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

SEISMIC SOIL-STRUCTURE INTERACTION WITH FINITE ELEMENTS AND THE METHOD OF SUBSTRUCTURES

PhD dissertation

George Stavroulakis

Advisor:

Professor Manolis Papadrakakis

June 2014

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

Seismic Soil-Structure Interaction with Finite Elements and the Method of Substructures

by George Stavroulakis

Advisor:
Professor Manois Papadrakakis

Athens,
June 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Πολιτικών Μηχανικών
Εργαστήριο Στατικής και Αντισεισμικών Ερευνών

Σεισμική αλληλεπίδραση εδάφους-κατασκευής με Πεπερασμένα Στοιχεία και τη μέθοδο των υποφορέων

από τον Γεώργιο Σταυρουλάκη

Επιβλέπων:
Καθηγητής Μανόλης Παπαδρακάκης

Αθήνα,
Ιούνιος 2014

Dedicated to my parents Michael and Elli,

my beloved brother Manolis,

my dearest friend and best man Spyros,

my colleague Pantelis

and my musical companions Chris, Dimitris and Michael

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.

Konstantinos V. Spiliopoulos
Associate 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.

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.

Konstantinos Spyrakos
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.

Manolis Kavvadas
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.

Vissarion Papadopoulos
Assistant 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

One of the most fundamental problems in structural engineering deals with the behavior of structures under seismic loading. Such a problem can prove to be very cumbersome to solve accurately due to the various factors that contribute to its solution, including the presence of water inside the soil pores, the inherent uncertainties of both the structure and the soil, the interaction of the structure foundations with the soil and the difficulty to solve computationally the resulting numerical model due to its massive scale.

This Thesis deals with all of the above problems with the aim to provide a computational toolbox for addressing the solution of this complicated problem and is organized as follows: Chapter 1 is an introduction, chapter 2 describes the u-p formulation of the porous media problem along with its spatial and temporal discretization. Chapter 3 describes the theoretical background for stochastic analysis and describes the Spectral Stocahastic Finite Element Method (SSFEM). Chapter 4 is dedicated to solution algorithms for solving problems described in the previous chapters and suggests novel and computationally efficient methods for the solution of both porous media and stochastic problems using the Monte Carlo and SSFEM. Chapter 5 describes the programming paradigms used in order to implement the methods described in the previous chapters while chapter 6 is dedicated to parallel programming for the CPU. Chapter 7 describes the nVidia GPU architecture and how domain decomposition methods were implemented on this architecture and chapter 8 is a collection of numerical examples based on the material of all the previous chapters. Finally, chapter 9 is concludes with an overview of the present work, followed by bibliography.

Περίληψη

Ένα από τα πλέον θεμελιώδη προβλήματα του Πολιτικού Μηχανικού είναι η ανάλυση της συμπεριφοράς των κατασκευών υπό σεισμική φόρτιση. Ένα τέτοιο πρόβλημα είναι αρκετά δύσκολο και περίπλοκο στο χειρισμό του λόγω των διάφορων παραγόντων που υπεισέρχονται κατά την επίλυσή του όπως η ύπαρξη ρευστού στους πόρους του εδάφους, οι εγγενείς αβεβαιότητες τόσο της κατασκευής όσο και του εδάφους. η αλληλεπίδραση της κατασκευής με το έδαφος καθώς και το μεγάλο υπολογιστικό κόστος το οποίο έχει η επίλυση ενός αριθμητικού μοντέλου που θα τα λαμβάνει αυτά υπόψιν.

Η παρούσα διατριβή εντρυφεί σε όλους τους ως άνω παράγοντες με στόχο να προτείνει μια υπολογιστική «εργαλειοθήκη» για την επίλυση αυτού του προβλήματος και είναι οργανωμένη ως ακολούθως: Το πρώτο κεφάλαιο αποτελεί την εισαγωγή της διατριβής ενώ το δεύτερο κεφάλαιο περιγράφει τη u-p μόρφωση του προβλήματος των πορώδων μέσων μαζί με τον τρόπο χωρικής και χρονικής του διακριτοποίησης. Το τρίτο κεφάλαιο περιέχει το θεωρητικό υπόβαθρο για στοχαστική ανάλυση και περιγράφει τη φασματική στοχαστική μέθοδο των πεπερασμένων στοιχείων. Το τέταρτο κεφάλαιο περιγράφει τους υπολογιστικούς αλγορίθμους για την επίλυση των προβλημάτων που περιγράφηκαν στα προηγούμενα κεφάλαια και προτείνει νέες και υπολογιστικά βέλτιστες μεθόδους για τη επίλυση τόσο προβλημάτων πορώδων μέσων όσο και στοχαστικών προβλημάτων, τόσο με τη μέθοδο Monte Carlo όσο και με τη φασματική στοχαστική μέθοδο των πεπερασμένων στοιχείων. Το πέμπτο κεφάλαιο περιγράφει το προγραμματιστικό μοντέλο που χρησιμοποιήθηκε για την υλοποίηση των αλγορίθμων του προηγούμενου κεφαλαίου ενώ το έκτο κεφάλαιο περιέχει πληροφορίες για τον παράλληλο προγραμματισμό υπολογιστών. Το έβδομο κεφάλαιο περιγράφει την αρχιτεκτονική των καρτών γραφικών (GPU) της nVidia καθώς και τον τρόπο υλοποίησης των αλγορίθμων του τρίτου κεφαλαίου σε αυτό το περιβάλλον. Το όγδοο κεφάλαιο είναι μια συλλογή αριθμητικών παραδειγμάτων βασισμένο στο υλικό των προηγούμενων κεφαλαίων. Τέλος το ένατο κεφάλαιο κλείνει με μια ανακεφαλαίωση της διατριβής και ακολουθεί η βιβλιογραφία.

Acknowledgements

First, I would like to thank my advisor, Professor Manolis Papadrakakis, for his spirit, vision, patience and scientific guidance throughout the years, starting from my graduate studies until now.

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

Special thanks to Pantelis Petrogiannakis, Kostas Rammos, Apostolos Zafiris and Nick Palladinos of Nessos for directing me towards sound programming foundations and for shaping my attitude towards life.

I would also like to thank Spyros Massinas for existing in this world, my co-authors Alex Karatarakis and Dimitris Giovanis for their work and commitment, Theofilos Manitaras for his support and Konstantinos Papanikolopoulos for this hard work, dedication and focused research. Finally I would like to thank the rest of the research team of Professor Papadrakakis for constantly providing stimuli and promoting research.

Athens, June 2014

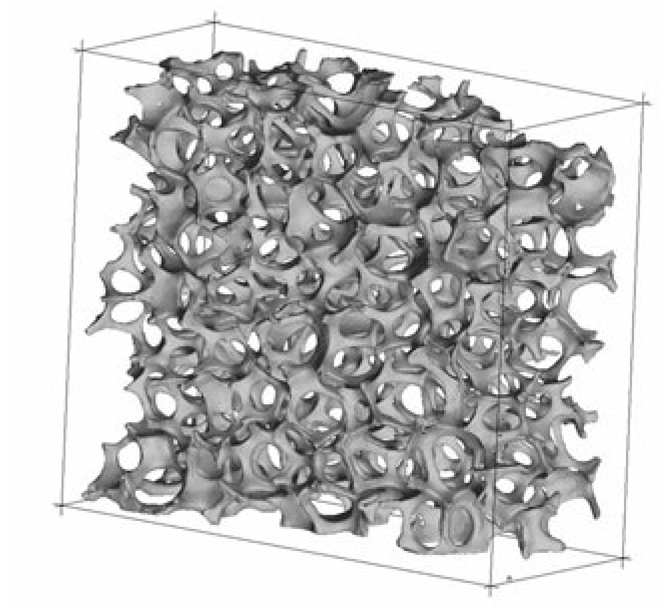
George Stavroulakis.

Εκτεταμένη περίληψη

Για υλικά τα οποία αποτελούνται από μία φάση (single phase materials), όπως αυτά τα οποία συναντούμε στη μηχανική όπως ο χάλυβας είναι αρκετά εύκολο να προσδιορίσουμε τα διάφορα φορτία αστοχίας για μια κατασκευή με σχετικά απλούς υπολογισμούς, ιδίως αν μας αφορούν στατικά προβλήματα. Κατ’ αντιστοιχία και στον τομέα της εδαφομηχανικής, με αντίστοιχους υπολογισμούς, μπορούμε να προσδιορίσουμε φορτία αστοχίας αν και όχι με εξ’ ίσου εύκολο τρόπο. Εντούτοις, σε προβλήματα εδαφοδυναμικής, η χρήση τέτοιων υπολογισμών είναι στις περισσότερες περιπτώσεις μη αποδεκτή.

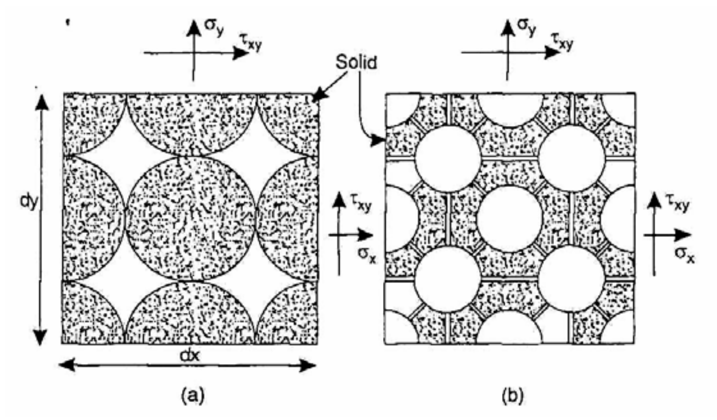
Ο λόγος για τον οποίο συμβαίνει αυτό, έγκειται στο γεγονός ότι η συμπεριφορά εδαφών ή βραχομαζών στις οποίες οι πόροι της στερεάς δομής τους είναι γεμάτοι με κάποιο υγρό, δεν μπορεί να παραλληλιστεί με τη συμπεριφορά υλικών μίας φάσης. Μάλιστα, για κάποιους αποτελεί ένα ανοιχτό ερώτημα το κατά πόσο τέτοιου είδους υλικά είναι δυνατόν να περιγραφούν με τις μεθόδους της μηχανικής συνεχούς μέσου.

Διαπιστώνουμε λοιπόν οτι στη γενική περίπτωση, η πλήρης λύση του προβλήματος της παραμόρφωσης του στερεού υλικού συζευγμένη με μια αιφνίδια (transient) ροή ρευστού χρειάζεται να ευρεθεί, λύση η οποία απαιτεί την κατασκευή και επίλυση συζευγμένων εξισώσεων οι οποίες θα περιγραφούν εκτενώς παρακάτω.



Σκίτσο 2.1 – Μια βραχόμαζα, μοντελοποιημένη ως πορώδες μέσο

Αν λοιπόν ορίσουμε την ολική τάση σ με τις συνιστώσες της σ_{ij} χρησιμοποιώντας τη σύμβαση των δεικτών, αυτές καθορίζονται από την άθροιση των κατάλληλων δυνάμεων στην *i*-κατεύθυνση στις διατομές dx_j. Οι επιφάνειες των διατομών για δυο διαφορετικά είδη πορωδών υλικών φαίνονται στο σκίτσο 2.2.



Σκίτσο 2.2 – Επιφάνεια τομών δύο διαφορετικών πορωδών μέσων

Έτσι έχουμε:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} \text{ ή } \boldsymbol{\sigma} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{bmatrix}$$

Επί πλέον αν οι τάσεις που ασκούνται στη στερεά φάση του υλικού μας οριστούν ως ενεργές τάσεις σ' επί των ανωτέρω διατομών τότε οι υδροστατικές πιέσεις που οφείλονται στις πιέσεις πόρων p και οι οποίες ενεργούν μόνο στην περιοχή των πόρων θα είναι ίσες με:

$$-\delta_{ij} np$$

όπου n είναι το πορώδες του υλικού και δ_{ij} είναι το δέλτα του Kronecker. Το αρνητικό πρόσημο εισάγεται για να ικανοποιηθεί η γενική σύμβαση του θετικού πρόσημου των εφελκυστικών τάσεων.

Οι ως άνω ορισμοί οδηγούν στην παρακάτω σχέση η οποία συνδέει τις ενεργές με τις ολικές τάσεις και έχει ως εξής:

$$\sigma_{ij} = \sigma'_{ij} - \delta_{ij} np$$

ενώ αν χρησιμοποιήσουμε διανυσματική σημειογραφία τότε:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}' - \mathbf{m} np \quad (1.1)$$

όπου **m** είναι το παρακάτω διάνυσμα:

$$\mathbf{m} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Σε πρακτικά προβλήματα, οι πόροι ενός πορώδους μέσου μπορεί να είναι κατειλημμένοι από δύο ή περισσότερα ρευστά. Στα παρακάτω, θα θεωρήσουμε μόνο δύο ρευστά με το βαθμό κορεσμού για κάθε ρευστό να ορίζεται από την αναλογία του συνολικού όγκου των πόρων *n* (πορώδες) ο οποίος είναι κατειλημμένος από κάθε ρευστό. Έχοντας κατά νου ότι το πρόβλημα των πορώδων μέσων αναφέρεται σε εδάφη, θα θεωρήσουμε ότι τα δύο ρευστά που μας απασχολούν είναι το νερό και ο αέρας. Κατ’ αυτόν τον τρόπο θα αναφερόμαστε μόνο σε δύο βαθμούς κορεσμού, αυτόν για το νερό και αυτόν για τον αέρα οι οποίοι είναι ίσοι με S_w και S_a αντίστοιχα, έχοντας όμως πάντα κατά νου ότι τα γραφόμενα ισχύουν για οποιαδήποτε δύο ρευστά

Είναι προφανές ότι αν αμφότερα τα δύο υγρά καταλαμβάνουν τους πόρους του υλικού μας, θα ισχύει πάντα:

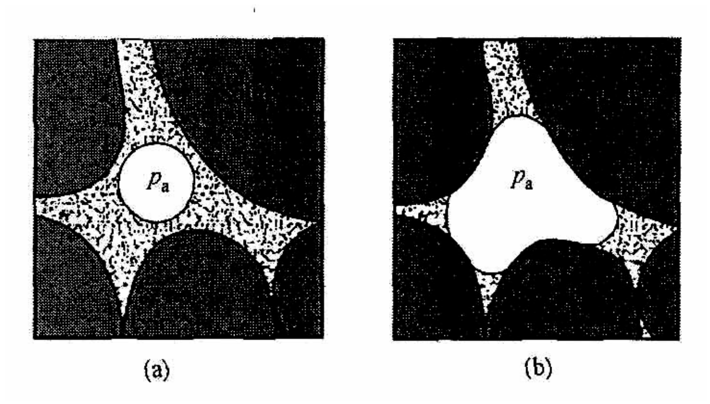
$$S_w + S_a = 1$$

Τα δύο ρευστά μπορεί να έχουν διάφορες επιφάνειες επαφής με τους κόκκους του υλικού όπως μπορεί να δει κανείς και στο σκίτσο 1.2. Έτσι η πίεση πόρων *p* η οποία χρησιμοποιήθηκε στον ορισμό των ενεργών τάσεων είναι ίση με:

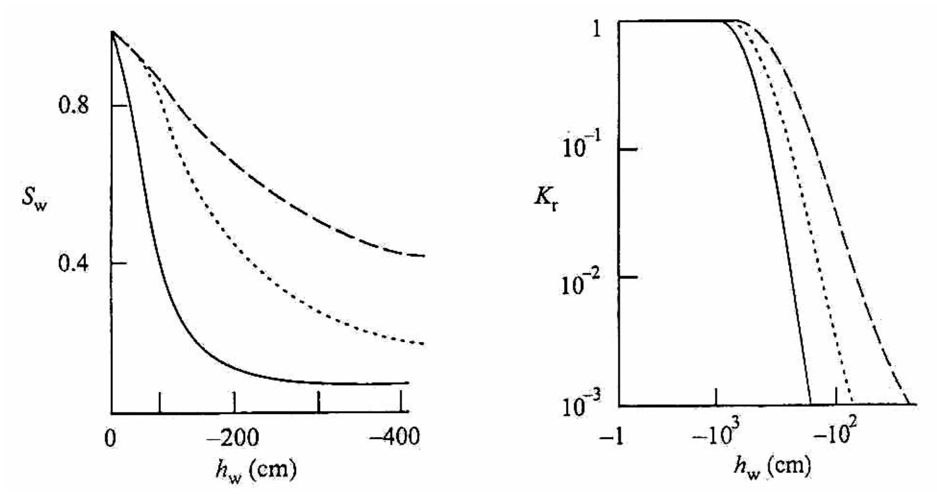
$$p = \chi_w p_w + \chi_a p_a$$

όπου οι συντελεστές χ_w και χ_a αναφέρονται στο νερό και στον αέρα αντίστοιχα και ικανοποιούν την παρακάτω σχέση:

$$\chi_w + \chi_a = 1$$



Σκίτσο 2.3 – Επιφάνεια διεπαφής μιας φυσαλίδας αέρα σε ένα πορώδες μέσο



Σκίτσο 2.4 – Συνάρτηση συσχέτισης κορεσμού και διαπερατότητας σε σχέση με τις πιέσεις πόρων

Χρησιμοποιώντας την προσαυξητική μέθοδο, οι καταστατικές σχέσεις έχουν ως εξής:

$$d\boldsymbol{\sigma}'' = d\boldsymbol{\sigma} + a\mathbf{m}^T dp$$

Για λόγους σαφήνειας, αξίζει να σημειώσουμε ότι:

$$d\mathbf{e} = \begin{bmatrix} d\varepsilon_x \\ d\varepsilon_y \\ d\varepsilon_z \\ d\gamma_{xy} \\ d\gamma_{yz} \\ d\gamma_{zx} \end{bmatrix}$$

Οι προσαυξήσεις των παραμορφώσεων της στερεάς φάσης μπορεί να καθορισθεί με όρους προσαυξητικών μετατοπίσεων du_i ως εξής:

$$d\mathbf{e} = \mathbf{S}d\mathbf{u}$$

με:

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \text{ και } \mathbf{S} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix}$$

Η εξίσωση ισορροπίας των δυνάμεων που ασκούνται σε όλο το υλικό μας (στερεά και υγρή φάση) έχει ως εξής:

$$\mathbf{S}^T\boldsymbol{\sigma} - \rho\ddot{\mathbf{u}} - \underline{\rho_f\left(\dot{\mathbf{w}} + \mathbf{w}\nabla^T\mathbf{w}\right)} + \rho\mathbf{b} = 0 \quad (2.2β)$$

όπου

$$\dot{w}_i \equiv \frac{dw_i}{dt} \text{ και } \ddot{u}_i = \frac{d^2u_i}{dt^2}$$

Η εξίσωση ισορροπίας των δυνάμεων που ασκούνται μόνο στο ρευστό που καταλαμβάνει του πόρους του υλικού μας, κάνοντας χρήση του ιδίου όγκου αναφοράς $dx \cdot dy \cdot dz$ που κάναμε και για τις εξισώσεις (2.2) και λαμβάνοντας κατά νου ότι το ρευστό αυτό κινείται μαζί με τη στερεά φάση, έχει ως εξής:

$$-\nabla p - \mathbf{R} - \rho_f\ddot{\mathbf{u}} - \underline{\frac{\rho_f\left(\dot{\mathbf{w}} + \mathbf{w}\nabla^T\mathbf{w}\right)}{n}} + \rho_f\mathbf{b} = 0 \quad (2.3β)$$

Οι δυνάμεις αντίστασης λόγω του ιξώδους του ρευστού συμβολίζονται με **R** και κάνοντας χρήση του νόμου του Darcy, έχουμε:

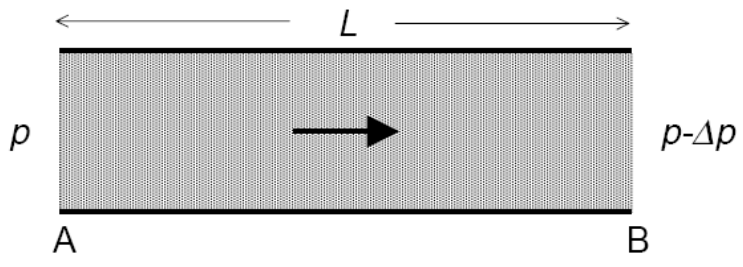
$$\mathbf{kR} = \mathbf{w} \quad (2.4β)$$

Τέλος, η επόμενη εξίσωση είναι εκείνη η οποία μας διασφαλίζει τη διατήρηση της μάζας κατά τη ροή του ρευστού (εξίσωση συνέχειας) και έχει ως εξής:

$$\nabla^T \mathbf{w} + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}}{Q} + n\frac{\dot{\rho}_f}{\rho_f} + \dot{s}_0 = 0 \quad \text{(2.5β)}$$

όπου:

$$\frac{1}{Q} = \frac{n}{K_f} + \frac{a-n}{K_S} \cong \frac{n}{K_f} + \frac{1-n}{K_S}$$



Σκίτσο 2.5 – Ροή κατά Darcy, όπως αυτή προκύπτει λογώ διαφοράς πίεσης μεταξύ των σημείων Α και Β

Για να παράξουμε την πρώτη εξίσωση η οποία θα είναι χωρικά διακριτοποιημένη, πολλαπλασιάζουμε την (3.2) με $\left(\mathbf{N}^u\right)^T$ και ολοκληρώνουμε κατά παράγοντες, οπότε και έχουμε:

$$\int_\Omega \mathbf{B}^T \boldsymbol{\sigma} d\Omega + \left(\int_\Omega \left(\mathbf{N}^u\right)^T \rho \mathbf{N}^u d\Omega \right) \ddot{\overline{\mathbf{u}}} = \mathbf{f}^{(1)}$$

με:

$$\mathbf{f}^{(1)} = \int_\Omega \left(\mathbf{N}^u\right)^T \rho \mathbf{b} d\Omega + \int_{\Gamma_t} \left(\mathbf{N}^u\right)^T \bar{\mathbf{t}} d\Gamma$$

το οποίο το ονομάζουμε και διάνυσμα φόρτισης, έχει διαστάσεις ίδιες με αυτές του διανύσματος $\overline{\mathbf{u}}$ και περιέχει τις δράσεις λόγω καθολικών και επιφανειακών δυνάμεων και με:

$$\mathbf{B} = \mathbf{S}\mathbf{N}^u$$

Κάνοντας χρήση της εξίσωσης (3.3) και λαμβάνοντας υπ' όψιν το φαινόμενο του μερικού κορεσμού, έχω:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}'' - a\chi_w \mathbf{m}^T p$$

Κατ’ αυτόν τον τρόπο, η παραπάνω διακριτοποιημένη συνήθης διαφορική εξίσωση παίρνει την ακόλουθη μορφή:

$$\mathbf{M}\ddot{\overline{\mathbf{u}}} + \int_\Omega \mathbf{B}^{\mathrm{T}} \boldsymbol{\sigma}'' d\Omega - \mathbf{Q}\overline{\mathbf{p}}^w - \mathbf{f}^{(1)} = 0 \quad (3.6)$$

όπου:

$$\mathbf{M} = \int_\Omega \left(\mathbf{N}^u\right)^T \rho \mathbf{N}^u d\Omega$$

είναι το μητρώο μάζας του συστήματος,

$$\mathbf{Q} = \int_\Omega \mathbf{B}^T a\chi_w \mathbf{m} \mathbf{N}^p d\Omega$$

είναι το συνδετικό μητρώο το οποίο συνδέει τις εξισώσεις ισορροπίας με τις εξισώσεις συνέχειας, ενώ:

$$\mathbf{f}^{(1)} = \int_\Omega \left(\mathbf{N}^u\right)^T \rho \mathbf{b} d\Omega + \int_{\Gamma_t} \left(\mathbf{N}^u\right)^T \bar{\mathbf{t}} d\Gamma$$

Ο υπολογισμός των ενεργών τάσεων γίνεται προσαυξητικά, οπότε η εξίσωση (3.4) μπορεί να γραφεί σε διακριτοποιημένη μορφή ως εξής:

$$d\boldsymbol{\sigma}'' = \mathbf{D}\left(\mathbf{B} d\overline{\mathbf{u}} - d\mathbf{e}^0\right) \quad (3.7)$$

Τέλος, διακριτοποιούμε την εξίσωση (3.5) πολλαπλασιάζωντας την με $\left(\mathbf{N}^u\right)^T$ και ολοκληρώνοντας κατά παράγοντες, οπότε και έχουμε:

$$\tilde{\mathbf{Q}}\dot{\overline{\mathbf{u}}} + \mathbf{H}\overline{\mathbf{p}}^w + \tilde{\mathbf{S}}\dot{\overline{\mathbf{p}}}^w - \mathbf{f}^{(2)} = 0 \quad (3.8)$$

όπου:

$$\tilde{\mathbf{Q}} = \int_\Omega \mathbf{B}^T a \mathbf{m} \mathbf{N}^p d\Omega$$

$$\mathbf{H} = \int_\Omega \left(\nabla \mathbf{N}^p\right)^T k \nabla \mathbf{N}^p d\Omega$$

$$\tilde{\mathbf{S}} = \int_\Omega \left(\mathbf{N}^p\right)^T \frac{1}{Q^*} \mathbf{N}^p d\Omega$$

$$\mathbf{f}^{(2)} = -\int_\Omega \left(\nabla \mathbf{N}^p\right)^T k S_w \rho_w \mathbf{b} d\Omega + \int_{\Gamma_t} \left(\mathbf{N}^p\right)^T \overline{\mathbf{q}} d\Gamma$$

με:

$$\frac{1}{Q^*} = C_s + \frac{nS_w}{K_f} + \frac{(a-n)\chi_w}{K_S}$$

όπου οι παράμετροι S_w, k_w και C_s εξαρτώνται από την πίεση p_w.

