering of interface nodes

Let u_i be the displacement of dof i . The displacement of each instance must be the same and the equality can be written in matrix form:

$$u_i^{(1)} = u_i^{(2)} \quad (3.33)$$

$$u_i^{(1)} - u_i^{(2)} = 0 \quad (3.34)$$

$$\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} u_i^{(1)} \\ u_i^{(2)} \end{bmatrix} = 0 \quad (3.35)$$

The general form of a polynomial with degree n is:

$$a_1 u_1 + a_2 u_2 + \dots + a_n u_n = 0 \quad (3.36)$$

For the case at hand, n will be the number of dof of the teared subdomain. For each interface dof i , this polynomial will have all coefficients equal to zero, except two. One of the non-zero coefficients will be $+1$ and the other will be -1 . For example, dof 24 is connected to dof 50. Hence:

$$0 \cdot u_1 + 0 \cdot u_2 + \dots + 1 \cdot u_{24} + \dots + (-1) \cdot u_{50} + \dots + 0 \cdot u_{63} + 0 \cdot u_{64} = 0 \quad (3.37)$$

$$\begin{bmatrix} 0 & 0 & \dots & 1 & \dots & -1 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{24} \\ \vdots \\ u_{50} \\ \vdots \\ u_{63} \\ u_{64} \end{bmatrix} = 0$$

$$\Leftrightarrow \begin{bmatrix} 0 & 0 & \dots & 1 & \dots & -1 & \dots & 0 & 0 \end{bmatrix} \mathbf{u}^{(TearedDomain)} = 0$$

where the $+1$ coefficient is on the 24th entry and the -1 is on the 50th entry.

By repeating this for every dof of the teared domain:

$$\mathbf{B}^{(TearedDomain)} \mathbf{u}^{(TearedDomain)} = \mathbf{0} \quad (3.38)$$

where \mathbf{B} is the signed Boolean matrix of the teared domain. Each dof of the teared domain is represented by a column of \mathbf{B} , where each line represents an equation-limitation between dof. There are two dof types in this regard:

1. If a dof of the original domain is not part of the interface, then it will only have one instance and will belong only to one subdomain. In this case, the whole corresponding column of this dof will be composed of zeros.
2. If a dof of the original domain is part of the interface, then it will have more than one instances and will belong to equal number of subdomains. In the example of Fig. 3.3, all boundary dofs have two instances each. Therefore, there is a single equation between them and consequently the number of equations is equal to the number of boundary dofs, hence 8.

Each row will have $+1$ on the column corresponding to the dofs that belongs to the first subdomain and -1 on the column corresponding to the dof that belongs to the second subdomain. The relevant numbering here is the subdomain-scope numbering shown in Fig. 3.5. All other entries of this row will be 0 .

Therefore, the signed Boolean matrix \mathbf{B} of the example shown in Fig. 3.3 will have:

- A number of rows equal to the required limitations. In this case the limitations are exactly equal to the boundary dofs, hence 8 (this is not always the case as explained later).
- A number of columns equal to the total dofs of the expanded domain, hence 64.

Therefore, the size of the matrix is $[8 \times 64]$. Each line only has a $+1$ and a -1 on the appropriate columns, and the rest of the row is filled with zeros. Each line is essentially an equation of the form shown in eq. (3.34). The signed Boolean matrix \mathbf{B} packs all limitations in a single matrix (eq. 3.38).

From the teared domain's signed Boolean matrix \mathbf{B} , it is easy to extract the signed Boolean matrices of the subdomains. For each one, all rows are included but only the columns belonging to the subdomain are selected. In the example, the first 32 columns belong to the first subdomain while the last 32 columns belong to the second subdomain. Consequently, the signed Boolean matrices of the two subdomains are each $[8 \times 32]$.

In subdomain terms, eq. (3.38) transforms as follows:

$$\mathbf{B}^{(TearedDomain)} \mathbf{u}^{(TearedDomain)} = \mathbf{0}$$

$$\mathbf{B}^{(1)} \mathbf{u}^{(1)} + \mathbf{B}^{(2)} \mathbf{u}^{(2)} = \mathbf{0}$$

and generally, for an arbitrary number of subdomains n_s :

$$\sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0} \quad (3.39)$$

These are the constraints that need to be applied in the Lagrange eq. (3.31) to maintain the continuity of the domain:

$$L(\mathbf{u}, \boldsymbol{\lambda}) = \frac{1}{2} \left((\mathbf{u}^g)^T \mathbf{K}^g \mathbf{u}^g \right) - (\mathbf{f}^g)^T \mathbf{u}^g + \boldsymbol{\lambda}^T \cdot [\text{constraints}], \quad g \equiv \text{TearedDomain}$$

$$L(\mathbf{u}, \boldsymbol{\lambda}) = \frac{1}{2} \left((\mathbf{u}^g)^T \mathbf{K}^g \mathbf{u}^g \right) - (\mathbf{f}^g)^T \mathbf{u}^g + \boldsymbol{\lambda}^T \sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s \quad (3.40)$$

- Differentiating with respect to \mathbf{u} and equating with zero yields the equilibrium equations:

$$\frac{\partial L(\mathbf{u}, \boldsymbol{\lambda})}{\partial \mathbf{u}} = \mathbf{K}^g \mathbf{u}^g - \mathbf{f}^g + \sum_{s=1}^{n_s} (\mathbf{B}^s)^T \boldsymbol{\lambda} = \mathbf{0}$$

$$\mathbf{K}^g \mathbf{u}^g = \mathbf{f}^g - \sum_{s=1}^{n_s} (\mathbf{B}^s)^T \boldsymbol{\lambda} \quad (3.41)$$

- Differentiating with respect to $\boldsymbol{\lambda}$ and equating with zero yields the limitations:

$$\frac{\partial L(\mathbf{u}, \boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}} = \sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s$$

$$\sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0}$$

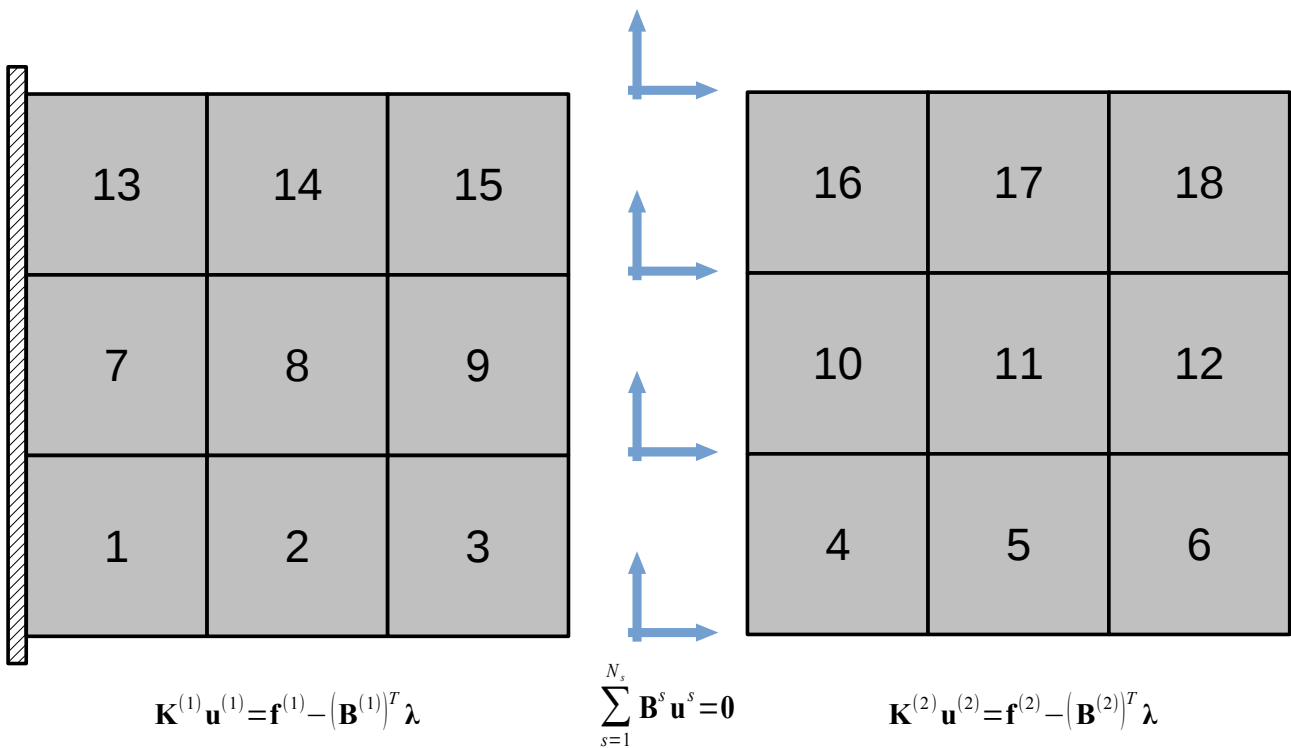


Fig. 3.6: Equations of subdomains and constraints

In sum, the original global problem, which is described by equation (3.27):

$$\mathbf{K} \mathbf{u} = \mathbf{f}$$

is now converted to a subdomain problem which is described by (3.41) subject to (3.39):

$$\mathbf{K}^g \mathbf{u}^g = \mathbf{f}^g - \sum_{s=1}^{n_s} (\mathbf{B}^s)^T \boldsymbol{\lambda}$$

$$\text{subject to: } \sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0}$$
(3.42)

Even though this system is mathematically indefinite, there is only one, unique solution. Eqs. (3.38-3.39), (3.41) can be written in a single equation as:

$$\begin{bmatrix} \mathbf{K}^g & (\mathbf{B}^g)^T \\ \mathbf{B}^g & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}^g \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix}, \quad g \equiv \text{TearedDomain}$$
(3.43)

The equilibrium equations for each subdomain s is:

$$\mathbf{K}^s \mathbf{u}^s = \mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}$$
(3.44)

3.2.2 Supported subdomains and supported degrees of freedom

The domain of Fig. 3.1 is supported on the left side (all left-most nodes are supported). As a result, subdomain 1 is a supported subdomain. We separate the subdomain's dofs in free (f) and constrained (c):

$$\mathbf{K}^s = \begin{bmatrix} \mathbf{K}_{ff}^s & \mathbf{K}_{fc}^s \\ \mathbf{K}_{cf}^s & \mathbf{K}_{cc}^s \end{bmatrix}$$
(3.45)

\mathbf{K}_{ff}^s is the part needed the most. The stiffness matrix of subdomain 1 $\mathbf{K}^{(1)}$ initially has a dimension of $[32 \times 32]$ and after removing all supported dofs we get $\mathbf{K}_{ff}^{(1)}$ whose dimension are $[24 \times 24]$.

Hence, the inverse matrix $(\mathbf{K}^s)^+$ in the case of a supported subdomain is:

$$(\mathbf{K}^s)^+ = (\mathbf{K}_{ff}^s)^{-1}$$
(3.46)

The same separation must also be performed on \mathbf{B} , but only on the columns. Hence:

$$\mathbf{B}^{(1)} = [\mathbf{B}_f^{(1)} \quad \mathbf{B}_c^{(1)}]$$
(3.47)

The initial dimensions are $[8 \times 32]$ and $\mathbf{B}_f^{(1)}$, whose dimensions are $[8 \times 24]$, is extracted.

3.2.3 Floating subdomains

The second subdomain of Fig. 3.3 is not supported and as such it is a floating subdomain. Floating subdomains require special handling because they behave like mechanisms and their stiffness matrix is not invertible. The rigid body modes have to be isolated to make them invertible. To that end, the initial stiffness matrix is separated as follows:

$$\mathbf{K}^s = \begin{bmatrix} \mathbf{K}_{11}^s & \mathbf{K}_{12}^s \\ \mathbf{K}_{21}^s & \mathbf{K}_{22}^s \end{bmatrix} \quad (3.48)$$

where \mathbf{K}_{11}^s is an invertible (full rank) matrix.

In order to make the aforementioned separation, the initial stiffness matrix is factorized. During factorization, some degrees of freedom will have zeros on the diagonal, and these constitute the degrees of submatrix \mathbf{K}_{22}^s . When a zero diagonal entry is found, the whole row and column are moved to the end of the matrix. At the end of the process, the dofs that were placed at the end of the matrix are the dofs that are to be artificially supported.

Hence, the inverse matrix $(\mathbf{K}^s)^+$ in the case of floating subdomains is:

$$(\mathbf{K}^s)^+ = \begin{bmatrix} (\mathbf{K}_{11}^s)^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad (3.49)$$

where the zero-matrices have a dimension such that the final size of $(\mathbf{K}^s)^+$ is the same as the original stiffness matrix (without removals). In the example of Fig. 3.3, the generalized inverse of the second subdomain $(\mathbf{K}^{(2)})^+$ has dimensions $[32 \times 32]$.

No alterations are required for the signed Boolean matrix of floating subdomains, but there are rigid body modes that need to be taken into account. These are the null space of \mathbf{K}^s :

$$\mathbf{R}^s = \text{null}(\mathbf{K}^s) \quad (3.50)$$

RBM can be computed by the following formula:

$$\mathbf{R}^s = \begin{bmatrix} (\mathbf{K}_{11}^s)^{-1} \mathbf{K}_{12}^s \\ \mathbf{I} \end{bmatrix} \quad (3.51)$$

It is obvious from eq. (3.51) that rigid body modes are a natural part of the stiffness matrix and are

therefore problem specific.

Eq. (3.51) provides the computational method of calculating the rigid body modes. Unfortunately, this method is prone to arithmetic errors, especially when the stiffness matrix is ill-conditioned [38], [43]. Furthermore, not all solution methods factorize the matrix and factorizing it specifically for calculating the rigid body modes is inefficient.

An alternative and preferred way for calculating rigid body modes is the analytical method [38]. Both methods produce a group of vectors and the vectors produced by one method are linearly dependent on the vectors produced by the other one – they define the same linear subspace and the displacements they describe are the same. However, the vectors produced through the analytical method have better arithmetic behavior, a property which is especially important in iterative solvers.

In 2D problems, there are 3 possible rigid body modes:

- Displacements parallel to: x -axis, y -axis
- Rotation parallel to: z -axis

For node i of fully floating subdomain s , the rigid body modes are:

$$\mathbf{R}_i^s = [\mathbf{R}_1 \quad \mathbf{R}_2 \quad \mathbf{R}_3] \tag{3.52}$$

$$\mathbf{R}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{R}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{R}_3 = \begin{bmatrix} -y_i \\ x_i \\ 0 \end{bmatrix}$$

where (x_i, y_i) coordinates of node i .

In 3D problems, there are 6 possible rigid body modes:

- Displacements parallel to: x -axis, y -axis, z -axis
- Rotations parallel to: x -axis, y -axis, z -axis

For node i of fully floating subdomain s , the rigid body modes are [41]:

$$\mathbf{R}_i^s = [\mathbf{R}_1 \ \mathbf{R}_2 \ \mathbf{R}_3 \ \mathbf{R}_4 \ \mathbf{R}_5 \ \mathbf{R}_6] \quad (3.53)$$

$$\mathbf{R}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{R}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{R}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R}_4 = \begin{bmatrix} 0 \\ -z_i \\ y_i \end{bmatrix}, \quad \mathbf{R}_5 = \begin{bmatrix} z_i \\ 0 \\ -x_i \end{bmatrix}, \quad \mathbf{R}_6 = \begin{bmatrix} -y_i \\ x_i \\ 0 \end{bmatrix}$$

where (x_i, y_i, z_i) coordinates of node i .

Similar analytical expressions can be defined for finite elements of various types of structural levels (plates, shells, etc). A thorough discussion of RBM handling which also extends to semi-definite problems and partially floating subdomains can be found in [44].

In the example (Fig. 3.5), each node has 2 degrees of freedom, namely x , y . The rotation of a node, which is expressed by the 3rd row of \mathbf{R}_i^s , should not be confused with the rotation of the subdomain, which is expressed by the 3rd column of \mathbf{R}_i^s . Despite not having a rotational degree of freedom on nodes, the subdomain can, of course, rotate. Consequently, for node i of the fully floating subdomain:

$$\mathbf{R}_i^s = [\mathbf{R}_1 \ \mathbf{R}_2 \ \mathbf{R}_3] \quad (3.54)$$

$$\mathbf{R}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{R}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{R}_3 = \begin{bmatrix} -y_i \\ x_i \end{bmatrix}$$

As mentioned, the advantage of the analytical method of calculating \mathbf{R} pertains to the precision of the calculations that are carried out in order to determine rigid body modes and consequently the zero energy modes \mathbf{a} . The number of rigid body modes of a floating subdomains is usually equal to the number of constraints that are required in order for the subdomain to be fully constrained. These are applied as “pseudo-constraints” at the last degrees of freedom of the reordered matrix – those that constitute \mathbf{K}_{22}^s .

The degrees of freedom to be constrained by the implementation of the computational method are usually the last degrees of the floating subdomain, which means that the matrix depicted by eq. (3.48) hasn't actually been reordered and the pseudo-constraints will just be applied on the last degrees of the original matrix. Hence, in the example (Fig. 3.5), the last 3 degrees of subdomain 2 will be constrained. Note that the degrees chosen are always the last, so different numbering of the

degrees leads to different selection of constrained degrees.

This, however, means that for a typical numbering of the domain, constraints will be applied in nodes very close to each other, as shown in Fig. 3.7. This subdomain is indeed fully constrained, but constraints this close create arithmetic errors. Furthermore, this affects round-off errors that tend to accumulate at the end of the factorization process – a problem that is most severe in ill-conditioned matrices. This round-off error accumulation results leads to a greater condition number for the FETI boundary problem and consequently to a slower convergence of the PCPG iterative algorithm. The constraints applied in Fig. 3.8 are better because they provide a more stable subdomain.

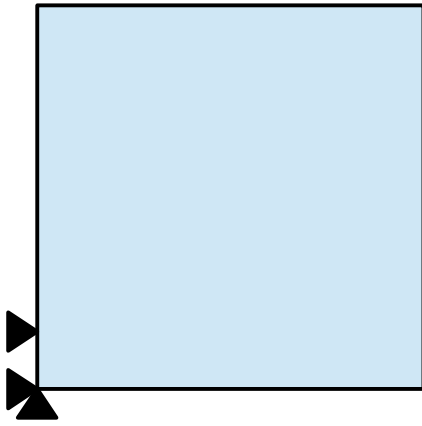


Fig. 3.7: Close constraints

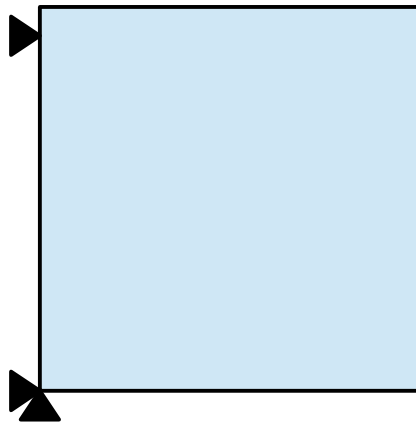


Fig. 3.8: Spaced-out constraints

When rigid body modes are determined analytically, the calculations are independent from the properties of the stiffness matrix, so the aforementioned precision problems do not apply. Therefore, it leads to a more stable and robust arithmetic process.

3.2.4 Linear equations of the FETI interface problem

Displacements of constrained subdomains are directly calculated by solving the subdomain equations for \mathbf{u} :

$$\mathbf{u}^s = (\mathbf{K}_{ff}^s)^{-1} (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) \quad (3.55)$$

In order to be able to also take into account floating subdomains, the formula is modified to include their rigid body modes:

$$\mathbf{u}^s = (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{R}^s \mathbf{a}^s \quad (3.56)$$

where:

- \mathbf{K}^+ is the generalized inverse stiffness matrix
- \mathbf{R} is the matrix containing rigid body modes
- \mathbf{a} represents zero energy modes

This formula covers both supported and floating subdomains, since:

- ✓ In constrained subdomains, where there are no rigid body modes, \mathbf{R} dimensions can be considered $[0 \times 0]$ and the last element of the formula is negated.
- ✓ In constrained subdomains, the generalized inverse is the inverse of \mathbf{K}_{ff}^s (the stiffness matrix with constrained degrees of freedom removed).

Zero energy modes are derived from zero energy conditions as follows:

$$(\mathbf{R}^s)^T \mathbf{K}^s \mathbf{u}^s = \mathbf{0}$$

and because of eq. (3.44):

$$(\mathbf{R}^s)^T (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) = \mathbf{0} \quad (3.57)$$

The final equations for subdomain s are (3.39, 3.56, 3.57):

$$(3.39) \quad \sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0}$$

$$(3.56) \quad \mathbf{u}^s = (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{R}^s \mathbf{a}^s$$

$$(3.57) \quad (\mathbf{R}^s)^T (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) = \mathbf{0}$$

which will be combined in a single system.

Multiply eq. (3.56) with \mathbf{B} :

$$\mathbf{u}^s = (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{R}^s \mathbf{a}^s \quad (3.58)$$

$$\mathbf{B}^s \mathbf{u}^s = \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{B}^s \mathbf{R}^s \mathbf{a}^s$$

$$\mathbf{B}^s \mathbf{u}^s = \mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s - \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T \boldsymbol{\lambda} + \mathbf{B}^s \mathbf{R}^s \mathbf{a}^s$$

Setting:

$$\mathbf{F}^s = \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T \quad (3.59)$$

$$\mathbf{d}^s = \mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s \quad (3.60)$$

$$\mathbf{G}^s = \mathbf{B}^s \mathbf{R}^s \quad (3.61)$$

Hence, the equation for subdomain s becomes:

$$\mathbf{B}^s \mathbf{u}^s = \mathbf{d}^s - \mathbf{F}^s \boldsymbol{\lambda} + \mathbf{G}^s \mathbf{a}^s \quad (3.62)$$

Summing all subdomains results to:

$$\sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \sum_{s=1}^{n_s} \mathbf{d}^s - \sum_{s=1}^{n_s} \mathbf{F}^s \boldsymbol{\lambda} + \sum_{s=1}^{n_s} \mathbf{G}^s \mathbf{a}^s \quad (3.63)$$

The left side of this equation is equal to $\mathbf{0}$ due to (3.39). We also set:

$$\mathbf{F} = \sum_{s=1}^{n_s} \mathbf{F}^s = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T \quad (3.64)$$

$$\mathbf{d} = \sum_{s=1}^{n_s} \mathbf{d}^s = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s \quad (3.65)$$

$$\mathbf{G} = [\mathbf{B}^{(1)} \mathbf{R}^{(1)} \quad \mathbf{B}^{(2)} \mathbf{R}^{(2)} \quad \dots \quad \mathbf{B}^{(n_s)} \mathbf{R}^{(n_s)}] \quad (3.66)$$

$$\mathbf{a} = \begin{Bmatrix} \mathbf{a}^{(1)} \\ \vdots \\ \mathbf{a}^{(n_s)} \end{Bmatrix} \quad (3.67)$$

Eq. (3.63) becomes:

$$\begin{aligned} \mathbf{0} &= \mathbf{d} - \mathbf{F} \boldsymbol{\lambda} + \mathbf{G} \mathbf{a} \\ \mathbf{F} \boldsymbol{\lambda} - \mathbf{G} \mathbf{a} &= \mathbf{d} \end{aligned} \quad (3.68)$$

Furthermore, for subdomain s :

$$\begin{aligned}
(\mathbf{R}^s)^T (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) &= \mathbf{0} \\
(\mathbf{R}^s)^T \mathbf{f}^s - (\mathbf{R}^s)^T (\mathbf{B}^s)^T \boldsymbol{\lambda} &= \mathbf{0}
\end{aligned} \tag{3.69}$$

And for all subdomains:

$$\begin{aligned}
&\begin{bmatrix} (\mathbf{R}^{(1)})^T \mathbf{f}^{(1)} \\ \vdots \\ (\mathbf{R}^{(N_s)})^T \mathbf{f}^{(n_s)} \end{bmatrix} - \begin{bmatrix} (\mathbf{R}^{(1)})^T (\mathbf{B}^{(1)})^T \\ \vdots \\ (\mathbf{R}^{(N_s)})^T (\mathbf{B}^{(n_s)})^T \end{bmatrix} \boldsymbol{\lambda} = \mathbf{0} \\
&\begin{bmatrix} (\mathbf{R}^{(1)})^T \mathbf{f}^{(1)} \\ \vdots \\ (\mathbf{R}^{(n_s)})^T \mathbf{f}^{(n_s)} \end{bmatrix} - [\mathbf{R}^{(1)} \mathbf{B}^{(1)} \quad \dots \quad \mathbf{R}^{(n_s)} \mathbf{B}^{(n_s)}]^T \boldsymbol{\lambda} = \mathbf{0}
\end{aligned} \tag{3.70}$$

We also set:

$$\mathbf{e} = \begin{bmatrix} (\mathbf{R}^{(1)})^T \mathbf{f}^{(1)} \\ \vdots \\ (\mathbf{R}^{(n_s)})^T \mathbf{f}^{(n_s)} \end{bmatrix} \tag{3.71}$$

Eq. (3.70) becomes:

$$\mathbf{e} = \mathbf{G}^T \boldsymbol{\lambda} \tag{3.72}$$

From eqs. (3.68), (3.72):

$$\begin{aligned}
&\begin{cases} \mathbf{F} \boldsymbol{\lambda} - \mathbf{G} \mathbf{a} = \mathbf{d} \\ \mathbf{G}^T \boldsymbol{\lambda} = \mathbf{e} \end{cases} \\
&\begin{cases} \mathbf{F} \boldsymbol{\lambda} - \mathbf{G} \mathbf{a} = \mathbf{d} \\ -\mathbf{G}^T \boldsymbol{\lambda} = -\mathbf{e} \end{cases} \\
&\begin{cases} \mathbf{F} \boldsymbol{\lambda} - \mathbf{G} \mathbf{a} = \mathbf{d} \\ -\mathbf{G}^T \boldsymbol{\lambda} + \mathbf{0} \mathbf{a} = -\mathbf{e} \end{cases} \\
&\begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix}
\end{aligned} \tag{3.73}$$

where \mathbf{F} , \mathbf{d} , \mathbf{G} , \mathbf{a} , \mathbf{e} are given by (3.64), (3.65), (3.66), (3.67), (3.71) respectively.

In the next sections, these formulas are applied on the example of Fig. 3.6 to help clarify useful details.

3.2.4.1 Matrix \mathbf{F}

$$\mathbf{F} = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T$$

$$\mathbf{F} = \mathbf{B}^{(1)} (\mathbf{K}^{(1)})^+ (\mathbf{B}^{(1)})^T + \mathbf{B}^{(2)} (\mathbf{K}^{(2)})^+ (\mathbf{B}^{(2)})^T$$

Subdomain 1 is supported, whereas subdomain 2 is floating, therefore:

$$\mathbf{F} = \mathbf{B}^{(1)} (\mathbf{K}_{\text{ff}}^{(1)})^{-1} (\mathbf{B}^{(1)})^T + \mathbf{B}^{(2)} (\mathbf{K}^{(2)})^+ (\mathbf{B}^{(2)})^T$$

The dimensions of the involved matrices are:

$$[8 \times 8] = [8 \times 24][24 \times 24][24 \times 8] + [8 \times 32][32 \times 32][32 \times 8]$$

As mentioned in Section 3.2.1, all signed Boolean matrices (for the teared domain and all subdomains) have a number of rows equal to the boundary degrees of freedom. Hence, all elements of the sum have equal dimensions regardless of the size of the subdomain (and its stiffness matrix). Specifically, the dimensions will be $[8 \times 8]$, or generally $[n_b \times n_b]$, where n_b is the number of boundary degrees of freedom. Hence, \mathbf{F} will also be $[n_b \times n_b]$.

3.2.4.2 Vector \mathbf{d}

$$\mathbf{d} = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s$$

$$\mathbf{d} = \mathbf{B}^{(1)} (\mathbf{K}^{(1)})^+ \mathbf{f}^{(1)} + \mathbf{B}^{(2)} (\mathbf{K}^{(2)})^+ \mathbf{f}^{(2)}$$

$$\mathbf{d} = \mathbf{B}^{(1)} (\mathbf{K}_{\text{ff}}^{(1)})^{-1} \mathbf{f}^{(1)} + \mathbf{B}^{(2)} (\mathbf{K}^{(2)})^+ \mathbf{f}^{(2)}$$

The dimensions of the matrices involved are:

$$[8 \times 1] = [8 \times 24][24 \times 24][24 \times 1] + [8 \times 32][32 \times 32][32 \times 1]$$

Just as in \mathbf{F} , \mathbf{B} projects each subdomain's matrix on the interface. Each element to be added as well as the resulting \mathbf{d} , will have a size of $[8 \times 1]$, or generally $[n_b \times 1]$.

3.2.4.3 Matrix \mathbf{G}

$$\mathbf{G} = \begin{bmatrix} \mathbf{B}^{(1)} \mathbf{R}^{(1)} & \mathbf{B}^{(2)} \mathbf{R}^{(2)} & \dots & \mathbf{B}^{(n_s)} \mathbf{R}^{(n_s)} \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} \mathbf{B}^{(1)} \mathbf{R}^{(1)} & \mathbf{B}^{(2)} \mathbf{R}^{(2)} \end{bmatrix}$$

Subdomain 1 is constrained and as such there are no rigid body modes. Therefore matrix $\mathbf{R}^{(1)}$ can be assumed to have zero number of columns: $[32 \times 0]$. Floating subdomains, like the second one, have $n_R=3$ rigid body modes in 2D problems or $n_R=6$ for 3D problems. Hence, if n_f is the number of floating subdomains then the dimensions of \mathbf{G} will be $[n_b \times (n_R \cdot n_f)]$. In the example, \mathbf{G} has dimensions $[8 \times 3]$.

3.2.4.4 Vector \mathbf{a}

$$\mathbf{a} = \begin{Bmatrix} \mathbf{a}^{(1)} \\ \vdots \\ \mathbf{a}^{(n_s)} \end{Bmatrix}$$

$$\mathbf{a} = \begin{Bmatrix} \mathbf{a}^{(1)} \\ \mathbf{a}^{(2)} \end{Bmatrix}$$

Supported subdomains have \mathbf{a} with zero number of rows. So, for subdomain 1, $\mathbf{a}^{(1)}$ has dimensions $[0 \times 1]$. Consequently, \mathbf{a} has the dimensions of $\mathbf{a}^{(2)}$, which are $[3 \times 1]$. In general, \mathbf{a} has dimensions $[(n_R \cdot n_f) \times 1]$.

3.2.4.5 Vector \mathbf{e}

$$\mathbf{e} = \begin{bmatrix} (\mathbf{R}^{(1)})^T \mathbf{f}^{(1)} \\ \vdots \\ (\mathbf{R}^{(N_s)})^T \mathbf{f}^{(N_s)} \end{bmatrix} \quad (3.74)$$

Constrained subdomains have \mathbf{R} with zero number of columns, so their \mathbf{R}^T has zero number of lines. According to this, $\mathbf{e}^{(1)}$ has dimensions $[0 \times 1]$ and consequently \mathbf{e} has the size of $\mathbf{e}^{(2)}$, which is $[3 \times 1]$. In general terms, \mathbf{e} has dimension $[(n_R \cdot n_f) \times 1]$.

FETI is a compliance method, as can be observed from the formula of \mathbf{F} (eq. 3.64), because the generalized inverse of the stiffness matrix, i.e. the compliance matrix is utilized. This is contrary to the primal subdomain implementation (Section 3.1), which uses the stiffness matrix. This is an advantage because it leads to better arithmetic behavior.

Also note that the final equations are reached with the same handling of internal and boundary degrees of freedom, which is another difference from the primal subdomain implementation. Hence, another advantage of FETI is that all nodes are treated the same way.

3.2.5 Matrices of the boundary problem

The signed Boolean matrix has elements that are either ± 1 or 0 . When \mathbf{B} is multiplied with another matrix, a certain degree is either picked (when its corresponding element in \mathbf{B} is ± 1) or ignored (when its corresponding element is 0). Therefore, \mathbf{B} essentially picks degrees of freedom that will participate in the interface problem calculations. With that in mind, the matrices involved in the interface problem can be explained more naturally.

3.2.5.1 Matrix \mathbf{F}

$$\mathbf{F} = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T$$

$\mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T$ comes from subdomain s , and \mathbf{B} is responsible for singling out the compliance entries of the boundary degrees of that subdomain. However, \mathbf{B} includes the degrees of the entire domain. Hence, \mathbf{F} is “global” and \mathbf{B} selects the corresponding compliance entries from the subdomains and places them in the appropriate positions. Summing this for all s , the contributions of all subdomains are collected for the degrees of the interface problem. As a result, \mathbf{F} is the compliance matrix of the interface problem.

3.2.5.2 Vector \mathbf{d}

$$\mathbf{d} = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s$$

$(\mathbf{K}^s)^+ \mathbf{f}^s$ is solving directly for subdomain s . Thus, it reflects displacements of s due to forces applied to s . By multiplying with \mathbf{B} , only the displacements needed for the interface problem remain: $\mathbf{B}^s (\mathbf{K}^s)^+ \mathbf{f}^s$ are the displacements of the boundary degrees of subdomain s . The sum for all s yields \mathbf{d} and it contains the displacements of all degrees of freedom due to each subdomain's forces.

3.2.5.3 Matrix \mathbf{G} and vector \mathbf{e}

$$\mathbf{G} = \begin{bmatrix} \mathbf{B}^{(1)} \mathbf{R}^{(1)} & \mathbf{B}^{(2)} \mathbf{R}^{(2)} & \dots & \mathbf{B}^{(n_s)} \mathbf{R}^{(n_s)} \end{bmatrix} \quad \mathbf{e} = \begin{bmatrix} (\mathbf{R}^{(1)})^T \mathbf{f}^{(1)} \\ \vdots \\ (\mathbf{R}^{(n_s)})^T \mathbf{f}^{(n_s)} \end{bmatrix}$$

With the effect of \mathbf{B} , matrix \mathbf{G} contains only rigid body modes of the interface problem, whereas vector \mathbf{e} contains displacements of boundary degrees of freedom caused by the aforementioned rigid body modes (just as \mathbf{d} contains displacements caused by deformation).

3.2.5.4 Vectors λ and \mathbf{a}

From the final system of equations (eq. 3.73)

$$\begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \lambda \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix}$$

it is clear that λ expresses the forces that need to be applied on the boundary degrees of freedom so that interface displacements expressed in \mathbf{d} are valid. Furthermore, \mathbf{a} reflects the zero energy modes of floating subdomains.

3.2.6 Special cases

The example used so far does not cover certain special cases. The domain depicted in Fig. 3.9 will be used to examine them. Domain numbering for nodes and degrees of freedom is shown in Fig. 3.10.

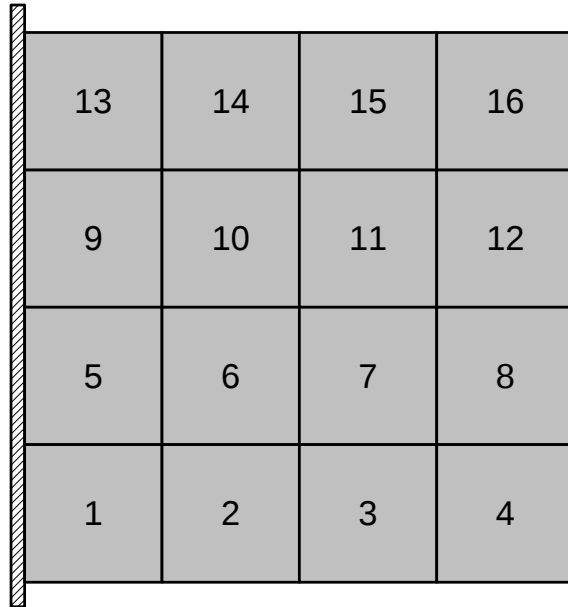


Fig. 3.9: Example 2

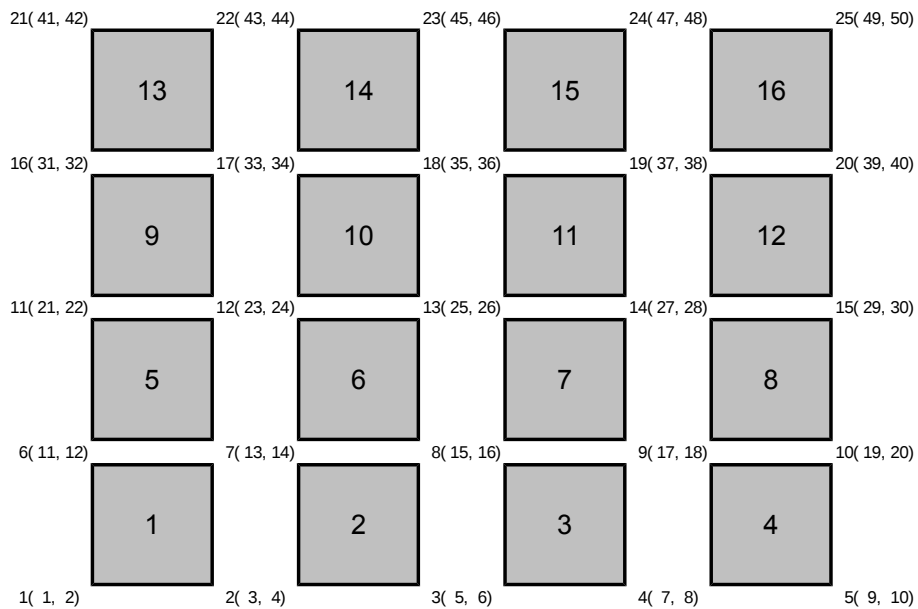


Fig. 3.10: Numbering of nodes and degrees of freedom of the domain

The domain is teared into 4 equal subdomains as shown in Fig. 3.11. Domain-scope numbering and subdomain-scope numbering for the teared domain is shown in Figs. 3.12, 3.13, respectively.

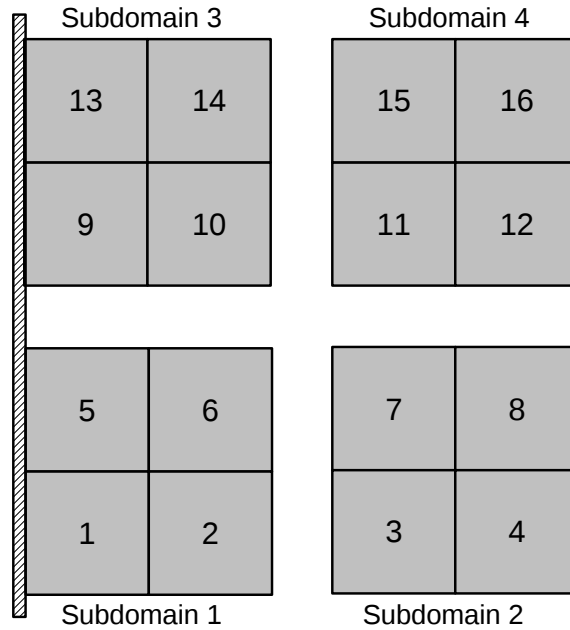


Fig. 3.11: Domain teared into 4 subdomains

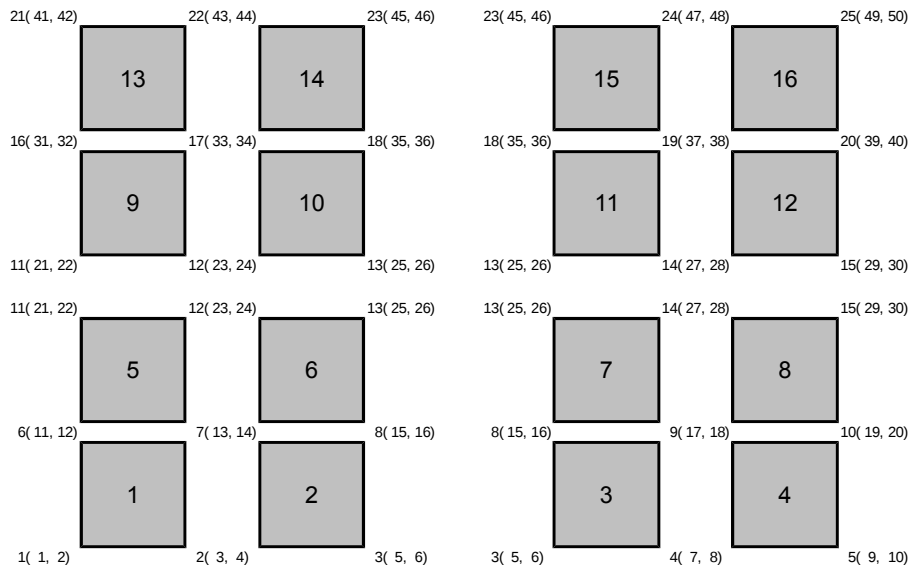


Fig. 3.12: Numbering of nodes and degrees of freedom of the teared domain

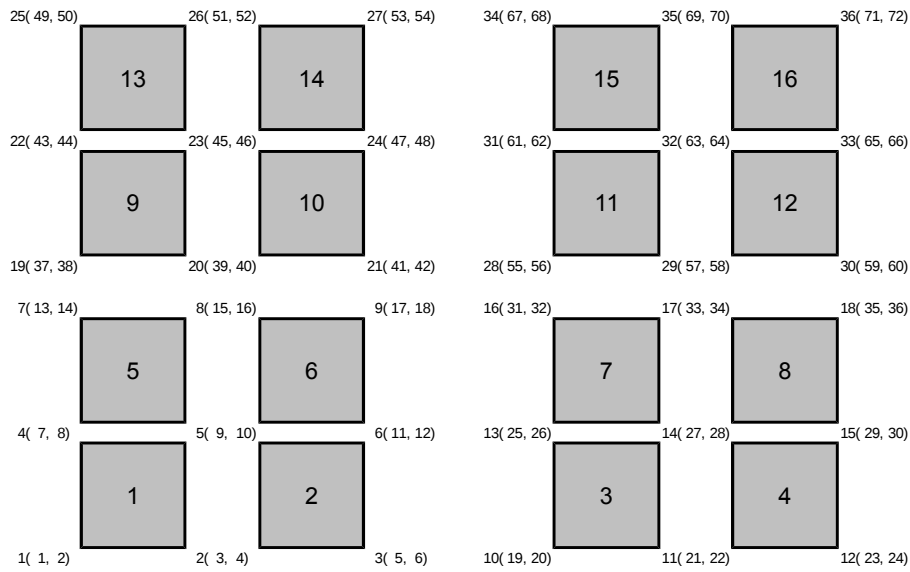


Fig. 3.13: Global numbering of nodes and degrees of freedom of the subdomains

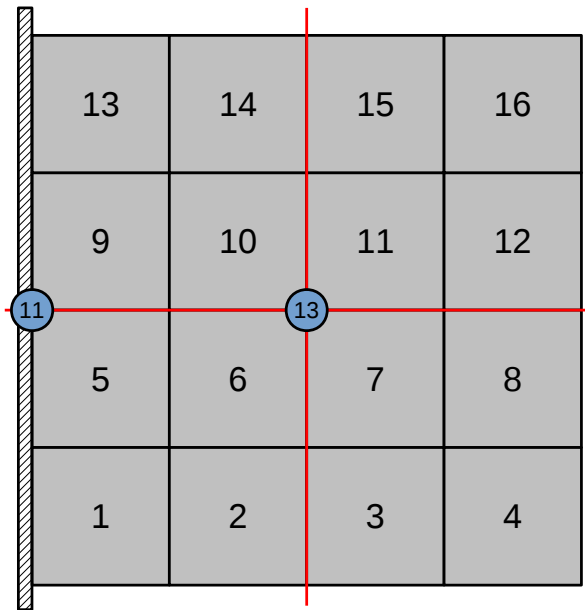


Fig. 3.14: Special node cases

There are 2 node types that need further examination. These are shown in Fig. 3.14.

Node 13 (domain numbering) is a boundary node that belongs to 4 subdomains. Its multiplicity is equal to 4 and its 4 instances in the subdomains are 9, 16, 21, 28.

Node 11 (domain numbering) is a boundary node that is constrained. Its multiplicity is equal to 2 and its 2 instances in the subdomains are 7, 19.

3.2.6.1 Boundary nodes with multiplicity >2

In the first example (Fig. 3.5), all nodes involved were either internal, therefore having multiplicity equal to 1, or boundary with multiplicity equal to 2. For boundary node i , it was sufficient to use only one equation for each degree of freedom of that node:

$$u_i^{(1)} = u_i^{(2)} \quad (3.75)$$

(where 1 and 2 are the two subdomains the node belongs to) so that all instances of the node have the same properties. For a node with multiplicity equal to 4 we have the following equations:

$$u_i^{(1)} = u_i^{(2)} = u_i^{(3)} = u_i^{(4)} \quad (3.76)$$

There are 3 ways of handling this set of equations:

- Minimum Constraints
- Non-Redundant Constraints
- Fully Redundant Constraints

3.2.6.1.1 Minimum Constraints

The fewest possible equations we can use are 3. For example (Fig. 3.15):

$$\begin{aligned}u_i^{(1)} &= u_i^{(2)} \\u_i^{(2)} &= u_i^{(4)} \\u_i^{(4)} &= u_i^{(3)}\end{aligned}\tag{3.77}$$

These equations are just enough to ensure that all instances of the node will have equal properties. However, with this method, the number of instances and the number of equations used between subdomains is not equal – instances belonging to subdomain 2 and 3 have two equations between them whereas instances belong to subdomain 1 and 3 only have one. A universal handling of the subdomains is simpler and conducive to programming. For this reason we favor the next method over this one.

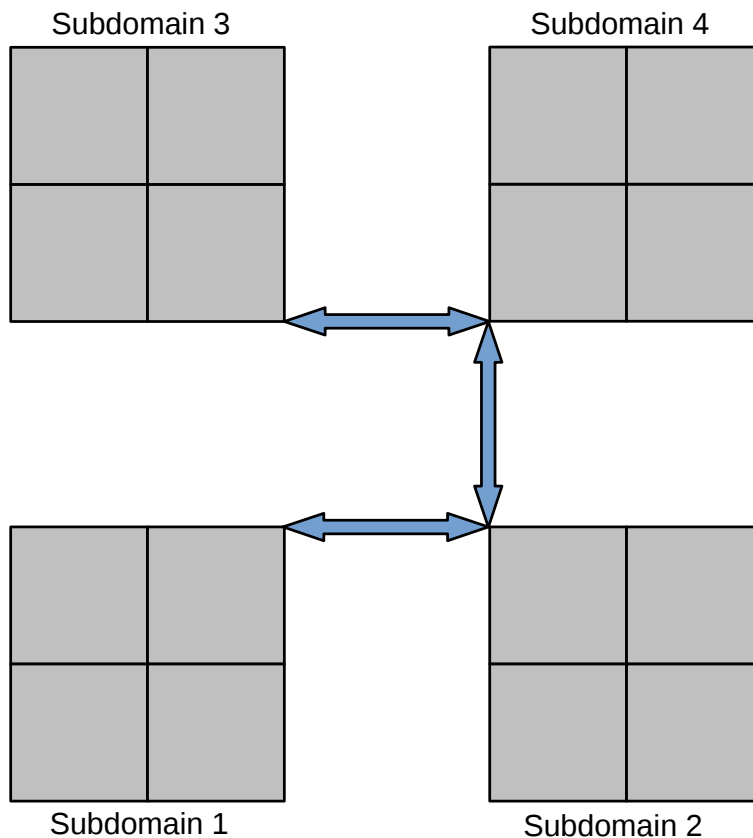


Fig. 3.15: Minimum constraints

3.2.6.1.2 Non-Redundant Constraints

Even though the name of this method suggests otherwise, one equation is actually redundant. However, we achieve universal handling of subdomains. The equations in this case are:

$$\begin{aligned}u_i^{(1)} &= u_i^{(2)} \\u_i^{(2)} &= u_i^{(4)} \\u_i^{(4)} &= u_i^{(3)} \\u_i^{(3)} &= u_i^{(1)}\end{aligned}\tag{3.78}$$

As depicted in the Fig. 3.16, each instance is connected to its horizontal and vertical neighbors but not with the diagonal ones. Each instance is connected to two other instances. We have 4 equations in total for each degree of freedom of the node in question.

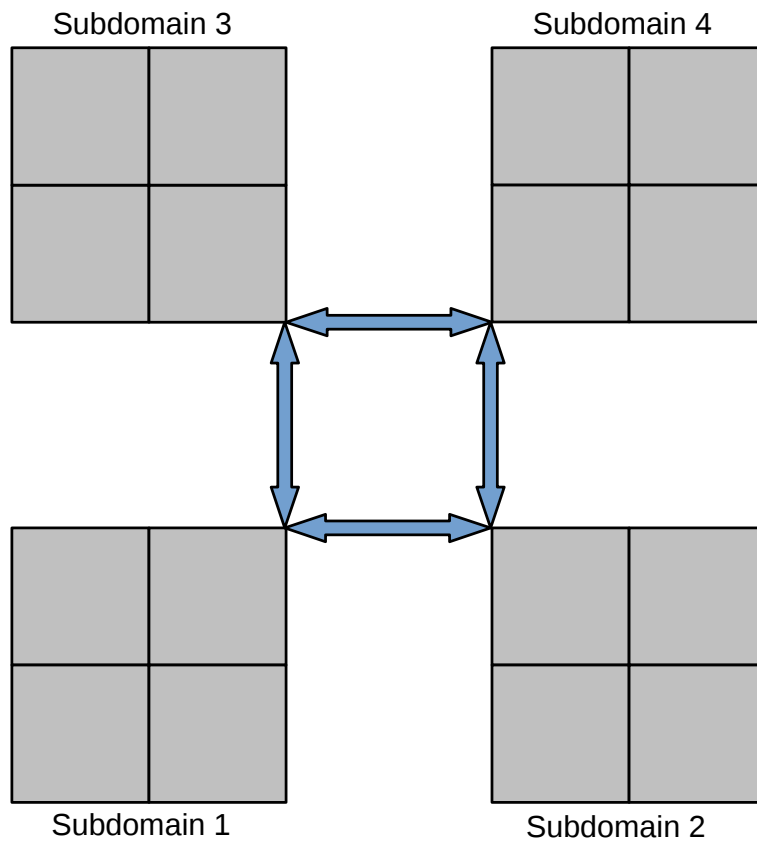


Fig. 3.16: Non-redundant constraints

3.2.6.1.3 Fully Redundant Constraints

In this variation, all instances are connected to all other instances.

$$\begin{aligned}
 u_i^{(1)} &= u_i^{(2)}, u_i^{(1)} = u_i^{(3)}, u_i^{(1)} = u_i^{(4)} \\
 u_i^{(2)} &= u_i^{(3)}, u_i^{(2)} = u_i^{(4)} \\
 u_i^{(3)} &= u_i^{(4)}
 \end{aligned}
 \tag{3.79}$$

As depicted in Fig. 3.17, each instance is connected to its horizontal and vertical neighbors as well as the diagonal ones. Each instance is connected to three others. There are 6 equations for each degree of the node in question, some of which are redundant. In this work, the fully redundant variation is used. Even though the extra equations increase the number of rows of \mathbf{B} , the convergence rate is improved [39].

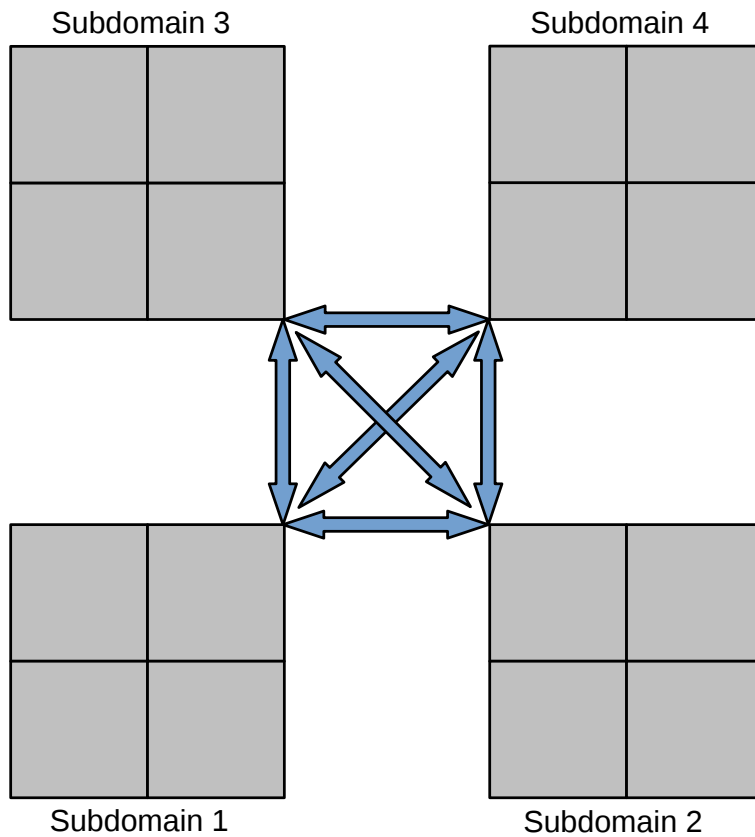


Fig. 3.17: Fully redundant constraints

3.2.6.2 Constrained boundary nodes

For typical constrained nodes (i.e. not necessarily boundary), the appropriate degrees of freedom are removed both from the stiffness matrix (both rows and columns) and the signed Boolean matrix (columns only). As mentioned in Section 3.2.1, the signed Boolean matrix \mathbf{B} matrix of a subdomain or of the expanded domain has a number of rows equal to the required limitations and a number of columns equal to the total degrees of freedom of the teared domain.

When a boundary degree is constrained, then the relevant equations-limitations need to be removed from the interface problem as well. The rows of \mathbf{B} are separated in “active” and “inactive”:

$$\mathbf{B}^{(1)} = \begin{bmatrix} \mathbf{B}_{active}^{(1)} \\ \mathbf{B}_{inactive}^{(1)} \end{bmatrix} \quad (3.80)$$

Only the active part is used when solving each participating subdomain.

In the example (Fig. 3.12) where each node has 2 degrees of freedom, the number of limitations that need to be applied is 28 : there are $6 \cdot 2 = 12$ for the middle node and $1 \cdot 2 = 2$ for each one of the other 8 boundary nodes, for the total of $12 + 8 \cdot 2 = 28$. However, the 2 equations related to the constrained node need to be ignored. Consequently, all signed Boolean matrices will have 26 rows, instead of 28 .

3.2.7 Solving the FETI interface problem

The system of linear equations (3.73) may be solved directly:

$$(3.73) \quad \begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix}$$

$$\begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix} \quad (3.81)$$

However, for problems with a very large number of degrees of freedom, solving the resulting system of algebraic equations with a direct solver is very time consuming. Additionally, it requires creating and storing the fully dense matrix \mathbf{F} . Solving problems with an iterative method is more efficient both time-wise and storage-wise since \mathbf{F} is handled indirectly (through subdomain-scope vectors), hence there is no need to ever create and store it.

However, the system is not positive definite so this prevents the usage of the standard PCG algorithm. The solution of the interface problem of eq. (3.73) is based on a projected PCG algorithm, where the vector search space is projected to a different subspace [37], [43]. This variant is called preconditioned conjugate projected gradient (PCPG) and has the ability to deal with the zero elements (in the block at the bottom right) that results from the constraints applied by (3.39).

The interface problem of eq. (3.73) can be rewritten as:

$$\min \Pi(\boldsymbol{\lambda}) = \frac{1}{2} \boldsymbol{\lambda}^T \mathbf{F} \boldsymbol{\lambda} - \boldsymbol{\lambda}^T (\mathbf{G} \mathbf{a} + \mathbf{d}) \quad (3.82)$$

subject to: $\mathbf{G}^T \boldsymbol{\lambda} = \mathbf{e}$

PCPG uses the orthogonal projection matrix (or projector) in order to solve the interface problem. The projector is calculated by the formula :

$$\mathbf{P} = \mathbf{I} - \mathbf{G} (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T \quad (3.83)$$

where \mathbf{I} is the identity matrix whose dimensions are $[n_b \times n_b]$. This is because the dimensions of \mathbf{G} are $[n_b \times (n_R \cdot n_f)]$ where n_R the number of rigid modes, n_f the number of floating subdomains, n_b the number of boundary degrees of freedom. The projection matrix \mathbf{P} is used in order to "fix" the current solution vector $\boldsymbol{\lambda}_m$ so that the equation $\mathbf{G}^T \boldsymbol{\lambda} = \mathbf{e}$ is satisfied.

Matrix $\mathbf{G}^T \mathbf{G}$ is sparse, symmetric and positive definite. Furthermore, its dimensions are small: $[(n_R \cdot n_f) \times (n_R \cdot n_f)]$, proportional to the number of rigid body modes of the floating subdomains.

The PCPG algorithm used for solving the interface problem (3.73) and symbolizing the preconditioner (see Section 3.3) with $\tilde{\mathbf{F}}^{-1}$ is shown in Fig. 3.18.

Initialization

$$\lambda_0 = \mathbf{G} (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{e} \quad (3.84)$$

$$\mathbf{r}_0 = \mathbf{d} - \mathbf{F} \lambda_0 \quad (3.85)$$

Iterate $m=0, 1, \dots$, until convergence

$$\mathbf{w}_{m-1} = \mathbf{P} \mathbf{r}_{m-1} \quad (3.86)$$

$$\mathbf{y}_{m-1} = \mathbf{P} \tilde{\mathbf{F}}^{-1} \mathbf{P} \mathbf{r}_{m-1}$$

$$\mathbf{z}_{m-1} = \tilde{\mathbf{F}}^{-1} \mathbf{w}_{m-1} \quad (3.87)$$

$$\mathbf{y}_{m-1} = \mathbf{P} \mathbf{z}_{m-1} \quad (3.88)$$

$$\beta_m = \frac{\mathbf{y}_{m-1}^T \mathbf{w}_{m-1}}{\mathbf{y}_{m-2}^T \mathbf{w}_{m-2}} \quad (\text{For } m=1, \beta_1=0) \quad (3.89)$$

$$\mathbf{p}_m = \mathbf{y}_{m-1} + \beta_m \mathbf{p}_{m-1} \quad (\text{For } m=1, \mathbf{p}_1 = \mathbf{y}_0) \quad (3.90)$$

$$\gamma_m = \frac{\mathbf{y}_{m-1}^T \mathbf{w}_{m-1}}{\mathbf{p}_m^T \mathbf{F} \mathbf{p}_m}$$

$$\lambda_m = \lambda_{m-1} + \gamma_m \mathbf{p}_m$$

$$\mathbf{r}_m = \mathbf{r}_{m-1} - \gamma_m \mathbf{F} \mathbf{p}_m$$

Fig. 3.18: The PCPG algorithm

There are two steps involving the projection matrix \mathbf{P} (eq. 3.86, 3.87), so that symmetry is maintained. Performing those two steps causes the propagation of arithmetic information to all subdomains in each iteration which results in faster convergence [37].

It has been observed that for most structural problems, the norm of residual forces of the global system $\mathbf{K} \mathbf{u} = \mathbf{f}$ (eq. 3.27) is usually $10^2 \div 10^3$ times greater [37] than the norm of boundary residual forces resulting from eq. (3.73). This means that the convergence criterion of the iterative procedure will have to be based on the norm of the global residual forces:

$$\|\mathbf{K}\mathbf{u}_m - \mathbf{f}\| \quad (3.91)$$

where \mathbf{u}_m is the displacement vector of iteration m . The straightforward approach is to extract λ_m and calculate \mathbf{a}_m . Afterwards, through (3.56):

$$\mathbf{u}^s = (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{R}^s \mathbf{a}^s$$

calculate the node displacements of each subdomain and consequently those of the domain. For a given precision ε , if:

$$\frac{\|\mathbf{K}\mathbf{u}_m - \mathbf{f}\|}{\|\mathbf{f}\|} \leq \varepsilon \quad (3.92)$$

then the process has converged and the approximate solution for precision ε has been reached.

The criterion outlined above is called the objective criterion. However, using the objective criterion for every iteration is very time-consuming. Instead, an approximate criterion is used with the intention of improving performance. An accepted estimation of the norm (3.91) is the norm $\|\mathbf{z}_m\|$, which is calculated on every iteration by multiplying the residual vector \mathbf{r}_m with the projector and the preconditioner (eq. 3.87). Hence, the approximate criterion will be:

$$\frac{\|\mathbf{z}_m\|}{\|\mathbf{f}\|} \leq \varepsilon \quad (3.93)$$

The results derived from the iterative solution are the values of Lagrange multipliers $\boldsymbol{\lambda}$ (see Section 3.2.1). In the system of linear equations (3.73)

$$\begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix}$$

after solving for $\boldsymbol{\lambda}$ the second equation is already satisfied. The first one still has the unknown vector \mathbf{a} . Solving for \mathbf{a} :

$$\mathbf{F}\boldsymbol{\lambda} - \mathbf{G}\mathbf{a} = \mathbf{d}$$

$$\mathbf{G}\mathbf{a} = \mathbf{F}\boldsymbol{\lambda} - \mathbf{d}$$

Multiplying with \mathbf{G}^T :

$$\mathbf{G}^T \mathbf{G} \mathbf{a} = \mathbf{G}^T (\mathbf{F} \boldsymbol{\lambda} - \mathbf{d})$$

and because $\mathbf{G}^T \mathbf{G}$ is invertible:

$$\mathbf{a} = (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T (\mathbf{F} \boldsymbol{\lambda} - \mathbf{d}) \quad (3.94)$$

At this point, all unknowns of the interface problem have been calculated. The next step is to calculate displacements from eq. (3.56):

$$\mathbf{u}^s = (\mathbf{K}^s)^+ (\mathbf{f}^s - (\mathbf{B}^s)^T \boldsymbol{\lambda}) + \mathbf{R}^s \mathbf{a}^s$$

which calculates displacements for the subdomains and consequently the teared domain. In order to find the displacements of the original domain, the displacements of the teared domain need to be averaged for degrees where the multiplicity is two or higher. If a node only has 1 instance, then its displacements will be the displacements of that instance. If a node has more instances, then there are more than one set of calculated displacements for the node. Due to the restriction from eq. (3.39):

$$\sum_{s=1}^{n_s} \mathbf{B}^s \mathbf{u}^s = \mathbf{0}$$

those sets are all equal, barring arithmetic differences. Thus, in order to find the displacements \mathbf{u}_i of node i with multiplicity equal to m :

$$\mathbf{u}_i = \frac{1}{m} \sum_{s=1}^m \mathbf{u}_i^s \quad (3.95)$$

where \mathbf{u}_i^s are the displacements of the instance of the node in subdomain s .

3.3 Preconditioners

The efficiency of the PCPG method is greatly affected by the preconditioning used. Two preconditioners that were presented in initial FETI articles are still being widely used to enhance the convergence speed of the PCPG algorithm [37], [39]: the powerful Dirichlet preconditioner and the low-cost lumped preconditioner. Despite intensive research on FETI and constant improvements of the method, the number of articles pertaining to preconditioning for PCPG is relatively limited. Alternative preconditioning methods suggested are based mostly on expanded versions of the original Dirichlet and lumped preconditioners.

In this work, the Dirichlet and lumped preconditioners are presented along with the Diagonal preconditioner, which more effective than lumped and is more economical to create than Dirichlet.

For every subdomain s , the stiffness matrix is rearranged:

$$\mathbf{K}^s = \begin{bmatrix} \mathbf{K}_{ii}^s & \mathbf{K}_{ib}^s \\ \mathbf{K}_{bi}^s & \mathbf{K}_{bb}^s \end{bmatrix} \quad (3.96)$$

where i represents internal degrees of subdomain s and b represents the boundary degrees of subdomain s . Since the stiffness matrix is symmetric:

$$\left(\mathbf{K}_{ib}^s\right)^T = \mathbf{K}_{bi}^s \quad (3.97)$$

The same rearrangement must be performed on matrix \mathbf{B} so that each row/column corresponds to the appropriate degree of the rearranged stiffness matrix. Only the columns of \mathbf{B} , which represent all degrees of subdomain s , need to be rearranged.

$$\mathbf{B}^s = \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix} \quad (3.98)$$

The biggest cost requirement of the preconditioner on each subdomain s is \mathbf{K}_{ii}^s which is the part of the subdomain stiffness matrix that contains the internal degrees of freedom. Hence, whereas the Dirichlet preconditioner uses the exact \mathbf{K}_{ii}^s matrix, other variations use lower cost approximations. For example, the Diagonal preconditioner uses the diagonal of \mathbf{K}_{ii}^s . Other variations use a matrix based on the symmetric successive over-relaxation method (SSOR) or one that is obtained from an incomplete Cholesky factorization.

3.3.1 General expression of preconditioners

The general form of preconditioners is:

$$\tilde{\mathbf{F}}^{-1} = \sum_{s=1}^{n_s} \mathbf{W}^s \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \\ \mathbf{0} & \mathbf{K}_{bb}^s - (\mathbf{K}_{ib}^s)^T (\tilde{\mathbf{K}}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \end{bmatrix} \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix}^T \mathbf{W}^s \quad (3.99)$$

The symbol $\tilde{\square}$ signifies an approximation of the matrix.

It should be noted that the preconditioner is an approximation of matrix \mathbf{F}^{-1} . If the exact \mathbf{F}^{-1} is used as a preconditioner then the method will converge in a single iteration but exact calculation of \mathbf{F}^{-1} is equivalent to solving the system directly. An important advantage of PCPG is that the time-consuming relevant calculations and storage of matrix \mathbf{F}^{-1} are avoided. For large problems, it is practically mandatory to approximate \mathbf{F}^{-1} . Another approximation pertains to \mathbf{K}_{ii}^s because it consumes the largest part of the storage of the preconditioner.

The general expression of eq. (3.99) can be simplified into a more useful form. The submatrices \mathbf{B}_i^s on each subdomain are all $\mathbf{0}$, since they represent internal degrees of freedom. Therefore, with $\mathbf{B}_i^s = \mathbf{0}$:

$$\tilde{\mathbf{F}}^{-1} = \sum_{s=1}^{n_s} \mathbf{W}^s \mathbf{B}_b^s \left(\mathbf{K}_{bb}^s - (\mathbf{K}_{ib}^s)^T (\tilde{\mathbf{K}}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \right) (\mathbf{B}_b^s)^T \mathbf{W}^s \quad (3.100)$$

Each part of the sum is independent and belongs to a different subdomain. This allows for an efficient handling in parallel processing.

Matrix \mathbf{W} is a diagonal matrix that takes the multiplicities of boundary degrees into account. Its dimensions are $n_b \times n_b$ where n_b is the size of the interface problem. Each entry is equal to the reciprocal of the multiplicity of the corresponding degree. In the example of Fig. 3.4, \mathbf{W} has size $[8 \times 8]$, and since each boundary degree of freedom belongs only in 2 subdomains, each diagonal entry is equal to $1/2$, hence:

3.3.3 Lumped preconditioner

The lumped preconditioner ignores the \mathbf{K}_{ii}^s matrix:

$$\begin{aligned}\widetilde{\mathbf{K}}_{ii}^s &= \mathbf{0} \\ \widetilde{\mathbf{F}}^{-1} &= \sum_{s=1}^{n_s} \mathbf{W}^s \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix} \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}_{bb}^s \end{bmatrix} \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix}^T \mathbf{W}^s \\ \widetilde{\mathbf{F}}^{-1} &= \sum_{s=1}^{n_s} \mathbf{W}^s \mathbf{B}_b^s \mathbf{K}_{bb}^s (\mathbf{B}_b^s)^T \mathbf{W}^s\end{aligned}\quad (3.103)$$

The lumped preconditioner is low-cost because it does not need \mathbf{K}_{ii}^s at all, but also because \mathbf{K}_{ib}^s and $\mathbf{K}_{bi}^s = (\mathbf{K}_{ib}^s)^T$ are also unneeded after nullifying \mathbf{K}_{ii}^s .

3.3.4 Diagonal preconditioner

The diagonal preconditioner approximates \mathbf{K}_{ii}^s with its diagonal:

$$\begin{aligned}\widetilde{\mathbf{K}}_{ii}^s &= \mathbf{D}_{ii}^s \\ \widetilde{\mathbf{F}}^{-1} &= \sum_{s=1}^{n_s} \mathbf{W}^s \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix} \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}_{bb}^s - (\mathbf{K}_{ib}^s)^T (\mathbf{D}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \end{bmatrix} \begin{bmatrix} \mathbf{B}_i^s & \mathbf{B}_b^s \end{bmatrix}^T \mathbf{W}^s \\ \widetilde{\mathbf{F}}^{-1} &= \sum_{s=1}^{n_s} \mathbf{W}^s \mathbf{B}_b^s \left(\mathbf{K}_{bb}^s - (\mathbf{K}_{ib}^s)^T (\mathbf{D}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \right) (\mathbf{B}_b^s)^T \mathbf{W}^s\end{aligned}\quad (3.104)$$

The diagonal preconditioner is intermediate in cost and efficiency to Dirichlet and lumped. By using only the diagonal elements of \mathbf{K}_{ii}^s , the diagonal preconditioner is close to Dirichlet's efficiency with a lower computational cost while also avoiding the arithmetic errors of matrix inversion.

3.3.5 Preconditioner usage

Between the two extremes regarding the approximation of \mathbf{K}_{ii}^s , Dirichlet is more efficient than lumped but costs a lot more memory. Depending on the type of the problem there are some suggestions [42], [45]:

- For 2D and 3D elasticity problems that are teared in a small number of subdomains, the lumped preconditioner is more efficient.
- For 2D and 3D elasticity problems that are teared in a large number of subdomains, the Dirichlet preconditioner should be favored.
- For ill-conditioned problems, like plate and shell problems, the utilization of the Dirichlet preconditioner is always advised.

More information on the properties and characteristics of the 2 conventional preconditioners as well as other ones can be found in references [39], [46].

3.4 Implementation considerations in the context of FETI

FETI exhibits fast convergence as a method so a computer implementation has to also make sure that each iteration is done as efficiently as possible. This chapter explores general techniques towards that goal. Among others, the special characteristics of the matrices will be exploited to improve their storage requirements and reduce the calculations required for the operations they are involved in.

3.4.1 Matrix format

The symmetry of the matrix must be utilized. The stiffness matrix as well as derivatives like $\mathbf{B} \mathbf{K} \mathbf{B}^T$ are all symmetric, i.e.:

$$\mathbf{K}^T = \mathbf{K}$$
$$(\mathbf{B} \mathbf{K} \mathbf{B}^T)^T = (\mathbf{B}^T)^T \mathbf{K} (\mathbf{B})^T = \mathbf{B} \mathbf{K} \mathbf{B}^T$$

Therefore, symmetry can be taken into account and the diagonal and upper or lower triangle of the matrix can be stored instead of the whole matrix.

Furthermore, and in conjunction with symmetry the particular pattern of the matrices can be taken into account: the matrices are sparse, i.e. there is a large number of non-zeros. Depending on the intended usage of the matrix there are several options: a sparse format, which stores non-zeros only can be used for the assembly of the matrix or the skyline format can be used if factorization is required. Matrix formats and their particular characteristics is presented extensively in Chapter 5. The special matrix storage also implies that matrices with zeros are avoided. Each matrix type performs operations (e.g. matrix-vector multiplication, multiplication with scalar etc) by specifically taking into account the underlying structure of the matrix. Therefore, useless operations (with zeros) that do not affect the result are kept to a minimum.

3.4.2 Variable type

Apart from the matrix storage format, the matrices should be stored with the smallest variable type that is required for the matrix. The stiffness matrix is stored in double precision because the extra precision is required. The preconditioner can be stored with single precision. Single precision preconditioners are thoroughly presented in [47]. Preconditioners are an approximation anyway

whose sole purpose is to accelerate convergence. Better preconditioners imply faster convergence, but the preconditioner itself must be as cheap as possible and storing it in single precision requires half the memory while performing the same.

The signed Boolean matrices on the other hand only contain -1 , 0 or $+1$ so they can be stored with much cheaper variables (even a single bit per value is possible here, see Section 3.4.4). See also the Section 3.4.5 pertaining to the multiplicity matrix.

3.4.3 Order of calculations

The first 3 items (eq. 3.86, 3.87, 3.88) of the iterative process of the PCPG algorithm (Fig. 3.18) perform the calculation $\mathbf{y}_{m-1} = \mathbf{P} \tilde{\mathbf{F}}^{-1} \mathbf{P} \mathbf{r}_{m-1}$ but they perform it from right to left. It is very efficient to avoid performing matrix-matrix multiplication and hold operations until a vector appears on the right. Then, perform the calculation as a series of matrix-vector multiplication as shown in Section 5.8.

There are two additional relevant items from the PCPG algorithm:

1. $\gamma_m = \frac{\mathbf{y}_{m-1}^T \mathbf{W}_{m-1}}{\mathbf{p}_m^T \mathbf{F} \mathbf{p}_m}$ and $\mathbf{r}_m = \mathbf{r}_{m-1} - \gamma_m \mathbf{F} \mathbf{p}_m$, from which $\mathbf{F} \mathbf{p}_m$ is interesting
2. $\mathbf{z}_{m-1} = \tilde{\mathbf{F}}^{-1} \mathbf{w}_{m-1}$

By expanding \mathbf{F} and $\tilde{\mathbf{F}}^{-1}$ according to eq. (3.64) and (3.100), respectively :

- $\mathbf{F} = \sum_{s=1}^{n_s} \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T$
- $\tilde{\mathbf{F}}^{-1} = \sum_{s=1}^{n_s} \mathbf{W}^s \mathbf{B}_b^s \left(\mathbf{K}_{bb}^s - (\mathbf{K}_{ib}^s)^T (\tilde{\mathbf{K}}_{ii}^s)^{-1} \mathbf{K}_{ib}^s \right) (\mathbf{B}_b^s)^T \mathbf{W}^s$

These matrices appear in each iteration. A natural process is to calculate the final matrices once before the iterative process and then keep reusing them. Due to the analysis of Section 5.8, its more efficient to perform the operations from right to left with the current iteration's vector. Even though this may initially seem that there are recalculation involved, the calculations are orders of magnitude less this way and a huge number of calculations would be required in order to reach the amount of calculations needed for performing the matrix-matrix multiplications as per the previous

idea. FETI exhibits fast convergence so performing the calculations as shown in Fig. 3.18 is much more efficient. Furthermore, intermediate dense matrices are avoided and the matrices \mathbf{F} and $\tilde{\mathbf{F}}^{-1}$ need not be explicitly formed.

Furthermore, note that matrices \mathbf{F} and $\tilde{\mathbf{F}}^{-1}$ comprise a sum of subdomain-level parts, which is another reason for not forming them: in each iteration, each subdomain performs calculations with the current vector and returns relevant result vectors. Eventually, only vector need to be summed in order to take the contributions of all subdomains into account.

3.4.4 Boolean matrices

In FETI features signed Boolean matrices \mathbf{B} (Section 3.2.1) which have only 2 entries per row: a +1 and a -1 ; all other entries are zero. This presents an opportunity to store \mathbf{B} very efficiently. Three alternative ways to store \mathbf{B} are presented and then the multiplication of \mathbf{B} with a vector directly from the special format is provided. The three variations are shown in Table 3.2.

Type	
A	Indexes only
B	Signed Indexes
C	Indexes and separate signs

Table 3.2: Storage formats for signed Boolean matrix \mathbf{B}

The storage formats will be presented with the help of the following example \mathbf{B} matrix with dimensions 4×10 :

$$\mathbf{B}^{(TearedDomain)} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & | & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 0 \\ 0 \\ 0 \end{matrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix} \end{matrix} \quad (3.105)$$

(4 boundary dof \times 10 total dof)

The matrix $\mathbf{B}^{(TearedDomain)}$ comprises subdomain parts \mathbf{B}^s where the first subdomain has degrees of freedom 1-5 and the second has degrees 6-10.

3.4.4.1 Type A: indexes only

In type A, the indexes of the two non-zero entries of each row are stored. The index corresponding to $+1$ is stored first and the one corresponding to -1 is stored second.

$$\mathbf{B}^{*(TearedDomain)} = \begin{bmatrix} 1 & 6 \\ 5 & 8 \\ 10 & 2 \\ 3 & 7 \end{bmatrix} \quad (3.106)$$

For the signed Boolean matrices of each subdomain, the indexes that pertain to the subdomain are stored. The column they are placed (left or right) depends on the sign of the represented entry. The other entry of the same row is flagged with a value that signifies that the corresponding index is not part of this subdomain. Below, the flag is 0 :

$$\mathbf{B}^{*(1)} = \begin{bmatrix} 1 & 0 \\ 5 & 0 \\ 0 & 2 \\ 3 & 0 \end{bmatrix} \quad \mathbf{B}^{*(2)} = \begin{bmatrix} 0 & 6 \\ 0 & 8 \\ 10 & 0 \\ 0 & 7 \end{bmatrix} \quad (3.107)$$

In all cases, the column of an entry in the Type A format (left or right) is important because it signifies the sign of the stored entry of \mathbf{B} . Variable types (Section 3.4.2) that are sufficient in this type are unsigned integers of enough range to cover the largest column index.

3.4.4.2 Type B: signed indexes

In this storage type, each stored column index is signed with the appropriate sign of the value represented. For the example of eq. (3.105):

$$\mathbf{B}^{*(TearedDomain)} = \begin{bmatrix} 1 & -6 \\ 5 & -8 \\ -2 & 10 \\ 3 & -7 \end{bmatrix} \text{ or } \mathbf{B}^{*(TearedDomain)} = \begin{bmatrix} 1 & -6 \\ 5 & -8 \\ 10 & -2 \\ 3 & -7 \end{bmatrix} \quad (3.108)$$

In contrast to Type A, the column of the entries (left or right) does not matter, which allows swapping the entries of a particular row. As a result, both representations of eq. (3.108) are valid.

The benefit here is on the subdomain level. Storing as in Type A would yield:

$$\mathbf{B}^{*(1)} = \begin{bmatrix} 1 & 0 \\ 5 & 0 \\ 0 & -2 \\ 3 & 0 \end{bmatrix} \quad \mathbf{B}^{*(2)} = \begin{bmatrix} 0 & -6 \\ 0 & -8 \\ 10 & 0 \\ 0 & -7 \end{bmatrix}$$

However, swaps are allowed, so all non-zeros (where 0 is the flag that signifies absence) can be placed in the same column:

$$\mathbf{B}^{*(1)} = \begin{bmatrix} 1 & 0 \\ 5 & 0 \\ -2 & 0 \\ 3 & 0 \end{bmatrix} \quad \mathbf{B}^{*(2)} = \begin{bmatrix} 0 & -6 \\ 0 & -8 \\ 0 & 10 \\ 0 & -7 \end{bmatrix}$$

Since an entire column is full of zeros, it can be discarded:

$$\mathbf{B}^{*(1)} = \begin{bmatrix} 1 \\ 5 \\ -2 \\ 3 \end{bmatrix} \quad \mathbf{B}^{*(2)} = \begin{bmatrix} -6 \\ -8 \\ 10 \\ -7 \end{bmatrix} \quad (3.109)$$

This allows the storage of subdomain-level \mathbf{B} matrices with half the entries compared to Type A.

Variable types (Section 3.4.2) that are sufficient in this type are integers of enough range to cover the largest column index.

3.4.4.3 Type C: Indexes and separate signs

Another way is to store the signs in a different array. For the example of eq. (3.105):

$$\mathbf{B}^{*(TearedDomain)} = \begin{bmatrix} 1 & 6 \\ 5 & 8 \\ 2 & 10 \\ 3 & 7 \end{bmatrix}, \quad \mathbf{Sign}^{*(TearedDomain)} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \quad (3.110)$$

Each entry has its sign stored in the corresponding position in the secondary matrix **Sign**. Swaps are allowed in this case as well as long as the relevant swaps are performed in the **Sign** matrix as well in order to preserve the invariants.

The subdomain matrices are:

$$\mathbf{B}^{*(1)} = \begin{bmatrix} 1 \\ 5 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{Sign}^{(1)} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{B}^{*(2)} = \begin{bmatrix} 6 \\ 8 \\ 10 \\ 7 \end{bmatrix}, \quad \mathbf{Sign}^{(2)} = \begin{bmatrix} -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \quad (3.111)$$

As a result, there are two arrays for each subdomain for Type C. The **Sign** matrix only takes values +1, -1 so it can be stored with a single bit per entry. Furthermore, the entries of **B** can be stored with unsigned integers of enough range to cover the largest column index. Note that the unsigned types can reach twice the maximum value of the signed types of the same bit size because the sign takes up one bit to store.

3.4.4.4 Matrix-vector multiplication for compact B

Matrix **B** is involved in matrix-vector multiplications and the multiplication is performed directly from the compact form of **B**. If a vector with all entries zero except a single entry equal to one is multiplied with a vector **x**.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = x_4 \quad (3.112)$$

The single entry is in the 4th column and it isolated the 4th entry of vector \mathbf{x} . More generally:

$$[0 \dots 0 \ 1 \ 0 \dots 0]_{\mathbf{x}} = x_i, \text{ where } i \text{ is the index of the "1" entry} \quad (3.113)$$

The single entry selects the entry of \mathbf{x} that is at in the index equal to its own index. Furthermore, if the single entry is equal to -1 it not only chooses the entry of \mathbf{x} but also applies a negative sign to it.

In the compact storage of \mathbf{B} , the column indexes of the $+1$ and -1 entries are stored. Therefore, by using these two entries from a row of \mathbf{B} it is possible to find which two entries would be selected if the multiplication was performed naively.

Let \mathbf{x} , \mathbf{w} be vectors of proper dimensions and \mathbf{B} an example signed Boolean Matrix. The $\mathbf{B}\mathbf{x}$ multiplication results in a vector of size equal to the number of rows of \mathbf{B} . Each row of \mathbf{B} contains exactly two entries. The $\mathbf{B}^T\mathbf{w}$ multiplication results in a vector of size equal to the number of columns of \mathbf{B} . The multiplications are shown in (eq. 3.114) and (eq. 3.115), respectively.

$$\mathbf{y} = \mathbf{B}^{(TearedDomain)} \mathbf{x} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix} \end{matrix} = \begin{matrix} \begin{bmatrix} x_1 - x_6 \\ x_5 - x_8 \\ -x_2 + x_{10} \\ x_3 - x_7 \end{bmatrix} \end{matrix} \quad (3.114)$$

$$\mathbf{v} = \left(\mathbf{B}^{(TearedDomain)} \right)^T \mathbf{w} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{matrix} \left[\begin{matrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{matrix} \right] \\ = \\ \begin{matrix} w_1 \\ -w_3 \\ w_4 \\ 0 \\ w_2 \\ -w_1 \\ -w_4 \\ -w_2 \\ 0 \\ w_3 \end{matrix} \end{matrix} \end{matrix} \quad (3.115)$$

The number of entries of each of a row of \mathbf{B}^T depends on the multiplicity of the corresponding degree of freedom (Section 3.2.6.1). In the example (eq. 3.115), there is only one non-zero entry per row. This is true for degrees that belong to exactly 2 subdomains, so a single equation-constraint is sufficient (Section 3.2.1). If a degree of freedom belongs to 4 subdomains and when using the fully redundant constraint scheme (Section 3.2.6.1.3), there will be three non-zero entries – as many as the equation-constraints pertaining to each such degree. This is shown from Fig. 3.19 because 3 arrows start/end from/to each instance of the center node. This details is relevant only for $\mathbf{B}^{(TearedDomain)}$ which is typically not needed. On the subdomain level signed Boolean matrices there is one entry per row.

As for every transpose, note that in the $\mathbf{B}^T \mathbf{w}$ multiplication, the transpose is not explicitly calculated.

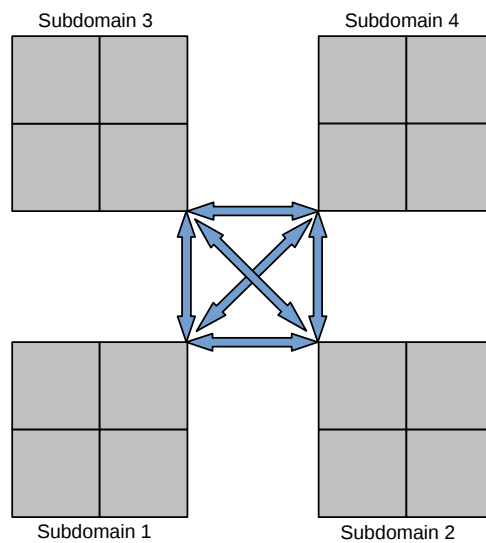


Fig. 3.19: Fully redundant constraints

3.4.4.4.1 Matrix-vector multiplication for Type A

For the multiplication $\mathbf{B}^{(TearedDomain)} \mathbf{x}$, comparing the compact \mathbf{B} (eq. 3.106) with the result \mathbf{y} (eq. 3.114) it is observed that the indexes remaining in \mathbf{y} are only those that are referenced in \mathbf{B} . Therefore, for entry i of the result vector (where $i=1,2,3,4$ in the example):

$$j_1=B(i, 1), j_2=B(i, 2) \quad \text{Get indexes } j_1, j_2 \text{ from row } i \text{ of matrix } \mathbf{B}$$

$$y(i)=x(j_1)-x(j_2) \quad \text{Choose entries } j_1, j_2 \text{ from } \mathbf{x}$$

The minus sign is because of the convention that the second column contains the negative entries.

For the multiplication $(\mathbf{B}^{(TearedDomain)})^T \mathbf{w}$, in the general case of arbitrary number of entries per row: Note that in the case of the transpose $i=1,2,3,4$ as well. Even though \mathbf{B}^T has 10 rows, the operations are logically different. Furthermore, as can be seen by the algorithm, there is no explicit transposition of \mathbf{B} .

$$j_1=B(i, 1), j_2=B(i, 2) \quad \text{Get indexes } j_1, j_2 \text{ from row } i \text{ of matrix } \mathbf{B}$$

$$v(j_1)+=w(i) \quad \text{Add/Subtract at entries } j_1, j_2 \text{ of the result}$$

$$v(j_2)-=w(i)$$

On the subdomain level (eq. 3.107), for the multiplication $\mathbf{B}^s \mathbf{x}$ the process is similar but either j_1 or j_2 will be zero (or whichever value is used for the flag). Thus:

$$j_1=B(i, 1), j_2=B(i, 2) \quad \text{Get indexes } j_1, j_2 \text{ from row } i \text{ of matrix } \mathbf{B}$$

If $j_1 == absenceFlag$ $y(i) = -x(j_2)$, else
 If $j_2 == absenceFlag$ $y(i) = x(j_1)$

For the multiplication $(\mathbf{B}^s)^T \mathbf{w}$ on the subdomain level, it is know that each row only has a single value. Therefore, the expressions are simplified:

$$j_1=B(i, 1), j_2=B(i, 2)$$

$$v(j_1)=w(i) \quad \text{Instead of } v(j_1)+=w(i) \Leftrightarrow v(j_1)=v(j_1)+w(i) \text{ of the domain-level}$$

$$v(j_2)=-w(i) \quad \text{Instead of } v(j_2)-=w(i) \Leftrightarrow v(j_2)=v(j_2)-w(i) \text{ of the domain-level}$$

3.4.4.4.2 Matrix-vector multiplication for Type B

For the multiplication $\mathbf{B}^{(TearedDomain)} \mathbf{x}$ (eq. 3.108) on the domain level:

$j_1= B(i,1) $, $j_2= B(i,2) $	Get indexes j_1, j_2 from \mathbf{B} . Absolute value
$sgn_1=sgn(i,1)$, $sgn_2=sgn(i,2)$	Get the sign of the stored entries
$y(i)=sgn_1 \cdot x(j_1)+sgn_2 \cdot x(j_2)$	Choose entries j_1, j_2 from \mathbf{x} and apply the sign

The absolute values is because the stored entries are not always positive. $sgn(x)$ is a function that extracts the sign of a number like $\frac{|x|}{x}$.

For the multiplication $(\mathbf{B}^{(TearedDomain)})^T \mathbf{w}$:

$j_1= B(i,1) $, $j_2= B(i,2) $	Get indexes j_1, j_2 from \mathbf{B} . Absolute value
$sgn_1=sgn(i,1)$, $sgn_2=sgn(i,2)$	Get the sign of the stored entries
$v(j_i)+=sgn_1 \cdot w(i)$	Add at entries j_1, j_2 of the result after applying the sign
$v(j_2)+=sgn_2 \cdot w(i)$	

On the subdomain level (eq. 3.109), there is only one column. Therefore the the expressions are simpler. For the multiplication $\mathbf{B}^s \mathbf{x}$:

$j= B(i) $	Get index j from \mathbf{B} . Absolute value
$sgn=sgn(i)$	Get the sign of the stored entry
$y(i)=sgn \cdot x(j)$	Choose entry j from \mathbf{x} and apply the sign

For the multiplication $(\mathbf{B}^s)^T \mathbf{w}$:

$j= B(i) $	Get index j from \mathbf{B} . Absolute value
$sgn=sgn(i)$	Get the sign of the stored entry
$v(j)=sgn \cdot w(i)$	Set index j of the result after applying the sign

3.4.4.4.3 Matrix-vector multiplication for Type C

For the multiplication $\mathbf{B}^{(TearedDomain)} \mathbf{x}$ (eq. 3.110) on the domain level:

$j_1 = B(i, 1), j_2 = B(i, 2)$	Get indexes j_1, j_2 from row i of matrix \mathbf{B}
$sgn_1 = Sign(i, 1), sgn_2 = Sign(i, 2)$	Get the signs from the Sign matrix
$y(i) = sgn_1 \cdot x(j_1) + sgn_2 \cdot x(j_2)$	Choose entries j_1, j_2 from \mathbf{x} and apply the sign

For the multiplication $(\mathbf{B}^{(TearedDomain)})^T \mathbf{w}$:

$j_1 = B(i, 1), j_2 = B(i, 2)$	Get indexes j_1, j_2 from row i of matrix \mathbf{B}
$sgn_1 = Sign(i, 1), sgn_2 = Sign(i, 2)$	Get the signs from the Sign matrix
$v(j_1) += sgn_1 \cdot w(i)$	Add at entries j_1, j_2 of the result after applying the sign
$v(j_2) += sgn_2 \cdot w(i)$	

On the subdomain level (eq. 3.111), there is only one column. Therefore the the formulas are simpler. For the multiplication $\mathbf{B}^s \mathbf{x}$:

$j = B(i)$	Get index j from \mathbf{B}
$sgn = Sign(i)$	Get the sign from the Sign matrix
$y(i) = sgn \cdot x(j)$	Choose entry j from \mathbf{x} and apply the sign

For the multiplication $(\mathbf{B}^s)^T \mathbf{w}$:

$j = B(i)$	Get index j from \mathbf{B}
$sgn = Sign(i)$	Get the sign from the Sign matrix
$v(j) = sgn \cdot w(i)$	Set index j of the result after applying the sign

3.4.4.5 Left multiplying vector with matrix for compact B

Section 3.4.4.4 describes the matrix-vector multiplication for efficiently stored \mathbf{B} : $\mathbf{B}\mathbf{x}$ and $\mathbf{B}^T\mathbf{w}$, where \mathbf{x} , \mathbf{w} vectors of consistent dimensions. Left multiplying, i.e. $\mathbf{w}^T\mathbf{B}$ and $\mathbf{x}^T\mathbf{B}^T$ is essentially the same (see Section 5.9). The following expression holds for any matrix vector:

$$(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^T \quad (3.116)$$

Therefore:

$$\begin{aligned} \mathbf{y} = \mathbf{B}\mathbf{x} &\Leftrightarrow \mathbf{y}^T = \mathbf{x}^T\mathbf{B}^T \\ \mathbf{v} = \mathbf{B}^T\mathbf{w} &\Leftrightarrow \mathbf{v}^T = \mathbf{w}^T\mathbf{B} \end{aligned} \quad (3.117)$$

Indeed:

$$\begin{aligned} \mathbf{x}^T\mathbf{B}^T &= \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \end{bmatrix} \begin{matrix} 1 & 2 & 3 & 4 \\ \left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array} \right| \end{matrix} \quad (3.118) \\ \Leftrightarrow \mathbf{x}^T\mathbf{B}^T &= \begin{bmatrix} x_1 - x_6 & x_5 - x_8 & -x_2 + x_{10} & x_3 - x_7 \end{bmatrix} = \mathbf{y}^T \quad (\text{see eq. 3.114}) \end{aligned}$$

$$\begin{aligned}
\mathbf{w}^T \mathbf{B} &= \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix} \end{matrix} \quad (3.119) \\
\Leftrightarrow \mathbf{w}^T \mathbf{B} &= \begin{bmatrix} w_1 & -w_3 & w_4 & 0 & w_2 & -w_1 & -w_4 & -w_2 & 0 & w_3 \end{bmatrix} = \begin{bmatrix} w_1 \\ -w_3 \\ w_4 \\ 0 \\ w_2 \\ -w_1 \\ -w_4 \\ -w_2 \\ 0 \\ w_3 \end{bmatrix}^T = \mathbf{v}^T \\
&\quad \text{(see eq. 3.115)}
\end{aligned}$$

By using the same expressions established in Section 3.4.4.4, the operations $\mathbf{w}^T \mathbf{B}$ and $\mathbf{x}^T \mathbf{B}^T$ can be performed. The only difference is that the result is a transposed vector but transposition in vectors is inconsequential (see Section 5.9).

3.4.4.6 Local to Global Mapping

In the example of eq. (3.105) presented so far, there are only two subdomains and the boundary belongs to both. In general, however, a subdomain is only participating in a small part of the global boundary problem. Therefore, all rows of matrix \mathbf{B}^s that pertain to boundary degrees of freedom not relevant to the current subdomain are filled with zeros. For example, for subdomain s :

$$\mathbf{B}^{*s} = \begin{bmatrix} \vdots & \vdots & \\ 1 & 0 & \text{row 37} \\ \vdots & \vdots & \\ 5 & 0 & \text{row 95} \\ \vdots & \vdots & \\ 0 & 2 & \text{row 123} \\ \vdots & \vdots & \\ 3 & 0 & \text{row 139} \\ \vdots & \vdots & \end{bmatrix} \quad (3.120)$$

In this case, it is more efficient to store only the part of the matrix relevant to the boundary of subdomain s . However, the row index of the entries is important because it signifies the identity of the boundary degree of freedom they refer to. As such, an auxiliary array is used to hold the initial row indexes of the entries. This is a very common technique in the context of sparse matrices (Section 5.6). The auxiliary array is named **map** here. Therefore:

$$\mathbf{B}_{local}^{*s} = \begin{bmatrix} 1 & 0 \\ 5 & 0 \\ 0 & 2 \\ 3 & 0 \end{bmatrix} \quad \mathbf{map}^s = \begin{bmatrix} 37 \\ 95 \\ 123 \\ 139 \end{bmatrix} \quad (3.121)$$

Matrix \mathbf{B} is essentially stored in the local boundary system of subdomain s . The **map** array contains the information needed to translate the local indexes to the global boundary indexes in order to be able to append the contributions of subdomain s with the rest of the subdomains.

For entry i of the auxiliary array:

$$k_i = \text{map}(i) \quad (3.122)$$

$$B_{global}^s(k_i) = B_{local}^s(i)$$

For example, the 3rd entry of eq. (3.121) yields:

$$k_3 = \text{map}(3) = 123$$

$$B_{global}^s(123) = B_{local}^s(3)$$

Calculations are usually performed on the subdomain level, so the rows that are not stored were full of zeros and would not affect the result anyway. The mapping leads to further efficiency in terms of

memory (no zeros stored) as well as operations (no calculations with zeros are performed).

3.4.5 Matrix \mathbf{W}

The matrix \mathbf{W} (see Section 3.3.1) is a diagonal matrix of size $n_b \times n_b$ where n_b is the size of the boundary problem (number of boundary degrees of freedom). Each (diagonal) entry is equal to the inverse multiplicity of the corresponding degree. The following is an example 9×9 matrix \mathbf{W} :

$$\mathbf{W} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \end{bmatrix} \quad (3.123)$$

Since the matrix is diagonal, the simplest storage format is to store the diagonal entries in an array (Section 5.4). Therefore:

$$\mathbf{w} = \left[\frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{2} \quad \frac{1}{2} \right]^T \quad (3.124)$$

with

$$\begin{aligned} W(i, i) &= w(i) \\ W(i, j) &= 0, \quad i \neq j \end{aligned} \quad (3.125)$$

This way, instead of storing a $n_b \times n_b$ matrix, only n_b are stored. However, there is still room for improvement. Matrix \mathbf{W} contains the inverse of the multiplicity of each degree, thus, it contains inverse values of integers. The multiplicity matrix \mathbf{M} can be stored instead:

$$\mathbf{M} = \mathbf{W}^{-1} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix} \quad (3.126)$$

and in diagonal storage:

$$\mathbf{m} = [2 \ 2 \ 2 \ 2 \ 4 \ 2 \ 4 \ 2 \ 2]^T \quad (3.127)$$

Storing the multiplicity allows the usage of integers instead of real numbers and this is generally cheaper. Not only is the multiplicity an integer, it is also a small integer. Using a single byte (8 bits) per value allows for numbers ranging from 0 to 255. The multiplicity is usually small enough to allow for even less than 8 bits per value. Real numbers require either 8 bytes/64 bits (double precision) or 4 bytes/32 bits (single precision) per value.

Another idea for further compactness is to take advantage of the fact that most boundary degrees have the same multiplicity, usually equal to 2, while a few of the degrees have multiplicity greater than 2. In this case, the multiplicity of all exception can be stored and anything not stored can be assumed to be equal to the default value 2. If the majority of the degrees has a different multiplicity, e.g. 4, then the default value can be set to 4 and any multiplicities not equal to 4 are stored.

Furthermore, the multiplicity matrix usually contains large consecutive sections with the same multiplicity. For example, there may be 100 consecutive degrees of freedom with multiplicity equal to 2, then a single node with multiplicity equal to 4 followed by 150 degrees with multiplicity 2. Thus, ranges of the same multiplicity can be stored:

- Multiplicity 2, from 0 to 100 (exclusive)
- Multiplicity 4, from 100 to 101 (exclusive)

– Multiplicity 2, from 101 to 250 (exclusive)

3.4.5.1 Matrix vector multiplication for compact \mathbf{W}

The multiplication here is fairly straightforward. A multiplication for the full form of \mathbf{W} is:

$$\mathbf{y} = \mathbf{W} \mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{5} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{6} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{7} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{9} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} x_1 \\ \frac{x_2}{2} \\ \frac{x_3}{3} \\ \frac{x_4}{4} \\ \frac{x_5}{5} \\ \frac{x_6}{6} \\ \frac{x_7}{7} \\ \frac{x_8}{8} \\ \frac{x_9}{9} \end{bmatrix} \quad (3.128)$$

$$y(i) = W(i, i) \cdot x(i) \quad (3.129)$$

For the diagonal storage of \mathbf{W} :

$$\mathbf{w} = \left[1 \quad \frac{1}{2} \quad \frac{1}{3} \quad \frac{1}{4} \quad \frac{1}{5} \quad \frac{1}{6} \quad \frac{1}{7} \quad \frac{1}{8} \quad \frac{1}{9} \right]^T$$

$$y(i) = w(i) \cdot x(i) \quad (3.130)$$

Eq. (3.129) and (3.130) are actually relevant to any diagonal matrix. If the multiplicity matrix is stored instead:

$$\mathbf{m} = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9]^T$$

$$y(i) = \frac{1}{m(i)} \cdot x(i) \quad (3.131)$$

4 Graphics Processing Units (GPUs)

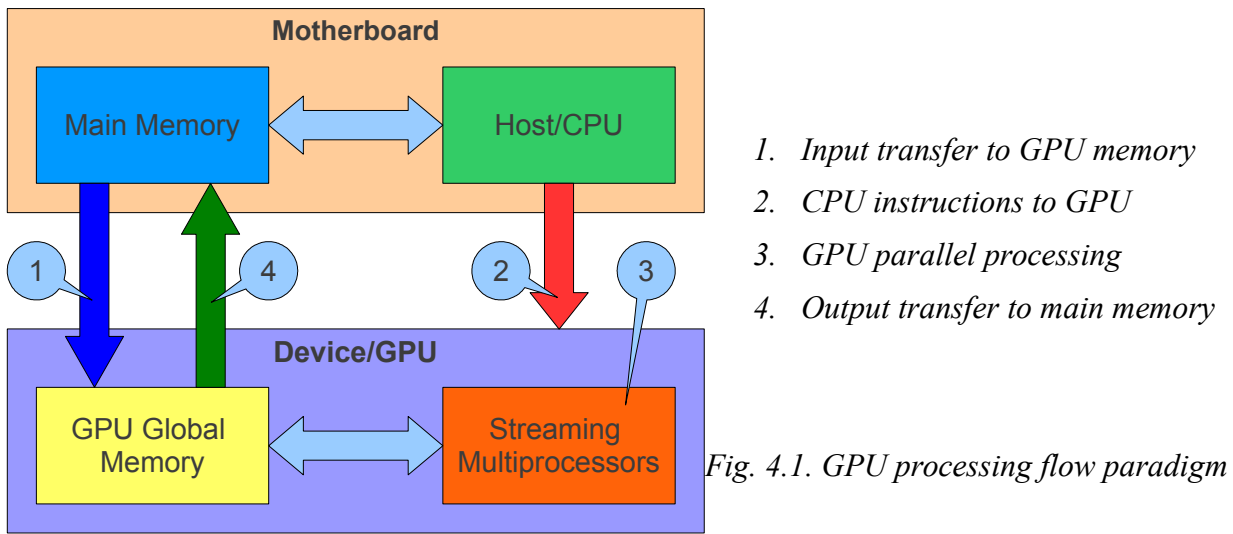
Applications of graphics processing units (GPUs) to scientific computations are attracting a lot of attention due to their low cost in conjunction with their inherently remarkable performance features. Parametric tests on 2D and 3D elasticity problems revealed the potential of the proposed approaches as a result of the exploitation of multi-core CPU hardware resources and the intrinsic software and hardware features of the GPUs.

Driven by the demands of the gaming industry, graphics hardware has substantially evolved over the years with remarkable floating point arithmetic performance. In the early years, these operations had to be programmed indirectly, by mapping them to graphic manipulations and using graphic libraries such as OpenGL and DirectX. This approach of solving general purpose problems is known as general purpose computing on GPUs (GPGPU). GPU programming was greatly facilitated with the initial release of the CUDA-SDK [48]–[50], which resulted in a rapid development of GPU computing and the appearance of GPU-powered clusters on the Top500 supercomputers [51]. Unlike CPUs, GPUs have an inherent parallel throughput architecture that focuses on executing many concurrent threads slowly, rather than executing a single thread very fast.

Work pertaining to GPUs has extended to a large spectrum of applications even before CUDA made their use easier. A number of studies in engineering applications have been recently reported on a variety of GPU platforms using implicit computational algorithms: in fluid mechanics [52]–[56], molecular dynamics [57], [58], topology optimization [59], wave propagation [60], Helmholtz problems [61], neurosurgical simulations [62]. Linear algebra applications have also been a topic of scientific interest for GPU implementations. Dense linear algebra algorithms are reported in [63], while a thorough analysis of algorithmic performance of basic linear algebra operations can be found in [64]. The performance of iterative solvers is analyzed in [65], and a parametric study of the PCG solver is performed on multi-GPU CUDA clusters in [66], [67]. A hybrid CPU-GPU implementation of domain decomposition methods is presented in [68] where speedups of the order of 40x have been achieved.

Graphics processing units (GPUs) are parallel devices of the SIMD (single instruction, multiple data) classification, which describes devices with multiple processing elements that perform the same operation on multiple data simultaneously and exploit data level parallelism. Programming in openCL or CUDA is easier than legacy general purpose computing on GPUs (GPGPU), since it only involves learning a few extensions to C and thus requiring no graphic-specific knowledge. In

openCL/CUDA context, the CPU is also referred to as a host and the GPU is also referred to as a device. The general processing flow of GPU programming is depicted in Fig. 4.1.



4.1 CPU vs GPU

The ratio of peak floating-point (double precision) calculation of GPUs is in the order of 1.5-3.0 TFLOPS versus 150 GFLOPS of CPUs. Despite the large peak-performance gap between many-threads GPUs and multicore CPUs, GPUs do not replace CPUs. Each one is suited for different tasks and one needs to use the best hardware for the task at hand to achieve high performance. CPUs and GPUs have fundamentally different design philosophies [49].

Two important terms for the discussion below are “latency” and “throughput”. Latency is the time needed to perform a task and is measured in units of time. Throughput is the number of such tasks performed per unit of time. For example, a company might need 12 hours to produce a car; this means that the latency is 12 hours. The company has several facilities that produce cars so it can produce 240 cars per day; this means that the throughput is 240 cars/day or 10 cars/hour.

The design of a CPU is latency-oriented and is optimized for sequential code performance i.e. to minimize the execution latency of a single thread. It employs sophisticated control logic to improve execution of a single thread and large cache memories to reduce data access latencies. However, the large cache memory, low-latency arithmetic units (ALUs) and sophisticated control

logic consume chip area (also power) that could be otherwise used to provide more arithmetic execution units and memory access channels.

On the other hand, the design of GPUs is throughput-oriented and it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially longer time to execute. The control logic is much simpler, meaning no (or at least very limited) branch prediction or data forwarding. There are long latency ALUs but this allows a large number of them to be on-chip. They are heavily pipelined for high throughput and are also more power efficient. Cache memories do exist but they are small and used to boost memory bandwidth. Memory bandwidth is an important issue because the speed of many applications is limited by the rate at which data can be delivered from the memory system into the processors. Graphics chips have been operating at approximately six times the memory bandwidth of contemporary available CPU chips. For example, the GTX 680 supports about 200 GB/s. Due to the fact that general-purpose processors have to satisfy legacy requirements, CPUs are expected to continue to be at a disadvantage in terms of memory bandwidth for some time [49]. Application software for throughput-oriented processors is expected to be written with a large number of parallel threads in mind in order to achieve good hardware utilization. The hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations.

For the above reasons, it should be clear that CPUs and GPUs are intended for different types of tasks and should be viewed as complementary co-processors. Winning applications should use both CPU and GPU: CPUs for sequential parts where latency is critical and GPUs for parallel parts where throughput prevails.

It should be also noted that all implementations prior to CUDA 1.3 are performed in single-precision, since support for double-precision floating point operation is added on CUDA 1.3. This has caused some misinterpretations in a number of published comparisons between the GPU and the CPU, usually in favor of the GPU.

4.2 CUDA and OpenCL

GPU implementations are typically based in either CUDA (Compute Unified Device Architecture) or OpenCL (Open Computing Language).

CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by their GPUs. It is currently the more popular of the two and as such learning material, documentation etc is more readily available for CUDA. However, it only runs in CUDA enabled devices. Kernels are written in CUDA C, which is essentially the C programming language with a few extensions to enable GPU implementations.

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of GPUS as well as CPUs, FPGA (field-programmable gate arrays) and other processors. As such, it is more portable but it should be noted that different hardware architecture may call for different optimizations. Differences between CUDA and openCL are explained in [69].

Kernels in this work can be written in both “languages” with very similar code. In this scope, the difference is mostly in the terminology used, as shown in Table 4.1, as well as API functions. When not explicitly mentioned otherwise, the terminology used in this text is the CUDA one. The pseudo-code, algorithm and general details are exactly the same in both CUDA and openCL.

CUDA	OpenCL
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory / registers	Private memory
<code>__syncthreads()</code>	<code>barrier()</code>
<code>blockDim</code>	<code>get_local_size()</code>
<code>blockIdx</code>	<code>get_group_id()</code>
<code>threadIdx</code>	<code>get_local_id</code>

Table 4.1: CUDA vs OpenCL terminology

4.3 GPU Hardware

GPUs have been rapidly evolving and as such their computing capabilities are changing. The GPUs used in this work are: NVIDIA GeForce GTX 580 and NVIDIA GeForce GTX 680. There were more GPUs tested, like the GTX 480 and AMD Radeon HD 7970, but the results presented are for the GTX 580 and GTX 680.

The GTX 580 is from the Fermi family and has Compute Capability 2.0. The GTX 680 is from the Kepler family and has Compute Capability 3.0. Illustration of the architectures is shown in Fig. 4.3, 4.4. The “Compute Capability” version is a way to succinctly describe a set of important properties and supported operations in the GPU. A summary of the most important properties for different Computing Capability is given in Table 4.2 and they are discussed in depth in the next sections. For an exhaustive listing of characteristics, see [70].

GPUs have a large number of streaming processors (SPs), which can collectively offer significantly more gigaflops than current high-end CPUs. The SPs are grouped together in streaming multiprocessors. Up until Fermi, they are denoted as SM and contain 32 cores each. Starting from Kepler they are denoted as SMX and contain 192 cores each (Fig. 4.2). In total, there are a lot more CUDA cores in GTX 680 than in GTX 580 (Fig. 4.3, Fig. 4.4) and it is increasing in newer GPUs like the GTX Titan (Table 4.3). The size of the global memory is also significantly increased in the latest generation. The current trend is that GPUs are improving at a fast pace.

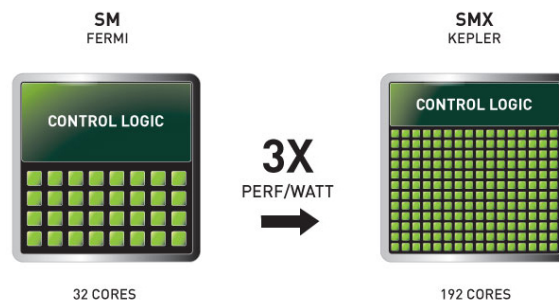


Fig. 4.2: SM vs SMX

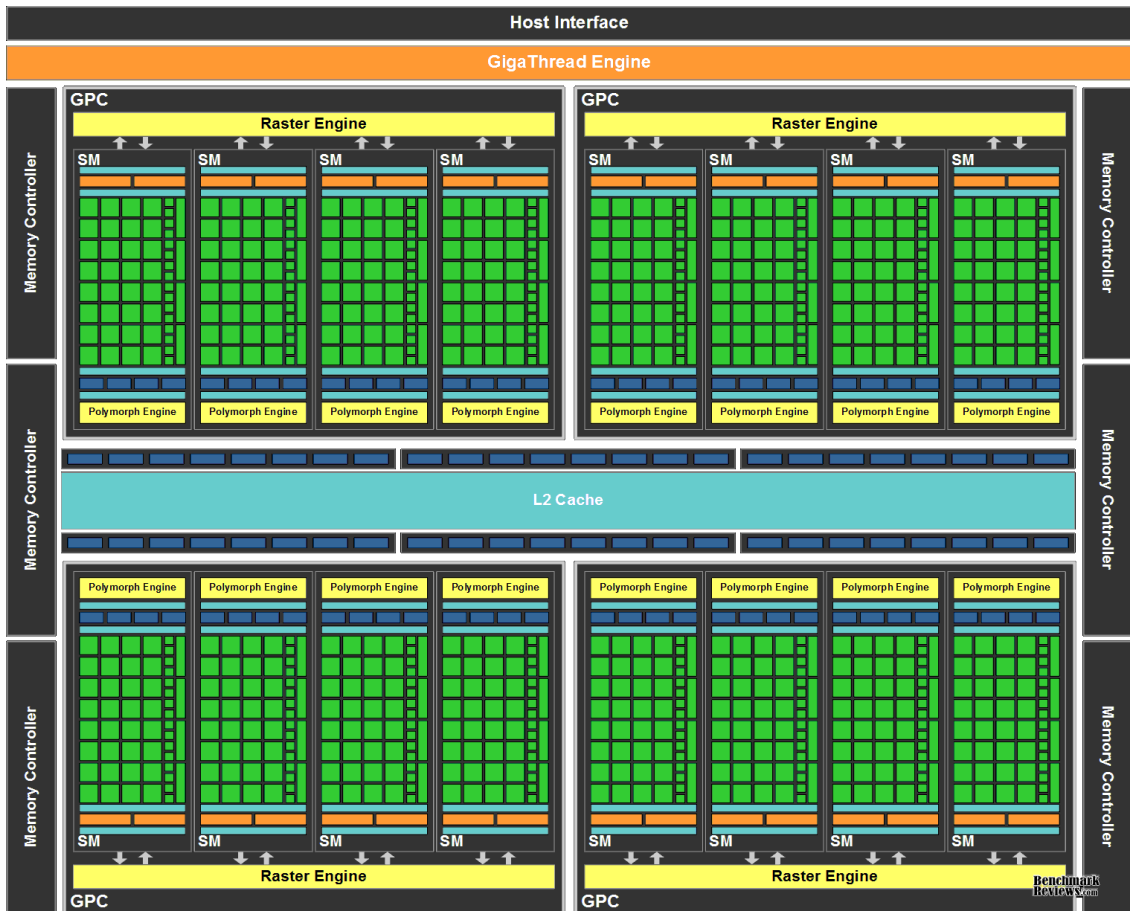


Fig. 4.3: NVIDIA Fermi (GF100) GPU block diagram



Fig. 4.4: NVIDIA Kepler (GK110) GPU block diagram

Technical specifications	Compute capability (version)						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum dimensionality of grid of thread blocks	2 (x,y)			3 (x,y,z)			
Maximum x-, y-, or z-dimension of a grid of thread blocks	2 ¹⁶ -1 (65535)					2 ³¹ -1 (2.1 bill)	
Maximum dimensionality of thread block	3 (x,y,z)						
Maximum dimensions of a block	512 x 512 x 64			1024 x 1024 x 64			
Maximum number of threads per block	512			1024			
Warp size (number of threads)	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24	32	48	64			
Maximum number of resident threads per multiprocessor	768	1024	1536	2048			
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K	64 K			
Maximum number of 32-bit registers per thread	128			63	255		
Maximum amount of shared memory per multiprocessor	16 KB			48 KB			
Constant memory size	64 KB						
Double precision	No			Yes			

Table 4.2: Important GPU properties

	GTX 580	GTX 680	GTX Titan
CUDA Cores	512	1536	2688
Core Clock	772MHz	1006MHz	837MHz
Memory Speed	4Gbps	6Gbps	6Gbps
Memory Interface	GDDR5	GDDR5	GDDR5
Memory Interface Width	384-bit	256-bit	384-bit
Global Memory Size	1.5GB	2GB	6GB
Memory Bandwidth	192.4GB/s	192.2GB/s	288.4GB/s
Compute Capability	2.0	3.0	3.5

Table 4.3: Specifications for GTX 580, GTX 680 and GTX Titan

4.4 GPU Threads

The GPU applies the same functions on a large number of data. These data-parallel functions are called kernels. Kernels generate a large number of threads in order to exploit data parallelism, hence the single instruction multiple thread (SIMT) paradigm. A thread is the smallest unit of processing that can be scheduled by an operating system. Threads in GPUs take very few clock cycles to generate and schedule due to the GPU's underlying hardware support, unlike CPUs where thousands of clock cycles are required. In a multi-threaded CPU, there is usually a small number of threads (4-12 threads). In the GPU however, the number of threads is at least in the thousands and usually much higher. Note that not all threads run at the same time but having large number of threads allows the GPU to schedule execution in a way that hides latency.

4.5 Thread Organization

All threads generated by a kernel define a grid and are organized in groups which are commonly referenced as thread blocks [in CUDA] or thread groups [in openCL]. A grid consists of a number of blocks (all equal in size), and each block consists of a number of threads (Fig. 4.5). Threads within the same block can cooperate with each other. Thread blocks are completely independent to other thread blocks which allows for flexible scheduling.

In general, a grid is a 3D array of blocks (after Compute capability 2.0, before it was 2D, see Table 4.2) and each block is a 3D array of threads. Fewer dimensions can be used by setting the unused dimensions to 1. There is a block index that shows the group-id of a thread block/group and it has up to three components $((x, y, z))$. Within a thread block/group, each thread has its own thread index which also has up to three components $((x, y, z))$. The indexes are symbolized as $blockIdx.(x, y, z)$ and $threadIdx.(x, y, z)$ and are used to assign different parts of the data for each thread. The dimensionality is intended to simplify index mapping (e.g. 2D for matrices) instead of always requiring linearized indexes. Fig. 4.6 shows a 2D grid with 2D blocks. The depicted “thread (1,2)” has $threadIdx.x=1$ and $threadIdx.y=2$ and since it belongs to “block (1,1)”, it has $blockIdx.x=1$ and $blockIdx.y=1$. These indexes are used to specify the parts of the data that will be processed by the thread.

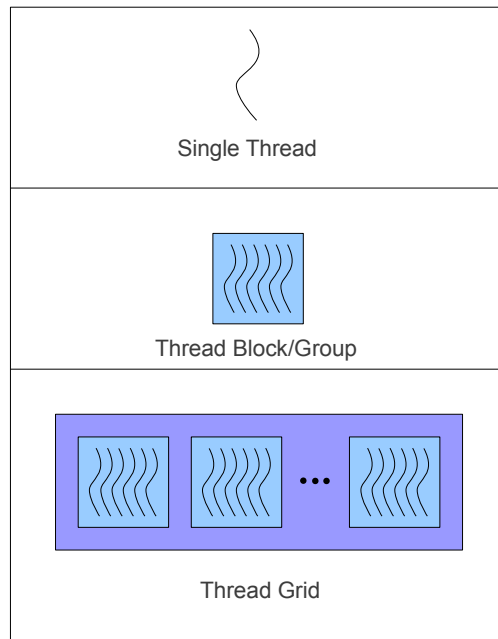


Fig. 4.5. Thread Organization

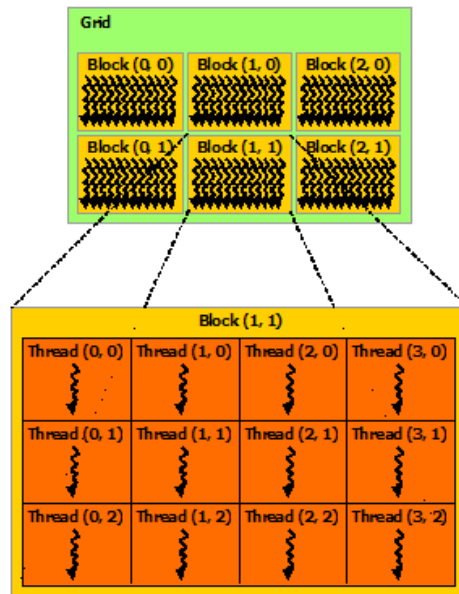


Fig. 4.6: 2D thread grid with 2D thread blocks.

4.6 Warps and control divergence

Thread blocks are partitioned into warps which are the scheduling units in the GPU. The number of threads in a warp is specific to the particular hardware implementation. This warp size is 32 for all known NVIDIA GPUs, as shown in Table 4.2. Threads within a warp are consecutive and have increasing indexes. Also, the split is one-dimensional even if the blocks are not. For example, a

16x16 thread will be partitioned into 8 warps. The order of the partition, which is important to keep in mind due to control divergence as discussed below, takes consecutive x thread indexes, then consecutive y thread indexes, then consecutive z thread indexes.

The purpose of warps is to ensure high hardware utilization through latency hiding (see Section 4.1). For example, if a warp initiates a long-latency operation and is waiting for results in order to continue, it is put on hold and another warp is selected for execution in order to avoid having idle processors while waiting for the operation to complete. When the long latency operation completes, the original warp will eventually resume execution. With a sufficient number of warps, the processors are likely to have a continuous workload in spite of the long-latency operations.

All threads of a particular warp have the same control flow instructions. This means that its possible to have control divergence when branching occurs and threads want to take different paths depending on some condition. Different such execution paths are serialized in current GPUs and this is undesirable so particular effort is made to reduce or remove control divergence when writing a kernel.

Divergence can arise only when a branch condition is a function of thread indexes. For the following example, lets assume a block has 512 threads. This is partition into 16 warps of 32 threads each. The following condition:

- `if (threadIdx.x > 2) {}`

creates two different control paths. The first is for threads 0, 1, 2 which fail the test and the other is for threads 3-511. In the first warp (threads 0-31) there are threads following different paths and there will be control divergence. However, for the other warps, all threads pass the test and there is no control divergence. An if-statement does not always imply control divergence. If all threads in a warp pass the condition or if all threads in a warp fail the condition, there is no divergence. For example the following condition:

- `if (threadIdx.x < 32) {}`

has no control divergence because all threads in the first warp pass the test while all other threads fail the test. Keep in mind that the size of the warp is not guaranteed and is subject to change, so the outcome may be different.

A condition such as:

- `if (threadIdx.x % 2 == 0){}`

which splits the threads into odd and even is particularly problematic because all warps will have control divergence. An example of thread divergence is shown as part of the reduction implementations in Section 4.12.

A related performance metric is occupancy, which is defined as the ratio of the active warps to the maximum allowed active warps in the multiprocessor. Resources are allocated by the GPU for the entire block and utilizing too many resources per thread may limit occupancy. Potential occupancy limiters are excessive usage of registers or shared memory (see Section 4.7).

4.7 Block size

The block size has certain restrictions and considerations that need to be taken into account. First of all there is a limit of 1024 threads in a block in recent GPUs or 512 in earlier ones, as shown in Table 4.2. Another constraint comes from the warps, as discussed in Section 4.6: thread block size should be a multiple of 32 [50].

There are more considerations dictated by the constraints of the streaming multiprocessors (SM/SMX) as shown in Table 4.2. There is a constraint in the number of blocks as well as the number of threads resident in a SM/SMX. For Compute capability 2.0, there can be up to 8 blocks in a SM/SMX and up to 1536 threads. Table 4.4 shows the total blocks and threads in a SM for all powers-of-two threads per block. For 32 threads per block, the restriction of 8 blocks in a SM means that only 256 threads will be assigned to a SM. This is much lower than the 1536 upper bound that is available and may lead to poor hardware utilization. For 1024 threads per block, only a single block can fit in the SM because another one would exceed the 1536 limit of the SM. This uses only 2/3 of the thread capacity. For 256 and 512 threads per block, we have full utilization and therefore these are good candidates for block size. For 2D blocks, a very commonly used size is 16x16 which has 256 threads.

Threads per block	Total blocks in SM	Total threads in SM
32	8	256
64	8	512
128	8	1024
256	6	1536
512	3	1536
1024	1	1024

Table 4.4: Block granularity considerations for Compute capability 2.0

4.8 GPU Memory

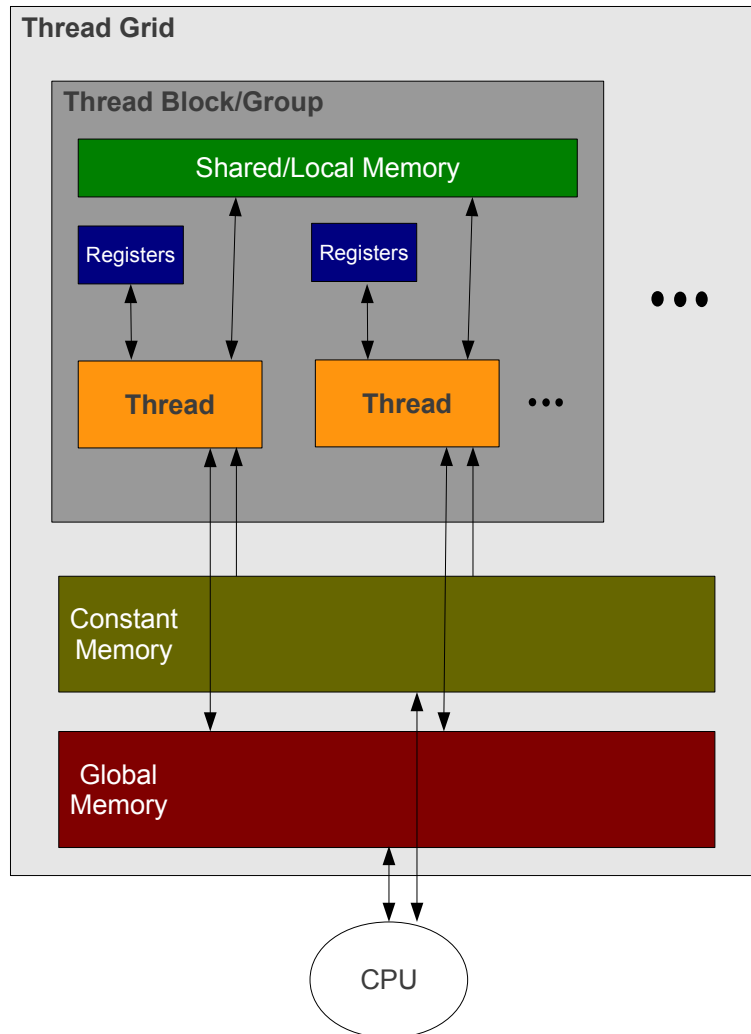


Fig. 4.7. Visual Representation of GPU Memory Model and Scope

GPGPU devices have a variety of different memories that can be utilized by programmers in order to achieve high performance. Fig. 4.7 shows a simplified representation of the different memories. The scope and lifetime of each type of memory is shown in Table 4.5, which is sorted by decreasing memory speed. Registers and shared memory are the fastest and the only two types of memory that actually reside on the GPU chip. Local, Global, Constant and Texture memory all reside off chip.

Memory	Scope	Lifetime
register	thread	thread
shared	block	block
constant	grid	application
texture	grid	application
local	thread	thread
global	grid	application

Table 4.5: Memory scope and lifetime

4.8.1 Global Memory

The global memory is the memory responsible for interaction with the host/CPU. The data to be processed by the device/GPU is first transferred from the host memory to the device global memory. Also, output data from the device needs to be placed here before being passed over to the host. Global memory is large in size (GB, see Table 4.3), off-chip and implemented with DRAMs. Data in this memory are visible by all threads of the entire grid.

The trade-off for its large size is that it is much slower than the other GPU memories, requiring hundreds of clock cycles to access (~400-600 clock cycles). Furthermore, the trend is that these memories keep becoming bigger but slower. Therefore one of the most important factors of CUDA kernel performance is accessing data in the global memory. In order to make up for the slow speed (and also due to the way DRAMs work), each time a memory location is accessed, many consecutive locations including the requested one are actually accessed. In Fig. 4.8 the areas marked with the same color are returned as a whole any time even a single one of their entries is requested. For example, if position 5 is requested, all entries from 4 to 7 will be returned. This is called DRAM bursting and modern DRAMs are designed to be always accessed in this manner. A typical burst size is 128 bytes. Any bytes not used are discarded, so it is best if all of them are used.

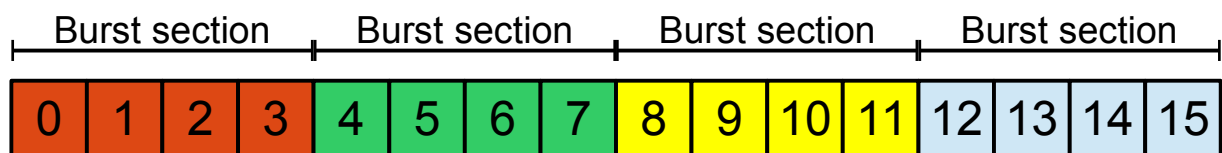


Fig. 4.8: DRAM burst example

When all threads of a warp load values in the same burst section, as in Fig. 4.9, only one DRAM request is made and the memory access is (fully) coalesced.

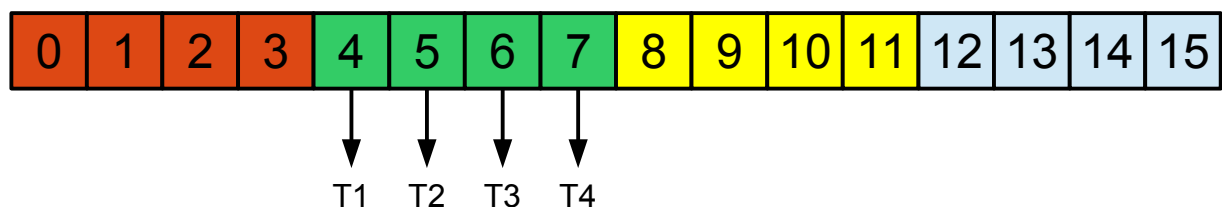


Fig. 4.9: Fully coalesced memory access

On the other hand, if threads in warp load values that are scattered across burst sections, as in Fig. 4.10, then four separate memory accesses will be made. Furthermore, only a small fraction of the bytes from each burst are actually need and the rest will be discarded.

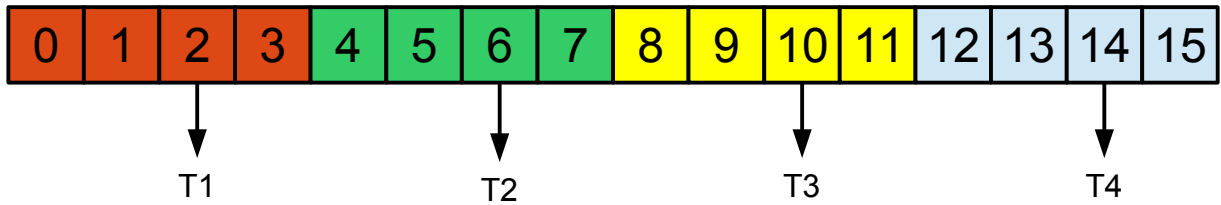


Fig. 4.10: Non coalesced memory access

Strided memory access, i.e. when threads do not access consecutive location results leads to non-coalesced access. Even if the stride is only 1, as in Fig. 4.11, the memory access is not fully coalesced and it multiple DRAM requests are made.

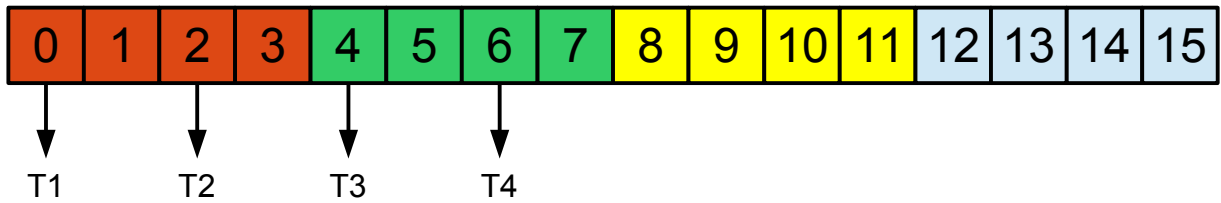


Fig. 4.11: Non coalesced strided memory access

Note that consecutive memory location accesses does not always guarantee coalesced reads. The accesses must also be aligned to the burst sections. Fig. 4.12 Shows an example of consecutive reads that span across different burst sections.

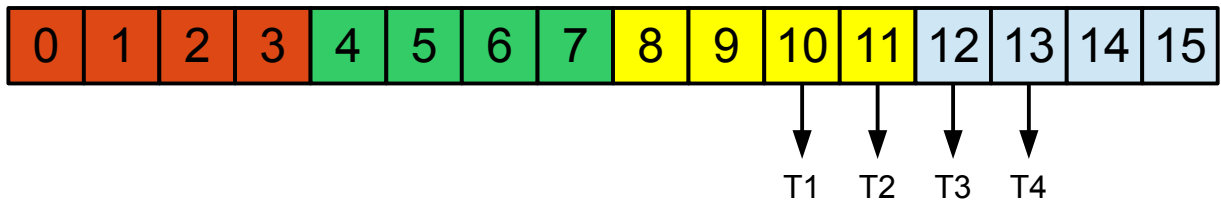


Fig. 4.12: Non coalesced memory access due to misalignment

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig. 4.13: A 4x4 matrix

Consider the 4×4 matrix of Fig. 4.13. Matrices in C-derivative languages (and consequently CUDA C) are stored in row-wise format (see Section 5.1.1), so the layout of the matrix in memory will be exactly the layout shown in Figs. 4.8-4.12. If threads of a warp access entries row-by-row, then the memory accesses are fully coalesced (Fig. 4.9). However, if threads of a warp access entries column-by-column, then the memory accesses are those shown in Fig. 4.10. The important pattern for coalesced access is not the access pattern of each thread, but the access pattern of neighboring threads in the same warp.

Consider a simple matrix-matrix multiplication $C = AB$ where each thread multiplies a row of A with a row of B to create an entry of C . Neighboring threads in the same warp access entries in A that are in the same column and entries in B that are in the same row. As a result, the access is coalesced in B but it is not coalesced in A . This is only a basic matrix-matrix multiplication scheme and better matrix-matrix multiplication kernels have coalesced accesses to both matrices among other improvements [49].

The speed of many applications is limited by the rate at which data can be delivered from the memory into the processors. Therefore, accesses to the slow global memory should be minimized by using the other memories where appropriate and the global memory accesses that are made should ideally be fully coalesced as possible.

4.8.2 Constant Memory

The constant memory also provides interaction with the host/CPU, but the device is only allowed to read from it and not write to it. The size of the constant memory is small (64KB in CUDA hardware, see Table 4.2) but it is aggressively cached and provides fast access for data needed by all threads of the grid. The constant memory is intended for small pieces of data that need to be globally accessed.

4.8.3 Shared Memory [CUDA] or Local Memory [OpenCL]

Shared memory is a form of scratch-pad memory designed to support efficient, high- bandwidth sharing of data among threads in a block. They are allocated to a thread block/group and allow all threads of the block/group to access them fast. Shared memories also allow cooperation between threads of the same block. A common technique when all threads of a block need access to a subset of the values is to have the threads cooperate in loading the values in the shared memory. Each thread loads one or more values and then all threads use them from the fast shared memory instead of each thread fetching the value from the global memory for itself.

The size of the shared memory is limited (16-48KB, see Table 4.2), so this creates an additional constraint on block sizes as well as the chunks of data handled by each thread block. Proper utilization of the shared memory to reduce global memory accesses is of critical importance to achieve great performance.

4.8.4 Registers

Any automatic variables (except arrays) are placed in the registers [CUDA] or private memories [openCL], which are the fastest memory types available in the GPU. Registers are thread-bound meaning that each thread can only access its own registers. Registers are typically used for holding variables that need to be accessed frequently but that do not need to be shared with other threads.

Registers are limited in number, though that number has increased significantly in the last generation (Table 4.2). Excessive use of them will cause spilling to the local memory (see Section 4.8.5).

4.8.5 Other memories

Texture memory is a read-only memory on the device. When all reads in a warp are physically adjacent, using texture memory can reduce memory traffic and increase performance compared to global memory. It is not used in this work but more information about texture memory can be found in [49].

Local memory [CUDA] is thread-local global memory so it is actually global memory and has the same performance. Any variable that cannot fit in the registers is “spilled” in the global memory

and this is referred to as local memory. Automatic variables that are large structures or arrays are also typically placed in local memory. The performance penalty of using the slow global memory when the intention was to use fast on-chip memories is heavy, though not as bad after Compute capability 2.0 when local memory became cached. Proper usage of the on-chip registers ensures that local memory is not used in this work.

Note that the term “private memory” in OpenCL means a memory that is bound to a single thread and appears to refer to both CUDA's local memory as well as registers.

4.8.6 Data transfer

A common bottleneck is encountered in data transfers between host and device. These transfers must pass through the peripheral component interconnect express (PCIe) bus which is commonly used to connect GPUs to the motherboard. The bottleneck is exacerbated on multi-GPU implementations because the GPUs cannot communicate directly (for CUDA applications) but only through the host. This implies multiple transfers through the PCIe bus. Thus, the measured speedup of an efficient parallel code based on message passing interface (MPI) could potentially decrease in a multi-GPU implementation. If the memory of the GPU is insufficient, then the data cannot be stored locally and it must be moved back and forth. It should be noted that GPU communication dramatically degrades performance, when computation count per communicating data is very low [49].

4.9 Synchronization

Thread within the same thread block can cooperate with each other by sharing values through the shared memory. Since threads may execute in any order, its important to have a mechanism that guarantees that all threads have finished the previous step of a calculation before moving on to the next one. For example, assume that each thread in the block loads a single value needed by all threads in the block. Before carrying on with calculations, it must be ensured that all threads have finished loading the values. This can be done with barrier synchronization which is a simple and widely used method for coordinating parallel activities. Barrier synchronization is performed with `__syncthreads()` in CUDA and `barrier()` in OpenCL (Table 4.1).

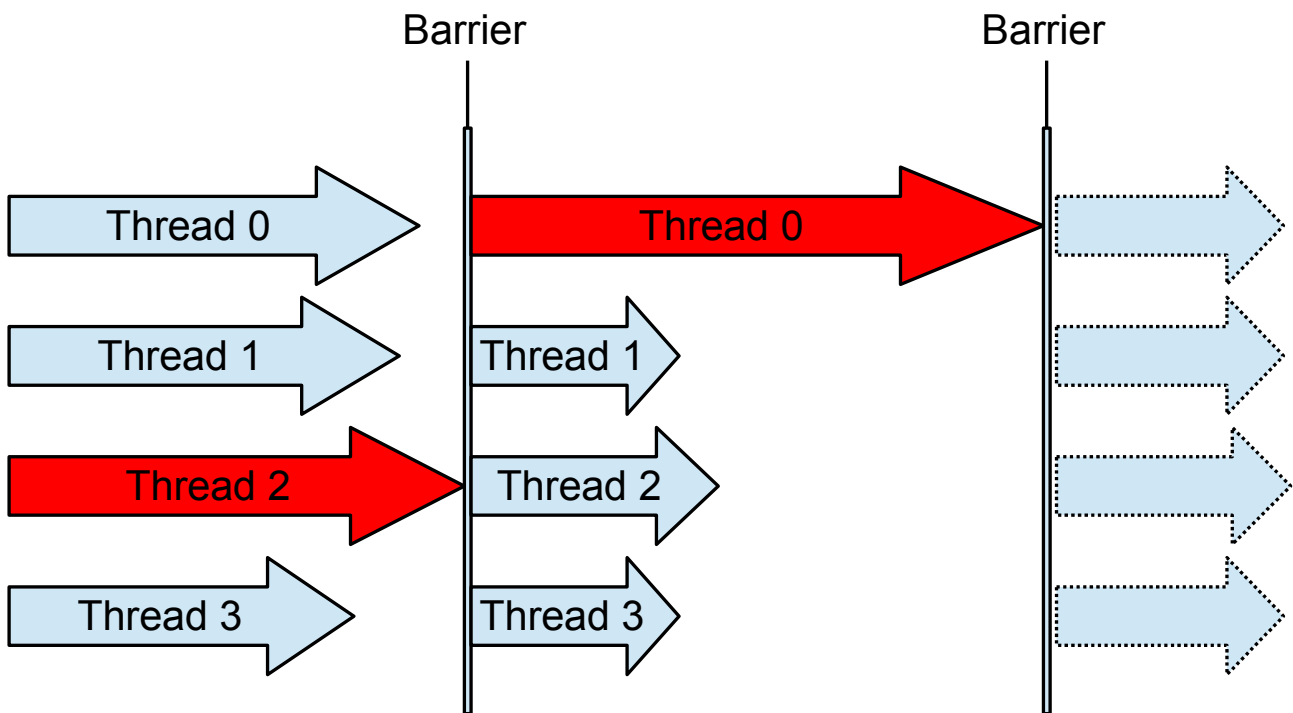


Fig. 4.14: Barrier synchronization

All threads must reach the synchronization point before being able to continue execution (Fig. 4.14). As a result, the slowest thread will delay execution for all thread of the block. Therefore, it is preferred that threads have approximately equal amount of work to do. In this case threads will reach the barrier at approximately the same time and there won't be much waiting. On the other hand, if a thread has more work to do than other threads (or if it is delayed), then other threads will have to wait for it resulting in underutilized resources, as is shown between the two barriers of Fig. 4.14.

Threads in different block cannot cooperate in the way outlined above. This is actually a design choice: by not allowing threads in different blocks to perform barrier synchronization with each other, blocks can be executed in any order relative to each other since none of them need to wait for any other block. This also enables transparent scalability which means the same implementation can run on any system without change and still benefit from the processing power of the system. For example, a small system may only be able to run 2 blocks at a time whereas a high-end system may be able to run 16 blocks at a time. Lack of synchronization constraints between blocks allow for the same code to run on both systems without extra development effort.

4.10 Privatization

When multiple threads write into an output location, it is useful to have partial and private output location for each thread or each thread block (in the registers or shared memory). This technique is called privatization and allows threads to do a large part of the work with no synchronization with other threads. For example, if a sum is being calculated then each thread may have its own partial sum. After each thread has finished its share of the data, then the partial results are summed (see also Section 4.12). Privatization also improves situations when atomic operations are required (see Section 4.11).

4.11 Atomic Operations

In a parallel environment there is the risk of race conditions. A simple case of race conditions is when multiple threads attempt to modify the same memory location. Threads need to read the current value, modify it and then write it again. These are three separate operations, even though it might not be visible from a simple statement like $a += 1$.

Assume two threads that are incrementing a memory location with initial value zero. The expected result is two and if the order of execution of the steps involved is either of the cases shown in Table 4.6 then the actual result will be two. However, if the order is like one of the cases in Table 4.7, then the result will be wrong. Case 3 in more detail goes like this: thread A reads value 0 and increments it to 1. Before it has a chance to write it, thread B reads the current value, which is still zero. Thread A writes its calculated value of 1. Thread B increments the value it read, which was 0, to 1 and then proceeds to write it to the result. Case 4 behaves similarly.

Order	Case 1		Case 2	
	Thread A	Thread B	Thread A	Thread B
1	Read			Read
2	Modify			Modify
3	Write			Write
4		Read	Read	
5		Modify	Modify	
6		Write	Write	

Table 4.6: Desirable order of execution

Order	Case 3		Case 4	
	Thread A	Thread B	Thread A	Thread B
1	Read			Read
2	Modify		Read	
3		Read		Modify
4	Write			Write
5		Modify	Modify	
6		Write	Write	

Table 4.7: Undesirable order of execution

To avoid such problems, there must be a way to guarantee that each thread finishes its read-modify-execute operation before another thread has a chance to access that particular memory location. This is done through atomic operations, which guarantee that either Case 1 or Case 2 of Table 4.6 will occur.

The pitfall of atomic operations is that it essentially serializes execution. Notice how in the previous example, Thread A executes before Thread B or vice versa. This is especially problematic when considering the particular characteristics of GPUs. First of all, GPUs are executing a massive number of threads and serializing a large number of them may severely limit performance. Furthermore, since atomic operations are serialized, latency determines throughput. GPUs are not latency-oriented processors but throughput-oriented processors (see Section 4.1) which means that individual operations are slow in favor of running many of them simultaneously, but the latter does not happen when atomic operations are involved. Then, there is the issue of slow global memory accesses (see Section 4.8.1). Each thread needs to perform both a read access and a write access. The total read-modify-write sequence may cost more than 1000 clock cycles and during this time other threads that want to access the same memory location are waiting.

The situation can sometimes be improved by privatization (Section 4.10). If the output is small and fits in the shared memory, then each thread block may have each own copy of the output. Any read-modify-write sequences are then done in the much faster shared memory. Also, threads in a block only have to wait for threads within the same block, not threads from other blocks. As a final step, individual copies are combined for the result. Even though this is much better, execution is still serialized.

For the above reasons, atomic operations should be avoided when possible. Sometimes the algorithm can be changed so that atomic operations are obviated entirely. For example, the scatter-to-gather transformation employed in the matrix assembly methods in this work (Section 8.3) completely avoids atomic operations.

4.12 Reduction

Reduction is an important parallel computation pattern and it is used to summarize a collection of input values into a single value. For example, finding the sum, maximum value or minimum value of an array are all reduction operations. More generally, reduction can be performed for any operations that is 1) associative 2) commutative and 3) has a well-defined identify value (e.g. 0 for sum, infinity for min etc).

Reduction can be a direct requirement of the calculation as is the case in the vector dot product. It can also be used as part of a parallel processing strategy. A commonly used strategy is to partition a large input into smaller chunks. Each thread processes a chunk and creates a partial results. The result is then summarized via reduction. Similarly, when privatization is used (Section 4.10), reduction can be used to collect all partial private results.

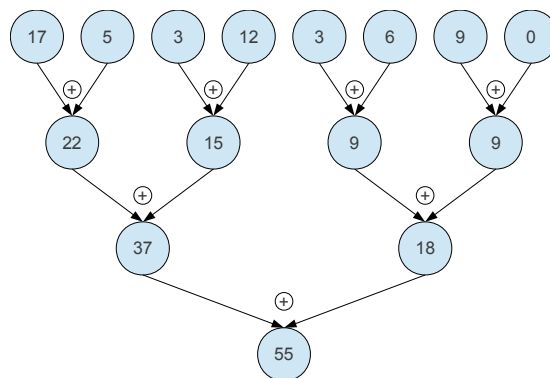


Fig. 4.15. Parallel reduction tree

On a sequential processor, reduction can be implemented very efficiently by iterating through the input and performing the reduction operation between the result value and the current value. Directly parallelizing this with atomic operations would create a serialization point and lead to very poor performance (Section 4.11). In order to efficiently perform reduction, a parallel reduction tree is used (Fig. 4.15). Each step divides the number of partial sums by half and ultimately produces the final sum after $\log_2 N$ steps. Parallel reduction trees are common but due to the special characteristics of the GPU, there are certain GPU-specific choices that need to be made.

Initially, each block loads the part of the values assigned to it in the shared memory. From then, all work is done within the shared memory and no interaction with the global memory until the result of the block is obtained. Each thread loads a value to the shared memory.

For a straightforward reduction kernel, each thread can sum two neighboring values, as shown in

Fig. 4.16. In each step, there are fewer and fewer threads working starting from multiples of 2, then 4, then 8 etc, doubling in each iteration. In each step, the active threads sum their value with a value that is located “stride” away from them. The stride is doubled in each iteration as well. Also, there needs to be barrier synchronization between the steps to ensure that all threads have properly updated their values before the next iteration starts reading them.

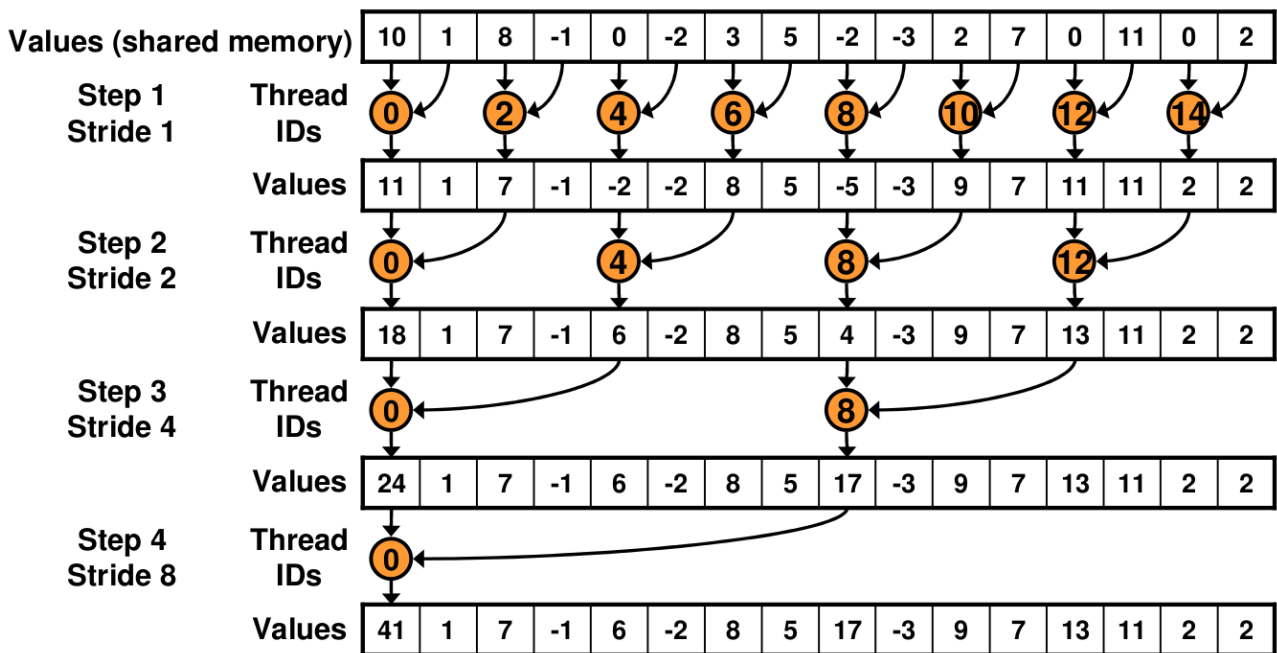


Fig. 4.16: Reduction pattern with thread divergence

While this reduction implementation would work well, there is a better approach for the GPU. The problem with the basic reduction is that there is a lot of control divergence (see Section 4.6). For example, in the first step odd threads are disabled but due to the way warps work there is no execution gain by having them disabled, it is as if they performed the calculations anyway. This is wasteful and it gets worse after a few iterations where entire warps will have only one active thread.