Έστω ότι:

$$\dot{\upsilon}_{t+\Delta t} = \dot{\upsilon}_t + [(1-\delta)\ddot{\upsilon}_t + \delta\ddot{\upsilon}_{t+\Delta t}]\Delta t \quad (3.10)$$

$$\upsilon_{t+\Delta t} = \upsilon_t + \dot{\upsilon}_t\Delta t + \left[\left(\frac{1}{2}-\alpha\right)\ddot{\upsilon}_t + \alpha\ddot{\upsilon}_{t+\Delta t}\right]\Delta t^2 \quad (3.11)$$

όπου α και δ είναι παράμετροι οι οποίες καθορίζονται ανάλογα με το πρόβλημα για ακρίβεια και σταθερότητα της μεθόδου. Αν $\alpha = \frac{1}{4}$ και $\delta = \frac{1}{2}$, τότε σύμφωνα με εργασίες του Newmark (1956) η μέθοδος είναι πάντα σταθερή και ονομάζεται μέθοδος της σταθερής-μέσης επιτάχυνσης (constant-average-acceleration method).

Κάνοντας χρήση της εξίσωσης (3.9) για το χρονικό σημείο $t + \Delta t$ έχουμε:

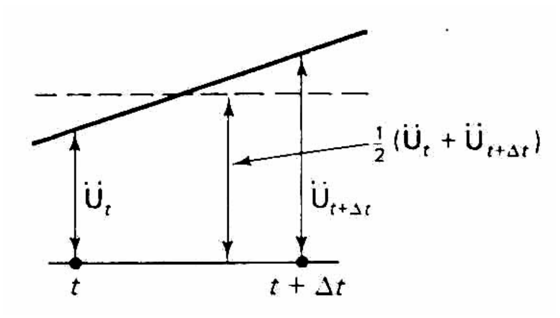
$$\widehat{\mathbf{M}}\ddot{\boldsymbol{\upsilon}}_{t+\Delta t} + \widehat{\mathbf{C}}\dot{\boldsymbol{\upsilon}}_{t+\Delta t} + \widehat{\mathbf{K}}\boldsymbol{\upsilon}_{t+\Delta t} = \mathbf{R}_{t+\Delta t} \quad (3.12)$$

Λύνοντας τις εξισώσεις (3.10) και (3.11) ως προς τις χρονικές παραγώγους του **υ**, μπορούμε να βρούμε τη σχέση που συνδέει τις χρονικές παραγώγους του **υ** με το **υ**. Κάνοντας χρήση των παραπάνω σχέσεων στην εξίσωση (3.12), μπορούμε να λύσουμε ως προς $\boldsymbol{\upsilon}_{t+\Delta t}$ και μετά, με γνωστή αυτή την ποσότητα, να βρούμε τις ποσότητες $\dot{\boldsymbol{\upsilon}}_{t+\Delta t}$ και $\ddot{\boldsymbol{\upsilon}}_{t+\Delta t}$.

Παρακάτω, παρατίθεται ο αλγόριθμος Newmark ο οποίος έχει ως εξής:

- Επιλέγουμε $\boldsymbol{\upsilon}_0$, $\dot{\boldsymbol{\upsilon}}_0$ και $\ddot{\boldsymbol{\upsilon}}_0$, το χρονικό βήμα Δt και τις παραμέτρους α και δ ούτως ώστε $\delta \geq \frac{1}{2}$ και $\alpha \geq \frac{1}{4}\left(\frac{1}{2}+\delta\right)^2$.
- Υπολογίζουμε τις παρακάτω σταθερές: $\alpha_0 = \frac{1}{\alpha\Delta t^2}$, $\alpha_1 = \frac{\delta}{\alpha\Delta t}$, $\alpha_2 = \frac{1}{\alpha\Delta t}$, $\alpha_3 = \frac{1}{2\alpha} - 1$, $\alpha_4 = \frac{\delta}{\alpha} - 1$, $\alpha_5 = \frac{\Delta t}{2}\left(\frac{\delta}{\alpha} - 2\right)$, $\alpha_6 = \Delta t(1-\delta)$, $\alpha_7 = \delta\Delta t$
- Μορφώνουμε το ενεργό μητρώο δυσκαμψίας το οποίο είναι ίσο με: $\breve{\mathbf{K}} = \alpha_0\widehat{\mathbf{M}} + \alpha_1\widehat{\mathbf{C}} + \widehat{\mathbf{K}}$

- Για κάθε χρονικό βήμα υπολογίζουμε τις ενεργές δράσεις στο χρονικό σημείο $t + \Delta t$ οι οποίες είναι ίσες με: $\breve{\mathbf{R}}_{t+\Delta t} = \mathbf{R}_{t+\Delta t} + \widehat{\mathbf{M}}(\alpha_0 \boldsymbol{\upsilon}_t + \alpha_2 \dot{\boldsymbol{\upsilon}}_t + \alpha_3 \ddot{\boldsymbol{\upsilon}}_t) + \widehat{\mathbf{C}}(\alpha_1 \boldsymbol{\upsilon}_t + \alpha_4 \dot{\boldsymbol{\upsilon}}_t + \alpha_5 \ddot{\boldsymbol{\upsilon}}_t)$, επιλύουμε το σύστημα $\breve{\mathbf{K}}\boldsymbol{\upsilon}_{t+\Delta t} = \breve{\mathbf{R}}_{t+\Delta t}$ και υπολογίζουμε τις χρονικές παραγώγους από τις σχέσεις: $\ddot{\boldsymbol{\upsilon}}_{t+\Delta t} = \alpha_0 (\boldsymbol{\upsilon}_{t+\Delta t} - \boldsymbol{\upsilon}_t) - \alpha_2 \dot{\boldsymbol{\upsilon}}_t - \alpha_3 \ddot{\boldsymbol{\upsilon}}_t$ και $\dot{\boldsymbol{\upsilon}}_{t+\Delta t} = \dot{\boldsymbol{\upsilon}}_t + \alpha_6 \ddot{\boldsymbol{\upsilon}}_t + \alpha_7 \ddot{\boldsymbol{\upsilon}}_{t+\Delta t}$.



Σκίτσο 2.6 – Η σταθερή-κατά μέσο όρο μέθοδος επιτάχυνσης

Οι πιο αποτελεσματικές άμεσες μέθοδοι επίλυσης βασίζονται στην κλασική μέθοδο της απαλοιφής του Gauss. Πρακτικά, αναδιατάσσοντας τις πράξεις της απαλοιφής του Gauss, λαμβάνονται οι διάφορες μέθοδοι. Έτσι, διαχωρίζοντας τις πράξεις που σχετίζονται με το δεξιό μέλος του προς επίλυση γραμμικού συστήματος από τις υπόλοιπες πράξεις προκύπτει η μέθοδος Cholesky.

Η μέθοδος Cholesky χρησιμοποιείται ευρέως στην Υπολογιστική Μηχανική και εφαρμόζεται επανειλημμένα στην παρούσα διατριβή. Το πρώτο στάδιο της μεθόδου αυτής αφορά τις πράξεις της απαλοιφής του Gauss που είναι ανεξάρτητες από το διάνυσμα του δεξιού μέλους και ισοδυναμούν με την ανάλυση του μητρώου των συντελεστών σε γινόμενο παραγόντων. Αναφορικά με το σύστημα εξισώσεων, παραγοντοποιείται το μητρώο συντελεστών A ως εξής:

$$A = LL^T$$

Μια άλλη αντιμετώπιση είναι η παραγοντοποίηση ως εξής:

$$A = LDL^T$$

Σε αυτή την περίπτωση η επίλυση γίνεται ως εξής:

- Υπολογισμός του ενδιάμεσου διανύσματος x1 μέσω μιας εμπρός αντικατάστασης:

$$Lx_1 = b \Leftrightarrow x_1 = L^{-1}b$$

- Υπολογισμός του ενδιάμεσου διανύσματος x2:

$$Dx_2 = x_1 \Leftrightarrow x_2 = D^{-1}x_1$$

- Υπολογισμός του άγνωστου διανύσματος x με μία πίσω αντικατάσταση:

$$L^T x = x_2 \Leftrightarrow x = L^{-T} x_2$$

Στην Υπολογιστική Μηχανική και ειδικότερα στην Υπολογιστική Μηχανική των Κατασκευών έχουν εφαρμοστεί πολλές επαναληπτικές μέθοδοι επίλυσης συστημάτων στο παρελθόν. Αναφέρουμε χαρακτηριστικά τις μεθόδους Jacobi, Gauss Seidel, διαδοχικής υπερχαλάρωσης (successive overrelaxation –SOR), απότομης καθόδου (steepest descent) και συζυγών διανυσματικών κλίσεων (conjugate gradient method – CG). Στην παρούσα παράγραφο θα παρουσιάσουμε την μέθοδο των συζυγών διανυσματικών κλίσεων με προσταθεροποίηση (ή αλλιώς την προσταθεροποιημένη μέθοδο των συζυγών διανυσματικών κλίσεων ή σε αγγλική ορολογία την preconditioned conjugate gradient method – PCG. Η μέθοδος PCG αφορά την επίλυση γραμμικών όπου το μητρώο A είναι συμμετρικό και θετικά ημιορισμένο (positive semidefinite). Ένα μητρώο ονομάζεται θετικά ημιορισμένο, αν οι ιδιοτιμές του είναι μεγαλύτερες ή ίσες από το μηδέν.

Σε περίπτωση συστημάτων κακής κατάστασης και γενικώς για περιπτώσεις μεγάλων σφαλμάτων στρογγυλοποίησης, ο αλγόριθμος PCG μπορεί να εφαρμοστεί με επανορθογωνοποίηση.

Αρχικοποίηση: $r^0 = b - Ax^0,\, z^0 = \tilde{A}^{-1}r^0,\, p^0 = z^0,\; q^0 = Ap^0,\; \eta^0 = \dfrac{p^{0^T} r^0}{p^{0^T} q^0}$

Επανάληψη για k=1,2,… μέχρι σύγκλισης:

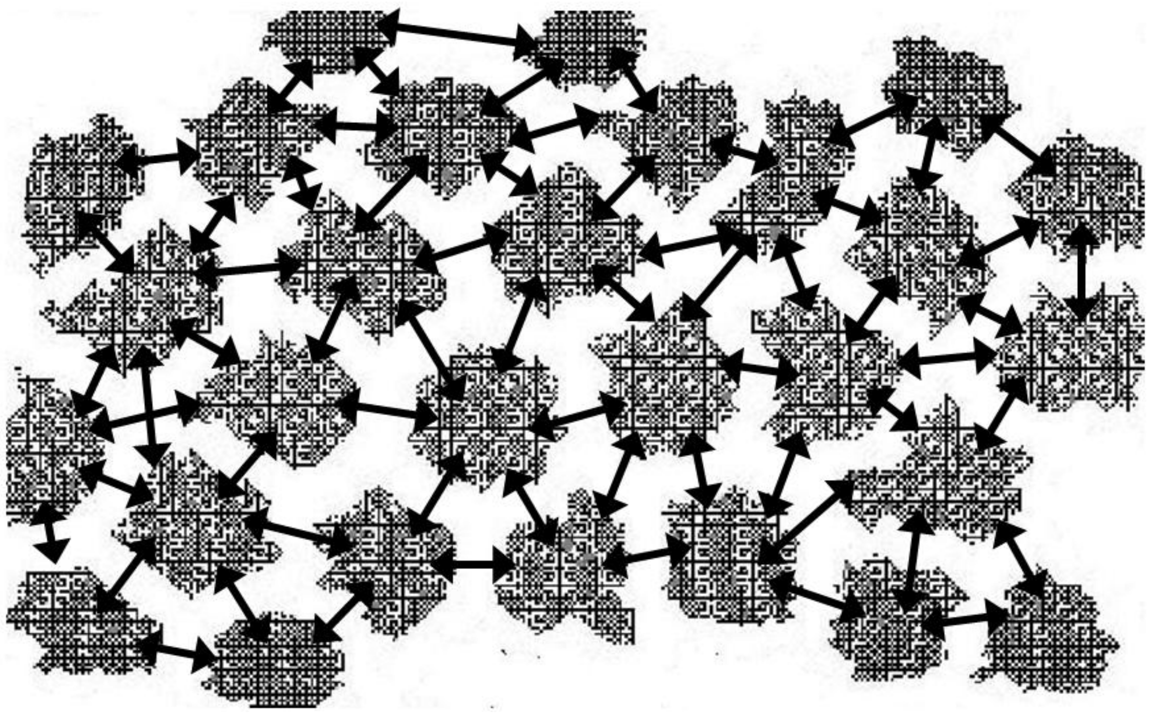
Εκτίμηση λύσης		$x^k = x^{k-1} + \eta^{k-1} p^{k-1}$
Υπολειμματικό διάνυσμα		$r^k = r^{k-1} - \eta^{k-1} q^{k-1}$
Προσταθεροποιημένο υπολειμματικό διάνυσμα		$z^k = \tilde{A}^{-1} r^k$
Διάνυσμα κατεύθυνσης	Απλή εκτίμηση	$p^k = z^k + \dfrac{z^{k^T} r^k}{z^{k-1^T} r^{k-1}} p^{k-1}$
	Επανορθογωνοποίηση	$p^k = z^k - \sum_{i=0}^{k-1} \dfrac{z^{k^T} q^i}{p^{i^T} q^i} p^i$
Γινόμενο διανύσματος κατεύθυνσης με το A		$q^k = Ap^k$

Υπολογισμός βήματος η	Απλή εκτίμηση	$\eta^k = \dfrac{z^{k^T} r^k}{p^{k^T} q^k}$
	Επανορθογωνοποίηση	$\eta^k = \dfrac{p^{k^T} r^k}{p^{k^T} q^k}$

Table 4.1 - The PCG algorithm

Στο χώρο της Υπολογιστικής Μηχανικής των Κατασκευών, οι DDM γνώρισαν σημαντική ανάπτυξη από την αρχή της δεκαετίας του ΄90. Την περίοδο εκείνη, η πιο δημοφιλής μέθοδος ήταν η μέθοδος του συμπληρώματος Schur (Schur complement method). Η μέθοδος αυτή φέρει δύο ακόμα ονόματα: μέθοδος Neumann-Neumann (Neumann-Neumann method) και πρωτογενής μέθοδος επίλυσης με υποφορείς (Primal Substructuring Method – PSM). Για την ακρίβεια, την τελευταία ονομασία την έλαβε αργότερα, όταν αναπτύχθηκε η δυϊκή μέθοδος επίλυσης με υποφορείς (Dual Substructuring Method – DSM). Η ονομασία «Πρωτογενής μέθοδος επίλυσης με υποφορείς (PSM)» οφείλεται στις διαφορές της μεθόδου αυτής με τη μέθοδο που φέρει το όνομα «Δυϊκή μέθοδος επίλυσης με υποφορείς (DSM)».

Στα πλαίσια της PSM, τα τοπικά προβλήματα υποφορέων αντιμετωπίζο νται με μία άμεση μέθοδο επίλυσης όπως η μέθοδος Cholesky, ενώ το καθολικό πρόβλημα του συνόρου μεταξύ των υποφορέων (subdomain interface) επιλύεται με μία επαναληπτική μέθοδο. Συνήθως, για το σκοπό αυτό χρησιμοποιείται η μέθοδος PCG, επειδή τα γραμμικά προβλήματα που απαντώνται στην ανάλυση κατασκευών με DDM είναι κατά κανόνα θετικά ορισμένα.



Σκίτσο 4.1 – Ένας δομητικός φορέας, διαχωρισμένος σε υποφορείς. Τα βέλη υποδεικνύουν τις δυνάμεις αλληλεπίδρασης μεταξύ των αποσυνδεδεμένων υποφορέων

Επίσης, στις αρχές της δεκαετίας του ΄90, ήταν ήδη γνωστό ότι οι DDM χρειάζονται ένα μηχανισμό, ικάνο να εξασφαλίσει μια καθολική ανταλλαγή πληροφορίας μεταξύ μακρινών υποφορέων σε κάθε επανάληψη (ακολουθώντας ανάλογη λογική με εκείνη που υποδεικνύει ότι απαιτείται ένα πρόβλημα αραιού πλέγματος (coarse-grid problem) στις μεθόδους πολλαπλών πλεγμάτων (multigrid methods). Εξάλλου, την εποχή εκείνη είχε γίνει αντιληπτό ότι ο αριθμός επαναλήψεων της PSM αυξάνεται σημαντικά όταν αυξάνεται ο αριθμός υποφορέων ενός προβλήματος. Η αύξηση οφείλεται στην έλλειψη επικοινωνίας μεταξύ μακρινών υποφορέων σε κάθε επανάληψη της μεθόδου αυτής.

Το μειονέκτημα αυτό των DDM της εποχής εκείνης διορθώθηκε το 1991, όταν οι Farhat και Roux εισήγαγαν τη μέθοδο FETI (Finite Element Tearing and Interconnecting method). Η μέθοδος FETI ήταν η πρώτη δημοφιλής μέθοδος που έφερε ένα μηχανισμό καθολικής διάδοσης πληροφορίας, ή κατά την ορολογία των DDM, ένα πρόβλημα αραιού πλέγματος (coarse-grid problem) ή αλλιώς, ένα αραιό πρόβλημα (coarse problem). Η ορολογία αυτή είναι δανεισμένη από τις μεθόδους πολλαπλών πλεγμάτων, όπου χρησιμοποιούνται αραιά πλέγματα (coarse grids) για τη μείωση του αριθμού επαναλήψεων. Τα αραιά προβλήματα χρησιμοποιούνται κατά παρόμοιο τρόπο στις DDM, αλλά δεν βασίζονται σε αραιά πλέγματα, καθότι οι συνήθεις DDM χρησιμοποιούν ένα μόνο πλέγμα, αυτό δηλαδή που ορίζεται ως συνήθως από το χρήστη. Τα αραιά προβλήματα των DDM προκύπτουν αυτόματα από τις ίδιες τις εξισώσεις των μεθόδων και αποτελούν απλά ένα γραμμικό πρόβλημα που, αφενός έχει μικρή σχετικά διάσταση σε σχέση με τη διάσταση του συνολικού προβλήματος, αφετέρου εξασφαλίζει την άμεση ανταλλαγή πληροφορίας μεταξύ μακρινών υποφορέων σε κάθε επανάληψη.

Η μέθοδος FETI έλαβε επίσης την ονομασία δυϊκή μέθοδος επίλυσης με υποφορείς (Dual Substructuring Method – DSM), λόγω μιας βασικής της διαφοράς με την PSM. Συγκεκριμένα, σε αντίθεση προς την PSM, της οποίας το συνοριακό πρόβλημα (interface problem) έχει ως άγνωστές τις μετατοπίσεις των κόμβων του συνόρου μεταξύ των υποφορέων (interface displacements), η FETI διαθέτει ένα συνοριακό πρόβλημα που έχει ως άγνωστες τις δυνάμεις αλληλεπίδρασης μεταξύ των υποφορέων (subdomain interaction forces). Επειδή λοιπόν οι συνοριακές αυτές δυνάμεις χαρακτηρίζονται ως δυϊκές μεταβλητές (dual variables) ως προς τις συνοριακές μετατοπίσεις, η FETI έλαβε την επιπλέον ονομασία «Δυϊκή μέθοδος επίλυσης με υποφορείς (Dual Substructuring Method – DSM)». Γενικά, όλες οι μέθοδοι των οποίων το συνοριακό πρόβλημα εκφράζεται ως προς τις συνοριακές μετατοπίσεις ή τις συνοριακές δυνάμεις χαρακτηρίζονται ως πρωτογενείς (primal) ή δυϊκές (dual) μέθοδοι αντίστοιχα. Ας σημειωθεί επίσης ότι στην ορολογία των DDM οι συνοριακές δυνάμεις αλληλεπίδρασης μεταξύ των υποφορέων ονομάζονται κοινώς ως «πολλαπλασιαστές Lagrange», διότι στη μαθηματική διατύπωση των DDM ως μεθόδων ελαχιστοποίησης μιας συνάρτησης δυναμικού, οι δυνάμεις αλληλεπίδρασης υπεισέρχονται ως πολλαπλασιαστές Lagrange που επιβάλλουν τη συνθήκη συμβιβαστού των μετατοπίσεων στο σύνορο μεταξύ των υποφορέων.

Επιπλέον, ένα ιδιαίτερο χαρακτηριστικό της μεθόδου FETI είναι η χρήση των κινήσεων μηδενικής ενέργειας (zero energy modes) των υποφορέων για το σχηματισμό του αραιού της προβλήματος. Ως κινήσεις μηδενικής ενέργειας μιας κατασκευής ορίζονται οι κινήσεις εκείνες που δεν προκαλούν ένταση της κατασκευής, οι κινήσεις εκείνες δηλαδή που πραγματοποιούνται με μηδενικό έργο. Ας υποθέσουμε για παράδειγμα την κίνηση στο χώρο ένος υποφορέα ο οποίος δεν έχει καμία εξωτερική στήριξη. Αν αγνοήσουμε τη σύνδεση του υποφορέα αυτού με τους γειτονικούς του υποφορείς, τότε οι πιθανές γραμμικά ανεξάρτητες κινήσεις μηδενικής ενέργειας του υποφορέα αυτού περιλαμβάνουν τις 6 ανεξάρτητες κινήσεις του στο χώρο ως στερεό σώμα (rigid body modes), καθώς και τους όποιους εσωτερικούς μηχανισμούς διαθέτει. Γενικά, οι κινήσεις μηδενικής ενέργειας των υποφορέων, οι οποίες χρησιμοποιήθηκαν για τη διατύπωση της FETI με ιδιαίτερη επιτυχία, αποτέλεσαν επίσης τη βάση για πολλές άλλες DDM που προτάθηκαν αργότερα.

Αν u και f ειναι τα διανύσματα μετατόπισης και φόρτισης σε όλον το φορέα και u^s και f^s ειναι διανύσματα που αναφέρονται στις αντίστοιχες ποσότητες σε όλον το φορέα τότε:

$$u^s = \begin{bmatrix} u^{(1)^T} & \dots & u^{(N_s)^T} \end{bmatrix}^T$$

$$f^s = \begin{bmatrix} f^{(1)^T} & \dots & f^{(N_s)^T} \end{bmatrix}^T$$

$$u^s = Lu$$

$$f = L^T f^s$$

με N_s τον αριθμό των υποφορέων και L ο τελεστής απεικόνισης καθολικού/τοπικου ο οποίος είναι ένα μητρώο Boolean.