Therefore, the values that are added by each thread can be modified to accommodate this behavior of the GPU. The idea is to try to gather the active threads to the front. Each thread adds its value with a value that comes after all other active threads, as is depicted in Fig. 4.17. The number of active threads keeps dividing but there will be no divergence: threads in a warp will be either all active or all inactive. This continues up until the last 32 values, where only 1 warp is left. After that point, there is some divergence but it is very limited since all other warps are inactive and have no divergence. The improved reduction is in the order of 4.7 times faster than the divergent version [71].

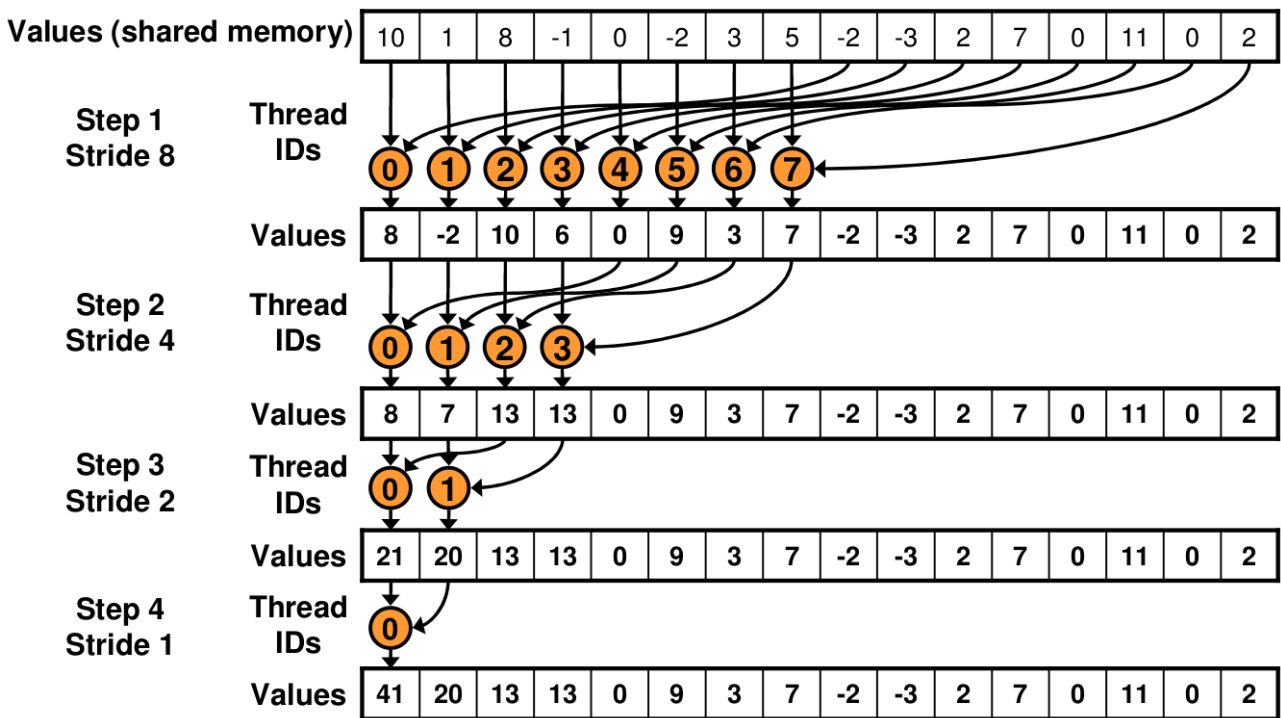


Fig. 4.17: Good reduction pattern

Note that due to the thread grid hierarchy (i.e. thread blocks are used and they do not cooperate) employed in the GPU, there will not be a single result but one result from each block. The values are orders of magnitude fewer than in the initial reduction, but they need to be reduced further to reach the final, single value. For example, if there are $1024 \cdot 1024$ values to be reduced, a thread grid with 1024 blocks of 1024 threads each can be launched (sizes within proper constraints, see Section 4.7). After execution there will be 1024 values, which will need to be reduced to reach the desired reduction result. If the number of intermediate values is still large, the GPU reduction can be applied recursively. When the values are fewer than a certain threshold, it's better to return them to the CPU and let it finish the calculation. This is applicable if the result of the reduction is needed in the CPU. If the result is needed in the GPU, the same reduction kernel can be launched again with a single thread block which will produce the final result of the reduction. Another alternative is to use atomic operations, specifically `atomicAdd()`. This eliminates the need for additional temporary arrays and repeated kernel launches.

The reduction presented above pertains to reducing scalar values. In order to reduce partial vectors into a single vector, a similar process is performed but each thread sums two vectors instead of two values in every step. A very thorough analysis of the GPU reduction implementation with more optimizations can be found in [71].

4.13 Pinned Memory

In order to discuss pinned memory, some background information are needed about virtual memory in modern computers [72]. Virtual memory maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. The primary benefits of virtual memory include obviating the need to manage a shared memory space on the application level, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging [73]. The last benefit means that not all data always reside in the physical memory. Each virtual address space is divided into pages when mapped into physical memory. These pages can be paged out to a secondary memory in order to make room in the main memory and paged in when required. Whether or not a particular piece of data resides in the physical memory is checked at address translation time.

A CPU-GPU data transfer is performed by DMA (Direct Memory Access) hardware for better efficiency. The DMA is responsible for data transfers, freeing the CPU for other tasks. The CPU gives the proper instructions (source, destination and how many bytes to transfer) to initialize the data transfer and then moves on to other tasks. The data is transferred over the systems interconnect, typically PCIe in today's systems (Fig. 4.18).

The DMA uses physical addresses and when a CPU-GPU data transfer is requested, the transfer is implemented with one or more DMA transfers. At the beginning of each transfer, the address is checked and it is made sure that the corresponding memory page is present in the physical memory. However, there are no further checks in the rest of the same DMA transfer so that high efficiency can be achieved. This means that the operating systems could page-out data that is being read or written by a DMA and page-in another virtual page into the same physical location. This is dangerous because the integrity of the data is destroyed.

Pinned memory is specifically marked so that its virtual memory pages cannot be paged out. CPU memory that is the source or destination of a DMA transfer must be allocated as pinned memory to avoid potential problems. In the GPU there is no paging, so this is not applicable; the problem is only on the CPU side. If a source or destination of a CPU-GPU data transfer (in the host memory side) is not allocated in pinned memory, it needs to be copied to pinned memory first.

This is extra overhead which can be avoided by explicitly using pinned memory for data that is intended to be transferred to/from the GPU. There are special commands to allocate pinned memory

and for CUDA this is done with `cudaHostAlloc()`. By using pinned memory, the memory transfers should be about twice as fast because there will not be an extra copy from non-pinned memory to pinned memory. However, it should be noted that pinned memory is a limited resource. Using too much will severely limit the operating system's ability to properly manage virtual memory and this can drastically reduce the overall system performance: the operating system will have a very small amount of available memory and it will constantly be paging in and out to accommodate requests.

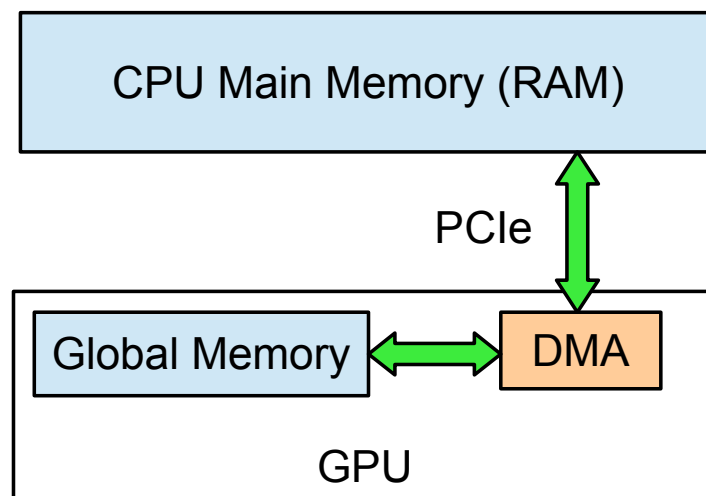


Fig. 4.18: Data transfer between CPU and GPU

4.14 GPU Task Parallelism

Apart from the data parallelism in the GPU discussed so far, there is opportunity for task parallelism. Contemporary GPU hardware can perform data transfers and calculations simultaneously and independently. Furthermore, the PCIe bus can make simultaneous transfer both ways, i.e. transfer data from the host to the device while also simultaneously transferring data from the device to the host. The situation without task parallelism is shown in Fig. 4.19. In order to be able to utilize task parallelism, the input can be divided into segments. The segments are processed in a way that allows the overlap of transfer and computation of adjacent segments. This situation with task parallelism is shown in Fig. 4.20.

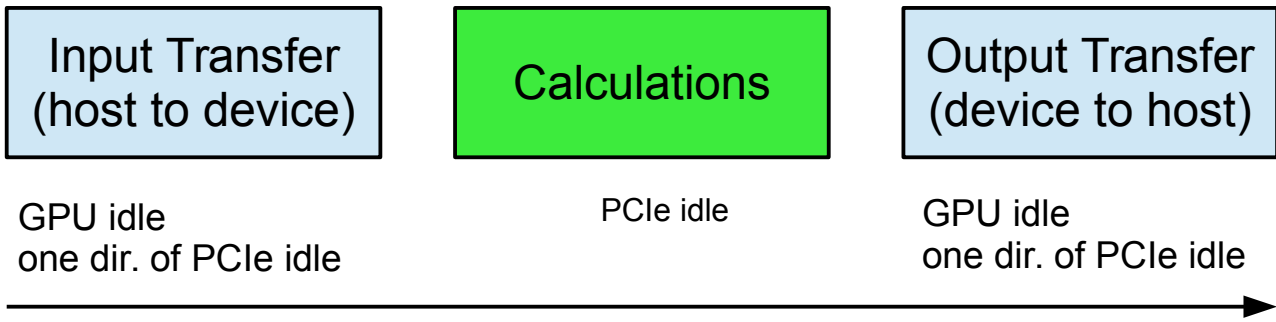


Fig. 4.19: Without task parallelism (*dir.* = direction)

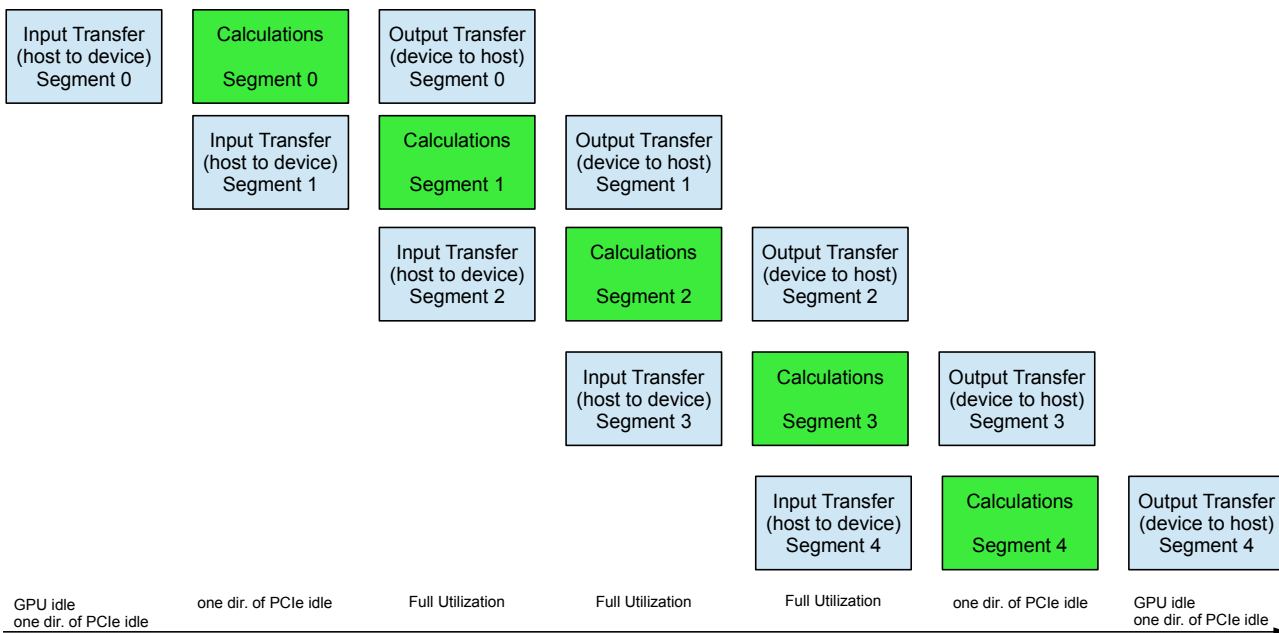


Fig. 4.20: With task parallelism (*dir.* = direction)

At the beginning and end of the process there will still be some idling of the GPU and/or PCIe but with enough segments the GPU can essentially be fully utilized at all times. The actual benefit from this overlap depends on the relative time between transfers and calculations. Figs. 4.19, 4.20 assume that the time is about equal, but the actual ratio is application dependent.

Task parallelism is achieved in CUDA by using streams. Each stream is a queue of transfers and GPU computations and the streams are independent of each other, meaning that tasks in different streams can be performed in parallel. The driver ensures that the commands within the same queue are processed in sequence, meaning that the calculations for a specific segment will not start before the input has successfully arrived nor will the transfer back be performed before the calculations have finished. By using multiple streams, a hardware utilization like the one in Fig. 4.20 can be achieved.

5 Handling of matrices

Matrix storage and matrix operations are important performance factors for large scale simulations. There is a large variety of different formats, all with different strengths and weaknesses. For example, the dense format is the most flexible format and supports any kind of operation but requires the most storage. Sparse formats store the minimum possible entries of the matrix, but have higher indexing cost and can only perform a specific subset of operations efficiently. As such, the choice of an appropriate format for the task at hand may significantly affect performance. This section presents matrix formats that are commonly used in simulations along with appropriate storage schemes and implementation considerations.

Note the distinction between a dense/banded/sparse matrix and a dense/banded/sparse storage scheme. The former is a characteristic of the matrix while the latter refers to the way it is stored. Any matrix can be stored with any storage scheme, even though it might not be the most efficient way. For example, a sparse matrix can be stored with a dense format and a banded matrix can be stored in sparse format. Sometimes the choices are dictated by the task at hand: if complete factorization is required, then the skyline format is favored over a sparse format. Conversely, if a sparse solver is employed, a sparse format is favored. The matrix is the same in both cases, but the storage scheme differs in order to accommodate particular needs.

5.1 Dense Matrix

When a matrix with m rows and n columns is stored in dense format, all $m \times n$ entries of the matrix are stored. If the matrix is square and has order n , all $n \times n$ are stored. This is the most generic format, has fast indexing and supports all operations. However, the space complexity is $O(mn)$ or $O(n^2)$. Thus, it should only be used for small matrices and when the represented matrix is actually full of non-zeros, like local stiffness matrices of finite elements.

Note that some programming languages use zero-based numbering, i.e. the indexes of an array of size n range from 0 to $n-1$, whereas others use one-based numbering, i.e. the indexes of an array of size n range from 1 to n . C-derivatives (C/C++, Java, C#) use zero-based numbering whereas FORTRAN, MATLAB, Octave, Scilab are one-based [74]. In the following sections, formulas are given in both formats.

5.1.1 Row-major & column-major entry order

Row-major order and column-major order describe methods for storing multidimensional arrays in linear memory. Since a matrix is two-dimensional, there are different implementations regarding the order of the entries. Row-major order is used in C-derivatives (C/C++, Java, C#) while column-major order is used in FORTRAN, MATLAB, Octave, Scilab. As a result, entry layout is important for correctly communicating with programs/libraries written with a specific layout in mind.

The layout is also important for performance because each layout is more appropriate for different operations: the row-major layout is suited for row-oriented operations whereas the column-major layout is more suited for column-oriented operations [75]. This stems from the fact that accessing memory in a contiguous manner is usually faster than accessing scattered memory entries. As an example, different implementations are needed for row-major and column major orders when performing a matrix-vector multiplication. For a matrix-matrix multiplication, the order of both matrices needs to be taken into account. Sometimes it is unclear which one is better and it may be worthwhile to experiment in order to find out what is faster for a particular application.

5.1.2 Implementations

There are several implementations for dense matrices. They differ in the entry order (row-major, column-major) and whether the storage is in a single array or one array per row or column.

5.1.2.1 Row-major, 2D array

One-based format				Zero-based format													
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8	0	0	1	2	3	4	5	6	7
2	9	10	11	12	13	14	15	16	1	8	9	10	11	12	13	14	15
3	17	18	19	20	21	22	23	24	2	16	17	18	19	20	21	22	23
4	25	26	27	28	29	30	31	32	3	24	25	26	27	28	29	30	31
5	33	34	35	36	37	38	39	40	4	32	33	34	35	36	37	38	39
6	41	42	43	44	45	46	47	48	5	40	41	42	43	44	45	46	47

Fig. 5.1: Row-major dense storage of a $[6 \times 8]$ matrix

The matrix is represented as a 2D array and each 1D array contains a whole row of size equal to the number of columns.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \end{bmatrix} \quad (5.1)$$

5.1.2.2 Row-major, 1D array

The matrix is represented as a single 1D array containing all the entries row-by-row.

$$A=[1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 44 \ 45 \ 46 \ 47 \ 48] \quad (5.2)$$

For one-based format, the index of an entry $[row, column]$ in the 1D array is given by:

$$index_1 = (row - 1) \cdot columnCount + column \quad (5.3)$$

$$index_{diagonal} = row \cdot (columnCount + 1) - columnCount \quad (5.4)$$

where $columnCount$ is the number of columns.

For zero-based format, the above relations are:

$$index_0 = (row \cdot columnCount) + column \quad (5.5)$$

$$index_{diagonal} = row \cdot (columnCount + 1) \quad (5.6)$$

Note that the number of rows does not appear in the relations - it is still needed, however, for index range checking.

5.1.2.3 Column-major, 2D array

One-based format				Zero-based format													
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1	7	13	19	25	31	37	43	0	0	6	12	18	24	30	36	42
2	2	8	14	20	26	32	38	44	1	1	7	13	19	25	31	37	43
3	3	9	15	21	27	33	39	45	2	2	8	14	20	26	32	38	44
4	4	10	16	22	28	34	40	46	3	3	9	15	21	27	33	39	45
5	5	11	17	23	29	35	41	47	4	4	10	16	22	28	34	40	46
6	6	12	18	24	30	36	42	48	5	5	11	17	23	29	35	41	47

Fig. 5.2: Column-major dense storage of a $[6 \times 8]$ matrix

The matrix is represented as a 2D array and each 1D array contains a whole column of size equal to the number of rows.

$$A = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{bmatrix} \begin{bmatrix} 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \end{bmatrix} \begin{bmatrix} 19 \\ 20 \\ 21 \\ 22 \\ 23 \\ 24 \end{bmatrix} \begin{bmatrix} 25 \\ 26 \\ 27 \\ 28 \\ 29 \\ 30 \end{bmatrix} \begin{bmatrix} 31 \\ 32 \\ 33 \\ 34 \\ 35 \\ 36 \end{bmatrix} \begin{bmatrix} 37 \\ 38 \\ 39 \\ 40 \\ 41 \\ 42 \end{bmatrix} \begin{bmatrix} 43 \\ 44 \\ 45 \\ 46 \\ 47 \\ 48 \end{bmatrix} \quad (5.7)$$

5.1.2.4 Column-major, 1D array

The matrix is represented as a single 1D array containing all the entries column-by-column.

$$A=[1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 44 \ 45 \ 46 \ 47 \ 48] \quad (5.8)$$

For one-based format, the index of an entry $[row, column]$ in the 1D array is given by:

$$index_1 = (column - 1) \cdot rowCount + row \quad (5.9)$$

$$index_{diagonal} = row \cdot (rowCount + 1) - rowCount \quad (5.10)$$

where $rowCount$ is the number of rows.

For zero-based format, the above relations are:

$$index_0 = (column \cdot rowCount) + row \quad (5.11)$$

$$index_{diagonal} = row \cdot (rowCount + 1) \quad (5.12)$$

Note that the number of columns does not appear in the relations - it is still needed, however, for index range checking.

5.2 Triangular Dense Matrix

Triangular matrices have one triangle of the matrix full of zeros: upper-triangle matrices only have non-zero entries on the upper-triangle while lower-triangle matrices only have non-zero entries on the lower-triangle. These typically include the diagonal as well, but there are strictly upper/lower variants that exclude it. Triangular dense matrices store $\frac{n(n+1)}{2}$ entries which are approximately half of that of a generic dense matrix, but still $O(n^2)$. Furthermore, less calculations need to be performed because operations with zeros can be avoided. Triangular systems of equations are easily solvable and occur as part of a solver that employs factorization either as the primary solution strategy or as part of solving subdomains.

The relevant triangular part of the matrix can be stored row-by-row or column-by-column. The column-by-column storage is also referenced as “packed storage”.

5.2.1 Implementations

Row-by-row and column-by column variants are demonstrated below. For both lower and upper triangle, one of the two variants has simple indexing, while the other is more complex. For the latter, alternative formulas are provided as well.

5.2.1.1 Lower triangular dense storage by row

One-based format								Zero-based format									
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1								0	0							
2	2	3							1	1	2						
3	4	5	6						2	3	4	5					
4	7	8	9	10					3	6	7	8	9				
5	11	12	13	14	15				4	10	11	12	13	14			
6	16	17	18	19	20	21			5	15	16	17	18	19	20		
7	22	23	24	25	26	27	28		6	21	22	23	24	25	26	27	
8	29	30	31	32	33	34	35	36	7	28	29	30	31	32	33	34	35

Fig. 5.3: Storage by row of an $[8 \times 8]$ lower triangular matrix

The matrix is represented as a 1D array containing all lower-triangle entries row-by-row.

$$A = [1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 32 \ 33 \ 34 \ 35 \ 36] \quad (5.13)$$

For one-based format, the index of a lower triangular entry $[row, column]$ in the 1D array is given by:

$$index_1 = column + \frac{row(row-1)}{2} \quad (5.14)$$

$row \geq column$

$$index_1 = 0 \quad (5.15)$$

$row < column$

$$index_1 = \frac{row(row+1)}{2} \quad (5.16)$$

$diagonal$

For zero-based format, the above relations are:

$$index_0 = column + \frac{row(row+1)}{2} \quad (5.17)$$

$$\underset{row < column}{index_0} = 0 \quad (5.18)$$

$$\underset{diagonal}{index_0} = \frac{row(row+3)}{2} \quad (5.19)$$

Note that the order of the matrix does not appear in the relations - it is still needed, however, for index range checking.

5.2.1.2 Lower triangular dense storage by column

One-based format								Zero-based format									
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1								0	0							
2	2	9							1	1	8						
3	3	10	16						2	2	9	15					
4	4	11	17	22					3	3	10	16	21				
5	5	12	18	23	27				4	4	11	17	22	26			
6	6	13	19	24	28	31			5	5	12	18	23	27	30		
7	7	14	20	25	29	32	34		6	6	13	19	24	28	31	33	
8	8	15	21	26	30	33	35	36	7	7	14	20	25	29	32	34	35

Fig. 5.4: Storage by column of an $[8 \times 8]$ lower triangular matrix

In this storage, also referenced as lower packed storage, the matrix is represented as a 1D array containing all lower triangle entries column-by-column.

$$A = [1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 32 \ 33 \ 34 \ 35 \ 36] \quad (5.20)$$

For one-based format, the index of a lower triangular entry $[row, column]$ in the 1D array is given by:

$$index_1 = \underset{row \geq column}{row} + \frac{(2 \ order - column)(column - 1)}{2} \quad (5.21)$$

$$index_1 = 0 \quad \underset{row < column}$$

$$index_1 = \underset{diagonal}{row} + \frac{(2 \ order - row)(row - 1)}{2} \quad (5.22)$$

For zero-based format, the above relations are:

$$index_0 = row + \frac{(2 \text{ order} - \text{column} - 1) \text{ column}}{2} \quad (5.23)$$

$row \geq column$

$$index_0 = 0 \quad (5.24)$$

$row < column$

$$index_0 = \frac{(2 \text{ order} - \text{row} + 1) \text{ row}}{2} \quad (5.25)$$

$diagonal$

There are alternative formulas for determining the index, shown below:

$$index_1 = TotalValues - \frac{(\text{order} - \text{column} + 3)(\text{order} - \text{column})}{2} + \text{row} - \text{column} \quad (5.26)$$

$row \geq column$

$$index_1 = TotalValues - \frac{(\text{order} - \text{row} + 3)(\text{order} - \text{row})}{2} \quad (5.27)$$

$diagonal$

$$index_0 = TotalValues - \frac{(\text{order} - \text{column} + 1)(\text{order} - \text{column})}{2} + \text{row} - \text{column} \quad (5.28)$$

$row \geq column$

$$index_0 = TotalValues - \frac{(\text{order} - \text{row} + 1)(\text{order} - \text{row})}{2} \quad (5.29)$$

$diagonal$

The total values are $\frac{n(n+1)}{2}$, but this is not recalculated every time since it is the length of the 1D array used as storage.

5.2.1.3 Upper triangular dense storage by row

One-based format								Zero-based format									
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8	0	0	1	2	3	4	5	6	7
2		9	10	11	12	13	14	15	1		8	9	10	11	12	13	14
3			16	17	18	19	20	21	2			15	16	17	18	19	20
4				22	23	24	25	26	3				21	22	23	24	25
5					27	28	29	30	4					26	27	28	29
6						31	32	33	5						30	31	32
7							34	35	6							33	34
8								36	7								35

Fig. 5.5: Storage by row of an $[8 \times 8]$ upper triangular matrix

The matrix is represented as a 1D array containing all upper-triangle entries row-by-row.

$$A = [1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 32 \ 33 \ 34 \ 35 \ 36] \quad (5.30)$$

For one-based format, the index of a lower triangular entry $[row, column]$ in the 1D array is given by:

$$index_1 = column + \frac{(2 \ order - row)(row - 1)}{2} \quad (5.31)$$

$row \leq column$

$$index_1 = 0 \quad (5.32)$$

$row > column$

$$index_1 = row + \frac{(2 \ order - row)(row - 1)}{2} \quad (5.33)$$

$diagonal$

For zero-based format, the above relations are:

$$index_0 = column + \frac{(2 \text{ order} - \text{row} - 1) \text{ row}}{2} \quad (5.34)$$

row ≤ column

$$index_0 = 0 \quad (5.35)$$

row > column

$$index_0 = \frac{(2 \text{ order} - \text{row} + 1) \text{ row}}{2} \quad (5.36)$$

diagonal

There are alternative formulas for determining the index, shown below:

$$index_1 = TotalValues - \frac{(\text{order} - \text{row} + 3)(\text{order} - \text{row})}{2} + \text{column} - \text{row} \quad (5.37)$$

row ≤ column

$$index_1 = TotalValues - \frac{(\text{order} - \text{row} + 3)(\text{order} - \text{row})}{2} \quad (5.38)$$

diagonal

$$index_0 = TotalValues - \frac{(\text{order} - \text{row} + 1)(\text{order} - \text{row})}{2} + \text{column} - \text{row} \quad (5.39)$$

row ≤ column

$$index_0 = TotalValues - \frac{(\text{order} - \text{row} + 1)(\text{order} - \text{row})}{2} \quad (5.40)$$

diagonal

The total values are $\frac{n(n+1)}{2}$, but this is not recalculated every time since it is the length of the 1D array used as storage.

5.2.1.4 Upper triangular dense storage by column

One-based format	
	1 2 3 4 5 6 7 8
1	1 2 4 7 11 16 22 29
2	3 5 8 12 17 23 30
3	6 9 13 18 24 31
4	10 14 19 25 32
5	15 20 26 33
6	21 27 34
7	28 35
8	36

Zero-based format	
	0 1 2 3 4 5 6 7
0	0 1 3 6 10 15 21 28
1	2 4 7 11 16 22 29
2	5 8 12 17 23 30
3	9 13 18 24 31
4	14 19 25 32
5	20 26 33
6	27 34
7	35

Fig. 5.6: Storage by column of an $[8 \times 8]$ upper triangular matrix

In this storage, also referenced as upper packed storage, the matrix is represented as a 1D array containing all upper-triangle entries column-by-column.

$$A = [1 \ 2 \ 3 \ 4 \ 5 \ \dots \ \dots \ 32 \ 33 \ 34 \ 35 \ 36] \quad (5.41)$$

For one-based format, the index of a lower triangular entry $[row, column]$ in the 1D array is given by:

$$index_1 = row + \frac{column(column-1)}{2} \quad (5.42)$$

$row \leq column$

$$index_1 = 0 \quad (5.43)$$

$row > column$

$$index_1 = \frac{row(row+1)}{2} \quad (5.44)$$

$diagonal$

For zero-based format, the above relations are:

$$\underset{row \leq column}{index_0} = row + \frac{column(column+1)}{2} \quad (5.45)$$

$$\underset{row > column}{index_0} = 0 \quad (5.46)$$

$$\underset{diagonal}{index_0} = \frac{row(row+3)}{2} \quad (5.47)$$

Note that the order of the matrix does not appear in the relations - it is still needed, however, for index range checking.

5.3 Symmetric Dense Matrix

Symmetric matrices are square matrices where $A_{ij} = A_{ji}$ for all valid i, j . Since the lower triangle is equal to the upper triangle, only one of them needs to be stored. The dense triangular storage formats can be used for the dense symmetric matrix as well to store only $\frac{n(n+1)}{2}$ values. Due to the symmetry of the matrix, the lower triangle storage by column leads to exactly the same layout as the upper triangle storage by row. Similarly, the lower triangle storage by row is the same as the lower triangle storage by column.

5.3.1 Implementations

The implementations demonstrated in this section use the triangular storages from Section 5.2. Of course, the “other” triangle is not treated as full of zeros, but as a symmetric triangle.

5.3.1.1 Lower Triangle by row or Upper Triangle by column

One-based format								Zero-based format									
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1	2	4	7	11	16	22	29	0	0	1	3	6	10	15	21	28
2	2	3	5	8	12	17	23	30	1	1	2	4	7	11	16	22	29
3	4	5	6	9	13	18	24	31	2	3	4	5	8	12	17	23	30
4	7	8	9	10	14	19	25	32	3	6	7	8	9	13	18	24	31
5	11	12	13	14	15	20	26	33	4	10	11	12	13	14	19	25	32
6	16	17	18	19	20	21	27	34	5	15	16	17	18	19	20	26	33
7	22	23	24	25	26	27	28	35	6	21	22	23	24	25	26	27	34
8	29	30	31	32	33	34	35	36	7	28	29	30	31	32	33	34	35

Fig. 5.7: Symmetric storage of an $[8 \times 8]$ symmetric matrix

$$index_1 = column + \frac{row(row-1)}{2} \quad (5.48)$$

$row \geq column$

$$index_1 = row + \frac{column(column-1)}{2} \quad (5.49)$$

$row \leq column$

$$index_1 = \frac{row(row+1)}{2} \quad (5.50)$$

diagonal

$$index_0 = column + \frac{row(row+1)}{2} \quad (5.51)$$

$$index_0 = row + \frac{column(column+1)}{2} \quad (5.52)$$

$row \leq column$

$$index_0 = \frac{row(row+3)}{2} \quad (5.53)$$

diagonal

5.3.1.2 Lower Triangle by column or Upper Triangle by row

One-based format								Zero-based format									
	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8	0	0	1	2	3	4	5	6	7
2	2	9	10	11	12	13	14	15	1	1	8	9	10	11	12	13	14
3	3	10	16	17	18	19	20	21	2	2	9	15	16	17	18	19	20
4	4	11	17	22	23	24	25	26	3	3	10	16	21	22	23	24	25
5	5	12	18	23	27	28	29	30	4	4	11	17	22	26	27	28	29
6	6	13	19	24	28	31	32	33	5	5	12	18	23	27	30	31	32
7	7	14	20	25	29	32	34	35	6	6	13	19	24	28	31	33	34
8	8	15	21	26	30	33	35	36	7	7	14	20	25	29	32	34	35

Fig. 5.8: Symmetric storage of an $[8 \times 8]$ symmetric matrix

$$index_1 = row + \frac{(2 \text{ order} - column)(column - 1)}{2} \quad (5.54)$$

$row \geq column$

$$index_1 = column + \frac{(2 \text{ order} - row)(row - 1)}{2} \quad (5.55)$$

$row \leq column$

$$index_1 = row + \frac{(2 \text{ order} - row)(row - 1)}{2} \quad (5.56)$$

$diagonal$

$$index_0 = row + \frac{(2 \text{ order} - column - 1) column}{2} \quad (5.57)$$

$row \geq column$

$$index_0 = column + \frac{(2 \text{ order} - row - 1) row}{2} \quad (5.58)$$

$row \leq column$

$$index_0 = \frac{(2 \text{ order} - row + 1) row}{2} \quad (5.59)$$

$diagonal$

$$index_1 = TotalValues - \frac{(order - column + 3)(order - column)}{2} + row - column \quad (5.60)$$

row ≥ column

$$index_1 = TotalValues - \frac{(order - row + 3)(order - row)}{2} + column - row \quad (5.61)$$

row ≤ column

$$index_1 = TotalValues - \frac{(order - row + 3)(order - row)}{2} \quad (5.62)$$

diagonal

$$index_0 = TotalValues - \frac{(order - column + 1)(order - column)}{2} + row - column \quad (5.63)$$

row ≥ column

$$index_0 = TotalValues - \frac{(order - row + 1)(order - row)}{2} + column - row \quad (5.64)$$

row ≤ column

$$index_0 = TotalValues - \frac{(order - row + 1)(order - row)}{2} \quad (5.65)$$

diagonal

5.4 Diagonal Dense Matrix

A diagonal matrix can only have non-zero entries on the diagonal – all off-diagonal entries are assumed zero. From an implementation perspective, the diagonal matrices can be seen as vectors so array-based (dense) or sparse implementations are applicable. The diagonal dense matrix stores all entries of the diagonal, requiring $O(n)$ space. A notable use of the diagonal dense matrix is the lumped mass matrix.

One-based format								Zero-based format								
1	1								0	0						
2		2							1		1					
3			3						2			2				
4				4					3				3			
5					5				4					4		
6						6			5						5	
7							7		6						6	
8							8		7							7

Fig. 5.9: Storage of an $[8 \times 8]$ diagonal matrix

Since a diagonal matrix has collapsed to a single dimension (similar to a vector), storage by row/column etc are not applicable here. The matrix can be stored in an array:

$$A = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8] \tag{5.66}$$

with trivial indexing, which for both one-based and zero-based arrays is:

$$\begin{matrix} index = 0 \\ row \neq column \end{matrix} \tag{5.67}$$

$$\begin{matrix} index = row \\ diagonal \end{matrix} \tag{5.68}$$

5.5 Bandwidth-aware storage

The matrix storage schemes mentioned so far are “dense” because they store the whole matrix or a whole area of the matrix. Banded matrices, i.e. matrices with “small” bandwidth can be stored more efficiently. The bandwidth of the matrix is the smallest number of adjacent diagonals to which the non-zero elements are confined [76]. In more detail:

- Below the diagonal, consider the closest possible line parallel to the diagonal such that all entries that are further away are all zero. The distance of that line from the diagonal is k_1 and is called left half-bandwidth (excludes the diagonal).
- Above the diagonal, consider the closest possible line parallel to the diagonal such that all entries that are further away are all zero. The distance of that line from the diagonal is k_2 and is called right half-bandwidth (excludes the diagonal).

The bandwidth of the matrix is k_1+k_2+1 .

When the bandwidth is small then it is worthwhile to store the matrix in appropriate formats that take this into account. The matrices (e.g. stiffness matrix) that are derived in a FEM/EFG/IGA analysis typically have non-zero entries relatively close to the diagonal. Note that the distance from the diagonal is dependent on the numbering of the degrees of freedom. Renumbering techniques are extensively researched and can be used to considerably reduce the bandwidth of a matrix. By making the bandwidth smaller, not only does the storage requirements become lower, but also matrix operations require less calculations to perform.

The following matrix will be used as an example:

$$\mathbf{K} = \begin{bmatrix} K_{11} & K_{12} & 0 & K_{14} & 0 & 0 & 0 & 0 \\ & K_{22} & K_{23} & 0 & 0 & 0 & 0 & 0 \\ & & K_{33} & K_{34} & 0 & K_{36} & 0 & 0 \\ & & & K_{44} & K_{45} & K_{46} & 0 & 0 \\ & & & & K_{55} & K_{56} & 0 & K_{58} \\ & & & & & K_{66} & K_{67} & 0 \\ & & & & & & K_{77} & K_{78} \\ & & & & & & & K_{88} \end{bmatrix} \quad (5.69)$$

For the ensuing analysis, it is important to define the row index of the highest (furthest from the diagonal) non-zero entry of a column j , which will be symbolized as m_j . In the example:

$$m_6 = 3$$

$$m_8 = 5$$

because the highest non-zero entry of column 6 is in row 3 and the highest non-zero entry of column 8 is in row 5.

Another important value is the height of a column, which is assumed to range from the diagonal of each column up to m_j . On the diagonal, the row index is equal to the column index, thus the diagonal entry of column j is on row j . The height of a column can be calculated by:

$$\underset{\text{diagonal excluded}}{\text{columnHeight}} = \text{columnIndex} - m_{\text{column}} \quad (5.70)$$

The height of eq. (5.70) does not include the diagonal. Simply add 1 for the diagonal inclusive height. For the heights of columns 6 and 8 of the example:

$$h_6 = 6 - m_6 = 6 - 3 = 3$$

$$h_8 = 8 - m_8 = 8 - 5 = 3$$

5.5.1 Symmetric Banded Matrix

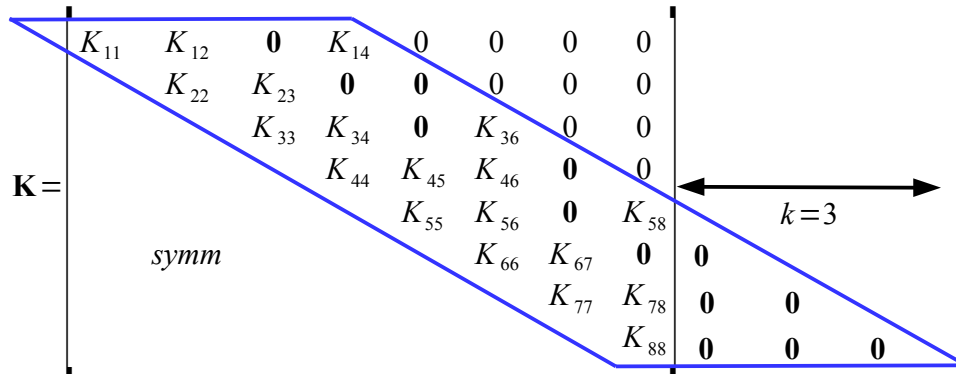


Fig. 5.10: Entries stored with symmetric banded storage for a symmetric matrix

In Fig. 5.10, the half-bandwidth (excluding the diagonal) is 3. As a result, 3+1=4 diagonals need to be stored. Note that there are zero elements included in the band and zeros are needed to pad diagonals to the size of the main diagonal, which is equal to the order of the matrix.

The entries of either triangle that lie inside the bandwidth can be seen as a rectangular matrix of size $order \times (k+1)$. In order to store this matrix, the storage schemes discussed in Section 5.1 can be used.

$$\begin{bmatrix} K_{11} & K_{22} & K_{33} & K_{44} & K_{55} & K_{66} & K_{77} & K_{88} \\ K_{12} & K_{23} & K_{34} & K_{45} & K_{56} & K_{67} & K_{78} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & K_{46} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ K_{14} & \mathbf{0} & K_{36} & \mathbf{0} & K_{58} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \quad (5.71)$$

For matrices in symmetric banded storage, m_j is calculated by: $j+1-b$. However, for the first few columns ($j < b+1$) this formula will yield negative m_j . These are the columns where “all” column elements (i.e. from line 1 up to the diagonal) are stored, like columns 1-4 in Fig. 5.10. In order to take these corner cases into account:

$$\begin{aligned} m_{1,j} &= \max(1, j+1-b) \quad (\text{one-based}) \\ m_{0,j} &= \max(0, j-b) \quad (\text{zero-based}) \end{aligned} \quad (5.72)$$

5.5.2 Symmetric Skyline Matrix

The symmetric banded storage can lead in huge savings in storage and calculations but it can be further improved. For the skyline format, we will consider the matrix in a column-wise manner, starting from the diagonal and going up. When using the symmetric banded storage, each column would require storing $k+1$ entries, where k is the half-bandwidth (excluding the diagonal). This would be fixed for all columns. The skyline matrix defines a separate boundary for each column. Appropriately, the skyline storage is also known as the variable band matrix storage.

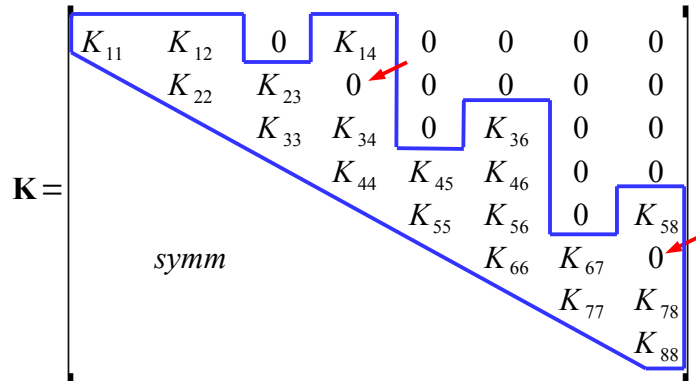


Fig. 5.11: Entries stored with symmetric skyline storage for an symmetric matrix

In Fig. 5.11 the blue line, called the “skyline”, encloses the values of the matrix that will be stored. Note that there are no “padding” zeros in this storage scheme, but there are still some zeros stored (marked with arrows in the example). For each column, all values from the diagonal to the last non-zero value (going upwards) is stored, so zeros that have non-zero values above them (not necessarily directly above them, just somewhere above them) are included.

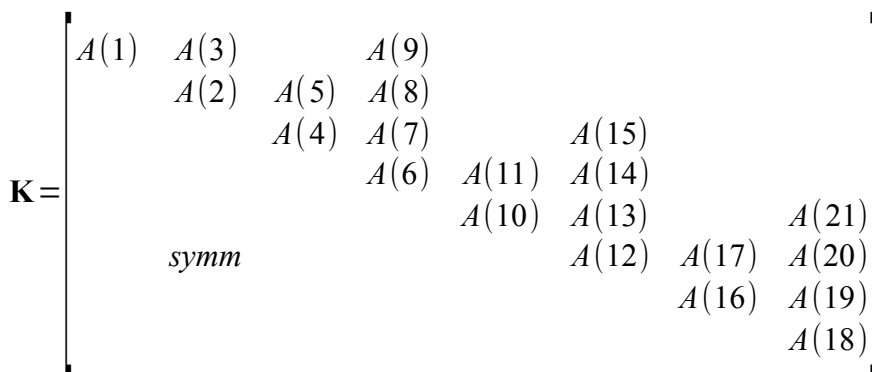


Fig. 5.12: Order of stored entries with symmetric skyline storage for an $[8 \times 8]$ symmetric matrix

The values are stored column-by-column in a 1D array. The order of the values starts from the diagonal of a column and goes upwards as shown in Fig. 5.12. $A(8)$ and $A(20)$ are the zeros that

are included in the storage scheme. The array which contains plain values is:

$$\mathbf{A}=[1 \ 2 \ 3 \ 4 \ \dots \ 18 \ 19 \ 20 \ 21] \quad (5.73)$$

However, each column may have any number of values stored, unlike the other storages where each column had a specific and predictable number of values. As a result, a formula by itself is not sufficient here; a supportive array is needed to be able to properly map the 1D array of values to proper matrix entries.

The supportive array is an integer array that contains the 1D-array index of all diagonal entries of the matrix. By inspecting the diagonal of Fig. 5.12, these indexes are:

$$\mathbf{diagIndexes}_1=[1 \ 2 \ 4 \ 6 \ 10 \ 12 \ 16 \ 18] \text{ (one-based)} \quad (5.74)$$

$$\mathbf{diagIndexes}_0=[0 \ 1 \ 3 \ 5 \ 9 \ 11 \ 15 \ 17] \text{ (zero-based)}$$

The entry in position i shows the index in \mathbf{A} that contains the diagonal entry of column i .

Using the **diagIndexes** array, it is easy to derive the start and end of each column: the diagonal entry is the start of each column while the end of a column is at the start of the next column. For example, $diagIndexes(4)=6$ shows that the diagonal of column $i=4$ is at position 6 in \mathbf{A} . Furthermore, the next column start at $diagIndexes(i+1)=diagIndexes(5)=10$ so the entries that correspond to column $i=4$ are those ranging from positions 6 to 10 (without 10). Therefore, the entries of a column are:

$$\mathbf{columnEntries}=[diagIndexes(columnIndex), diagIndexes(columnIndex+1)] \quad (5.75)$$

With the **diagIndexes** array as defined so far, we cannot find how many entries are in the last column – there is no $i+1$ for the last column. For this reason, an extra entry is included. The entry is equal to the total number of entries in \mathbf{A} plus 1. It can be considered as the diagonal entry of a virtual next column, as shown Fig. 5.13.

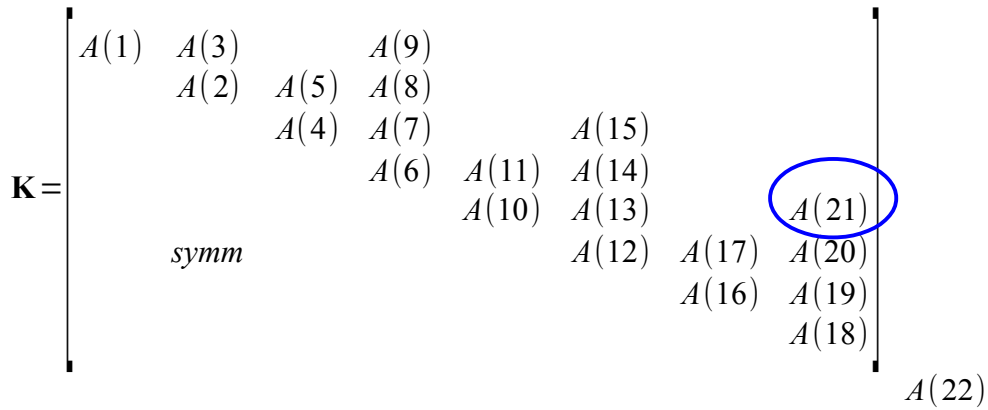


Fig. 5.13: The number of entries in the skyline storage as well as the extra entry needed in the **diagIndexes** array.

In the example, the skyline format stores 21 values, so we need to add a “22” element in the **diagIndexes** array. As a result, the full form of the skyline format stores which stores values in one array and diagonal indexes in a second array is the following:

$$\mathbf{A} = [1 \ 2 \ 3 \ 4 \ \dots \ 18 \ 19 \ 20 \ 21] \quad (5.76)$$

$$\mathbf{diagIndexes} = [1 \ 2 \ 4 \ 6 \ 10 \ 12 \ 16 \ 18 \ 22]$$

When the matrix is stored in skyline format, the height of a column can be easily derived by:

$$columnHeight_{diagonal\ excluded} = diagIndexes(columnIndex + 1) - diagIndexes(columnIndex) - 1 \quad (5.77)$$

Due to (5.76),

$$h_6 = diagIndexes(7) - diagIndexes(6) - 1 = 16 - 12 - 1 = 3$$

$$h_8 = diagIndexes(9) - diagIndexes(8) - 1 = 22 - 18 - 1 = 3$$

The calculation for column 8 also demonstrates the usage of the extra entry in the **diagIndexes** array.

In order to calculate m_j , we use eq. (5.70) as follows:

$$m_{column} = columnIndex - columnHeight_{diagonal\ excluded} \quad (5.78)$$

For example:

$$m_6 = 6 - 3 = 3$$

$$m_8 = 8 - 3 = 5$$

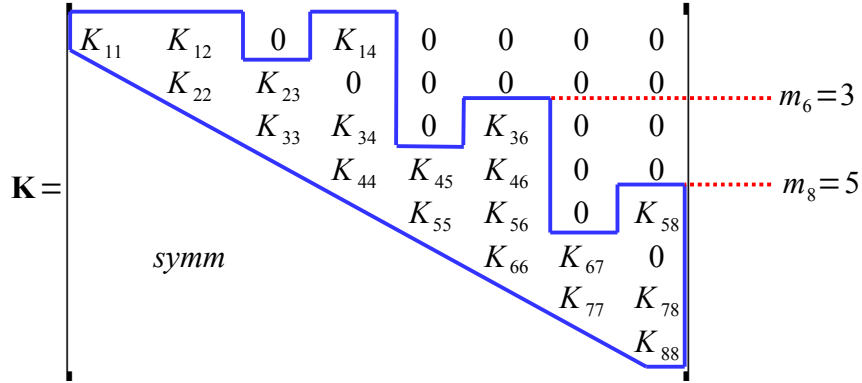


Fig. 5.14: The highest entries of columns 6 and 8.

To retrieve an entry $K(\text{rowIndex}, \text{columnIndex})$ of the matrix, where $\text{rowIndex} \leq \text{columnIndex}$ since the upper triangle is stored, it is initially verified that it is inside the active (stored) part of the corresponding column, i.e. inside the “skyline”, otherwise the entry is implicitly zero. An entry is outside the active column and assumed zero if:

$$\text{columnIndex} - \text{rowIndex} > \text{columnHeight} \quad (5.79)$$

If the entry is inside the “skyline”, then the index where it is stored is calculated as follows:

$$\underset{\text{rowIndex} \leq \text{columnIndex}}{\text{index}} = \text{diagIndexes}(\text{columnIndex}) + \text{columnIndex} - \text{rowIndex} \quad (5.80)$$

Finally, the entry is retrieved from the array \mathbf{A} :

$$K(\text{rowIndex}, \text{columnIndex}) = A(\text{index}) \quad (5.81)$$

For example, for column 6, for which $m_6 = 3$:

$$K_{36} = A(\text{diagIndexes}(6) + 6 - 3) = A(12 + 6 - 3) = A(15)$$

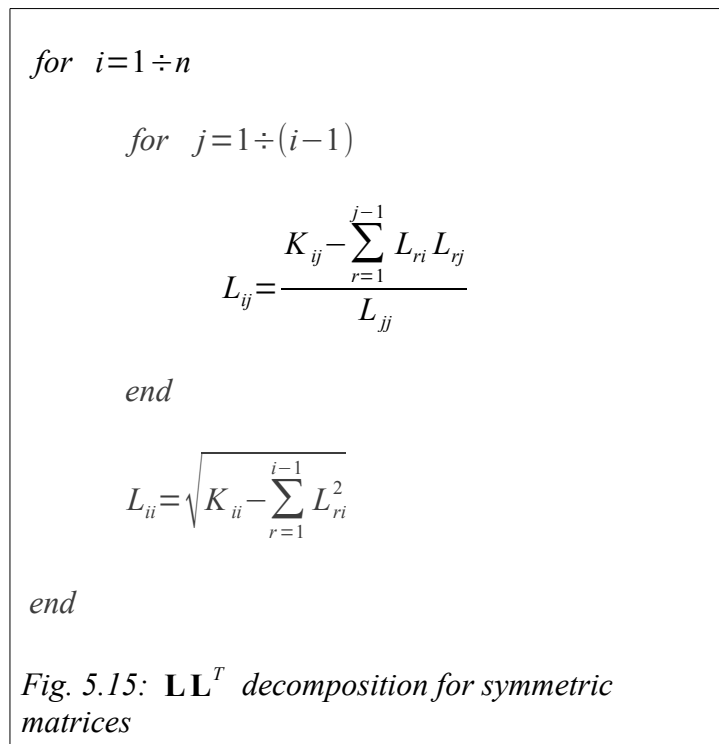
$$K_{66} = A(\text{diagIndexes}(6) + 6 - 6) = A(12 + 6 - 6) = A(12)$$

Formulas (5.79) and (5.80) are valid for both zero-based and one-based but only when $\text{rowIndex} \leq \text{columnIndex}$. However, the matrix is symmetric so $K_{ij} = K_{ji}$. Thus, in order to calculate an entry in the lower triangle of the matrix, where $\text{rowIndex} \geq \text{columnIndex}$, the symmetric entry $K(\text{columnIndex}, \text{rowIndex})$, which is in the upper triangle, can be retrieved with the aforementioned formulas.

5.5.3 Factorization

One of the important benefits of bandwidth aware storage when compared to dense as well as sparse (non-zeros only) formats is factorization. In a matrix with small bandwidth, it can save a lot of calculations compared to dense matrix formats while also retaining the ability to be performed in-place, unlike the sparse formats (note: complete factorization). The ensuing analysis pertains to symmetric matrices and \mathbf{LL}^T (Cholesky) decomposition, but can be easily expanded to general matrices and \mathbf{LU} decomposition as well as for \mathbf{LDL}^T and \mathbf{LDU} decompositions.

A simple implementation of the general-purpose (i.e. for any symmetric matrix) Cholesky decomposition can be performed with the algorithm demonstrated in Fig. 5.15.



The sums in the algorithm are essentially dot products between column i and column j (note: any row i is equivalent to column i since the matrix is symmetric). The columns entries involved in the dot product are shown in Fig. 5.16 and range from row indexes 1 to $\min(i, j)$ (note: in Figs. 5.15, 5.16 $\min(i, j)=j$).

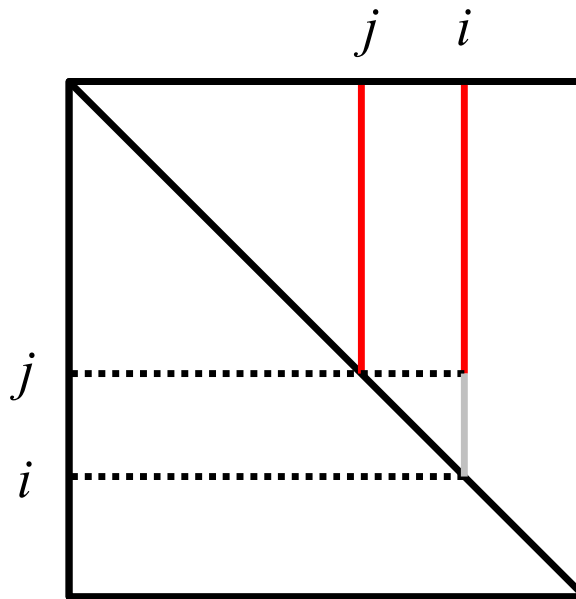


Fig. 5.16: Entries involved in dot product between columns i , j (red line)

5.5.4 Banded Factorization

When the matrix is banded, there are a lot of redundant calculations involved in the generic algorithm since a large part of both columns may be full of zeros. In particular, in Fig. 5.17, all entries above the blue line are zeros. The multiplication of columns i , j from rows 1 through m involves zero values in at least one of the columns and thus can be completely avoided. Thus, it is sufficient to multiply the entries from rows $\max(m_i, m_j)$ through $\min(i, j)$ (note: in the algorithm the indexes are always $i > j$, so $\max(m_i, m_j) = m_i$, $\min(i, j) = j$). The section that is multiplied is highlighted with red color in Fig. 5.17.

When factorizing a banded matrix, no entries outside the bandwidth will become non-zero. This is because when a zero entry has only zero entries above, it will still be zero after the factorization. However, if any of the entries above a zero entry is non-zero, then in general the entry will have a non-zero value after the factorization. Therefore, entries outside the bandwidth, which are all zero, are guaranteed to remain zero.

Since all entries outside the bandwidth remain zero, another change to the algorithm is the range of index j , whose range is changed from $1 \div (i-1)$ to $(m_i+1) \div (i-1)$. An alternative observation with the same conclusion is the following: when performing calculations for column i , all columns $j=1 \div m_i$ only have entries in rows $j=1 \div m_i$. These entries correspond to zero entries of column i since by definition column i has non-zero entries from m_i and below.

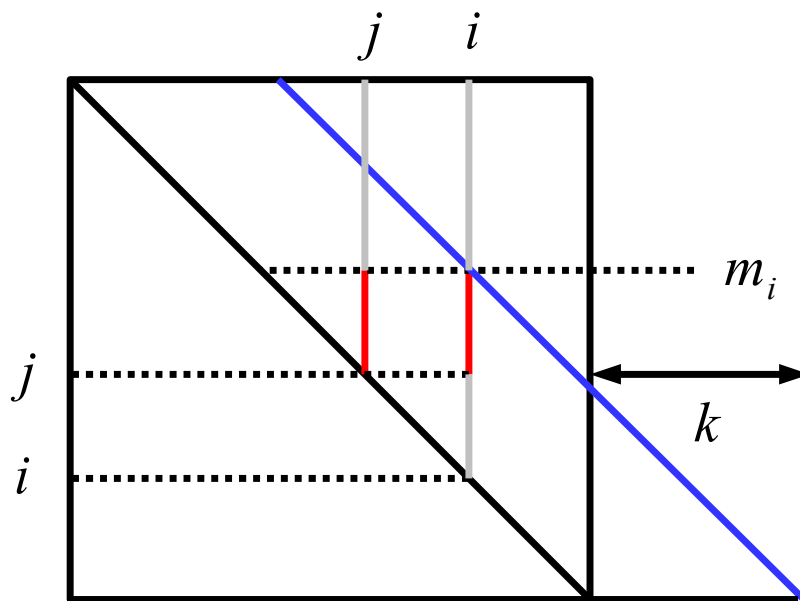


Fig. 5.17: Entries involved in dot product between columns i , j (red line) for banded matrices

for $i=1 \div n$

$$m_i = \max(1, i+1-b)$$

for $j=(m_i+1) \div (i-1)$

$$L_{ij} = \frac{K_{ij} - \sum_{r=m_i}^{j-1} L_{ri} L_{rj}}{L_{jj}}$$

end

$$L_{ii} = \sqrt{K_{ii} - \sum_{r=m_i}^{i-1} L_{ri}^2}$$

end

Fig. 5.18: $\mathbf{L}\mathbf{L}^T$ decomposition for symmetric banded matrices

The banded storage allows in-place factorization since all non-zeros are inside the bandwidth even after the factorization and calculations do not depend on overwritten entries.

5.5.5 Skyline Factorization

When the matrix is stored in skyline format, only rows $[m_j, j]$ are stored for column j . In Fig. 5.19, m_i, m_j are the highest non-zero entries of columns $i=6, j=7$, respectively. For the multiplication of those columns, only calculations on the part highlighted with red is needed.

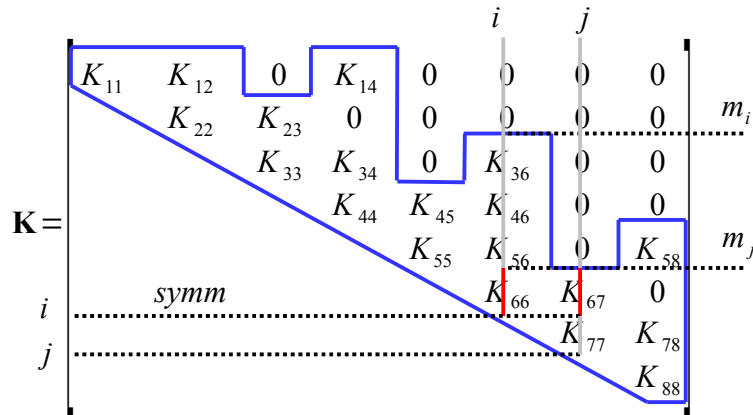


Fig. 5.19: Entries involved in dot product between columns i, j (red line) for a skyline matrix

More generally, the multiplication starts from $\max(m_i, m_j)$, because one of the two columns will have zeros in all rows above $\max(m_i, m_j)$, and continues up to $\min(i, j)$ (note: in the algorithm the indexes are always $i > j$, so $\min(i, j) = j$, but $\max(m_i, m_j)$ can be anything here!). In the example of Fig. 5.19:

$$\begin{aligned}
 [\text{column } 6] \cdot [\text{column } 7] &= K_{66} \cdot K_{67} \\
 [\text{column } 6] \cdot [\text{column } 8] &= K_{56} \cdot K_{58} + K_{66} \cdot K_{68}
 \end{aligned}$$

One change in the code is the new range for the column multiplications. Another is the range of index j , which is similar to the change in the banded algorithm.

for $i=1 \div n$

$m_i = \dots \rightarrow \text{eq. (5.78)}$

for $j=(m_i+1) \div (i-1)$

$m_j = \dots \rightarrow \text{eq. (5.78)}$

$m = \max(m_i, m_j)$

$$L_{ij} = \frac{K_{ij} - \sum_{r=m}^{j-1} L_{ri} L_{rj}}{L_{jj}}$$

end

$$L_{ii} = \sqrt{K_{ii} - \sum_{r=m_i}^{i-1} L_{ri}^2}$$

end

Fig. 5.20: \mathbf{LL}^T decomposition for symmetric banded matrices

As mentioned in Section 5.5.4, a zero entry with zeros above it will remain zero after factorization, while a zero entry which has a non-zero anywhere above it will have a non-zero value after factorization. Considering that skyline only stores from the diagonal up to the highest non-zero entry for each column, it is obvious that any entry outside the pattern (which is zero) will remain zero after factorization. More importantly though, the zero entries that the skyline format stores are the only zeros that will be non-zeros after the factorization. The skyline format stores exactly the entries that are needed in the factorized form of the matrix. It is, therefore, ideal for solution methods where complete \mathbf{LL}^T factorization is used. The skyline storage also allows in-place factorization since all non-zeros are inside the “skyline” even after the factorization and calculations do not depend on overwritten entries.

5.5.6 Numbering considerations

The numbering of the degrees of freedom is important for bandwidth-aware matrix storages. The numbering affects the bandwidth which in turn can critically affect the space required as well as the calculations involved when working with such storage schemes. A simple example is illustrated in Fig. 5.21, 5.22 for a linear arrangement with 4 truss elements (it is assumed $(EA)/L=1$). The red line shows the bandwidth of the matrix.

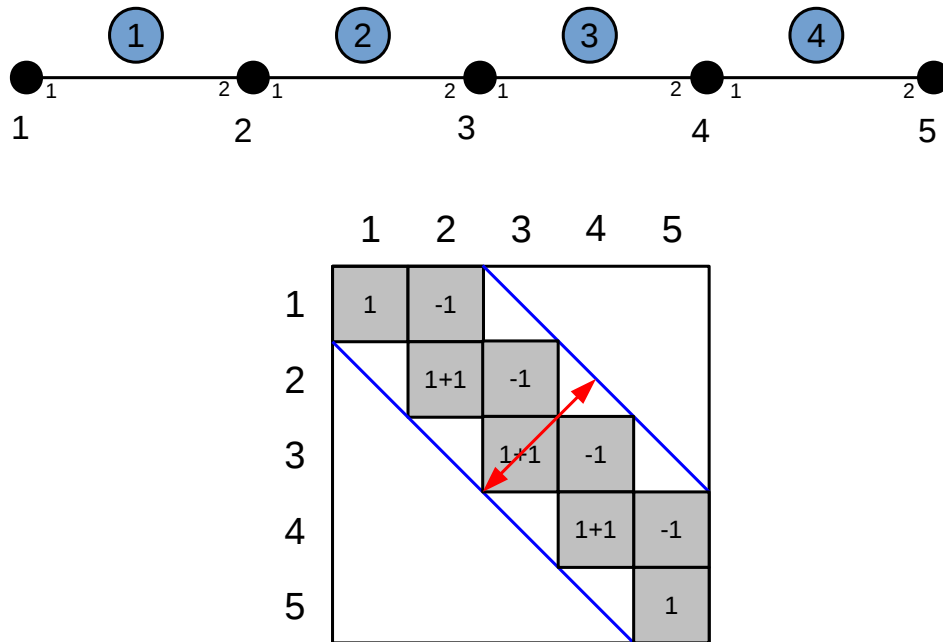


Fig. 5.21: Good numbering of degrees of freedom. Grey = non-zero entries

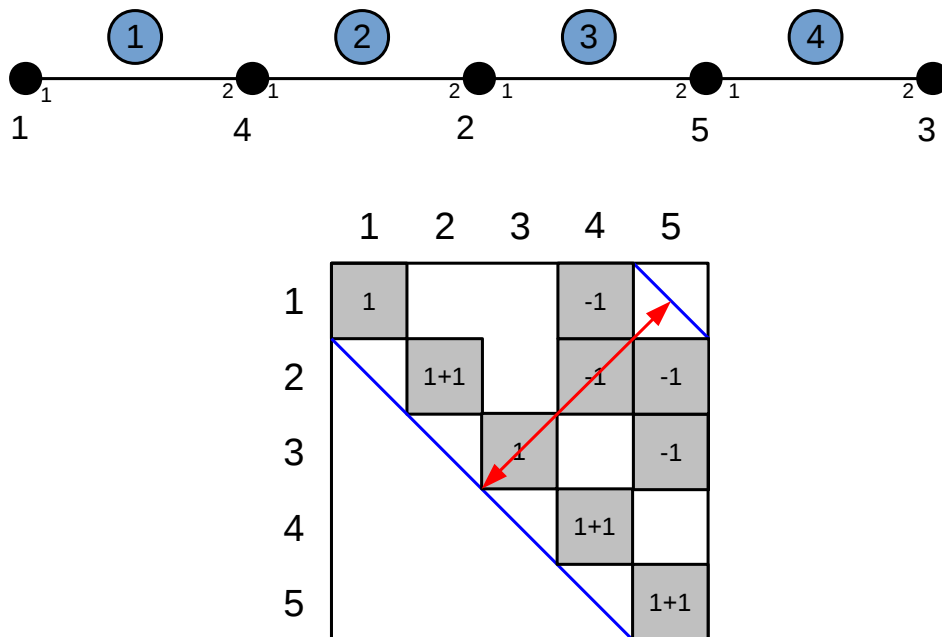


Fig. 5.22: Bad numbering of degrees of freedom. Grey = non-zero entries

A more complex example for triangular elements with one degree of freedom per node is the following:

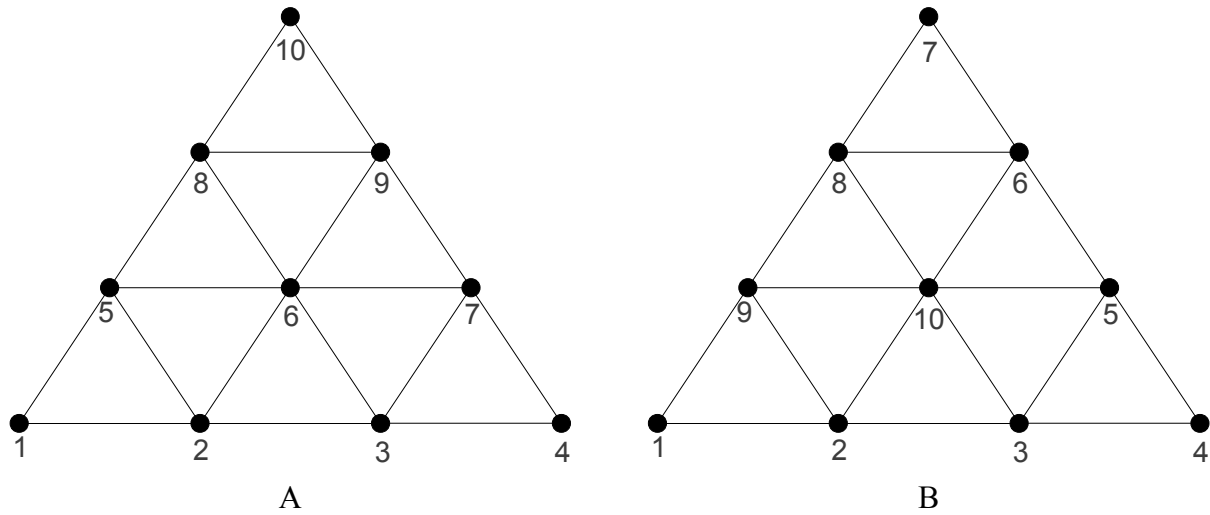


Fig. 5.23: Global numbering

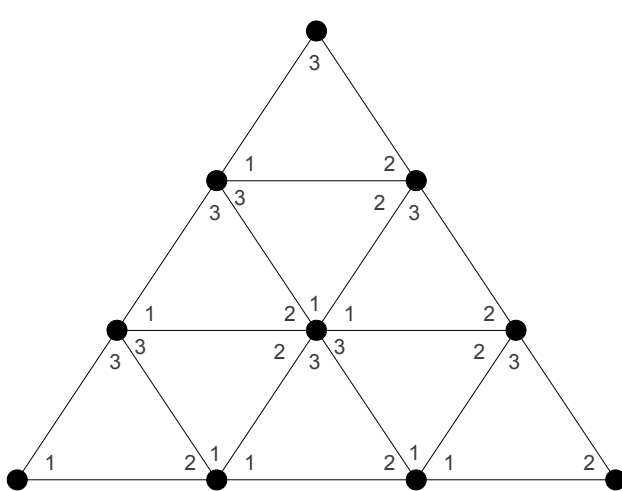


Fig. 5.24: Local numbering

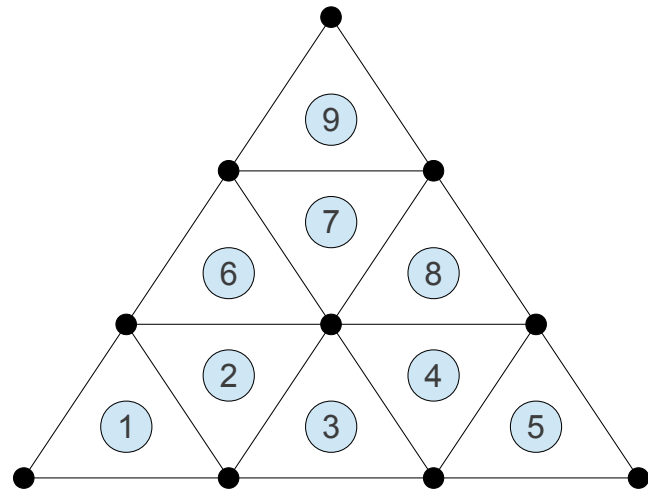


Fig. 5.25: Element numbering

Table 5.1 shows the column height for each of the degrees of freedom for numbering A while Table 5.2 shows the column heights for numbering B. A schematic depiction from the diagonal to the highest column entry of the matrix is shown in Figs. 5.26, 5.27 (not all grayed entries are necessarily non-zero). The red line shows the bandwidth of the matrix.

Global degree of freedom (dof)	Finite elements involved	Minimum dof involved	Column height $h_i = i - m_i$
1	1	1	1-1 = 0
2	1,2,3	1	2-1 = 1
3	3,4,5	2	3-2 = 1
4	5	3	4-3 = 1
5	1,2,6	1	5-1 = 4
6	2,3,4,6,7,8	2	6-2 = 4
7	4,5,8	3	7-3 = 4
8	6,7,9	5	8-5 = 3
9	7,8,9	6	9-6 = 3
10	9	8	10 - 8 = 2

Table 5.1: Column height (diagonal exclusive) for each column for numbering A

Global degree of freedom (dof) i	Finite elements involved	Minimum dof involved m_i	Column height $h_i = i - m_i$
1	1	1	1-1 = 0
2	1,2,3	1	2-1 = 1
3	3,4,5	2	3-2 = 1
4	5	3	4-3 = 1
5	4,5,8	3	5-3 = 2
6	7,8,9	5	6-5 = 1
7	9	6	7-6 = 1
8	6,7,9	6	8-6 = 2
9	1,2,6	1	9-1 = 8
10	2,3,4,6,7,8	2	10 - 2 = 8

Table 5.2: Column height (diagonal exclusive) for each column for numbering B

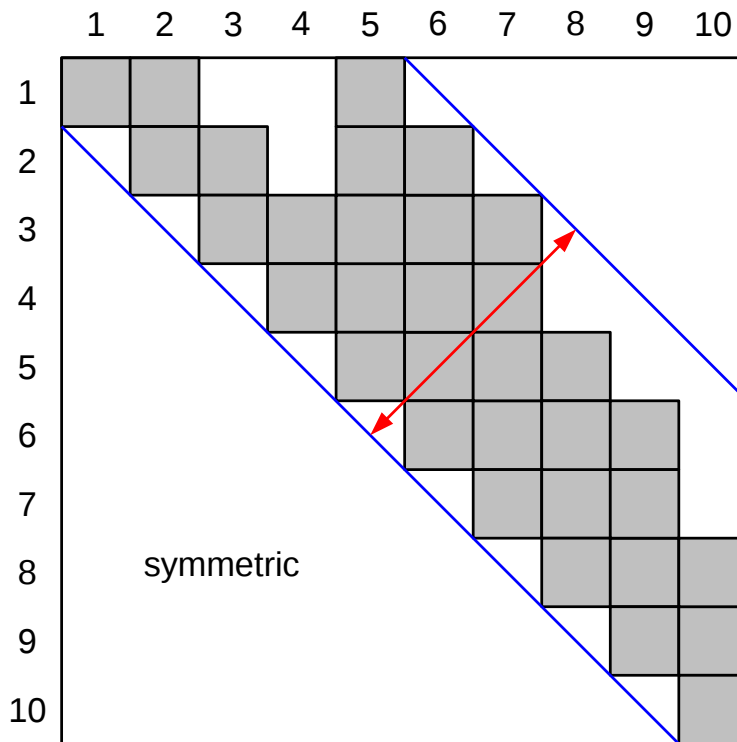


Fig. 5.26: Numbering A. Grey = entries up to the highest non-zero entry of the column

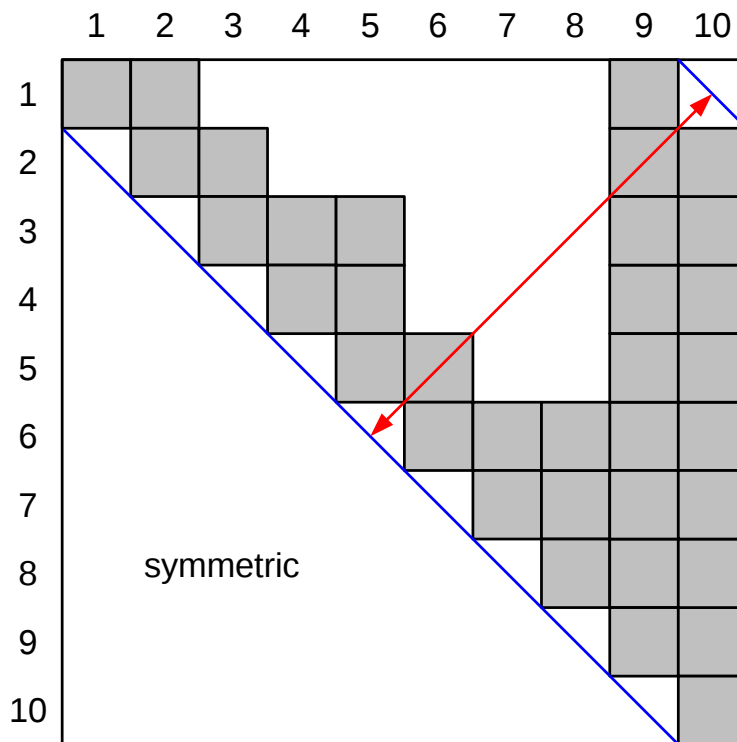


Fig. 5.27: Numbering B. Grey = entries up to the highest non-zero entry of the column

There is a plethora of algorithms to improve the numbering of the domain in order to reduce the matrix bandwidth. One widely-used one is the Cuthill–McKee algorithm [77]. Fig. 5.28 shows an example of applying the (reverse) Cuthill–McKee algorithm on a 60×60 matrix¹.