Εφαρμόζωντας τις εξισώσεις. (4.4.1)-(4.4.4), στους β.ε. της διεπαφής έχουμε:

$$u_b^s = \left[u_b^{(1)^T} \quad \ldots \quad u_b^{(Ns)^T} \right]^T$$

$$f_b^s = \left[f_b^{(1)^T} \quad \ldots \quad f_b^{(Ns)^T} \right]^T$$

$$u_b^s = L_b u_b$$

$$f_b = L_b^{\ T} f_b^s$$

Οι δυνάμεις αλληλεπίδρασης μεταξύ των κόμβων διεπαφής είναι ίσες με:

$$t = f^s - B^T \lambda$$

ή

$$t_b = f_b^s - B_b^{\ T} \lambda$$

Επίσης ισχύει οτι:

$$Bu_s = \begin{bmatrix} B^{(1)} & \cdots & B^{(Ns)} \end{bmatrix} \begin{bmatrix} u^{(1)} \\ \vdots \\ u^{(Ns)} \end{bmatrix} = 0$$

ή

$$B_b u_b^s = \begin{bmatrix} B_b^{(1)} & \cdots & B_b^{(Ns)} \end{bmatrix} \begin{bmatrix} u_b^{(1)} \\ \vdots \\ u_b^{(Ns)} \end{bmatrix} = 0$$

Στην περίπτωση τοπικών προβλημάτων, πρέπει να λυθούν οι παρακάτω εξισώσεις:

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

ενώ για την περίπτωση πρωτογενών μεθόδων υποφορέων έχουμε:

$$S^{(s)}u_b^{(s)} = \hat{f}_b^{(s)} - B_b^{(s)^T}\lambda$$

Αν αναδιατάξουμε το μητρώο K ούτως ώστε:

$$\begin{bmatrix} K_{bb}^{(s)} & K_{bi}^{(s)} \\ K_{ib}^{(s)} & K_{ii}^{(s)} \end{bmatrix}\begin{bmatrix} u_b^{(s)} \\ u_i^{(s)} \end{bmatrix} = \begin{bmatrix} f_b^{(s)} \\ f_i^{(s)} \end{bmatrix} - \begin{bmatrix} B_b{}^T \\ 0 \end{bmatrix}\lambda$$

τότε:

$$S^{(s)} = K_{bb}^{(s)} - K_{bi}^{(s)}\left(K_{ii}^{(s)}\right)^{-1}K_{ib}^{(s)}$$

$$\hat{f}_b^{(s)} = f_b^{(s)} - K_{bi}^{(s)}\left(K_{ii}^{(s)}\right)^{-1} f_i^{(s)}$$

Για την πρωτογενή μέθοδο PSM, πρέπει να λυθούν οι παρακάτω εξισώσεις:

$$\hat{S}u_b = \hat{f}_b$$

με

$$\hat{S} = L_b^T S^s L_b,\ \hat{f}_b = L_b^T \hat{f}_b^s$$

$$\hat{f}_b^s = \begin{bmatrix} \hat{f}_b^{(1)^T} & \cdots & \hat{f}_b^{(Ns)^T} \end{bmatrix}^T$$

$$S^s = \begin{bmatrix} S^{(1)} & & \\ & \ddots & \\ & & S^{(Ns)} \end{bmatrix}$$

Ως προσταθεροποιητή, χρησιμοποιείται η παρακάτω έκφραση:

$$\tilde{A}^{-1} = L_{p_b}^T S^{s^+} L_{p_b}$$

η οποία υλοποιείται ως:

$$\tilde{A}^{-1} = L_{p_b}^T N_b^s K^{s^+} N_b^{s^T} L_{p_b}$$

Στην περίπτωση της δυϊκής μεθόδου, έχουμε το εξής σύστημα:

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

με

$$F_I = BK^{s^+}B^T,\ G = BR^s,\ d = BK^{s^+} f^s,\ e = R^{s^T} f^s$$

Για την αποσύζευξη του παραπάνω συστήματος χρησιμοποιούμε τον προβολέα:

$$P = I - QG\left(G^T QG\right)^{-1} G^T$$

Έτσι τελικά, πρέπει να λυθούν τα παρακάτω αποσυζευγμένα συστήματα:

$$P^T F_I \lambda = P^T d$$

$$G^T a = e$$

Στην περίπτωση της πρωτογενούς-δυϊκής μεθόδου λύνουμε τις εξισώσεις της πρωτογενούς μεθόδου των υποφορέων με τον εξής προσταθεροποιητή:

$$\tilde{A}^{-1} = L_{p_b}^T H_b^T S^{s^+} H_b L_{p_b}$$

με

$$H_b = I - B_b^T QG\left(G^T QG\right)^{-1} R_b^{sT}$$

Στην περίπτωση δυναμικών προβλημάτων, η δυική μέθοδος των υποφορέων εφαρμόζεται για τη λύση του:

$$F_I \lambda = d$$

ενώ στην περίπτωση εισαγωγής ενός τεχνητού αραιού προβλήματος έχουμε:

$$PF_I \lambda = Pd$$

με

$$P = I - C\left(C^T F_I C\right)^{-1} C^T F_I$$

Στην αντίστοιχη πρωτογενής-δυϊκή υλοποιηση, λύνουμε τις εξισώσεις της πρωτογενούς μεθόδου των υποφορέων με τον εξής προσταθεροποιητή:

$$\tilde{A}^{-1} = L_{p_b}^T \left(S^{s^{-1}} - S^{s^{-1}} B_b^T C\left(C^T F_I C\right)^{-1} C^T B_b S^{s^{-1}}\right) L_{p_b}$$

Για την περίπτωση πορωδών μέσων μπορούμε να χρησιμοποιήσουμε, είτε ένα αραιό πρόβλημα το οποίο σχετίζεται με τον στερεό σκελετό:

$$C = QG_I$$

με:

$$G_I = \left[B^{(1)} R^{(1)} \quad \cdots \quad B^{(Ns)} R^{(Ns)} \right]$$

$$R^{(s)} = \mathrm{null}\left(\widehat{K}^{(s)} \right)$$

είτε με ένα αραιό πρόβλημα το οποίο σχετίζεται με την διαπερατότητα του πορώδους μέσου:

$$C = E_I$$

με

$$E_I = \left[B^{(1)} {R_H}^{(1)} \quad \cdots \quad B^{(Ns)} {R_H}^{(Ns)} \right]$$

$${R_H}^{(s)} = \mathrm{null}\left(\widehat{H}^{(s)} \right)$$

Οι αντίστοιχοι προσταθεροποιητές για την πρωτογενή-δυϊκή υλοποίηση είναι:

$$\tilde{A}^{-1} = L_{p_b}^T \left(S^{s^{-1}} - S^{s^{-1}} B_b^T QG \left(G^T Q^T F_I QG \right)^{-1} G^T Q^T B_b S^{s^{-1}} \right) L_{p_b}$$

και

$$\tilde{A}^{-1} = L_{p_b}^T \left(S^{s^{-1}} - S^{s^{-1}} B_b^T E_I \left({E_I}^T F_I E_I \right)^{-1} {E_I}^T B_b S^{s^{-1}} \right) L_{p_b}$$

Η αντικειμενοστραφής τεχνολογία προγραμματισμού βασίζεται σε ένα στέρεο μηχανιστικό οικοδόμημα, στοιχεία του οποίου ομαδικά καλούμε αντικειμενοστραφές μοντέλο. Το αντικειμενοστραφές μοντέλο προωθεί και χρησιμοποιεί τις αρχές της άρθρωσης (modularity), της αφαίρεσης (abstraction), της απομόνωσης (encapsulation) και της ιεραρχίας (hierarchy). Από μόνες τους, αυτές οι αρχές δεν είναι νέες ή καινοτομικές-αυτό το οποίο είναι σημαντικό όμως είναι το γεγονός οτι τα παραπάνω στοιχεία ενώνονται με έναν συνεργικό τρόπο

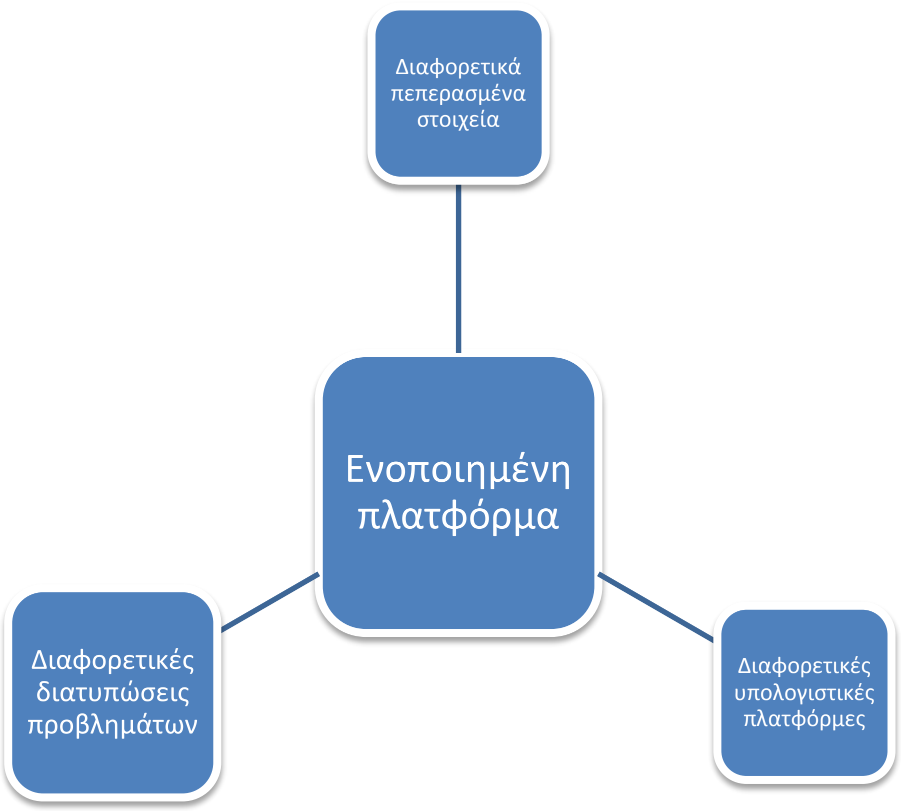
Θα πρέπει να αναφέρουμε οτι ο αντικειμενοστραφής τρόπος ανάλυσης και σχεδιασμού είναι εκ θεμελίων διαφορετικές από τον παραδοσιακό δομημένο προγραμματισμο (structured programming) και τους τρόπους ανάλυσης στους οποίους αυτός βασίζεται. Θα λέγαμε πως απαιτεί ένα διαφορετικό τρόπο σκέψης όσον αφορά την αποδόμηση του προς λύση προβλήματος, γεγονός το οποίο παράγει λογισμικά οικοδομήματα τα οποία απέχουν κατά πολύ από αυτά της σχολής του δομημένου προγραμματισμού.

Συνήθως κατά την πραγμάτωση κωδίκων πεπερασμένων στοιχείων σε ερευνητικό επίπεδο, παράγονται κάποια προγράμματα τα οποία υλοποιούν συγκεκριμένες μεθόδους επίλυσης προβλημάτων. Επειδή ο σκοπός είναι καθαρά ερευνητικός, δεν δίνεται έμφαση στο πλήθος των ειδών πεπερασμένων στοιχείων τα οποία χρησιμοποιεί το πρόγραμμα, ούτε και στα είδη των προβλημάτων τα οποία επιλύονται. Αντίθετα, κατασκευάζονται, όσο πιο απλά γίνεται, κάποια πεπερασμένα στοιχεία και λύνουμε με αυτούς τους επιλύτες στατικά προβλήματα αφού αυτά είναι τα πιο έυκολα υλοιποίησημα. Πραγματοποιούμε δηλαδή μια υποδομή (infrastructure) τέτοια που να εξυπηρετεί απλά τις ανάγκες μας σε δοκιμαστικό επίπεδο. Μάλιστα, επειδή οι μέθοδοι επίλυσης που υλοποιούνται είναι πολύ διαφορετικές μεταξύ τους, η προαναφερθείσα υποδομή χρειάζεται να επαναδημιουργηθεί ή να αλλάξει ριζικά για να ταιριάζει στις ανάγκες ενός νέου κώδικα που θα υλοποιεί μια άλλη μέθοδο επίλυσης.

Τα πράγματα περιπλέκονται ακόμα περισσότερο όταν μιλάμε για επιλύτες οι οποίοι λειτουργούν σε περιβάλλον πολυεπεξεργασίας. Υπάρχουν διάφορα πρωτόκολλα τα οποία υλοποιούν την πολυεπεξεργασία (δυο εκ των οποίων είναι το MPI και PVM) και τα οποία έχουν διαφορετικούς τρόπους υλοποίησης και λειτουργίας. Η απόδοση των πρωτοκόλλων αυτών έχει να κάνει μεταξύ άλλων και με τα μηχανήματα τα οποία χρησιμοποιούνται ενώ πολλές φορές χρειάζεται να συνδεθούν μηχανήματα με διαφορετικές πλατφόρμες τα οποία θα συνεργάζονται και θα ανταλλάσσουν δεδομένα. Τέλος τα δεδομένα τα οποία ανταλλάσσονται μεταξύ των μηχανημάτων αλλάζουν ανάλογα με τη φύση του επιλύτη, οπότε η υποδομή που χρειάζεται για την ανταλλαγή μηνυμάτων αλλάζει για να ταιριάζει στις ανάγκες ενός νέου κώδικα που θα υλοποιεί μια άλλη μέθοδο επίλυσης.

Παρατηρούμε λοιπόν οτί στα πλαίσια της έρευνας, πολύτιμος χρόνος και πόροι καταναλώνονται για την κατασκευή υποδομής η οποία θα υποστηρίξει τον κυρίως κώδικα ενώ παράλληλα πολλά κοινά υποπρογράμματα τα οποία χρησιμοποιούνται από επιλύτη σε επιλύτη χρειάζεται να μεταγραφούν ή να ξαναγραφούν για να ταιριάξουν με τις ανάγκες του εκάστοτε κώδικα.

Στόχος λοιπόν αυτού του κώδικα είναι να ενοποιήσει τις παραπάνω υποδομές και κώδικες ούτως ώστε η συγγραφή νέων επιλυτών να παίρνει τον λιγότερο δυνατό χρόνο.


Διαφορετικά πεπερασμένα στοιχεία
Ενοποιημένη πλατφόρμα
Διαφορετικές διατυπώσεις προβλημάτων
Διαφορετικές υπολογιστικές πλατφόρμες

Σκίτσο 5.3 – Μια ενοποιημένη πλατφόρμα πεπερασμένων στοιχείων

Η άρθρωση, η οποία είναι η κατάτμηση ενός προγράμματος σε επιμέρους μέρη, μπορεί να μειώσει την πολυπλοκότητα ενός συστήματος ως ένα αρκετά μεγάλο βαθμό. Παρ' όλο που η κατάτμηση ενός προγράμματος βοηθάει πολύ για αυτό το λόγο, ένας πιο ισχυρός λόγος για την κατάτμηση ενός προγράμματος είναι οτι δημιουργεί έναν αριθμό καλά προσδιορισμένων και επεξηγημένων ορίων μέσα στο πρόγραμμα. Αυτά τα όρια είναι ανεκτίμητης αξίας για την κατανόησή του.

Το να αποφασίζει κανείς το σωστό σύνολο των αρθρωμάτων για ένα δεδομένο πρόβλημα είναι μια πάρα πολύ δύσκολη διαδικασία. Αυτό συμβαίνει διότι η λύση μπορεί να μην είναι γνωστή κατά τη διαδικασία σχεδίασης και για αυτό η αποδόμηση σε μικρότερα αρθρώματα να είναι αρκετά δύσκολη.

Πρακτικά, τα αρθρώματα εξυπηρετούν ως φυσικοί υποδοχείς μέσα στους οποίους δηλώνουμε τις κλάσεις μας και τα αντικείμενά μας (classes and objects) του λογικού μας σχεδιασμού. Η έλλειψη τυποποιημένων αρθρωμάτων για τη λύση του προβλήματός μας, δίνει στον προγραμματιστή σαφώς περισσότερους βαθμούς ελευθερίας στο πώς θα κατασκευάσει τα αρθρώματά του. Έτσι, σε μικρά προβλήματα, ο προγραμματιστής μπορεί να επιλέξει να δηλώσει κάθε κλάση και κάθε αντικείμενο σε ένα άρθρωμα. Παρ' όλα αυτά, στις περισσότερες περιπτώσεις μια καλύτερη λύση θα ήταν να ομαδοποιήσουμε τις συσχετιζόμενες κλάσεις και αντικείμενα στο ίδιο άρθρωμα και να εκθέσουμε (να κάνουμε δηλαδή φανερά) μόνο εκείνα τα στοιχεία τα οποία τα οποία πρέπει να είναι φανερά για τη λειτουργία των υπόλοιπων αρθρωμάτων τα οποία απαρτίζουν το πρόγραμμά μας.

Στον παραδοσιακό δομημένο προγραμματισμό, η άρθρωση νοείται κυρίως με μια σημασίας ομαδοποίηση των υποπρογραμμάτων, χρησιμοποιώντας τα κριτήρια της σύνδεσης και της λογικής συνέχειας. Στον αντικειμενοστραφή προγραμματισμό, το πρόβλημα είναι κάπως διαφορετικο: αυτό που πρέπει να γίνει είναι να ομαδοποιήσουμε τις κλάσεις και τα αντικείμενα της λογικής μας δομής τα οποία είναι προφανώς διαφορετικης φύσεως από τα υποπρογράμματα.

Η εμπειρία υποδεικνύει οτι υπάρχουν αρκετές τεχνικής και μη φύσεως οδηγίες τις οποίες μπορεί να ακολουθήσει κανείς ούτως ώστε να επιτύχει μια ευφυή άρθρωση των κλάσεων και των αντικειμένων. Άλλωστε ο ευρύτερος στόχος της αποδόμησης ενός προγράμματος σε αρθρώματα είναι η μείωση του κόστους ανάπτυξης, επιτρέποντας το σχεδιασμό και τη διόρθρωση αυτών κατά ανεξάρτητο τρόπο. Η δομή κάθε αρθρώματος θα πρέπει να είναι αρκετά απλή σε σύλληψη ούτως ώστε να είναι κατανοητή από μόνη της. Μάλιστα θα πρέπει να είναι δυνατή η αλλαγή της υλοποίησης ενός αρθρώματος χωρίς να είναι απαραίτητη η γνώση της υλοποίησης των λοιπών αρθρωμάτων τα οποία απαρτίζουν το υπόλοιπο πρόγραμμα. Τέλος μια τέτοια πιθανή αλλαγή δε θα πρέπει να επηρρεάζει την υλοποίηση των υπόλοιπων αρθρωμάτων.

Υπάρχουν και άλλες παράμετροι οι οποίες μπορούν να επηρρεάσουν τον τρόπο με τον οποίο θα επιλέξουμε να δημιουργήσουμε τα αρθρώματά μας. Επειδή τα αρθρώματα μπορεί πολλές φορές να επαναχρησιμοποιηθούν σε διαφορετικά προγράμματα, ο προγραμματιστής μπορεί να επιλέξει να ομαδοποιήσει κλάσεις και αντικείμενα κατά τέτοιο τρόπο ούτως ώστε η επαναχρησιμοποίησή τους να είναι βολική. Ακόμα, επειδή η κατάτμηση εργασίας γίνεται με βάση τα αρθρώματα, θα πρέπει η σχεδίασή τους να είναι τέτοια ούτως ώστε να εξυπηρετεί την ανάγκη υλοποίησής τους από διαφορετικούς προγραμματιστές.

Θα πρέπει πάντως να τονίσουμε πως ίσως το σοβαρότερο κριτήριο για την σχεδίαση της άρθρωσης ενός προγράμματος είναι ο καταμερισμός του προβλήματος σε επιμέρους ανεξάρτητες οντότητες οι οποίες μάλιστα έχουν το χαρακτηριστικό να έχουν φυσική και νοηματική ανεξαρτησία. Αυτό σημαίνει να έχουμε αρθρώματα τα οποία περιέχουν κλάσεις και αντικείμενα τα οποία είναι αυτοτελή και έχουν τη μικρότερη δυνατή γνώση για τη δομή των υπολοίπων αρθρωμάτων.

Η αφαίρεση είναι ένας ακόμα τρόπος που οι άνθρωποι διαχειρίζονται την πολυπλοκότητα. Άλλωστε η αφαιρετική ικανότητα προκύπτει από το γεγονός της αναγνώρισης ομοιοτήτων μεταξύ συγκεκριμένων αντικειμένων, καταστάσεων ή διαδικασιών στον πραγματικό κόσμο και στην απόφασή μας να συγκεντρωθούμε σε αυτές τις ομοιότητες προσπερνώντας και παρακάμπτοντας προσωρινά τις διαφορές. Έτσι μπορεί κανείς να προβεί σε μια απλοποιημένη περιγραφή ενός πολύπλοκου, εν γένει, συστήματος δίνοντας έμφαση σε κάποιες λεπτομέρειες ή ιδιότητές του ενώ την ίδια στιγμή να παραβλέπει κάποιες άλλες. Μάλιστα, θα μπορούσε κανείς να πεί οτι μια σωστή χρήση της αφαίρεσης θα ήταν το να δίνεται έμφαση στις λεπτομέρεις οι οποίες είναι χρήσιμες και σημαντικές και να παραβλέπονται αυτές οι οποίες, τουλάχιστον για τη συγκεκριμένη στιγμή, θα ήταν ανούσιες ή ακόμα και παραπλανητικές.

Όσον αφορά τον αντικειμενοστραφή σχεδιασμό και με δεδομένα τα ανωτέρω, η αφαίρεση έχει ως στόχο να προβάλλει τα ουσιώδη και ουσιαστικά χαρακτηριστικά ενός αντικειμένου τα οποία μάλιστα το προσδιορίζουν και το διαχωρίζουν από όλα τα υπόλοιπα είδη αντικειμένων και έτσι να προσδίδει σαφώς ορισμένα νοηματικά όρια, σχετικά πάντα με την προοπτική του προγραμματιστή.

Πιο συγκεκριμένα, η αφαίρεση συκεντρώνεται στην εξωτερική «όψη» ενός αντικειμένου και άρα χρησιμεύει στο να διαχωρίσει την κατ΄ ουσίαν συμπεριφορά ενός αντικειμένου με την προγραμματιστική υλοποίησή του. Στην διεθνή βιβλιογραφία, αυτή η διαίρεση της συμπεριφοράς με την υλοποίηση καλείται αφαιρετικό φράγμα (abstraction barrier) και επιτυγχάνεται εφαρμόζωντας την αρχή της ελάχιστης αφοσίωσης (least commitment) μέσω της οποίας ο ορισμός ενός αντικειμένου προσδίδει την στοιχειώδη συμπεριφορά του και τίποτε παραπάνω.

Το να αποφασίσει κανείς για το σωστό σύνολο των αφαιρέσεων για ένα δεδομένο πρόβλημα είναι η κεντρική ιδέα και το κεντρικό πρόβλημα στον αντικειμενοστραφή σχεδιασμό και, μάλιστα, είναι αρκετά δύσκολο αν φανταστεί κανείς οτι υπάρχει ένα φάσμα επιλογών από αντικείμενα τα οποία προσομοιώνουν πολύ καλά φυσικές οντότητες του προβλήματος το οποίο πάμε να λύσουμε μέχρι αντικείμενα τα οποία δεν έχουν κανένα λόγο ύπαρξης.

Η αφαίρεση (abstraction) και η απομόνωση (encapsulation) είναι συμπληρωματικές έννοιες – η αφαίρεση έχει να κάνει με την εμφανή συμπεριφορά ενός αντικειμένου ενώ η απομόνωση επικεντρώνεται στην υλοποίηση η οποία κάνει φανερή τη συμπεριφορά του στα υπόλοιπα αντικείμενα. Προφανώς, η αφαιρετική διαδικασία η οποία αφορά ένα αντικείμενο θα πρέπει να προηγείται αυτής που αφορά την υλοποίηση άρα και την απομόνωσή του. Από τη στιγμή που μια συγκεκριμένη υλοποίηση έχει επιλεχθεί, θα πρέπει αυτή να παραμένει άγνωστη από την αφαιρετική διαδικασία αλλά και από τους χρήστες του αντικειμένου. Άλλωστε, κανένα μέρος ενός σύνθετου συστήματος δεν θα πρέπει να εξαρτάται από τις λεπτομέρειες υλοποίησης των υπόλοιπων μερών του. Έτσι ενώ η αφαιρετική διαδικασία βοηθά τον προγραμματιστή να διατηρεί την προοπτική του στο τι ακριβώς επιτυγχάνει με ένα αντικείμενο, η απομόνωση τον βοηθάει να κάνει αλλαγές σε αυτό με περιορισμένο κόπο.

Η απομόνωση επιτυγχάνεται τις περισσότερες φορές με μια διαδικασία την οποία αποκαλούμε απόκρυψη πληροφοριών (information hiding), η οποία είναι η διαδικασία της απόκρυψης όλων των στοιχείων του αντικειμένου τα οποία δεν συνεισφέρουν στα ουσιαστικά του χαρακτηριστικά. Έτσι, σε τυπικές περιπτώσεις, η δομή ενός αντικειμένου είναι κρυμμένη καθώς επίσης και η υλοποίηση των μεθόδων του.

Η εφαρμογή της απομόνωσης στον αντικειμενοστραφή σχεδιασμό μας παρέχει άμεσες και καλά ορισμένες διαχωριστικές γραμμές ανάμεσα σε διαφορετικά επίπεδα αφαίρεσης και με αυτό τον τρόπο μας οδηγεί σε έναν καθαρό διαχωρισμό των ρόλων των αντικειμένων. Έτσι, για παράδειγμα, για να καταλάβουμε τον τρόπο με τον οποίο λειτουργεί ένας επιλύτης σε γενικές γραμμές, μπορούμε να αγνοήσουμε λεπτομέρειες όπως η φύση ή η γεωμετρία του προς επίλυση φορέα ή το τι είδους πεπερασμένα στοιχεία απαρτίζουν τον φορέα αυτόν. Βλέπουμε λοιπόν οτι αντικείμενα τα οποία βρίσκονται σε κάποιο επίπεδο αφαίρεσης (level of abstraction), είναι προστατευμένα από τις λεπτομέρειες υλοποίησης άλλων επιπέδων αφαίρεσης.

Συμπεραίνουμε λοιπόν, οτι για να είναι αποτελεσματική η αφαιρετική διαδικασία, οι υλοποιήσεις των διαφόρων αντικειμένων πρέπει να είναι απομονωμένες (encapsulated). Στην πράξη, αυτό σημαίνει ότι κάθε αντικείμενο πρέπει να αποτελείται από δυο μέρη – τη διασύνδεσή του (interface) και την υλοποίησή του (implementation). Η διασύνδεση ενός αντικειμένου αντικατοπτρίζει μόνο την εξωτερική του όψη περικλείοντας την συμπεριφορά του η οποία είναι κοινή προς όλα τα υπόλοιπα αντικείμενα. Η υλοποίηση του αντικειμένου συντελεί στην υλοποίηση αυτής της συμπεριφοράς καθώς επίσης και των μηχανισμών εκείνων που επιτυγχάνουν την ως άνω συμπεριφορά. Η διασύνδεση του αντικειμένου είναι το μέρος εκείνο στο οποίο προβάλλουμε όλες τις υποθέσεις που χρειάζεται να κάνουν τα υπόλοιπα αντικείμενα τα οποία αλληλεπιδρούν με αυτό ενώ στην υλοποίηση απομονώνονται (are encapsulated) όλες εκείνες οι λεπτομέρειες οι οποίες δεν έχουν καμία χρησιμότητα στα υπόλοιπα αντικείμενα.

Έχοντας υπ’ όψη μας τα ανωτέρω, μπορούμε να πούμε οτι η απομόνωση είναι μια διαδικασία που έχει ως στόχο τη διαμερισματοποίηση των στοιχείων ενός αντικειμένου τα οποία συνθέτουν τη δομή του και τη συμπεριφορά του. Η απομόνωση χρησιμεύει στο να μπορεί να διαχωρίζει τη διασύνδεση (interface) ενός αντικειμένου από την υλοποίηση (implementation) του.

Όπως είδαμε και παραπάνω, η αφαίρεση είναι ένα πολύ καλό εργαλείο για την κατανόηση ενός σύνθετου προβλήματος αλλά σε όλα εκτός των πιο τετριμμένων προβλημάτων, μπορεί να συναντήσουμε αρκετά περισσότερες αφαιρετικές διαδικασίες από αυτές τις οποίες μπορούμε να χειριστούμε και να κατανοήσουμε. Η απομόνωση μας βοηθάει να χειριστούμε και να διαχειριστούμε αυτή την εγγενή πολυπλοκότητα με την απόκρυψη των υλοποιήσεών μας. Ακόμα και η άρθρωση μας βοηθάει δίνοντας μας μια μέθοδο για να κατατμήσουμε λογικά συσχετιζόμενες αφαιρετικές διαδικασίες. Εντούτοις, οι ιδιότητες αυτές δεν είναι αρκετές – ένα σύνολο αφαιρετικών διαδικασιών μπορεί συχνά να μορφώσει μια ιεραρχία και με το να ταυτοποιούμε και να αναγνωρίζουμε αυτές τις ιεραρχίες, μπορούμε να απλουστεύσουμε δραστικά την κατανόηση του προβλήματός μας.

Η ιδιότητα της ιεραρχίας, λοιπόν, είναι η διαβάθμιση ή η ταξινόμηση των αφαιρετικών διαδικασιών οι οποίες απαρτίζουν το πρόβλημά μας. Τα δύο πιο σπουδαία είδη ιεράρχησης τα οποία συναντούμε στον αντικειμενοστραφή σχεδιασμό σύνθετων συστημάτων είναι η δομή των κλάσεων (ιεραρχία είδους – "is a" hierarchy, κληρονομικότητα - inheritance) και η δομή των αντικειμένων (ιεραρχία μέρους – "part of" hierarchy, συσσώρευση - aggregation).