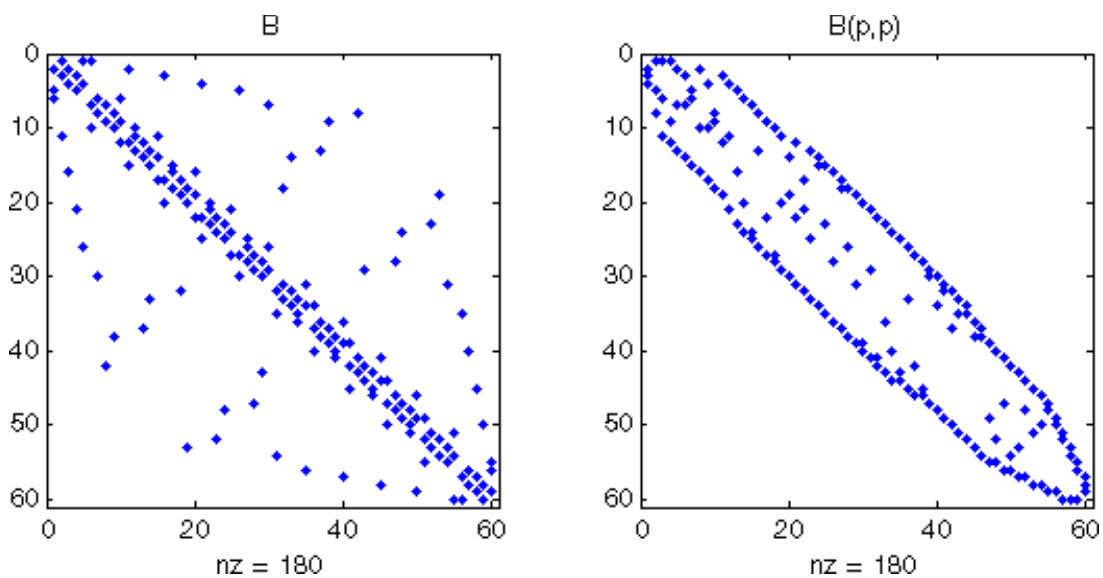


Fig. 5.28: Bandwidth improvement through the (reverse) Cuthill–McKee algorithm

It should be noted that the bandwidth problem, or equivalently the graph bandwidth problem is NP-hard [78]. As such, it is unlikely that there exists an efficient (polynomial) algorithm that finds the minimum bandwidth, but many of the available algorithms have been shown to give good results.

¹ Image from: <http://www.mathworks.com/help/matlab/ref/symrcm.html>

5.6 Sparse Matrix

For a sparse storage of a matrix to be practical, the non-zero entries should be few enough (compared to the total possible entries of the matrix) so that it is worth taking advantage of them to reduce both storage requirements and number of calculations in matrix operations. Ideally, it would be desirable to store and operate on non-zero entries only, but this is not necessarily a clear win in either storage or computational effort. Regarding storage, the complication is that sparse data formats include more overhead than the plain arrays used for denser types of matrices because they need to store indexes in addition to the values of the non-zero entries. As far as calculations are concerned, matrix operations with sparse formats cannot be performed as fast as with denser formats but the arithmetic calculations involved are (or should be) considerably fewer in typical cases.

For these reasons, a practical requirement for a matrix to be “effectively” sparse, i.e. to warrant using the sparse formats discussed in this section, is that it contains $O(n)$ non-zero entries. If each row/column contains a small/constant number of non-zeros, independent of the matrix dimension, then the requirements is satisfied. This is true for the large (domain/subdomain) matrices encountered in finite element methods etc. Apart from the number of non-zeros, their particular locations in the matrix can often be exploited. Physical problems usually exhibit a characteristic pattern that can lead to more efficient sparse storage.

The sparse storages presented in this work only need to store the non-zero entries. For the benefit of storing only non-zeros, each sparse format is typically good for only a few usages, so using the correct format for the appropriate task is very important. For the purposes of this work, there are two primary categories: formats that are good for operations (e.g. sparse-matrix vector multiplication or SpMV) and formats that are good for incrementally building the matrix, which are referred to as sparse matrix builders. The subcategories are analyzed in the corresponding sections.

5.6.1 Sparse Matrix Builders

As the name suggests, these formats are good for incrementally building a matrix in sparse format. There are formats that are more suitable for operations so these are used for the assembly phase and then converted to another format for the solution phase. There are two subcategories: formats that support (fast) lookups, i.e. finding an existing entry and updating its value, and formats that do not support (fast) lookups. Formats that support lookups are in generally more complex but must be

used when the values need to be continually updated. On the other hand, if the task involves calculating the final values of the matrix entries before inserting them in the builder, then no lookups are required and the simpler formats are applicable.

Two formats are examined: the Coordinate List (COO) and the Dictionary of Keys (DOK) [79]. Another popular format is the List of Lists (LIL), but it falls into one of the categories discussed below. For each of the implementations, there is a symmetric variant that uses the same backing structures while taking care of symmetry considerations.

5.6.1.1 Coordinate list (COO)

COO stores a list of (row, column, value) tuples. The entries can be added in any order and no requirements are present. Also, duplicate entries are allowed. The simplest implementation of this is to use 3 arrays, one for the row indexes, one for the column indexes and one for the value, where an entry i of the arrays corresponds to an entry of the matrix. The 4×5 matrix of (5.82) is shown in COO format in (5.83).

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ 0 & K_{22} & 0 & 0 & 0 \\ K_{31} & 0 & 0 & K_{34} & K_{35} \\ 0 & 0 & K_{43} & 0 & K_{45} \end{bmatrix} \quad (5.82)$$

$$\begin{aligned} \text{rowIndexes} &= [3 & 1 & 1 & 4 & 3 & 2 & 4 & 4] \\ \text{columnIndexes} &= [4 & 1 & 2 & 5 & 1 & 2 & 3 & 5] \\ \text{values} &= [K_{34} & K_{11} & K_{12} & K_{45} & K_{31} & K_{22} & K_{43} & K_{45}] \end{aligned} \quad (5.83)$$

While it is possible to have the entries in sorted order, this would mean $O(NZ)$ time to add a random entry to the matrix, where NZ is the number of entries contained. This is because it's $O(\log NZ)$ to find the correct spot, but $O(NZ)$ to move everything else by one slot. Thus, keeping the entry unsorted is preferred and in this case adding an entry is just $O(1)$ and trivial. Note that it is possible for the COO format to support lookups, but it entails iterating through all the values, which is $O(NZ)$ and much too slow.

This format is ideal when the final value of each entry is calculated before being inserted into the

builder, but is not good for continually updating partial values. Another advantage of this format is that it is easily converted to CSR/CSC formats for the solution phase.

5.6.1.2 Dictionary of Keys (DOK)

In contrast to COO, DOK format allows lookups so it is intended for cases where the final values are incrementally created inside the builder, which entails finding the current K_{ij} value and updating it appropriately. There are several ways to implement this. One is to map $(rowIndex, columnIndex)$ tuples to the corresponding values. This can be done with a hash-table or binary search tree (BST) or similar structures. The former allows $O(1)$ lookups while the latter allows $O(\log NZ)$ lookups, where NZ is the number of entries contained. BST also contains the entries in sorted order. The mapping of this approach for matrix (5.82) is shown in (5.84).

$$\begin{array}{l}
 \begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ 0 & K_{22} & 0 & 0 & 0 \\ K_{31} & 0 & 0 & K_{34} & K_{35} \\ 0 & 0 & K_{43} & 0 & K_{45} \end{bmatrix} \\
 (1,1) \rightarrow K_{11} \\
 (1,2) \rightarrow K_{12} \\
 (2,2) \rightarrow K_{22} \\
 (3,1) \rightarrow K_{31} \\
 (3,4) \rightarrow K_{34} \\
 (3,5) \rightarrow K_{35} \\
 (4,3) \rightarrow K_{43} \\
 (4,5) \rightarrow K_{45}
 \end{array} \tag{5.84}$$

Another way, borrowing from the LIL format as well, is to treat each row (or column) separately: instead of having a single data structure for the whole matrix, have a data structure for each row (or column). Then, all rows/columns need to be stored to an external data structure. This is shown in (5.85) for a row-wise mapping while (5.86) shows a column-wise mapping.

$$\begin{array}{l}
 row(1) \rightarrow [(1) \rightarrow K_{11} \quad (2) \rightarrow K_{12}] \\
 row(2) \rightarrow [(2) \rightarrow K_{22}] \\
 row(3) \rightarrow [(1) \rightarrow K_{31} \quad (4) \rightarrow K_{34}, (5) \rightarrow K_{35}] \\
 row(4) \rightarrow [(3) \rightarrow K_{43} \quad (5) \rightarrow K_{45}]
 \end{array} \tag{5.85}$$

$$\begin{aligned}
\text{column}(1) &\rightarrow [(1) \rightarrow K_{11} \quad (3) \rightarrow K_{31}] \\
\text{column}(2) &\rightarrow [(2) \rightarrow K_{22}] \\
\text{column}(3) &\rightarrow [(4) \rightarrow K_{43}] \\
\text{column}(4) &\rightarrow [(3) \rightarrow K_{34}] \\
\text{column}(5) &\rightarrow [(3) \rightarrow K_{35} \quad (4) \rightarrow K_{45}]
\end{aligned} \tag{5.86}$$

There are several combinations of external and internal data structures, each one with its own characteristics. Combinations with the most typical data structures are shown in Table 5.3, where “internal” refers to the data structures that stores rows/columns, whereas “external” is the data structure that stores references to them. A sparse matrix with very few entries might be better of using a DOK implementation backed by a single hash-table. On the other hand, if entries are concentrated in only a few columns and each column's entries need to be sorted, then a column-wise Hash-BST implementation might be more effective. If every row is going to be populated (as is the case for the stiffness matrix and other characteristic matrices), using an array as the external data structure to store each row in the corresponding *rowIndex* is simpler compared to Hash/BST while also keeping the rows sorted.

Data structure		Expected size		Lookup	Sorted	
external	internal	external	internal		external	internal
Single Hash Table		NZ		$O(1)$	No	
Single BST		NZ		$O(\log NZ)$	Yes	
Hash Table	Hash Table	$A \leq M$	$B \leq N$	$O(1)$	No	No
Hash Table	BST	$A \leq M$	$B \leq N$	$O(\log B)$	No	Yes
BST	Hash Table	$A \leq M$	$B \leq N$	$O(\log A)$	Yes	No
BST	BST	$A \leq M$	$B \leq N$	$O(\log A + \log B)$	Yes	Yes
Array (list)	Hash Table	M	$B \leq N$	$O(1)$	Yes	No
Array (list)	BST	M	$B \leq N$	$O(\log B)$	Yes	Yes

- A expected number of stored rows/columns
- B expected number of stored entries within a row/column
- M major dimension of the matrix
- N minor dimension of the matrix
- NZ total number of stored entries

Table 5.3: Combinations of typical backing data structures for the DOK format. “Internal” refers to the data structures that stores rows/columns, whereas “external” is the data structure that stores references to them.

Note that the lookup time of the BST structures is based on the actual size of the structure (A or B) and not the maximum size of the structure (M or N), although of course A, B can be as big as M, N respectively. Also note that the LIL format essentially corresponds to a BST-BST or Array-BST backed implementation of the DOK format.

5.6.2 Sparse Matrix formats for operations

The matrix is prepared with one of the sparse matrix builders and then converted to the Compressed Sparse Row (CSR) or Compressed Sparse Column format (CSC). These two formats are appropriate for arithmetic operations, though again care must be taken in the usage of each type depending on the operations involved in the solution phase.

5.6.2.1 Compressed Sparse Row (CSR)

Consider the 4×5 matrix of eq. (5.82) once again:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ 0 & K_{22} & 0 & 0 & 0 \\ K_{31} & 0 & 0 & K_{34} & K_{35} \\ 0 & 0 & K_{43} & 0 & K_{45} \end{bmatrix}$$

The indexes of the CSR format are first sorted by row index and then within the same row by column index. Note that it is assumed that each pair of row and column indexes appears only once. This is essentially the COO format, but with sorted entries:

$$\begin{aligned} \text{rowIndexes} &= [1 & 1 & 2 & 3 & 3 & 3 & 4 & 4] \\ \text{columnIndexes} &= [1 & 2 & 2 & 1 & 4 & 5 & 3 & 5] \\ \text{values} &= [K_{11} & K_{12} & K_{22} & K_{31} & K_{34} & K_{35} & K_{43} & K_{45}] \end{aligned} \quad (5.87)$$

Notice how the row indexes have consecutive repetitions of the same index and also that the repetitions are in increasing, fully predictable order. Thus, instead of storing the row indexes as shown in (5.87), the compressed version only stores the offset that a particular index would appear for the first time. For one-based format:

$$\begin{aligned} \text{rowIndexes} &= [1 & 3 & 4 & 7 & 9] \\ \text{columnIndexes} &= [1 & 2 & 2 & 1 & 4 & 5 & 3 & 5] \\ \text{values} &= [K_{11} & K_{12} & K_{22} & K_{31} & K_{34} & K_{35} & K_{43} & K_{45}] \end{aligned} \quad (5.88)$$

For zero-based format:

$$\begin{aligned} \text{rowIndexes} &= [0 & 2 & 3 & 6 & 8] \\ \text{columnIndexes} &= [0 & 1 & 1 & 0 & 3 & 4 & 2 & 4] \end{aligned} \quad (5.89)$$

$$\begin{aligned} \text{rowIndexes} &= [0 \quad 2 \quad 3 \quad 6 \quad 8] \\ \text{values} &= [K_{11} \quad K_{12} \quad K_{22} \quad K_{31} \quad K_{34} \quad K_{35} \quad K_{43} \quad K_{45}] \end{aligned}$$

Row i 's values are located in the range: $\text{rowIndexes}(i)$ (inclusive) up to $\text{rowIndexes}(i+1)$ exclusive. There is an extra entry at the end to accommodate the last row, so the size of the rowIndexes array is $n+1$, where n is the number of rows of the matrix.

The CSR format features efficient row slicing and fast matrix vector products but slow column slicing operations, where the CSC format excels.

5.6.2.2 Compressed Sparse Column (CSC)

Consider the 4×5 matrix of eq. (5.82) once again:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 \\ 0 & K_{22} & 0 & 0 & 0 \\ K_{31} & 0 & 0 & K_{34} & K_{35} \\ 0 & 0 & K_{43} & 0 & K_{45} \end{bmatrix}$$

The indexes of the CSC format are first sorted by column index and then within the same column by row index. Note that it is assumed that each pair of row and column indexes appears only once.

This is essentially the COO format, but with sorted entries:

$$\begin{aligned} \text{rowIndexes} &= [1 \quad 3 \quad 1 \quad 2 \quad 4 \quad 3 \quad 3 \quad 4] \\ \text{columnIndexes} &= [1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 5 \quad 5] \\ \text{values} &= [K_{11} \quad K_{31} \quad K_{12} \quad K_{22} \quad K_{43} \quad K_{34} \quad K_{35} \quad K_{45}] \end{aligned} \tag{5.90}$$

Notice how the column indexes have consecutive repetitions of the same index and also that the repetitions are in increasing, fully predictable order. Thus, instead of storing the column indexes as shown in (5.90), the compressed version only stores the offset that a particular index would appear for the first time. For one-based format:

$$\begin{aligned} \text{rowIndexes} &= [1 \quad 3 \quad 1 \quad 2 \quad 4 \quad 3 \quad 3 \quad 4] \\ \text{columnIndexes} &= [1 \quad 3 \quad 5 \quad 6 \quad 7 \quad 9] \\ \text{values} &= [K_{11} \quad K_{12} \quad K_{22} \quad K_{31} \quad K_{34} \quad K_{35} \quad K_{43} \quad K_{45}] \end{aligned} \tag{5.91}$$

For zero-based format:

$$\begin{aligned}
rowIndexes &= [0 \quad 2 \quad 0 \quad 1 \quad 3 \quad 2 \quad 2 \quad 3] \\
columnIndexes &= [0 \quad 2 \quad 4 \quad 5 \quad 6 \quad 8] \\
values &= [K_{11} \quad K_{12} \quad K_{22} \quad K_{31} \quad K_{34} \quad K_{35} \quad K_{43} \quad K_{45}]
\end{aligned} \tag{5.92}$$

Column i 's values are located in the range: $columnIndexes(i)$ (inclusive) up to $columnIndexes(i+1)$ exclusive. There is an extra entry at the end to accommodate the last column, so the size of the $columnIndexes$ array is $n+1$, where n is the number of columns of the matrix.

The CSC format features efficient column slicing and fast transposed matrix vector products but slow row slicing operations, where the CSR format excels. Note that matrix-vector products are also efficient with CSC, though the CSR format may be faster.

5.6.2.3 Other sparse formats

There are other formats which are suited for different use-cases. A use case that is related to this work is application in GPUs. The access pattern of (standard) sparse formats might not allow the GPU to reach its peak capability. For this reason, there are several “general-purpose” matrix formats that are more GPU oriented, like the Ellpack-Itpack (ELL) which is demonstrated by NVIDIA in [80]. Alternatively, or perhaps in conjunction with such formats, characteristic patterns in the matrix should be exploited to improve GPU hardware utilization.

5.7 Matrix multiplication

Table 5.4 shows the number of calculations required for the multiplication of various types of matrices. The type of the result also implies the amount of memory needed for it. Note that in general the result is dense (e.g. upper triangular dense, lower triangular dense etc) even if the operands of the multiplication are not (for example if they are sparse).

Matrix	Multiplied with	Result	Multiplications	Ratio with n^3 for large n
Upper Triangular	Dense Matrix (incl. Symmetric)	Dense Matrix	$\frac{n^2(n+1)}{2}$	$\frac{1}{2}$
	Upper Triangular Matrix	Upper Triangular Matrix	$\frac{n(n+1)(n+2)}{6}$	$\frac{1}{6}$
	Lower Triangular Matrix	Dense Matrix	$\frac{n(n+1)(2n+1)}{6}$	$\frac{1}{3}$
	Diagonal Matrix	Upper Triangular Matrix	$\frac{n(n+1)}{2}$	$\frac{1}{2n}$
	vector	vector	$\frac{n(n+1)}{2}$	$\frac{1}{2n}$
Lower Triangular	Dense Matrix (incl. Symmetric)	Dense Matrix	$\frac{n^2(n+1)}{2}$	$\frac{1}{2}$
	Upper Triangular Matrix	Dense Matrix	$\frac{n(n+1)(2n+1)}{6}$	$\frac{1}{3}$
	Lower Triangular Matrix	Lower Triangular Matrix	$\frac{n(n+1)(n+2)}{6}$	$\frac{1}{6}$
	Diagonal Matrix	Lower Triangular Matrix	$\frac{n(n+1)}{2}$	$\frac{1}{2n}$
	vector	vector	$\frac{n(n+1)}{2}$	$\frac{1}{2n}$

Table 5.4: Matrix multiplication calculations for various types of matrices

Table 5.4 is based on the standard matrix multiplication which runs in $O(n^3)$. There are algorithms that have a smaller lower bound.

The Strassen algorithm [81] is an algorithm for matrix multiplication which is faster than standard matrix multiplication and is useful in practice for large matrices. Strassen published this algorithm in 1969. Although his algorithm is only slightly faster than the standard algorithm for matrix multiplication, he was the first to point out that the standard approach is not optimal. The Strassen algorithm runs in $O(n^{\log_2 7}) = O(N^{2.8074})$. However, the algorithm exhibits somewhat reduced numerical stability and it requires significantly more memory compared to the standard algorithm.

For multiplication of square matrices, the Coppersmith-Winograd algorithm [82] manages to lower the complexity to $O(N^{2.375477})$. Since its original presentation in 1990, improvements in 2010 lowered the complexity to $O(N^{2.3736})$ and in 2011 to $O(N^{2.3727})$. However, unlike the Strassen algorithm, it is not used in practice because it only provides an advantage for matrices so large that they cannot be processed by modern hardware anyway.

Even though the optimal matrix multiplication may have yet to be discovered, it has a lower bound of $\Omega(n^2)$ because the resulting matrix has a number of values proportional to n^2 . The n^2 bound is still very limiting to the size of the matrices that can be processed. Furthermore, due to the result having $\sim n^2$ values, the space required for the result is also proportional to n^2 . So, even if lower memory formats (sparse, skyline, etc) are used for the input matrices to keep the space usage low, the memory requirement will still be $\sim n^2$ due to the result matrix.

The $\sim n^2$ running time and memory requirement makes matrix multiplication prohibitively expensive for large matrices. When a matrix multiplication is involved in calculations, the calculations are not performed until a vector appears next to them (see section 5.8).

5.8 Order of calculations

Let \mathbf{M}_1 , \mathbf{M}_2 be two matrices whose multiplication yields:

$$\mathbf{M}_1 \mathbf{M}_2 = \mathbf{R} \quad (5.93)$$

In general, the dimensions are:

$$[a \times b][b \times c] = [a \times c] \quad (5.94)$$

Each row of \mathbf{M}_1 has b values, as many as the values of a column of \mathbf{M}_2 .

To multiply a row of \mathbf{M}_1 with a column of \mathbf{M}_2 the calculations required are: b multiplications which will yield b values that need to be summed, and $b-1 \simeq b$ additions to add them.

In order to multiply all a rows of \mathbf{M}_1 , we require: ab multiplications and ab additions.

In order to multiply all columns, the total effort required is: abc multiplications and abc additions.

Assume that there are three matrices to multiply, with dimensions:

$$[a \times b][b \times c][c \times d] \quad (5.95)$$

The multiplication is mathematically correct regardless of the order the matrices are multiplied in due to the associative property of matrix multiplication. However, there is a difference in the number of calculations performed. The calculations required for the two ways of ordering the calculations is shown in Table 5.5 and Table 5.6. Table 5.5 shows the calculations from left to right while Table 5.6 shows the calculations from right to left.

Step		Multiplications	Additions
1	$[a \times b][b \times c] = [a \times c]$	abc	abc
2	$[a \times c][c \times d] = [a \times d]$	acd	acd
Total		$abc + acd$	$abc + acd$

Table 5.5: Multiplication from left to right

Step		Multiplications	Additions
1	$[b \times c][c \times d] = [b \times d]$	bcd	bcd
2	$[a \times b][b \times d] = [a \times d]$	abd	abd
Total		$bcd + abd$	$bcd + abd$

Table 5.6: Multiplication from right to left

Note that the number of calculations required is different. Therefore, depending on the size of the matrices, the order of calculations is important for the efficiency of calculating the result, even though the result is always the same (barring arithmetic differences). For example:

$$[1000 \times 800][800 \times 600][600 \times 400][400 \times 200] \quad (5.96)$$

Step		Multiplications/Additions
1	$[1000 \times 800][800 \times 600] = [1000 \times 600]$	$1000 \cdot 800 \cdot 600 = 48 \cdot 10^7$
2	$[1000 \times 600][600 \times 400] = [1000 \times 400]$	$1000 \cdot 600 \cdot 400 = 24 \cdot 10^7$
3	$[1000 \times 400][400 \times 200] = [1000 \times 200]$	$1000 \cdot 400 \cdot 200 = 8 \cdot 10^7$
Total		$80 \cdot 10^7 = 800 \cdot 10^6$

Table 5.7: Example: multiplication from left to right

Step		Multiplications/Additions
1	$[600 \times 400][400 \times 200] = [600 \times 200]$	$600 \cdot 400 \cdot 200 = 48 \cdot 10^6$
2	$[800 \times 600][600 \times 200] = [800 \times 200]$	$800 \cdot 600 \cdot 200 = 96 \cdot 10^6$
3	$[1000 \times 800][800 \times 200] = [1000 \times 200]$	$1000 \cdot 800 \cdot 200 = 160 \cdot 10^6$
Total		$304 \cdot 10^6$

Table 5.8: Example: multiplication from right to left

Table 5.7 and Table 5.8 show that the number of operations from right to left are significantly less. In general, starting calculations from the smaller matrices leads to significantly less number of calculations in total. The order of operations is taken into account in some linear algebra libraries, like Armadillo². Fig. 5.29 shows the multiplication of 4 matrices $\mathbf{Z} = \mathbf{A} \mathbf{B} \mathbf{C} \mathbf{D}$ for three linear

² <http://arma.sourceforge.net/speed.html>

algebra libraries and highlights the difference between doing the operations in the most efficient order.

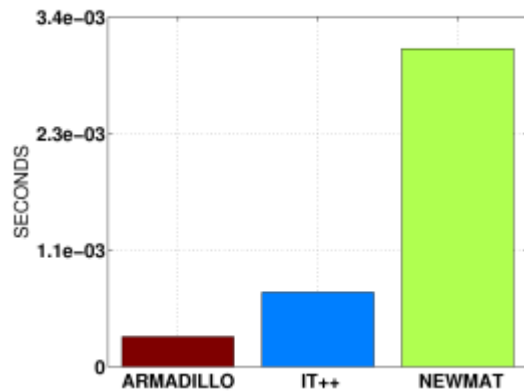
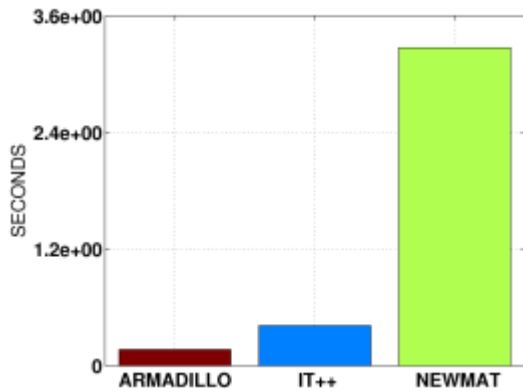
Assume that the result matrix must then be multiplied with a vector:

$$[1000 \times 800][800 \times 600][600 \times 400][400 \times 200][200 \times 1] \quad (5.97)$$

Multiplication of three matrices followed by a vector is common. For example, in PCPG there is $\mathbf{y}_{m-1} = \mathbf{P} \tilde{\mathbf{F}}^{-1} \mathbf{P} \mathbf{r}_{m-1}$. So a natural idea is to calculate the matrices first especially considering that they remain unchanged for all iterations of process, unlike the vector. In order to make the matrix-vector multiplication $[1000 \times 200][200 \times 1] = [1000 \times 1]$, an additional $1000 \cdot 200 \cdot 1 = 2 \cdot 10^5$ multiplications and equal number of additions are required. This is pretty insignificant compared to the calculations already done for the matrix-matrix multiplications so we ignore them.

On the other hand, if the calculations are not performed until the vector appears on the right and then the calculations were performed from right to left as a series of matrix-vector multiplications, then the number of operations required is shown in Table 5.9. The collective results of the various options are shown in Table 5.10. The number of operations in the last case are orders of magnitude less.

It is clear that it is preferable to only perform matrix-vector operations. There is additional gain when considering that a matrix-matrix multiplication would result in a dense matrix (see Section 5.7) while performing only matrix-vector multiplication keeps the memory requirements to a minimum. Furthermore, an implementation of matrix-matrix multiplication that takes all the possible combinations (for the type of matrix e.g. triangular as well as the storage format e.g. skyline) into account is complex. Relying on matrix-vector multiplication is significantly easier in an efficiency-oriented implementation.



$$\mathbf{A}=[100 \times 80], \mathbf{B}=[80 \times 60]$$

$$\mathbf{C}=[60 \times 40], \mathbf{D}=[40 \times 20]$$

10× faster with proper ordering of operations

$$\mathbf{A}=[1000 \times 800], \mathbf{B}=[800 \times 600]$$

$$\mathbf{C}=[600 \times 400], \mathbf{D}=[400 \times 200]$$

20× faster with proper ordering of operations

Fig. 5.29: Ordering of operations in linear algebra libraries

Step		Multiplications/Additions
1	$[400 \times 200][200 \times 1]=[400 \times 1]$	$400 \cdot 200 \cdot 1 = 8 \cdot 10^4$
2	$[600 \times 400][400 \times 1]=[600 \times 1]$	$600 \cdot 400 \cdot 1 = 24 \cdot 10^4$
3	$[800 \times 600][600 \times 1]=[800 \times 1]$	$800 \cdot 600 \cdot 1 = 48 \cdot 10^4$
4	$[1000 \times 800][800 \times 1]=[1000 \times 1]$	$1000 \cdot 800 \cdot 1 = 80 \cdot 10^4$
Total		$152 \cdot 10^4$

Table 5.9: Multiplication from right to left with vector in the right

	Multiplications/Additions
Left to right	$80000 \cdot 10^4 + 20 \cdot 10^4$
Right to left matrices first, then vector	$30400 \cdot 10^4 + 20 \cdot 10^4$
Right to left	$152 \cdot 10^4$

Table 5.10: Multiplication of 3 matrices and a vector

5.9 Transpose

Explicit transposition of a matrix can usually be avoided. Specific handling can perform the operations directly from the initial matrix without having to explicitly transpose it. An exception might be when the transposed matrix is be reused a large number of times. In this case, transposing the matrix in order to have the entries in consecutive memory locations with respect to the operations the matrix is reused in might be beneficial.

As for vectors, the transpose is only a semantic and can always be avoided. The previous reason stated for matrices is not applicable for vectors. The vector transpose is only symbolic so that the dimensions of the operands are consistent. For example, if \mathbf{x} is $n \times 1$, \mathbf{x}^T is $1 \times n$ so in order to left multiply it with a $n \times n$ matrix the expression $\mathbf{x}^T \mathbf{A}$ would be consistent.

The following equality is useful:

$$(\mathbf{A B})^T = \mathbf{B}^T \mathbf{A}^T \quad (5.98)$$

When applied to matrix with vector:

$$\begin{aligned} \mathbf{y} = \mathbf{A x} &\Leftrightarrow \mathbf{y}^T = \mathbf{x}^T \mathbf{A}^T \\ \mathbf{v} = \mathbf{A}^T \mathbf{w} &\Leftrightarrow \mathbf{v}^T = \mathbf{w}^T \mathbf{A} \end{aligned} \quad (5.99)$$

This means that left multiplying a matrix with a vector can be indirectly performed with the standard matrix-vector multiplication. The only difference is seemingly that the result is a transposed vector but, as it has already been established transposition in vectors is inconsequential.

5.10 Matrices as a collection of vectors

It is sometimes convenient to view a matrix as a collection of vectors (this can be useful in case of multiple right hand sides for example). Matrix \mathbf{X} with dimensions $m \times n$ can be decomposed as follows:

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & & \vdots \\ X_{m1} & X_{m2} & \dots & X_{mn} \end{bmatrix} = \left[\begin{bmatrix} X_{11} \\ X_{21} \\ \vdots \\ X_{m1} \end{bmatrix} \begin{bmatrix} X_{12} \\ X_{22} \\ \vdots \\ X_{m2} \end{bmatrix} \dots \begin{bmatrix} X_{1n} \\ X_{2n} \\ \vdots \\ X_{mn} \end{bmatrix} \right] = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \dots \quad \mathbf{x}_n] \quad (5.100)$$

where each vector \mathbf{x}_i $i=1, \dots, n$ has size $m \times 1$.

For example, in order to multiply a matrix \mathbf{A} with matrix \mathbf{X} , where there is already an efficient matrix-vector multiplication for matrix \mathbf{A} , the multiplication $\mathbf{A}\mathbf{X}$ can be performed with consecutive matrix-vector multiplications. The result is another collection of vectors:

$$\mathbf{A}\mathbf{X} = \mathbf{Y} = \left[\begin{bmatrix} Y_{11} \\ Y_{21} \\ \vdots \\ Y_{m1} \end{bmatrix} \begin{bmatrix} Y_{12} \\ Y_{22} \\ \vdots \\ Y_{m2} \end{bmatrix} \dots \begin{bmatrix} Y_{1n} \\ Y_{2n} \\ \vdots \\ Y_{mn} \end{bmatrix} \right] = [\mathbf{y}_1 \quad \mathbf{y}_2 \quad \dots \quad \mathbf{y}_n] \quad (5.101)$$

Similarly, matrix \mathbf{X} can be decomposed to as:

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & & \vdots \\ X_{m1} & X_{m2} & \dots & X_{mn} \end{bmatrix} = \left[\begin{bmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & & \vdots \\ X_{m1} & X_{m2} & \dots & X_{mn} \end{bmatrix} \right] = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} \quad (5.102)$$

For example, in order to multiply a matrix \mathbf{X} with matrix \mathbf{A} , where there is already an efficient left multiplication for matrix \mathbf{A} , the multiplication $\mathbf{X}\mathbf{A}$ can be performed with consecutive vector-matrix left multiplications. The result is another collection of vectors:

$$\mathbf{X}\mathbf{B} = \mathbf{Y} = \left[\begin{bmatrix} Y_{11} & Y_{21} & \dots & Y_{m1} \\ Y_{12} & Y_{22} & \dots & Y_{m2} \\ \vdots & \vdots & & \vdots \\ Y_{1n} & Y_{2n} & \dots & Y_{mn} \end{bmatrix} \right] = \begin{bmatrix} \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_n^T \end{bmatrix} \quad (5.103)$$

6 Domain decomposition methods in hybrid CPU-GPU architectures

6.1 Introduction

In scientific computing, there is a constant need for solving new and highly computationally demanding problems with increased accuracy and enhanced numerical performance. In simulation-based applied science and engineering, there have been considerable improvements in sparse matrix solution algorithms as well as in domain decomposition methods to mitigate execution bottlenecks, thus leading to faster calculation times and reduced memory requirements for the solution of increasingly larger problems. To further increase the speed of their applications, scientists have also relied on advances in hardware and utilization of expensive specialized computing systems with parallel and/or distributed processing capabilities, as well as clusters of interconnected workstations. Moreover, since power density issues limit the increase of the clock frequency, manufacturers have turned to adding more cores to their processors. However, these advancements pose a challenge to software developers since sequential codes run on one of the cores and do not take advantage of the full processing capabilities. Parallel codes do not have this limitation, so incentive for their further development has increased, especially since they have the potential for exploiting the processing power of the graphics processing units (GPUs).

Driven by the demands of the gaming industry, graphics hardware has substantially evolved over the years with remarkable floating point arithmetic performance. These processing capabilities motivated the utilization of graphics hardware for general purpose applications, eventually leading to their initial use for non-graphic operations in 1999. In the early years, these operations had to be programmed indirectly, by mapping them to graphic manipulations and using graphic libraries such as OpenGL and DirectX. This approach of solving general purpose problems is known as general purpose computing on GPUs (GPGPU). Despite the cumbersome programming, it was soon apparent that the GPUs' potential and capabilities could be utilized for accelerating arithmetic operations, especially since they have considerably lower cost than current supercomputers or workstation clusters.

GPU programming was greatly facilitated with the initial release of the CUDA-SDK [48], [83], [84] in 2007, which resulted in a rapid development of GPU computing and the appearance of GPU-powered clusters on the Top500 supercomputers [51]. CUDA, which stands for “compute unified device architecture”, is a parallel computing architecture developed by NVIDIA. CUDA gives

developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs become easily accessible for general-purpose applications by eliminating the need for special casting. Recently, openCL has been released as an open industry standard to facilitate portability and vendor-independence, targeting heterogeneous platforms consisting of CPUs, GPUs as well as other types of processors [85]. Unlike CPUs, GPUs have an inherent parallel throughput architecture that focuses on executing many concurrent threads slowly, rather than executing a single thread very fast. Massive hardware multithreading aims to overcome latencies that inevitably derive from device communication. A comparison of the current GPU architecture as well as potential future GPU and modern multi-core processor architecture is provided in [86].

Work pertaining to GPUs has extended to a large variety of applications even before CUDA made their use easier. Non-linear finite element implementations for surgical simulation can be found in [87] with GPGPU and in [62] with CUDA. Engineering applications in the field of fluid mechanics [52]–[55], molecular dynamics [57], [58], topology optimization [59], wave propagation [60], Helmholtz problems with the boundary element method [61], have been recently reported on a variety of GPU platforms using explicit computational algorithms. Linear algebra applications have also been a topic of scientific interest for GPU implementations. A thorough analysis of algorithmic performance of basic linear algebra operations can be found in [64]. Performance of iterative solvers is analyzed in [65], while a parametric study of the PCG solver is performed on multi-GPU CUDA clusters in [66], [67]. A hybrid CPU-GPU implementation of dense linear algebra algorithms is reported in [63].

It should be noted that all implementations prior to CUDA 1.3 are performed in single-precision, since support for double-precision floating point operation is added on CUDA 1.3. This has caused some misinterpretations in a number of published comparisons between the GPU and the CPU, usually in favor of the GPU. However, GPUs were (and still are) perfectly suitable for mixed-precision solvers. Performance and accuracy of mixed-precision iterative and multigrid solvers is thoroughly discussed in [47].

Domain decomposition methods (DDM) constitute today an important category of methods for the solution of highly demanding problems in simulation-based applied science and engineering. Among them, dual domain decomposition methods have been successfully applied in a variety of problems in both sequential as well as in parallel/distributed processing systems. In this work, the

implementation of the FETI method is demonstrated in hybrid CPU-GPU computing platforms [68]. DDM are generally considered unsuitable for GPU applications due to their difficulty in exploiting the full capacity of the fine-grained parallelism of the GPUs. However, this weakness of DDM is overcome in the proposed implementation, with customized parallelization routines applied for every part of the solution algorithm. Parametric tests on an implicit finite element structural mechanics benchmark problem reveals the tremendous potential of this type of hybrid computing environment as a result of the full exploitation of the intrinsic software and hardware features of the GPUs as well as the numerical properties of the solution method.

6.2 Dual DDM (FETI) method

The Dual-DDM (FETI) method is implemented in a hybrid CPU-GPU computing environment. The details of the implementation choices used for this application are described here. Assuming a decomposition of the domain in n_s non-overlapping subdomains

The solution of the interface problem:

$$\begin{bmatrix} \mathbf{F} & -\mathbf{G} \\ -\mathbf{G}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ -\mathbf{e} \end{bmatrix} \quad (6.1)$$

is based on a projected PCG algorithm (Section 3.2.7), where the vector search space is projected to a different subspace using the following projector:

$$\mathbf{P} = \mathbf{I} - \mathbf{G}(\mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T \quad (6.2)$$

The algorithmic description of the resulting PCPG algorithm is given in Fig. 6.1.

Initialization

$$\boldsymbol{\lambda}_0 = \mathbf{G} (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{e} \quad (6.3)$$

$$\mathbf{w}_0 = \mathbf{P}^T (\mathbf{d} - \mathbf{F} \boldsymbol{\lambda}_0) \quad (6.4)$$

Iterate $k=0,1,\dots$, until convergence

$$\mathbf{y}_k = \mathbf{P} \tilde{\mathbf{F}}^{-1} \mathbf{w}_k \quad (6.5)$$

$$\mathbf{p}_k = \mathbf{y}_k - \sum_{i=0}^{k-1} \frac{\mathbf{y}_k^T \mathbf{F} \mathbf{p}_i}{\mathbf{p}_i^T \mathbf{F} \mathbf{p}_i} \mathbf{p}_i \quad (6.6)$$

$$\eta_k = \frac{\mathbf{p}_k^T \mathbf{w}_k}{\mathbf{p}_k^T \mathbf{F} \mathbf{p}_k} \quad (6.7)$$

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \eta_k \mathbf{p}_k \quad (6.8)$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \mathbf{P}^T \mathbf{F} \mathbf{p}_k \quad (6.9)$$

Fig. 6.1: The PCPG algorithm used for the solution of the interface problem

$\tilde{\mathbf{F}}^{-1}$ being is an appropriate preconditioner. The matrix-vector multiplication $\mathbf{F} \mathbf{p}_i$ in eq. (6.6) is performed implicitly as follows: Map \mathbf{p}_i to \mathbf{p}_i^s using the \mathbf{B}^T operator, solve $\mathbf{K} \mathbf{p}^s = \mathbf{p}_i^s$ for \mathbf{p}^s and then map \mathbf{p}^s again using the \mathbf{B} operator. The lumped preconditioner with scaling is used:

$$\tilde{\mathbf{F}}^{-1} = \sum_{s=1}^{n_s} (\mathbf{M}^s)^{-1} \mathbf{B}^s \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{K}_{\text{bb}}^s \end{bmatrix} (\mathbf{B}^s)^T (\mathbf{M}^s)^{-1} \quad (6.10)$$

In the past, a number of versions of the Lagrange mapping operator that incorporate scaling effects have been used in the preconditioning step of the FETI method [40]. In the case of redundant Lagrange multipliers and homogeneous problems, the Lagrange mapping operator in the preconditioning step can be written as

$$\mathbf{B}_p = (\mathbf{M}^s)^{-1} \mathbf{B} \quad \mathbf{B}_{p_b} = (\mathbf{M}_b^s)^{-1} \mathbf{B}_b \quad (6.11)$$

where \mathbf{M}^s and \mathbf{M}_b^s are diagonal matrices whose diagonal entries contain the multiplicity of the corresponding dof. An extensive investigation of DDM mapping operators and their interconnection can be found in [26], [27]. The lumped preconditioner becomes:

$$\tilde{\mathbf{F}}^{-1} = \sum_{s=1}^{N_s} \mathbf{B}_p^s \mathbf{K}^s (\mathbf{B}_p^s)^T \quad (6.12)$$

6.3 Hybrid CPU-GPU implementation

The Dual DDM FETI solver has been implemented in hybrid CPU-GPU workstations with the purpose of exploiting all available processing power and memory resources in order to handle even larger problems. Due to the fact that the CPU and GPU platforms are heterogeneous and feature different programming paradigms, special considerations had to be made in a number of steps of the FETI algorithm to achieve optimum efficiency. One of the main issues which has to be dealt with is the difference in performance between the CPU and GPU, which is mainly affected by the arithmetic operation being executed as well as by other parameters. Furthermore, this difference in performance between the CPU and GPU is not the same when calculating dot products, executing matrix-vector multiplications or solving linear systems directly with the Cholesky factorization.

The most important step of the FETI algorithm, from the computer implementation point of view, is the calculation of \mathbf{d} of eq. (6.1), since it involves the solution of the local subdomain problems and the subdomain data to be handled by the CPU and GPU memory. Two different implementations have been considered for the solution of local subdomain problems. The first one performs the solution of local problems with the direct Cholesky solver and the second one with the iterative PCG solver. These methods, apart from being quite different in their parallel programming implementation, also feature different memory needs which affect the amount of subdomain data processed by the CPU and GPU.

6.3.1 The Choleksy direct solver

The Cholesky direct solver for computing the solution of $\mathbf{K}\mathbf{u}=\mathbf{f}$ comprises the following steps:

- Factorization of matrix \mathbf{K} to the form $\mathbf{K}=\mathbf{L}\mathbf{D}\mathbf{L}^T$
- Forward substitution so that $\mathbf{L}\mathbf{x}_1=\mathbf{f}\Leftrightarrow\mathbf{x}_1=\mathbf{L}^{-1}\mathbf{f}$ and trivial solution of $\mathbf{D}\mathbf{x}_2=\mathbf{x}_1\Leftrightarrow\mathbf{x}_2=\mathbf{D}^{-1}\mathbf{x}_1$ since \mathbf{D} is diagonal.
- Backward substitution so that $\mathbf{L}^T\mathbf{u}=\mathbf{x}_2\Leftrightarrow\mathbf{u}=\mathbf{L}^{-T}\mathbf{x}_2$

For the case of solving a problem with multiple or repeated right-hand sides, the factorization process is carried out once and, for each right-hand side, the forward and backward substitution steps are performed. The factorization process is a recursive operation which consists of the following steps:

$$\mathbf{D}_j = \mathbf{K}_{kk} - \sum_{k=1}^{j-1} \mathbf{L}_{jk}^2 \mathbf{D}_k \quad (6.13)$$

$$\mathbf{L}_{ij} = \frac{1}{\mathbf{D}_j} \left(\mathbf{K}_{ij} - \sum_{k=1}^{j-1} \mathbf{L}_{jk} \mathbf{L}_{jk} \mathbf{D}_k \right) \quad (6.14)$$

where the indexes define the position of the matrix where the corresponding value is present. Memory consumption is increased for the direct solver since for each subdomain, we need to store the stiffness matrix both in a compressed sparse row (CSR) format, in order to use it for the preconditioning step of eq. (6.5) with the lumped type preconditioner of eq. (6.12), and in skyline format, in order to perform the factorization of the subdomain matrices for the solution of local subdomain problems.

The proposed strategy for the parallel implementation of the factorization process in the GPU is different from what is usually implemented in a parallel sparse solver. Parallel sparse solvers try to utilize all available processors in order to process partial data from one big sparse matrix. There are open issues as to how the rather poor scalability of parallel sparse solvers can be improved, especially in very fine-grained parallelism of GPU architectures. For the case of domain decomposition methods, primal or dual, there is no big sparse matrix but rather hundreds, or even thousands, for the case of large-scale problems, of smaller sparse matrices. This enables us to take advantage of the numerical scalability properties of the FETI method and fully exploit the GPU's fine-grained parallelism by assigning each subdomain matrix factorization process to a warp of threads. This strategy allows the utilization of all available GPU cores and use shared memory for parallel reductions without the need of synchronization points. The same strategy and benefits hold true for the forward and backward substitutions performed for the solution of subdomain problems.

One of the main concerns when implementing GPU kernels for execution is thread occupancy. In order to fully exploit the capabilities of the GPU, the streaming multiprocessors (SMPs) have to be overloaded with work which essentially means that the number of simultaneous running threads has to be much larger than the quantity of SMPs. This happens because global memory access is very slow so the GPU scheduler suspends a thread accessing global memory until the requested data is fetched from it. In the meantime, the GPU executes another thread that has its data available in local memory for processing. In order to evaluate occupancy for the case of parallel Cholesky factorizations of subdomain matrices, a parametric study was conducted with respect to the amount of concurrent matrix factorizations being computed at the GPUs used for this work. The results are

shown in Fig. 6.2, where it is evident that computing time is practically stabilized for 10 concurrent factorization computations and above. Taking these results into account, the GPU is constantly loaded with more than 10 concurrent matrix factorizations and forward and backward substitutions.

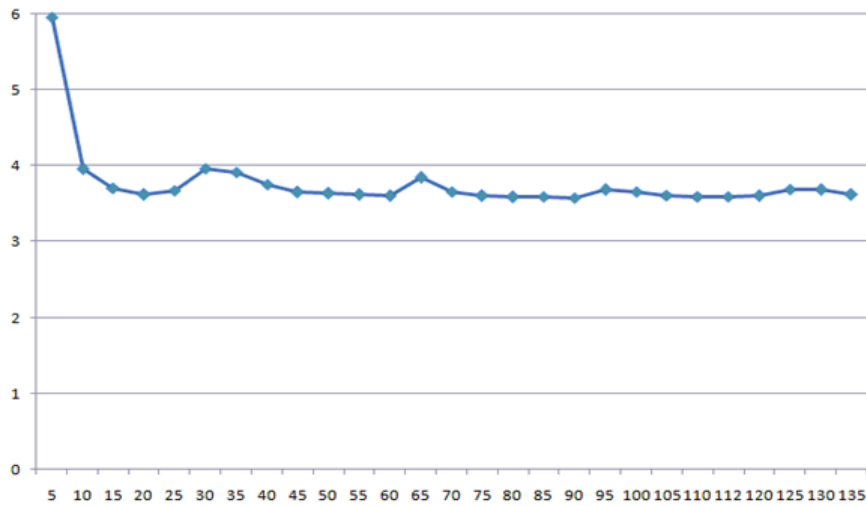


Fig. 6.2. Time in ms for factorizing a subdomain kernel. Horizontal axis represents the simultaneous factorizations computed at the GPU in parallel

6.3.2 The PCG iterative solver

The PCG iterative solver for computing the solution of $\mathbf{K} \mathbf{u} = \mathbf{f}$, comprises the following steps with preconditioner $\tilde{\mathbf{K}}$:

Initialization

$$\mathbf{r}_0 = \mathbf{f} - \mathbf{K} \mathbf{u}_0, \quad \mathbf{p}_0 = \mathbf{z}_0 = \tilde{\mathbf{K}}^{-1} \mathbf{r}_0, \quad \mathbf{q}_0 = \mathbf{K} \mathbf{p}_0, \quad \eta_0 = \frac{\mathbf{p}_0^T \mathbf{r}_0}{\mathbf{p}_0^T \mathbf{q}_0} \quad (6.15)$$

Iterate $k=0,1,\dots$, until convergence

$$\mathbf{u}_k = \mathbf{u}_{k-1} + \eta_{k-1} \mathbf{p}_{k-1} \quad (6.16)$$

$$\mathbf{r}_k = \mathbf{r}_{k-1} - \eta_{k-1} \mathbf{q}_{k-1} \quad (6.17)$$

$$\mathbf{z}_k = \tilde{\mathbf{K}}^{-1} \mathbf{r}_k \quad (6.18)$$

$$\mathbf{p}_k = \mathbf{z}_k - \sum_{i=0}^{k-1} \frac{\mathbf{z}_k^T \mathbf{q}_i}{\mathbf{p}_i^T \mathbf{q}_i} \mathbf{p}_i \quad (6.19)$$

$$\mathbf{q}_k = \mathbf{K} \mathbf{p}_k \quad (6.20)$$

$$\eta_k = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{q}_k} \quad (6.21)$$

Fig. 6.3: The PCG algorithm

For the subdomain problems solved in this work, the diagonal preconditioner is used for performing step (6.18) and the coefficient matrices \mathbf{K}^s of step (6.20) are stored in CSR format (Section 5.6.2.1). In the case of semi-positive definite matrices, the null space of matrix \mathbf{K} cannot be derived as a by-product of the PCG solution, as in the case of the Cholesky solver. For this case, the analytical evaluation of the rigid body modes (Section 3.2.3) is applied.

6.3.3 The solution at the projection step

The projection matrix-vector multiplication encountered in eqs. (6.4) and (6.9) involves the solution of

$$\mathbf{G}^T \mathbf{G} \mathbf{x}_1 = \mathbf{x}_2 \Leftrightarrow \mathbf{x}_1 = (\mathbf{G}^T \mathbf{G})^{-1} \mathbf{x}_2 \quad (6.22)$$

at the initialization step and at each PCPG iteration, respectively, where \mathbf{x}_1 , \mathbf{x}_2 are temporary vectors. This solution is usually performed with a direct solver since the order of the coefficient matrix $\mathbf{G}^T \mathbf{G}$ is related to the rigid body modes of the floating subdomains and is thus small for a coarse to a medium grained subdivision. In our implementation, the size of this matrix may not be negligible due to the fine grained decomposition of the domain which is better suited for a GPU environment. Bearing in mind that this matrix is global, spanning across the whole domain and that it is not associated with subdomains, a direct solver is generally not appropriate to perform this task. For this reason, a PCG solver with a diagonal preconditioner is applied in parallel at each projection step of the PCPG algorithm.

Furthermore, since the solution of this problem is performed at each PCPG iteration, the re-orthogonalization procedure performed in eq. (6.19) is applied with search vectors computed in previous PCPG iterations as well. This implementation is impractical when applied to the full problem $\mathbf{K} \mathbf{u} = \mathbf{f}$ due to excessive storage requirements. However, this methodology can be efficiently utilized for the projection step, where the size of the $\mathbf{G}^T \mathbf{G}$ matrix is small compared to the global matrix, which significantly accelerates the convergence of PCG for subsequent solutions. This implementation is also performed for the solution of the subdomain problems with PCG since the solution is also repeated at each PCPG iteration and the size of the subdomain problems is small particularly for fine-grained subdivisions.

6.3.4 Dot products

Apart from the presence of dot products in sparse matrix vector (SpMV) multiplications, both PCPG and PCG algorithms feature a number of dot product computations at each iteration. Specifically, during the re-orthogonalization step (eqs. 6.6 and 6.19 for the PCPG and PCG algorithms respectively), these dot products can consume a non-negligible amount of processing power and for this reason, they have to be implemented efficiently. Furthermore, during the Cholesky factorization and the forward-backward substitutions a large number of dot products are

performed.

A dot product operation can be separated into two discrete tasks. The first consists of multiplying the elements of each vector one by one and the second task consists of computing the sum of each of these products for obtaining the final result. The multiplication step is inherently parallel making it an excellent candidate for implementation on a GPU. In this work, the product of the elements of each vector are stored in a vector which overwrites the contents of the first vector by a simple GPU kernel of the form $a[i]=a[i]\cdot b[i]$. On the other hand, the summation process is not that trivial and needs a reduction operation (Section 4.12). From the options available to finalized the reduction process, the reduction implementation in this work uses the option of launching an additional kernel with a single thread block.

6.3.5 Sparse matrix – vector multiplications

At every PCPG iteration, the preconditioning step (eq. 6.5) is applied in order to improve the convergence rate of the method. These preconditioning matrices depend on the stiffness matrices of each subdomain which are stored in CSR format and, at the time that the preconditioning step is executed, they are multiplied by a given vector. Similar matrix-vector multiplications are performed in step (6.20) of the PCG algorithm and in the solution of the projection step of eq. (6.22) with PCG. In order to achieve maximum efficiency of this time-consuming operation, an optimized CUDA kernel calculating the result of a SpMV multiplication has to be implemented.

Since the (sparse) dot product between a row of the stiffness matrix and the given vector may be computed independently of all other rows, the CSR SpMV operation is easily parallelized using one thread per matrix row. Several variants of this approach are documented in [88]. While this approach exhibits fine-grained parallelism, its performance suffers mainly by the way in which threads within a warp access the CSR indices and data arrays. Specifically, while the column indices and nonzero values for a given row are stored contiguously in the CSR data structure, these values are not accessed simultaneously but are read sequentially by each thread. Moreover, when this implementation strategy is applied to a matrix with a highly variable number of non-zeros per row, it is likely that many threads within a warp will remain idle while the thread with the longest row continues iterating, thus resulting to poor GPU utilization.

In order to circumvent this weakness, an alternative algorithm is implemented in this work where

one warp is assigned to each matrix row. Unlike previous approaches, which use one thread per matrix row, the implemented kernel features a warp-wide parallel reduction to sum the per-thread results together which requires coordination among threads within the same warp. Moreover, shared memory is used for the summation process which greatly improves the performance of this algorithm, while indexes and data are accessed contiguously, therefore overcoming the principal deficiency of the approaches documented in [88]. The only limitation of this implementation is that its efficient execution demands that matrix rows contain a number of non-zeros greater than the warp size (32 for current CUDA 2.0 compute capability GPUs), which is not an issue for large-scale problems.

6.4 Dynamic load-balancing

6.4.1 Task Parallelism

The heterogeneity of computer components has been addressed in this work by implementing a dynamic load balancing procedure based on task queues. In particular, the CPU creates a queue of tasks that have to be executed at a certain step of the algorithm. In the case of the direct Cholesky solver, the subdomain matrices are stored in skyline format and are factorized in parallel by both the CPU and the GPU. A queue of tasks is created for performing the factorization and the forward and backward substitutions. This queue is filled with the appropriate subdomain matrices and the CPU and GPU are fed with tasks in an asynchronous manner. Upon finishing the corresponding calculation, they pull another task from the queue, as is schematically shown in Fig. 6.4. Thus, both CPU and GPU are constantly busy with calculations until the queue is emptied.

For the case of the PCG solver with diagonal preconditioner, the most time consuming operation is the sparse matrix-vector multiplication (SpMV) of eq. (6.20) between the subdomain stiffness matrices and the corresponding search vectors. The same SpMV operation is required for the solution of the preconditioning step in eq. (6.5) of the PCPG algorithm, while a similar operation is performed during the solution of the projection step of eq. (6.22) with PCG. For all these cases, a typical queue of tasks is created, as with the Cholesky solver, which is filled with the appropriate subdomain matrices and their corresponding vectors, while the CPU and GPU are fed with tasks from the queue for performing the SpMV multiplications in an asynchronous manner.

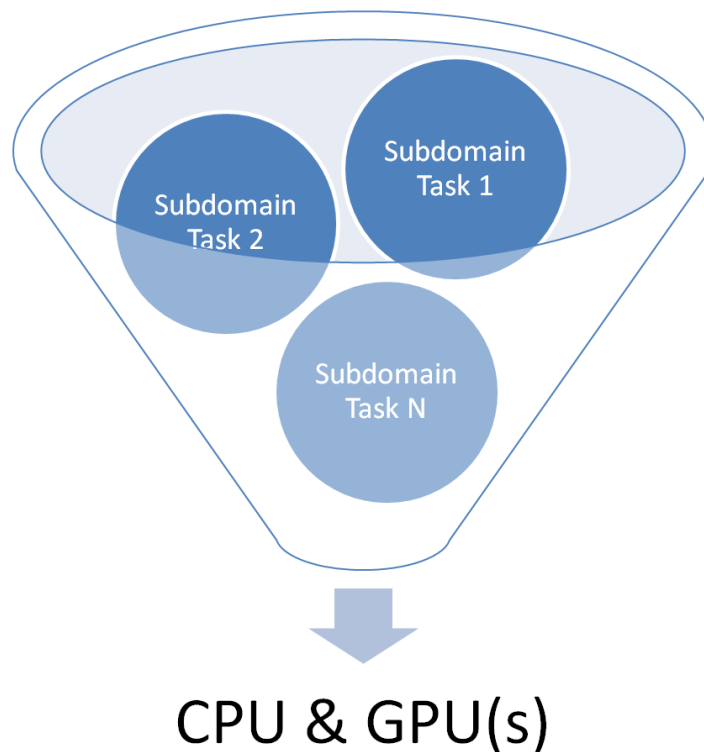


Fig. 6.4. The task queue contains numerical computations to be performed by available resources

6.5 Dynamic load-balancing implementation

An implementation of the task parallelism on a typical workstation, featuring an x -core CPU with y GPUs, consists of $x + y + 1$ independent CPU threads executing concurrently. The threads that actually perform numerical computations are the x ones, called “CPU threads”, while the y threads, called “GPU control threads”, simply instruct each GPU to launch a specific GPU kernel for execution. The last thread is the “master thread” which is responsible for managing the task queue defining the numerical computations to be performed and the data structures that participate to that specific computation. All these threads are executing a while-loop which, for each case, has a different body and a different termination criterion.

Each CPU thread while-loop body consists of the actual function calls needed to perform the computations described by the task that was fetched from the task queue. Since memory access to the CPU is concurrent, memory has to be locked in order for the assigned tasks to be deleted from the task queue, thus avoiding the infamous race conditions. CPU threads are synchronous which

means that each program statement must finish executing before the next starts its execution. When all computations for the given task have finished executing, the while-loop termination criterion checks for any remaining tasks in the queue. When there are no tasks left, the thread is terminated and the master thread is notified that this CPU thread has terminated.

On the other hand, GPU threads are asynchronous which means that GPU kernels will be launched concurrently when there is a series of program statements that launches GPU kernels. CUDA provides an event mechanism which notifies the launching thread when a specific kernel has finished executing. This mechanism is used in this work, in order to orchestrate the flow of GPU kernel execution. Thus, the while-loop body of a GPU thread consists of GPU kernel launches corresponding to the actual calculations to be performed, followed by another inner while-loop whose body performs Thread.Sleep operations. Thread.Sleep provides an elegant mechanism for a thread to wait without blocking the operating system. The termination criterion of the inner while-loop checks for incoming kernel termination events while the termination criterion of the outer while-loop is similar to that of a CPU thread. Similar to the case of a CPU thread, the thread is terminated when there are no tasks left and the master thread is notified that this GPU thread has terminated.

The master thread's body contains program statements that handle memory allocation, perform memory copies from the CPU memory to the GPU global memory and vice versa, build the appropriate task queues and spawn the CPU and GPU threads which execute the discrete tasks contained in a task queue. After the CPU and GPU threads are spawned, the master thread executes a while-loop similar to the inner while-loop of a GPU thread which waits for CPU and GPU thread termination. Following the concept of CPU, GPU and master threads, the PCPG algorithm is implemented and executed in a parallel environment with the master thread's source code having the look and feel of a serial program. This is a very important feature for source code maintainability, extensibility and debugging since all internal work associated with the parallel implementation is encapsulated to the CPU and GPU threads.

6.6 Numerical results

In order to assess the efficiency of the previously discussed implementations, their performance is demonstrated in a parametric study of 3D linear elasticity problems ($E=39\text{ MPa}$, $\nu=0.2$) with a cubic geometry. The domain is fully restrained at the bottom surface, partially restrained on the horizontal directions of the side surfaces while the top surface is subjected to a distributed load, and are discretized with 8-node hexahedral finite elements, resulting in linear systems with 1,058,610 dof. The number of subdomains ranges from 125 to 2744 (see Section 2.3). More detailed results are provided in [89].

Two workstations are used, both having an Intel Core i7-950 Processor, 3.06 GHz, which has 4 physical cores/8 logical cores and 8 MB cache along with 6 GB RAM. Two different NVIDIA GPUs are tested independently: GTX285 and GTX580 with 1GB GDDR3 and 1.5 GB GDDR5 memory, respectively. The workstation is adequate for performing all calculations in double precision without disk caching.

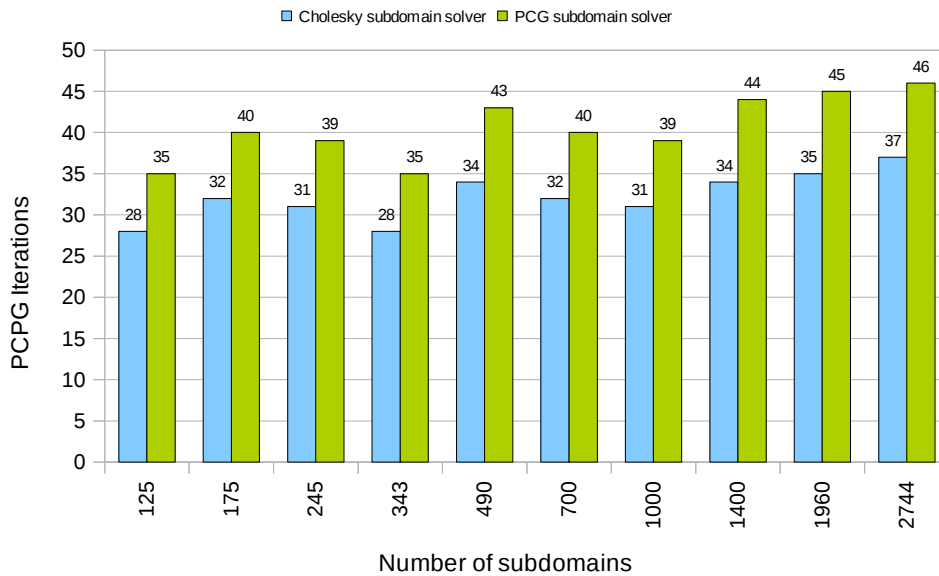


Fig. 6.5: Number of iterations for the PCPG solution of the interface problem with Cholesky and PCG subdomain solvers

Fig. 6.5 shows the required number of iterations for the PCG solution ($\varepsilon=10^{-4}$) of the subdomain problems with re-orthogonalization (mean values for all repeated solution within PCPG) and the interface problem with PCPG ($\varepsilon=10^{-4}$). The convergence behavior of PCG for the solution of the subdomain problems and for the projection step problem ($\varepsilon=10^{-7}$), with and without re-

orthogonalization, is shown in Table 6.1, where one order of magnitude reduction on the number of iterations is achieved with the re-orthogonalization procedure for solving the repeated right-hand side problems.

Number of subdomains	GtG dof	Subdomain Problems		Projection Step Problem	
		Iterations without	Iterations with	Iterations without	Iterations with
		reorthogonalization		reorthogonalization	
125	720	219	18	47	5
175	1020	168	13	67	7
245	1440	150	12	95	11
343	2016	136	11	133	15
490	2898	119	10	210	23
700	4158	102	8	301	33
1000	5940	86	7	429	48
1400	8340	74	6	603	67
1960	11700	64	5	846	94
2744	16380	56	4	1184	132

Table 6.1. Performance of PCG with and without re-orthogonalization

Figs. 6.6-6.9 present the computing time per subdomain required for the Cholesky factorization, the forward/backward substitutions and the PCG solution, as well as for one SpMV multiplication, for different number of subdomains. It can be seen that the required factorization time and forward/backward substitutions on GTX 580 is improved by one to two orders of magnitude compared to the GTX285, while the corresponding performance of i7 is in-between. Fig. 6.8 shows the performance of each component for the solution of the subdomain problems with PCG, which is following the trend of the SpMV multiplication shown in Fig. 6.9. The GTX285 has similar performance (of the same order) with i7 while the GTX580 is faster by more than one order of magnitude.

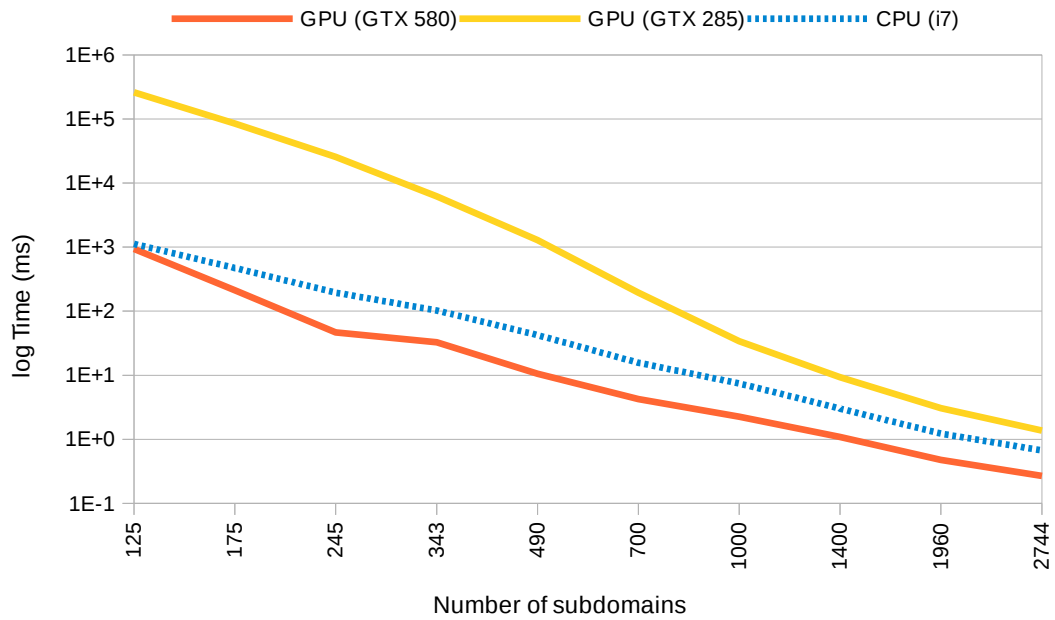


Fig. 6.6: Computing time per subdomain for Cholesky factorization

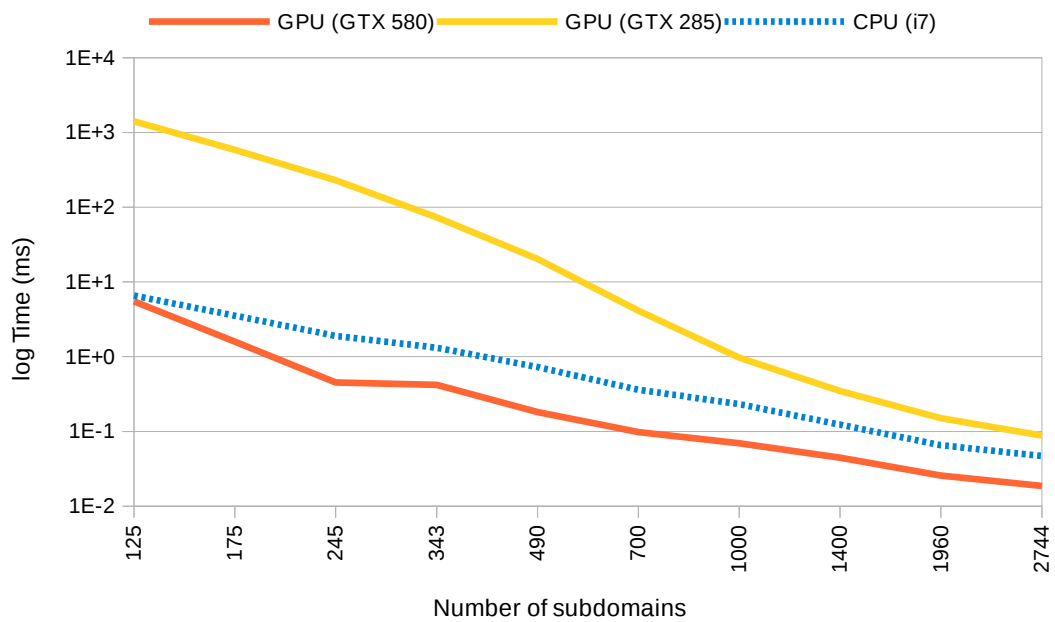


Fig. 6.7: Computing time per subdomain for forward and backward substitutions

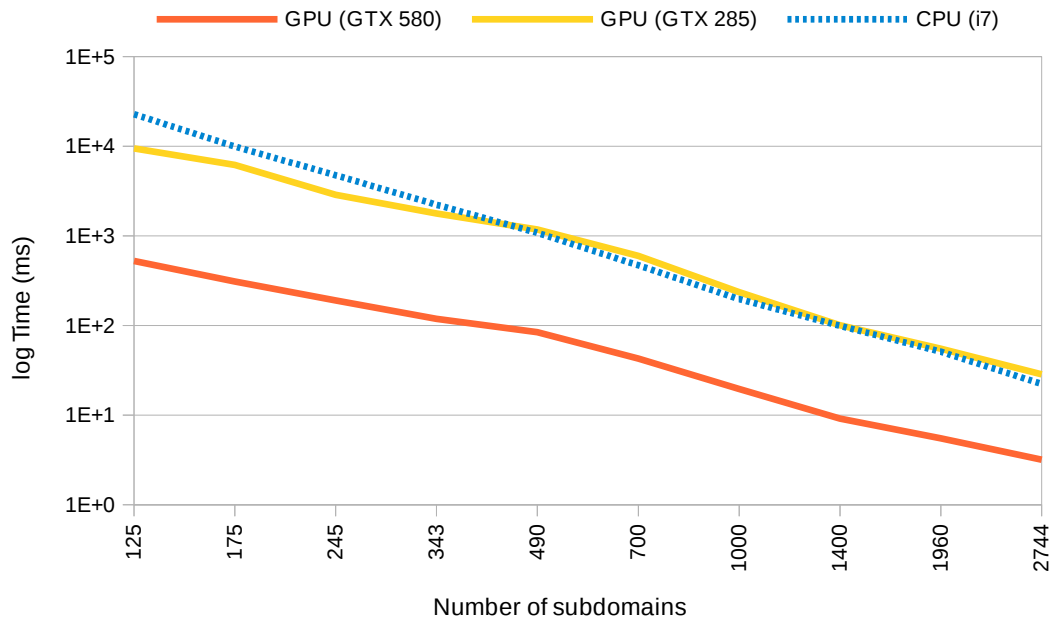


Fig. 6.8: Computing time per subdomain for PCG solution

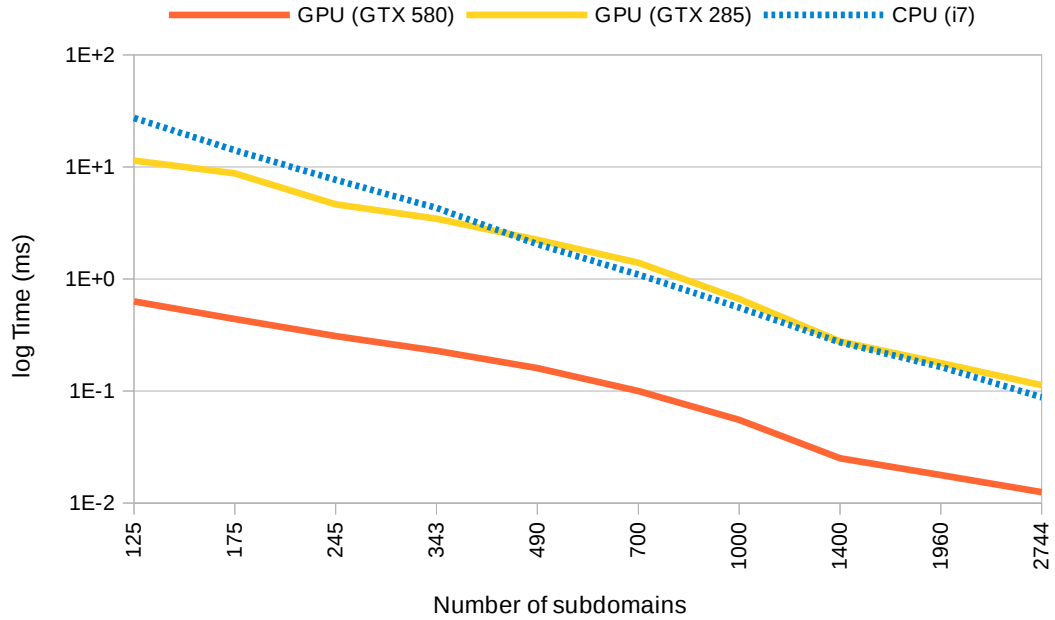


Fig. 6.9: Computing time per subdomain for sparse matrix-vector multiplication

The optimum subdomain distribution between CPU and GPU for different number of subdomains, resulting from the dynamic load balancing described in Section 6.5, for performing the Cholesky factorization and the forward and backward substitutions of the subdomain problems, is presented in Figs. 6.10 and 6.12 for the GTX 285 and the GTX 580, respectively. The percentage for the optimum subdomain distribution for GTX 285 is in the range of 25% and for GTX 580 in the range of 75%, for the most efficient decompositions. The corresponding optimum subdomain distribution for the PCG subdomain solver, as shown in Figs. 6.11 and 6.13, is 50% for the GTX 285 and 90% for the GTX 580.

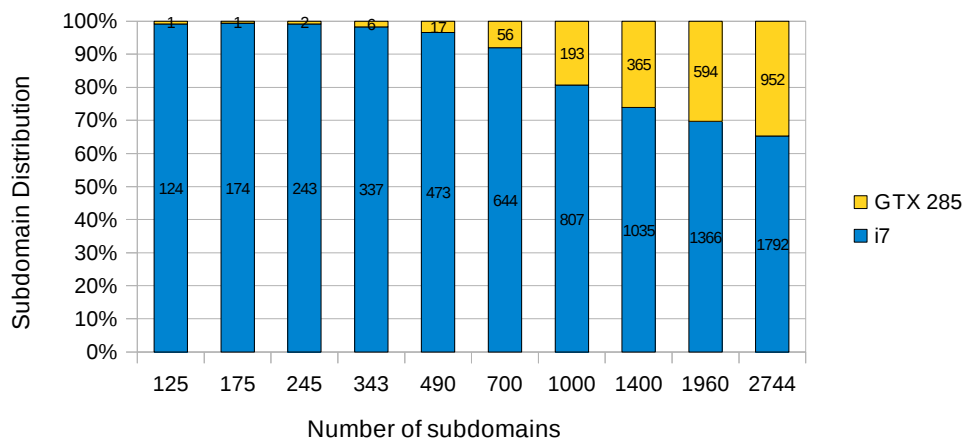


Fig. 6.10: Optimum subdomain distribution between CPU and GPU for the factorization and forward/backward substitutions of the Cholesky subdomain solver with the i7 and GTX 285 combination

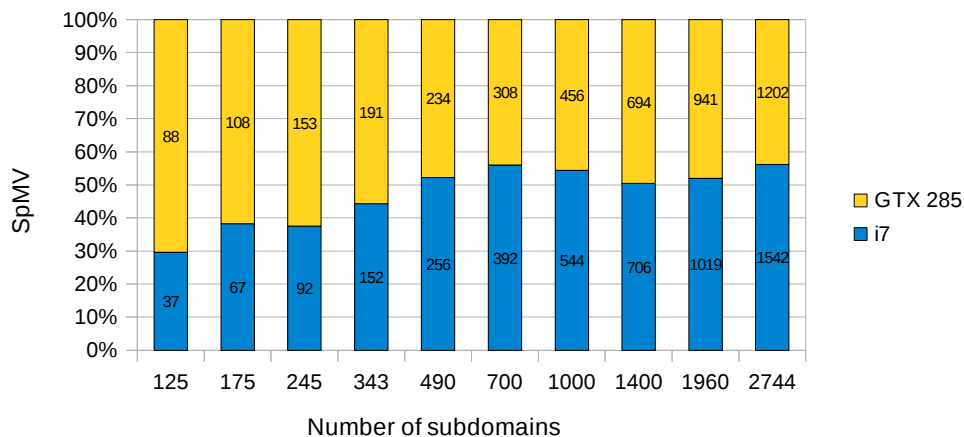


Fig. 6.11: Optimum subdomain distribution between CPU and GPU for SpMV multiplications and PCG subdomain solver with the i7 and GTX 285 combination

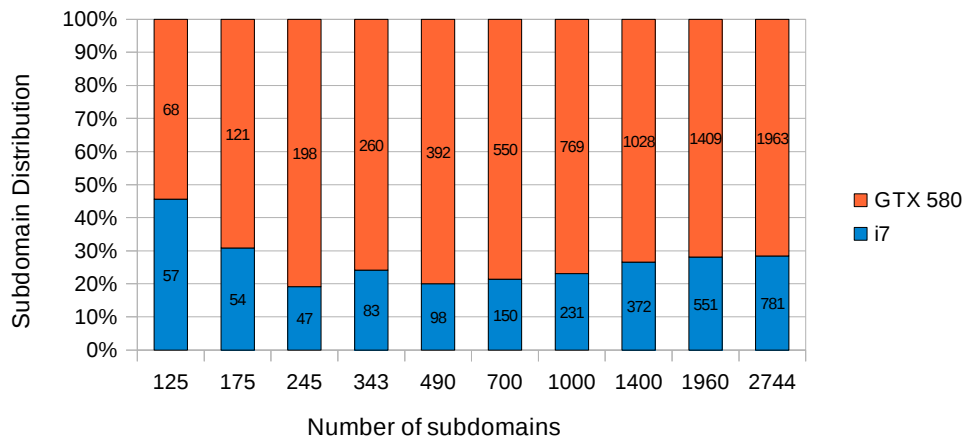


Fig. 6.12: Optimum subdomain distribution between CPU and GPU for the factorization and forward/backward substitutions of the Cholesky subdomain solver with the i7 and GTX 580 combination

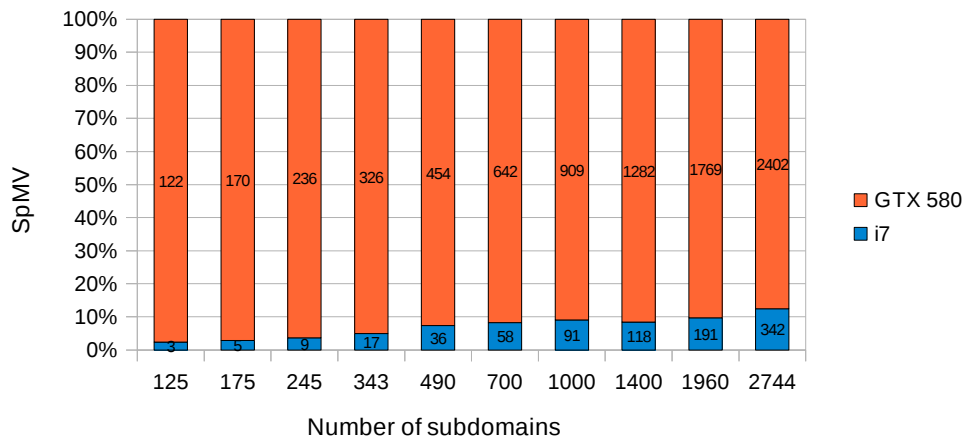


Fig. 6.13: Optimum subdomain distribution between CPU and GPU for SpMV multiplications and PCG subdomain solver with the i7 and GTX 580 combination

The performance of the two versions of FETI is demonstrated in Figs .6.14, 6.15. From this comparison, it is verified that both FETI versions demonstrate a better performance for fine-grained subdivisions in all three workstation configurations. The relative speedup ratios of i7 and GTX285 are shown in Figs. 6.16, 6.17, for the two FETI versions, while Figs. 6.18, 6.19, depict the corresponding speedup ratios of the GPU and hybrid workstation vs 1 CPU core. Similar comparisons for the workstation with GTX580 are presented in Figs. 6.20-6.25. From these figures, the following observations can be made. For the case of i7-285, the optimum performance is achieved with the finest-grained decomposition. The improvement is more pronounced for the Cholesky subdomain solver, as a result of the larger reduction in the cost of factorization for smaller subdomains, compared to the corresponding reduction in the cost of SpMV multiplications which dominate the PCG subdomain solver. This trend is followed with GTX580 albeit the difference in computing times is less pronounced with respect to the number of subdomains. The CPU-only (i7) implementation gave better results than the GPU when the GTX285 was used, but this picture is completely reversed with the GTX580. In both workstation configurations, the hybrid computing implementations achieved the solution in reduced computing times.

A more quantitative picture of the performance of the proposed implementation of FETI in hybrid CPU/GPU architectures is given by comparing the speedup ratios of the different workstation configurations considered in this work. Looking at the performance of the method in the range of the optimum number of subdomains, the speedup ratios for the two versions of FETI of hybrid (i7/GTX285) vs CPU (i7) are 1.5x and 2.0x and of hybrid (i7/GTX285) vs GPU (GTX285) are 3.8x and 2.3x, respectively. For GTX580, the corresponding speedup ratios of hybrid (i7/GTX580) vs CPU (i7) are 7x and 12x and of hybrid (i7/GTX580) vs GPU (GTX580) are 1.3x and 1.2x. These speedups become more than quadrupled when compared to 1 CPU core, due to the utilization of i7's 8 logical cores.

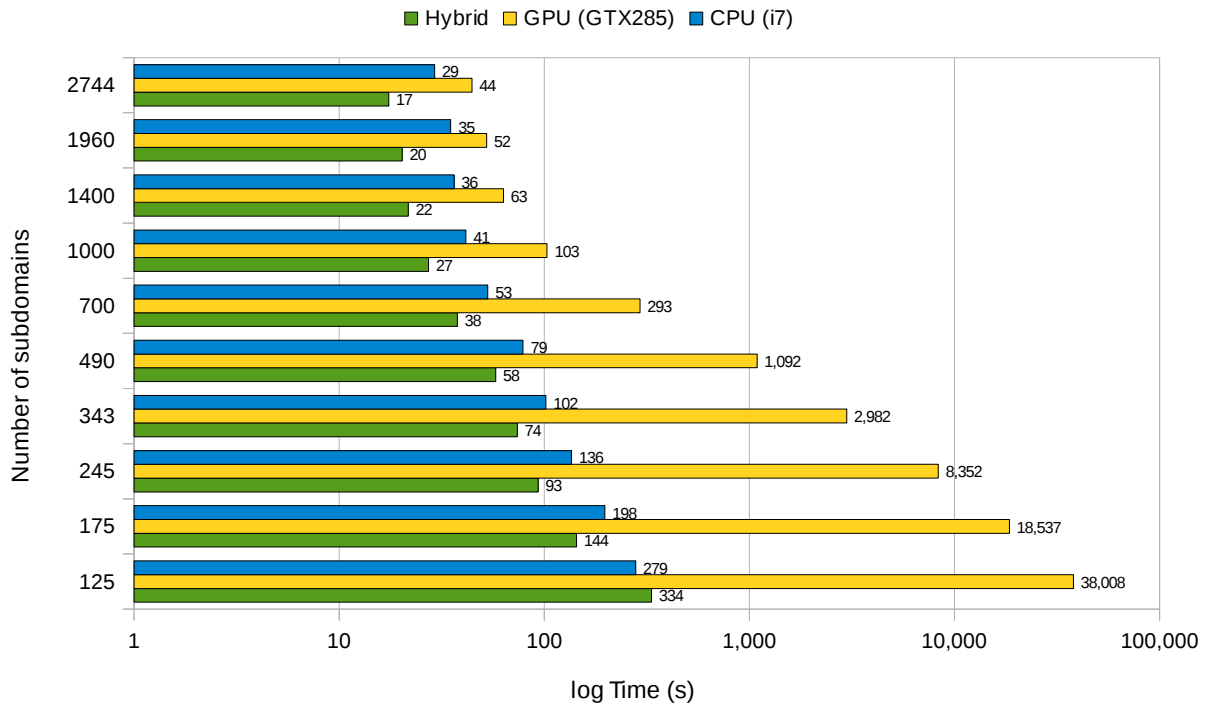


Fig. 6.14: Total solution time of FETI for the Hybrid, GPU (GTX285) only and CPU (i7) only cases with the Cholesky subdomain solver

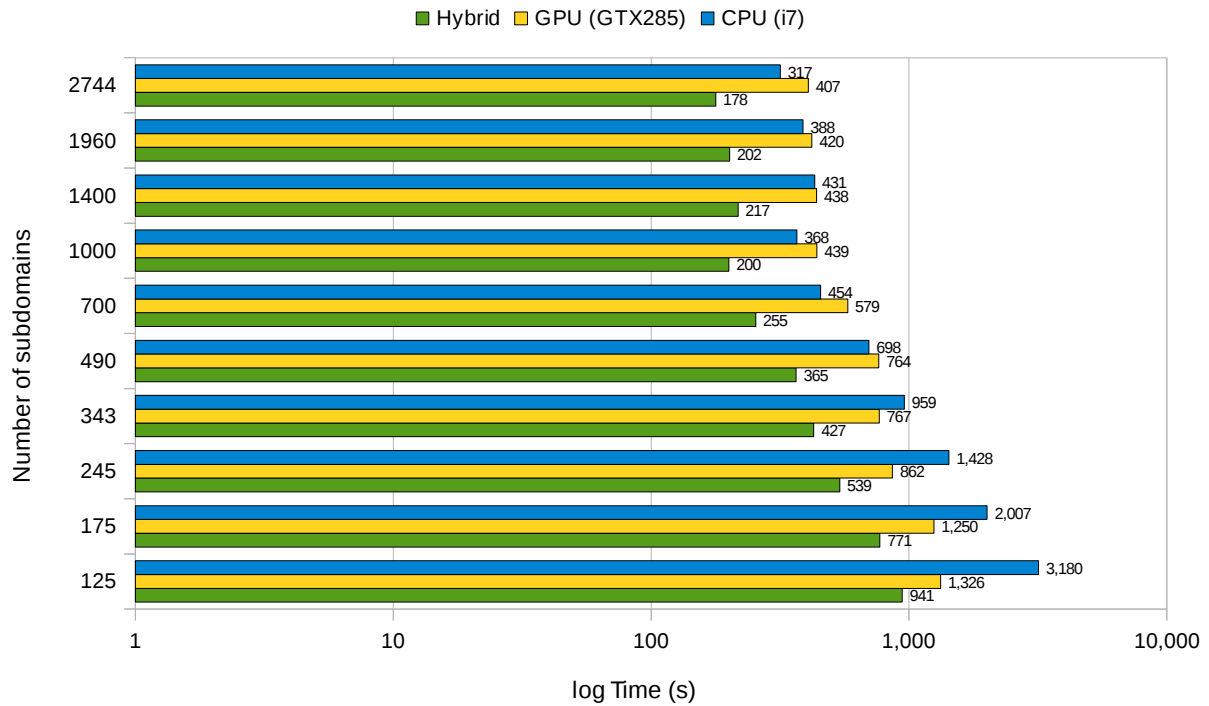


Fig. 6.15: Total solution time of FETI for the Hybrid, GPU (GTX285) only and CPU (i7) only cases with the PCG subdomain solver

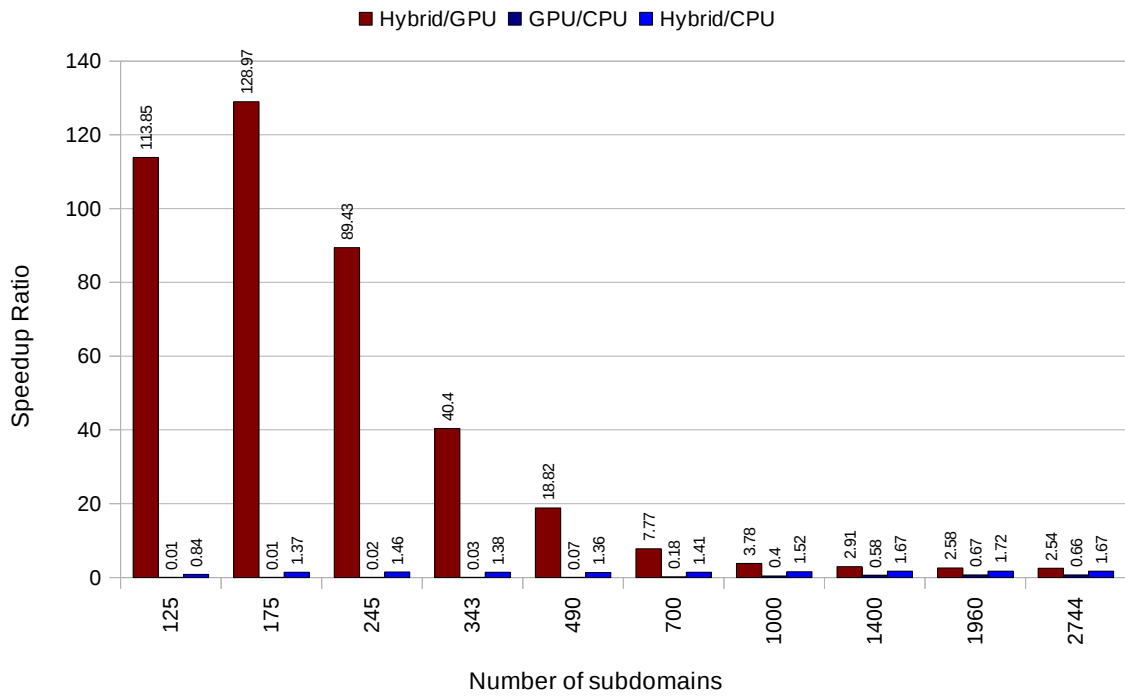


Fig. 6.16: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 285) with the Cholesky subdomain solver

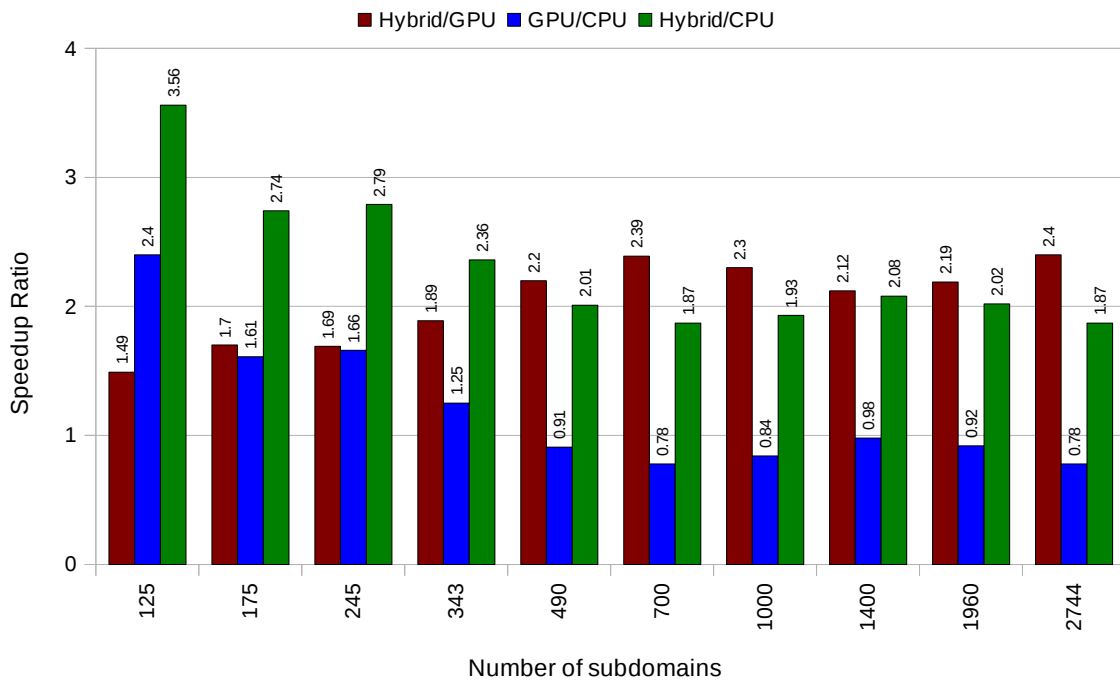


Fig. 6.17: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 285) with the PCG subdomain solver

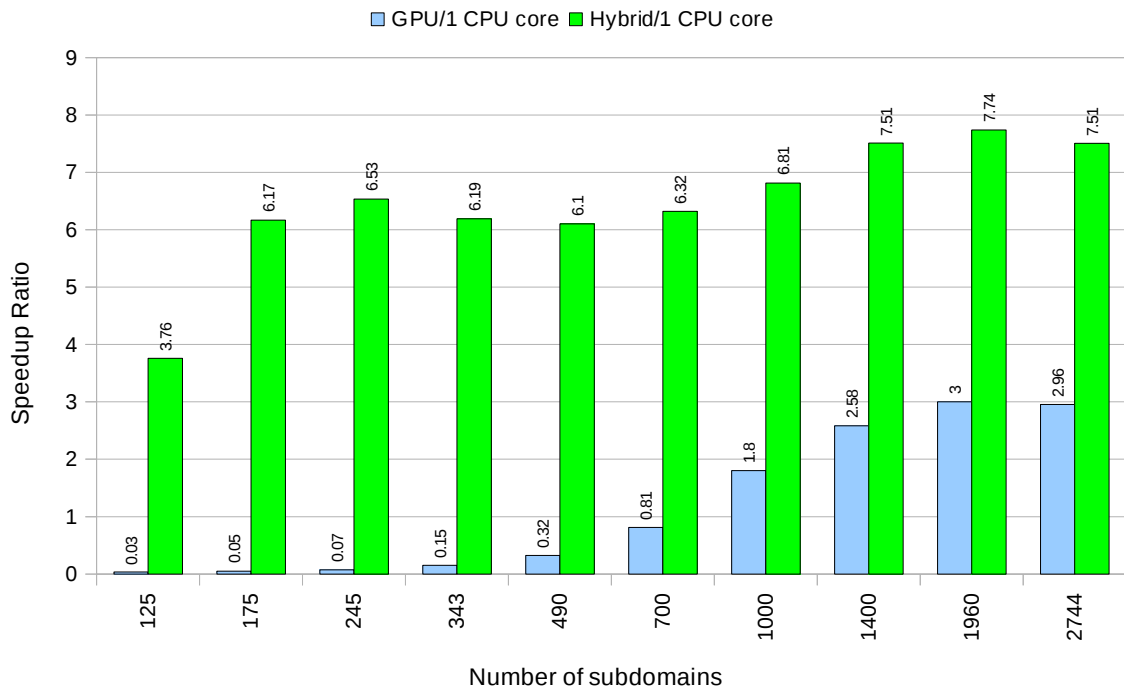


Fig. 6.18: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 285) with the Cholesky subdomain solver

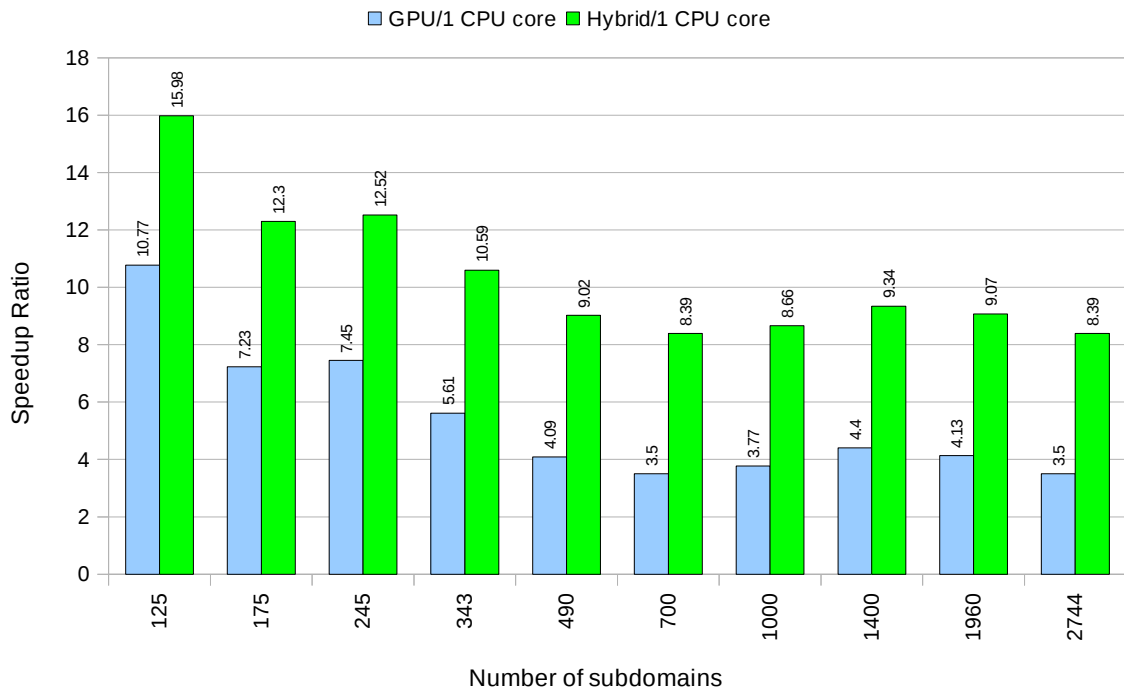


Fig. 6.19: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 285) with the PCG subdomain solver

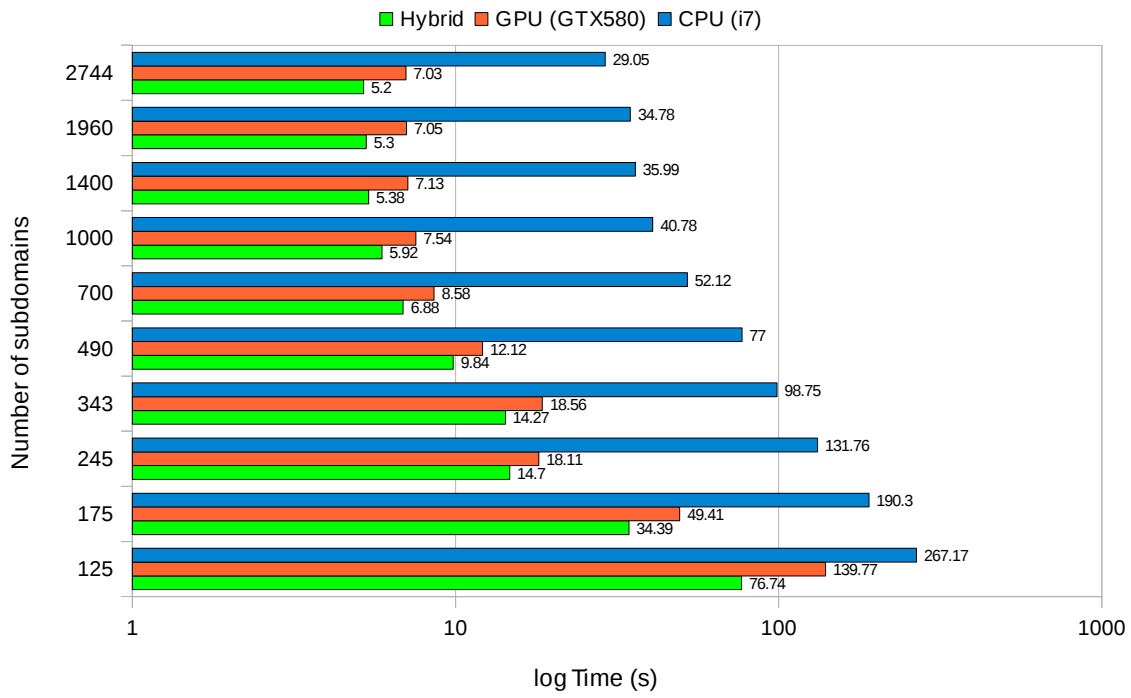


Fig. 6.20: Total solution time of FETI for the Hybrid, GPU (GTX580) only and CPU (i7) only cases with the Cholesky subdomain solver

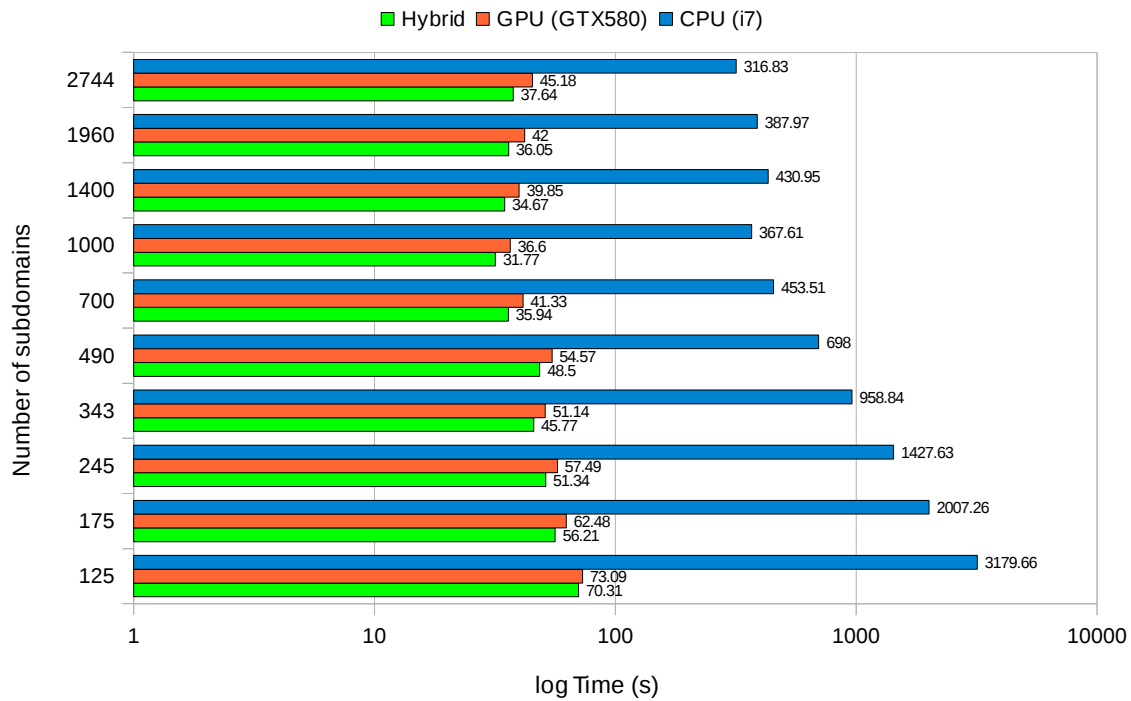


Fig. 6.21: Total solution time of FETI for the Hybrid, GPU (GTX580) only and CPU (i7) only cases with the PCG subdomain solver

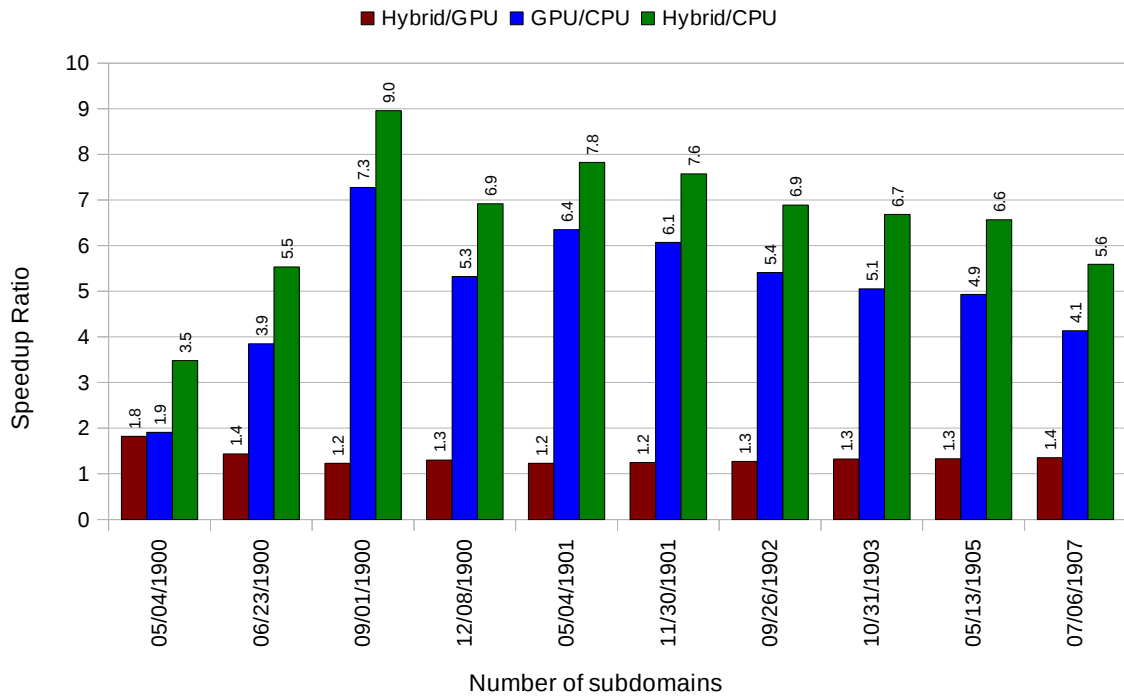


Fig. 6.22: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 580) with the Cholesky subdomain solver

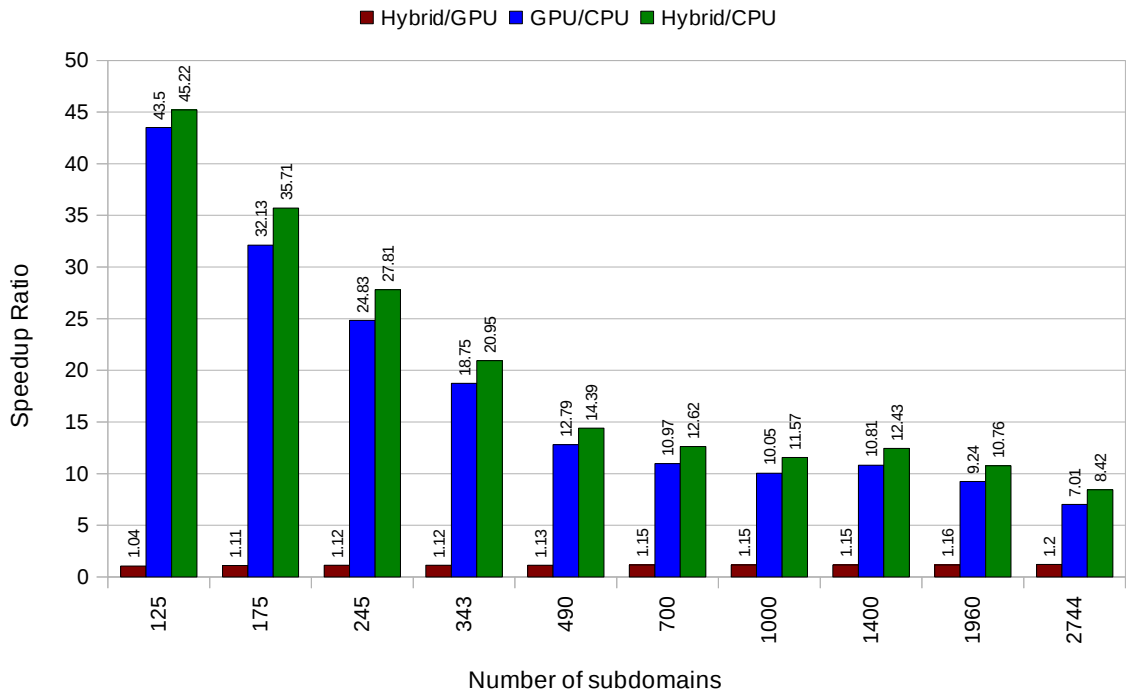


Fig. 6.23: Performance speedup ratios for different combinations of CPU (i7) and GPU (GTX 580) with the PCG subdomain solver

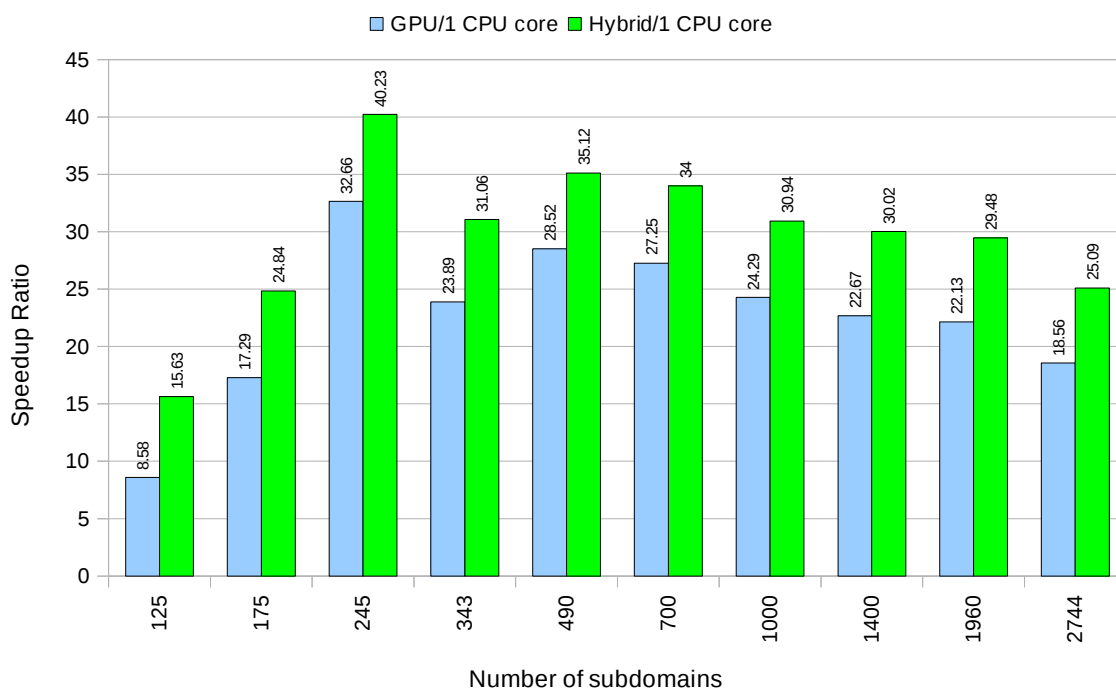


Fig. 6.24: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 580) with the Cholesky subdomain solver

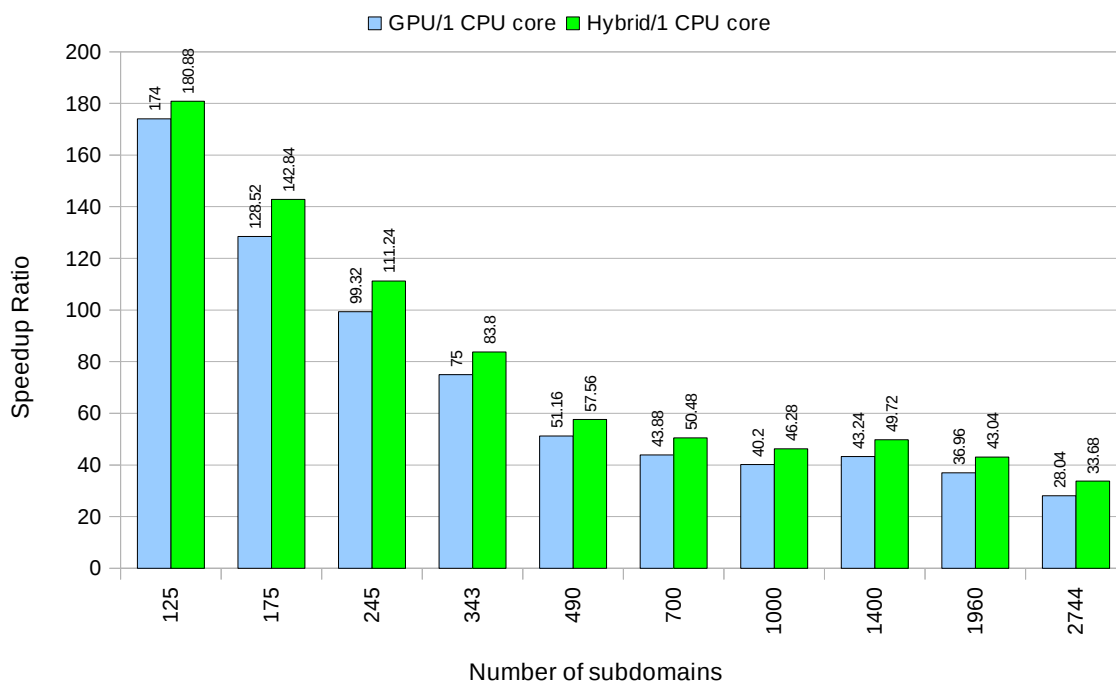


Fig. 6.25: Performance speedup ratios per CPU core for different combinations of CPU (i7) and GPU (GTX 580) with the PCG subdomain solver

6.7 Remarks

Based on the performed numerical tests in this chapter, the following remarks may be drawn. The FETI version with the direct Cholesky solver for the solution of subdomain problems performs better than the PCG subdomain solver in the examined tests and with different workstation configurations. This is attributed to the dynamic load balancing implementation of the factorization and forward backward substitution tasks. However, the performance improvement with the faster GPU is more pronounced with the PCG solver than with the Cholesky solver as a result of faster execution by GPUs of SpMV multiplications which dominate the performance of PCG.

Fine-grained subdivisions gave much better results than coarse-grained subdivisions particularly for the Cholesky solver of the subdomain problems. This is due to the high cost of the factorization for large subdomains in connection to the numerical scalability of FETI, where the convergence of the PCPG algorithm is not sensitive to the size of the interface problem. The performance of FETI with the PCG solver is less susceptible to the number of subdomains since it is dominated by the SpMV multiplications which are performed with the same efficiency by the workstation components irrespective of the size of the matrices and the corresponding vectors.

The dynamic load balancing with the task parallelism and the parallel implementation of the SpMV multiplications and dot products ensure that all components of the workstation are constantly busy with calculations resulting in full exploitation of their computing resources. This is evidenced by the high speedup ratios achieved in the test example for all hardware combinations and different number of subdomains. The dynamic load balancing allows the efficient utilization of different CPUs and GPUs as well as any number of CPU cores or GPUs, while making sure that all components are used to their full potential. Therefore, the computational handling of the finite element solution method achieves high portability for different CPU/GPU architectures while the source code exhibits maintainability, extensibility and ease of debugging, since all internal work associated with the parallel implementation is encapsulated to the CPU and GPU threads.

In conclusion, the parametric tests performed in the framework of this study showed the tremendous potential of the proposed realization of the dual domain decomposition FETI method in hybrid CPU/GPU computing platforms. The presented implementation ensures high hardware utilization and minimizes idle time which is a major issue for the efficient exploitation of the available computing power of hybrid CPU/GPU high performance computing architectures. The load balancing scheme can optimally exploit the capabilities of the available heterogeneous hardware

equipment and in conjunction with the parallel implementation of the solution algorithms of the subdomains as well as of the interface problem, can accomplish high speedup factors in the range of 50x for just one-CPU/one-GPU configuration.

7 Relations between basic entities of Gauss quadrature

In an abstract examination of the Gauss quadrature assembly, there are two types of entities: nodal entities (N) and Gauss entities (G). In FEA/MMs the nodal entities are the nodes and in IGA the nodal entities are the control points, which constitute the main parameters of the simulation method. The Gauss entities are individual Gauss points or groups of Gauss points which contribute to the calculation of the quantities associated with the nodal entities (e.g. stiffness coefficients). In FEA/IGA the Gauss entities are elements, which group several Gauss points, whereas in MMs the Gauss entities are individual Gauss points. The abstraction is helpful because each method may have different entities (for example FEA: nodes + elements, IGA: control points + elements, MMs: nodes/particles + Gauss points).

Relations between the basic entities are established for the creation of the characteristic matrices as well as for other parts of the analysis. Relations can be stored as maps or, more accurately, multimaps, i.e. associative containers in which each key is associated with multiple values. The underlying implementation of the multimaps may vary in space and time complexity allowing for flexibility in choosing the right implementation for a given scenario. These relation multimaps allow for a straightforward implementation of the assembly methods, but their identification may be non-trivial. Each of the multimaps may be stored in appropriate data structures for subsequent use or (re)calculated temporarily when required.

For MMs the definition of entity relations (nodes-Gauss points) is more involved than in FEA and IGA. Sections 7.5.2.2 and 7.5.2.3 are dedicated to the presentation of strategies towards efficient entity relations in MMs.

7.1 N-G correlations

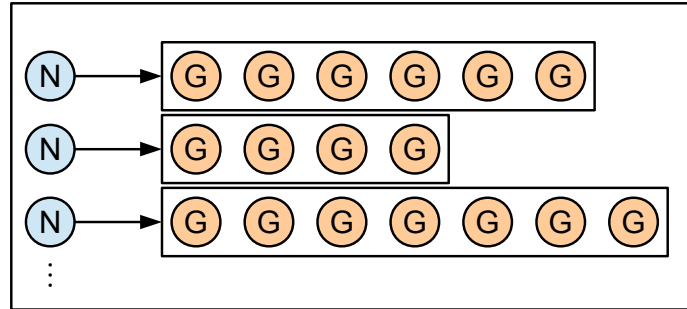


Fig. 7.1: N-G correlations: each nodal entity N is associated to all Gauss entities G influencing it.

This multimap associates each nodal entity (N) to all Gauss entities (G) that influence it, as can be seen in Fig. 7.1. Examples of N-G correlations for EFG and IGA are shown in Figs. 7.10, 7.21, respectively. An exhaustive search that tests all nodal-Gauss entity combinations is not practical and therefore simulation-specific methods need to be used to derive this information. For example in MMs a geometric search has to be performed whereas in IGA, which exhibits a structured control point grid, this information can be easily derived from the grid.

7.2 G-N correlations

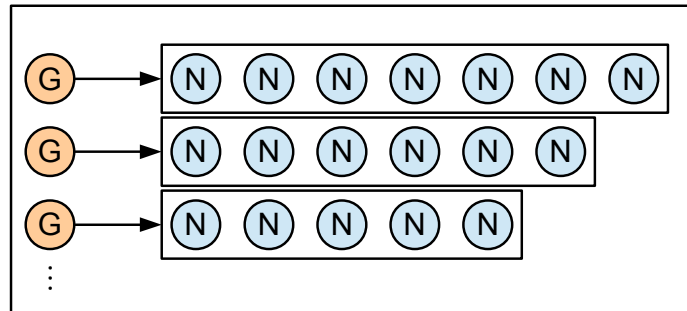


Fig. 7.2: G-N correlations: each Gauss entity G is associated with all nodal entities N it influences.

This multimap associates each Gauss entity (G) with all nodal entities it influences, as can be seen in Fig. 7.2. Examples of what the G-N correlations are in EFG and IGA are shown in Figs. 7.9, 7.22, respectively. This is the reverse of the N-G correlations and is directly used in the contribution-wise methods which iterate through Gauss points/elements. It is also used in the first phase of the interaction-wise methods for the calculation of the quadrature values, e.g. shape function derivatives. The G-N correlations can be derived with simulation-specific methods similarly to the N-G correlations. For example in MMs methods, a geometric search has to be

performed, while in FEA the G-N correlations of a Gauss point are simply the nodes of the enclosing element. In IGA the G-N correlations can be easily derived from the structured grid.

G-N correlations can also be derived from the N-G correlations if the latter are already calculated and stored. The reversing process can be performed as follows: (i) Initialize a collection s_j for each Gauss entity G_j of the domain. (ii) Take the correlations of a nodal entity N_i from the N-G correlations. (iii) Iterate through all influencing Gauss entities G_j of N_i and add N_i to the appropriate collection (s_j) if not already present.

Calculating the N-G correlations from the G-N correlations is also possible, which implies that calculating either of the correlations can also yield the other one. Reversing either of the N-G or G-N correlations to obtain the other is useful if the cost of searching for correlations is high. However, correlation searches are embarrassingly parallel whereas the reversing process as outlined above involves scatter parallelism.

7.3 Interactions

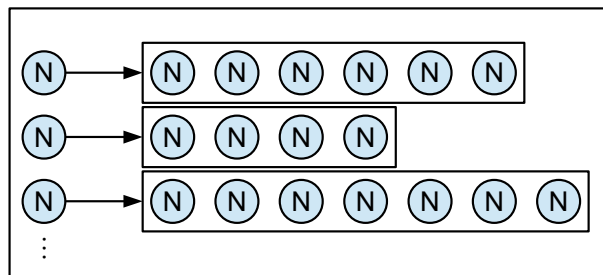


Fig. 7.3: Nodal entity interactions.

The interactions of a nodal entity (N) comprise all other nodal entities that interact with it as defined in the corresponding simulation method (e.g. having at least one shared Gauss point or element). The interactions used by the contribution-wise methods to predict the non-zero pattern for certain matrix formats (like the skyline matrix). It is also the main ingredient for the interaction-wise methods, as they iterate through interacting pairs.

There are several generic ways to identify the interactions. The brute-force method checks all possible pairs of nodal entities and keeps those that are interacting. This is $O(n^2)$ to the number of nodal entities and thus it is only applicable for small problems. A better way is to utilize the N-G and G-N correlations: Each nodal entity N_A has a list of Gauss entities influencing it and each

Gauss point has a list of nodal entities it influences. Therefore, to identify the interactions of a nodal entity, the nodal entities associated with the related Gauss entities of N_A are inspected. This is exactly the same as modeling the nodal and Gauss entities in a graph and running breadth-first search (BFS) [90]. More specifically, the correlations can be modeled in a bipartite graph where there are vertices for nodal and Gauss entities. Edges connect correlated entities: nodal entities are connected to the Gauss entities that influence them and Gauss entities are connected to the nodal entities they influence. In that case, running BFS from a nodal entity N_A yields all influencing Gauss entities of N_A in the first layer of BFS and all interacting nodal entities of N_A in the second layer of BFS.

Fig. 7.4 shows node N_A which is influenced by Gauss points G_i, G_j, G_k and each of these Gauss points influences various nodes, including node N_A itself. Those nodes are guaranteed to interact with N_A since there is at least one Gauss point in common between them. Thus, since N_A interacts with N_A, N_B, N_C due to G_i , N_A, N_B, N_D due to G_j and N_A, N_B, N_E due to G_k , the interactions of N_A are N_A, N_B, N_D, N_E .

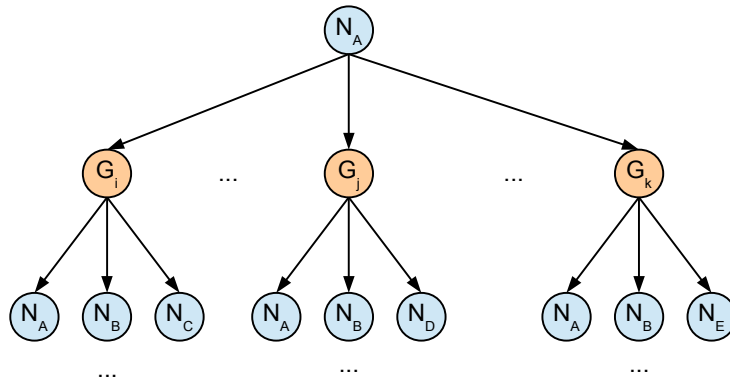


Fig. 7.4: Identifying interacting pairs for nodal entity N_A .

This technique is parallel-friendly and works without assuming any specific characteristics about the basic entities involved. It can be further improved by checking only the Gauss points/elements that are near the border of the domain of influence of the nodal entity, as the Gauss entities closer to the nodal entity will provide duplicates.

Apart from the aforementioned generic approaches, the interactions can also be identified with approaches that take advantage of the actual simulation method characteristics (e.g. structured grid, geometry etc), leading to potentially faster implementations. In IGA, the generic technique takes very little time compared to the total matrix formulation time so it can be applied in lieu of a faster

method based on the structured grid. On the other hand, in EFG, it is worthwhile to apply a method-specific approach.

7.4 Synergies

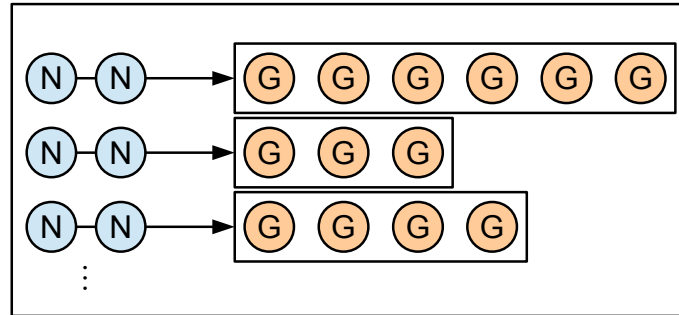


Fig. 7.5: Interacting pairs with shared Gauss entities.

The second vital ingredient of the interaction-wise method is to identify the Gauss points or elements shared by an interacting pair. These Gauss points are used for the calculation of the matrix coefficients corresponding to that particular interacting pair. A generic, parallel-friendly technique which works without assuming any specific characteristics about the basic entities involved is to use the N-G correlations to facilitate the identification of synergies: For each pair $i-j$, find the Gauss entities that are contained in the N-G correlations of both i and j . The Gauss entities of the N-G correlations can be stored in data structures that can quickly find if an item is contained or not (e.g. hash tables). Alternatively, simulation specific characteristics can be used, as in the case of EFG where the influencing Gauss entities of i can be geometrically checked to determine the subset of them that are also within the domain of influence of j . In IGA, the generic technique takes little time compared to the total matrix formulation time but the synergies can also be identified through the structured grid.

7.5 Domain of influence in the simulation methods

The three methods used in this work (FEA, MMs, IGA) greatly differ in the range of the domain of influence as well as the characteristics of the influences between their basic entities (nodes, particles, control points, Gauss points, elements). The domains of influence determine the correlations between nodes/control points and Gauss points/elements. They also determine the node-node or control point-control point interactions. The most general definition for interacting pairs in Gauss quadrature based methods is: two nodal entities are interacting, and thus have non-zero entries in the corresponding matrix entries, if and only if there is at least 1 Gauss point influencing both nodal entities. Easier definitions can apply depending on the simulation methods.

7.5.1 Domain of influence in FEA

The basic entities in FEA are nodes and elements. The elements influencing a node are only the ones that are adjacent to the node. Fig. 7.6 shows different cases of influencing elements depending on the position of a node. Node C is a corner node and is influenced by 4 elements, node S is a side node and is influenced 2 elements while node I is internal and is influenced solely by the element it is enclosed in. In 3D the situation is similar, with a maximum of 8 elements influencing a corner node and 1 element influencing an internal node. Fig. 7.7 shows the area influenced by a single Gauss point. This is true for all Gauss points of that element, so in FEA, each element only influences the nodes it is directly adjacent to or that are directly within it.

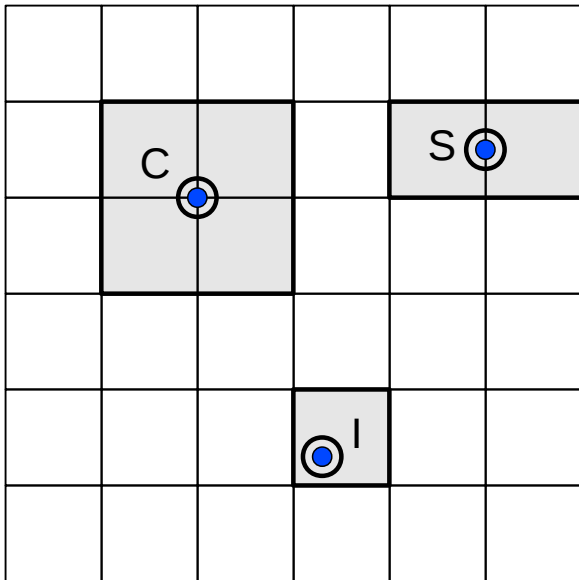



Fig. 7.6: FEA: domain of influence of node 

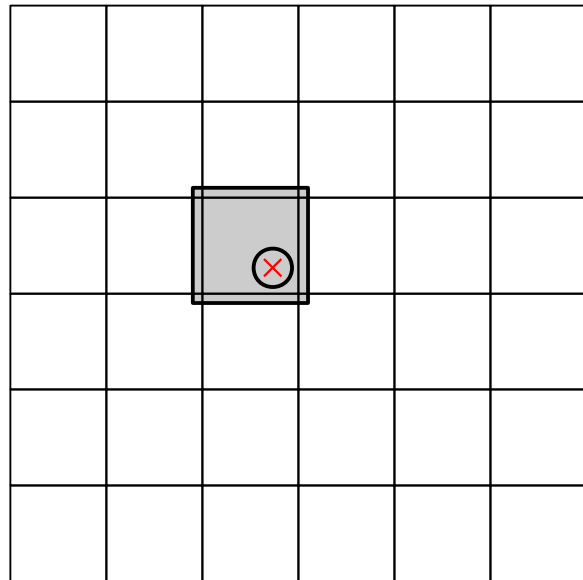



Fig. 7.7: FEA: domain of infl. of Gauss p. 

The nodes that are interacting are shown in Fig. 7.8. Each node interacts with all nodes in its adjacent elements because the Gauss points of those elements are guaranteed to be shared by both nodes. Corner elements have the most interactions with other nodes, while internal nodes have the least interactions.

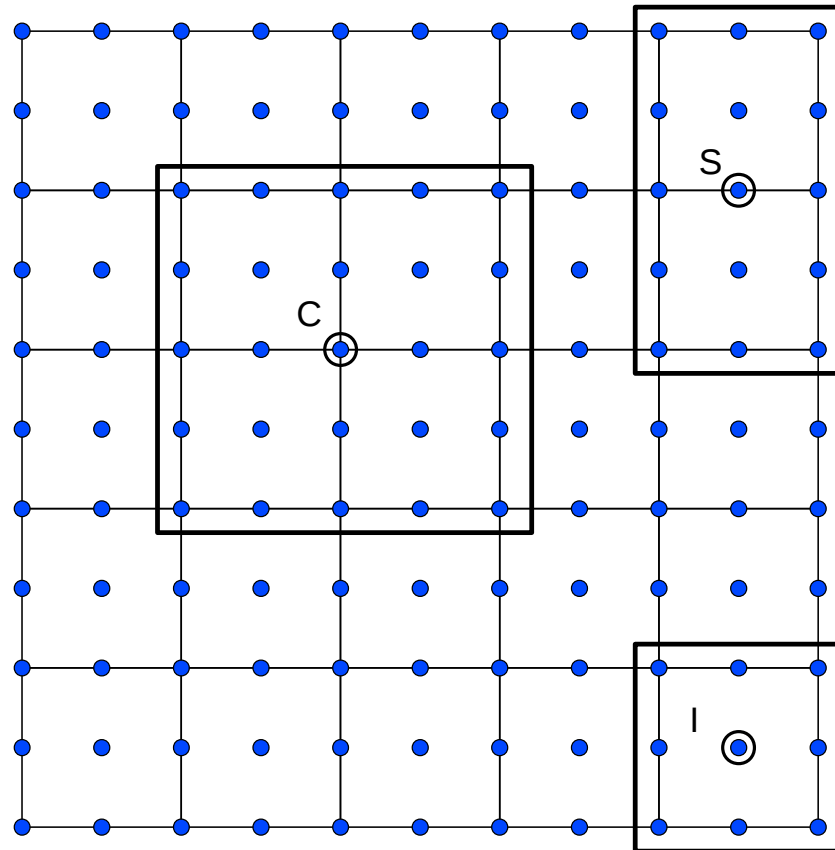


Fig. 7.8: FEA: interacting nodes for $p = 2$

Compared to the other two simulation methods, the domains of influence in FEA are limited: there is a constant number (max 8) of elements affecting each node, regardless of the order p of the elements and each Gauss point/element only influences the nodes adjacent to it or directly within it. Another characteristic is that the shape functions in FEA are predefined for each element type whereas shape functions in MMs/IGA are more “dynamic”. As a result, the cost for the matrix formulation in FEA is significantly less than the cost of MMs and IGA for similar analysis parameters. For finite elements of typical order, the solution time dominates the formulation time.

More information on FEA in the context of the domain of influence and matrix formulation cost is found in the comparisons made with the other methods in the next two sections.

7.5.2 Domain of influence in MMs

Due to the absence of elements, the basic entities in MMs are nodes/particles and Gauss points. The domains of influence of Gauss points are much larger than the corresponding domains in FEA as is schematically shown in Fig. 7.9 for a domain discretized with MMs and FEA. Note that the comparison is for equal number of nodes and Gauss points; the two methods do not exhibit the same accuracy in this case. In the figures, the radius of the domain of influence is assumed 2,5 times the distance between two consecutive nodes.

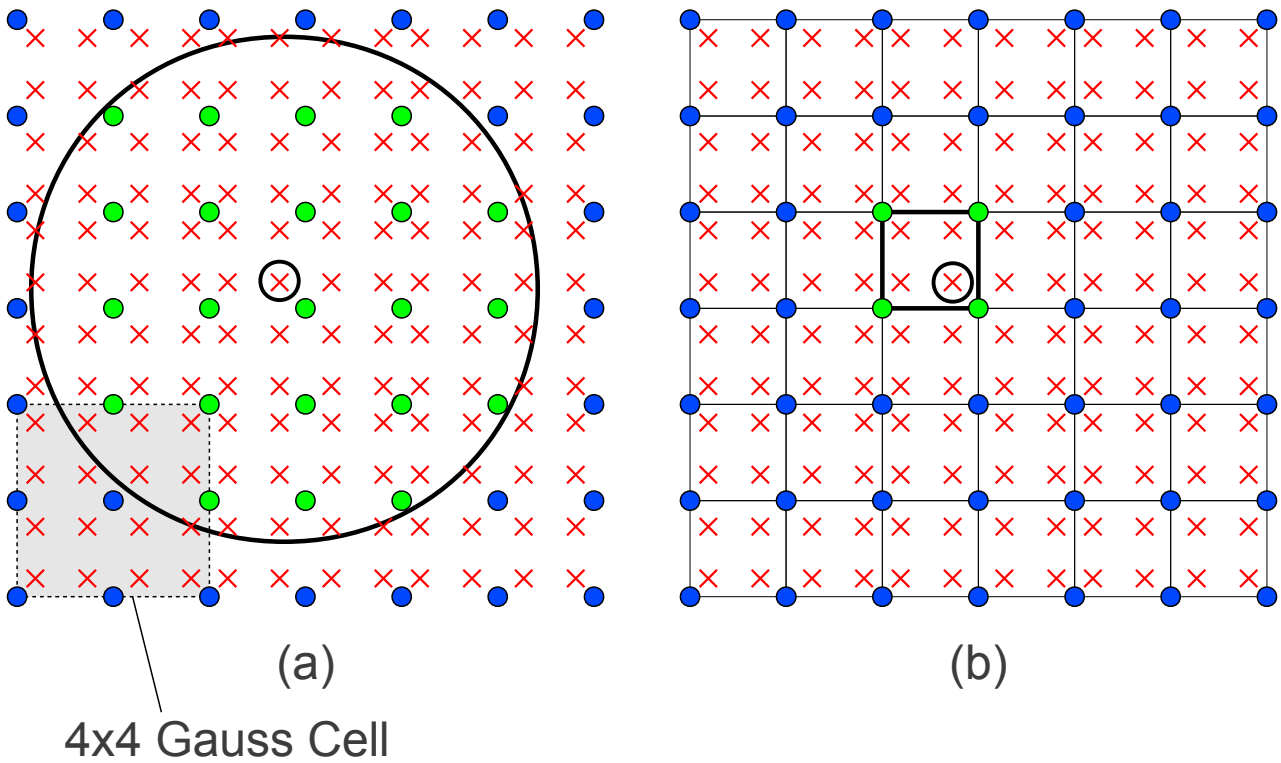


Fig. 7.9: Domain of influence of Gauss point \otimes in (a) MMs; (b) FEA, for the same number of nodes \bullet and Gauss points \times

In FEA each Gauss point is involved in element-level computations for the formation of the element stiffness matrix which is then added to the appropriate positions of the global stiffness matrix. Moreover, the shape functions and their derivatives are predefined for each element type and need to be evaluated on all combinations of nodes and Gauss points within each element. In MMs methods, however, the contribution of each Gauss point is directly added to the global stiffness matrix while the shape functions are not predefined and span across larger domains with a significantly higher amount of Gauss point-node interactions.

Although, in MMs methods there is no need to construct a mesh, the correlation between nodes and Gauss points needs to be defined. This preliminary step required before building the stiffness matrix is implicitly performed with the mesh creation in FEA but must be explicitly done in MMs and can be time-consuming if not appropriately handled. For the aforementioned reasons, computing the stiffness matrix in MMs is a computationally demanding task which needs special attention in order to be affordable in real-world applications.

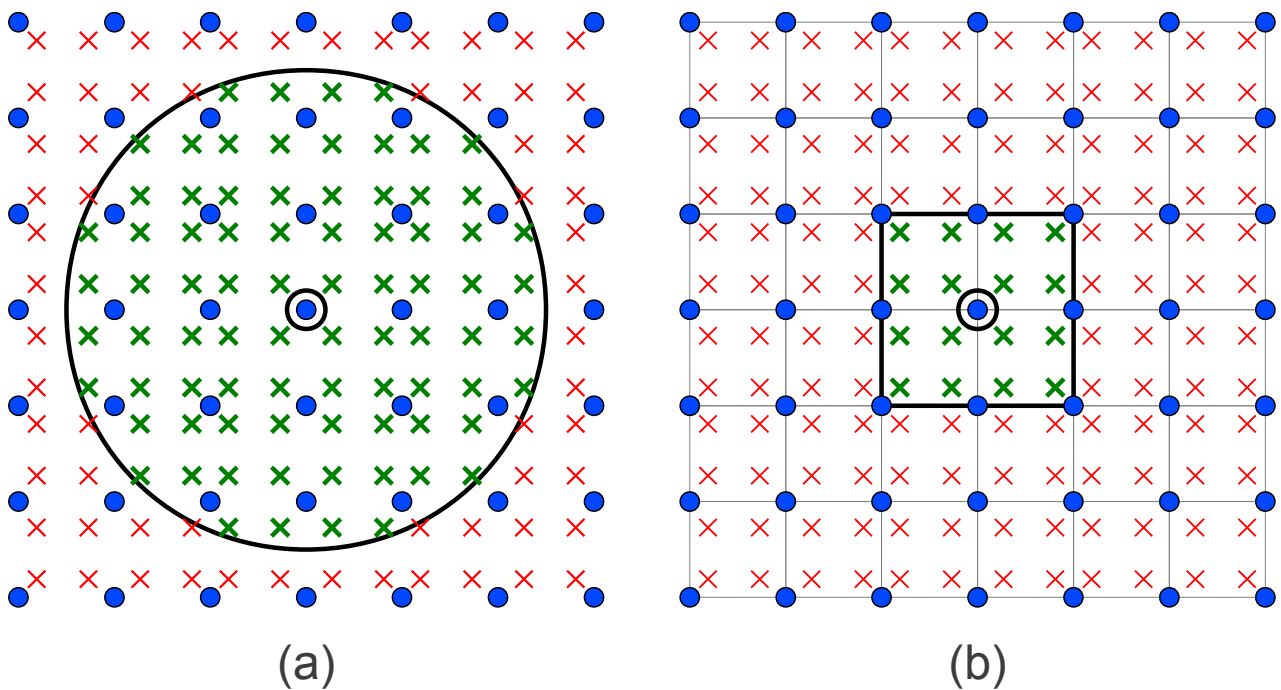


Fig. 7.10: Domain of influence of node \odot (a) MMs; (b) FEA, for the same number of nodes \bullet and Gauss points \times

Fig. 7.11 shows interacting nodes A, B as well as node C, which does not interact with neither A or B. Due to the nature of MMs, the general definition of interactions is used, i.e: two nodes are interacting if there is at least 1 Gauss point shared by both nodes.

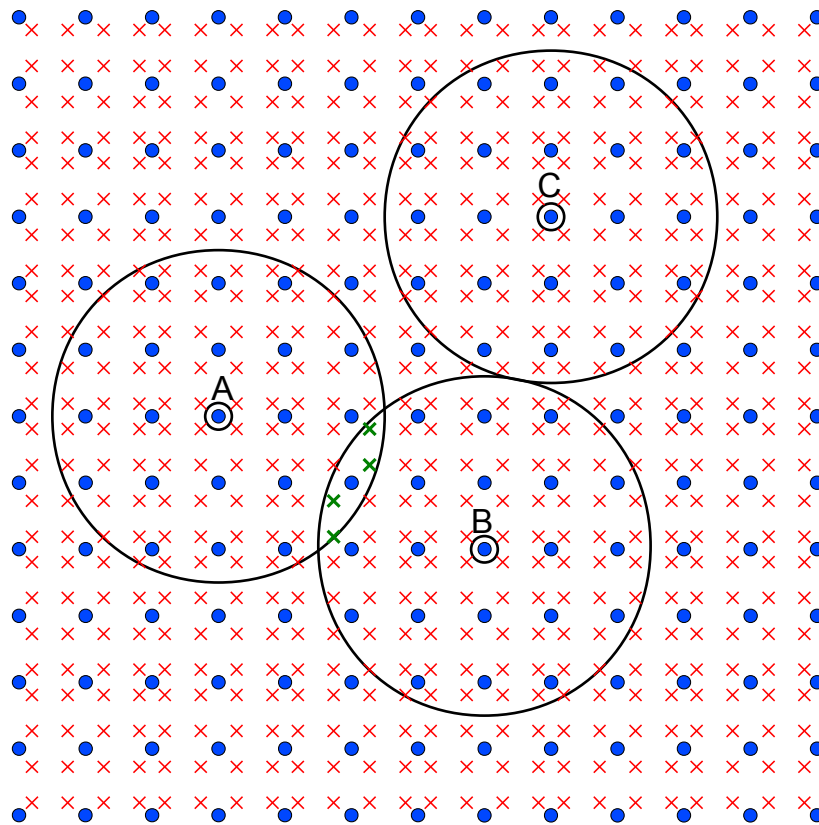


Fig. 7.11: MMs: node interactions (nodes:●, Gauss points ×)

7.5.2.1 Comparison with FEA

Table 7.1 shows the number of Gauss points influencing a single node and the number of nodes influenced by a single Gauss point in typical 2D and 3D problems. The elements representing FEA are Quad4 and Hexa8 which comprise corner nodes only.

	2D		3D	
	EFG (doi=2.5)	FEA (QUAD4)	EFG (doi=2.5)	FEA (HEXA8)
Gauss points influencing a node	100	16	1000	64
Nodes influenced by a Gauss point	25	4	125	8

Table 7.1: Influences per node and Gauss point for EFG and FEA

Table 7.2 shows the total number of correlations for the six examples considered (see Table 2.2). The significantly higher number in EFG methods is a direct consequence of the larger domain of influence, as shown in Figs. 7.9 And 7.10.

Example	Nodes	Gauss points	Total Correlations		Ratio
			EFG	FEA	
2D-1	25,921	102,400	2,534,464	409,600	6.2
2D-2	75,625	300,304	7,463,824	1,201,216	6.2
2D-3	126,025	501,264	12,475,024	2,005,056	6.2
3D-1	9,221	64,000	7,077,888	512,000	13.8
3D-2	19,683	140,608	16,003,008	1,124,864	14.2
3D-3	35,937	262,144	30,371,328	2,097,152	14.5

Table 7.2: Total number of node-Gauss point correlations in EFG and FEA

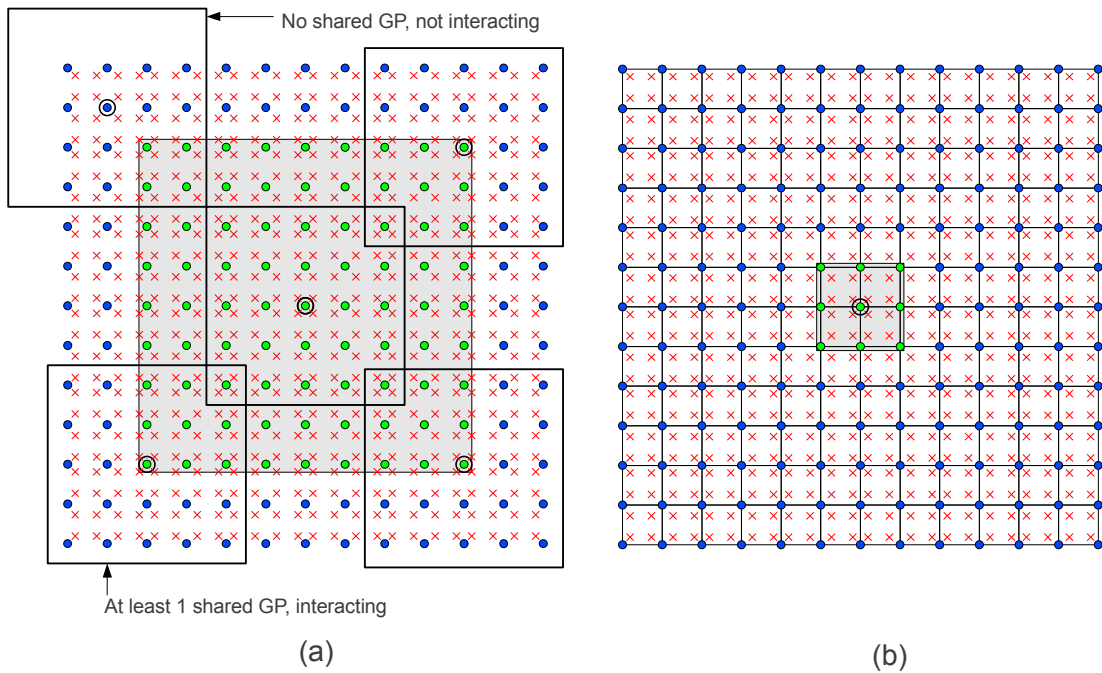


Fig. 7.12: Interacting nodes: (a) MMs; (b) FEA for the same number of nodes (●) and Gauss points (×)

Fig. 7.12 compares MMs and FEA for equal number of nodes and Gauss points. Table 7.3 shows the number of interacting node pairs in MMs and FEA for equal number of nodes and Gauss points. Interactions in MMs extend in much larger regions than in FEA, as is shown in Fig. 7.12. Furthermore, the numbers are indicative of the total non-zeros of the corresponding stiffness matrices. The total non-zeros can be calculated by

$$NZ = 4 \cdot NP - n \quad (2D) \qquad \qquad \qquad NZ = 9 \cdot NP - 3 \cdot n \quad (3D) \qquad (7.1)$$

where NP is the number of interacting node pairs and n is the number of nodes.