Η κληρονομικότητα είναι πολύ σπουδαίο είδος ιεράρχησης και όπως αναφέρθηκε και προηγουμένως, αποτελεί ουσιαστικό στοιχείο των αντικειμενοστραφών συστημάτων. Βασικά, η κληρονομικότητα ορίζει μια σχέση μεταξύ των κλάσεων στην οποία μία κλάση μοιράζεται τη δομή ή τη συμπεριφορά η οποία ορίζεται σε κάποια ή κάποιες κλάσεις (ενδεικνύοντας μονή κληρονομικότητα – single inheritance ή πολλαπλή κληρονομικότητα – multiple inheritance αντιστοίχως). Έτσι, η κληρονομικότητα αντιπροσωπεύει μια ιεραρχία αφαιρετικών διαδικασιών, στην οποία μια υπο-κλάση κληρονομεί από μια ή περισσότερες υπερ-κλάσεις. Στις περισσότερες περιπτώσεις, μια υπο-κλάση, μεγεθύνει ή επαναπροσδιορίζει την υπάρχουσα δομή και συμπεριφορά των υπερ-κλάσεών της.

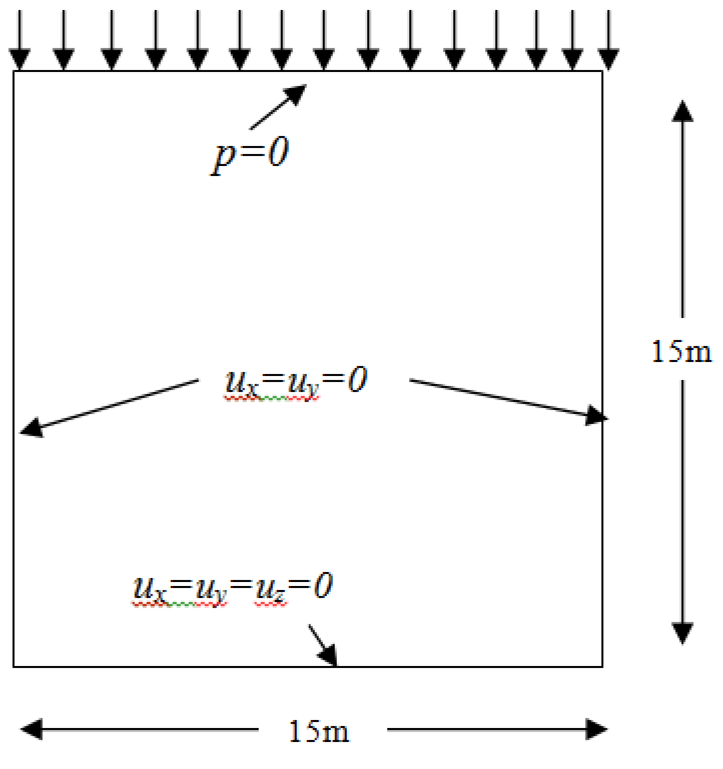
Σημασιολογικά, η κληρονομικότητα υποδηλώνει μια σχέση είδους ("is-a" relationship). Έτσι για παράδειγμα, ένας σκύλος είναι είδους θηλαστικό, ή ένα εξαεδρικό πεπερασμένο στοιχείο είναι είδους τρισδιάστατο πεπερασμένο στοιχείο. Κατ' αυτόν τον τρόπο, η κληρονομικότητα ορίζει εμμέσως μια ιεραρχία γενίκευσης/εξειδίκευσης, όπου μια υποκλάση εξειδικεύει την γενικότερη δομή ή συμπεριφορά των υπερ-κλάσεών της. Πράγματι, αυτός ο έμμεσος ορισμός είναι ένας καλός έλεγχος για την εφαρμογή της κληρονομικότητας – αν η κλάση Y δεν είναι είδους X, τότε το Y δεν πρέπει να κληρονομήσει χαρακτηριστικά από το X.

Καθώς εξελίσσουμε την ιεραρχία της κληρονομικότητας, η δομή και η συμπεριφορά η οποία είναι κοινή για διαφορετικές κλάσεις, τείνει να αποδημήσει και να συσσωρευτεί σε μια κοινή υπερ-κλάση. Κατ' αυτόν τον τρόπο, η κληρονομικότητα μας επιτρέπει να διατυπωσουμε τις αφαιρετικές μας διαδικασίες με οικονομικό τρόπο. Έτσι, αμελώντας τις ιεραρχίες είδους οι οποίες υπαρχουν, μπορεί να οδηγηθούμε σε έναν διογκωμένο και άκομψο σχεδιασμό. Άλλωστε, χωρίς την κληρονομικότητα κάθε κλάση θα ήταν μια ανεξάρτητη μονάδα η οποία θα ήταν χτισμένη εξ' αρχής. Οι διάφορες κλάσεις δε θα φέραν καμία σχέση μεταξύ τους, καθ' ότι ο προγραμματιστής καθε μιας θα παρείχε τη συμπεριφορά τους καθ' όποιον τρόπο αυτός επιθυμούσε. Οποιαδήποτε συνέπεια μεταξύ των κλάσεων είναι αποτέλεσμα πειθαρχίας μεταξύ των προγραμματιστών. Η κληρονομικότητα καθιστά δυνατό τον ορισμό ενός νέου λογισμικού με τον ίδιο τρόπο με τον οποίο εισάγουμε ένα σκεπτικό σε έναν αρχάριο, συγκρίνοντάς το δηλαδή με κατί το οποίο του είναι ήδη οικείο.

Από την άλλη μεριά, έχουμε τη συσσώρευση η οποία είναι μια ιεραρχία μέρους ("part of" hierarchy), η οποία μάλιστα δεν εμφανίζεται μόνο στα αντικειμενοστραφή συστήματα και στις αντικειμενοστραφείς γλώσσες. Πράγματι, οποιαδήποτε γλώσσα υποστηρίζει δομές τύπου εγγραφής (record-type structures), υποστηρίζει και διέπεται από την έννοια της συσσώρευσης. Εντούτοις, ο συνδυασμός κληρονομικότητας και συσσώρευσης φέρει μεγάλη ισχύ – η συσσώρευση επιτρέπει τη φυσική ομαδοποίηση λογικά συσχετιζόμενων δομών και η κληρονομικότητα επιτρέπει σε αυτές τις κοινές ομάδες να μπορούν εύκολα να επαναχρησιμοποιηθούν ανάμεσα σε διαφορετικές αφαιρετικές διαδικασίες.

Η συσσώρευση φέρνει στο προσκήνιο και το θέμα της ιδιοκτησίας. Η αφαιρετική διαδικασία που συνδέεται με έναν υποφορέα, επιτρέπει διαφορετικού είδους πεπερασμένα στοιχεία να χρησιμοποιούνται στο πέρας του χρόνου αλλά δεν αλλάζει την ιδιότητα του υποφορέα ως μια ολότητα, ούτε το να διαγράψουμε έναν υποφορέα σημαίνει απαραίτητα να διαγράψουμε όλα του τα πεπερασμένα στοιχεία (μπορεί αυτά να έχουν γίνει μέρος ενός άλλου υποφορέα). Με άλλα λόγια, ο κύκλος ζωής ενός υποφορέα και των στοιχείων που τον αποτελούν είναι εν γένει ανεξάρτητα.

Σε ό,τι αφορά τις αριθμητικές επιδόσεις των μεθόδων, εξετάζεται το παρακάτω πρόβλημα στερεοποιήσης:



Σκίτσο 8.1 – Τομή του προβλήματος στερεοποιήσης

Διακριτοποιώντας σε υποφορείς έχουμε τα παρακάτω ενδεικτικά σχήματα:

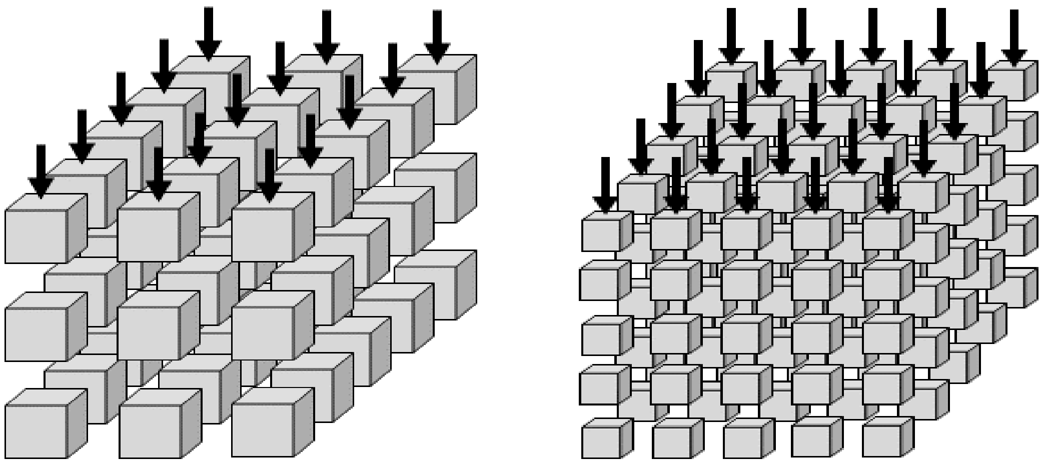
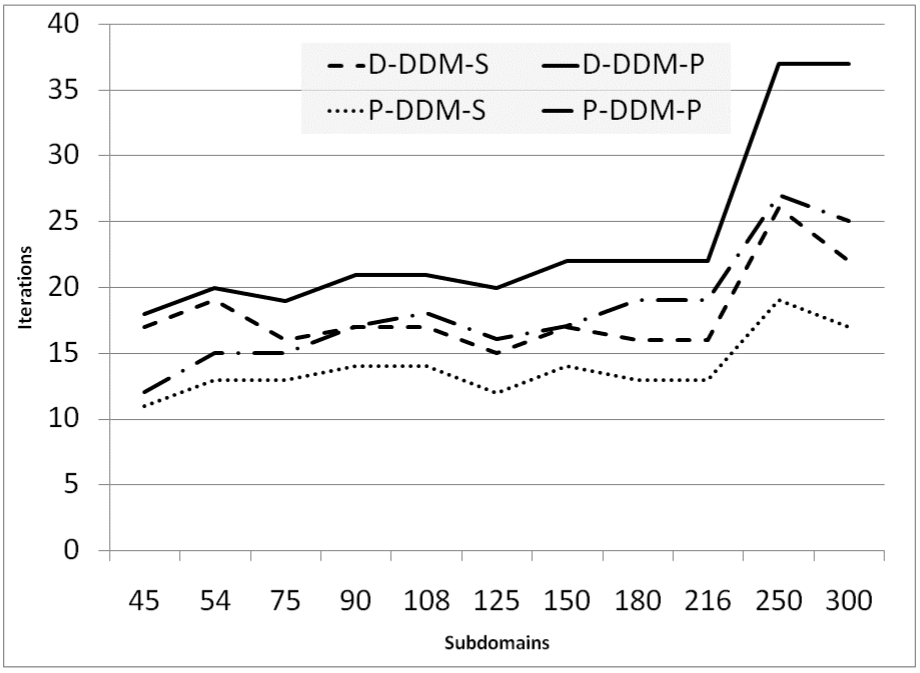


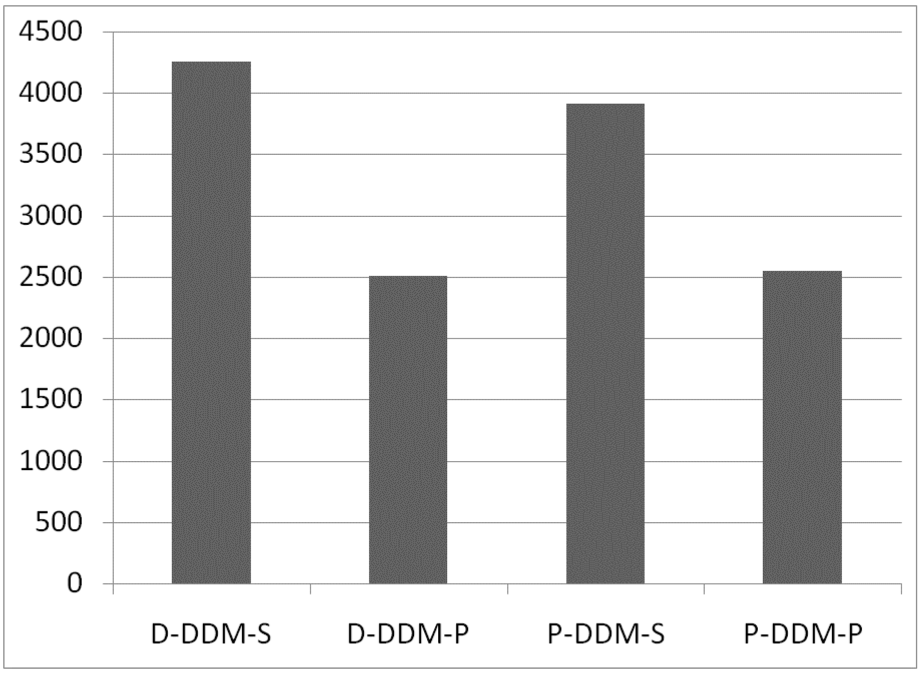
Figure 8.1 - Partitioning in 45 and 125 subdomains

Στο παρακάτω σκίτσο, βλεπουμε τον αριθμό επαναλήψεων των μεθόδων για διαφορετική κατάτμηση σε υποφορείς.



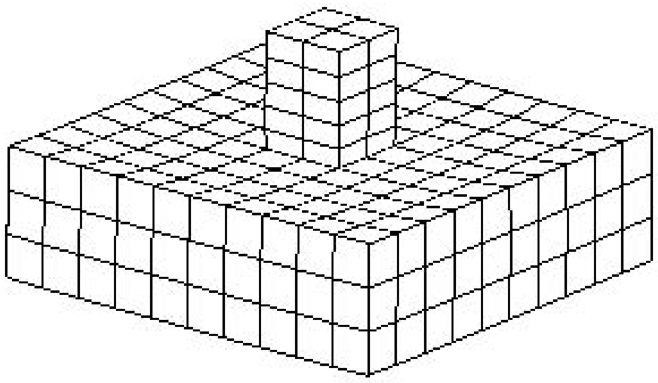
Σκίτσο 8.3 – Γράφημα αριθμού υποφορέων και επαναλήψεων

Σε ό,τι αφορά τις βέλτιστες επιδόσεις των μεθόδων, έχουμε το παρακάτω γράφημα:



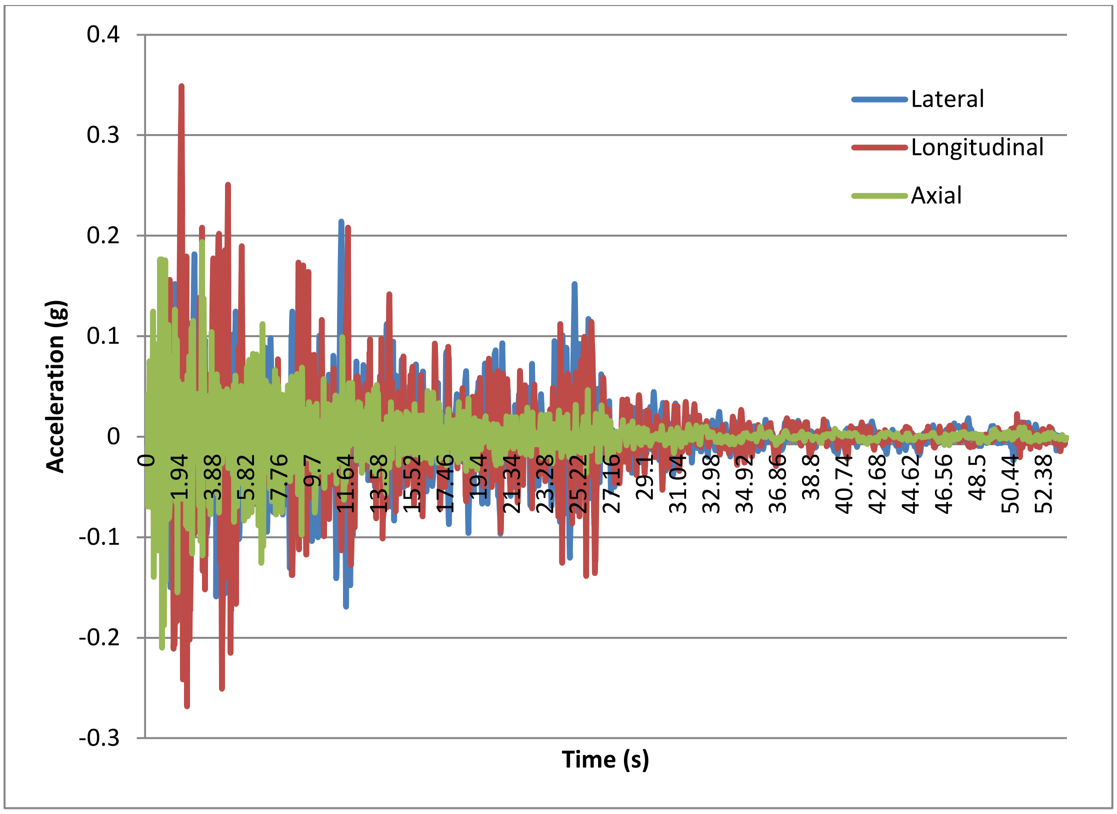
Σκίτσο 8.4 – Συνολική επίδοση των μεθόδων

Σε ό,τι αφορά προβλήματα αλλεπίδρασης εδάφους κατασκευής, γίνεται μια παραμετρική διερεύνηση ενος 5-όροφου κτιρίου από χάλυβα σε ένα έδαφος με τρεις στρώσεις, όπως φαίνεται στο παρακάτω σκίτσο.



Σκίτσο 8.5 – Απλοποιημένο μοντέλο αλληλεπίδρασης εδάφους-κατασκευής

Ο σεισμός που εφαρμόζεται απεικονίζεται στο παρακάτω σκίτσο.



Σκίτσο 8.7 – Επιταχυνσιογράφημα σεισμού στις 3 διευθύνσεις

Οι στρωματογραφία του εδάφους έχει ως εξής:

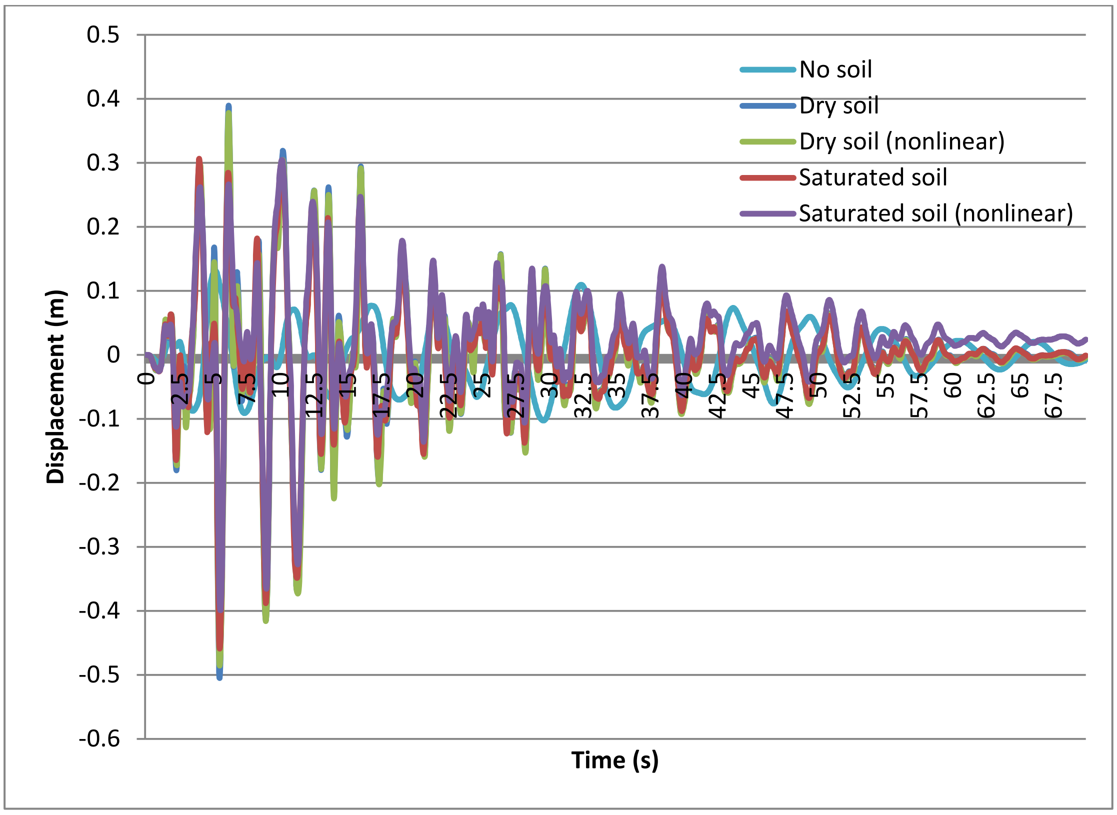
Soil	Young modulus (kPa)	v	Cohesion (kPa)	Friction (deg)	Dilation (deg)	Dry density (t/m^3)	Poro-sity	Permeabi-lity (m/s)
Clay (0-5m)	6000	0.25	17.5	20	0	1.9	0.35	1.02E-06
Sand (5-15m)	20000	0.3	0	35	5	1.8	0.455	3.06E-03
Dense sand (15-20m)	60000	0.35	300	30	0	1.85	0.3	1.02E-07

Για την παραμετρική ανάλυση η οποια αφορά το κτίριο χωρίς την επίδραση του εδάφους (No soil), με επιδραση εδάφους γραμμική (Dry soil), με επίδραση πλήρως κορεσμένου εδάφους (Saturated soil), με επίδραση εδάφους μη-γραμμική (Dry soil-nonlinear) και με επίδραση πλήρως κορεσμένου εδάφους μη-γραμμική (Saturated soil – nonlinear), οι διάφορες μετακινήσεις έχουν ως ακολούθως:

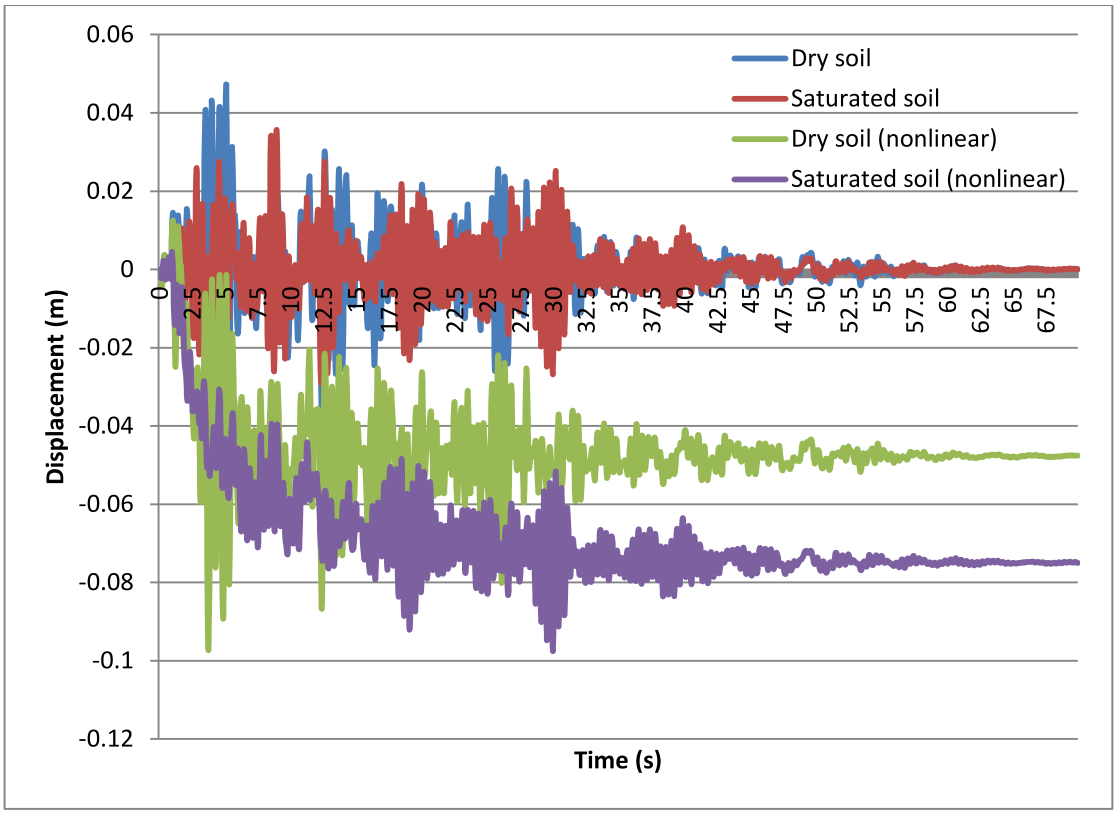
Foundation	Max roof displacement	Max soil displacement	Max axial displacement	Settlement (m)

	(m)	**(m)**	**(m)**	
No soil	0.1313	0	0.1067	0
Dry soil	0.50494	0.11278	0.06578	0
Saturated soil	0.45885	0.10137	0.03569	0
Dry soil (nonlinear)	0.48531	0.10538	0.09729	0.04763
Saturated soil (nonlinear)	0.39891	0.08723	0.09755	0.07502

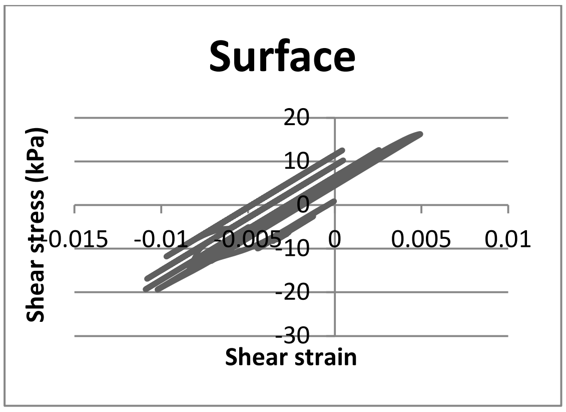
Η χρονοϊστορία των μετατοπίσεων οροφής, απεικονίζεται στο παρακάτω σκίτσο.