Example	Nodes	Interacting node pairs		Ratio
		EFG	FEA	
2D-1	25,921	1,033,981	128,641	8.0
2D-2	75,625	3,051,325	376,477	8.1
2D-3	126,025	5,103,325	627,997	8.1
3D-1	9,221	2,418,035	118,121	20.5
3D-2	19,683	5,554,625	256,361	21.7
3D-3	35,937	10,644,935	474,305	22.4

Table 7.3: Number of interacting node pairs in EFG and FEA

Each interacting node pair corresponds to a non-zero submatrix of the stiffness matrix, whose size is equal to the number of dof of each node. To calculate the corresponding coefficients, contributions from several Gauss points are summed to form the final submatrix. The total number of synergies is shown in Table 7.4.

Example	Gauss points	Total Synergies		Ratio
		EFG	FEA	
2D-1	102,400	32,725,544	1,024,000	32.0
2D-2	300,304	96,647,624	3,003,040	32.2
2D-3	501,264	161,681,224	5,012,640	32.3
3D-1	64,000	408,317,728	2,304,000	177.2
3D-2	140,608	942,981,088	5,061,888	186.3
3D-3	262,144	1,813,006,048	9,437,184	192.1

Table 7.4: Total synergies for EFG and FEA.

From the above tables it is clear that the computational effort required for MMs methods is much higher than in FEA. Large domains of influence lead to large number of node interactions which increases the bandwidth of the matrix. Furthermore, large domains of influence imply that the cost to calculate each of them is relatively higher compared to FEA. In sum, MMs pose the following additional challenges compared to FEA concerning performance:

- Definition of correlations, interactions, etc
- Increased cost of non-zero entries
- Increased matrix bandwidth

7.5.2.2 Identification of correlations in MMs

With the absence of an element mesh, the node-Gauss point and Gauss point-node correlations (Sections 7.1, 7.2 respectively) must be established explicitly in MM simulations through geometric searches. In the ensuing analysis, the focus is on finding the influencing Gauss points of a node. The same techniques apply for finding the influenced nodes of a Gauss point by swapping the role of nodes and Gauss points.

The so-called naive approach is to globally search for the Gauss points belonging to the domain of influence of each node (Fig. 7.13). This approach performs a large amount of unnecessary calculations since the domains of influence are localized areas. All combinations of nodes and Gauss points must be tested so the algorithmic complexity for this kind of approach would be $O(nn_G)$, where n is the number of nodes and n_G is the number of Gauss points. Since the number of Gauss points is typically much higher than the number of nodes, this is at least $O(n^2)$ and a quadratic running time is poor. In order to reduce the time spent for identifying the correlations, Gauss points far away from the nodes under inspection must be quickly excluded.

Regioning is a technique that can be used to localize geometric searches. A rectangular grid is created and we refer to each of the regions defined as a Gauss region. The grid is always rectangular, and usually with square regions, regardless of the domain. Each Gauss region contains a group of Gauss points. Given the coordinates of a particular node, it is immediately known in which region it is located (by dividing the coordinate of an axis with the region size in the same axis). The search per node is conducted over the neighboring Gauss regions only instead of the global domain (Fig. 7.14). The neighboring Gauss regions are the $c \times c$ regions surrounding the node, where c is determined by the domain of influence. Regardless of the size of the problem, the search per node is restricted to a small number of Gauss regions (c^2). Since each node needs to check a constant number of Gauss points regardless of the size of the problem, the time complexity is $O(n)$. Note that Gauss regions can be formed from a single Gauss cell or a cluster of Gauss cells or can be totally unrelated to Gauss cells (Fig. 7.15).

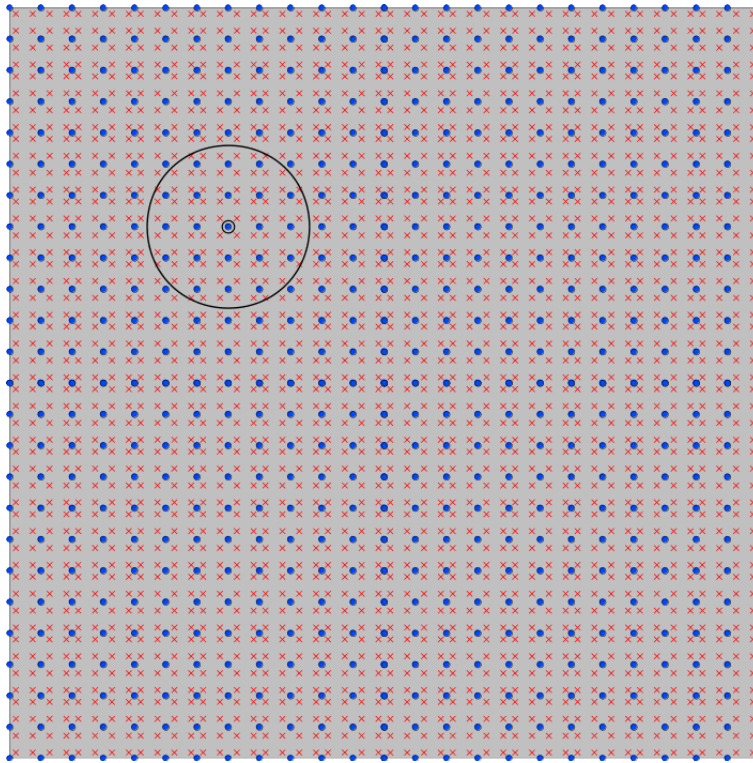


Fig. 7.13: Global search for correlations
(nodes: ●, Gauss points ×)

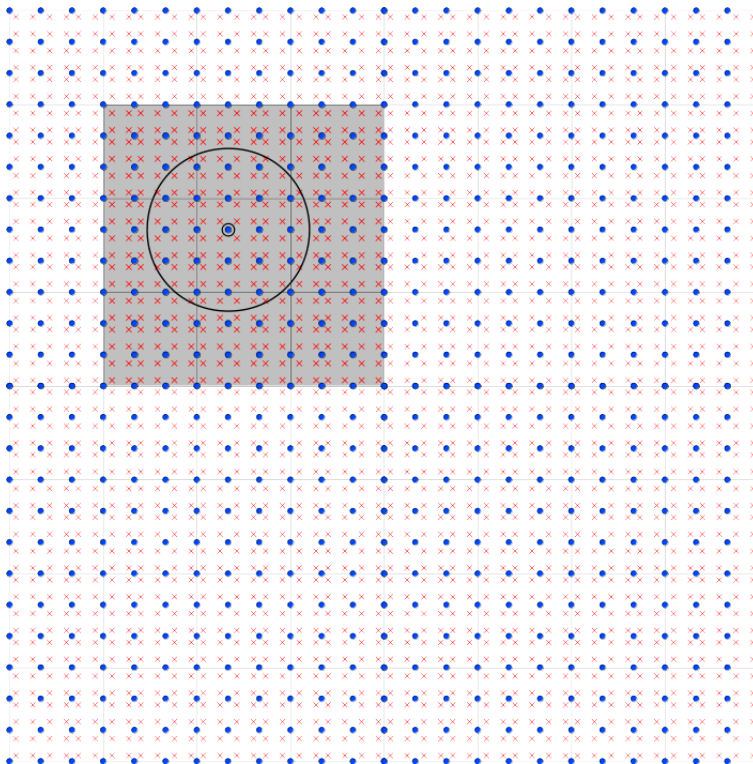


Fig. 7.14: Regioned search for correlations
(nodes: ●, Gauss points ×)

One optimization that can be applied to regioning, is to quickly be able to decide whether a neighboring Gauss region will be search or not. For example, the bottom right grayed region of Fig. 7.14 can be entirely skipped. For that purpose, the centroid of each Gauss region is used as a representative point for the whole region. If the centroid of a Gauss region lies inside the domain of influence of a node, then all Gauss points of that region will be processed for possible correlation with the node, otherwise they will be ignored. However, there may be cases of Gauss points which are inside the domain of influence of a node but the centroid of their Gauss region lies outside the domain of influence, as can be seen in Fig. 7.15. In order to account for such cases, the centroids are tested with regard to an extended domain of influence. The extended domain of influence is only used for the centroids so the contribution of Gauss points is evaluated based on the actual domain of influence of the node.

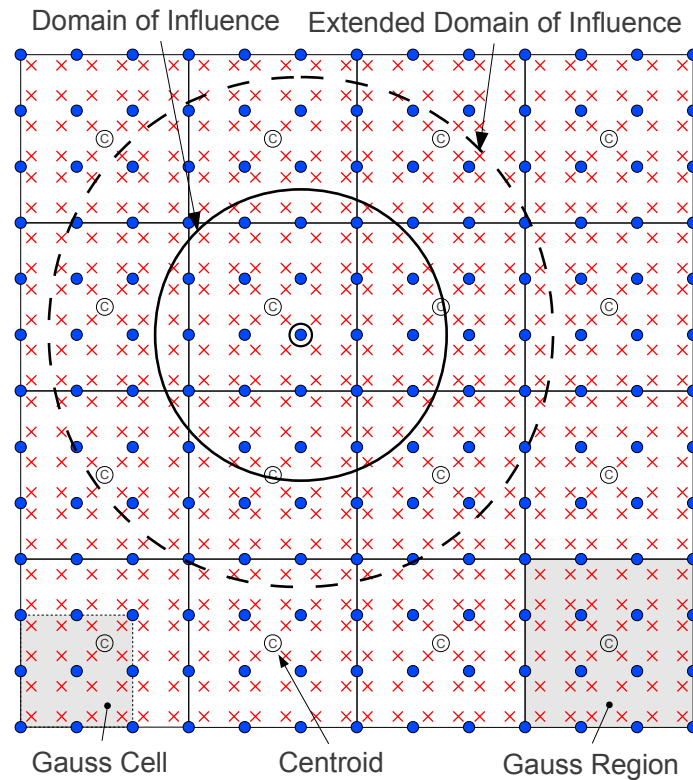


Fig. 7.15: Identifying the influencing Gauss points of node \odot

The extended domain of influence should be large enough to include the centroids of regions that would be outside the actual domain of influence and small enough to avoid false positives, i.e. regions that test true but contain no influencing Gauss points. In order to accomplish this, the maximum distance between the centroid and a point on the border of the respective Gauss region is computed. For rectangular regions, this is simply the half-diagonal. The extended domain of

influence is then defined by adding this distance to the initial domain of influence.

The time required to define correlations in three 2D and three 3D elasticity problems with varying number of degrees of freedom (dof) are shown in Table 7.5. The 2D problems correspond to square domains and the 3D to cubic domains, with rectangular domains of influence (doi) with dimensionless parameter 2.5. These domains maximize the number of correlations and consequently the computational cost for the given number of nodes. In these examples, each Gauss region comprises a single Gauss cell, meaning the the Gauss cell “grid” is re-used as a Gauss region “grid”. Thus, in the 2D examples each Gauss cell/region contains 16 Gauss points (4×4 rule) and in the 3D examples 64 Gauss points ($4 \times 4 \times 4$ rule). The examples are run on a Core i7-980X which has 6 physical cores (12 logical cores) at 3.33GHz and 12MB cache. Each node can define its correlation independently of other nodes, thus the process is amenable to parallel computations.

Example	Nodes	Gauss points	Search Time (seconds)		
			Global Serial	Regioned Serial	Regioned Parallel
2D-1	25,921	102,400	23	1.3	0.5
2D-2	75,625	300,304	300	3.4	1.0
2D-3	126,025	501,264	836	5.4	1.4
3D-1	9,221	64,000	7	3.7	0.9
3D-2	19,683	140,608	45	7.8	1.7
3D-3	35,937	262,144	157	15.7	3.3

Table 7.5: Computing time required for all node-Gauss point correlations

With the implementation of Gauss regions, the initialization phase of MMs in complex domains can take less time than in FEA, since the generation of a finite element mesh can sometimes be laborious and time consuming [91]. At the end of this step, each node has a list of influencing Gauss points and each Gauss point has a list of influenced nodes (Sections 7.1, 7.2).

Another technique that can be used to accelerate geometric searches is through the usage of kd-trees. The running time in that case would be $O(n \log n_G)$, which is lower than the $O(nn_G)$ of regioning, but kd-trees can handle degenerate cases of clustering points more easily. More information on the application of kd-trees is MMs can be found in [92].

7.5.2.3 Interactions and shared Gauss points in MMs

The identification of interactions and synergies (Sections 7.3, 7.4), is also a laborious task in MMs. The naive approach is to check all possible combinations of node pairs and find their shared Gauss points. If there is at least one Gauss point (by definition, see start of Section 7.5) those node are interacting and their synergies are also identified as part of the process. The shared Gauss points are located in the intersection of the domains of influence of two interacting nodes (Fig. 7.16). This approach, however, takes a prohibitive amount of time because it needs to calculate the shared Gauss points for all possible $n(n+1)/2$ combinations of node pairs, where n is the number of nodes. Table 7.6 shows all the number of possible combinations of node pairs, the number of interacting nodes as well as the associated computing time for a naive identification.

The strategies presented here identify the interactions and synergies in separate steps. Having the interactions as a first step avoids unnecessary searches for Gauss points of non-interacting nodes. As in the naive approach, all $n(n+1)/2$ combinations can be checked and if there is at least one Gauss point in common the node pair is marked as interacting. However, this is still a $O(n^2)$ process so it does not scale well and quickly grows into unacceptable running times.

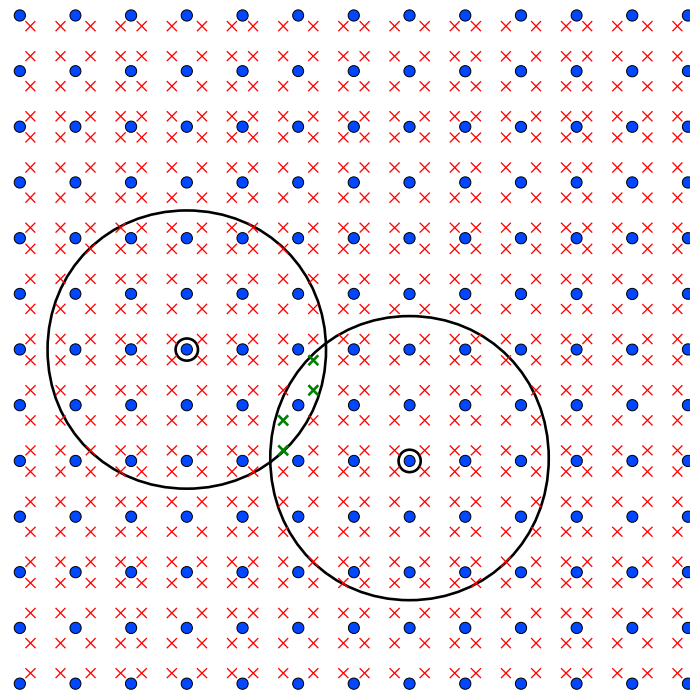


Fig. 7.16: Interaction nodes with their shared Gauss points

Example	Nodes	All combinations	Interacting	Time (seconds)
2D-1	25.921	335.962.081	1.033.981	771
2D-2	75.625	2.859.608.125	3.051.325	6.908
2D-3	126.025	7.941.213.325	5.103.325	23.380
3D-1	9.221	42.518.031	2.418.035	608
3D-2	19.683	193.720.086	5.554.625	3.021
3D-3	35.937	645.751.953	10.644.935	16.290

Table 7.6: Computing time required for a naive identification of interacting nodes and their shared Gauss points

The interactions can be identified by using the N-G and G-N correlations that have already been calculated (Section 7.5.2.2). The generic technique described in Section 7.3 is used and the computing times required are shown in Table 7.7. The examples are run on a Core i7-980X which has 6 physical cores (12 logical cores) at 3.33 GHz. Each node can search for interacting nodes independently of other nodes, so parallelism offers very good acceleration.

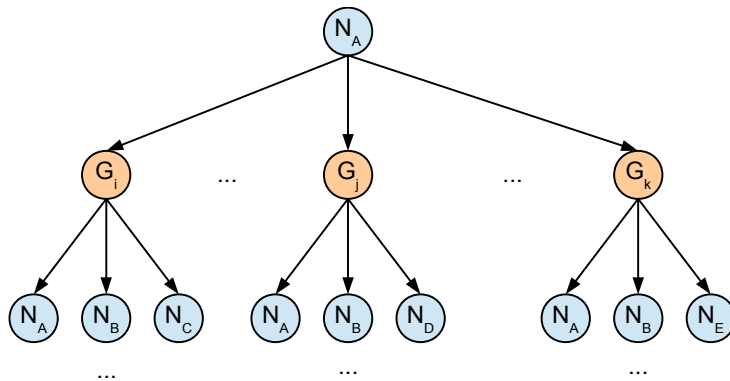


Fig. 7.17: Identifying interacting node pairs for node N_A .

Example	Time (seconds)	
	Serial	Parallel
2D-1	1,5	0,2
2D-2	4,5	0,7
2D-3	9,8	1,6
3D-1	20,1	2,8
3D-2	42,6	5,6
3D-3	85,6	11,2

Table 7.7: Computing time for the identification of interacting nodes

Example	Time (seconds)	
	Serial	Parallel
2D-1	0.2	<0,1
2D-2	0.5	<0,1
2D-3	0.8	<0,1
3D-1	0.5	<0,1
3D-2	0.9	0.2
3D-3	1.6	0.3

Table 7.8: Computing time for the identification of interacting nodes by only inspecting Gauss points near the border

With this approach the identification of interacting nodes is improved, but it can be further accelerated by noting that an interacting node may be in the lists of several Gauss points of N_A , as is node N_B in Fig. 7.17. Since the number of influencing Gauss points of a node is large (1000 for the majority of nodes in our 3D examples), there will be a large amount of duplicates in the process (duplicates are discarded). To reduce the number of duplicates, it is appropriate to only inspect those Gauss points that are near the border of the domain of influence of the node (Fig. 7.18). These Gauss points reach the interactions with further away nodes while including all closer nodes. This considerably reduces the time as can be seen in Table 7.8.

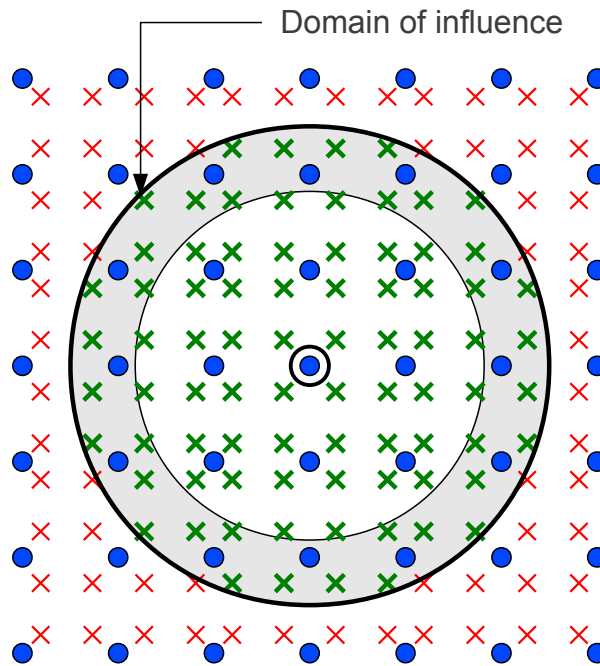


Fig. 7.18 Identifying interacting node pairs by considering Gauss points near the border of the domain of influence

Following the identification of the interacting node pairs, the identification of shared Gauss points is performed the least possible number of times, i.e. only once for every interacting node pair, in contrast to the $n(n+1)/2$ times of the naive approach. This leads to a vast reduction of the required amount of computing time compared to the naive approach (Table 7.6) as can be seen in Table 7.9. The times presented are for the shared Gauss point identification via hash-tables, as described in Section 7.4.

For further improvement, regioning (Fig. 7.19) can be utilized and the results are shown in Table 7.10. The Gauss regions may be the same as those in the correlation phase (Section 7.5.2.2) but can also be different. Shared Gauss points are only searched within regions shared by both node pairs.

In both shared Gauss point identifications, with and without regions, each node pair can identify its shared Gauss points independently of other node pairs, so parallelism offers very good accelerations, as shown in Tables 7.9 and 7.10.

Example	Time (seconds)	
	Serial	Parallel
2D-1	2,1	0,4
2D-2	6,1	1,2
2D-3	8,8	1,5
3D-1	46,6	7,4
3D-2	135,6	18,8
3D-3	315,7	45,8

Table 7.9: Computing time to identify the shared Gauss points of an interacting node pair

Example	Time (seconds)	
	Serial	Parallel
2D-1	2,4	0,6
2D-2	6,8	1,6
2D-3	11,0	2,8
3D-1	24,9	4,8
3D-2	57,9	10,7
3D-3	118,0	22,4

Table 7.10: Computing time to identify the shared Gauss points of an interacting node pair with regioning

In the 2D examples considered, each region has 16 Gauss points and the results are slightly worse with regioning because skipping 16 Gauss points per skipped region was not enough to compensate for the added overhead. Higher number of Gauss points per region eventually makes regioning worthwhile in the 2D examples. In the 3D examples, the extra dimension and the fact that each region has 64 Gauss points makes regioning more important. Regioning benefits become greater as the number of Gauss points per region increases.

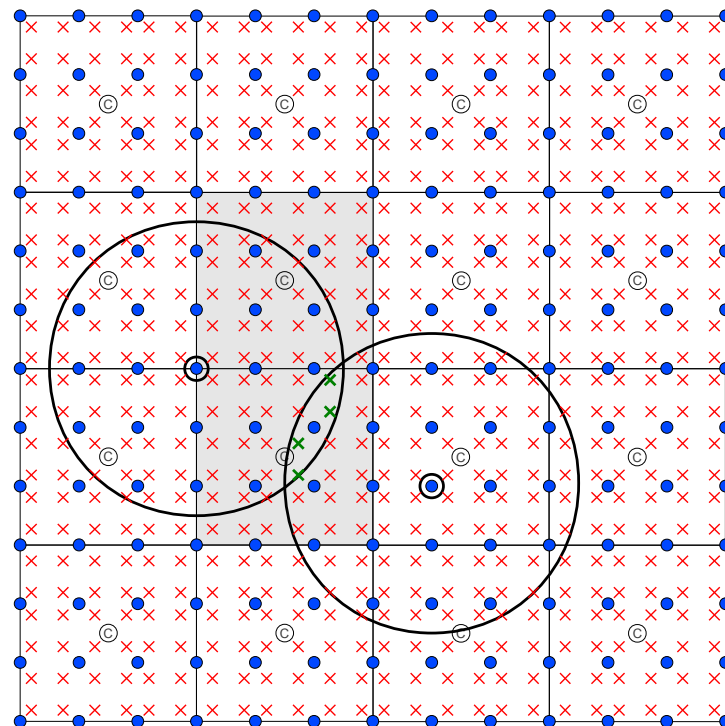


Fig. 7.19: Region-wise search for interacting nodes. Only the shaded regions are inspected for shared Gauss points.

7.5.3 Domain of influence in IGA

The basic entities in IGA are control points and elements. The areas influencing a control point are shown in Fig. 7.20 for various values of p in the 1D case. A comparison between the influencing areas of a control point/node in IGA and FEA is depicted in Fig. 7.21, for 2D case and for different p . It should be noted that the actual correlation is between control points and Gauss points. Elements provide a convenient grouping so the correlations are easier to handle and store. Note that this comparison does not imply the same accuracy for the two methods.

In FEA, each Gauss point is involved in computations with nodes within its own element only. The shape functions and their derivatives are predefined for each element type and need to be evaluated on all combinations of nodes and Gauss points within the element. In IGA, however, each Gauss point is involved in computations with control points of the surrounding areas as well (Fig. 7.22), while the shape functions are not predefined and span across larger domains with a significantly higher amount of control point -Gauss point correlations.

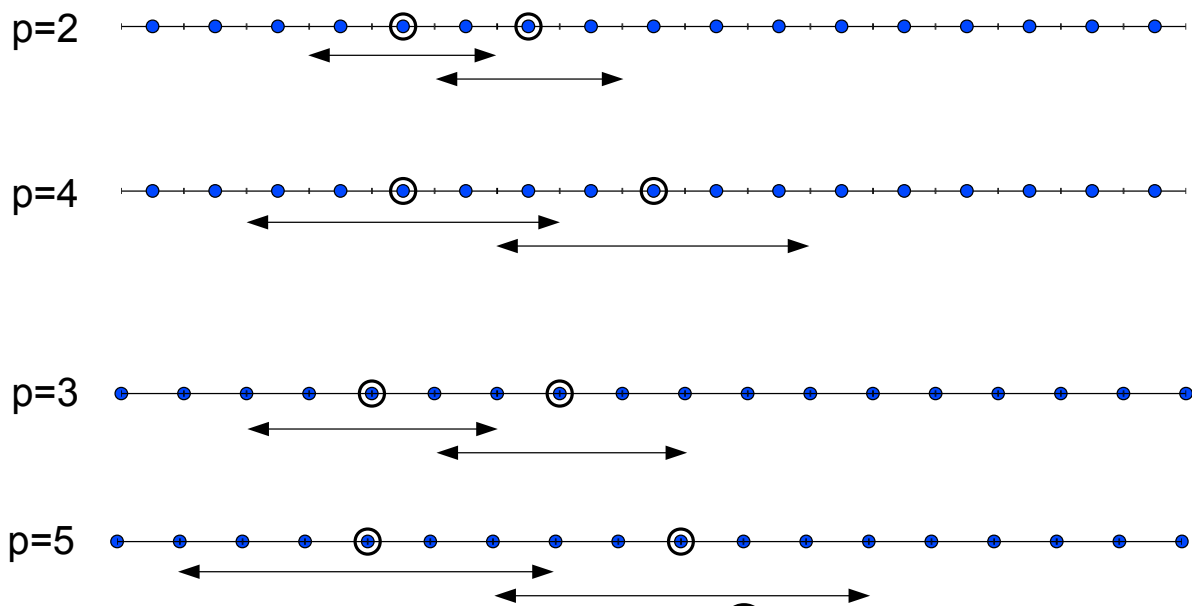


Fig. 7.20. IGA 1D domains of influence of control points \odot for various values of p .

For equivalent meshes, the bandwidth is the same between the two methods but IGA has a larger amount of control point interactions and, consequently, denser stiffness matrices. Furthermore, the computation of each non-zero coefficient is more laborious because the control point pairs have a lot more shared elements (on average) and consequently significantly more Gauss points that contribute to the final values.

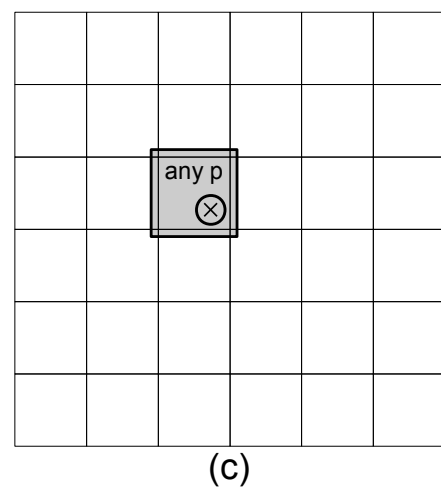
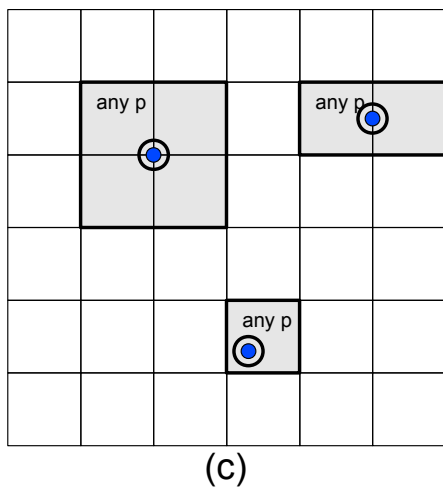
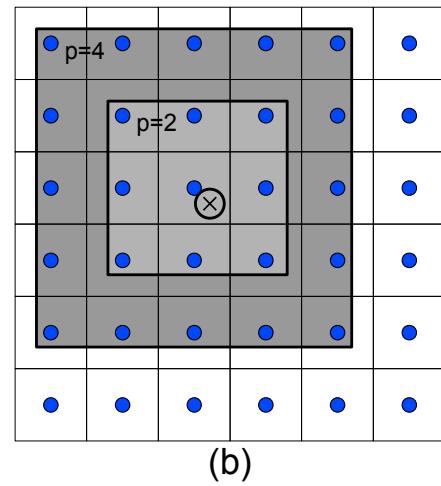
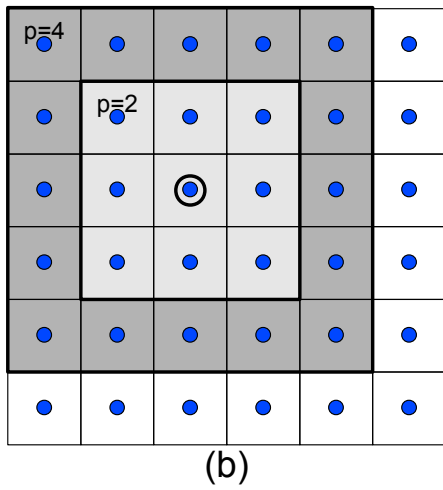
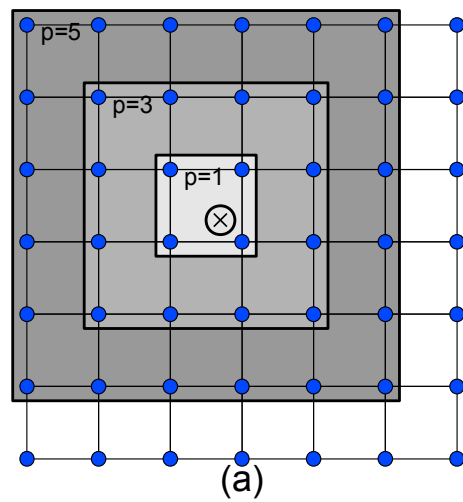
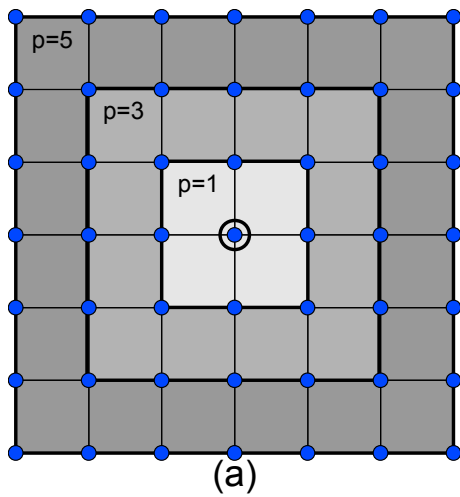


Fig. 7.21. Areas influencing control point \odot in: (a) IGA (p odd); (b) IGA (p even); (c) FEA. The influencing entities are the Gauss points in the shaded areas.

Fig. 7.22. Control points/nodes influenced by Gauss point \otimes in: (a) IGA (p odd); (b) IGA (p even); (c) FEA.

7.5.3.1 Comparison with FEA

Table 7.11 gives a quantitative comparison of the total number of elements and Gauss points in IGA and FEA for $n=121$ control points/nodes in each axis, for 2D and 3D simulations. A $p+1$ integration rule is adopted in both IGA and FEA for the sake of comparison. For large numbers n of control points/nodes, the ratio of elements in IGA and FEA asymptotically approaches p^d , where d is the dimension of the problem (Table 7.12). Since each element has the same number of Gauss points in both IGA and FEA, the ratio holds for both elements and Gauss points.

n = 121		Total Elements		Total Gauss points		Ratio	
p	GP per Element	IGA	FEA	IGA	FEA		
2D	1	4	14,400	14,400	57,600	57,600	1.0
	2	9	14,161	3,600	127,449	32,400	3.9
	3	16	13,924	1,600	222,784	25,600	8.7
	4	25	13,689	900	342,225	22,500	15.2
	5	36	13,456	576	484,416	20,736	23.4
3D	1	8	1,728,000	1,728,000	13,824,000	13,824,000	1.0
	2	27	1,685,159	216,000	45,499,293	5,832,000	7.8
	3	64	1,643,032	64,000	105,154,048	4,096,000	25.7
	4	125	1,601,613	27,000	200,201,625	3,375,000	59.3
	5	216	1,560,896	13,824	337,153,536	2,985,984	112.9

Table 7.11. Total elements and Gauss points in IGA and FEA, for $n = 121$ control points/nodes and different p , in 2D and 3D square and cubic domains.

	Total elements			
	IGA	IGA lim	FEA	FEA lim
1D	$(n-p)^1$	n^1	$[(n-1)/p]^1$	$(n/p)^1$
2D	$(n-p)^2$	n^2	$[(n-1)/p]^2$	$(n/p)^2$
3D	$(n-p)^3$	n^3	$[(n-1)/p]^3$	$(n/p)^3$

Table 7.12. Total elements in IGA and FEA with respect to p .

Figs. 7.23 and 7.24 show a visual comparison when zooming in on a region of the domain with the same number of control points/nodes. Variations in the number of correlations in the boundary of the domain are ignored in the following analysis. Fig. 7.23 depicts the control points/nodes for even numbers of p for IGA and $p=2$ for FEA, while Fig. 7.24 depicts them for odd numbers of p for IGA and $p=3$ for FEA.

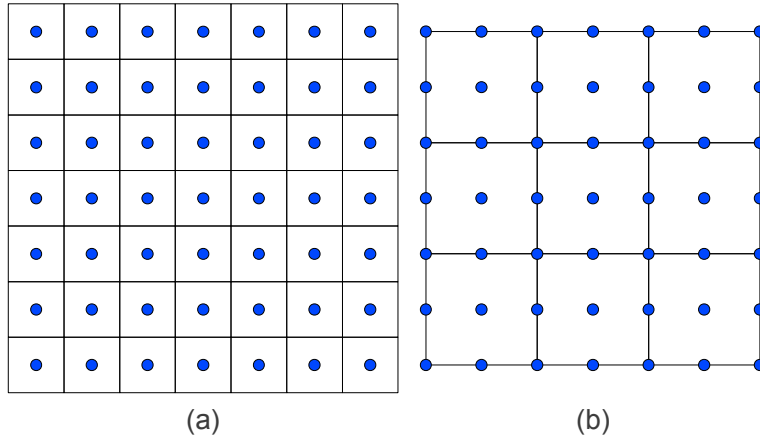


Fig. 7.23. Visual comparison between (a) IGA p =even; (b) FEA p =2, for the same number of control points/nodes.

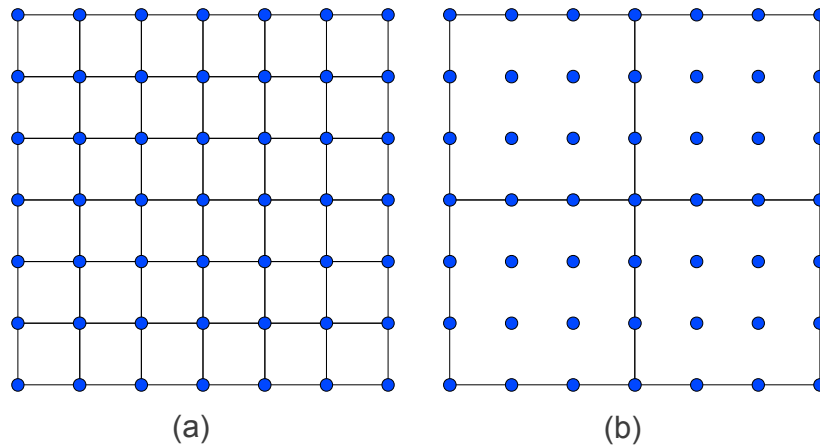


Fig. 7.24. Visual comparison between (a) IGA p =odd; (b) FEA p =3, for the same number of control points/nodes.

Table 7.13 shows the number of correlations of nodes with elements and Gauss points in 2D and 3D problems for FEA. The first column shows the order p while the second shows the number of Gauss points per element. The third column shows the number of nodes influenced by a Gauss point or element. The next columns show the number of elements and the number of Gauss points influencing a node. As can be seen from Figs. 7.23 and 7.24, there are variations as to how many elements/Gauss points influence a node. Corner nodes have the max number of correlations, while internal nodes are only affected by the Gauss points of the element they belong to, while side/edge nodes are in-between.

	p	GP per element	Nodes influenced by GP/element	Elements influencing a node		Gauss Points influencing a node	
				min	max	min	max
2D	1	4	4	4	4	16	16
	2	9	9	1	4	9	36
	3	16	16	1	4	16	64
	4	25	25	1	4	25	100
	5	36	36	1	4	36	144
3D	1	8	8	8	8	64	64
	2	27	27	1	8	27	216
	3	64	64	1	8	64	512
	4	125	125	1	8	125	1,000
	5	216	216	1	8	216	1,728

Table 7.13. Correlations of nodes with elements and Gauss points for FEA.

	p	GP per element	Control points influenced by GP/element	Elements influencing a control point	Gauss points influencing a control point
2D	1	4	4	4	16
	2	9	9	9	81
	3	16	16	16	256
	4	25	25	25	625
	5	36	36	36	1,296
3D	1	8	8	8	64
	2	27	27	27	729
	3	64	64	64	4,096
	4	125	125	125	15,625
	5	216	216	216	46,656

Table 7.14. Correlations of control points with elements and Gauss points for IGA.

Table 7.14 shows the corresponding number of correlations of control points with elements and Gauss points in 2D and 3D problems for IGA. In this case, the way in which a control point is correlated with an element/Gauss point is different and there are no variations in the number of Gauss points or elements affecting a control point.

In FEA each Gauss point affects only the nodes within its own element and the number of nodes is increased with the order p . In IGA each Gauss point affects surrounding areas (range depending on p), but the number of control points affected by each Gauss point is the same as the number of influenced nodes in FEA. On the other hand, each control point is affected by more elements in IGA than in FEA and consequently by a lot more Gauss points. The correlations are increasing much faster in IGA than in FEA as p increases. Table 7.15 shows the number of Gauss points

influencing a control point/node with respect to p , demonstrating the growth rate.

Gauss points influencing a control point/node			
Problem Type	IGA	FEA <i>min</i>	FEA <i>max</i>
1D	$(p+1)^2$	$(p+1)^1$	$2^1 (p+1)^1$
2D	$(p+1)^4$	$(p+1)^2$	$2^2 (p+1)^2$
3D	$(p+1)^6$	$(p+1)^3$	$2^3 (p+1)^3$

Table 7.15. Number of Gauss points influencing a control point/node with respect to p .

It should be noted that in FEA corner nodes are constantly 4 or 8 (for 2D and 3D analysis respectively) and the number of internal nodes increases faster than side/edge nodes. Thus, for higher values of p , most nodes are internal and therefore the average number of Gauss points/element influencing a node is closer to the *min* value than to the *max* value. Therefore, for large problems (where deviations in the boundary can be ignored) and for increasing values of p , the amount of correlations in IGA are approaching the square of those in FEA.

7.5.3.2 Interactions

Due to the structured grid featured in IGA, the interacting control points associated with a specific control point are those located within a fixed range dictated by the order p in each axis.

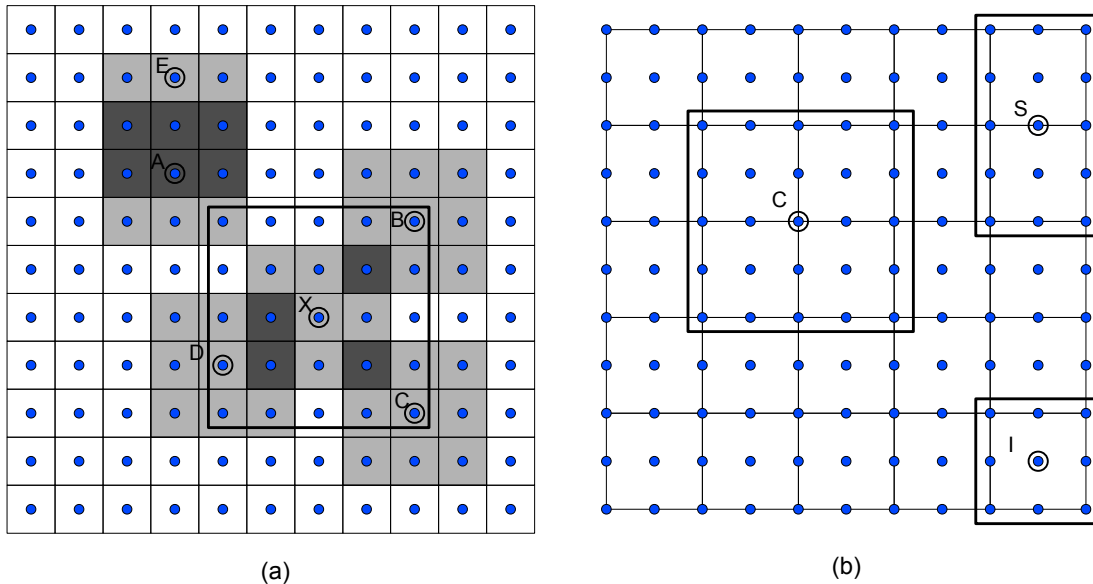


Fig. 7.25. Interacting control points/nodes for $p = 2$: (a) IGA; (b) FEA.

In FEA the nodes interact with nodes in adjacent elements only and thus the interacting node pairs can be easily defined from the element-node connectivity (Fig. 7.25b). In IGA, however, a control

point pair is interacting if there is at least one (non-empty) element shared between the two control points (Fig. 7.25a). Thus, control point X interacts with B,C,D, but not with A or E. If the basis order is p , then the interacting control points extend up to p elements in all directions. This can be observed for $p=2$ in Fig. 7.25a. The gray shaded regions are the influence domains of each control point. The thick-lined rectangles in Fig. 7.25 include all control points/nodes that are interacting with the corresponding control point/node.

Fig. 7.26 shows the interactions for $p=3$. It also shows the case where there is a trivial knot span on the y-axis. This creates a row of elements that are empty and affects interactions due to the general definition of interaction that requires at least one shared Gauss point (see Section 7.5). As a result, X and C are not interacting despite having a common element, since this element has no Gauss points. Control points X and D are still interacting because there is one shared element with Gauss points. The effect of a trivial knot span is that it limits the range of the interaction area in that direction, as can be seen in Fig. 7.26. Similarly, multiple trivial knot spans further limit the range of the interaction.

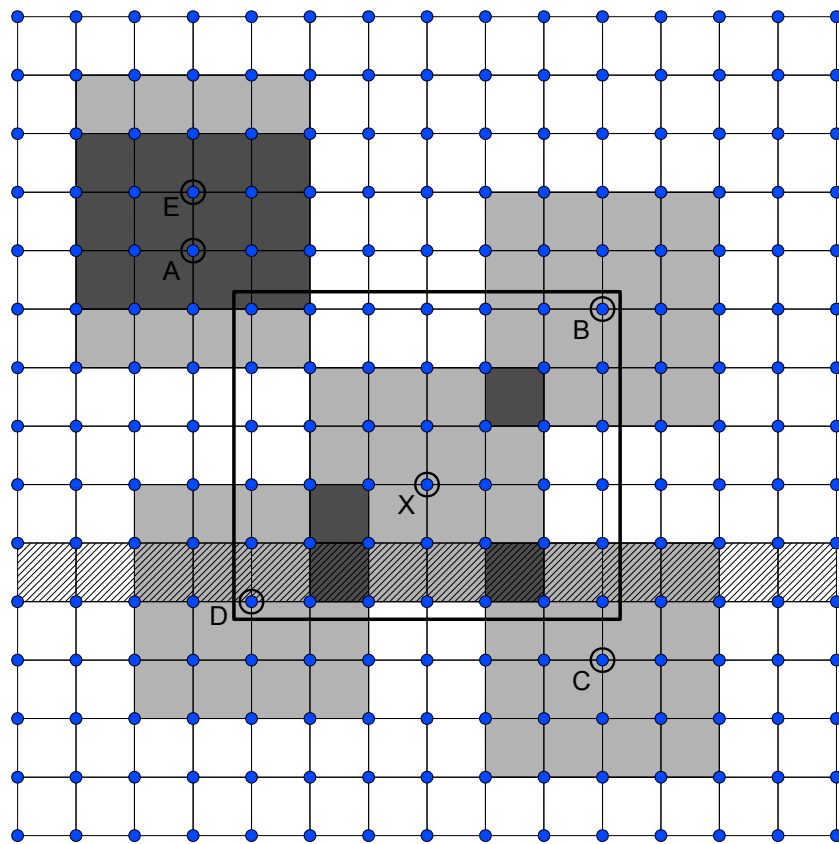


Fig. 7.26. IGA Interacting control points for $p=3$.

7.5.3.3 Interaction comparison with FEA for equal number of freedom degrees

Interactions per control point/node				
	p	IGA	FEA min	FEA max
2D	1	9	4	9
	2	25	9	25
	3	49	16	49
	4	81	25	81
	5	121	36	121
3D	1	27	8	27
	2	125	27	125
	3	343	64	343
	4	729	125	729
	5	1,331	216	1,331

Table 7.16. Interactions per control point/node in IGA and FEA.

Interactions per control point/node			
Problem Type	IGA	FEA min	FEA max
1D	$(2p+1)^1$	$(p+1)^1$	$(2p+1)^1$
2D	$(2p+1)^2$	$(p+1)^2$	$(2p+1)^2$
3D	$(2p+1)^3$	$(p+1)^3$	$(2p+1)^3$

Table 7.17. Interactions per control point/node with respect to p .

Tables 7.16 and 7.17 show the number of interactions per control point/node. The *max* case for FEA is equal to the case in IGA (assuming no trivial knot spans and ignoring boundaries). As it has already been mentioned, internal nodes are increasing faster than other nodes as p increases, so the average of FEA is going to be closer to the *min* values.

There are variations in the number of shared elements between two interacting control points/nodes in both IGA and FEA. Closer nodes have many shared elements, as is the case of points A and E, or only a single one, as is the case of distant elements like X and B in Fig. 7.25a and Fig. 7.26. In FEA, the variations are limited to less shared elements: internal nodes only have a single shared element with any of their interacting nodes, while non-internal nodes have at most 2, in 2D problems, or 4, in 3D problems, shared elements with other non-internal nodes they interact with.

From the above analysis it is clear that the computational effort for assembling the stiffness matrix in IGA is much higher than in FEA for the same number of freedom degrees.

8 Formulation of the characteristic matrices

Matrix formulation can be an expensive part of the simulation, as has been established in Sections 7.5.2, 7.5.3 for MMs and IGA methods. This chapter defines the blueprints of two methods for formulating the characteristic matrices (mostly having the stiffness matrix in mind) before getting into simulation method specifics. The two methods that are explored are the standard contribution-wise method and the parallel-friendly interaction-wise method.

8.1 The contribution-wise (CW) method for assembling a matrix

Assembly by summation of contributions is the typical, straightforward method for assembling matrices with Gaussian quadrature. The matrix is built from the contributions of all Gauss points, as follows:

$$\mathbf{K} = \sum_G w_G \mathbf{Q}_G \quad (8.1)$$

where w_G is the weight factor of the Gauss point and \mathbf{Q}_G is a matrix depending on the application. In structural mechanics applications $\mathbf{Q}_G = \mathbf{B}_G^T \mathbf{E} \mathbf{B}_G$ for FEA, MMs and IGA. The deformation matrix \mathbf{B}_G is computed at the corresponding Gauss point and \mathbf{E} is the constitutive matrix describing the properties of the material. Two variants of the contribution-wise method are considered. The first and more generic one handles Gauss points individually while the second is suitable for element-driven simulations and handles groups of Gauss points through elements.

8.1.1 Gauss point-wise variant of the CW method

In the assembly by Gauss point, each Gauss point needs to be handled individually and may affect different degrees of freedom and consequently different entries of the matrix. This is the most generic approach and is the one which is typically used in methods which do not utilize elements, like MMs. Two major factors that affect the performance of this process are: (i) the calculation of the $w_G \mathbf{Q}_G$ matrices which is performed at every Gauss point and (ii) the indexing time to append the partial matrices to the global matrix. For MMs, these factors have been addressed in [93] and achieved an acceleration of $10\times$. The indexing time for the contributions of each Gauss point to the global stiffness matrix is further explored in this work.

In the Gauss point-wise assembly it is necessary to specify which nodes are affected by each Gauss point. The strategy to obtain them can vary greatly. For example, in FEA the correlations can be easily derived from the element connectivity whereas in MMs a geometric search needs to be performed in order to identify them. Strategies for finding the correlations in MMs methods can be found in [92]. The influenced nodes of each Gauss point are needed for the calculation of the corresponding quadrature values (shape function derivatives for the stiffness matrix in the presented structural mechanics examples) and for placing the contribution of the Gauss point in the appropriate entries of the matrix.

8.1.2 Element-wise variant of the CW method

The assembly by element is the standard approach used in element-driven simulation methods, like FEA and IGA. In a more general sense, this variant applies to any simulation method where a group of Gauss points affect the same degrees of freedom and contribute to the same positions of the matrix and thus can be treated in the same way. The Gauss point-wise assembly can be used in element-driven simulations as well, but utilizing the grouping provided by the elements is highly beneficial. It should be noted that, from the perspective of the assembly, the important entities are still the Gauss points and the elements merely provide a natural grouping.

In this variant, instead of adding each individual Gauss point's contribution to the global matrix, the process is firstly to build the matrix of each element by adding the contributions of all Gauss points G of the element to its local matrix:

$$\mathbf{K}_E = \sum_{E_G} w_G \mathbf{Q}_G = \sum_{E_G} w_G \mathbf{B}_G^T \mathbf{E} \mathbf{B}_G \quad (8.2)$$

Since all Gauss points of an element affect the same entries of the global matrix, making several additions locally has significant savings on indexing time. After local assembly, the global matrix is updated with the collective contribution of all Gauss points of the element:

$$\mathbf{K} = \sum_E \mathbf{K}_E \quad (8.3)$$

For an element with 64 Gauss points, the Gauss point-wise assembly would make 64 times more global entry updates than the element-wise assembly. Thus, the grouping provided by elements reduces the number of required global entry updates and thus mitigates the indexing cost.

In contrast to the sparse global matrix, the element's local matrix is usually fully populated – all entries are non-zero – since it contains only the degrees of freedom relevant to the element. Thus, dense matrix formats, which have very fast indexing, can be used without being wasteful. Furthermore, working on the element level enables better utilization of memory locality which is important in modern processors.

8.2 The interaction-wise (IW) method for assembling a matrix

The standard approach analyzed in the previous sections computes different parts of the sum of eq. (8.1) and gradually sums them together. The interaction-wise approach computes the final values of \mathbf{K}_{ij} submatrices and appends them to the matrix \mathbf{K} . For each node combination $i-j$, \mathbf{K}_{ij} describes the interactions between the two nodes. Each \mathbf{K}_{ij} is formed from contributions by those Gauss points that are shared between the two nodes $i-j$:

$$\mathbf{K}_{ij} = \sum_{Sh.G} w_G \mathbf{Q}_{ij} = \sum_{Sh.G} w_G \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j \quad (8.4)$$

Two nodes are interacting and therefore have a non-zero \mathbf{K}_{ij} if there is at least 1 Gauss point that influences both nodes. We refer to these as shared Gauss points and they are a core part of the IW method. An in-depth analysis of interactions in MMs and IGA is provided in Sections 7.5.2, 7.5.3.

8.2.1 IW variant with individual Gauss points

This is the generic version of the interaction-wise method and, similarly to the contribution-wise assembly with Gauss points, each Gauss point needs to be handled separately and may affect different degrees of freedom and consequently different entries of the matrix.

The computation of the stiffness elements for each interacting node pair is split in two phases. In the first phase, the quadrature values for each influenced node of every Gauss point are calculated as in the Gauss point-wise method. Then, instead of continuing with the calculation of the stiffness matrix coefficients corresponding to a particular Gauss point, the quadrature values are stored for the calculation of \mathbf{Q}_{ij} matrices in the next phase. In our test cases, the memory needed for the quadrature values (shape function derivatives) is relatively small compared to the overall amount of memory needed for the initialization and assembly phases. In the second phase, the stiffness matrix

coefficients of each interacting node pair is computed. For each interacting node pair $i-j$, the matrix $w_G \mathbf{Q}_{ij}$ of eq. (8.4) is calculated over all shared Gauss points and summed to form the final values \mathbf{K}_{ij} of the corresponding coefficients of the global matrix.

Both phases are amenable to parallelization - the first with respect to Gauss points and the second with respect to interacting node pairs - and involve no race conditions or the need for synchronization, which makes the interaction-wise approach an ideal method for massively parallel systems.

8.2.2 IW variant for element-driven applications

In element-driven simulations, the Gauss points enclosed within an element are handled as a group, so for each interacting node eq. (8.4) becomes:

$$\mathbf{K}_{ij} = \sum_{Sh.G} w_G \mathbf{Q}_{ij} = \sum_{Sh.E} \left(\sum_{G_E} w_G \mathbf{Q}_{ij} \right) = \sum_{Sh.E} \left(\sum_{G_E} w_G \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j \right) \quad (8.5)$$

The inner summation is performed over the Gauss points G_E of an element E and calculates the total contribution of element E pertaining to the interacting pair $i-j$, while the contribution of each element is added by the outer summation to reach the final value for the coefficients of $i-j$ by the outer sum. For element-driven applications, it is more convenient to define interacting nodes if there is at least one element that influences both nodes, but since the actual influencing entities are the Gauss points, care must be taken in cases where elements may contain no Gauss points, a situation which is possible in IGA.

8.3 Scatter-gather parallelism of the matrix assembly methods

The most important advantage of the interaction-wise approach is its amenability to parallelism, especially in massively parallel systems. Since each Gauss entity may affect a large number of nodal entities, as is the case in MMs and IGA, each \mathbf{K}_{ij} submatrix is formed by a large number of Gauss entity contributions. Parallelizing the contribution-wise approach involves scatter parallelism, which is schematically shown in Fig. 8.1 for two Gauss entities C and D . Each part of the sum can be calculated in parallel but there are conflicting updates to the same entries of the stiffness matrix. These race conditions can be avoided with proper synchronization but in massively parallel systems, where thousands of threads may be working concurrently, it is very detrimental to performance because all updates are serialized with atomic operations [94].

In the interaction-wise approach, instead of constantly updating the matrix, the final values for the submatrix \mathbf{K}_{ij} of each interacting pair $i-j$ are calculated and then appended to the matrix. For the calculation of a submatrix \mathbf{K}_{ij} , all contributions of the Gauss entities belonging to the intersection of the domains of influence of an interacting pair should be summed together. Thus, the interaction-wise approach utilizes gather parallelism as shown schematically in Fig. 8.2.

In a parallel implementation, each thread prepares a submatrix \mathbf{K}_{ij} related to a specific interacting pair $i-j$. It gathers all contributions from the Gauss entities and writes to a specific memory location accessed by no other thread. Thus, this method requires no synchronization or atomic operations.

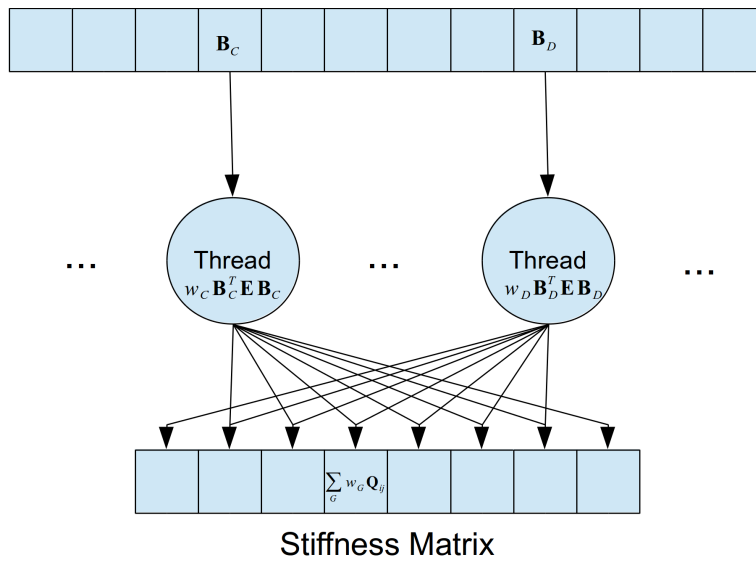


Fig. 8.1: Scatter parallelism in the contribution-wise approach.

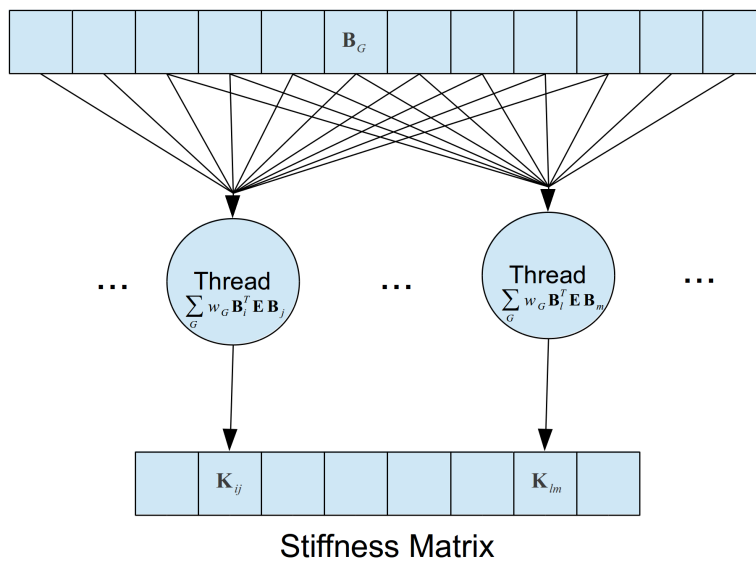


Fig. 8.2: Gather parallelism in the interaction-wise approach.

8.4 GPU implementation of the interaction-wise approach

The interaction-wise approach for the assembly of the stiffness matrix is well-suited for the GPU. The calculations are split into two kernels, each one utilizing different levels of parallelism. There are different versions for both phases depending on whether the analysis deals with individual Gauss points or elements. GPU details are given in Chapter 4. The implementations in this work are written in openCL for greater portability [93], [95].

8.4.1 Phase 1 – Calculation of quadrature values

In the first phase, application-specific quadrature values are calculated on the influenced entities of every Gauss point. For example, in structural mechanics problems the shape function values are needed for the mass matrix and the shape function derivatives are needed for the stiffness matrix. In this phase, the calculations are performed at each Gauss point, so in element-driven simulations this phase will need to access the Gauss points grouped by an element.

8.4.1.1 Individual Gauss point variant

In this section we demonstrate the calculation of quadrature values calculation for cases where Gauss points are handled individually. The calculations are independent between Gauss points and as such can be processed in parallel. Two levels of parallelism are exploited: the primary over the Gauss points and the secondary over the influenced nodes. A thread block/group is assigned to each Gauss point and each thread handles one influenced node at a time. This is schematically shown in Fig. 8.3, where it is assumed that each thread handles a single influencing node. The number of threads in a block/group is chosen as a power of two (see Section 4.7) but the number of influenced nodes i is not necessarily a power of two. Thus, threads $i+1$ and $i+2$ do not have corresponding influenced nodes.

The calculations on the nodes of a particular Gauss point are usually interdependent to each other. For example, in EFG simulations the moment matrix, which is derived from contributions from all influenced nodes, has to be calculated and inverted. Phases like these are not embarrassingly parallel but the only synchronization needed is within a block of threads as each Gauss point is completely independent from other Gauss points.

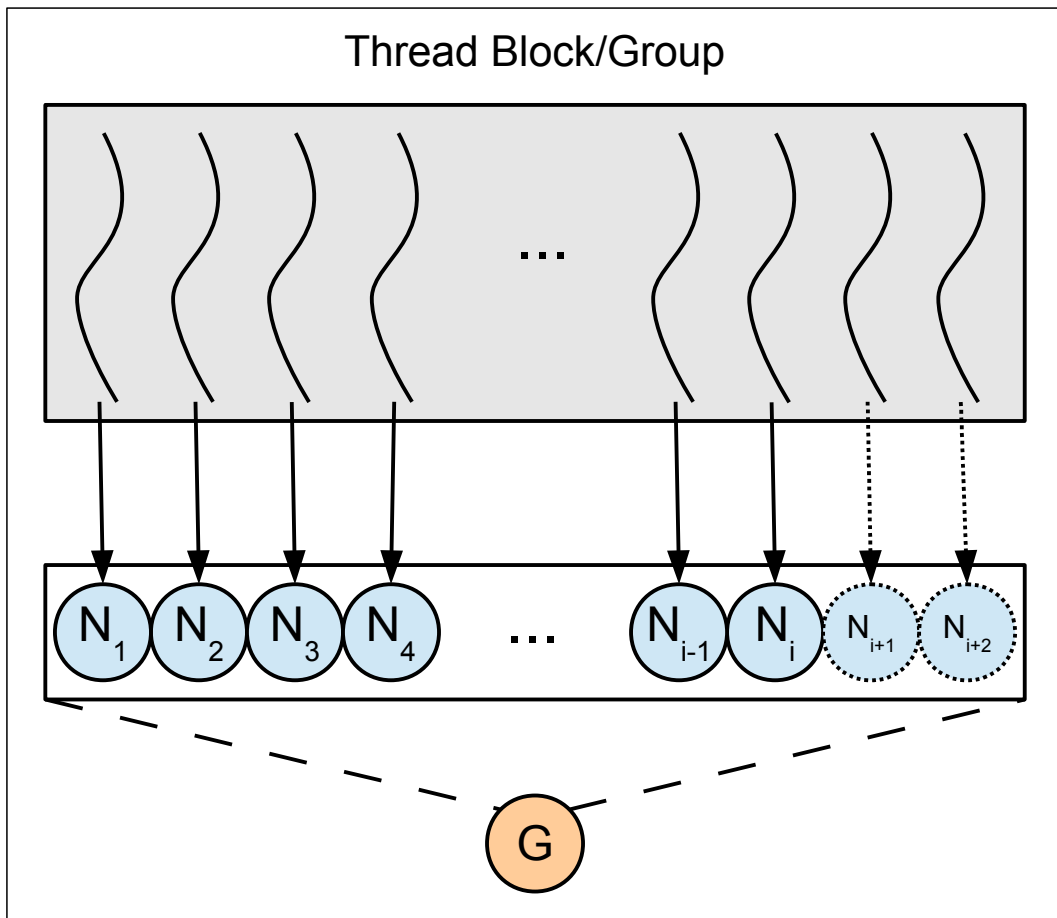


Fig. 8.3: Thread organization in phase 1 for the individual Gauss point variant. A thread block/group is assigned to Gauss point G and each thread of the block/group is assigned to an influenced node N_i of G .

Since each Gauss point has its own thread block, all data related to a particular Gauss point is stored in the shared/local memory. In EFG simulation, this includes the aforementioned moment matrix and several vectors. Each thread is assigned to an influenced node so node-specific data can be stored in registers or shared/local memory. Gauss point-specific data needs to be accessed by all threads of the block assigned to that Gauss point, usually multiple times, so utilizing the shared/local memory is most appropriate and greatly reduces expensive accesses to the global memory. In fact, interaction with the global memory is performed only at the beginning of the process, i.e. when loading input data, and at the end of the process where the resulting values are written to the global memory. In both cases the accesses are coalesced. Constant memory is used for storing analysis parameters, like the ranges of the influence domains (EFG) and can also be used for storing nodal data (e.g. coordinates) which are needed by all blocks. However, the small size of the

constant memory (64KB total in contemporary GPUs) dictates that such data must be appropriately small, otherwise the global memory needs to be used. Thus, all calculations are performed with data found in fast memories.

8.4.1.2 Element variant

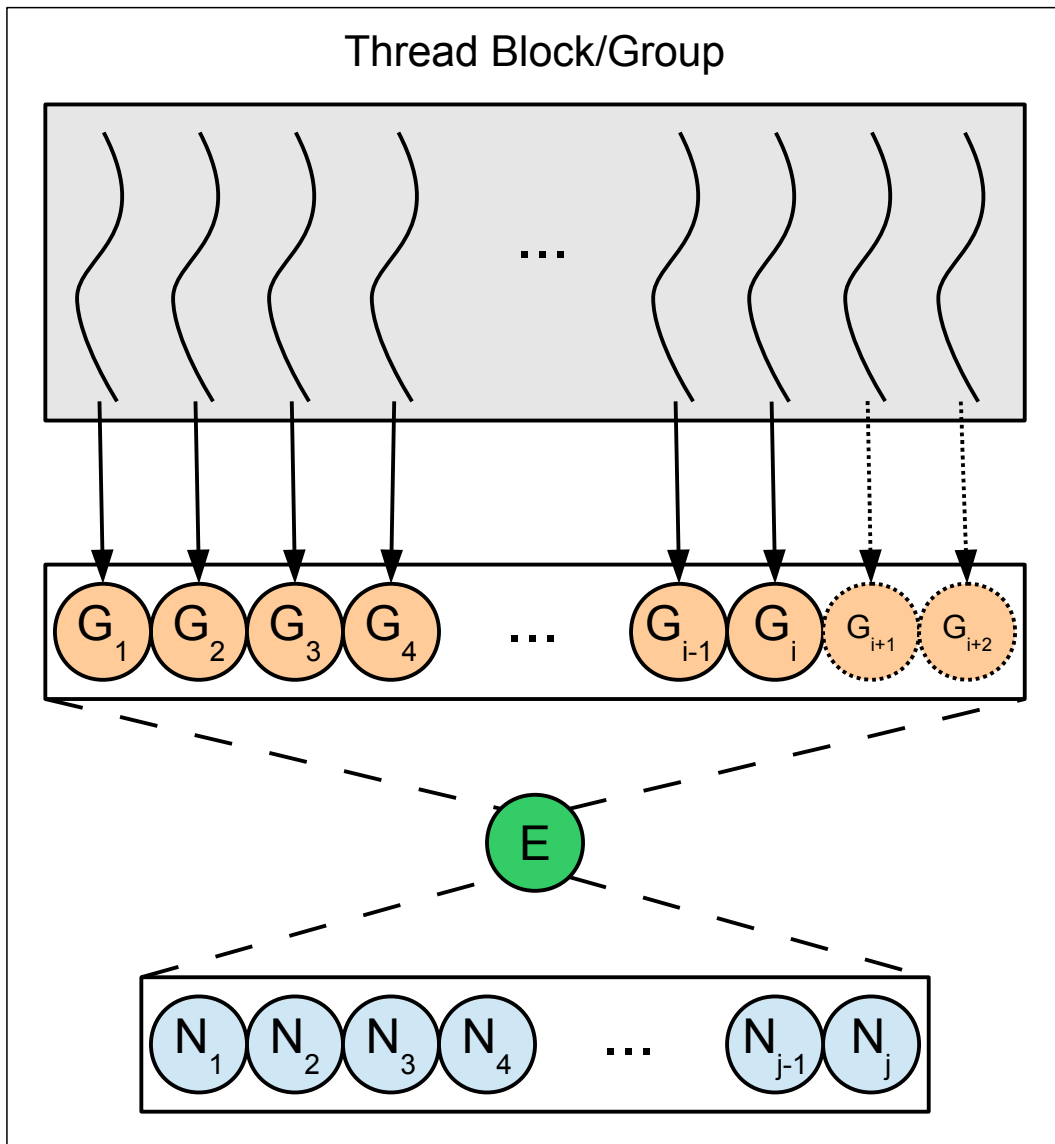


Fig. 8.4: Thread organization in phase I for the element variant. A thread block/group is assigned to element E and each thread of the block/group is assigned to a Gauss point G_i of E . Each thread iterates over all influenced nodes N_j .

This variant utilizes elements and generally has better parallel behavior and memory utilization because all Gauss points of a particular element have the same influenced nodes. Two levels of parallelism are exploited here as well, the primary over the elements and the secondary over the

Gauss points. A thread block/group is assigned to each element and each thread handles one Gauss point at a time and iterates over all influenced nodes. The thread organization is schematically shown in Fig. 8.4, where it is assumed that each thread handles a single Gauss point. There are i Gauss points, but the block has a power-of-two number of threads, so excess threads do not have corresponding Gauss points. Each thread iterates over the influenced nodes so the number of influenced nodes is unrelated to the number of threads in a block/group and not subject to the constraints outlined in Section 4.7.

The element variant has several favorable differences over the individual Gauss point variant. Each element is assigned to a specific thread block and thus all values related to a particular element (i.e. Gauss point values as well as nodal entity values) are stored in the shared/local memory so they can be accessed efficiently by all threads in the block. The element values can be transferred to the shared/local memory in a coalesced manner and node-related data are also shared within a block so there is better usage of on-chip memories. Since all threads of a block iterate over the same number of influenced nodes, they are all tasked with an equal amount of work and thus there is no thread divergence which would have a negative impact on performance. The calculation of quadrature values for each Gauss point is independent from other Gauss points and thus threads in a block need no synchronization between them.

8.4.2 Phase 2 – Calculation of matrix entries

In the second and final phase, the submatrices \mathbf{K}_{ij} are calculated. For the sparse matrix format used by the interaction-wise approach (COO), this entails the indexes and values of the submatrices. A straightforward scheme is to process each interacting pair concurrently. For the GPU implementation however, we opt for a second level of parallelism with respect to shared Gauss points. The output data of the previous phase, namely the quadrature values, are used as input in this phase. The quadrature values already reside in the GPU if calculated there, as described in the previous section, otherwise they are copied there.

8.4.2.1 Two-level individual Gauss point variant

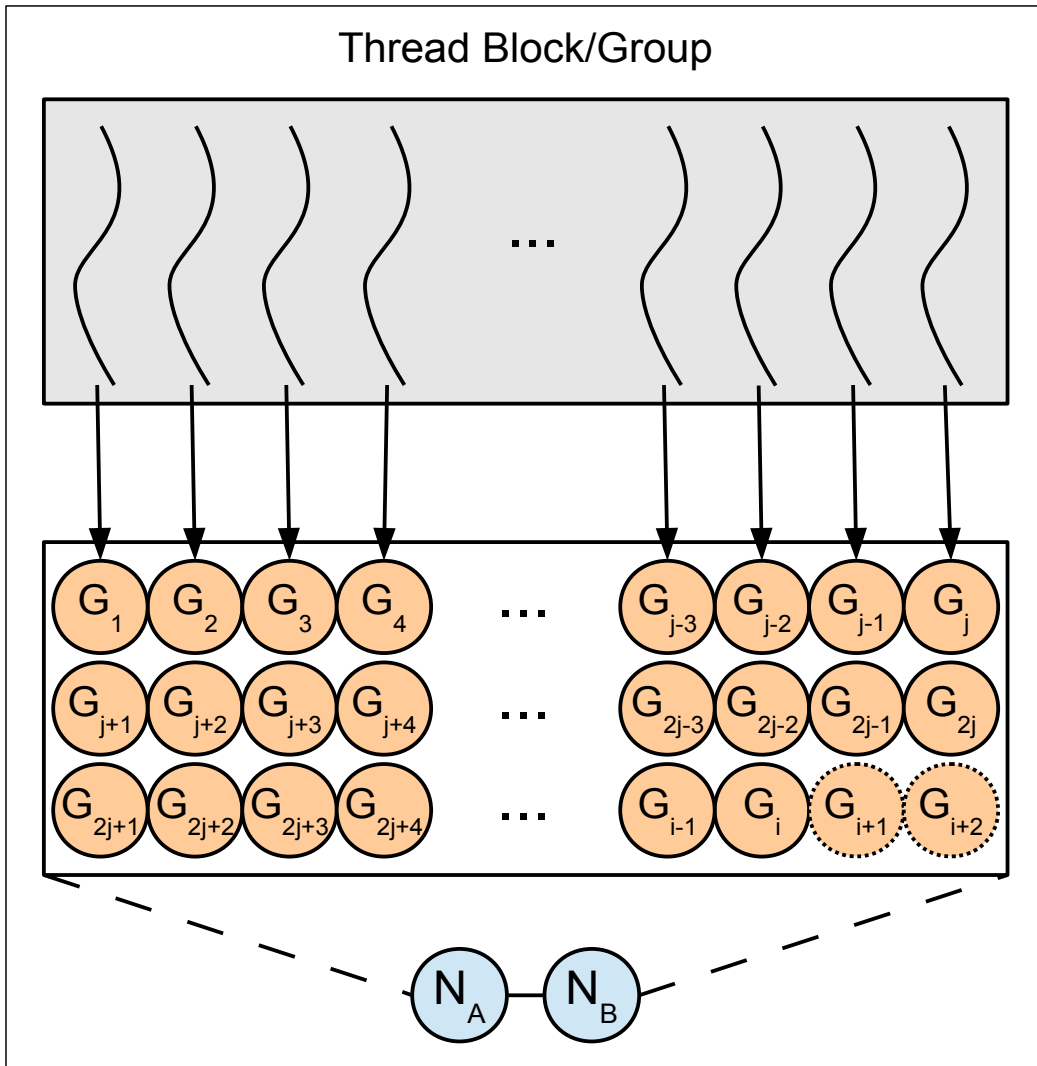


Fig. 8.5: Thread organization in phase 2 for the individual Gauss point variant showing the threads assigned to the shared Gauss points G of interacting pair $N_A - N_B$.

As in phase 1, there are two levels of parallelism, the primary one over interacting pairs and the secondary one over Gauss points. A thread block/group is assigned to each interacting pair and each thread of the block handles one Gauss point at a time. This is schematically shown in Fig. 8.5, where it is assumed that each thread handles (up to) 3 shared Gauss points. In each of the first two passes, the block handles a number of Gauss points equal to the number of threads j . In the last pass, threads $i+1$ and $i+2$ of Fig. 8.5 do not have corresponding Gauss points because i is not necessarily a multiple of the number of threads.

In this phase, all threads of a block go through all available shared Gauss points of the interacting pair and calculate the $w_G \mathbf{Q}_{ij}$ submatrices (eq. 8.4), as described in section 8.2.1. Each thread t of the block sums contributions from different shared Gauss points and updates its own partial \mathbf{K}_{ij}^t so there is no need for atomic operations. After all shared Gauss points have been processed, the partial \mathbf{K}_{ij}^t matrices of each thread of the block are summed with a parallel reduction into the final values of the stiffness coefficients \mathbf{K}_{ij} . Each step of the parallel reduction strategy divides the number of partial sums by half and ultimately produces the final sum after $\log_2 N$. Thread utilization within a block is shown in Fig. 8.6.