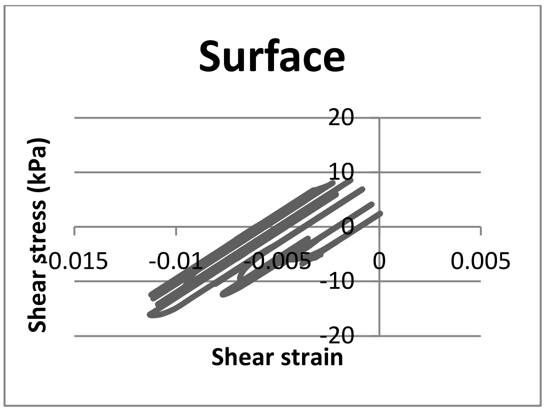


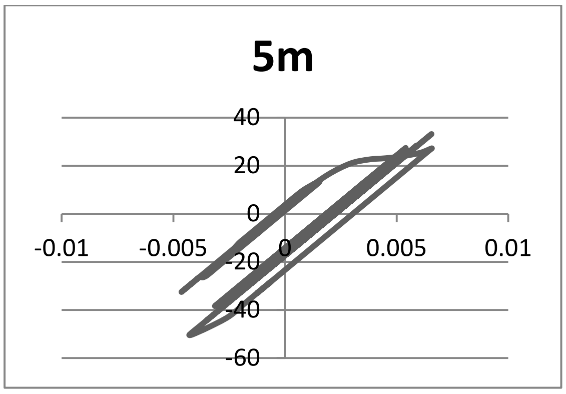
Σε ό,τι αφορά τις καθιζήσεις, η χρονοϊστορία απεικονίζεται στο παρακάτω σκίτσο.

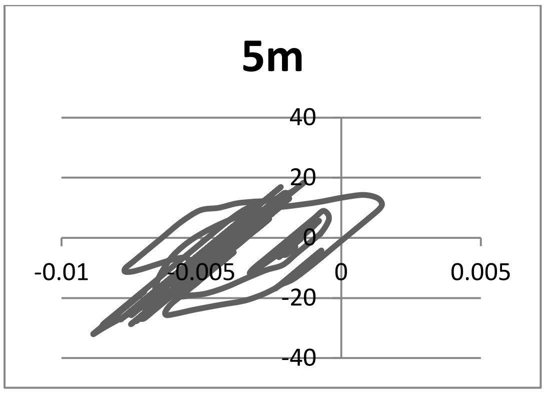


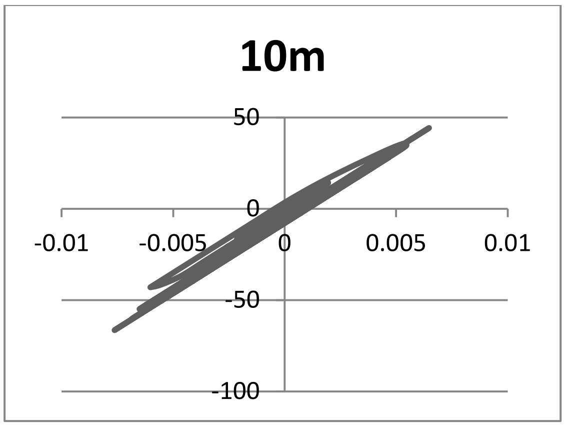
Τέλος, οι καμπύλες τάσεων-παραμορφώσεων διάτμησης, έχουν ως ακολούθως:

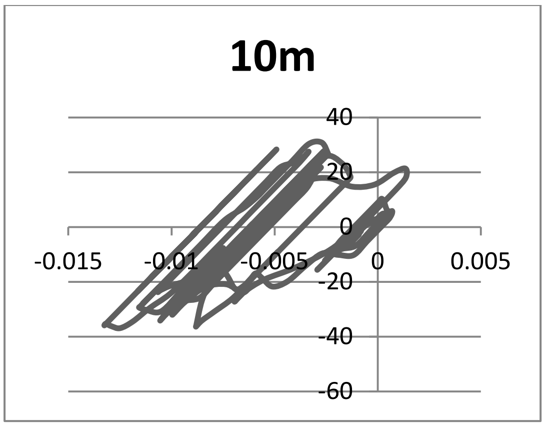


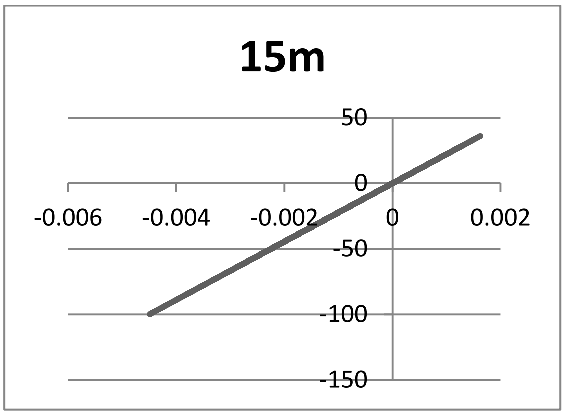


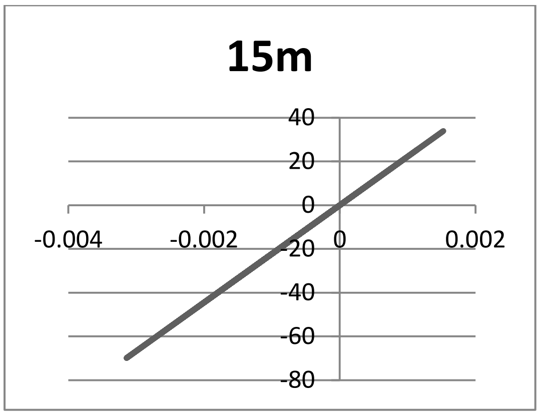




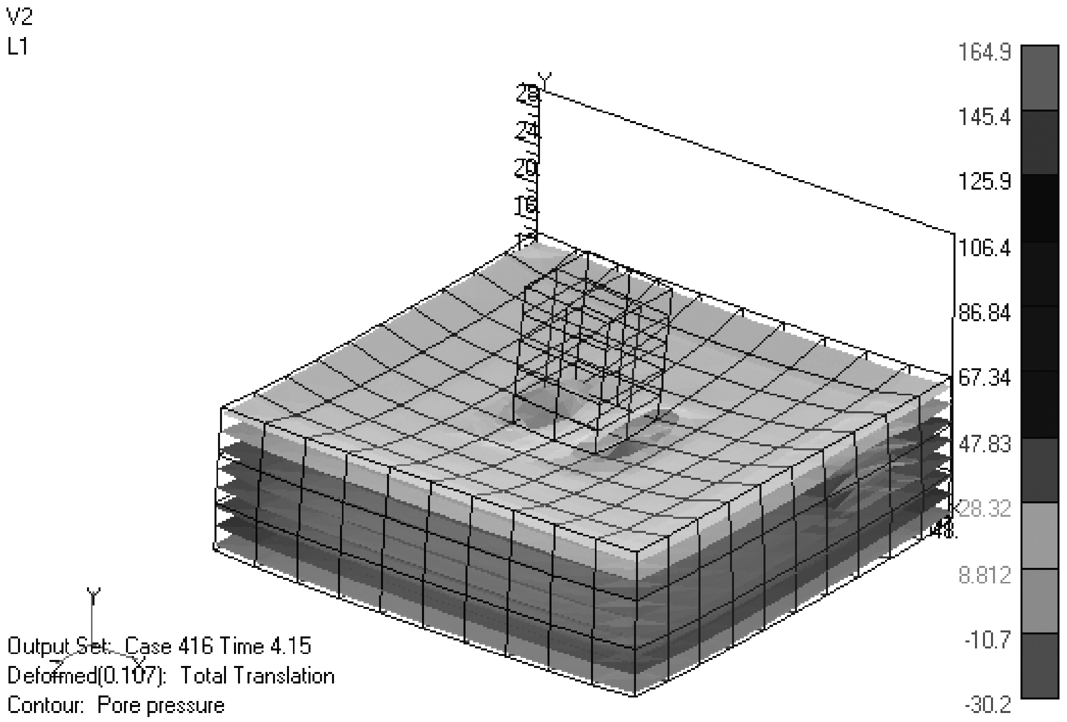








Τέλος, στο παρακάτω σκίτσο φαίνεται ένα στιγμιότυπο της σεισμικής διέγερσης μαζί με τις πιέσεις πόρων.



0 CONTENTS

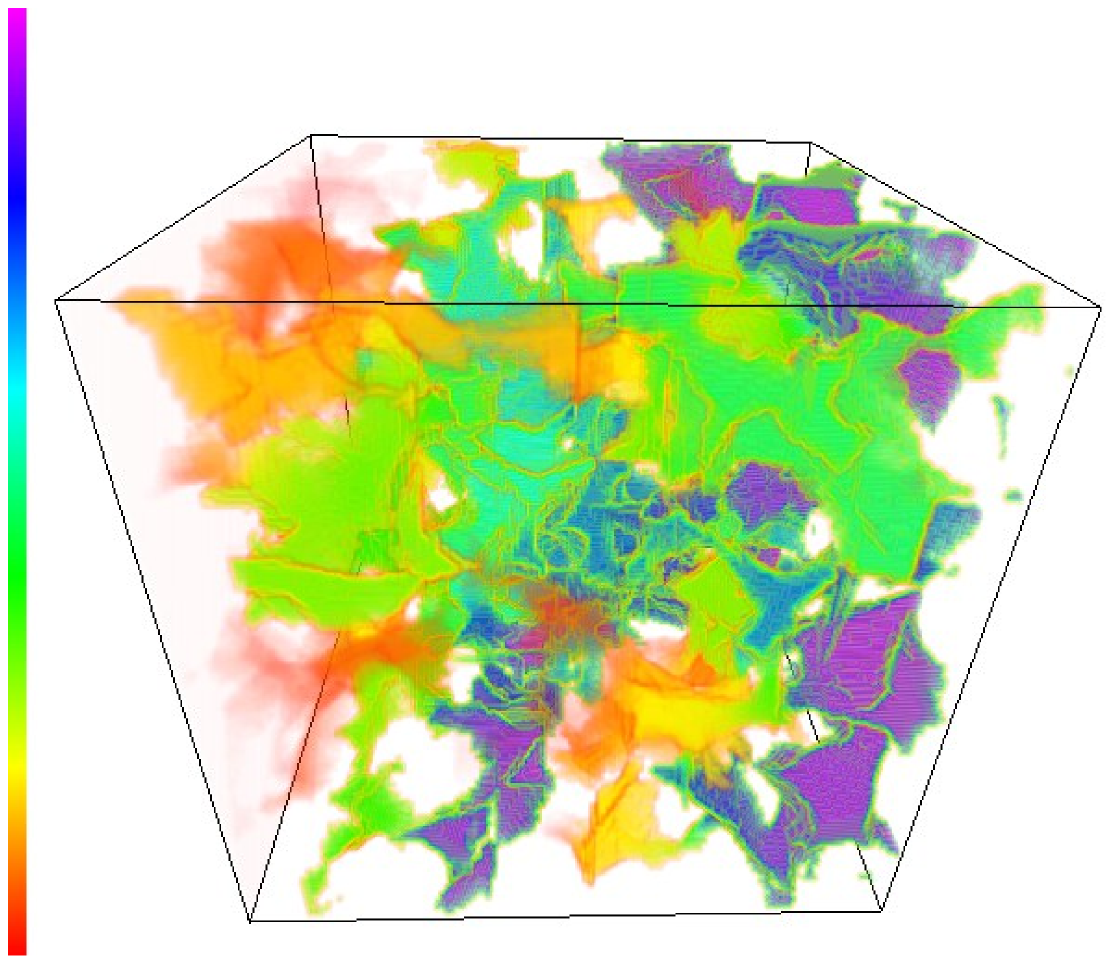
1 INTRODUCTION

One of the most fundamental problems in structural engineering deals with the behavior of structures under seismic loading. Such a problem can prove to be very cumbersome to solve accurately due to the various factors that contribute to its solution, including the presence of water inside the soil pores, the inherent uncertainties of both the structure and the soil, the interaction of the structure foundations with the soil and the difficulty to solve computationally the resulting numerical model due to its massive scale.

This Thesis deals with all of the above problems with the aim to provide a computational toolbox for addressing the solution of this complicated problem. In the sequence of chapters to follow, the reader will get a grasp of porous media mechanics, stochastic analysis and high-performance algebraic solvers along with the programming techniques and programming platforms that were utilized in order to efficiently tackle these kinds of problems.

The final chapter gathers all numerical data produced during this investigation and presents an extensive comparison of the capabilities of the methodologies proposed in this Thesis.

2 POROUS MEDIA



The flow of fluid inside a porous medium specimen.

2.1 NATURE OF SOILS AND POROUS MEDIA

The numerical simulation of the behavior of structures composed of single phase material is quite straightforward, especially for the case of static loading. In the case of a soil medium under static loading the same principle applies, although calculations involved can be more intricate, particularly for dynamic loading which needs special care in order to avoid non-realistic predictions.

This discrepancy between the static and dynamic case is due to the fact that the behavior of media, whose pores are partially or fully filled with a fluid is very different when compared to the behavior of single phase material. Moreover, the usage of continuum mechanics for modeling such media is still an open issue. Since the dimension of interest and the so called 'infinitesimals' dx, dy, dz are large enough when compared to the size of the grains and the pores, it is evident that the approximation of a continuum behaviour holds. Moreover, intergranular forces will be affected by the pore pressures due to the fluid's presence. Consequently, the evaluation of these pore pressures is of great importance for the accurate prediction of the medium's strains and stresses.

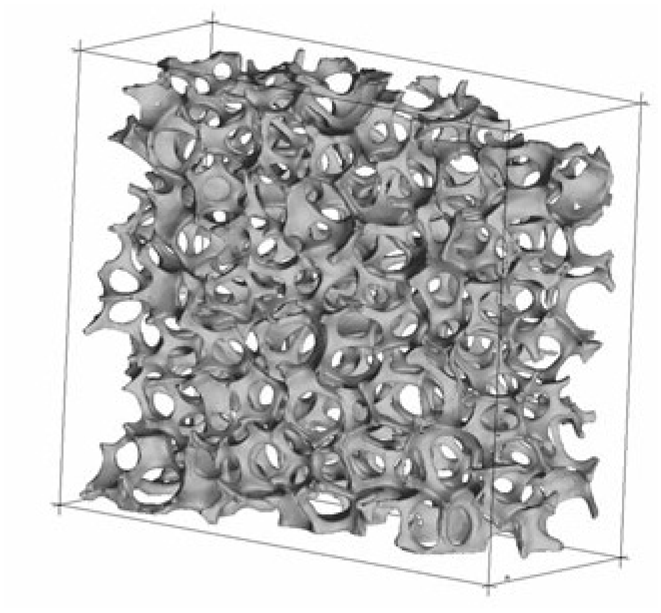


Figure 2.1 – A rock mass modeled as a porous medium

Having evaluated the pore pressures, the notion of effective stress is used which allows the handling of porous media as if they were single phase media. In certain cases, such as the long-term loading conditions on materials of known permeability, limit loading procedures can be used for the evaluation of a soil's behavior. This possibility stems from the fact that draining is constant and the pore pressures are not related with the medium's strains, thus enabling the usage of uncoupled computations.

Such drained behavior, however is rarely observed, even in problems which may be tempting to consider as static due to the slow movement of the pore fluid and the (theoretically) infinite time required to reach this asymptotic behaviour. In very finely grained materials, such as silts or clays, this situation may never be established, even as an approximation.

As an alternative, the so-called undrained behaviour can be adopted and is frequently assumed for rapidly loaded soil. Indeed, if the fluid motion is prevented by zero permeability or by extreme speed of the loading phenomena, the pressures developed in the fluid will be linked in a unique manner to deformation of the solid material and a single-phase behaviour can again be specified. While this approach is occasionally useful in static studies, it is not applicable to dynamic phenomena such as those which occur in earthquakes as the pressures developed will, in general, be linked again to the straining (or loading) history and this must always be taken into account.

In the general case, precise evaluation of the material's behavior requires the simultaneous solution of both strains of the solid matrix and the transient fluid flow which occurs inside the medium's pores. The form of these coupled equations will be described and analyzed in the following sections.

2.2 THE NOTION OF EFFECTIVE STRESS

The basic principles which describe the stress distribution that defines the strength and constitutive behavior of a porous medium with internal pore pressures caused by the presence of a fluid inside the medium's pores, have been defined at the end of the 19th century [1, 2, 3, 4, 5, 6]. These works are related to both soils and concrete and other rock-type media. In all of these works, a certain principle is adopted where total stress is divided in two parts: a part which is undertaken by the solid matrix and a part which is undertaken by the fluid that fills the pores. Moreover, it is assumed that the solid matrix strains are intrinsic and are not related to pore pressures.

By defining total the stress σ with its components σ_{ij} and using the index convention, these components are defined by the summation of the corresponding forces at the i direction at sections dx_j. The surfaces of these sections for two different kinds of porous media are depicted in Figure 2.2 for 2D cases.

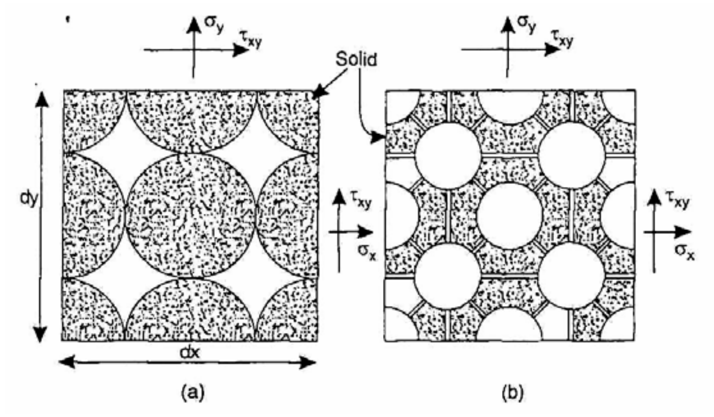


Figure 2.2 – Surface of two sections of different porous media.

For the sake of simplicity and in conformity with the finite element formulation, stresses and other quantities will be described using matrix and vector notation:

$$\sigma = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} \text{(2.2.1) or } \boldsymbol{\sigma} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{bmatrix} \text{(2.2.2)}$$

Moreover, if the stresses applied on the solid matrix of a porous medium are defined as effective stresses σ', hydrostatic pressures due to pore pressures p which are present at the vicinity of the pores are equal to:

$$-\delta_{ij} np \tag{2.2.3}$$

where n is the medium's porosity and δ_{ij} is Kronecker's delta. The negative sign is introduced in order to comply with the assumption that tensile stresses are positive.

The above definitions lead to the following equation which couples the effective and the total stresses as follows:

$$\sigma_{ij} = \sigma'_{ij} - \delta_{ij} np \text{ (2.2.4) or } \boldsymbol{\sigma} = \boldsymbol{\sigma}' - \mathbf{m} np \text{ (2.2.5)}$$

with

$$\mathbf{m} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{2.2.6}$$

However, eqs. (2.2.4) and (2.2.5) do not seem to be validated through experimental procedures since with porosity values in the range of 0.1 to 0.2 it would be possible to induce failure of a porous medium specimen, such as a specimen of concrete, by exposing to simultaneous internal and external pressures. Moreover, it is evident from eq. (2.2.5) that the strength of a certain porous medium would always be related to the pore pressures p. A new definition of effective stress was suggested [7, 8] which is as follows:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}' - \mathbf{m} n_w p \tag{2.2.7}$$

where n_w is the effective area coefficient which, for most cases, is near the vicinity of unity.

2.3 AN ALTERNATIVE APPROACH TO EFFECTIVE STRESS

Assuming that a total hydrostatic pressure and a change of pore pressure, both equal to Δp, is applied at a porous medium specimen, the total increase of total stress would be equal to:

$$\Delta\sigma_{ij} = -\delta_{ij}\Delta p \text{ (2.3.1) or } \Delta\boldsymbol{\sigma} = -\mathbf{m}\Delta p \text{ (2.3.2)}$$

It is evident that for the described loading, only a uniform and very small volumetric strain will be applied in the skeleton while the material will not suffer any damage, provided that the grains of the solid are all made of an identical material. This is simply because all parts of the porous medium solid component will be subjected to an identical compressive stress. However, if the grains are composed by different materials, non-uniform concentrated stresses are likely to occur, which may result to localized grain fractures. Experimental data related to soils, rock masses and other similar materials have shown that such phenomena are of secondary importance and it is safe to assume that all grains of the specimen are under the state of volumetric strain which is equal to:

$$\Delta\varepsilon_v \approx \Delta\varepsilon_{ii} = \Delta\varepsilon_{11} + \Delta\varepsilon_{22} + \Delta\varepsilon_{33} = -\frac{1}{K_s}\Delta p \text{ (2.3.3)or } \Delta\varepsilon_v = \mathbf{m}^T\Delta\mathbf{e} = -\frac{1}{K_s}\Delta p \text{ (2.3.4)}$$

where K_s is the mean bulk modulus of our specimen's solid matrix. It is obvious that if the above specimen is under a simultaneous change of total stress equal to $\Delta\sigma$ and its pore pressure equal to Δp, the resulting incremental strain is equal to:

$$\Delta\varepsilon_{kl} = C_{klij}\left(\Delta\sigma_{ij} + \delta_{ij}\Delta p\right) - \delta_{kl}\frac{1}{3K_s}\Delta p + \Delta\varepsilon^0{}_{kl} \tag{2.3.5}$$

or

$$\Delta\mathbf{e} = \mathbf{D}^{-1}\left(\Delta\boldsymbol{\sigma} + \mathbf{m}\Delta p\right) - \mathbf{m}\frac{1}{3K_s}\Delta p + \Delta\mathbf{e}^0 \tag{2.3.6}$$

with:

$C_{ijkl}D_{mnop} = \delta_{im}\delta_{jn}\delta_{ko}\delta_{lp}$ (2.3.7) or $\mathbf{C}\cdot\mathbf{D} = \mathbf{I}$ (2.3.8)

The last term of eqs. (2.3.5) and (2.3.6) represents the initial strain increment which may be attributed to various factors such as temperature fluctuations. The second term represents the strain increment due to grain compression of the specimen. Matrix $\mathbf{D}$ represents the material constitutive law which in this study is assumed elastic and contains elastic coefficients.

Although the effects of skeleton deformation due to the effective stress defined in eq. (2.2.7) ($n_w = 1$) have been simply added to the uniform volumetric compression, the principal of superposition is not going to be used, so that the resulting equations will be valid in cases of non linear, irreversible elastoplastic and viscoplastic material behavior.

By manipulating eqs. (2.3.5) and (2.3.6), we have:

$$\begin{aligned}
&\Delta\varepsilon_{kl} = C_{klij}\left(\Delta\sigma_{ij} + \delta_{ij}\Delta p - D_{ijkl}\delta_{kl}\frac{1}{3K_s}\Delta p\right) + \Delta\varepsilon^0{}_{kl} \Leftrightarrow \\
&\Leftrightarrow \Delta\varepsilon_{kl} - \Delta\varepsilon^0{}_{kl} = C_{klij}\left(\Delta\sigma_{ij} + a\delta_{ij}\Delta p\right) \Leftrightarrow \\
&\Leftrightarrow D_{klij}\left(\Delta\varepsilon_{kl} - \Delta\varepsilon^0{}_{kl}\right) = \Delta\sigma_{ij} + a\delta_{ij}\Delta p \Rightarrow \\
&\Rightarrow \Delta\sigma''_{ij} = \Delta\sigma_{ij} + a\delta_{ij}\Delta p
\end{aligned} \tag{2.3.9}$$

or

$$\begin{aligned}
&\Delta\mathbf{e} = \mathbf{D}^{-1}\left(\Delta\boldsymbol{\sigma} + \mathbf{m}^T\Delta p - \mathbf{D}\mathbf{m}^T\frac{1}{3K_s}\Delta p\right) + \Delta\mathbf{e}^0 \Leftrightarrow \\
&\Leftrightarrow \Delta\mathbf{e} - \Delta\mathbf{e}^0 = \mathbf{D}^{-1}\left(\Delta\boldsymbol{\sigma} + a\mathbf{m}^T\Delta p\right) \Leftrightarrow \\
&\Leftrightarrow \mathbf{D}\left(\Delta\mathbf{e} - \Delta\mathbf{e}^0\right) = \Delta\boldsymbol{\sigma} + a\mathbf{m}^T\Delta p \Rightarrow \\
&\Rightarrow \Delta\boldsymbol{\sigma}'' = \Delta\boldsymbol{\sigma} + a\mathbf{m}^T\Delta p
\end{aligned} \tag{2.3.10}$$

with σ'' being an alternative effective stress and:

$a\delta_{ij} = \delta_{ij} - D_{ijkl}\delta_{kl}\frac{1}{3K_s}$ (2.3.11) or $a\mathbf{m}^T = \mathbf{m}^T - \mathbf{D}\mathbf{m}^T\frac{1}{3K_s}$ (2.3.12)

However, due to the fact that $\delta_{ij}\delta_{ij} = 3$ or, in matrix notation, $\mathbf{m}\mathbf{m}^T = 3$, the following stands:

$$a\delta_{ij}\delta_{ij} = \delta_{ij}\delta_{ij} - \delta_{ij}D_{ijkl}\delta_{kl}\frac{1}{3K_s} \Leftrightarrow a = 1 - \delta_{ij}D_{ijkl}\delta_{kl}\frac{1}{9K_s} \tag{2.3.13}$$

or

$$a\mathbf{m}\mathbf{m}^T = \mathbf{m}\mathbf{m}^T - \mathbf{m}\mathbf{D}\mathbf{m}^T \frac{1}{3K_s} \Leftrightarrow a = 1 - \frac{\mathbf{m}\mathbf{D}\mathbf{m}^T}{9K_s} \tag{2.3.14}$$

For the case of isotropic material behavior, the following relation stands:

$$\frac{\mathbf{m}\mathbf{D}\mathbf{m}^T}{9} = \frac{\delta_{ij} D_{ijkl} \delta_{kl}}{9} = \frac{\delta_{ij}\left(\lambda\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right)\right)\delta_{kl}}{9} = \frac{9\lambda + 6\mu}{9} = K_T \tag{2.3.15}$$

where K_T is the tangent bulk modulus of an isotropic material with λ and μ being Lame's coefficients. In this case:

$$a = 1 - \frac{K_T}{K_S} \tag{2.3.16}$$

By using non-incremental notation, eqs. (2.3.9) and (2.3.10) can be written as:

$\sigma''_{ij} = \sigma_{ij} + a\delta_{ij} p$ (2.3.17) or $\boldsymbol{\sigma}'' = \boldsymbol{\sigma} + a\mathbf{m}^T p$ (2.3.18)

The above equations are obtained with the assumption that total stress and pore pressures start from a null initial condition (eg. material which is exposed to the air is considered to have zero pore pressure). The above definition corresponds to the one used by Biot [9] but is resulted in a more straightforward manner. In these equations, α is close to unity when K_S of the grains is much larger than the bulk modulus of the whole specimen. In this case, the following relation stands:

$\sigma''_{ij} = \sigma'_{ij} = \sigma_{ij} + \delta_{ij} p$ (2.3.19) or $\boldsymbol{\sigma}'' = \boldsymbol{\sigma}' = \boldsymbol{\sigma} + \mathbf{m}^T p$ (2.3.20)

recovering the common definition of effective stress as defined by Terzaghi [8].

2.4 EFFECTIVE STRESS AND PARTIAL SATURATION

In real life problems, the pores of a porous medium might be occupied by two or more fluids. In this work, it will be assumed that two fluids occupy the pores and the corresponding saturation degree being defined by the ratio of the corresponding fluid volume to the total pore volume n. Under the perspective of soil mechanics, the two fluids are water and air. This way, reference will be made to two saturation degrees, water saturation S_w and air saturation S_a.

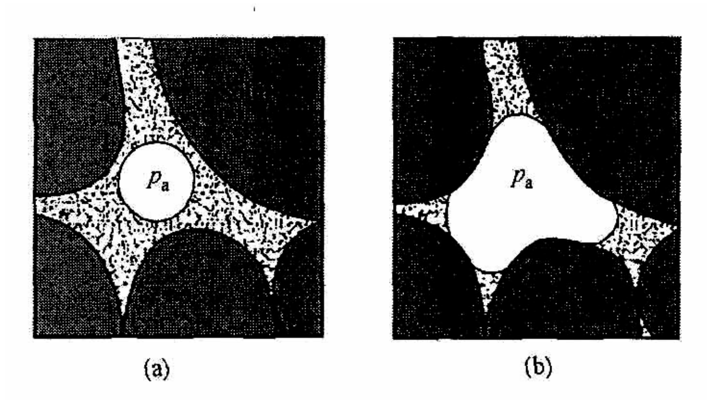


Figure 2.3 – Contact surface of air bubble inside a porous medium

It is evident that if both fluids occupy the pores of the porous medium, the following condition stands:

$$S_w + S_a = 1 \tag{2.4.1}$$

The contact surface of the two fluids with the porous medium solid matrix can have various shapes like the ones depicted in Figure 2.3. This way, pore pressure p is equal to:

$$p = \chi_w p_w + \chi_a p_a \tag{2.4.2}$$

with coefficients χ_w and χ_a referring to water and air respectively, while complying with the condition:

$$\chi_w + \chi_a = 1 \tag{2.4.3}$$

Pressures p_w and p_a refer to water and air pressure respectively and their difference which is equal to

$$p_c = p_w - p_a \tag{2.4.4}$$

is directly related to surface tension and degree of saturation. Due to the fact that surface tension occurs from capillary phenomena, it is often called capillary pressure.

Depending on the nature of the solid matrix surface, contact surface between the solid matrix and the two fluids can take various forms like the ones shown in Figure 2.3 with

$$\chi_w = \chi_w\left(S_w\right) \tag{2.4.5}$$

and

$$\chi_a = \chi_a\left(S_a\right) \tag{2.4.6}$$

As shown in Figure 2.3a, no contact of one of the two fluids with the solid matrix may exist. If this happens for the case of air, remote air bubbles are created which results to

$$\chi_a = 0, \chi_w = 1 \tag{2.4.7}$$

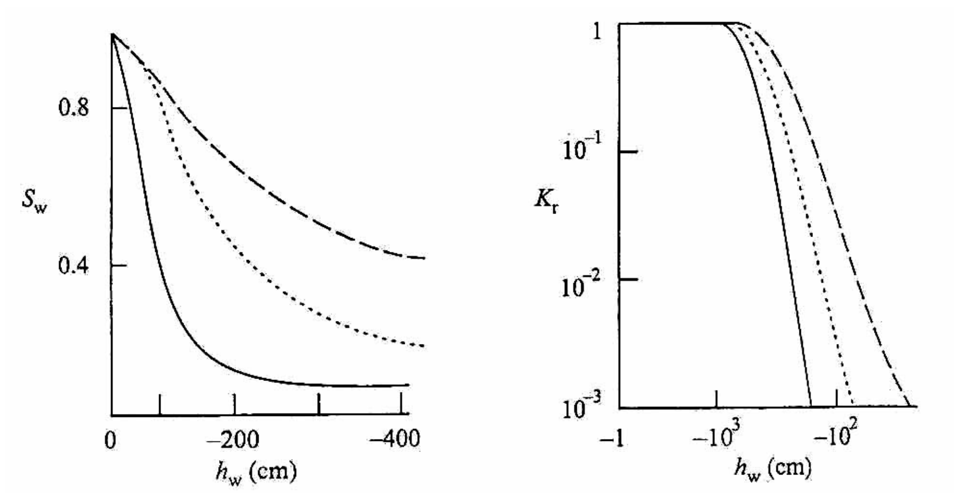


Figure 2.4 – Relation of saturation and permeability with pore pressure

Saturation degree influences other variables related to fluid flow as well. One of these is peremeability k so that:

$$k_w = k_w(S_w) \tag{2.4.8}$$

and

$$k_a = k_a(S_a) \tag{2.4.9}$$

In many real life problems, air pressure is very close to zero while the pores themselves are interconnected. In other cases, negative pore pressures occur with the creation of cavities. In every case, the influence of negative air pressure p_a can be neglected when water pressure becomes negative. Such negative pore pressures are responsible for the increase of soil cohesion which is of great importance while studying the conditions met at the free surface of soils.

2.5 SOLID MATRIX AND PORE PRESSURE DYNAMIC INTERACTION

Following the previous definition of the effective stress, the equations describing the static and dynamic behavior of porous media will be produced. Specifically, the equations describing a porous medium whose pores are occupied by a single fluid will be considered since such conditions include all the basic elements of soil behavior and can describe the majority of real life soil mechanics problems. An extension of these equations will be provided for partially saturated soils for the cases of constant air pressure and of simultaneous water and air flow. The derivation of these equations is being done following a physical – mechanical approach which makes the solid and fluid phase interaction evident.

As seen in the previous section, effective stresses are described by the eqs. (2.3.17) and (2.3.18), while effective stresses as introduced by Terzaghi [8] are described by eqs. (2.3.19) and (2.3.20).

For sands and clays, eqs. (2.3.17) and (2.3.19) and their equivalent (2.3.18) and (2.3.20) provide almost identical results since for these kinds of soils, $a \approx 1$. However, for rock masses or concrete, coefficient a can be in the vicinity of 0.5.

Using incremental notation, the effective stresses can be written as follows:

$$d\sigma''_{ij} = d\sigma_{ij} + a\delta_{ij} dp \quad (2.5.1) \text{ or } d\boldsymbol{\sigma}'' = d\boldsymbol{\sigma} + a\mathbf{m}^T dp \quad (2.5.2)$$

For the case of three dimensions, strains can be written as:

$$d\varepsilon = \begin{bmatrix} d\varepsilon_{11} \\ d\varepsilon_{22} \\ d\varepsilon_{33} \\ d\varepsilon_{12} \\ d\varepsilon_{23} \\ d\varepsilon_{31} \end{bmatrix} \quad (2.5.3) \text{ or } \quad d\mathbf{e} = \begin{bmatrix} d\varepsilon_{x} \\ d\varepsilon_{y} \\ d\varepsilon_{z} \\ d\gamma_{xy} \\ d\gamma_{yz} \\ d\gamma_{zx} \end{bmatrix} \quad (2.5.4)$$

Strain increments of the solid matrix can be related to displacement increments using the following equation:

$$d\varepsilon_{ij} = \frac{1}{2}\left(du_{i,j} + du_{j,i}\right), du_{i,j} \equiv \frac{\partial du_i}{\partial x_j} \quad (2.5.5) \text{ or } \quad d\mathbf{e} = \mathbf{S}d\mathbf{u} \quad (2.5.6)$$

with the comma in the suffix denoting differentiation with respect to the appropriate specified coordinate and the differential operator **S** defined as:

$$\mathbf{S} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \quad (2.5.7)$$

and:

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \tag{2.5.8}$$

The equation of both solid and fluid phase equilibrium is as follows:

$$\sigma_{ij,j} - \rho \ddot{u}_i - \underline{\rho_f \left(\dot{w}_i + w_j w_{i,j} \right)} + \rho b_i = 0 \tag{2.5.9}$$

or

$$\mathbf{S}^T \boldsymbol{\sigma} - \rho \ddot{\mathbf{u}} - \underline{\rho_f \left(\dot{\mathbf{w}} + \mathbf{w} \nabla^T \mathbf{w} \right)} + \rho \mathbf{b} = 0 \tag{2.5.10}$$

Dotted quantities denote differentiation with respect to time. Mean velocity of the percolating water according to Darcy's law is equal to $\mathbf{w}$ while field-induced accelerations acting upon the whole material mass, such as gravity are equal to $\mathbf{b}$. Moreover, ρ_f is the fluid's density and ρ is the whole medium's density, including both solid and fluid phase. These two magnitudes are interrelated as follows:

$$\rho = n\rho_f + (1-n)\rho_s \tag{2.5.11}$$

with ρ_s being the solid phase density and n being equal to the medium's porosity.

The underlined terms of eqs. (2.5.9) and (2.5.10) describe the forces occurring due to fluid acceleration, in distinction to the solid phase which are usually small and will be generally omitted. Moreover, the aforementioned equations are referring to an infinitesimal reference volume equal to $dx_1 \cdot dx_2 \cdot dx_3$ or $dx \cdot dy \cdot dz$ which refers to both the solid and fluid phase of the porous medium.

With the usage of the same aforementioned reference volume, the following equation of equilibrium for the fluid phase of the porous medium is obtained:

$$-p_{,i} - R_i - \rho_f \ddot{u}_i - \underline{\frac{\rho_f \left(\dot{w}_i + w_j w_{i,j} \right)}{n}} + \rho_f b_i = 0 \tag{2.5.12}$$

or

$$-\nabla p - \mathbf{R} - \rho_f \ddot{\mathbf{u}} - \underline{\frac{\rho_f \left(\dot{\mathbf{w}} + \mathbf{w} \nabla^T \mathbf{w} \right)}{n}} + \rho_f \mathbf{b} = 0 \tag{2.5.13}$$

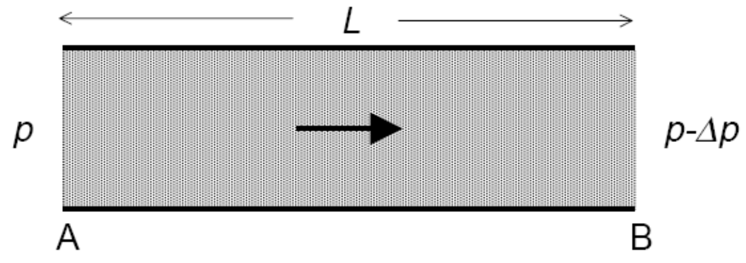


Figure 2.5 – Darcian fluid flow caused by a pressure differential between A and B

Drag forces due to fluid viscosity are equal to $\mathbf{R}$ and by applying the law of Darcy we obtain:

$$k_{ij}R_j = w_i \tag{2.5.14}$$

which in matrix notation can be written as:

$$\mathbf{kR} = \mathbf{w} \tag{2.5.15}$$

where the porous medium permeability is equal to $\mathbf{k}$. It should be noted that permeability dimensions used in these equations are $\frac{[length]^3 \bullet [time]}{[mass]}$. These dimensions are different from those used in soil mechanics problems which are $\frac{[length]}{[time]}$. If the latter permeability is equal to k' then the following relation stands:

$$k = \frac{k'}{\rho'_f\, g'} \tag{2.5.16}$$

with the fluid density being equal to ρ'_f and the acceleration of gravity being equal to g' as measured during the permeability testing for the soil referenced.

The underlined terms of eqs. (2.5.12) and (2.5.13) describe forces occurring due to acceleration of the fluid phase in relation to the solid one. These terms are also small and are usually neglected.

Finally, mass and flow conservation of the fluid is governed by the following equation:

$$w_{i,i} + \dot{\varepsilon}_{ii} + \frac{n\dot{p}}{K_f} + \frac{(1-n)\dot{p}}{K_s} - \frac{K_T}{K_S}\left(\dot{\varepsilon}_{ii} + \frac{\dot{p}}{K_S}\right) + \underline{n\frac{\dot{\rho}_f}{\rho_f} + \dot{s}_0} = 0 \tag{2.5.17}$$

This equation describes the balance between velocity deviation $w_{i,i}$ and the increased fluid storage at the pores of the porous medium having volume equal to the reference volume during an infinitesimal amount of time dt. This increased storage is described by a multitude of terms which are presented below with order of importance:

- Change of volume due to strain change: $\delta_{ij} d\varepsilon_{ij} = d\varepsilon_{ii} = \mathbf{m}^T d\mathbf{e}$
- Increase of volume due to fluid compression as a result of fluid pressure increase: $\dfrac{ndp}{K_f}$
- Increase of volume due to grain compression as a result of fluid pressure increase: $\dfrac{(1-n)dp}{K_S}$
- Change of solid phase volume due to change of intergranular effective stress $\sigma'_{ij} = \sigma_{ij} + \delta_{ij} p$: $-\dfrac{1}{3}\dfrac{\delta_{ij} d\sigma'_{ij}}{K_S} = -\dfrac{K_T}{K_S}\left(d\varepsilon_{ii} + \dfrac{dp}{K_S}\right)$.

Using eq. (2.3.13), eq. (2.5.17) can be transformed as follows:

$$w_{i,i} + a\dot{\varepsilon}_{ii} + \frac{\dot{p}}{Q} + \underline{n\frac{\dot{\rho}_f}{\rho_f} + \dot{s}_0} = 0 \; (2.5.18) \quad \text{or} \quad \nabla^T \mathbf{w} + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}}{Q} + \underline{n\frac{\dot{\rho}_f}{\rho_f} + \dot{s}_0} = 0 \; (2.5.19)$$

where:

$$\frac{1}{Q} = \frac{n}{K_f} + \frac{a-n}{K_S} \cong \frac{n}{K_f} + \frac{1-n}{K_S} \tag{2.5.20}$$

The underlined terms of eqs. (2.5.18) and (2.5.19) describe the change in the fluid density and the change of volume change rate due to temperature changes which are small and are generally neglected in real life problems.

Eqs. (2.5.9), (2.5.12) and (2.5.18), along with their matrix notation counterparts (2.5.10), (2.5.13) and (2.5.19), in combination with proper constitutive relations in the form of (2.5.1) define the behavior of a porous medium, taking into consideration both the solid and fluid phases under both static and dynamic conditions. The unknown variables of the system composed of the aforementioned triad of equations are:

- Pore pressures p
- Fluid flow velocities $\mathbf{w}$
- Solid phase displacements $\mathbf{u}$.

In order to complete the problem formulation, boundary conditions are imposed as follows:

- Solid phase boundaries are split into two discrete cases; traction forces $\mathbf{t}$ acting at the surface and displacements $\mathbf{u}$ are known. If the boundary is Γ_s, the following stands:

$$\begin{aligned} &\Gamma_s = \Gamma_t \cup \Gamma_u \\ &\mathbf{t} = \overline{\mathbf{t}}, \Gamma = \Gamma_t \\ &\mathbf{u} = \overline{\mathbf{u}}, \Gamma = \Gamma_u \end{aligned} \tag{2.5.21}$$

- Fluid phase boundaries are also split into two discrete sections: Pore pressures p and fluid velocities $\mathbf{w}$ are known. If the fluid boundary is Γ_f, the following stands:

$$\begin{aligned} &\Gamma_f = \Gamma_p \cup \Gamma_w \\ &p = \bar{p}, \Gamma = \Gamma_p \\ &\mathbf{w} = \bar{\mathbf{w}}, \Gamma = \Gamma_w \end{aligned} \tag{2.5.22}$$

2.6 THE $\mathbf{u}-p$ FORMULATION FOR FULLY SATURATED BEHAVIOR

Eqs. (2.5.10), (2.5.13) and (2.5.19), in combination with proper constitutive relations in the form of (2.5.1), can be used for numerical simulations of porous media [10]. In fact, the system of equations formed from the above triad is an excellent candidate for explicit time integration algorithms [11, 12, 13], at the expense of the necessity to use small time steps. On the other hand, implicit time integration in another candidate for solving coupled equations provided that efficient strategies are applied for the handling of the resulting algebraic systems. In order to reduce their size, it is prudent to decrease the amount of unknown variables by neglecting the underlined quantities of eqs. (2.5.10) and (2.5.13) which refer to the fluid velocities $\mathbf{w}$. This way, the simplified porous medium equilibrium equation can be written as:

$\sigma_{ij,j} - \rho \ddot{u}_i + \rho b_i = 0$ (2.6.1) or $\mathbf{S}^T \boldsymbol{\sigma} - \rho \ddot{\mathbf{u}} + \rho \mathbf{b} = 0$ (2.6.2)

For the complete exclusion of fluid velocities, eqs. (2.5.13) and (2.5.19) are combined while drag forces $\mathbf{R}$ are substituted with their Darcian counterpart as shown in eq. (2.5.14). By neglecting density changes the following relation stands:

$$\left(k_{ij}\left(-p_{,j} - \rho_f \ddot{u}_j + \rho_f b_j\right)\right)_{,i} + a\dot{\varepsilon}_{ii} + \frac{\dot{p}}{Q} + \dot{s}_0 = 0 \tag{2.6.3}$$

or

$$\nabla^T \mathbf{k}\left(-\nabla p - \rho_f \ddot{\mathbf{u}} + \rho_f \mathbf{b}\right) + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}}{Q} + \dot{s}_0 = 0 \tag{2.6.4}$$

This simplified system of eqs. (2.6.1) and (2.6.3) is called the $\mathbf{u}-p$ formulation since the unknown variables are now the displacements $\mathbf{u}$ of the solid phase along with the pore pressures p of the fluid phase. The omission of the dynamic terms of these equations produce the mathematical formulas used for consolidation problems. It is also possible to use the static part of the equations for static porous media problems when all time-related derivatives are equal to zero.

The $\mathbf{u}-p$ formulation may result to loss of accuracy when applied to problems involving high frequency phenomena such as high speed impacts [14]. However, this formulation is acceptable for low to relatively low frequency phenomena such as earthquakes and is adopted in this work.

2.7 THE $\mathbf{u}-p$ FORMULATION FOR PARTIALLY SATURATED BEHAVIOR

In the general case of non linear behavior, effective stresses and pore pressures have to be evaluated incrementally since the solution of non linear problems is incremental in nature, involving a step-by-step solution procedure.

On the other hand, it is very common for certain family of soils to condense under loading. Such a behavior is related to the soil's constitutive model where strain history induces condensation of the solid matrix which results to condensation of the whole porous medium. Such a condensation leads to an increase of pore pressure and to a decrease of intergranular surface stresses. When these stresses reach zero, liquefaction occurs.

However, the exact opposite may happen if, using strain history, an expansion of the porous medium is imposed which will result to negative pore pressures. Such negative pore pressures cause the formation of separation surfaces and the capillary phenomena that come along. During this process, voids will be created which by themselves are incapable of handling any tensile stresses. At that time, pore pressures will become close to zero.

The presence of negative pore pressures obviously increases the strength of a porous medium and can be seen as being beneficiary. This strength increase can be observed above water level or the phreatic line as it is alternatively called. The presence of negative pore pressures above this line guarantees a certain level of cohesion, even for materials that are not cohesive by nature. Such cohesion fortifies the porous medium when it is exposed to dynamic loading conditions.

As seen in section 2.4, there is an one-way relation between saturation degree and pore pressures. Such relations may be expressed using mathematical formulas or graphs as the one shown in Figure 2.4. Using such a relation, the equations of partially saturated behavior can be formulated by modifying the equations shown in section 2.4. Moreover, with the assumption that air pressure remains constant and equal to the atmospheric pressure, there is no need to introduce new variables.

With the last assumption in mind, pore pressures with respect to effective stress are now equal to:

$$p = \chi_w p_w + \chi_a p_a = \chi_w p_w + \left(1-\chi_w\right) p_a \approx \chi_w p_w \tag{2.7.1}$$

while the density of the whole porous medium is now equal to:

$$\rho = nS_w\rho_w + \left(1-n\right)\rho_s \tag{2.7.2}$$

This way, the simplified porous medium equilibrium equation can be written as:

$$\sigma_{ij,j} - \left(nS_w\rho_w + (1-n)\rho_s\right)\ddot{u}_i + \left(nS_w\rho_w + (1-n)\rho_s\right)b_i = 0 \qquad (2.7.3)$$

or

$$\mathbf{S}^T\boldsymbol{\sigma} - \left(nS_w\rho_w + (1-n)\rho_s\right)\ddot{\mathbf{u}} + \left(nS_w\rho_w + (1-n)\rho_s\right)\mathbf{b} = 0 \qquad (2.7.4)$$

Moreover, the equation of equilibrium for the fluid phase of the porous medium can be written as:

$-p_{w,i} - R_i - \rho_w\ddot{u}_i + \rho_w b_i = 0$ (2.7.5) or $-\nabla p_w - \mathbf{R} - \rho_w\ddot{\mathbf{u}} + \rho_w\mathbf{b} = 0$ (2.7.6)

Finally, the equation of mass and flow conservation of the fluid is re-derived. This equation describes the balance between velocity deviation $w_{i,i}$ and the increased fluid storage at the pores of the porous medium having volume equal to the reference volume during dt time period. This increased storage, considering the existence of both fluids, is described by the following terms which are presented in order of importance:

- Change of volume due to strain change: $\delta_{ij}d\varepsilon_{ij} = d\varepsilon_{ii} = \mathbf{m}^T d\mathbf{e}$.
- Increase of volume due to fluid compression as a result of fluid pressure increase: $\dfrac{nS_w dp_w}{K_f}$.
- Increase of volume due to grain compression as a result of fluid pressure increase: $\dfrac{(1-n)\chi_w dp_w}{K_S}$.
- Change of solid phase volume due to change of intergranular effective stress $\sigma'_{ij} = \sigma_{ij} + \delta_{ij}p : -\dfrac{1}{3}\dfrac{\delta_{ij}d\sigma'_{ij}}{K_S} = -\dfrac{K_T}{K_S}\left(d\varepsilon_{ii} + \dfrac{\chi_w dp_w}{K_S}\right)$.
- Change of saturation degree: ndS_w.

Using the above terms, the equation of mass and flow conservation is written as:

$$\begin{aligned} &w_{i,i} + a\dot{\varepsilon}_{ii} + \frac{nS_w\dot{p}_w}{K_f} + \frac{(a-n)\chi_w\dot{p}_w}{K_s} + n\dot{S}_w + \underline{nS_w\frac{\dot{\rho}_w}{\rho_w} + \dot{s}_0} = \\ &w_{i,i} + a\dot{\varepsilon}_{ii} + \frac{\dot{p}_w}{Q^*} + \underline{nS_w\frac{\dot{\rho}_w}{\rho_w}} + \dot{s}_0 = 0 \end{aligned} \qquad (2.7.7)$$

or

$$\nabla^T\mathbf{w} + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}_w}{Q^*} + \underline{n\frac{\dot{\rho}_w}{\rho_w}} + \dot{s}_0 = 0 \qquad (2.7.8)$$

where:

$$\frac{1}{Q^*} = C_s + \frac{nS_w}{K_f} + \frac{(a-n)\chi_w}{K_S} \tag{2.7.9}$$

The parameter C_s is defined from the relation:

$$n\frac{dS_w(p_w)}{dt} = n\frac{dS_w(p_w)}{dp_w}\frac{dp_w}{dt} = C_s\dot{p}_w \tag{2.7.10}$$

Parameter Q is now replaced by a new parameter Q^* which takes into account the change of saturation degree. It is worth noting that for the case of full saturation, where $S_w = 1$ and $\chi_w = 1$, both parameters Q and Q^* are equal.

In order to derive the $\mathbf{u}-p$ formulation for the partially saturated case, a procedure similar to the one described in section 2.6 is followed. This procedure leads to the following equation:

$$\left(k_{ij}\left(-p_{w,j} - S_w\rho_w\ddot{u}_j + S_w\rho_w b_j\right)\right)_{,i} + a\dot{\varepsilon}_{ii} + \frac{\dot{p}}{Q^*} + \dot{s}_0 = 0 \tag{2.7.11}$$

or

$$\nabla^T\mathbf{k}\left(-\nabla p_w - S_w\rho_w\ddot{\mathbf{u}} + S_w\rho_w\mathbf{b}\right) + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}}{Q^*} + \dot{s}_0 = 0 \tag{2.7.12}$$

By observing eqs. (2.7.4) and (2.7.12), it is evident that by implementing a computer code which performs analysis of partially saturated soils is prudent since, with the proper choice of the parameter values, the same equations apply for fully saturated soils. During the computation of each time step, parameters S_w, k_w and C_s are slowly changing and will be considered constant for each time step. This approach can be used for the simulation of a large variety of soil mechanics problems [14].