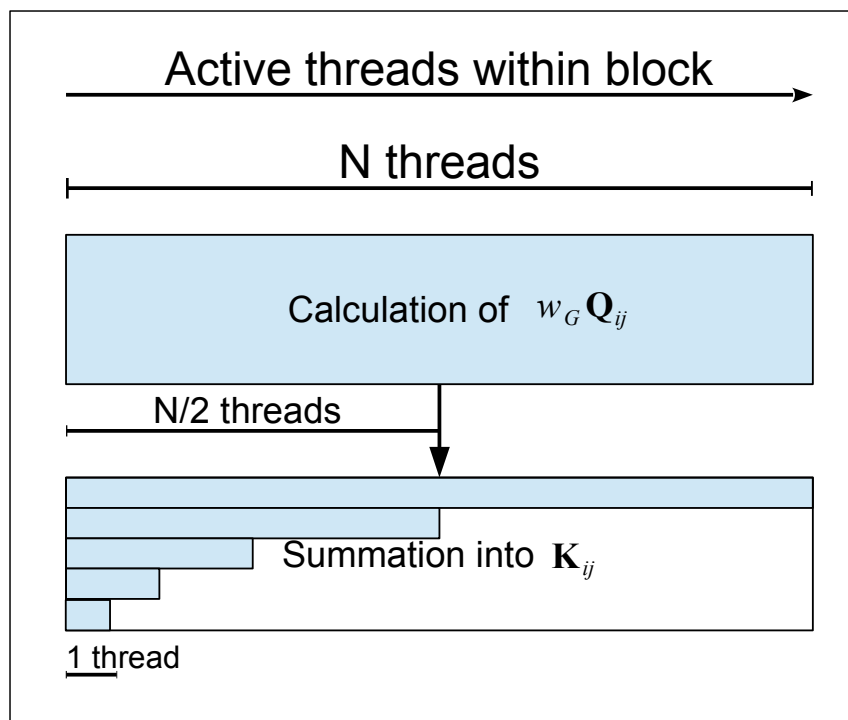


Fig. 8.6: Phase 2: concurrency within a block/group for the assembly phase in the GPU.

Using the GPU for reductions, e.g. as part of a vector dot product, is a well-studied problem [71]. However, the reductions used in the proposed implementations are only within a group/block of threads, not on a global level. That is, all reductions are within a block and not between blocks and thus no global synchronization is needed. Furthermore, the aforementioned reductions exist only because of an implementation choice: multiple threads are assigned to a pair so reductions are needed to reach the final values for the corresponding pair, but those reductions are isolated within that particular group of threads. If desired, a single thread can be assigned to each pair thus obviating reductions. However, a block/group of threads is assigned to each pair due to the amount

of work involved and in order to better utilize the particular properties of the GPU. It should also be noted that only the fast shared/local memories are involved in the reductions.

8.4.2.2 Two-level element variant

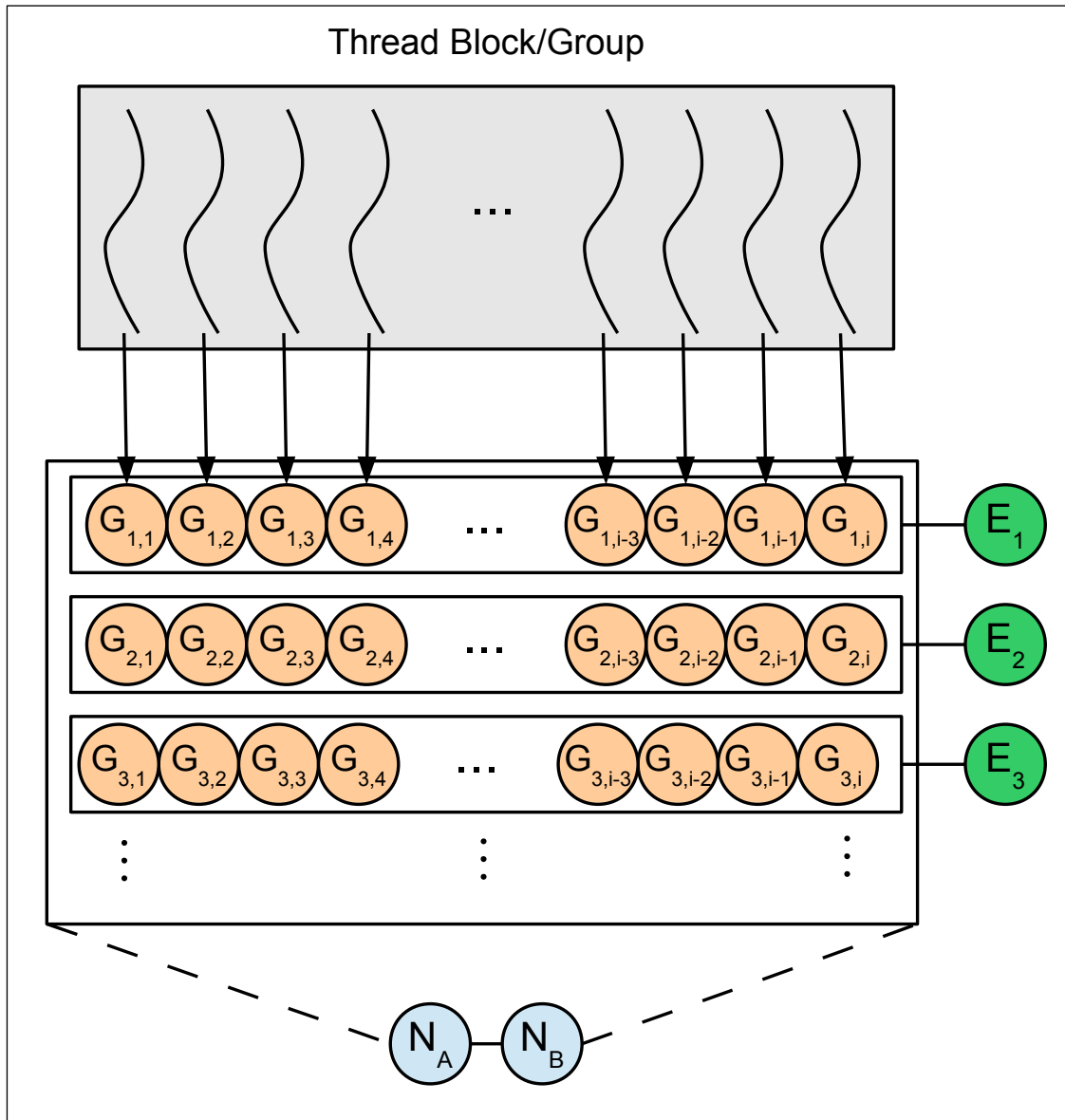


Fig. 8.7. Thread organization in phase 2 for the element variant showing the threads assigned to the Gauss points G of the shared elements E of interacting pair $N_A - N_B$.

This is similar to the individual Gauss point variant, except that the shared entities are elements instead of Gauss points. The two levels of parallelism are the same as in the Gauss point variant, i.e. the primary one over interacting pairs and the secondary one over Gauss points. A thread block/group is assigned to each interacting pair and each thread of the block handles one Gauss

point from the shared elements at a time. In the individual Gauss point variant, each shared Gauss point reference had to be done separately, but in this variant the shared entities are elements and thus for each element referenced, there is a group of Gauss points that will need to be processed. This leads to more efficient memory usage, because significantly fewer reads are required for the same number of calculations than in the previous variant. Furthermore, the reads are more coalesced. In fact, with enough number of Gauss points per element, the memory accesses can be fully coalesced, thus utilizing the hardware more efficiently.

This variant is schematically shown in Fig. 8.7, where it is assumed that each thread handles only one Gauss point from each of the three shared elements shown. Thread utilization within a block is the same as in Fig. 8.6.

8.5 Memory layout of quadrature values for coalesced access

The quadrature values (e.g. shape function derivatives) are initially grouped by Gauss point since this is the natural order for their calculation. This yields a memory layout which is depicted in Fig. 8.8 for an element with 4 Gauss points and 4 nodes. The quadrature values are needed in phase 2 of the GPU implementation, but the values are accessed by nodal entity in the interaction-wise approach instead of by Gauss point as in the contribution-wise approach. The values are scattered in various memory locations and therefore accesses from consecutive threads are not going to be coalesced.

In general, when accessing global memory, peak performance occurs when all threads in a warp access continuous memory locations. Thus, the desired memory layout for an element with 4 nodes and 4 Gauss points is the one depicted in Fig. 8.9. After the calculation of the quadrature values, a transformation between the two memory layouts is performed. The transformation takes place during phase 1 of the GPU implementation because it has conducive levels of parallelism (block \rightarrow elements, threads \rightarrow Gauss points). The resulting memory layout ensures that consecutive threads access consecutive memory locations, as shown in Fig. 8.9, thus achieving memory coalescence.

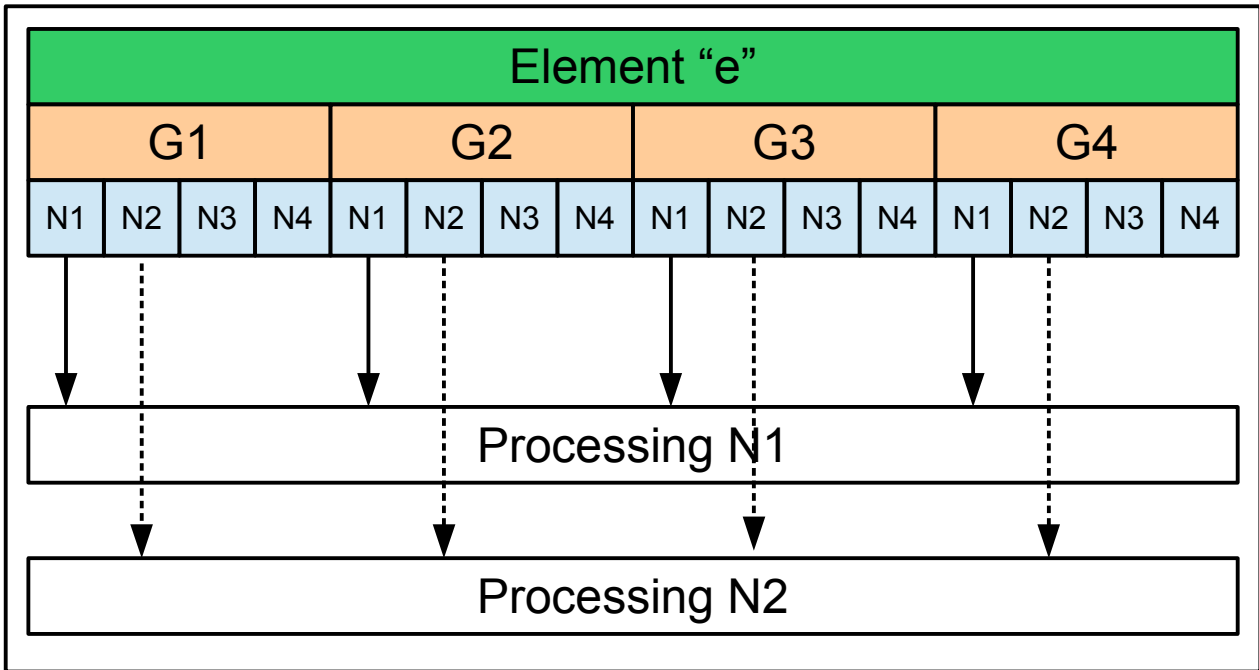


Fig. 8.8. Memory layout of quadrature values of an element "e". The values are grouped together by Gauss point (G), and each group contains values for all influenced nodal entities (N). This leads to non-coalesced access pattern: consecutive threads access non-consecutive memory locations.

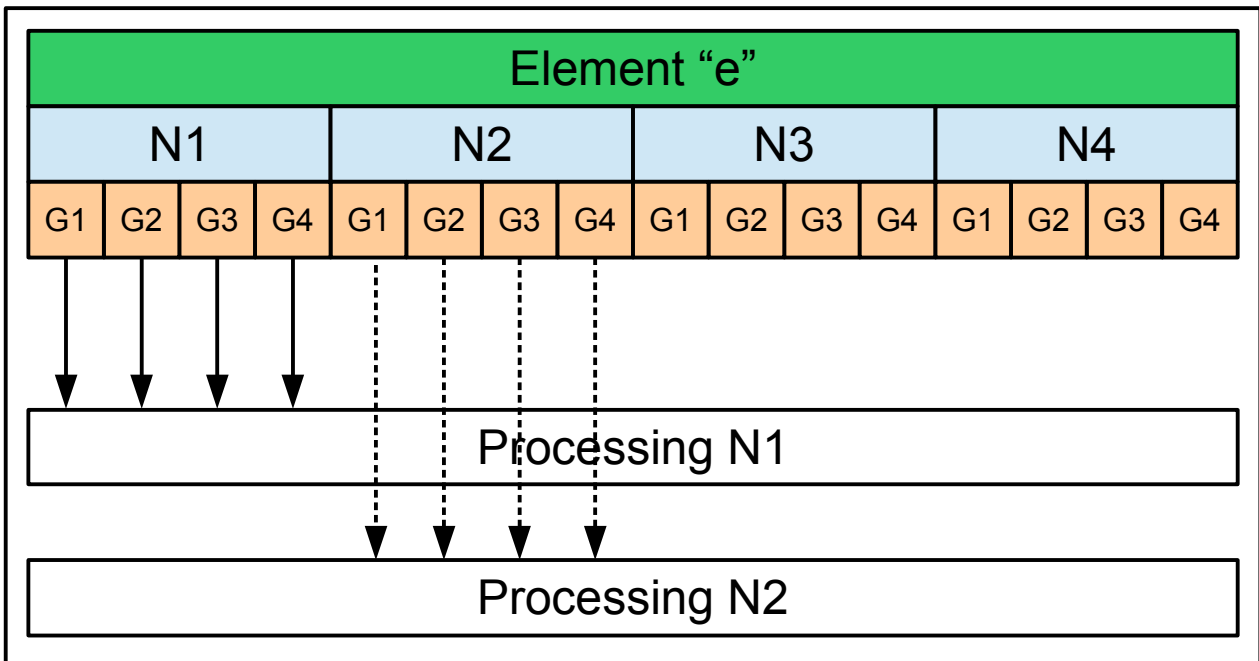


Fig. 8.9. Memory layout of quadrature values of an element "e". The values are grouped together by nodal entity (N), and each group contains values for all influencing Gauss points (G). This leads to coalesced access pattern: consecutive threads access consecutive memory locations.

8.6 Utilization of available hardware

In order to take advantage of the available hardware, e.g. for processing by multiple GPUs, CPUs and/or workstations which may contain several GPU(s) and CPU(s), the assembly of the matrix can be split into multiple independent parts by exploiting the fact that interacting pairs in the interaction-wise approach are completely independent to each other.

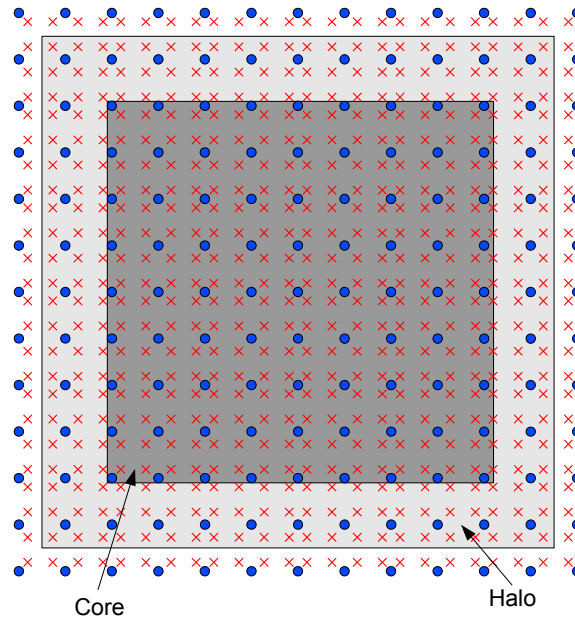


Fig. 8.10: Fully processing the nodes of the core also requires data from the halo surrounding it.

Ideally, the independent parts should be contiguous. Fig. 8.10 shows a group of nodes in the dark shaded area that are assigned to a particular processor. For calculations pertaining to these nodes, the processor needs access to the data inside the dark area as well as the data of the surrounding area, denoted as a halo. The halo includes nodes interacting with the core nodes as well as all Gauss points present in the core nodes' synergies. Since every part needs access only to the aforementioned subset of the data, contiguous parts minimize the amount of data that would need to be transferred to processors and the number of times quadrature values are re-calculated in different processors. In IGA, MMs and FEA the GTX 680's 2GB of global memory is sufficient to handle large enough chunks of the domain so that the halos are much smaller compared to the core. Newer GPUs like the GTX Titan offer 6GB of global memory and thus can handle even larger chunks.

A simple partitioning scheme can be employed to create parts with approximately equal workload. If the number of correlations is roughly the same across the domain, then a simple split into parts of equal number of nodes is sufficient for providing approximately equal amount of work to the

processors. More sophisticated load balance can be achieved by inspecting the number of interactions of the node as well as the synergies of the corresponding pairs. The load balancing becomes even more complex in heterogeneous computing environments, where equal partitions lead to idle times.

In a domain decomposition-based analysis, the matrix computation can be handled in different ways. For large subdomains, the previously described techniques can be applied at the level of each subdomain, resulting in multiple processors working concurrently to produce the characteristic matrices of a subdomain. Smaller subdomains may be handled concurrently by different processors, utilizing the natural parallelism offered in subdomain-based simulations. However, in the case of very small subdomains, the hardware is underutilized if only one subdomain is processed by each processor at a time. A reasonable approach in this case is to assign many subdomains per processor. This is appropriate when the subdomains require much fewer resources than those available in a processor. Sometimes, however, a particular resource may prevent assigning enough subdomains to efficiently utilize the hardware. For example, if a subdomain requires 51% of the global memory of the GPU, a second subdomain cannot be assigned to the GPU due to insufficient memory.

The fine-grained parallelism offered by the interaction-wise approach allows the implementation of a better scheme. This scheme enables dynamic load balancing and it is applicable to all cases discussed in this section while also providing high hardware utilization, even in a heterogeneous environment. Interacting pairs are produced and added to a pool (Fig. 8.11). Processing can be done by any available CPUs, GPUs or other processing units thanks to the huge amount of interacting pairs and the fact that each pair is completely independent of other pairs. Each processor is supplied with as many pairs as it can handle at a time in an asynchronous manner. Upon finishing a chunk of pairs, a processor pulls another chunk from the pool, continuing until all pairs are exhausted.

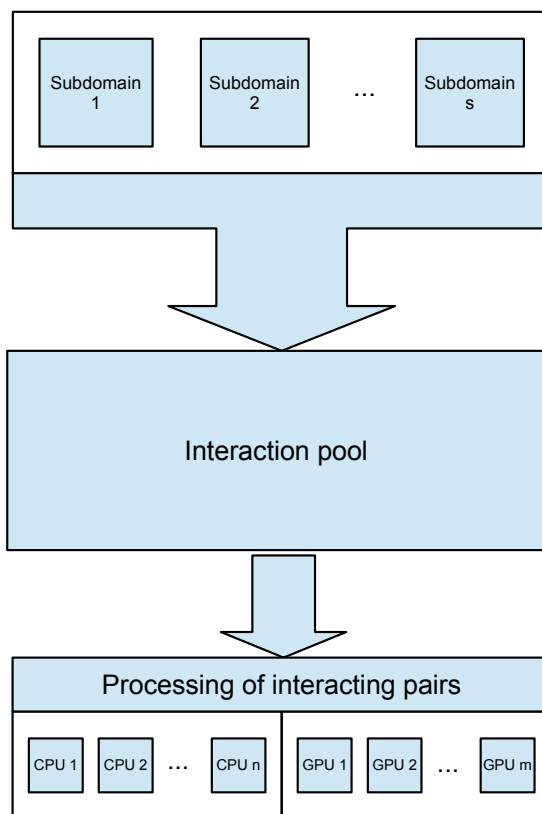


Fig. 8.11: Schematic representation of the processing of node pairs utilizing all available hardware.

8.7 GPU accelerated formulation of the EFG stiffness matrix

The individual Gauss point variants of the assembly methods are used for the assembly of the EFG stiffness matrix.

8.7.1 Computation of stiffness contribution for each Gauss point

8.7.1.1 Shape function derivative calculation

The shape functions in EFG formulation span across larger domains of influence than in FEA and their evaluation is performed over a large number of correlated Gauss points-nodes. For the evaluation of the deformation matrix \mathbf{B} the shape functions and their derivatives are calculated with the following procedure for each Gauss point (see Section 2.1): (i) Calculate the weight function coefficients w , w_x , w_y , w_z for each node in the domain of influence of the Gauss point. (ii) Calculate the moment matrix \mathbf{A} and its derivatives, \mathbf{A}_x , \mathbf{A}_y , \mathbf{A}_z of the Gauss point with contributions from all influenced nodes. (iii) Use the moment matrix and its derivatives along with the weight coefficients to calculate the shape function and derivative values for all influenced nodes of the Gauss point.

The moment matrix and its derivatives are functions of the polynomial \mathbf{p} , which is a complete polynomial of order q for any material point of the domain. In the case of a linear basis (see Section 2.1), the moment matrix \mathbf{A} and its derivatives are 3×3 or 4×4 matrices for 2D and 3D elasticity problems, respectively. The contribution of each node to the moment matrix and its derivatives is related to the product $\mathbf{p}\mathbf{p}^T$. The moment matrix and its derivatives are given by:

$$\begin{aligned} \mathbf{A} &= \sum_i w_i (\mathbf{p}\mathbf{p}^T)_i, \quad \forall i \in Infl.Nodes \\ \mathbf{A}_x &= \sum_i (w_x)_i (\mathbf{p}\mathbf{p}^T)_i \quad \mathbf{A}_y = \sum_i (w_y)_i (\mathbf{p}\mathbf{p}^T)_i \quad \mathbf{A}_z = \sum_i (w_z)_i (\mathbf{p}\mathbf{p}^T)_i, \quad \forall i \in Infl.Nodes \end{aligned} \quad (8.6)$$

Thus, the moment matrix consists of the following terms

$$\mathbf{A} = \begin{bmatrix} \sum_i w_i & \sum_i w_i x_i & \sum_i w_i y_i \\ & \sum_i w_i x_i^2 & \sum_i w_i x_i y_i \\ & & \sum_i w_i y_i^2 \end{bmatrix} \quad (8.7)$$

while similar expressions define its derivatives \mathbf{A}_x , \mathbf{A}_y , \mathbf{A}_z .

The shape function value $\Phi_i(\mathbf{x})$ associated with node i at point \mathbf{x} and the derivatives while the derivatives $\Phi_{i,x}$, $\Phi_{i,y}$, $\Phi_{i,z}$ are given by

$$\Phi_i(\mathbf{x}) = \mathbf{p}_G^T \mathbf{A}^{-1} w_i \mathbf{p}_i \quad (8.8)$$

$$\begin{aligned} \Phi_{i,x} &= w_{i,x} \mathbf{p}_G^T (\mathbf{A}^{-1} \mathbf{p}_i) + w_i \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) + (-w_i) \mathbf{p}_G^T \mathbf{A}^{-1} \mathbf{A}_x (\mathbf{A}^{-1} \mathbf{p}_i) \\ \Phi_{i,y} &= w_{i,y} \mathbf{p}_G^T (\mathbf{A}^{-1} \mathbf{p}_i) + w_i \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) + (-w_i) \mathbf{p}_G^T \mathbf{A}^{-1} \mathbf{A}_y (\mathbf{A}^{-1} \mathbf{p}_i) \\ \Phi_{i,z} &= w_{i,z} \mathbf{p}_G^T (\mathbf{A}^{-1} \mathbf{p}_i) + w_i \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) + (-w_i) \mathbf{p}_G^T \mathbf{A}^{-1} \mathbf{A}_z (\mathbf{A}^{-1} \mathbf{p}_i) \end{aligned} \quad (8.9)$$

where the polynomials \mathbf{p}_G and \mathbf{p}_i are evaluated at the Gauss point G and the influenced node i , respectively. In equations (8.8) and (8.9) the following operations are repeated for all influenced nodes of a Gauss point:

$$\mathbf{p}_A^T = \mathbf{p}_G^T \mathbf{A}^{-1} \quad \mathbf{p}_{Ax} = \mathbf{p}_A^T \mathbf{A}_x \mathbf{A}^{-1} \quad \mathbf{p}_{Ay} = \mathbf{p}_A^T \mathbf{A}_y \mathbf{A}^{-1} \quad \mathbf{p}_{Az} = \mathbf{p}_A^T \mathbf{A}_z \mathbf{A}^{-1} \quad (8.10)$$

These matrix-vector multiplications can be reused in several calculations for every influenced node of a particular Gauss point. For large size of the moment matrix \mathbf{A} , the direct computation of its inverse is burdensome, so an LU factorization is typically performed [2]. In this implementation, an explicit algorithm is used for the inversion of the moment matrix in order to minimize the calculations.

For each influenced node i , the following three groups of calculations are then performed:

$$\begin{aligned} \Phi_i &= w_i \mathbf{p}_A^T \mathbf{p}_i & \Phi_{i,x}^{(2)} &= w_i \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) & \Phi_{i,x}^{(3)} &= -w_i \mathbf{p}_{Ax}^T \mathbf{p}_i \\ \Phi_{i,x}^{(1)} &= w_{i,x} \mathbf{p}_A^T \mathbf{p}_i & \Phi_{i,y}^{(2)} &= w_i \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) & \Phi_{i,y}^{(3)} &= -w_i \mathbf{p}_{Ay}^T \mathbf{p}_i \\ \Phi_{i,y}^{(1)} &= w_{i,y} \mathbf{p}_A^T \mathbf{p}_i & \Phi_{i,z}^{(2)} &= w_i \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} (\mathbf{A}^{-1} \mathbf{p}_i) & \Phi_{i,z}^{(3)} &= -w_i \mathbf{p}_{Az}^T \mathbf{p}_i \\ \Phi_{i,z}^{(1)} &= w_{i,z} \mathbf{p}_A^T \mathbf{p}_i & & & & \end{aligned} \quad (8.11)$$

8.7.1.2 B^TEB Calculation

A fast computation of the matrix product (Section 8.1):

$$\mathbf{Q}_G = \mathbf{B}_G^T \mathbf{E} \mathbf{B}_G \quad (8.12)$$

is important because it is repeated at each integration point. This may not be so critical in FEA compared to the total simulation time, but it is very important in MMs where the number of Gauss points and the number of influenced nodes per Gauss point are both significantly greater.

The computations of eq. (8.12) can be broken into smaller operations for each combination of influenced nodes i, j belonging to the domain of influence of the Gauss point:

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j = \mathbf{Q}_{ji}^T \quad (8.13)$$

Once a submatrix \mathbf{Q}_{ij} is calculated, it is added to the corresponding positions of the stiffness matrix \mathbf{K} . The computation of \mathbf{Q}_{ij} together with the associated indexing to access the entries of \mathbf{K} dominate the total effort for the formulation of the global stiffness matrix [96].

The \mathbf{Q}_{ij} for an isotropic material in 3D elasticity takes the form:

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j = \begin{matrix} (3 \times 3) & (3 \times 6) & (6 \times 6) & (6 \times 3) \end{matrix} \begin{bmatrix} \Phi_{i,x} & 0 & 0 & \Phi_{i,y} & 0 & \Phi_{i,z} \\ 0 & \Phi_{i,y} & 0 & \Phi_{i,x} & \Phi_{i,z} & 0 \\ 0 & 0 & \Phi_{i,z} & 0 & \Phi_{i,y} & \Phi_{i,x} \end{bmatrix} \begin{matrix} M & \lambda & \lambda \\ \lambda & M & \lambda \\ \lambda & \lambda & M \end{matrix} \begin{matrix} \mu \\ \mu \\ \mu \end{matrix} \begin{bmatrix} \Phi_{j,x} & 0 & 0 \\ 0 & \Phi_{j,y} & 0 \\ 0 & 0 & \Phi_{j,z} \\ \Phi_{j,y} & \Phi_{j,x} & 0 \\ 0 & \Phi_{j,z} & \Phi_{j,y} \\ \Phi_{j,z} & 0 & \Phi_{j,x} \end{bmatrix} \quad (8.14)$$

$$\mathbf{Q}_{ij} = \begin{matrix} (3 \times 3) \end{matrix} \begin{bmatrix} \Phi_{i,x} \Phi_{j,x} M + \Phi_{i,y} \Phi_{j,y} \mu + \Phi_{i,z} \Phi_{j,z} \mu & \Phi_{i,x} \Phi_{j,y} \lambda + \Phi_{i,y} \Phi_{j,x} \mu & \Phi_{i,x} \Phi_{j,z} \lambda + \Phi_{i,z} \Phi_{j,x} \mu \\ \Phi_{i,y} \Phi_{j,x} \lambda + \Phi_{i,x} \Phi_{j,y} \mu & \Phi_{i,y} \Phi_{j,y} M + \Phi_{i,x} \Phi_{j,x} \mu + \Phi_{i,z} \Phi_{j,z} \mu & \Phi_{i,y} \Phi_{j,z} \lambda + \Phi_{i,z} \Phi_{j,y} \mu \\ \Phi_{i,z} \Phi_{j,x} \lambda + \Phi_{i,x} \Phi_{j,z} \mu & \Phi_{i,z} \Phi_{j,y} \lambda + \Phi_{i,y} \Phi_{j,z} \mu & \Phi_{i,z} \Phi_{j,z} M + \Phi_{i,y} \Phi_{j,y} \mu + \Phi_{i,x} \Phi_{j,x} \mu \end{bmatrix}$$

\mathbf{E} and $\mathbf{B}_i / \mathbf{B}_j$ are never formed. Instead three values for \mathbf{E} , the two Lamé parameters λ , μ and the P-Wave modulus $M = 2\mu + \lambda$ and three values for \mathbf{B}_i , specifically $N_{i,x}$, $N_{i,y}$, $N_{i,z}$, are stored. Since some of the multiplications are repeated, the calculations in eq. (8.14) can be efficiently performed with 30 multiplications and 12 additions. More detailed analysis for the $\mathbf{B}^T \mathbf{E} \mathbf{B}$ calculation expanding to other types of material as well can be found in Appendix A.

8.7.2 Performance of the Gauss point-wise variant of the CW method

During the contribution-wise assembly, the global matrix coefficients are continually updated with new contributions. Thus, a matrix format with fast indexing time is needed for quickly looking up and updating the relevant matrix entries. This is especially important in the Gauss point-wise variant since each individual Gauss point's contribution needs to be directly appended to the global matrix. The dense and skyline formats feature very fast indexing time but require prohibitive amount of memory in large-scale simulations. Thus, an efficient implementation for building the stiffness matrix in sparse format is needed for the Gauss point-wise approach.

The sparse matrix formats are discussed more thoroughly in Section 5.6. Relevant details are repeated here. The compressed sparse row (CSR) and compressed sparse column (CSC) are the most widely used sparse formats. These formats are extensively used in linear algebra libraries and the CSC is used in MATLAB. These formats are good for matrix operations but for the assembly phase there are more suitable options like the dictionary of keys (DOK), the list of lists (LIL) and the coordinate list (COO). The COO format maintains a list of row-column-value tuples while the DOK format maintains a dictionary mapping of row-column tuples to values. The LIL format stores one list per row, where each entry stores a column index and value, typically sorted. Note that, after the assembly phase, the matrix should be converted to a format more suitable for the solution phase, like CSR/CSC if sparse solvers are utilized.

There can be several variations in the implementations depending on the specific needs of the application. For the contribution-wise matrix assembly, the process requires updating previous values of the matrix, thus a sparse matrix type that allows fast lookups is needed. Updates happen a large number of times for every non-zero element of the matrix, so they can take considerable amount of time. Since our main concern is lookup, implementations that feature fast lookups are desired. In particular, two types of implementation are tested (see Section 5.6.1.2): the first implementation is backed by a single hash-table, whose keys are based on both the row and column index of a particular element, and exhibits $O(1)$ lookups. The second implementation has a list of hash tables, one for each row instead of a single hash-table. The hash table for the appropriate row is selected from the list, and a $O(1)$ lookup is performed on that using only the column index as key. Alternatively, a column-major approach, i.e. having one hash-table for each column can be used. Both implementations have constant lookup time and space proportional to the number of non-zero entries.

Table 8.1 provides relevant metrics for the EFG examples considered while Table 8.2 provides computing times for the two sparse matrix builder implementations tested. The examples are run on a Core i7-980X which has 6 physical cores (12 logical cores) at 3.33GHz and 12MB cache. All floating point calculations are in double-precision arithmetic. Overall running time when using a list of hash-tables was about 30% better compared to using a single hash table. The computing times include the calculation of shape function derivatives and the matrix assembly. Both matrix implementations utilize symmetry and store the upper triangle of the matrix.

EFG Example	Total non-zero entries	Total entry updates	Average updates per entry
2D-1	4,110,003	128,367,712	31
2D-2	12,129,675	379,126,672	31
2D-3	20,287,275	634,249,872	31
3D-1	21,734,532	3,653,625,888	168
3D-2	49,932,576	8,438,820,768	169
3D-3	95,696,604	16,225,940,448	170

Table 8.1: Metrics of the EFG 2D and 3D elasticity problems.

Since the total formulation time is heavily affected by indexing time, a matrix format with better indexing properties, like the skyline format (Section 5.5.2), would be very beneficial. The skyline format stores the values in a column-wise manner from the diagonal up to the last non-zero entry of the column. This means that zero entries in that span are included as well. All matrix entries are stored in a preallocated vector and an additional vector is used to denote the index range of each column. Thus, the skyline format exhibits faster indexing time but increased memory requirements compared to a sparse matrix format which contains only non-zero entries.

EFG Example	Time (s)		Ratio
	Single Hash T.	List of Hash T.	
2D-1	9.8	7.6	1.28
2D-2	28	22	1.28
2D-3	44	34	1.28
3D-1	197	150	1.31
3D-2	506	386	1.31
3D-3	955	724	1.32

Table 8.2: Computing times for the formulation of the EFG stiffness matrix with the Gauss point-wise method for different sparse matrix builder implementations.

EFG Example	Time (s)		Ratio
	Sparse	Skyline	
2D-1	7.6	7.3	1.05
2D-2	22	20	1.12
2D-3	34	31	1.10
3D-1	150	68	2.21
3D-2	386	174	2.21
3D-3	724	329	2.20

Table 8.3: Computing times for the formulation of the EFG stiffness matrix when using sparse and skyline matrix formats in the Gauss point-wise approach.

Tables 8.3 and 8.4 compare the Gauss point-wise approach for building the stiffness matrix in EFG methods when using sparse and skyline formats. The skyline format does perform better, as shown in Table 8.3, but it stores significantly more elements, as shown in Table 8.4, and thus requires considerably more memory compared to the sparse format. It should also be noted that the skyline format is dependent on the numbering of nodes in the domain and the comparison is made with optimum numbering. This dependency may lead to even more zeros stored in real-world applications and further exacerbate the required memory of the skyline format, whereas sparse formats always have the same amount of elements regardless of numbering.

EFG Example	Number of Stored Entries		Ratio
	Skyline	Sparse	
2D-1	66,221,715	4,110,003	16
2D-2	331,150,875	12,129,675	27
2D-3	713,161,275	20,287,275	35
3D-1	136,041,444	21,734,532	6
3D-2	486,852,444	49,932,576	10
3D-3	1,343,011,428	95,696,604	14

Table 8.4: Stored entries comparison of the EFG stiffness matrix when using sparse and skyline matrix formats.

8.7.3 Performance of the interaction-wise approach

The final values of each \mathbf{K}_{ij} submatrix are calculated and written once in the corresponding positions of the global stiffness matrix instead of being gradually updated as in the contribution-wise approach. Apart from the reduced number of accesses to the matrix, this method does not require lookups, which allows the use of a simpler and more efficient sparse matrix format, like the coordinate list (COO) format (Section 5.6.1.1). A simple implementation based on three arrays, one for row indexes, one for column indexes and one for the value of each non-zero matrix coefficient is sufficient and is easily applied both in the CPU and the GPU. The memory required is less than in the implementations discussed in Section 8.7.2, though still proportional to the number of non-zero entries. Note that the interaction-wise method has no indexing time due to its nature, in contrast to the contribution-wise approach.

EFG Example	Time (s)
2D-1	9.3
2D-2	22
2D-3	36
3D-1	112
3D-2	269
3D-3	525

Table 8.5: Computing times for the formulation of the EFG stiffness matrix when using the interaction-wise variant with individual Gauss points.

Table 8.5 shows the computing times required for the interaction-wise variant with Gauss points in the EFG test examples. It can be seen that the interaction-wise approach performed better than the contribution-wise approach with sparse formats (Table 8.2), but not from the memory-intensive skyline format (Table 8.3).

8.7.4 GPU implementation of the interaction-wise approach

Contrary to the contribution-wise approach, the interaction-wise approach for the formation of the stiffness matrix is well suited for the GPU. There are two phases, as described in Section 8.7.4.1, and each one of the two phases is calculated with its own kernel and exhibits different levels of parallelism. The implementations in in this section are written in openCL for greater portability.

8.7.4.1 Phase 1 – Calculation of shape function and derivative values

In the first phase the shape function and its derivatives are calculated for all influenced nodes of every Gauss point. The calculations in this phase are described in detail in Section 8.7.1.1. There are two levels of parallelism: the major over the Gauss points and the minor over the influenced nodes. A thread block/group is assigned to each Gauss point and each thread handles one influenced node at a time. This is schematically shown in Fig. 8.3 where N corresponds to nodes and G corresponds to Gauss points.

For the most part of this phase all threads of a block are busy. The exceptions are the inversion of the moment matrix \mathbf{A} and the reductions which are used to sum the contributions of all influenced nodes in the moment matrix \mathbf{A} and the vectors \mathbf{p}_A , \mathbf{p}_{Ax} , \mathbf{p}_{Ay} , \mathbf{p}_{Az} . The process is shown schematically in Fig. 8.12.

Since each Gauss point has its own thread block, all values related to a particular Gauss point are stored in the shared/local memory. This includes the moment matrix and all vectors (\mathbf{p}_A , \mathbf{p}_{Ax} , \mathbf{p}_{Ay} , \mathbf{p}_{Az}). The interaction with the global memory is performed only at the beginning of the process, where each thread reads the coordinates of the corresponding Gauss point and influenced node and stores them in registers, and at the end of the process where the resulting shape function values are written to the global memory. Constant memory is used for storing the ranges of the influence domains. As a result, all calculations are performed with data found in fast memories which is very beneficial from a performance point of view.

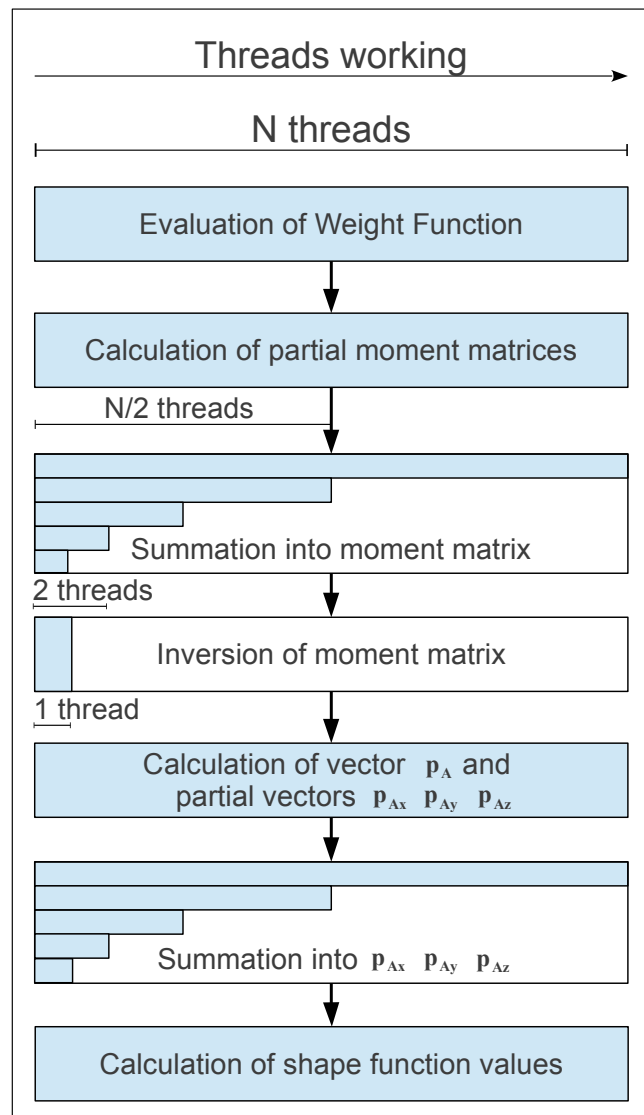


Fig. 8.12: Phase 1 - Concurrency level for the calculation of shape function values in the GPU

8.7.4.2 Phase 2 – Calculation of the global stiffness coefficients

In the second phase, there are also two levels of parallelism, the major one over interacting node pairs and the minor one over Gauss points. The process closely follows the higher-level presentation given in Section 8.4.2.1. A thread block/group is assigned to each node pair and each thread of the block handles one Gauss point at a time. This is schematically shown in Fig. 8.5, where N corresponds to nodes and G corresponds to Gauss points. More details for this phase are given in Section 8.4.2.1.

8.7.5 Performance of the GPU implementation of the interaction-wise approach

Table 8.6 shows the computing times needed for the GPU implementation of interaction-wise variant with individual Gauss points. A GeForce GTX680 with 1536 CUDA cores and 2GB GDDR5 memory is used. For this implementation, kernel 1 corresponds to phase 1 of the GPU implementation, i.e. the calculation of the quadrature points, and kernel 2 corresponds to phase 2, i.e. the calculation of the matrix coefficients.

EFG Example	Time (s)		
	Kernel 1	Kernel 2	Total
2D-1	0.05	0.19	0.2
2D-2	0.13	0.56	0.7
2D-3	0.21	0.89	1.1
3D-1	0.17	2.41	2.6
3D-2	0.32	6.17	6.5
3D-3	0.62	12.31	12.9

Table 8.6: Computing times for the formulation of the EFG stiffness matrix in the GPU implementation of the interaction-wise approach with individual Gauss points.

8.7.6 Numerical results

The individual Gauss point variants of the contribution-wise and interaction-wise approaches are implemented and tested for the computation of the stiffness matrix in 2D and 3D structural mechanics problems for EFG methods. The geometric domains of these problems maximize the number of correlations and consequently the computational cost for the given number of nodes. The test examples are run on the following hardware. CPU: Core i7-980X which has 6 physical cores (12 logical cores) at 3.33 GHz and 12MB cache. GPU: GeForce GTX680 with 1536 CUDA cores and 2GB GDDR5 memory. All floating point operations are in double precision.

Fig. 8.13 shows an overview of the numerical results obtained for EFG, which uses the individual Gauss point variants of the contribution-wise (CW) and interaction-wise (IW) methods, while Table 8.7 shows the speedups of the GPU implementation compared to all CPU implementations. Compared to the CW method with the best sparse matrix builder the speedup of the GPU implementation is almost $60\times$, while with the skyline format the speedup is around $25\times$, but the memory requirements are significantly higher (Table 8.4). Compared to the IW method in the CPU, the speedup of the GPU implementation is about $40\times$.

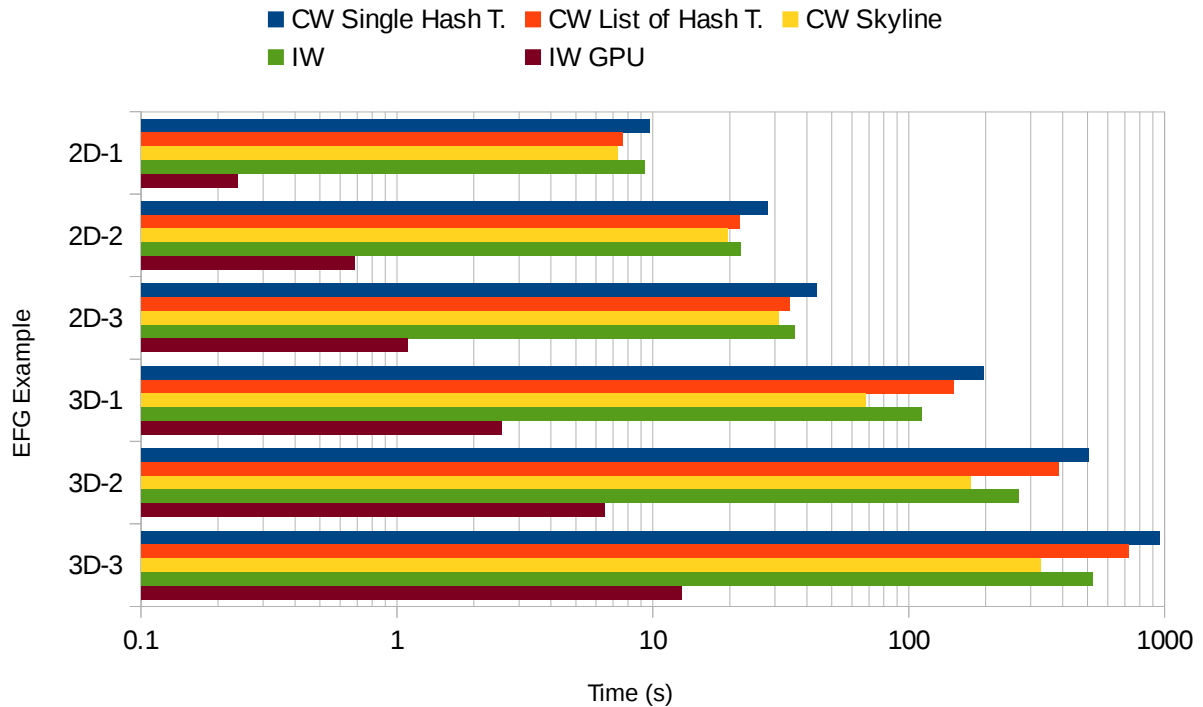


Fig. 8.13: Overview of the numerical results obtained for EFG with the contribution-wise (CW) and interaction-wise (IW) methods (individual Gauss point variants).

EFG Example	CW Single Hash T.	CW List of Hash T.	CW Skyline	IW
2D-1	41	32	31	39
2D-2	41	32	29	32
2D-3	40	31	28	33
3D-1	76	58	26	44
3D-2	78	59	27	41
3D-3	74	56	25	41

Table 8.7: EFG: speedups obtained with the GPU implementation compared to the CPU implementations for the matrix formulation.

In order to get the total time elapsed from the beginning of the simulation until the stiffness matrix has been created, the time required to identify correlations, interactions and synergies is also be taken into account. The best total elapsed time is shown in Table 8.8. In this implementation, the bottleneck is the identification of synergies. The synergy identification is performed in the CPU and

the formulation of the stiffness matrix in the GPU. Therefore, in a hybrid implementation, it is possible to have the CPU producing synergies and the GPU processing them concurrently. A projection of the time required in that case is shown in Table 8.9.

EFG Example	Best achieved time (seconds)				Total
	Correlations	Interactions	Synergies	Formulation	
	CPU parallel	CPU parallel	CPU parallel	GPU	
2D-1	0.5	<0.1	0.4	0.2	1.2
2D-2	1.0	<0.1	1.2	0.7	2.9
2D-3	1.4	<0.1	1.5	1.1	4.0
3D-1	0.9	<0.1	4.8	2.6	8.2
3D-2	1.7	0.2	10.7	6.5	19.1
3D-3	3.3	0.3	22.4	12.9	38.9

Table 8.8: Best achieved elapsed time until the finish of the formulation of the stiffness matrix

EFG Example	Best achieved time (seconds)				Total
	Correlations	Interactions	Synergies	Formulation	
	CPU parallel	CPU parallel	CPU parallel	GPU	
2D-1	0.5	<0.1	0.4		0.9
2D-2	1.0	<0.1	1.2		2.2
2D-3	1.4	<0.1	1.5		2.9
3D-1	0.9	<0.1	5.0		5.9
3D-2	1.7	0.2	11.5		13.4
3D-3	3.3	0.3	24.0		27.6

Table 8.9: Best projected elapsed time until the finish of the formulation of the stiffness matrix in a hybrid implementation

8.8 GPU accelerated formulation of the IGA stiffness matrix

The element variants of the assembly methods are used for the IGA stiffness matrix.

8.8.1 B^TEB Calculation

Similarly to EFG, a fast computation of the matrix product (Section 8.1):

$$\mathbf{Q}_G = \mathbf{B}_G^T \mathbf{E} \mathbf{B}_G \quad (8.15)$$

is important because it is repeated at each integration point. This may not be so critical in FEA compared to the total simulation time, but it is very important in IGA where the number of Gauss points and the number of influenced nodes/control points per Gauss point are both significantly greater. The computations of eq. (8.15) can be broken into smaller operations for each combination of influenced control points i, j belonging to the domain of influence of the Gauss point under consideration:

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j \quad (8.16)$$

Once a submatrix \mathbf{Q}_{ij} is calculated, it is multiplied with the weight factor of the corresponding Gauss point and then added to the appropriate positions of the element's stiffness matrix. The computation related to \mathbf{Q}_{ij} together with the associated entry updates of \mathbf{K} are a significant part of the total effort for the formulation of the global stiffness matrix.

The \mathbf{Q}_{ij} for an isotropic material in 3D elasticity takes the form (same as EFG, but repeated here for completeness):

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j = \begin{bmatrix} \Phi_{i,x} & 0 & 0 & \Phi_{i,y} & 0 & \Phi_{i,z} \\ 0 & \Phi_{i,y} & 0 & \Phi_{i,x} & \Phi_{i,z} & 0 \\ 0 & 0 & \Phi_{i,z} & 0 & \Phi_{i,y} & \Phi_{i,x} \end{bmatrix} \begin{bmatrix} M & \lambda & \lambda \\ \lambda & M & \lambda \\ \lambda & \lambda & M \end{bmatrix} \mu \begin{bmatrix} \Phi_{j,x} & 0 & 0 \\ 0 & \Phi_{j,y} & 0 \\ 0 & 0 & \Phi_{j,z} \\ \Phi_{j,y} & \Phi_{j,x} & 0 \\ 0 & \Phi_{j,z} & \Phi_{j,y} \\ \Phi_{j,z} & 0 & \Phi_{j,x} \end{bmatrix} \mu \quad (8.17)$$

$$\mathbf{Q}_{ij} = \begin{bmatrix} \Phi_{i,x} \Phi_{j,x} M + \Phi_{i,y} \Phi_{j,y} \mu + \Phi_{i,z} \Phi_{j,z} \mu & \Phi_{i,x} \Phi_{j,y} \lambda + \Phi_{i,y} \Phi_{j,x} \mu & \Phi_{i,x} \Phi_{j,z} \lambda + \Phi_{i,z} \Phi_{j,x} \mu \\ \Phi_{i,y} \Phi_{j,x} \lambda + \Phi_{i,x} \Phi_{j,y} \mu & \Phi_{i,y} \Phi_{j,y} M + \Phi_{i,x} \Phi_{j,x} \mu + \Phi_{i,z} \Phi_{j,z} \mu & \Phi_{i,y} \Phi_{j,z} \lambda + \Phi_{i,z} \Phi_{j,y} \mu \\ \Phi_{i,z} \Phi_{j,x} \lambda + \Phi_{i,x} \Phi_{j,z} \mu & \Phi_{i,z} \Phi_{j,y} \lambda + \Phi_{i,y} \Phi_{j,z} \mu & \Phi_{i,z} \Phi_{j,z} M + \Phi_{i,y} \Phi_{j,y} \mu + \Phi_{i,x} \Phi_{j,x} \mu \end{bmatrix}$$

\mathbf{E} and $\mathbf{B}_i / \mathbf{B}_j$ are never formed. Instead three values for \mathbf{E} , the two Lamé parameters λ , μ and the P-Wave modulus $M = 2\mu + \lambda$ and three values for \mathbf{B}_i , specifically $N_{i,x}$, $N_{i,y}$, $N_{i,z}$, are

stored. Since some of the multiplications are repeated, the calculations in eq. (8.17) can be efficiently performed with 30 multiplications and 12 additions. More detailed analysis for the $\mathbf{B}^T \mathbf{E} \mathbf{B}$ calculation expanding to other types of material as well can be found in Appendix A.

8.8.2 Performance of the element-wise variant of the CW method

During the contribution-wise assembly, the global matrix coefficients are continually updated with new contributions. Unlike EFG, however, a lot of the updating work is offloaded to local computations, so the importance of the sparse matrix builder is diminished. It is still a common source of significant slowdown and must be handled appropriately. The implementations of the DOK matrix format (Section 5.6.1.2) used in EFG (Section 8.7.2) are perfectly suitable for the element-wise variant utilized in IGA, since they perform well in the much more intensive Gauss point-wise variant.

Table 8.10 shows relevant metrics of the analyzed IGA examples, while Table 8.11 shows the computing times for assembling IGA stiffness matrices with both variants of the contribution-wise method. The examples are run on a Core i7-980X which has 6 physical cores (12 logical cores) at 3.33GHz and 12MB cache. The examples have no trivial knot spans in order to maximize the number of calculations. All floating point calculations are in double-precision arithmetic. The computing times include the calculation of shape function derivatives and the matrix assembly. Table 8.10 shows the local and global updates required for the element-wise assembly. In the case of the Gauss point-wise assembly, there are no extra global updates because all the local updates of Table 8.10 are global since all contributions are directly added to the global stiffness matrix. Even though the Gauss point-wise variant is applicable as well, Table 8.11 shows that making element-level computations is significantly better.

Table 8.12 compares the element-wise approach for building the stiffness matrix when using sparse and skyline format. Although the skyline format offers very fast indexing times (almost the same as the dense format) it stores an increasingly higher number of stiffness elements, as shown in Table 8.13, and thus imposes restrictions in memory requirements of large-scale simulations. Tables 8.12 and 8.13 show that the employed sparse matrix implementation is very close to the skyline format in terms of the required computing time, while storing a significantly lower amount of stiffness elements.

IGA Example	Total non-zero entries	Total local entry updates	Average local updates/entry	Total global entry updates	Average global updates/entry	Local/Global update ratio
2D-P2-1	2,554,947	76,532,931	30	8,503,659	3.3	9
2D-P2-2	12,690,072	381,678,156	30	42,408,684	3.3	9
2D-P2-3	20,359,251	612,769,779	30	68,085,531	3.3	9
2D-P3-1	4,936,563	416,351,232	84	26,021,952	5.3	16
2D-P3-2	10,030,368	848,931,072	85	53,058,192	5.3	16
2D-P3-3	14,773,776	1,252,204,800	85	78,262,800	5.3	16
2D-P4-1	4,058,400	775,710,000	191	31,028,400	7.6	25
2D-P4-2	8,090,675	1,556,806,875	192	62,272,275	7.7	25
2D-P4-3	12,129,675	2,340,931,875	193	93,637,275	7.7	25
3D-P2-1	3,182,649	440,533,971	138	16,316,073	5.1	27
3D-P2-2	8,606,172	1,239,556,608	144	45,909,504	5.3	27
3D-P2-3	18,142,461	2,671,269,597	147	98,935,911	5.5	27
3D-P3-1	7,982,313	4,857,004,032	608	75,890,688	9.5	64
3D-P3-2	11,085,579	6,915,538,944	624	108,055,296	9.7	64
3D-P3-3	22,134,864	14,427,531,264	652	225,430,176	10.2	64
3D-P4-1	6,849,000	11,729,437,500	1,713	93,835,500	13.7	125
3D-P4-2	10,594,236	19,361,062,500	1,828	154,888,500	14.6	125
3D-P4-3	15,503,568	29,742,187,500	1,918	237,937,500	15.3	125

Table 8.10: Metrics of the IGA 2D and 3D elasticity problems.

IGA Example	Time (seconds)		Speedup ratio
	Gauss p.-wise	Element-wise	
2D-P2-1	6.5	5.1	1.3
2D-P2-2	27	20	1.3
2D-P2-3	53	32	1.7
2D-P3-1	23	14	1.6
2D-P3-2	44	27	1.7
2D-P3-3	81	39	2.0
2D-P4-1	34	19	1.8
2D-P4-2	65	36	1.8
2D-P4-3	97	57	1.7
3D-P2-1	21	7.6	2.7
3D-P2-2	50	21	2.5
3D-P2-3	127	43	3.0
3D-P3-1	201	59	3.4
3D-P3-2	255	83	3.1
3D-P3-3	517	168	3.1
3D-P4-1	415	131	3.2
3D-P4-2	695	218	3.2
3D-P4-3	1,230	333	3.7

Table 8.11: Computing times for the formulation of the IGA stiffness matrix with the Gauss point-wise and element-wise approaches.

IGA Example	dof	Time (seconds)		Ratio
		Sparse	Skyline	
2D-P2-1	101,250	5	4	1.2
2D-P2-2	500,000	20	17	1.2
2D-P2-3	801,378	32	27	1.2
2D-P3-1	101,250	14	13	1.1
2D-P3-2	204,800	27	25	1.1
2D-P3-3	301,088	39	35	1.1
2D-P4-1	51,200	19	18	1.1
2D-P4-2	101,250	36	34	1.1
2D-P4-3	151,250	57	52	1.1
3D-P2-1	20,577	8	8	1.0
3D-P2-2	52,728	21	18	1.1
3D-P2-3	107,811	43	37	1.2
3D-P3-1	20,577	59	54	1.1
3D-P3-2	27,783	83	75	1.1
3D-P3-3	52,728	168	154	1.1
3D-P4-1	10,125	131	128	1.0
3D-P4-2	14,739	218	216	1.0
3D-P4-3	20,577	333	323	1.0

Table 8.12. Single core CPU computing time for the formulation of the stiffness matrix in the element-wise (EW) approach with sparse and skyline storage.

Example	dof	Number of Stored Elements		Ratio
		Skyline	Sparse	
P2-1	101,250	91,071,675	2,554,947	36
P2-2	500,000	999,744,000	12,690,072	79
P2-3	801,378	2,028,680,811	20,359,251	100
P3-1	101,250	136,226,475	4,936,563	28
P3-2	204,800	392,296,720	10,030,368	39
P3-3	301,088	699,568,656	14,773,776	47
P4-1	51,200	64,992,000	4,058,400	16
P4-2	101,250	181,177,875	8,090,675	22
P4-3	151,250	331,150,875	12,129,675	27
P2-1	20,577	43,366,569	3,182,649	14
P2-2	52,728	209,681,004	8,606,172	24
P2-3	107,811	693,626,571	18,142,461	38
P3-1	20,577	63,172,473	7,982,313	8
P3-2	27,783	104,801,445	11,085,579	9
P3-3	52,728	308,053,200	22,134,864	14
P4-1	10,125	24,421,500	6,849,000	4
P4-2	14,739	46,342,884	10,594,236	4
P4-3	20,577	81,740,508	15,503,568	5

Table 8.13. Number of stored stiffness elements for skyline and sparse storage.

8.8.3 Performance of the interaction-wise approach

The final values of each \mathbf{K}_{ij} submatrix are calculated and written once in the corresponding positions of the global stiffness matrix instead of being gradually updated as in the contribution-wise approach. This method does not require lookups, which allows the use of a simpler and more efficient sparse matrix format, like the coordinate list (COO) format (Section 5.6.1.1). A simple implementation based on three arrays, one for row indexes, one for column indexes and one for the value of each non-zero matrix coefficient is sufficient and is easily applied both in the CPU and the GPU. The memory required is less than in the DOK implementations used in the contribution-wise approach, though still proportional to the number of non-zero entries. Note that the interaction-wise method has no indexing time due to its nature, in contrast to the contribution-wise approach.

Table 8.14 shows the computing times required for the interaction-wise variant with elements in the IGA test examples. Our implementation of the interaction-wise approach (Table 8.14) performed better than the contribution-wise approach (Table 8.12). Regardless of the relative performance of the two methods, the most important advantage of the interaction-wise approach is its amenability to parallelism which is discussed in later sections.

Example	dof	Time (seconds)			
		Shape Functions	Assembly	Total	
P2-1	101,250	2	2	4	
P2-2	500,000	10	6	17	
P2-3	801,378	17	10	27	
2D	P3-1	101,250	7	5	12
	P3-2	204,800	13	10	23
	P3-3	301,088	18	16	34
	P4-1	51,200	8	8	16
	P4-2	101,250	15	17	31
	P4-3	151,250	21	26	47
3D	P2-1	20,577	2	4	7
	P2-2	52,728	6	10	17
	P2-3	107,811	12	23	36
	P3-1	20,577	9	33	42
	P3-2	27,783	12	46	58
	P3-3	52,728	24	95	119
	P4-1	10,125	11	73	84
	P4-2	14,739	18	123	141
	P4-3	20,577	27	184	211

Table 8.14: Computing times for the formulation of the IGA stiffness matrix when using the interaction-wise variant with elements.

8.8.4 GPU implementation of the interaction-wise approach

Contrary to the contribution-wise approach, the interaction-wise approach for the formation of the stiffness matrix is well suited for the GPU. There are two phases, as described in Section 8.4, and each one of the two phases is calculated with its own kernel and exhibits different levels of parallelism. The implementations in in this section are written in openCL for greater portability.

8.8.4.1 Phase 1 – Calculation of shape function and derivative values

In the first phase, the shape functions and their derivatives are calculated for all influenced nodes of every Gauss point. All Gauss points of a particular element have the same influenced nodes. There are two levels of parallelism: the major over the elements and the minor over the Gauss points. A thread block/group is assigned to each element and each thread handles one Gauss point at a time and iterates over all influenced nodes. Since all threads iterate over the same number of influenced nodes, there is no thread divergence which would have a negative impact on performance. The thread organization is schematically shown in Fig. 8.4, where N corresponds to control points, E corresponds to elements and G corresponds to Gauss points.

For the most part of this phase all threads of a block are busy (Fig. 8.14). The exceptions are in the first step, which loads B spline values for each axis into the shared/local memory, and in the last step which rearranges the output values, as described in Section 8.5. Since each element has its own thread block, all values related to a particular element are stored in the shared/local memory so they can be accessed efficiently by all threads in the block. In particular, all B-spline values and their derivatives (for all axes) that are relevant to the element assigned to this block are needed by each thread for processing all influenced nodes. The constant memory is used for storing values such as the number of influenced control points per element, the number of influenced Gauss points per element, the number of Gauss points per axis in each element, etc. As a result, many calculations are performed with data found in fast memories which is very beneficial from a performance point of view.

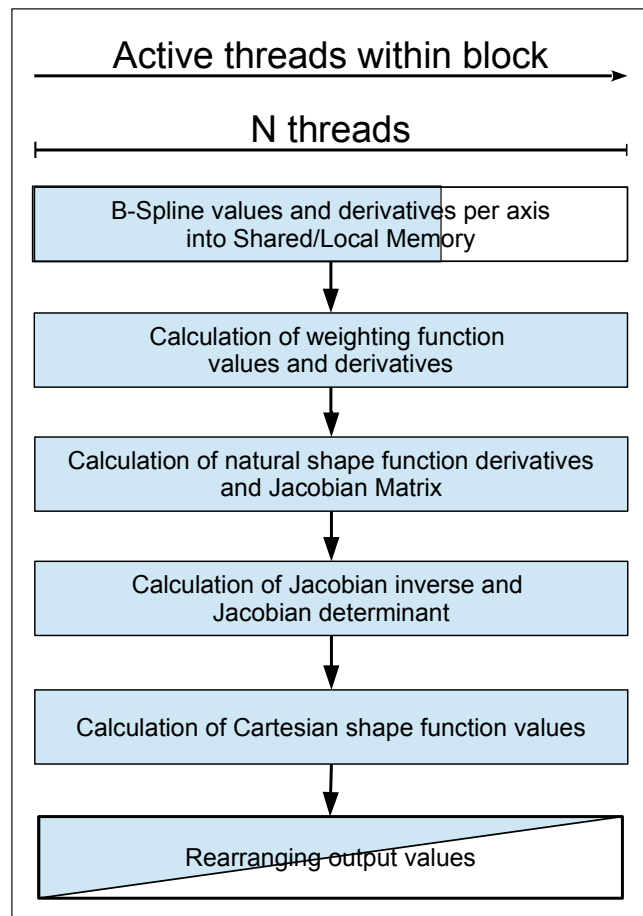


Fig. 8.14. IW approach: Phase 1 - Concurrency within a block/group of threads for the calculation of shape function values in the GPU

8.8.4.2 Phase 2 – Calculation of the global stiffness coefficients

In the second phase, there are also two levels of parallelism, the major one over interacting node pairs and the minor one over Gauss points. The process closely follows the higher-level presentation given in Section 8.4.2.2. A thread block/group is assigned to each node pair and each thread of the block handles one Gauss point at a time. This is schematically shown in Fig. 8.7, where N corresponds to nodes and G corresponds to Gauss points. More details for this phase are given in Section 8.4.2.2.

8.8.5 Performance of the coalesced and non-coalesced GPU implementations of the interaction-wise approach

Tables 8.15 and 8.16 show the computing time needed for the non-coalesced and coalesced GPU implementations, respectively, of the interaction-wise variant with elements. A GeForce GTX680 with 1536 CUDA cores and 2GB GDDR5 memory is used. In the non-coalesced implementation, kernel 1 and kernel 2 correspond to phase 1 and phase 2, respectively, of the process outlined above. However, in the coalesced implementation, kernel 1 is also tasked with rearranging the values in the preferred memory layout, as described in Section 8.5. As a result, it takes slightly longer in the coalesced version in order to rearrange the values, but the benefits on the second phase are clear. The speedup ratio between the two versions is also shown in Table 8.16.

IGA Example	Time (seconds)		
	Kernel 1	Kernel 2	Total
2D-P2-1	0.05	0.08	0.1
2D-P2-2	0.22	0.36	0.6
2D-P2-3	0.34	0.55	0.9
2D-P3-1	0.12	0.28	0.4
2D-P3-2	0.23	0.56	0.8
2D-P3-3	0.34	0.79	1.1
2D-P4-1	0.17	0.48	0.6
2D-P4-2	0.31	0.90	1.2
2D-P4-3	0.44	1.37	1.8
3D-P2-1	0.06	0.20	0.3
3D-P2-2	0.14	0.54	0.7
3D-P2-3	0.30	1.13	1.4
3D-P3-1	0.28	2.77	3.1
3D-P3-2	0.40	3.95	4.3
3D-P3-3	0.77	8.27	9.0
3D-P4-1	0.37	7.52	7.9
3D-P4-2	0.61	12.5	13.1
3D-P4-3	0.88	18.9	19.8

Table 8.15: Computing times for the formulation of the IGA stiffness matrix in the non-coalesced GPU implementation of the interaction-wise approach with elements.

IGA Example	Time (seconds)			Speedup Ratio
	Kernel 1	Kernel 2	Total	
P2-1	0.05	0.08	0.1	1.0
P2-2	0.23	0.34	0.6	1.0
P2-3	0.35	0.53	0.9	1.0
P3-1	0.13	0.18	0.3	1.3
P3-2	0.23	0.35	0.6	1.4
P3-3	0.36	0.51	0.9	1.3
P4-1	0.21	0.20	0.4	1.6
P4-2	0.38	0.38	0.8	1.6
P4-3	0.60	0.54	1.1	1.6
P2-1	0.07	0.10	0.2	1.5
P2-2	0.18	0.25	0.4	1.6
P2-3	0.38	0.51	0.9	1.6
P3-1	0.37	0.56	0.9	3.3
P3-2	0.51	0.78	1.3	3.4
P3-3	1.03	1.61	2.6	3.4
P4-1	0.49	1.07	1.6	5.1
P4-2	0.80	1.79	2.6	5.1
P4-3	1.19	2.68	3.9	5.1

Table 8.16: Computing times for the formulation of the IGA stiffness matrix in the coalesced GPU implementation of the interaction-wise approach with elements.

8.8.6 Numerical results

Fig. 8.15 shows an overview of the numerical results obtained for IGA, which uses element variants of the CW and IW methods. Table 8.17 shows the speedups of both non-coalesced and coalesced GPU implementations compared to the CPU implementations, with the exception of the speedup of the CW variant without elements (i.e. the Gauss point-wise variant) because it is misrepresentative. The coalesced GPU implementation offers speedup up to $85\times$ compared to the CW method and up to $55\times$ compared to the IW method.

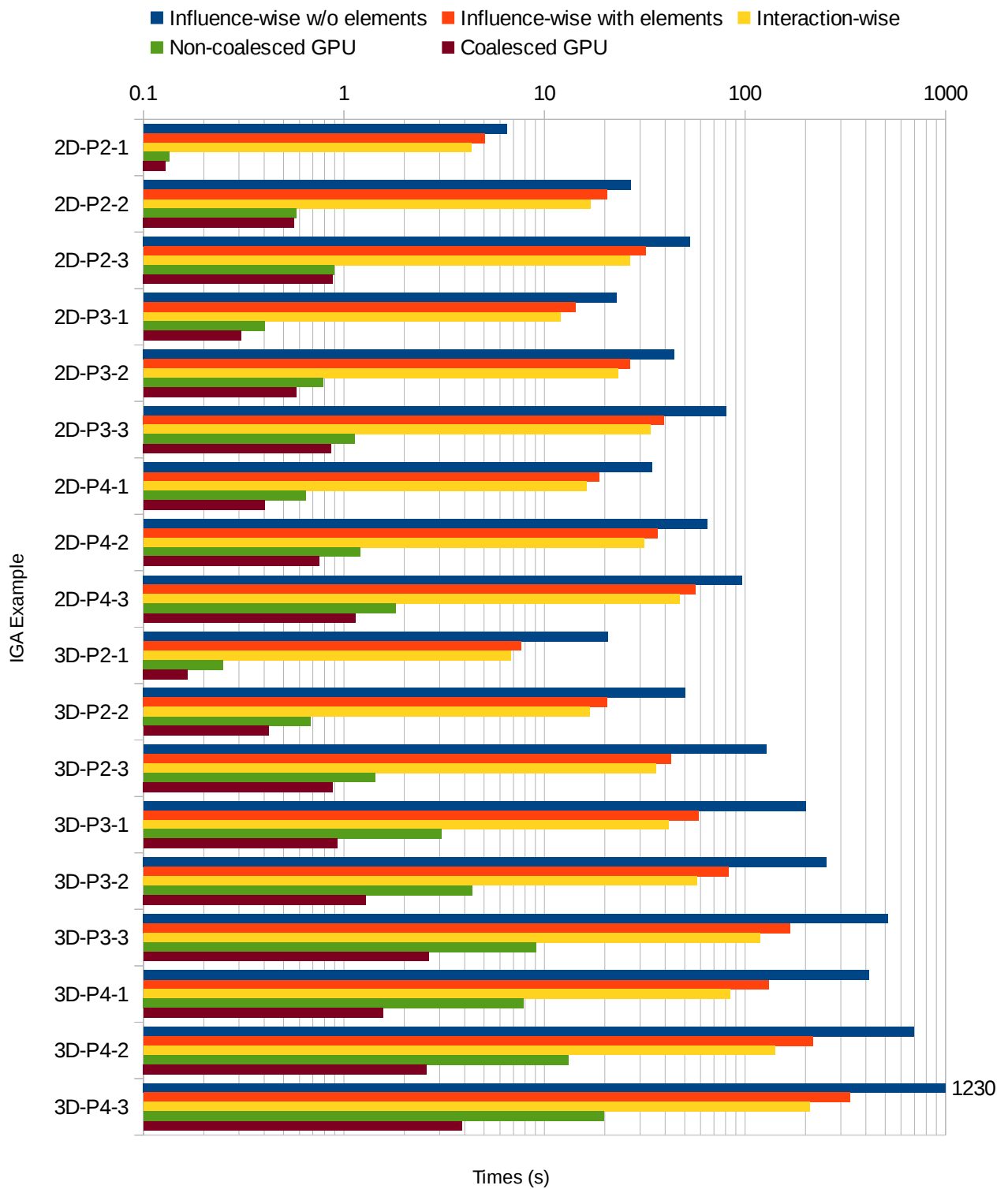


Fig. 8.15: Overview of the numerical results obtained for IGA with the contribution-wise (CW) and interaction-wise (IW) methods (element variants, except “CW w/o elements”).

IGA Example	Non-coalesced		Coalesced	
	CW	IW	CW	IW
2D-P2-1	38	32	39	33
2D-P2-2	35	29	36	30
2D-P2-3	36	30	36	30
2D-P3-1	36	30	46	39
2D-P3-2	34	30	46	40
2D-P3-3	35	30	46	39
2D-P4-1	29	25	47	40
2D-P4-2	30	26	49	42
2D-P4-3	31	26	50	42
3D-P2-1	30	27	46	41
3D-P2-2	30	25	49	40
3D-P2-3	30	25	49	41
3D-P3-1	19	14	63	45
3D-P3-2	19	13	64	45
3D-P3-3	19	13	64	45
3D-P4-1	17	11	84	54
3D-P4-2	17	11	84	54
3D-P4-3	17	11	86	54

Table 8.17: IGA: Speedups obtained with the non-coalesced and coalesced GPU implementations compared to the CPU implementations.

The cost of the EFG examples greatly differs from the cost of the IGA examples due to several reasons. IGA has significantly more Gauss points per dof than EFG, as can be seen in Tables 2.2, 2.3. More importantly, the number of N-G and G-N correlations is higher in IGA than in EFG. Each EFG node is influenced by an average of 100 and 1000 Gauss points in 2D and 3D respectively, whereas each IGA control point is influenced by 256 and 4,096 ($p=3$) or 625 and 15,625 ($p=4$) in 2D and 3D respectively. On the other hand, Gauss points in IGA are handled in groups and not individually, as in MMs. These two competing factors make the cost characteristics very different between the two simulation methods.