2.8 FINITE ELEMENT METHOD DISCRETIZATION

In this work, the numerical solution of the system of equations describing porous media problems with the $\mathbf{u}-p$ formulation will be spatially discretized using the finite element method [15, 16]. The solution of this system is similar to the solution of partial differential equations of the following form:

$$\mathbf{A}\ddot{\Phi} + \mathbf{B}\dot{\Phi} + \mathbf{L}(\Phi) = 0 \tag{2.8.1}$$

where $\mathbf{A}$ and $\mathbf{B}$ are matrices filled with constant values and $\mathbf{L}$ is an operator containing spatial differentials of the form $\frac{\partial}{\partial x}, \frac{\partial}{\partial y}$ etc, which may be linear or non linear. Finally, Φ is a vector of unknown variables, like those of the solid phase displacements or the fluid phase pressures.

In order to reach a solution using the finite element method, the unknown functions Φ are approximated or discretized by using a set of discrete parameters $\overline{\Phi}_\kappa$ and shape functions N_κ which are defined in spatial dimensions. This approximation is of the form:

$$\Phi \cong \Phi^h = \sum_{k=1}^{n} N_k \overline{\Phi}_k \tag{2.8.2}$$

and shape functions are defined in spatial coordinates as follows:

$$\begin{aligned} N_k &= N_k\left(x, y, z\right) \\ N_k &= N_k\left(\mathbf{x}\right) \\ \overline{\Phi}_i &\equiv \overline{\Phi}_i\left(t\right) \end{aligned} \tag{2.8.3}$$

with $\overline{\Phi}_i$ being the values of the unknown functions Φ in certain discrete spatial points which are called nodes with their values subject to change with respect to time.

Moreover, by introducing the approximation Φ^h of the unknown functions Φ in the partial differential eq. (2.8.1), a residual occurs which is not identical to zero but for which a set of weighted residual equations can be derived, having the form:

$$\int_\Omega \mathbf{W}_j^T \left(\mathbf{A}\ddot{\Phi}^{\mathrm{h}} + \mathbf{B}\dot{\Phi}^{\mathrm{h}} + \mathbf{L}(\Phi^h)\right) d\Omega = 0 \tag{2.8.4}$$

which upon integration is simplified to the following form:

$$\mathbf{M}\ddot{\overline{\Phi}} + \mathbf{C}\dot{\overline{\Phi}} + \mathbf{P}(\overline{\Phi}) = 0 \tag{2.8.5}$$

where $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{P}$ are matrices which are sized according to the size of parameters $\overline{\Phi}_\kappa$.

An appropriate choice for weight functions $\mathbf{W}_j$ are the shape functions $\mathbf{N}_j$ so that the following stands:

$$\mathbf{W}_j = \mathbf{N}_j \tag{2.8.6}$$

In fact, such a choice is optimal with respect to accuracy and is known as the Galerkin procedure.

If parameters $\overline{\Phi}_{\kappa}$ are time-dependent, eq. (2.8.5), which is now a regular differential equation, requires solution in the time domain. This solution can be evaluated by applying a time discretization scheme combined with the spatial discretization scheme described.

Usually the parameters $\overline{\Phi}_{\kappa}$ just depict values Φ^h in certain nodes and the shape functions are derived as interpolation polyonyms of the finite elements that ae used for the discretization of the spatial field of a specific problem.

2.9 SPATIAL DISCRETIZATION OF THE $\mathbf{u}-p$ FORMULATION

Spatial discretization using the finite element method is simplified with the use of matrix notation. Hence all derived equations will be subsequently described using matrix notation.

Eq. (2.6.2) describes the porous medium equation of equilibrium while strains are described by eq. (2.5.6), total stresses $\boldsymbol{\sigma}$ are described by eq. (2.2.2) while effective stresses are described by eq. (2.3.18).

Total density is equal to:

$$\rho = nS_w\rho_w + (1-n)\rho_s \tag{2.9.1}$$

and is usually regarded as being a constant while p is the pore pressure which is taken as equal to:

$$p = \chi_w p_w + \chi_a p_a = \chi_w p_w + (1-\chi_w) p_a \approx \chi_w p_w \tag{2.9.2}$$

Effective stress is usually evaluated by using a proper constitutive law which can be written in incremental form as follows:

$$d\boldsymbol{\sigma}'' = \mathbf{D}\left(d\mathbf{e} - d\mathbf{e}^0\right) \tag{2.9.3}$$

with $\mathbf{D}$ being the tangent matrix which depends on state variables and loading history while $\mathbf{e}^0$ corresponds to initial strains which may be attributed to temperature fluctuations or other phenomena such as creeping.

The main variables of the problem are solid matrix displacements $\mathbf{u}$ and pore pressures p_w. Effective stresses $\boldsymbol{\sigma}''$can be evaluated incrementally at any stage and pore pressures p_w also define saturation degree S_w and effective area χ_w as seen in section 2.3. In many cases, the following simplification is adopted:

$$\chi_w = S_w \tag{2.9.4}$$

Finally, the combined fluid equilibrium and mass/flow conservation equation is used as follows:

$$\nabla^T \mathbf{k}\left(-\nabla p_w + S_w \rho_w \mathbf{b}\right) + a\mathbf{m}\dot{\varepsilon} + \frac{\dot{p}}{Q^*} + \dot{s}_0 = 0 \tag{2.9.5}$$

with $\mathbf{k} = \mathbf{k}\left(S_w\right)$.

Eq. (2.9.5) is further simplified compared to eq. (2.7.12) by omitting the term corresponding to the acceleration of the solid phase as influence of this term to the solution accuracy is insignificant [17].

Finally, in order to completely describe the problem, boundary conditions are introduced which are as follows:

$$\mathbf{t} = \overline{\mathbf{t}} \text{ at the boundary } \Gamma = \Gamma_t \tag{2.9.6}$$

$$\mathbf{u} = \overline{\mathbf{u}} \text{ at the boundary } \Gamma = \Gamma_u \tag{2.9.7}$$

and

$$p = \overline{p} \text{ at the boundary } \Gamma = \Gamma_p \tag{2.9.8}$$

$$\mathbf{w} = \mathbf{k}\left(-\nabla p_w + S_w \rho_f \mathbf{b}\right) = \overline{\mathbf{w}}_n \text{ at the boundary } \Gamma = \Gamma_w \tag{2.9.9}$$

Spatial discretization with respect to variables $\mathbf{u}$ and p is implemented by choosing the proper shape functions such as:

$$\mathbf{u} \cong \mathbf{u}^h = \sum_{k=1}^{n} N_k^u \overline{\mathbf{u}}_k = \mathbf{N}^u \overline{\mathbf{u}} \tag{2.9.10}$$

$$p_w \cong p_w^h = \sum_{k=1}^{m} N_k^p \overline{\mathbf{p}}_k^w = \mathbf{N}^p \overline{\mathbf{p}}^w \tag{2.9.11}$$

It is assumed that the above equations are properly constructed so that strong boundary conditions for $\Gamma = \Gamma_p \cup \Gamma_u$ are automatically satisfied by assigning proper values at the nodal parameters. As for the case of most problem formulations with the application of the finite element method, natural boundary conditions will be derived by integrating the weighted equation by parts.

For the derivation of the first of spatially discretized equations, eq. (2.6.2) is multiplied with $\left(\mathbf{N}^u\right)^T$ and is subsequently integrated, obtaining the following:

$$\int_\Omega \mathbf{B}^T \boldsymbol{\sigma} d\Omega + \left(\int_\Omega \left(\mathbf{N}^u\right)^T \rho \mathbf{N}^u d\Omega\right)\ddot{\mathbf{u}} = \mathbf{f}^{(1)} \tag{2.9.12}$$

where

$$\mathbf{B} = \mathbf{S}\mathbf{N}^u \tag{2.9.13}$$

and

$$\mathbf{f}^{(1)} = \int_{\Omega} \left(\mathbf{N}^u\right)^T \rho \mathbf{b} d\Omega + \int_{\Gamma_t} \left(\mathbf{N}^u\right)^T \overline{\mathbf{t}} d\Gamma \tag{2.9.14}$$

which is called load vector. This vector has the same dimensions as the displacement vector $\overline{\mathbf{u}}$ and contains all loads due to traction and field forces.

Using eq. (2.3.18) and taking partial saturation under consideration, the following equation is obtained:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}'' - a\chi_w \mathbf{m}^T p \tag{2.9.15}$$

By combining eqs. (2.9.12) and (2.9.15) the following is obtained:

$$\mathbf{M}\ddot{\overline{\mathbf{u}}} + \int_{\Omega} \mathbf{B}^{\mathbf{T}} \boldsymbol{\sigma}'' d\Omega - \mathbf{Q}\overline{\mathbf{p}}^w - \mathbf{f}^{(1)} = 0 \tag{2.9.16}$$

where

$$\mathbf{M} = \int_{\Omega} \left(\mathbf{N}^u\right)^T \rho \mathbf{N}^u d\Omega \tag{2.9.17}$$

is the mass matrix of the system and

$$\mathbf{Q} = \int_{\Omega} \mathbf{B}^T a\chi_w \mathbf{m} \mathbf{N}^p d\Omega \tag{2.9.18}$$

is the coupling matrix and couples the equations of equilibrium and flow.

Calculation of stresses is done incrementally thus providing the discretized form of eq. (2.9.3) as follows:

$$d\boldsymbol{\sigma}'' = \mathbf{D}\left(\mathbf{B}d\overline{\mathbf{u}} - d\mathbf{e}^0\right) \tag{2.9.19}$$

Finally, discretization of eq. (2.9.5) is done by multiplying the latter equation with $\left(\mathbf{N}^u\right)^T$ and integrating by parts so that:

$$\tilde{\mathbf{Q}}\dot{\overline{\mathbf{u}}} + \mathbf{H}\overline{\mathbf{p}}^w + \tilde{\mathbf{S}}\dot{\overline{\mathbf{p}}}^w - \mathbf{f}^{(2)} = 0 \tag{2.9.20}$$

where

$$\tilde{\mathbf{Q}} = \int_{\Omega} \mathbf{B}^T a \mathbf{m} \mathbf{N}^p d\Omega \tag{2.9.21}$$

is the secondary coupling matrix,

$$\mathbf{H} = \int_{\Omega} \left(\nabla \mathbf{N}^{p}\right)^{T} k \nabla \mathbf{N}^{p} d\Omega \tag{2.9.22}$$

is the permeability matrix

$$\tilde{\mathbf{S}} = \int_{\Omega} \left(\mathbf{N}^{p}\right)^{T} \frac{1}{Q^{*}} \mathbf{N}^{p} d\Omega \tag{2.9.23}$$

is the saturation matrix with

$$\frac{1}{Q^{*}} = C_{s} + \frac{nS_{w}}{K_{f}} + \frac{(a-n)\chi_{w}}{K_{S}} \tag{2.9.24}$$

and

$$\mathbf{f}^{(2)} = -\int_{\Omega} \left(\nabla \mathbf{N}^{p}\right)^{T} k S_{w} \rho_{w} \mathbf{b} d\Omega + \int_{\Gamma_{t}} \left(\mathbf{N}^{p}\right)^{T} \overline{\mathbf{q}} d\Gamma \tag{2.9.25}$$

is the secondary load vector.

If the solid matrix behaves linearly, the effective stress can be written as

$$\boldsymbol{\sigma}'' = \mathbf{D}\mathbf{B}\overline{\mathbf{u}} \tag{2.9.26}$$

In such a case, eqs. (2.9.16) and (2.9.20) can be written as follows:

$$\mathbf{M}\ddot{\overline{\mathbf{u}}} + \mathbf{K}\overline{\mathbf{u}} - \mathbf{Q}\overline{\mathbf{p}} - \mathbf{f}^{(1)} = 0 \tag{2.9.27}$$

$$\tilde{\mathbf{Q}}\dot{\overline{\mathbf{u}}} + \mathbf{H}\overline{\mathbf{p}} + \tilde{\mathbf{S}}\dot{\overline{\mathbf{p}}} - \mathbf{f}^{(2)} = 0 \tag{2.9.28}$$

respectively with $\overline{\mathbf{p}} = \overline{\mathbf{p}}^{w}$ and $\mathbf{K} = \int_{\Omega} \mathbf{B}^{T} \mathbf{D} \mathbf{B} d\Omega$ being the well-known finite element stiffness matrix which, along with matrices $\tilde{\mathbf{S}}$ and $\mathbf{H}$, is symmetric. The above system of eqs. (2.9.27) and (2.9.28) can be written as:

$$\widehat{\mathbf{M}}\ddot{\boldsymbol{\upsilon}} + \widehat{\mathbf{C}}\dot{\boldsymbol{\upsilon}} + \widehat{\mathbf{K}}\boldsymbol{\upsilon} = \mathbf{r} \tag{2.9.29}$$

where

$$\widehat{\mathbf{M}} = \begin{bmatrix} \mathbf{M} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{2.9.30}$$

$$\widehat{\mathbf{C}} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \tilde{\mathbf{Q}}^{\mathbf{T}} & \tilde{\mathbf{S}} \end{bmatrix} \tag{2.9.31}$$

$$\widehat{\mathbf{K}} = \begin{bmatrix} \mathbf{K} & -\mathbf{Q} \\ \mathbf{0} & \mathbf{H} \end{bmatrix} \tag{2.9.32}$$

and

$$\boldsymbol{\upsilon} = \begin{bmatrix} \overline{\mathbf{u}} \\ \overline{\mathbf{p}} \end{bmatrix} \tag{2.9.33}$$

If damping of the solid matrix is also taken into account, eq. (2.9.31) can be rewritten as:

$$\widehat{\mathbf{C}} = \begin{bmatrix} \tilde{\mathbf{C}} & \mathbf{0} \\ \tilde{\mathbf{Q}}^{\mathrm{T}} & \tilde{\mathbf{S}} \end{bmatrix} \tag{2.9.34}$$

where $\tilde{\mathbf{C}}$ is the damping matrix of the solid matrix.

2.10 TEMPORAL DISCRETIZATION OF THE $\mathbf{u}-p$ FORMULATION

Eq. (2.9.29) depicts a system of second order differential equations. Implicit integration methods are excellent candidates for solving such equations where, instead of satisfying eq. (2.9.29) in the whole time continuum, it is enough for the equation to be satisfied in discrete finite temporal points t_k which have a temporal distance Δt. This means that static equilibrium is imposed which also takes into account inertial and damping forces. Such a modified static equilibrium problem can be solved utilizing well known solution procedures used for static problems. Moreover, change of vector $\boldsymbol{\upsilon}$ along with its first and second time derivatives at the time span equal to Δt is assumed to be known. The formulas for calculating these changes define the accuracy, stability and cost of each implicit integration method.

Assuming that vectors $\boldsymbol{\upsilon}$, $\dot{\boldsymbol{\upsilon}}$ και $\ddot{\boldsymbol{\upsilon}}$ are known at time 0 and equal to $\boldsymbol{\upsilon}_0$, $\dot{\boldsymbol{\upsilon}}_0$ and $\ddot{\boldsymbol{\upsilon}}_0$, we seek the solution of eq. (2.9.29) for the time interval from 0 to *T*. During the solution procedure, this time interval is being divided into *n* equal time intervals so that the following stands:

$$\Delta t = \frac{T}{n} \tag{2.10.1}$$

while the chosen integration method is providing approximate solutions of eq. (2.9.29) for each time step $0, \Delta t, 2\Delta t, ..., t, t+\Delta t, ...T$. During the formulation of the aforementioned integration methods, we assume that the solution for each time step $0, \Delta t, 2\Delta t, ..., t$ is known and the evaluation of the solution for time step $t+\Delta t$ is required. The necessary calculations for evaluating the solution at time step $t+\Delta t$ are the same with the ones required for evaluating the solution at a time distance Δt from the current time step and this way, the solution for all time intervals from 0 to *T* is evaluated.

The Newmark method which belongs to the direct integration family of methods, is the most suitable method for solving dynamic problems that are formulated using the **u**-*p* formulation [18, 19].

Let:

$$\dot{\upsilon}_{t+\Delta t} = \dot{\upsilon}_t + \left[(1-\delta)\ddot{\upsilon}_t + \delta\ddot{\upsilon}_{t+\Delta t}\right]\Delta t \tag{2.10.2}$$

$$\upsilon_{t+\Delta t} = \upsilon_t + \dot{\upsilon}_t\Delta t + \left[\left(\frac{1}{2}-\alpha\right)\ddot{\upsilon}_t + \alpha\ddot{\upsilon}_{t+\Delta t}\right]\Delta t^2 \tag{2.10.3}$$

where *α* and *δ* are parameters for which the values are selected depending on the problem and the required accuracy and stability of the method. If $\alpha = \frac{1}{4}$ and $\delta = \frac{1}{2}$ then, the method is unconditionally stable and is named as constant-average acceleration method [20].

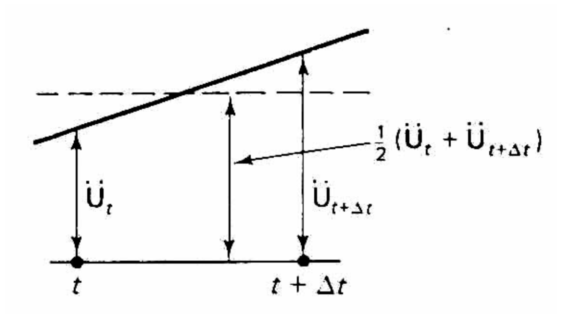


Figure 2.6 - The constant-average acceleration method

Applying eq. (2.9.29) for time step $t+\Delta t$, the following is obtained:

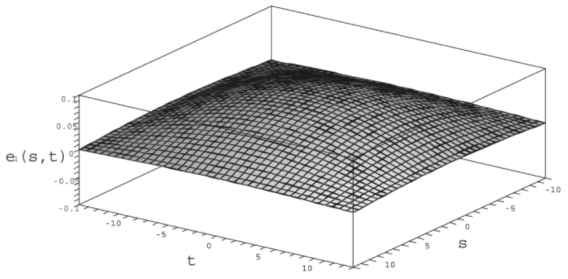
$$\widehat{\mathbf{M}}\ddot{\boldsymbol{\upsilon}}_{t+\Delta t} + \widehat{\mathbf{C}}\dot{\boldsymbol{\upsilon}}_{t+\Delta t} + \widehat{\mathbf{K}}\boldsymbol{\upsilon}_{t+\Delta t} = \mathbf{R}_{t+\Delta t} \tag{2.10.4}$$

By using eqs. (2.10.2) and (2.10.3) along with eq. (2.10.4), $\boldsymbol{\upsilon}_{t+\Delta t}$ is evaluated and using this evaluation, $\dot{\boldsymbol{\upsilon}}_{t+\Delta t}$ and $\ddot{\boldsymbol{\upsilon}}_{t+\Delta t}$ are evaluated.

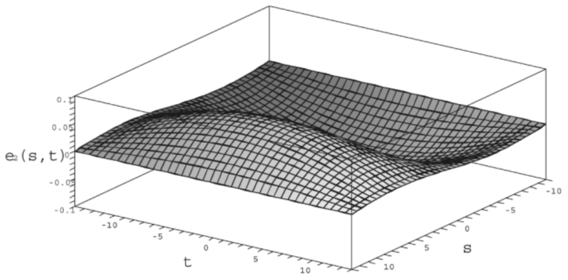
The algorithm used for implementing the Newmark method is as follows:

- The quantities $\boldsymbol{\upsilon}_0$, $\dot{\boldsymbol{\upsilon}}_0$ και $\ddot{\boldsymbol{\upsilon}}_0$, the time step Δt and the values for parameters *α* and *δ* so that $\delta \geq \frac{1}{2}$ and $\alpha \geq \frac{1}{4}\left(\frac{1}{2}+\delta\right)^2$ are chosen.

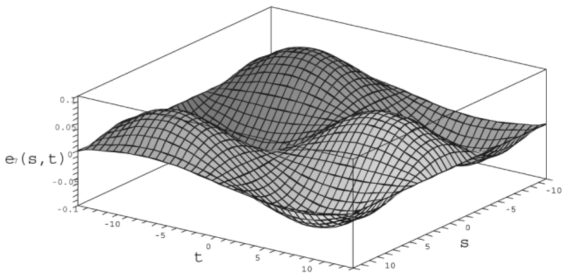
- The following constants are evaluated: $\alpha_0 = \dfrac{1}{\alpha \Delta t^2}$, $\alpha_1 = \dfrac{\delta}{\alpha \Delta t}$, $\alpha_2 = \dfrac{1}{\alpha \Delta t}$, $\alpha_3 = \dfrac{1}{2\alpha} - 1$, $\alpha_4 = \dfrac{\delta}{\alpha} - 1$, $\alpha_5 = \dfrac{\Delta t}{2}\left(\dfrac{\delta}{\alpha} - 2\right)$, $\alpha_6 = \Delta t(1-\delta)$, $\alpha_7 = \delta \Delta t$
- The effective stiffness matrix is formulated which is equal to: $\breve{\mathbf{K}} = \alpha_0 \widehat{\mathbf{M}} + \alpha_1 \widehat{\mathbf{C}} + \widehat{\mathbf{K}}$
- For each time step, the effective loads at time step $t + \Delta t$ are evaluated which are equal to: $\breve{\mathbf{R}}_{t+\Delta t} = \mathbf{R}_{t+\Delta t} + \widehat{\mathbf{M}}(\alpha_0 \boldsymbol{\upsilon}_t + \alpha_2 \dot{\boldsymbol{\upsilon}}_t + \alpha_3 \ddot{\boldsymbol{\upsilon}}_t) + \widehat{\mathbf{C}}(\alpha_1 \boldsymbol{\upsilon}_t + \alpha_4 \dot{\boldsymbol{\upsilon}}_t + \alpha_5 \ddot{\boldsymbol{\upsilon}}_t)$, the linear system $\breve{\mathbf{K}} \boldsymbol{\upsilon}_{t+\Delta t} = \breve{\mathbf{R}}_{t+\Delta t}$ is solved and the time derivatives are being evaluated using the equations: $\ddot{\boldsymbol{\upsilon}}_{t+\Delta t} = \alpha_0 (\boldsymbol{\upsilon}_{t+\Delta t} - \boldsymbol{\upsilon}_t) - \alpha_2 \dot{\boldsymbol{\upsilon}}_t - \alpha_3 \ddot{\boldsymbol{\upsilon}}_t$ and $\dot{\boldsymbol{\upsilon}}_{t+\Delta t} = \dot{\boldsymbol{\upsilon}}_t + \alpha_6 \ddot{\boldsymbol{\upsilon}}_t + \alpha_7 \ddot{\boldsymbol{\upsilon}}_{t+\Delta t}$.



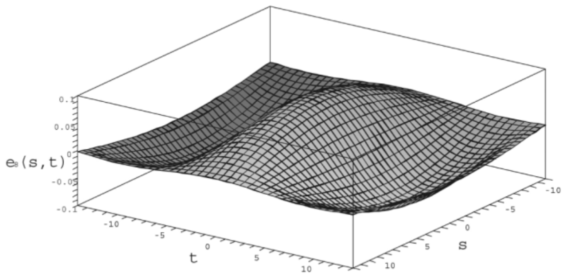
The first eigen-function



The second eigen-function



The seventh eigen-function



The eighth eigen-function

3.1 STOCHASTIC MECHANICS APPROACHES

Modeling a mechanical system can be defined as the mathematical idealization of the physical processes governing its evolution. This requires the definitions of basic variables like system geometry, loading and material properties, response variables like displacement, strain and stresses and the relationships between them. A lot of effort has been made on improving structural models and constitutive laws and with the development of computer science, a great amount of work has been devoted to numerically evaluate approximated solutions of the boundary value problems describing the mechanical system.

The finite element method is one of the most popular approaches for solution of these problems. However the increasing accuracy of the constitutive models and the constant enhancement of available computational tools do not solve the problem of identification of the model parameters and the uncertainties associated with their estimation. Moreover, in most structural engineering applications, the intrinsic randomness of materials or loads is such that deterministic models using average characteristics lead to rough representations of real-life behavior. One of the tasks of stochastic or probabilistic mechanics which has developed rapidly over the last years is accounting for randomness and spatial variability of the mechanical properties of material.

The existing theories for stochastic mechanics approaches are classified with respect to the type of results they provide, as follows:

1. Theories aiming at calculating the first two statistical moments of the response quantities, i.e. the mean, variance and correlation coefficients. These theories are mainly based on the perturbation method.
2. Reliability methods, aiming at evaluating the probability of failure of the system. These methods are based on the definition of a limit state function. As failure is usually associated with rare events, the tails of the probability density functions (PDFs) of response quantities are of interest in this matter.
3. Stochastic finite element methods aiming at evaluating the global probabilistic structure of the response quantities considered as random processes. In this work, the so-called spectral stochastic finite element methods (SSFEM) are considered.

The above theories and methods may overlap since results obtained as byproducts of the main analysis tend to break the walls between these classes, i.e.:

- By means of sensitivity analysis, it is always possible to compute the probability density function (PDF) of a response quantity after the main reliability analysis.
- The expression of response random processes obtained by SSFEM is generally not used directly. Closed form expressions yield the second-moment statistics, and the PDFs can be obtained by simulation.

3.2 RANDOM FIELD DISCRETIZATION

The engineering applications of Computational Structural Mechanics require representation of uncertainties in the mechanical properties of continuous media. The mathematical theory of this representation is called random fields.

The observation of a random phenomenon is called a trial. All the possible outcomes of a trial form the sample space of the phenomenon, denoted hereinafter by **Θ**. An event E is defined as a subset of **Θ** containing outcomes $\theta \in \Theta$. Probability theory aims at associating numbers to events, namely their probability of occurrence.

Let P denote this probability measure. The collection of possible events having well-defined probabilities is called the σ-algebra associated with **Θ**, denoted here by F. Finally the probability space constructed by means of these notions is denoted by (**Θ**, F, P).