The speedups of Table 8.17 are in the same tier within a group of examples, but differ between groups. In the 3D examples, for $p=3$, there are 64 Gauss points in each element, whereas for $p=4$ there are 125 Gauss points in each element. As a result, there are more calculations per memory access for $p=4$ than for $p=3$. Data transferred to the shared/local memory of a thread

block are reused more times. Furthermore, in the second phase of the GPU implementation, each shared element contains more Gauss points to process. There are also more interacting pairs and more shared elements per pair. These reasons make the coalesced GPU implementation perform increasingly better as p increases, but the non-coalesced GPU implementation performs increasingly worse. The latter is because of the memory layout of the non-coalesced implementation (Section 8.5): higher values of p lead to more distant memory accesses while also further preventing the cache to improve the situation by avoiding some of the global memory accesses.

8.9 Remarks

Several numerical simulation methods, among them meshless and isogeometric analysis methods, exhibit expensive matrix computation. This is attributed to the significantly increased number of correlations as well as interactions, caused by extended domains of influence and/or large number of Gauss points required. More interactions also imply more non-zeros in the characteristic matrices leading to increased computational effort for the matrix assembly as well as solution of the resulting algebraic equations. The cost of the matrix assembly per se is significant and must be handled accordingly to enable real-world applications of these numerical simulation methods.

The contribution-wise (CW) approach is the typical approach for assembling matrices with Gauss quadrature. For simulation methods with a large number of contributions to the global matrix, a fine tuned matrix format for the assembly phase can greatly improve the performance properties of the CW approach. Dense or skyline formats feature fast indexing but use considerably more memory than sparse formats and thus can be prohibitive for large scale simulations. Sparse matrix formats have the lowest memory cost but higher indexing cost, so sparse formats specifically tailored for the assembly phase are used in this work.

The interaction-wise (IW) method has several advantages with respect to the CW approach. The most important one is its amenability to parallelism especially in massively parallel systems like the GPUs. Each interacting pair can be processed separately by any available processor in order to compute the corresponding submatrix. The IW approach is characterized as “embarrassingly parallel” since it requires no synchronization whatsoever between pairs.

A GPU implementation is applied to the IW approach offering great speedups compared to CPU implementations. The interacting pairs keep the GPU constantly busy with calculations resulting in

high hardware utilization which is evidenced by the high speedup ratios of approximately two orders of magnitude in the test examples presented with a single GPU. The IW approach offers great portability since it can be applied as is to any available hardware achieving even lower computing times when combined with many GPUs, hybrid CPU(s)/GPU(s) implementations and generally any available processing unit. The importance of this flexibility becomes apparent when considering contemporary and future developments like heterogeneous computing systems architecture.

In conclusion, the parametric tests performed in the framework of this study showed that with the proposed implementation along with the exploitation of currently available low cost hardware, the expensive formulation of Gauss quadrature-based matrices can be reduced by orders of magnitude. The presented interaction-wise approach enables the efficient utilization of any available hardware and in conjunction with fast initialization and its inherently parallelization features can accomplish high speedup ratios, which convincingly addresses the main shortcoming of simulation methods like MMs and IGA, which is the computational cost for the assembly characteristic matrices and making them computationally competitive in solving large-scale problems.

9 Overview and concluding remarks

This work explores massively parallel computer implementations for the most widely used simulation method, namely the finite element method (FEM/FEA), as well as meshless methods (MMs) and isogeometric analysis methods which have recently attracted a lot of interest from the scientific community. The main drawback of MMs and IGA when addressing real-world problems is the significantly increased cost for the formulation of the characteristic matrices. This is attributed to the excessive number of correlations as well as interactions, caused by extended domains of influence and/or large number of Gauss points required for the numerical integration. More interactions also imply more non-zeros in the characteristic matrices leading to increased computational effort for the matrix assembly as well as for the solution of the resulting algebraic equations. The cost of the matrix formulation is significant and must be handled accordingly to enable real-world applications of these numerical simulation methods. Therefore, in order to make them affordable in large-scale simulations, these methods require massively parallel algorithms not only for the solution phase of the algebraic equations but also for the assembly phase of the characteristic matrices.

Domain decomposition methods are used for the solution phase because they allow the exploitation of the natural parallelism offered by the subdivision of the physical domains to a number of subdomains. The primal domain decomposition method is briefly described followed by an extensive presentation of the dual domain decomposition method (DDM/FETI), which is used in this work. The basic ingredients of FETI are presented, including floating subdomains which constitute a particular characteristic of the method. The solution of the linear equations of the FETI interface problem is discussed along with preconditioners and implementation considerations.

Graphics processing units (GPUs) and their characteristic properties are thoroughly presented. In a massively parallel context, GPUs are particularly interesting. This is due to their low cost, low energy consumption and high performance. GPUs are parallel devices of the SIMD (single instruction, multiple data) classification and require a large number of threads to be effectively utilized (thousand, usually more). As a result, the principles of massively parallel programming directly apply to GPUs. GPU technology has matured considerably in the last years and is currently improving at a very fast pace. The implementation details presented need to be considered in any GPU implementation and are thus vital for the efficiency of the simulation methods considered in

this work.

The matrices involved in the simulation phase need to be handled appropriately. Matrix storage and matrix operations are important performance factors for large-scale simulations. The choice of an appropriate format for the task at hand may significantly affect performance. Matrix formats that are commonly used in simulations are presented along with appropriate storage schemes and implementation considerations. These include dense, banded, skyline formats as well as a variety of sparse formats.

The implementation of the FETI domain decomposition method in a hybrid CPU-GPU environment is presented along with supporting numerical results. The solution of the subdomain problems was performed both with a direct Cholesky solver as well as with an iterative PCG solver. The FETI version with the direct Cholesky solver performed better than the PCG solver in the tests considered and with different workstation configurations. This is attributed to the dynamic load balancing implementation of the factorization and forward backward substitution tasks. However, the performance improvement with the faster GPU is more pronounced with the PCG solver than with the Cholesky solver as a result of faster GPU execution of sparse matrix-vector multiplications (SpMV) which dominate the performance of PCG. Fine-grained subdivisions gave much better results than coarse-grained subdivisions particularly for the Cholesky solver of the subdomain problems. This is due to the high cost of the factorization for large subdomains in connection to the numerical scalability of FETI, where the convergence of the PCPG algorithm is not sensitive to the size of the interface problem. The performance of FETI with the PCG solver is less susceptible to the number of subdomains since it is dominated by the SpMV which are performed with the same efficiency by the workstation components irrespective of the size of the matrices and the corresponding vectors.

An important aspect of the hybrid FETI implementation is the dynamic load-balancing. The dynamic load balancing with task parallelism and the parallel implementation of the SpMV multiplications and dot products ensure that all components of the workstation are constantly busy with calculations resulting in full exploitation of their computing resources. This is evidenced by the high speedup ratios achieved in the test example for all hardware combinations and different number of subdomains. The dynamic load balancing allows the efficient utilization of different CPUs and GPUs as well as any number of CPU cores or GPUs, while making sure that all components are used to their full capacity.

Relations between the basic entities (nodes, Gauss points, control points) are needed for quadrature-based simulation methods and need to be identified at the start of the simulation. Relations are presented in an abstract manner to cover all quadrature-based methods before being applied to the specific simulation methods used in this work (FEA, MMs, IGA). The domain of influence and its particular characteristics is also extensively discussed for each simulation method. The domain of influence is a fundamental factor that dictates the density and cost of the characteristic matrices of each method. A cost comparison of MMs and IGA with FEM is included to further highlight the differences and challenges between the methods. Generic techniques are presented for the identification of the relations. Furthermore, in the case of MMs where the identification is quite laborious, efficient algorithms are presented to improve the cost of identification.

The formulation of the characteristic matrices in MMs and IGA take considerable time and thus needs to be appropriately handled. Two primary formulation methods are presented: the standard contribution-wise (CW) method and the parallel-friendly interaction-wise (IW) method. Both methods have two variants, one that handles Gauss points explicitly and one that handles Gauss points as part of elements (or other groups). The CW approach is the typical approach for assembling matrices with Gauss quadrature. For simulation methods with a large number of contributions to the global matrix, a fine tuned matrix format for the assembly phase can greatly improve the performance properties of the CW approach. Dense or skyline formats feature fast indexing but use considerably more memory than sparse formats and thus can be prohibitive for large-scale simulations. Sparse matrix formats have the lowest memory cost but higher indexing cost, so sparse formats specifically tailored for the assembly phase are used in this work. The IW method has several advantages with respect to the CW approach. The most important one is its amenability to parallelism especially in massively parallel systems like the GPUs. Each interacting pair can be processed separately by any available processor in order to compute the corresponding submatrix.

GPU implementations are applied to the IW approach offering great speedups compared to CPU implementations. The interacting pairs keep the GPU constantly busy with calculations resulting in high hardware utilization which is evidenced by the high speedup ratios of approximately two orders of magnitude in the test examples presented with a single GPU. The IW approach offers great portability since it can be applied to any available hardware achieving even lower computing

times when combined with many GPUs, hybrid CPU(s)/GPU(s) implementations and generally any available processing unit. The importance of this flexibility becomes apparent when considering contemporary and future developments like heterogeneous computing systems architecture.

9.1 Future work

Below are some interesting extensions and future considerations of the present work:

- GPU implementation of the solution of the resulting algebraic equations in MMs and IGA. The main principles are similar to those outlined for FEA, but there are additional challenges that should be addressed as a result of the overlapping subdomains which is a direct consequence of their large domains of influence.
- Hybrid implementation of the solution of MMs and IGA.
- Multi-GPU implementation of formulation and solution. This is important when taking into consideration that each workstation may contain more than 1 GPU (even “home” workstations can easily include up to 4).
- Multi-workstation implementation. Required for large-scale simulation and enabled by clusters and cloud computing.
- Implementation in new architectures and types of processors like Accelerated Processing Units (APUs), “Phi” co-processors etc.

10 Appendix A: BEB calculations

10.1 The elasticity tensor in 3D problems

The elasticity tensor or stiffness tensor is the tensor expression of Hooke's law [97].

$$\boldsymbol{\sigma} = \mathbf{E} \boldsymbol{\varepsilon} \quad (10.1)$$

Due to the symmetry of the stress tensor, strain tensor, and stiffness tensor, only 21 elastic coefficients are independent. The matrix form of \mathbf{E} is always symmetric.

10.1.1 Anisotropic material

An anisotropic material has 21 independent elastic coefficients and its matrix representation is:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \begin{bmatrix} E_{xx \cdot xx} & E_{xx \cdot yy} & E_{xx \cdot zz} & E_{xx \cdot xy} & E_{xx \cdot yz} & E_{xx \cdot zx} \\ & E_{yy \cdot yy} & E_{yy \cdot zz} & E_{yy \cdot xy} & E_{yy \cdot yz} & E_{yy \cdot zx} \\ & & E_{zz \cdot zz} & E_{zz \cdot xy} & E_{zz \cdot yz} & E_{zz \cdot zx} \\ & & & E_{xy \cdot xy} & E_{xy \cdot yz} & E_{xy \cdot zx} \\ & \text{symm} & & & E_{yz \cdot yz} & E_{yz \cdot zx} \\ & & & & & E_{zx \cdot zx} \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{zx} \end{bmatrix} \quad (10.2)$$

10.1.2 Orthotropic material

An orthotropic material has 9 independent elastic coefficients and its matrix representation is:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \begin{bmatrix} E_{xx \cdot xx} & E_{xx \cdot yy} & E_{xx \cdot zz} \\ & E_{yy \cdot yy} & E_{yy \cdot zz} \\ & & E_{zz \cdot zz} \\ & & & E_{xy \cdot xy} \\ & \text{symm} & & & E_{yz \cdot yz} \\ & & & & & E_{zx \cdot zx} \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{zx} \end{bmatrix} \quad (10.3)$$

10.1.3 Isotropic material

An isotropic material has 2 independent elastic coefficients and its matrix representation with Lamé parameters is:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \begin{bmatrix} 2\mu + \lambda & \lambda & \lambda \\ \lambda & 2\mu + \lambda & \lambda \\ \lambda & \lambda & 2\mu + \lambda \\ & & & \mu \\ & & & & \mu \\ & & & & & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{zx} \end{bmatrix} \quad (10.4)$$

Only 2 values need to be saved in place of the whole matrix. If 3 values are stored, recalculations can be avoided:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \begin{bmatrix} M & \lambda & \lambda \\ \lambda & M & \lambda \\ \lambda & \lambda & M \\ & & & \mu \\ & & & & \mu \\ & & & & & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{yz} \\ 2\varepsilon_{zx} \end{bmatrix} \quad (10.5)$$

where M is the P-wave modulus which is equal to $M = 2\mu + \lambda$. In 6 separate equations, eq. (10.5) is the familiar:

$$\begin{aligned} \sigma_{xx} &= M \varepsilon_{xx} + \lambda \varepsilon_{yy} + \lambda \varepsilon_{zz} & \sigma_{xy} &= 2\mu \varepsilon_{xy} \\ \sigma_{yy} &= \lambda \varepsilon_{xx} + M \varepsilon_{yy} + \lambda \varepsilon_{zz} & \sigma_{yz} &= 2\mu \varepsilon_{yz} \\ \sigma_{zz} &= \lambda \varepsilon_{xx} + \lambda \varepsilon_{yy} + M \varepsilon_{zz} & \sigma_{zx} &= 2\mu \varepsilon_{zx} \end{aligned} \quad (10.6)$$

10.2 The deformation matrix in 3D problems

The deformation matrix \mathbf{B} handles the conversion between strains and nodal displacements:

$$\underset{(6 \times 1)}{\boldsymbol{\varepsilon}} = \underset{(6 \times 3n)}{\mathbf{B}} \underset{(3n \times 1)}{\mathbf{d}} \quad (10.7)$$

\mathbf{B} can be broken down to sub-matrices, each one corresponding to one node:

$$\underset{(6 \times 3n)}{\mathbf{B}} = \left[\underset{(6 \times 3)}{\mathbf{B}_1} \quad \underset{(6 \times 3)}{\mathbf{B}_2} \quad \dots \quad \underset{(6 \times 3)}{\mathbf{B}_n} \right] \quad (10.8)$$

Each sub-matrix \mathbf{B}_i in eq. (10.8) has size 6×3 , but only has 3 distinct values: $N_{i,x}$, $N_{i,y}$, $N_{i,z}$:

$$\underset{\substack{\mathbf{B}_i \\ (6 \times 3)}}{=} \begin{bmatrix} N_{i,x} & 0 & 0 \\ 0 & N_{i,y} & 0 \\ 0 & 0 & N_{i,z} \\ N_{i,y} & N_{i,x} & 0 \\ 0 & N_{i,z} & N_{i,y} \\ N_{i,z} & 0 & N_{i,x} \end{bmatrix} \quad (10.9)$$

Instead of storing a 6×3 matrix, these 3 values can be stored and are sufficient to fully represent the underlying matrix. To store all info the whole matrix \mathbf{B} contains, 3 values per node are required, hence $3n$ values instead of a $6 \times 3n$ matrix. Note that \mathbf{B} never need to actually be formed.

10.3 Explicit calculation in 3D problems

The fast computation of the matrix product:

$$\underset{(3n \times 6)}{\mathbf{B}^T} \underset{(6 \times 6)}{\mathbf{E}} \underset{(6 \times 3n)}{\mathbf{B}} \quad (10.10)$$

is important because it is repeated at each integration point. This is important for finite elements but is even more important for EFG and IGA methods where the number of Gauss points and the number of influenced nodes per Gauss point are both significantly greater.

The operation can be broken down to smaller node pair operations:

$$\begin{bmatrix} \mathbf{B}_1^T \\ \mathbf{B}_2^T \\ \vdots \\ \mathbf{B}_n^T \end{bmatrix}_{(3 \times 6)} \underset{(6 \times 6)}{\mathbf{E}} \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \dots & \mathbf{B}_n \end{bmatrix}_{(6 \times 3)} = \begin{bmatrix} \mathbf{B}_1^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_1^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_1^T \mathbf{E} \mathbf{B}_n \\ \mathbf{B}_2^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_2^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_2^T \mathbf{E} \mathbf{B}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_n^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_n^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_n^T \mathbf{E} \mathbf{B}_n \end{bmatrix} \quad (10.11)$$

For each pair of nodes i, j . we need to calculate the product:

$$\underset{(3 \times 3)}{\mathbf{Q}_{ij}} = \underset{(3 \times 6)}{\mathbf{B}_i^T} \underset{(6 \times 6)}{\mathbf{E}} \underset{(6 \times 3)}{\mathbf{B}_j} = \underset{(3 \times 3)}{\mathbf{Q}_{ji}} \quad (10.12)$$

Note that, with the exception of $i=j$, \mathbf{Q}_{ij} is generally not symmetric.

The product $\mathbf{B}^T \mathbf{E} \mathbf{B}$ need not be calculated as a whole. Its parts \mathbf{Q}_{ij} can be produced and consumed immediately (the exact way differs depending on the assembly method used). Computation of \mathbf{Q}_{ij} plus associated indexing to access the \mathbf{K} entries, dominates the total effort [96]. The computation is split in two steps:

$$\begin{array}{ll} \text{Step 1: } \underset{(6 \times 3)}{\mathbf{C}_j} = \underset{(6 \times 6)}{\mathbf{E}} \underset{(6 \times 3)}{\mathbf{B}_j} & \text{Step 2: } \underset{(3 \times 3)}{\mathbf{Q}_{ij}} = \underset{(3 \times 6)}{\mathbf{B}_i^T} \underset{(6 \times 3)}{\mathbf{C}_j} \end{array} \quad (10.13)$$

10.3.1 Anisotropic material

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} & \begin{matrix} E_{xx,xx} & E_{xx,yy} & E_{xx,zz} & E_{xx,xy} & E_{xx,yz} & E_{xx,zx} \end{matrix} \\ \begin{matrix} (6 \times 3) & (6 \times 6) & (6 \times 3) \end{matrix} & \begin{matrix} E_{xx,xx} & E_{xx,yy} & E_{xx,zz} & E_{xx,xy} & E_{xx,yz} & E_{xx,zx} \\ E_{yy,yy} & E_{yy,xy} & E_{yy,zy} & E_{yy,xy} & E_{yy,yz} & E_{yy,zx} \\ & E_{zz,zz} & E_{zz,xy} & E_{zz,yz} & E_{zz,zx} & \\ & & E_{xy,xy} & E_{xy,yz} & E_{xy,zx} & \\ & & & E_{yz,yz} & E_{yz,zx} & \\ & & & & E_{zx,zx} & \end{matrix} \end{matrix} \begin{bmatrix} N_{j,x} & 0 & 0 \\ 0 & N_{j,y} & 0 \\ 0 & 0 & N_{j,z} \\ N_{j,y} & N_{j,x} & 0 \\ 0 & N_{j,z} & N_{j,y} \\ N_{j,z} & 0 & N_{j,x} \end{bmatrix}$$

(column-wise pattern)

(10.14)

$$\begin{bmatrix} N_{j,x} E_{xx,xx} + N_{j,y} E_{xx,xy} + N_{j,z} E_{xx,xz} & N_{j,y} E_{xx,yy} + N_{j,x} E_{xx,xy} + N_{j,z} E_{xx,yz} & N_{j,z} E_{xx,zz} + N_{j,y} E_{xx,yz} + N_{j,x} E_{xx,zx} \\ N_{j,x} E_{xx,yy} + N_{j,y} E_{yy,xy} + N_{j,z} E_{yy,xz} & N_{j,y} E_{yy,yy} + N_{j,x} E_{yy,xy} + N_{j,z} E_{yy,yz} & N_{j,z} E_{yy,zz} + N_{j,y} E_{yy,yz} + N_{j,x} E_{yy,zx} \\ N_{j,x} E_{xx,zz} + N_{j,y} E_{zz,xy} + N_{j,z} E_{zz,xz} & N_{j,y} E_{yy,zz} + N_{j,x} E_{zz,xy} + N_{j,z} E_{zz,yz} & N_{j,z} E_{zz,zz} + N_{j,y} E_{zz,yz} + N_{j,x} E_{zz,zx} \\ N_{j,x} E_{xx,xy} + N_{j,y} E_{xy,xy} + N_{j,z} E_{xy,xz} & N_{j,y} E_{yy,xy} + N_{j,x} E_{xy,xy} + N_{j,z} E_{xy,yz} & N_{j,z} E_{zz,xy} + N_{j,y} E_{xy,yz} + N_{j,x} E_{xy,zx} \\ N_{j,x} E_{xx,yz} + N_{j,y} E_{xy,yz} + N_{j,z} E_{yz,xz} & N_{j,y} E_{yy,yz} + N_{j,x} E_{xy,yz} + N_{j,z} E_{yz,yz} & N_{j,z} E_{zz,yz} + N_{j,y} E_{yz,yz} + N_{j,x} E_{yz,zx} \\ N_{j,x} E_{xx,zx} + N_{j,y} E_{xy,zx} + N_{j,z} E_{xz,zx} & N_{j,y} E_{yy,zx} + N_{j,x} E_{xy,zx} + N_{j,z} E_{yz,zx} & N_{j,z} E_{zz,zx} + N_{j,y} E_{yz,zx} + N_{j,x} E_{zx,zx} \end{bmatrix}$$

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j \begin{matrix} \begin{matrix} N_{i,x} & 0 & 0 & N_{i,y} & 0 & N_{i,z} \\ 0 & N_{i,y} & 0 & N_{i,x} & N_{i,z} & 0 \\ 0 & 0 & N_{i,z} & 0 & N_{i,y} & N_{i,x} \end{matrix} \\ \begin{matrix} (3 \times 3) & (3 \times 6) & (6 \times 3) \end{matrix} \end{matrix} \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \\ C_{41} & C_{42} & C_{43} \\ C_{51} & C_{52} & C_{53} \\ C_{61} & C_{62} & C_{63} \end{bmatrix}$$

(10.15)

$$\mathbf{Q}_{ij} = \begin{matrix} \begin{matrix} N_{i,x} C_{11} + N_{i,y} C_{41} + N_{i,z} C_{61} & N_{i,x} C_{12} + N_{i,y} C_{42} + N_{i,z} C_{62} & N_{i,x} C_{13} + N_{i,y} C_{43} + N_{i,z} C_{63} \\ N_{i,y} C_{21} + N_{i,x} C_{41} + N_{i,z} C_{51} & N_{i,y} C_{22} + N_{i,x} C_{42} + N_{i,z} C_{52} & N_{i,y} C_{23} + N_{i,x} C_{43} + N_{i,z} C_{53} \\ N_{i,z} C_{31} + N_{i,y} C_{51} + N_{i,x} C_{61} & N_{i,z} C_{32} + N_{i,y} C_{52} + N_{i,x} C_{62} & N_{i,z} C_{33} + N_{i,y} C_{53} + N_{i,x} C_{63} \end{matrix} \\ \begin{matrix} (3 \times 3) \end{matrix} \end{matrix}$$

The number of calculations required is shown in Table 10.1.

10.3.2 Orthotropic material

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} (6 \times 3) & (6 \times 6) & (6 \times 3) \end{matrix} \begin{bmatrix} E_{xx,xx} & E_{xx,yy} & E_{xx,zz} & & & \\ & E_{yy,yy} & E_{yy,zz} & & & \\ & & E_{zz,zz} & & & \\ & & & E_{xy,xy} & & \\ & & & & E_{yz,yz} & \\ & & & & & E_{zx,zx} \end{bmatrix} \begin{bmatrix} N_{j,x} & 0 & 0 \\ 0 & N_{j,y} & 0 \\ 0 & 0 & N_{j,z} \\ N_{j,y} & N_{j,x} & 0 \\ 0 & N_{j,z} & N_{j,y} \\ N_{j,z} & 0 & N_{j,x} \end{bmatrix} \quad (10.16)$$

$$\mathbf{C}_j = \begin{matrix} (6 \times 3) \end{matrix} \begin{bmatrix} N_{j,x} E_{xx,xx} & N_{j,y} E_{xx,yy} & N_{j,z} E_{xx,zz} \\ N_{j,x} E_{xx,yy} & N_{j,y} E_{yy,yy} & N_{j,z} E_{yy,zz} \\ N_{j,x} E_{xx,zz} & N_{j,y} E_{yy,zz} & N_{j,z} E_{zz,zz} \\ N_{j,y} E_{xy,xy} & N_{j,x} E_{xy,xy} & 0 \\ 0 & N_{j,z} E_{yz,yz} & N_{j,y} E_{yz,yz} \\ N_{j,z} E_{zx,zx} & 0 & N_{j,x} E_{zx,zx} \end{bmatrix}$$

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j = \begin{matrix} (3 \times 3) & (3 \times 6) & (6 \times 3) \end{matrix} \begin{bmatrix} N_{i,x} & 0 & 0 & N_{i,y} & 0 & N_{i,z} \\ 0 & N_{i,y} & 0 & N_{i,x} & N_{i,z} & 0 \\ 0 & 0 & N_{i,z} & 0 & N_{i,y} & N_{i,x} \end{bmatrix} \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \\ C_{41} & C_{42} & 0 \\ 0 & C_{52} & C_{53} \\ C_{61} & 0 & C_{63} \end{bmatrix} \quad (10.17)$$

$$\mathbf{Q}_{ij} = \begin{matrix} (3 \times 3) \end{matrix} \begin{bmatrix} N_{i,x} C_{11} + N_{i,y} C_{41} + N_{i,z} C_{61} & N_{i,x} C_{12} + N_{i,y} C_{42} & N_{i,x} C_{13} + N_{i,z} C_{63} \\ N_{i,y} C_{21} + N_{i,x} C_{41} & N_{i,y} C_{22} + N_{i,x} C_{42} + N_{i,z} C_{52} & N_{i,y} C_{23} + N_{i,z} C_{53} \\ N_{i,z} C_{31} + N_{i,x} C_{61} & N_{i,z} C_{32} + N_{i,y} C_{52} & N_{i,z} C_{33} + N_{i,y} C_{53} + N_{i,x} C_{63} \end{bmatrix}$$

The number of calculations required is shown in Table 10.1.

10.3.3 Isotropic material

Eq (10.16) can be simplified:

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} \begin{matrix} M & \lambda & \lambda \\ \lambda & M & \lambda \\ \lambda & \lambda & M \end{matrix} \\ (6 \times 3) \end{matrix} \mu \begin{matrix} \begin{matrix} N_{j,x} & 0 & 0 \\ 0 & N_{j,y} & 0 \\ 0 & 0 & N_{j,z} \end{matrix} \\ \mu \end{matrix} = \begin{matrix} \begin{matrix} N_{j,x} M & N_{j,y} \lambda & N_{j,z} \lambda \\ N_{j,x} \lambda & N_{j,y} M & N_{j,z} \lambda \\ N_{j,x} \lambda & N_{j,y} \lambda & N_{j,z} M \\ N_{j,y} \mu & N_{j,x} \mu & 0 \\ 0 & N_{j,z} \mu & N_{j,y} \mu \\ N_{j,z} \mu & 0 & N_{j,x} \mu \end{matrix} \\ (6 \times 3) \end{matrix} \quad (10.18)$$

Some calculations are repeated. We multiply M , λ , μ with each of the 3 derivatives $N_{i,x}$, $N_{i,y}$, $N_{i,z}$. Therefore we only need 9 computations, specifically:

$$\begin{aligned} M_x &= N_{j,x} M & M_y &= N_{j,y} M & M_z &= N_{j,z} M \\ \lambda_x &= N_{j,x} \lambda & \lambda_y &= N_{j,y} \lambda & \lambda_z &= N_{j,z} \lambda \\ \mu_x &= N_{j,x} \mu & \mu_y &= N_{j,y} \mu & \mu_z &= N_{j,z} \mu \end{aligned} \quad (10.19)$$

Substituting (10.19) in (10.18):

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} \begin{matrix} M_x & \lambda_y & \lambda_z \\ \lambda_x & M_y & \lambda_z \\ \lambda_x & \lambda_y & M_z \\ \mu_y & \mu_x & 0 \\ 0 & \mu_z & \mu_y \\ \mu_z & 0 & \mu_x \end{matrix} \\ (6 \times 3) \end{matrix} \quad (10.20)$$

As for the second step:

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j = \begin{matrix} \begin{matrix} N_{i,x} & 0 & 0 & N_{i,y} & 0 & N_{i,z} \\ 0 & N_{i,y} & 0 & N_{i,x} & N_{i,z} & 0 \\ 0 & 0 & N_{i,z} & 0 & N_{i,y} & N_{i,x} \end{matrix} \\ (3 \times 3) \end{matrix} \begin{matrix} \begin{matrix} M_x & \lambda_y & \lambda_z \\ \lambda_x & M_y & \lambda_z \\ \lambda_x & \lambda_y & M_z \\ \mu_y & \mu_x & 0 \\ 0 & \mu_z & \mu_y \\ \mu_z & 0 & \mu_x \end{matrix} \\ (6 \times 3) \end{matrix} \quad (10.21)$$

$$\mathbf{Q}_{ij} = \begin{matrix} \begin{matrix} N_{i,x} M_x + N_{i,y} \mu_y + N_{i,z} \mu_z & N_{i,x} \lambda_y + N_{i,y} \mu_x & N_{i,x} \lambda_z + N_{i,z} \mu_x \\ N_{i,y} \lambda_x + N_{i,x} \mu_y & N_{i,y} M_y + N_{i,x} \mu_x + N_{i,z} \mu_z & N_{i,y} \lambda_z + N_{i,z} \mu_y \\ N_{i,z} \lambda_x + N_{i,x} \mu_z & N_{i,z} \lambda_y + N_{i,y} \mu_z & N_{i,z} M_z + N_{i,y} \mu_y + N_{i,x} \mu_x \end{matrix} \\ (3 \times 3) \end{matrix}$$

This calculation can be performed in as low as few as 27 multiplications, if we take the few repetitions into account or 30 if we don't (preferable for the GPU because we save a few variables).

We also need 12 additions.

We have the following calculations:

$$\begin{array}{lll}
\times 1 & N_{i,x} M_x & N_{i,y} M_y & N_{i,z} M_z \\
\times 1 & N_{i,x} \lambda_y, N_{i,x} \lambda_z & N_{i,y} \lambda_x, N_{i,y} \lambda_z & N_{i,z} \lambda_x, N_{i,z} \lambda_y \\
\times 2 & N_{i,x} \mu_x & N_{i,y} \mu_y & N_{i,z} \mu_z \\
\times 1 & N_{i,x} \mu_y, N_{i,x} \mu_z & N_{i,y} \mu_x, N_{i,y} \mu_z & N_{i,z} \mu_x, N_{i,z} \mu_y
\end{array} \quad (10.22)$$

For these calculations we need 6 multiplications for each axis for a total of 18 multiplications. The number of calculations is shown in Table 10.1 for the case of recalculating and not recalculating repeated values.

An alternative for the isotropic would be to utilize the repetition of derivative products. Expanding eq. (10.22):

$$\begin{array}{lll}
N_{i,x} N_{j,x} M & N_{i,y} N_{j,y} M & N_{i,z} N_{j,z} M \\
N_{i,x} N_{j,y} \lambda, N_{i,x} N_{j,z} \lambda & N_{i,y} N_{j,x} \lambda, N_{i,y} N_{j,z} \lambda & N_{i,z} N_{j,x} \lambda, N_{i,z} N_{j,y} \lambda \\
N_{i,x} N_{j,x} \mu & N_{i,y} N_{j,y} \mu & N_{i,z} N_{j,z} \mu \\
N_{i,x} N_{j,y} \mu, N_{i,x} N_{j,z} \mu & N_{i,y} N_{j,x} \mu, N_{i,y} N_{j,z} \mu & N_{i,z} N_{j,x} \mu, N_{i,z} N_{j,y} \mu
\end{array} \quad (10.23)$$

$$\mathbf{Q}_{ij}^{(3 \times 3)} = \begin{bmatrix} N_{i,x} N_{j,x} M + N_{i,y} N_{j,y} \mu + N_{i,z} N_{j,z} \mu & N_{i,x} N_{j,y} \lambda + N_{i,y} N_{j,x} \mu & N_{i,x} N_{j,z} \lambda + N_{i,z} N_{j,x} \mu \\ N_{i,y} N_{j,x} \lambda + N_{i,x} N_{j,y} \mu & N_{i,y} N_{j,y} M + N_{i,x} N_{j,x} \mu + N_{i,z} N_{j,z} \mu & N_{i,y} N_{j,z} \lambda + N_{i,z} N_{j,y} \mu \\ N_{i,z} N_{j,x} \lambda + N_{i,x} N_{j,z} \mu & N_{i,z} N_{j,y} \lambda + N_{i,y} N_{j,z} \mu & N_{i,z} N_{j,z} M + N_{i,y} N_{j,y} \mu + N_{i,x} N_{j,x} \mu \end{bmatrix} \quad (10.24)$$

We now calculate all $N_{ab} = N_{i,a} N_{j,b}$, $a, b = x, y, z$:

$$\begin{array}{lll}
N_{xx} M & N_{yy} M & N_{zz} M \\
N_{xy} \lambda, N_{xz} \lambda & N_{yx} \lambda, N_{yz} \lambda & N_{zx} \lambda, N_{zy} \lambda \\
N_{xx} \mu & N_{yy} \mu & N_{zz} \mu \\
N_{xy} \mu, N_{xz} \mu & N_{yx} \mu, N_{yz} \mu & N_{zx} \mu, N_{zy} \mu
\end{array} \quad (10.25)$$

This requires 9 multiplications. Then, eq. (10.24) becomes:

$$\mathbf{Q}_{ij}^{(3 \times 3)} = \begin{bmatrix} N_{i,x}N_{j,x}M + N_{i,y}N_{j,y}\mu + N_{i,z}N_{j,z}\mu & N_{i,x}N_{j,y}\lambda + N_{i,y}N_{j,x}\mu & N_{i,x}N_{j,z}\lambda + N_{i,z}N_{j,x}\mu \\ N_{i,y}N_{j,x}\lambda + N_{i,x}N_{j,y}\mu & N_{i,y}N_{j,y}M + N_{i,x}N_{j,x}\mu + N_{i,z}N_{j,z}\mu & N_{i,y}N_{j,z}\lambda + N_{i,z}N_{j,y}\mu \\ N_{i,z}N_{j,x}\lambda + N_{i,x}N_{j,z}\mu & N_{i,z}N_{j,y}\lambda + N_{i,y}N_{j,z}\mu & N_{i,z}N_{j,z}M + N_{i,y}N_{j,y}\mu + N_{i,x}N_{j,x}\mu \end{bmatrix} \quad (10.26)$$

$$\mathbf{Q}_{ij}^{(3 \times 3)} = \begin{bmatrix} N_{xx}M + N_{yy}\mu + N_{zz}\mu & N_{xy}\lambda + N_{yx}\mu & N_{xz}\lambda + N_{zx}\mu \\ N_{yx}\lambda + N_{xy}\mu & N_{yy}M + N_{xx}\mu + N_{zz}\mu & N_{yz}\lambda + N_{zy}\mu \\ N_{zx}\lambda + N_{xz}\mu & N_{zy}\lambda + N_{yz}\mu & N_{zz}M + N_{yy}\mu + N_{xx}\mu \end{bmatrix}$$

$$\begin{array}{l} \times 1 \\ \times 2 \end{array} \quad \begin{array}{ccc} N_{xx}M & N_{yy}M & N_{zz}M \\ N_{xx}\mu & N_{yy}\mu & N_{zz}\mu \end{array} \quad (10.27)$$

$$\begin{array}{l} \times 1 \\ \times 1 \end{array} \quad \begin{array}{cccccc} N_{xy}\lambda & N_{xz}\lambda & N_{yx}\lambda & N_{yz}\lambda & N_{zx}\lambda & N_{zy}\lambda \\ N_{xy}\mu & N_{xz}\mu & N_{yx}\mu & N_{yz}\mu & N_{zx}\mu & N_{zy}\mu \end{array} \quad (10.28)$$

Therefore, this requires 18 multiplications. For these calculations we need 6 multiplications for each axis for a total of 18 multiplications. The number of calculations is shown in Table 10.1 for the case of recalculating and not recalculating repeated values.

The 4 variations for handling the isotropic material may differ depending on the exact implementation and underlying hardware (GPU is an interesting case here). However, in typical implementations the difference is expected to be small.

The evaluation of \mathbf{Q}_{ij} as mentioned above should always be preferred to a generic matrix multiplication, which requires:

- Step1: $\mathbf{C}_j = \mathbf{E} \mathbf{B}_j \Rightarrow 6 \cdot 6 \cdot 3 = 108$ multiplications, $6 \cdot 5 \cdot 3 = 90$ additions
 $(6 \times 3) \quad (6 \times 6) (6 \times 3)$
- Step2: $\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j \Rightarrow 3 \cdot 6 \cdot 3 = 54$ multiplications, $3 \cdot 5 \cdot 3 = 45$ additions
 $(3 \times 3) \quad (3 \times 6) (6 \times 3)$

The computation of each \mathbf{Q}_{ij} block requires the number of operations and temporary variables shown in Table 10.1. There can be minor savings for $i = j$. This calculation is repeated for all node pairs (see eq. 10.11) and for every Gauss point.

Material Type	Multiplications	Additions	Temporary Variables
Generic multiplication	$108+54=162$	$90+45=135$	18 (for C_j)
Anisotropic	$54+27=81$	$36+18=54$	18 (for C_j)
Orthotropic	$15+21=36$	$0+12=12$	15 (for C_j)
Isotropic 1 (without recalculation)	$9+18=27$	$0+12=12$	9 (for C_j) + 3 for repeated values of (10.22)
Isotropic 1 (with recalculation)	$9+21=30$	$0+12=12$	9 (for C_j)
Isotropic 2 (without recalculation)	$9+18=27$	$0+12=12$	9 (for all N_{ab}) + 3 for repeated values of (10.27)
Isotropic 2 (with recalculation)	$9+21=30$	$0+12=12$	9 (for all N_{ab})

Table 10.1: Multiplications, additions and temporary variables required for a single pair of nodes and at a single Gauss point

10.4 Total number of calculations required

The calculations of Table 10.1 will be performed for all i, j but due to symmetry, only $\frac{n(n+1)}{2}$ times instead of n^2 times. The total multiplications per Gauss point are shown in Table 10.2 and the total additions can be derived similarly.

Material Type	Multiplications per Gauss point		
	Step 1	Step 2	Total
Generic multiplication	$108 \frac{n(n+1)}{2}$	$54 \frac{n(n+1)}{2}$	$81 n(n+1)$
Anisotropic	$54 \frac{n(n+1)}{2}$	$27 \frac{n(n+1)}{2}$	$81 \frac{n(n+1)}{2}$
Orthotropic	$15 \frac{n(n+1)}{2}$	$21 \frac{n(n+1)}{2}$	$18 n(n+1)$
Isotropic 1 & 2 (without recalculation)	$9 \frac{n(n+1)}{2}$	$18 \frac{n(n+1)}{2}$	$27 \frac{n(n+1)}{2}$
Isotropic 1 & 2 (with recalculation)	$9 \frac{n(n+1)}{2}$	$21 \frac{n(n+1)}{2}$	$15 n(n+1)$

Table 10.2: Multiplications required for calculations at a single Gauss point

The contribution-wise approaches (Gauss point wise & element wise) externally iterate through nodes and internally through node pairs. Since the deformation matrix \mathbf{B} is evaluated for a particular Gauss point, the method can take advantage of the repetition of $\mathbf{E}\mathbf{B}_j$ elements to save on calculations at the cost of additional memory. For n number of nodes, we can perform n calculations of \mathbf{C}_j and reuse them properly to calculate \mathbf{Q}_{ij} . Therefore, step 1 is performed n and step 2 is performed $\frac{n(n+1)}{2}$ times. Table 10.3 shows the total multiplications per Gauss point when performing calculations this way, and the total additions can be derived similarly.

Material Type	Multiplications per Gauss point		
	Step 1	Step 2	Total
Generic multiplication	$108n$	$54 \frac{n(n+1)}{2}$	$27n(n+5)$
Anisotropic	$54n$	$27 \frac{n(n+1)}{2}$	$\frac{27}{2}n(n+5)$
Orthotropic	$15n$	$21 \frac{n(n+1)}{2}$	$\frac{3}{2}n(7n+17)$
Isotropic 1 & 2 (without recalculation)	$9n$	$18 \frac{n(n+1)}{2}$	$9n(n+2)$
Isotropic 1 & 2 (with recalculation)	$9n$	$21 \frac{n(n+1)}{2}$	$\frac{1}{2}n(21n+39)$

Table 10.3: Multiplications required for calculations at a single Gauss point

Note that to generically calculate $\mathbf{B}^T \mathbf{E} \mathbf{B}$ as a whole, it would require:

- $\mathbf{E} \mathbf{B} \Rightarrow 6 \cdot 6 \cdot 3n = 108n$ multiplications, $6 \cdot 5 \cdot 3n = 90n$ additions per integration point
(6×6)($6 \times 3n$)
- $\mathbf{B}^T \mathbf{C} \Rightarrow 3n \cdot 6 \cdot 3n = 54n^2$ multiplications, $3n \cdot 5 \cdot 3n = 45n^2$ per integration point
($3n \times 6$)($6 \times 3n$)

The total effort for generic matrix multiplications is:

- $54n(n+2)$ multiplications, $45n(n+2)$ additions
- $18n$ temporary variables for $\mathbf{E} \mathbf{B}$. Additionally, it requires the formation of \mathbf{B} and \mathbf{E} and corresponding memory for storing them.

10.5 The elasticity tensor in 2D problems

The elasticity tensor or stiffness tensor is the tensor expression of Hooke's law [97].

$$\boldsymbol{\sigma} = \mathbf{E} \boldsymbol{\varepsilon} \quad (10.29)$$

The matrix form of \mathbf{E} is always symmetric.

10.5.1 Anisotropic material

An anisotropic material has 6 independent elastic coefficients and its matrix representation is:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} E_{xx.xx} & E_{xx.yy} & E_{xx.xy} \\ & E_{yy.yy} & E_{yy.xy} \\ \text{symm} & & E_{xy.xy} \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix} \quad (10.30)$$

10.5.2 Orthotropic material

An orthotropic material has 4 independent elastic coefficients and its matrix representation is:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} E_{xx.xx} & E_{xx.yy} & 0 \\ & E_{yy.yy} & 0 \\ \text{symm} & & E_{xy.xy} \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix} \quad (10.31)$$

10.5.3 Isotropic material under Plane Stress

An isotropic material has 2 independent elastic coefficients. Hooke's law in this case is:

$$\begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & 2(1+\nu) \end{bmatrix} \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} \quad (10.32)$$

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix} \quad (10.33)$$

It can be stored with 2 values only, or, to avoid recalculations, define:

$$E_{11} := \frac{E}{1-\nu^2} \quad (10.34)$$

Furthermore, the last element is equal to the Shear Modulus:

$$\frac{E}{1-\nu^2} \frac{1-\nu}{2} = \frac{E}{2(1+\nu)} = G \quad (10.35)$$

Therefore, the formula can be written as:

$$\mathbf{E} = \begin{bmatrix} E_{11} & \nu E_{11} & 0 \\ \nu E_{11} & E_{11} & 0 \\ 0 & 0 & G \end{bmatrix} \quad (10.36)$$

By storing 3 values, namely E_{11} , νE_{11} and G , recalculations are avoided.

10.6 The deformation matrix in 2D problems

The deformation matrix \mathbf{B} handles the conversion between strains and nodal displacements:

$$\underset{(3 \times 1)}{\boldsymbol{\varepsilon}} = \underset{(3 \times 2n)}{\mathbf{B}} \underset{(2n \times 1)}{\mathbf{d}} \quad (10.37)$$

\mathbf{B} can be broken down to sub-matrices, each one corresponding to one node:

$$\underset{(3 \times 2n)}{\mathbf{B}} = \left[\underset{(3 \times 2)}{\mathbf{B}_1} \quad \mathbf{B}_2 \quad \dots \quad \mathbf{B}_n \right] \quad (10.38)$$

Each sub-matrix \mathbf{B}_i in eq. (10.38) has size 3×2 , but only has 2 distinct values: $N_{i,x}$, $N_{i,y}$.

$$\underset{(6 \times 3)}{\mathbf{B}_i} = \begin{bmatrix} N_{i,x} & 0 \\ 0 & N_{i,y} \\ N_{i,y} & N_{i,x} \end{bmatrix} \quad (10.39)$$

Instead of storing a 3×2 matrix, these 2 values can be stored and are sufficient to fully represent the underlying matrix. To store all info the whole matrix \mathbf{B} contains, 2 values per node are required, hence $2n$ values instead of a $3 \times 2n$ matrix. Note that \mathbf{B} never need to actually be formed.

10.7 Explicit calculation in 2D problems

The fast computation of the matrix product:

$$\mathbf{B}^T \mathbf{E} \mathbf{B} \quad (10.40)$$

$(2n \times 3)(3 \times 3)(3 \times 2n)$

is important because it is repeated at each integration point. This is important for finite elements but is even more important for EFG and IGA methods where the number of Gauss points and the number of influenced nodes per Gauss point are both significantly greater.

The operation can be broken down to smaller node pair operations:

$$\begin{bmatrix} \mathbf{B}_1^T \\ \mathbf{B}_2^T \\ \vdots \\ \mathbf{B}_n^T \end{bmatrix}_{(2 \times 3)} \mathbf{E} \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \dots & \mathbf{B}_n \end{bmatrix}_{(3 \times 2)} = \begin{bmatrix} \mathbf{B}_1^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_1^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_1^T \mathbf{E} \mathbf{B}_n \\ \mathbf{B}_2^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_2^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_2^T \mathbf{E} \mathbf{B}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_n^T \mathbf{E} \mathbf{B}_1 & \mathbf{B}_n^T \mathbf{E} \mathbf{B}_2 & \dots & \mathbf{B}_n^T \mathbf{E} \mathbf{B}_n \end{bmatrix} \quad (10.41)$$

For each pair of nodes i, j . we need to calculate the product:

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{E} \mathbf{B}_j = \mathbf{Q}_{ji}^T \quad (10.42)$$

$(2 \times 2) \quad (2 \times 3)(3 \times 3)(3 \times 2) \quad (2 \times 2)$

Note that, with the exception of $i=j$, \mathbf{Q}_{ij} is generally not symmetric.

The product $\mathbf{B}^T \mathbf{E} \mathbf{B}$ need not be calculated as a whole. Its parts \mathbf{Q}_{ij} can be produced and consumed immediately (the exact way differs depending on the assembly method used). Computation of \mathbf{Q}_{ij} plus associated indexing to access the \mathbf{K} entries, dominates the total effort [96]. The computation is split in two steps:

$$\text{Step 1: } \mathbf{C}_j = \mathbf{E} \mathbf{B}_j \quad \text{Step 2: } \mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j \quad (10.43)$$

$(3 \times 2) \quad (3 \times 3)(3 \times 2) \quad (2 \times 2) \quad (2 \times 3)(3 \times 2)$

10.7.1 Anisotropic material

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} (3 \times 2) & (3 \times 3) & (3 \times 2) \end{matrix} \begin{bmatrix} E_{xx \cdot xx} & E_{xx \cdot yy} & E_{xx \cdot xy} \\ & E_{yy \cdot yy} & E_{yy \cdot xy} \\ \text{symm} & & E_{xy \cdot xy} \end{bmatrix} \begin{bmatrix} N_{j,x} & 0 \\ 0 & N_{j,y} \\ N_{j,y} & N_{j,x} \end{bmatrix} \quad (10.44)$$

(column-wise pattern)

$$\mathbf{C}_j = \begin{matrix} (3 \times 2) \end{matrix} \begin{bmatrix} N_{j,x} E_{xx \cdot xx} + N_{j,y} E_{xx \cdot xy} & N_{j,y} E_{xx \cdot yy} + N_{j,x} E_{xx \cdot xy} \\ N_{j,x} E_{xx \cdot yy} + N_{j,y} E_{yy \cdot xy} & N_{j,y} E_{yy \cdot yy} + N_{j,x} E_{yy \cdot xy} \\ N_{j,x} E_{xx \cdot xy} + N_{j,y} E_{xy \cdot xy} & N_{j,y} E_{yy \cdot xy} + N_{j,x} E_{xy \cdot xy} \end{bmatrix}$$

$$\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j = \begin{matrix} (2 \times 2) & (2 \times 3) & (3 \times 2) \end{matrix} \begin{bmatrix} N_{i,x} & 0 & N_{i,y} \\ 0 & N_{i,y} & N_{i,x} \end{bmatrix} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{bmatrix} \quad (10.45)$$

$$\mathbf{Q}_{ij} = \begin{matrix} (2 \times 2) \end{matrix} \begin{bmatrix} N_{i,x} C_{11} + N_{i,y} C_{31} & N_{i,x} C_{12} + N_{i,y} C_{32} \\ N_{i,y} C_{21} + N_{i,x} C_{31} & N_{i,y} C_{22} + N_{i,x} C_{32} \end{bmatrix}$$

10.7.2 Orthotropic material

$$\mathbf{C}_j = \mathbf{E} \mathbf{B}_j = \begin{matrix} (3 \times 2) & (3 \times 3) & (3 \times 2) \end{matrix} \begin{bmatrix} E_{xx \cdot xx} & E_{xx \cdot yy} & \\ & E_{yy \cdot yy} & \\ \text{symm} & & E_{xy \cdot xy} \end{bmatrix} \begin{bmatrix} N_{j,x} & 0 \\ 0 & N_{j,y} \\ N_{j,y} & N_{j,x} \end{bmatrix} \quad (10.46)$$

$$\mathbf{C}_j = \begin{matrix} (3 \times 2) \end{matrix} \begin{bmatrix} N_{j,x} E_{xx \cdot xx} & N_{j,y} E_{xx \cdot yy} \\ N_{j,x} E_{xx \cdot yy} & N_{j,y} E_{yy \cdot yy} \\ N_{j,y} E_{xy \cdot xy} & N_{j,x} E_{xy \cdot xy} \end{bmatrix}$$

The computation $\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j$ is the same as in eq (10.45).
 $(2 \times 2) \quad (2 \times 3) \quad (3 \times 2)$

10.7.3 Isotropic material under Plane Stress

Eq (10.46) can be simplified:

$$\begin{aligned}
\mathbf{C}_j &= \mathbf{E} \mathbf{B}_j = \begin{matrix} (3 \times 2) & (3 \times 3) & (3 \times 2) \end{matrix} \begin{bmatrix} E_{11} & \nu E_{11} & 0 \\ \nu E_{11} & E_{11} & 0 \\ 0 & 0 & G \end{bmatrix} \begin{bmatrix} N_{j,x} & 0 \\ 0 & N_{j,y} \\ N_{j,y} & N_{j,x} \end{bmatrix} \\
\mathbf{C}_j &= \begin{matrix} (3 \times 2) \end{matrix} \begin{bmatrix} N_{j,x} E_{11} & N_{j,y} \nu E_{11} \\ N_{j,x} \nu E_{11} & N_{j,y} E_{11} \\ N_{j,y} G & N_{j,x} G \end{bmatrix}
\end{aligned} \tag{10.47}$$

The computation $\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j$ is the same as in eq (10.45).
 $(2 \times 2) \quad (2 \times 3) \quad (3 \times 2)$

The evaluation of \mathbf{Q}_{ij} as mentioned above should always be preferred to a generic matrix multiplication, which requires:

- Step1: $\mathbf{C}_j = \mathbf{E} \mathbf{B}_j \Rightarrow 3 \cdot 3 \cdot 2 = 18$ multiplications, $3 \cdot 2 \cdot 2 = 12$ additions
 $(3 \times 2) \quad (3 \times 3) \quad (3 \times 2)$
- Step2: $\mathbf{Q}_{ij} = \mathbf{B}_i^T \mathbf{C}_j \Rightarrow 2 \cdot 3 \cdot 2 = 12$ multiplications, $2 \cdot 2 \cdot 2 = 8$ additions
 $(2 \times 2) \quad (2 \times 3) \quad (3 \times 2)$

The computation of each \mathbf{Q}_{ij} block requires the number of operations and temporary variables shown in Table 10.4. There can be minor savings for $i = j$. This calculation is repeated for all node pairs (see eq. 10.41) and for every Gauss point.

Material Type	Multiplications	Additions	Temporary Variables
Generic multiplication	18+12=40	12+8=20	6 (for \mathbf{C}_j)
Anisotropic	12+8=20	6+4=10	6 (for \mathbf{C}_j)
Orthotropic	6+8=14	0+4=4	6 (for \mathbf{C}_j)
Isotropic (place stress)	6+8=14	0+4=4	6 (for \mathbf{C}_j)

Table 10.4: Multiplications, additions and temporary variables required for a single pair of nodes and at a single Gauss point

10.8 Total number of calculations required in 2D problems

The calculations of Table 10.4 will be performed for all i, j but due to symmetry, only $\frac{n(n+1)}{2}$ times instead of n^2 times. The total multiplications per Gauss point are shown in Table 10.5 and the total additions can be derived similarly.

Material Type	Multiplications per Gauss point		
	Step 1	Step 2	Total
Generic multiplication	$18 \frac{n(n+1)}{2}$	$12 \frac{n(n+1)}{2}$	$20 n(n+1)$
Anisotropic	$12 \frac{n(n+1)}{2}$	$8 \frac{n(n+1)}{2}$	$10 n(n+1)$
Orthotropic	$6 \frac{n(n+1)}{2}$	$8 \frac{n(n+1)}{2}$	$7 n(n+1)$
Isotropic (place stress)	$6 \frac{n(n+1)}{2}$	$8 \frac{n(n+1)}{2}$	$7 n(n+1)$

Table 10.5: Multiplications required for calculations at a single Gauss point

The contribution-wise approaches (Gauss point wise & element wise) externally iterate through nodes and internally through node pairs. Since the deformation matrix \mathbf{B} is evaluated for a particular Gauss point, the method can take advantage of the repetition of $\mathbf{E}\mathbf{B}_j$ elements to save on calculations at the cost of additional memory. For n number of nodes, we can to perform n calculations of \mathbf{C}_j and reuse them properly to calculate \mathbf{Q}_{ij} . Therefore, step 1 is performed n and step 2 is performed $\frac{n(n+1)}{2}$ times. Table 10.6 shows the total multiplications per Gauss point when performing calculations this way, and the total additions can be derived similarly.

Material Type	Multiplications per Gauss point		
	Step 1	Step 2	Total
Generic multiplication	$18n$	$12 \frac{n(n+1)}{2}$	$6n(n+4)$
Anisotropic	$12n$	$8 \frac{n(n+1)}{2}$	$4n(n+4)$
Orthotropic	$6n$	$8 \frac{n(n+1)}{2}$	$2n(2n+5)$
Isotropic (plane stress)	$6n$	$8 \frac{n(n+1)}{2}$	$2n(2n+5)$

Table 10.6: Multiplications required for calculations at a single Gauss point (alternative way)

Note that to generically calculate $\mathbf{B}^T \mathbf{E} \mathbf{B}$ as a whole, it would require:

- $\mathbf{E} \mathbf{B} \Rightarrow 3 \cdot 3 \cdot 2n = 18n$ multiplications, $3 \cdot 2 \cdot 2n = 12n$ additions per integration point
(3×3)($3 \times 2n$)
- $\mathbf{B}^T \mathbf{C} \Rightarrow 2n \cdot 3 \cdot 2n = 12n^2$ multiplications, $2n \cdot 2 \cdot 2n = 8n^2$ per integration point
($2n \times 3$)($3 \times 2n$)

The total effort for generic matrix multiplications is:

- $6n(2n+3)$ multiplications, $4n(2n+3)$ additions
- $6n$ temporary variables for $\mathbf{E} \mathbf{B}$. Additionally, it requires the formation of \mathbf{B} and \mathbf{E} and corresponding memory for storing them.

11 References

- [1] “Meshfree methods,” *Wikipedia, the free encyclopedia*. 13-Dec-2013.
- [2] V. P. Nguyen, T. Rabczuk, S. Bordas, and M. Duflot, “Meshless methods: A review and computer implementation aspects,” *Mathematics and Computers in Simulation*, vol. 79, no. 3, pp. 763–813, 2008.
- [3] S. Li and W. K. Liu, “Meshfree and particle methods and their applications,” *Applied Mechanics Reviews*, vol. 55, no. 1, pp. 1–34, 2002.
- [4] G. R. Liu, *Meshfree methods: moving beyond the finite element method*. Boca Raton: CRC Press, 2010.
- [5] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl, “Meshless methods: An overview and recent developments,” *Computer Methods in Applied Mechanics and Engineering*, vol. 139, no. 1–4, pp. 3–47, 1996.
- [6] K. T. Danielson, S. Hao, W. K. Liu, R. A. Uras, and S. Li, “Parallel computation of meshless methods for explicit dynamic analysis,” *International Journal for Numerical Methods in Engineering*, vol. 47, no. 7, pp. 1323–1341, 2000.
- [7] K. T. Danielson, R. A. Uras, M. D. Adley, and S. Li, “Large-scale application of some modern CSM methodologies by parallel computation,” *Advances in engineering software*, vol. 31, no. 8, pp. 501–509, 2000.
- [8] G. R. Liu, K. Y. Dai, and T. T. Nguyen, “A smoothed finite element method for mechanics problems,” *Computational Mechanics*, vol. 39, no. 6, pp. 859–877, 2007.
- [9] J. G. Wang and G. R. Liu, “A point interpolation meshless method based on radial basis functions,” *International Journal for Numerical Methods in Engineering*, vol. 54, no. 11, pp. 1623–1648, 2002.
- [10] Y. T. Gu and G. R. Liu, “A coupled element free Galerkin/boundary element method for stress analysis of tow-dimensional solids,” *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 34, pp. 4405–4419, 2001.
- [11] W.-R. Yuan, P. Chen, and K.-X. Liu, “High performance sparse solver for unsymmetrical linear equations with out-of-core strategies and its application on meshless methods,” *Applied Mathematics and Mechanics (English Edition)*, vol. 27, no. 10, pp. 1339–1348, 2006.
- [12] S. C. Wu, H. O. Zhang, C. Zheng, and J. H. Zhang, “A high performance large sparse symmetric solver for the meshfree Galerkin method,” *International Journal of Computational Methods*, vol. 5, no. 4, pp. 533–550, 2008.
- [13] E. Divo and A. Kassab, “Iterative domain decomposition meshless method modeling of incompressible viscous flows and conjugate heat transfer,” *Engineering Analysis with Boundary Elements*, vol. 30, no. 6, pp. 465–478, 2006.
- [14] P. Metsis and M. Papadrakakis, “Overlapping and non-overlapping domain decomposition methods for large-scale meshless EFG simulations,” *Computer Methods in Applied Mechanics and Engineering*, vol. 229–232, pp. 128–141, 2012.
- [15] P. Metsis, “Meshless Methods for solving large-scale problems,” Αθήνα, 2014.
- [16] B. Nayroles, G. Touzot, and P. Villon, “Generalizing the finite element method: Diffuse approximation and diffuse elements,” *Computational Mechanics*, vol. 10, no. 5, pp. 307–318, Sep. 1992.
- [17] Y. Y. Lu, T. Belytschko, and L. Gu, “A new implementation of the element free Galerkin method,” *Computer Methods in Applied Mechanics and Engineering*, vol. 113, no. 3–4, pp. 397–414, 1994.
- [18] T. J. R. Hughes, J. A. Cottrell, and Y. Bazilevs, “Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement,” *Computer Methods in Applied Mechanics and*

- Engineering*, vol. 194, no. 39–41, pp. 4135–4195, 2005.
- [19] F. Auricchio, F. Calabrò, T. J. R. Hughes, A. Reali, and G. Sangalli, “A simple algorithm for obtaining nearly optimal quadrature rules for NURBS-based isogeometric analysis,” *Computer Methods in Applied Mechanics and Engineering*, vol. 249–252, pp. 15–27, 2012.
- [20] T. J. R. Hughes, A. Reali, and G. Sangalli, “Efficient quadrature for NURBS-based isogeometric analysis,” *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 5–8, pp. 301–313, 2010.
- [21] J. A. Cottrell, T. J. R. Hughes, and Y. Bazilevs, *Isogeometric Analysis: Toward Integration of CAD and FEA*, 1st ed. Wiley, 2009.
- [22] F. Auricchio, L. Beirão da Veiga, T. J. R. Hughes, A. Reali, and G. Sangalli, “Isogeometric collocation for elastostatics and explicit dynamics,” *Computer Methods in Applied Mechanics and Engineering*, vol. 249–252, pp. 2–14, 2012.
- [23] D. Schillinger, J. A. Evans, A. Reali, M. A. Scott, and T. J. R. Hughes, “Isogeometric collocation: Cost comparison with Galerkin methods and extension to adaptive hierarchical NURBS discretizations,” *Computer Methods in Applied Mechanics and Engineering*, vol. 267, pp. 170–232, 2013.
- [24] J. Mandel, “Balancing domain decomposition,” *Communications in Numerical Methods in Engineering*, vol. 9, no. 3, pp. 233–241, 1993.
- [25] C. Farhat and F.-X. Roux, “Method of finite element tearing and interconnecting and its parallel solution algorithm,” *International Journal for Numerical Methods in Engineering*, vol. 32, no. 6, pp. 1205–1227, 1991.
- [26] Y. Fragakis and M. Papadrakakis, “The mosaic of high performance domain Decomposition Methods for Structural Mechanics: Formulation, interrelation and numerical efficiency of primal and dual methods,” *Computer Methods in Applied Mechanics and Engineering*, vol. 192, no. 35–36, pp. 3799–3830, 2003.
- [27] Y. Fragakis and M. Papadrakakis, “The mosaic of high-performance domain decomposition methods for structural mechanics - Part II: Formulation enhancements, multiple right-hand sides and implicit dynamics,” *Computer Methods in Applied Mechanics and Engineering*, vol. 193, no. 42–44, pp. 4611–4662, 2004.
- [28] V. Vondrák, T. Kozubek, A. Markopoulos, and Z. Dostál, “Parallel solution of contact shape optimization problems based on Total FETI domain decomposition method,” *Structural and Multidisciplinary Optimization*, vol. 42, no. 6, pp. 955–964, 2010.
- [29] G. M. Stavroulakis and M. Papadrakakis, “Advances on the domain decomposition solution of large scale porous media problems,” *Computer Methods in Applied Mechanics and Engineering*, vol. 198, no. 21–26, pp. 1935–1945, 2009.
- [30] A. Mobasher Amini, D. Dureisseix, P. Cartraud, and N. Buannic, “A domain decomposition method for problems with structural heterogeneities on the interface: Application to a passenger ship,” *Computer Methods in Applied Mechanics and Engineering*, vol. 198, no. 41–44, pp. 3452–3463, 2009.
- [31] D. Ghosh, P. Avery, and C. Farhat, “A FETI-preconditioned conjugate gradient method for large-scale stochastic finite element problems,” *International Journal for Numerical Methods in Engineering*, vol. 80, no. 6–7, pp. 914–931, 2009.
- [32] D. Ghosh and C. Farhat, “Strain and stress computations in stochastic finite element methods,” *International Journal for Numerical Methods in Engineering*, vol. 74, no. 8, pp. 1219–1239, 2008.
- [33] P. Avery and C. Farhat, “The FETI family of domain decomposition methods for inequality-constrained quadratic programming: Application to contact problems with conforming and nonconforming interfaces,” *Computer Methods in Applied Mechanics and Engineering*, vol.

- 198, no. 21–26, pp. 1673–1683, 2009.
- [34] Z. Cheng and M. Paraschivoiu, “A posteriori finite element bounds to linear functional outputs of the three-dimensional Navier-Stokes equations,” *International Journal for Numerical Methods in Engineering*, vol. 61, no. 11, pp. 1835–1859, 2004.
- [35] C. Farhat, R. Tezaur, and J. Toivanen, “A domain decomposition method for discontinuous Galerkin discretizations of Helmholtz problems with plane waves and Lagrange multipliers,” *International Journal for Numerical Methods in Engineering*, vol. 78, no. 13, pp. 1513–1531, 2009.
- [36] M. Yano and D. L. Darmofal, “BDDC preconditioning for high-order Galerkin Least-Squares methods using inexact solvers,” *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 45–48, pp. 2958–2969, 2010.
- [37] C. Farhat and F.-X. Roux, “Implicit parallel processing in structural mechanics,” *Computational Mechanics Advances*, vol. 2, no. 1, 1994.
- [38] S. Bitzarakis, M. Papadrakakis, and A. Kotsopoulos, “Parallel solution techniques in computational structural mechanics,” *Computer Methods in Applied Mechanics and Engineering*, vol. 148, no. 1–2, pp. 75–104, 1997.
- [39] Δ. Χ. Χαρμπής, “Ανάλυση Φορέων με Συμβατικά, Προσαρμοστικά και Στοχαστικά Πεπερασμένα, Στοιχεία σε Διαδικτυωμένους Ηλεκτρονικούς Υπολογιστές,” Αθήνα, 2002.
- [40] D. J. Rixen and C. Farhat, “A simple and efficient extension of a class of substructure based preconditioners to heterogeneous structural mechanics problems,” *International Journal for Numerical Methods in Engineering*, vol. 44, no. 4, pp. 489–516, 1999.
- [41] M. Papadrakakis and Y. Tsompanakis, “Domain decomposition methods for parallel solution of shape sensitivity analysis problems,” *International Journal for Numerical Methods in Engineering*, vol. 44, no. 2, pp. 281–303, 1999.
- [42] C. Farhat, A. Macedo, M. Lesoinne, F.-X. Roux, F. Magoulès, and A. De La Bourdonnaie, “Two-level domain decomposition methods with Lagrange multipliers for the fast iterative solution of acoustic scattering problems,” *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2–4, pp. 213–239, 2000.
- [43] C. Farhat, L. Crivelli, and F. X. Roux, “Extending substructure based iterative solvers to multiple load and repeated analyses,” *Computer Methods in Applied Mechanics and Engineering*, vol. 117, no. 1–2, pp. 195–209, 1994.
- [44] M. Papadrakakis and Y. Fragakis, “An integrated geometric-algebraic method for solving semi-definite problems in structural mechanics,” *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 49–50, pp. 6513–6532, 2001.
- [45] C. Farhat, A. Macedo, and M. Lesoinne, “A two-level domain decomposition method for the iterative solution of high frequency exterior Helmholtz problems,” *Numerische Mathematik*, vol. 85, no. 2, pp. 283–308, 2000.
- [46] Ι. Φραγκάκης, “Μέθοδοι Υψηλών Επιδόσεων για τη Στατική και Δυναμική Ανάλυση Φορέων με Πεπερασμένα Στοιχεία,” Αθήνα, 2004.
- [47] D. Göttsche, R. Strzodka, and S. Turek, “Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, 2007.
- [48] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.
- [49] D. Kirk and W. Hwu, *Programming massively parallel processors: a hands-on approach*, 2nd ed. San Francisco, Calif.; Oxford: Morgan Kaufmann ; Elsevier Science distributor, 2013.
- [50] NVIDIA Corporation, “CUDA C Best Practices Guide,” *NVIDIA GPU Computing Documentation | NVIDIA Developer Zone*, 2014. [Online]. Available:

- <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>. [Accessed: 02-Feb-2014].
- [51] “TOP500 Supercomputing Sites.” [Online]. Available: <http://www.top500.org/>. [Accessed: 09-Dec-2010].
- [52] I. C. Kambolis, X. S. Trompoukis, V. G. Asouti, and K. C. Giannakoglou, “CFD-based analysis and two-level aerodynamic optimization on graphics processing units,” *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 9–12, pp. 712–722, 2010.
- [53] E. Elsen, P. LeGresley, and E. Darve, “Large calculation of the flow over a hypersonic vehicle using a GPU,” *Journal of Computational Physics*, vol. 227, no. 24, pp. 10148–10161, 2008.
- [54] J. C. Thibault and I. Senocak, “Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms,” *Journal of Supercomputing*, vol. 59, no. 2, pp. 693–719, 2012.
- [55] M. De La Asunción, J. M. Mantas, and M. J. Castro, “Simulation of one-layer shallow water systems on multicore and CUDA architectures,” *Journal of Supercomputing*, vol. 58, no. 2, pp. 206–214, 2011.
- [56] H. Zhou, G. Mo, F. Wu, J. Zhao, M. Rui, and K. Cen, “GPU implementation of lattice Boltzmann method for flows with curved boundaries,” *Computer Methods in Applied Mechanics and Engineering*, vol. 225–228, pp. 65–73, 2012.
- [57] A. Sunarso, T. Tsuji, and S. Chono, “GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows,” *Journal of Computational Physics*, vol. 229, no. 15, pp. 5486–5497, 2010.
- [58] J. A. Anderson, C. D. Lorenz, and A. Traveset, “General purpose molecular dynamics simulations fully implemented on graphics processing units,” *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.
- [59] E. Wadbro and M. Berggren, “Megapixel topology optimization on a graphics processing unit,” *SIAM Review*, vol. 51, no. 4, pp. 707–721, 2009.
- [60] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster,” *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.
- [61] T. Takahashi and T. Hamada, “GPU-accelerated boundary element method for Helmholtz’ equation in three dimensions,” *International Journal for Numerical Methods in Engineering*, vol. 80, no. 10, pp. 1295–1321, 2009.
- [62] G. R. Joldes, A. Wittek, and K. Miller, “Real-time nonlinear finite element computations on GPU - Application to neurosurgical simulation,” *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 49–52, pp. 3305–3314, 2010.
- [63] S. Tomov, J. Dongarra, and M. Baboulin, “Towards dense linear algebra for hybrid GPU accelerated manycore systems,” *Parallel Computing*, vol. 36, no. 5–6, pp. 232–240, 2010.
- [64] O. Schenk, M. Christen, and H. Burkhart, “Algorithmic performance studies on graphics processing units,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1360–1369, 2008.
- [65] J. M. Elble, N. V. Sahinidis, and P. Vouzis, “GPU computing with Kaczmarz’s and other iterative algorithms for linear systems,” *Parallel Computing*, vol. 36, no. 5–6, pp. 215–231, 2010.
- [66] A. Cevahir, A. Nukada, and S. Matsuoka, “Fast conjugate gradients with multiple GPUs,” presented at the 9th International Conference on Computational Science, ICCS 2009, Baton Rouge, LA, 2009, vol. 5544 LNCS, pp. 893–903.
- [67] A. Cevahir, A. Nukada, and S. Matsuoka, “High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning,” *Computer Science - Research and Development*, vol. 25, no. 1–2, pp. 83–91, 2010.

- [68] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, “A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures,” *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13–16, pp. 1490–1508, 2011.
- [69] “Porting CUDA Applications to OpenCL™,” *AMD*. [Online]. Available: <http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl/>. [Accessed: 01-Feb-2014].
- [70] “CUDA,” *Wikipedia, the free encyclopedia*. 02-Feb-2014.
- [71] M. Harris, “Optimizing parallel reduction in CUDA,” *Proc. ACM SIGMOD*, vol. 13, pp. 104–110, 2007.
- [72] “Virtual memory,” *Wikipedia, the free encyclopedia*. 22-Feb-2014.
- [73] “Paging,” *Wikipedia, the free encyclopedia*. 16-Feb-2014.
- [74] “Zero-based numbering,” *Wikipedia, the free encyclopedia*. 06-Dec-2013.
- [75] “Row-major order,” *Wikipedia, the free encyclopedia*. 13-Nov-2013.
- [76] “Band matrix,” *Wikipedia, the free encyclopedia*. 16-Dec-2013.
- [77] “Cuthill–McKee algorithm,” *Wikipedia, the free encyclopedia*. 09-Jan-2014.
- [78] “Graph bandwidth,” *Wikipedia, the free encyclopedia*. 05-Sep-2013.
- [79] “Sparse matrix,” *Wikipedia, the free encyclopedia*. 2013.
- [80] “CUSPARSE.” [Online]. Available: <http://docs.nvidia.com/cuda/cusparses/#ellpack-itpack-format-ell>. [Accessed: 27-Jan-2014].
- [81] “Strassen algorithm,” *Wikipedia, the free encyclopedia*. 16-Feb-2014.
- [82] “Coppersmith–Winograd algorithm,” *Wikipedia, the free encyclopedia*. 15-Feb-2014.
- [83] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann, 2010.
- [84] NVIDIA Corporation, “CUDA Programming Guide Version 3.2.” [Online]. Available: <http://developer.nvidia.com/object/gpucomputing.html>.
- [85] Khronos OpenCL Working Group, “The OpenCL Specification.” [Online]. Available: <http://www.khronos.org/opencl/>. [Accessed: 08-Dec-2010].
- [86] K. Fatahalian and M. Houston, “A closer look at GPUs,” *Communications of the ACM*, vol. 51, no. 10, pp. 50–57, 2008.
- [87] Z. A. Taylor, M. Cheng, and S. Ourselin, “Real-time nonlinear finite element analysis for surgical simulation using graphics processing units.,” *Medical image computing and computer-assisted intervention : MICCAI ... International Conference on Medical Image Computing and Computer-Assisted Intervention*, vol. 10, no. Pt 1, pp. 701–708, 2007.
- [88] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, p. 40, Mar. 2008.
- [89] G. Stavroulakis, “Seismic soil-structure interaction with finite elements and the method of substructures,” Αθήνα, 2014.
- [90] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.
- [91] R. Trobec, M. Šterk, and B. Robič, “Computational complexity and parallelization of the meshless local Petrov-Galerkin method,” *Computers and Structures*, vol. 87, no. 1–2, pp. 81–90, 2009.
- [92] E. Barbieri and M. Meo, “A fast object-oriented Matlab implementation of the Reproducing Kernel Particle Method,” *Computational Mechanics*, vol. 49, no. 5, pp. 581–602, 2012.
- [93] A. Karatarakis, P. Metsis, and M. Papadrakakis, “GPU-acceleration of stiffness matrix calculation and efficient initialization of EFG meshless methods,” *Computer Methods in Applied Mechanics and Engineering*, vol. 258, pp. 63–80, 2013.
- [94] W. W. Hwu and D. B. Kirk, “Parallelism Scalability,” in *Programming and Tuning Massively Parallel Systems (PUMPS)*, Barcelona, 2011.

- [95] A. Karatarakis, P. Karakitsios, and M. Papadrakakis, “GPU accelerated computation of the isogeometric analysis stiffness matrix,” *Computer Methods in Applied Mechanics and Engineering*, vol. 269, pp. 334–355, 2014.
- [96] C. Felippa, “Chapter 15 - Solid Elements: Overview,” in *Advanced Finite Element Methods (ASEN 6367) Course Material*, 2011.
- [97] “Hooke’s law,” *Wikipedia, the free encyclopedia*. 16-Feb-2014.