A real random variable X is a mapping $X:(\Theta, F, P) \rightarrow \mathbb{R}$. For continuous random variables, the probability density function (PDF) and cumulative distribution function (CDF) are denoted by $f_X(x)$ and $F_X(x)$ respectively, with the subscript X being possibly dropped when there is no risk of ambiguity. In order to emphasize on the random nature of X, the dependency on the outcomes may be added in some cases as in X(θ). A random vector is a collection of random variables.

The mathematical expectation will be denoted by $E[\cdot]$. The mean, variance and n-th moment of X are given by:

$$\mu \equiv E[X] = \int_{-\infty}^{\infty} x f_X(x)\,dx \tag{3.2.1}$$

$$\sigma^2 = \mathrm{E}\left[(X-\mu)^2\right] = \int_{-\infty}^{\infty} (x-\mu)^2 f_X(x)\,dx \tag{3.2.2}$$

$$E\left[X^n\right] = \int_{-\infty}^{\infty} x^n f_X(x)\,dx \tag{3.2.3}$$

Furthermore, the covariance of two random variables X and Y is:

$$\mathrm{Cov}[X,Y] = E\left[(X-\mu_X)(Y-\mu_Y)\right] \tag{3.2.4}$$

Introducing the joint distribution $f_{X,Y}(x,y)$ of these variables, eq. (3.2.4) can be rewritten as:

$$\mathrm{Cov}[X,Y] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (x-\mu_X)(y-\mu_Y)\, f_{X,Y}(x,y)\,dx\,dy \tag{3.2.5}$$

3.3 RELATED HILBERT SPACES

The vector space of real random variables with finite second moment ($E\left[X^2\right]<\infty$) is denoted by $L^2\left(\Theta,F,P\right)$. The expectation operation allows the definition of an inner product with the corresponding norm as follows:

$$\langle X,Y\rangle \equiv E\left[XY\right] \tag{3.3.1}$$

$$\|X\| = \sqrt{E\left[X^2\right]} \tag{3.3.2}$$

$L^2\left(\Theta,F,P\right)$ is complete [21], which makes it a Hilbert space.

A random field $H\left(x,\theta\right)$ can be defined as a curve in $L^2\left(\Theta,F,P\right)$; that is a collection of random variables indexed by a continuous parameter $x\in\Omega$, where Ω is an open set of $\mathbb{R}^d$, describing the system geometry. This means that for a given x_0, $H\left(x_0,\theta\right)$ is a random variable while for a given outcome θ_0, $H\left(x,\theta_0\right)$ is a realization of the field. This realization is assumed to be an element of the Hilbert space $L^2\left(\Omega\right)$ of square integrable functions over Ω with the natural inner product associated with $L^2\left(\Omega\right)$ being defined by:

$$\langle f,g\rangle_{L^2(\Omega)} = \int_\Omega f\left(x\right)g\left(x\right)d\Omega \tag{3.3.3}$$

Hilbert spaces have convenient properties to develop approximate solutions of boundary value problems, such as the Galerkin procedure. A random field is called univariate or multivariate, depending on whether the quantity H(x) attached to point x is a random variable or a random vector. It can be one- or multi- dimensional according to the dimension d of x, that is d = 1 or d > 1. In this work, we consider univariate multidimensional fields which corresponds to the modeling of mechanical properties including Young's modulus, Poisson's ratio, yield stress, etc., as statistically independent fields.

The random field is Gaussian if any vector $\left[H\left(x_1\right) \quad \dots \quad H\left(x_n\right)\right]$ is Gaussian. A Gaussian field is completely defined by its mean $\mu\left(x\right)$, variance $\sigma^2\left(x\right)$ and autocorrelation coefficient $\rho\left(x,x'\right)$ functions. Moreover, it is homogeneous if the mean and variance are constant and ρ is a function of the difference x_0-x only. This one-argument function is being denoted by $\tilde{\rho}\left(\cdot\right)$. The correlation length is a characteristic parameter appearing in the definition of the correlation function and for one-dimensional homogeneous fields, the power spectrum is defined as the Fourier transform of the autocorrelation function, as follows:

$$S_{HH}(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{\rho}(x) e^{-i\omega x} dx \tag{3.3.4}$$

A discretization procedure is the approximation of $H(\cdot)$ by $\widehat{H}(\cdot)$, defined by means of a finite set of random variables $\chi_i,\ i = 1, \ldots n$, grouped in a random vector denoted by **χ**:

$$H(x) \overset{Discretization}{\rightarrow} \widehat{H}(x) = F[x, \chi] \tag{3.3.5}$$

It is essential to define the most efficient approximation while minimizing the approximation error with respect to some error estimator; that is the one using the minimal number of random variables. The discretization methods can be divided into three groups:

- Point discretization, where the random variables χ_i are selected values of $H(\cdot)$ at some given points x_i.
- Average discretization, where the random variables χ_i are weighted integrals of $H(\cdot)$ over a domain Ω_e so that $\chi_i = \int_{\Omega_e} H(x)\omega(x)d\Omega$
- Series expansion methods, where the field is exactly represented as a series involving random variables and deterministic spatial functions. The approximation is then obtained as a truncation of the series

In this work, a series expansion method called Karhunen-Loeve is used for the discretization of the random field.

3.4 THE KARHUNEN-LOÈVE EXPANSION

The Karhunen-Loève (KL) expansion [22] of a random field $H(\cdot)$ is based on the spectral decomposition of its autocovariance function $C_{HH}(x, x') = \sigma(x)\sigma(x')\rho(x, x')$. The set of deterministic functions over which any realization of the field $H(x, \theta_0)$ is expanded is defined by the eigenvalue problem:

$$\forall i = 1, \ldots \quad \int_{\Omega} C_{HH}(x, x') \varphi_i(x') d\Omega_{x'} = \lambda_i \varphi_i(x) \tag{3.4.1}$$

Eq. (3.4.1) is a Fredholm integral equation with the kernel $C_{HH}(\cdot,\cdot)$ being an autocovariance function and as a result, being bounded, symmetric and positive definite. Thus, the set of $[\varphi_i]$ form a complete orthogonal basis of $L^2(\Omega)$ while the set of eigenvalues (spectrum) is real, positive, numerable, its only possible accumulation point is zero. Any realization of $H(\cdot)$ can thus be expanded over this basis as follows:

$$H(x,\theta)=\mu(x)+\sum_{i=1}^{\infty}\sqrt{\lambda_i}\xi_i(\theta)\varphi_i(x) \tag{3.4.2}$$

where $\xi_i(\theta)$ denotes the coordinates of the realization of the random field with respect to the set of deterministic functions $[\varphi_i]$. Taking now into account all possible realizations of the field, $\xi_i,\ i=1,\ldots$ becomes a numerable set of random variables.

When calculating $\mathrm{Cov}[H(x),H(x')]$ using eq. (3.4.2) and requiring it to be equal to $C_{HH}(x,x')$, the following stands:

$$E[\xi_k\xi_l]=\delta_{kl} \tag{3.4.3}$$

where δ_{kl} denotes the Kronecker symbol. This means that $\xi_i,\ i=1,\ldots$ forms a set of orthonormal random variables with respect to the inner product of eq. (3.3.1), making eq. (3.4.2) to correspond to a separation of the space and randomness variables in $H(x,\theta)$.

3.5 KARHUNEN-LOÈVE PROPERTIES

The KL expansion possesses the following properties:

- Due to non-accumulation of eigenvalues around a non-zero value, it is possible to order them in a descending series converging to zero. Truncating the ordered series (3.4.2) after the M-th term, the following KL-approximated field is obtained:

$$\hat{H}(x,\theta)=\mu(x)+\sum_{i=1}^{M}\sqrt{\lambda_i}\xi_i(\theta)\varphi_i(x) \tag{3.5.1}$$

- The covariance eigenfunction basis $\varphi_i(x)$ is optimal in the sense that the mean square error (integrated over Ω), resulting from a truncation after the M-th term, is minimized with respect to the value it would take when any other complete basis $h_i(x)$ is chosen.
- The set of random variables appearing in (3.4.2) is orthonormal, i.e. verifying (3.4.3), if and only if the basis functions $h_i(x)$ and the constants λ_i are solution of the eigenvalue problem (3.4.1).
- Due to the orthonormality of the eigenfunctions, it is easy to get a closed form for each random variable appearing in the series according to the following linear transform:

$$\xi_i(\theta)=\frac{1}{\sqrt{\lambda_i}}\int_{\Omega}\left[H(x,\theta)-\mu(x)\right]\varphi_i(x)\,d\Omega \tag{3.5.2}$$

Hence when $H(\cdot)$ is a Gaussian random field, each random variable ξ_i is Gaussian. It follows that ξ_i form in this case a set of independent standard normal variables. Furthermore, the KL expansion of Gaussian fields is almost surely convergent [22].

- From eq. (3.5.1), the error variance obtained when truncating the KL expansion after M terms turns out to be after some basic algebra manipulation, as follows:

$$\mathrm{Var}\left[H(x)-\widehat{H}(x)\right]=\sigma^2(x)-\sum_{i=1}^{M}\lambda_i\varphi_i^2(x)=\mathrm{Var}\left[H(x)\right]-\mathrm{Var}\left[\widehat{H}(x)\right] \quad (3.5.3)$$

The right hand side of the above equation is always positive because it is the variance of some quantity. This means that the KL expansion always under-represents the true variance of the field.

3.6 THE INTEGRAL EIGENVALUE PROBLEM

Eq. (3.4.1) can be solved analytically only for few autocovariance functions and geometries of Ω. Detailed closed form solutions for triangular and exponential covariance functions for one-dimensional homogeneous fields can be found in [23, 24], where Ω = [-α, α]. Extension to two-dimensional fields defined for similar correlation functions on a rectangular domain can be obtained as well. Except in these particular cases, the integral eigenvalue problem has to be solved numerically. A Galerkin-type numerical procedure is described below.

Let $h_i(x)$ be a complete basis of the Hilbert space $L^2(\Omega)$. Each eigenfunction of $C_{HH}(x,x')$ may be represented by its expansion over this basis, say:

$$\varphi_k(x)=\sum_{i=1}^{\infty}d_i^k h_i(x) \quad (3.6.1)$$

where d_i^k are the unknown coefficients. The Galerkin procedure aims at obtaining the best approximation of $\varphi_\kappa(\cdot)$ when truncating the above series after the N-th term. This is accomplished by projecting φ_k onto the space H_N spanned by $h_i(\cdot),\ i=1,\ldots,N$. Introducing a truncation of (3.6.1) in (3.4.1), the residual becomes:

$$\varepsilon_N(x)=\sum_{i=1}^{N}d_i^k\left[\int_\Omega C_{HH}(x,x')h_i(x')d\Omega_{x'}-\lambda_k h_i(x)\right] \quad (3.6.2)$$

Requiring the truncated series being the projection of $\varphi_\kappa(\cdot)$ onto H_N implies that this residual is orthogonal to H_N in $L^2(\Omega)$. This implies:

$$\left\langle \varepsilon_N, h_j\right\rangle \equiv \int_\Omega \varepsilon_N(x)h_j(x)d\Omega=0,\ j=1,\ldots,N \quad (3.6.3)$$

After some basic algebra, these conditions reduce to a linear system:

$$CD = \Lambda BD \tag{3.6.4}$$

where the different matrices are defined as follows:

$$B_{ij} = \int_{\Omega} h_i(x) h_j(x) d\Omega \tag{3.6.5}$$

$$C_{ij} = \int_{\Omega}\int_{\Omega} C_{HH}(x, x') h_i(x) h_j(x') d\Omega_x d\Omega_{x'} \tag{3.6.6}$$

$$D_{ij} = d_i^j \tag{3.6.7}$$

$$\Lambda_{ij} = \delta_{ij}\lambda_j \tag{3.6.8}$$

where δ_{kl} denotes the Kronecker symbol. This is a discrete eigenvalue problem which may be solved for eigenvectors D and eigenvalues λ_i. This solution scheme can be implemented using the finite element mesh shape functions as the basis $h_i(\cdot)$ [24].

Due to its useful properties, the KL expansion has been widely used in stochastic finite element approaches. The main issue when using the KL expansion is to solve the eigenvalue problem (3.4.1). In most applications found in the literature, closed form solution based on the exponential autocovariance function in conjunction with square geometries have been applied. However, for industrial applications where complex geometries will be encountered, closed form solutions are not possible and thus the scheme presented in this section for numerically solving (3.4.1) requires additional computations while the obtained approximated basis $\varphi_i(\cdot)$ is no more optimal.

It is therefore recommended, for general geometries, to embed Ω in a square-shape volume and use the latter to solve the eigenvalue problem in a closed form, when possible.

3.7 THE SPECTRAL STOCHASTIC FINITE ELEMENT METHOD

The spectral stochastic finite element method (SSFEM) was proposed by Ghanem and Spanos in [25, 26] and presented in a comprehensive monograph in [24]. It is an extension of the deterministic finite element method (FEM) for boundary value problems involving random material properties.

To understand what kind of discretization is introduced in SSFEM, we consider a deterministic mechanical system with deterministic geometry, material properties and loading. The evolution of such a system is governed by a set of partial differential equations (PDE), the associated boundary conditions and the initial state. When no closed-form solution to these equations exists, a discretization procedure has to be applied in order to handle the problem numerically.

In the standard finite element method, the geometry is replaced by a set of points $x_i,\ i=1,\dots,N$ that correspond to the nodes of the finite element mesh. In the same manner, the response of the system, i.e. the displacement field u(x) is approximated by means of nodal displacements $u_i,\ i=1,\dots,N$ gathered into a vector U. The set of PDE can then be transformed to a system of equations in $\{u_i\}_{i=1}^{N}$.

If a material property such as the Young's modulus is now modeled as a random field, the system will be governed by a set of stochastic PDE, and the response will be the displacement random field $u(x,\theta)$, where θ denotes a basic outcome in the space of all possible outcomes Θ. A spatial discretization procedure such as the one described in the above paragraph results in approximating the response as a random vector of nodal displacements U(θ), each component being a random variable $u^i(\theta)$ yet to be characterized.

A random variable is completely determined by the value it takes for all possible outcomes $\theta \in \Theta$. Adopting the same kind of discretization as for the spatial part would result in selecting a finite set of points $\{\theta_1,\dots\theta_Q\}$ in Θ. The Monte Carlo simulation of the problem corresponds to this kind of strategy. The realizations θ_i have to be selected with some rules to ensure that the space is correctly sampled. It is however well known that an accurate description of the response would require a large value for Q.

SSFEM aims at discretizing the random dimension in a more efficient way using series expansions. Two different procedures are used:

- The input random field is discretized using the truncated KL expansion.
- Each random nodal displacement $u^i(\theta)$ is represented by its coordinates in an appropriate basis of the space of random variables $L^2(\Theta,F,P)$, namely the polynomial chaos.

For the sake of simplicity, rather than presenting SSFEM in a general way, the main ideas are first developed in the following section on a simple example, namely the accounting of the spatial variability of the Young's modulus in an elastic mechanical system. In this case, the deterministic finite element method is assumed to be well-known and only the approximated solution in the random dimension is developed.

3.8 SSFEM IN LINEAR ELASTIC PROBLEMS

Using classical notation, the finite element method in linear elasticity eventually yields a linear system of size NxN (N being the number of degrees of freedom):

$$KU = F \tag{3.8.1}$$

where the global stiffness matrix K is obtained after assembling the element stiffness matrices k^e :

$$k^e = \int_{\Omega_e} B^T D B d\Omega_e \tag{3.8.2}$$

In the above equation, D stands for the elasticity matrix and B is the deformation matrix that relates the components of strain to the element nodal displacements.

Assuming that the material Young's modulus is a Gaussian random field, the elasticity matrix in point x can thus be written as:

$$D(x,\theta) \equiv H(x,\theta) D_0 \tag{3.8.3}$$

where D_0 is a constant matrix. Substituting (3.8.3), (3.4.2) in (3.8.2) yields:

$$k^e(\theta) = k_0^e + \sum_{i=1}^{\infty} k_i^e \xi_i(\theta) \tag{3.8.4}$$

where k_0^e is the mean element stiffness matrix and k_i^e are deterministic matrices obtained by using the KL expansion (eq. (3.5.1)):

$$k_i^e = \sqrt{\lambda_i} \int_{\Omega_e} \varphi_i(x) B^T D_0 B d\Omega_\varepsilon \tag{3.8.5}$$

Assembling the above element contributions provides the stochastic counterpart of the equilibrium eq. (3.8.1):

$$\left[K_0 + \sum_{i=1}^{\infty} K_i \xi_i(\theta) \right] U(\theta) = F \tag{3.8.6}$$

assuming a deterministic load vector F. In the above equation, K_i are deterministic matrices obtained by assembling k_i^e in a way similar to the deterministic case.

The vector of nodal displacements U(θ) is obtained by solving eq. (3.8.6). However no closed-form solution exists for such an inverse. An early strategy adopted in [24] consists in using a Neumann series expansion of the inverse stochastic stiffness matrix to get an approximate response. Eq. (3.8.6) can thus be rewritten as:

$$K_0 \left[I + \sum_{i=1}^{\infty} K_0^{-1} K_i \xi_\iota(\theta) \right] U(\theta) = F \tag{3.8.7}$$

which leads to:

$$U(\theta) = \left[I + \sum_{i=1}^{\infty} K_0^{-1} K_i \xi_\iota(\theta) \right]^{-1} U^0,\ U^0 = K_0^{-1} F \tag{3.8.8}$$

The Neumann series expansion of the above equation has the form:

$$U(\theta)=\sum_{k=0}^{\infty}(-1)^k\left[\sum_{i=1}^{\infty}K_0{}^{-1}K_i\xi_i(\theta)\right]^k U^0 \tag{3.8.9}$$

whose first terms are written explicitly as follows:

$$U(\theta)=\left[I-\sum_{i=1}^{\infty}K_0{}^{-1}K_i\xi_i(\theta)+\sum_{i=1}^{\infty}\sum_{j=1}^{\infty}K_0{}^{-1}K_iK_0{}^{-1}\xi_i(\theta)\xi_j(\theta)+\ldots\right]U^0 \tag{3.8.10}$$

Truncating both the KL and the Neumann expansions (indices i and k in eq. (3.8.9), respectively) yields an approximate solution for U(θ).

Based on eq. (3.8.10), each random displacement $u^i(\theta)$ can be represented as a series of polynomials in the standard normal variables $\left\{\xi_k(\theta)\right\}_{k=1}^{\infty}$. Reordering all terms by means of a single index j, this representation is expressed as:

$$u^i(\theta)=\sum_{j=0}^{\infty}u_j^iP_j\left(\left\{\xi_k(\theta)\right\}_{k=1}^{\infty}\right) \tag{3.8.11}$$

where $P_0\equiv 1$ and $P_j\left(\left\{\xi_k(\theta)\right\}_{k=1}^{\infty}\right)$ are polynomials in standard normal variables, i.e.:

$$P_j\left(\left\{\xi_k(\theta)\right\}_{k=1}^{\infty}\right)=\xi_{i_1}^{\alpha_1}\xi_{i_2}^{\alpha_2}\ldots\xi_{i_p}^{\alpha_p} \tag{3.8.12}$$

The set of $\left\{P_j\right\}_{j=0}^{\infty}$ in eq. (3.8.12) forms a basis of the space of all random variables $L^2(\Theta,F,P)$ and the coefficients u_j^i are interpreted as the coordinates of $u^i(\theta)$ in this basis.

Referring to the inner product defined in $L^2(\Theta,F,P)$ by eq. (3.3.1), the above basis is however not orthogonal. For instance, $\xi_1(\theta)$ and $\xi_1^3(\theta)$ are two basis random variables whose inner product is $E\left[\xi_1^4\right]=3$. For further exploitation of the response, such as computing its moments, an orthogonal basis appears more appealing.

The polynomial chaos proposed in [24] possesses the orthogonality property. The details of its construction are quite technical and are given in the following section. In order to proceed, let us assume that any random variable u(θ) element of $L^2(\Theta,F,P)$ can be represented as follows:

$$u(\theta)=\sum_{j=0}^{\infty}u_j\Psi_j(\theta) \tag{3.8.13}$$

where $\left\{\Psi_j(\theta)\right\}_{j=0}^{\infty}$ is a complete set of orthogonal random variables defined as polynomials in $\left\{\xi_k(\theta)\right\}_{j=0}^{\infty}$, satisfying:

$$\Psi_0 \equiv 1 \tag{3.8.14}$$

$$E\left[\Psi_j\right] = 0,\ j > 0 \tag{3.8.15}$$

$$E\left[\Psi_j(\theta)\Psi_k(\theta)\right] = 0,\ j \neq k \tag{3.8.16}$$

The expansion of the nodal displacements vector is consequently written as:

$$U(\theta) = \sum_{j=0}^{\infty} U_j \Psi_j(\theta) \tag{3.8.17}$$

with the coordinates U_j being deterministic vectors having N components. Note that the first term U_0 in the above equation is different from the first term in the Neumann expansion (3.8.10). The latter, denoted by U^0, is the one obtained by a perturbation approach.

By denoting $\xi_0(\theta) \equiv 1$ and substituting the above equation in eq. (3.8.6), the following is obtained:

$$\left(\sum_{i=0}^{\infty} K_i \xi_i(\theta)\right)\left(\sum_{i=0}^{\infty} U_j \Psi_j(\theta)\right) - F = 0 \tag{3.8.18}$$

For computational purposes, the series involved in eq. (3.8.18) are truncated after a finite number of terms, denoted by (M+1) for the stiffness matrix expansion (KL expansion) and by P for the displacements vector expansion. As a result, the residual in eq. (3.8.18) due to the truncation reads:

$$\varepsilon_{M,P} = \sum_{i=0}^{M}\sum_{j=0}^{P-1} K_i U_j \xi_i(\theta)\Psi_j(\theta) - F \tag{3.8.19}$$

The best approximation of the exact solution U(θ) in the space H_P spanned by $\left\{\Psi_k(\theta)\right\}_{k=0}^{P-1}$ is obtained by minimizing this residual in a mean square sense. In the Hilbert space $L^2(\Theta, F, P)$, this is equivalent to requiring this residual to be orthogonal to H_P, yielding:

$$E\left[\varepsilon_{M,P}\Psi_K\right] = 0,\ k = 0,\ldots,P-1 \tag{3.8.20}$$

Let us introduce the following notation:

$$c_{ijk} = E\left[\xi_i \quad \Psi_j \quad \Psi_k\right] \tag{3.8.21}$$

$$F_k = E\left[\Psi_k \quad F\right] \qquad (3.8.22)$$

Note that in the case of deterministic loading, as considered in this work, F_k is zero for k > 0. Using eq. (3.8.19), eq. (3.8.20) can be rewritten as:

$$\sum_{i=0}^{M}\sum_{j=0}^{P-1} c_{ijk} K_i U_j = F_k,\ k = 0,\ldots,P-1 \qquad (3.8.23)$$

For the sake of simplicity, let us define:

$$K_{jk} = \sum_{i=0}^{M} c_{ijk} K_i \qquad (3.8.24)$$

Hence, eq. (3.8.23) can be written as:

$$\sum_{j=0}^{P-1} K_{jk} U_j = F_k,\ k = 0,\ldots,P-1 \qquad (3.8.25)$$

In the above equations, each U_j is a N-dimensional vector and each K_{jk} a matrix of size NxN. The P different equations can be cast in a linear system of size NPxNP as follows:

$$\begin{bmatrix} K_{00} & \cdots & K_{0,P-1} \\ K_{10} & \cdots & K_{1,P-1} \\ \vdots & & \vdots \\ K_{P-1,0} & \cdots & K_{P-1,P-1} \end{bmatrix} \begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{P-1} \end{bmatrix} = \begin{bmatrix} F_0 \\ F_1 \\ \vdots \\ F_{P-1} \end{bmatrix} \qquad (3.8.26)$$

After solving this system for $\bar{U} = \{U_k, k = 0,\ldots,P-1\}$, the best approximation for U(θ) in the subspace H_P spanned by $\{\Psi_k(\theta)\}_{k=0}^{P-1}$ is given by:

$$U(\theta) = \sum_{j=0}^{P-1} U_j \Psi_j(\theta) \qquad (3.8.27)$$

The dimension P of HP usually varied between 10 to 35 in real world applications. This means that any nodal displacement is characterized as a random variable by 15-35 coefficients. The amount of computation required for solving the linear system (3.8.26) is thus orders of magnitude greater than that required for the deterministic analysis of the same problem.

The coefficients themselves in eq. (3.8.27) do not provide a clear interpretation of the response randomness. The following useful quantities are however readily obtained:

- The mean nodal displacement vector E [U] is the first term of the expansion, namely U_0, since $E\left[\Psi_j(\theta)\right] = 0$ for j > 0.
- The covariance matrix of the components of vector U is:

$$\mathrm{Cov}[U,U]=\sum_{i=1}^{P-1}E\left[\Psi_i^2\right]U_iU_i^T \tag{3.8.28}$$

with the coefficients $E\left[\Psi_i^2\right]$ being easily computed due to the definition of the Ψ_i s, as it will be shown in the following section.

- The probability density function of any component U^i of the nodal displacement vector can be obtained by simulating the basis random variables $\Psi_j(\theta)$, then using eq. (3.8.27). In the case when this equation is limited to quadratic terms (second order polynomial chaos expansion), a closed-form expression for the characteristic function of U has been given in [27], which can then be numerically Fourier-transformed to obtain the required PDF.

As it can be seen in eq. (3.8.26), the size of the linear system resulting from the SSFEM approach increases rapidly with the series cut-off number P. Whenever classical direct methods are used to solve the system, the computational time grows exponentially. This is the reason why early applications of SSFEM were limited to a small problem with limited number of degrees of freedom N.

Eqs. (3.8.24) and (3.8.25) suggest that the global matrix K is completely determined by the matrices K_i and the coefficients c_{ijk}. By not storing K as a whole but instead storing these building blocks K_i along with the c_{ijk} coefficients, the required amount of memory is reduced considerably. Using a second (resp. third) order polynomial chaos in 4-term KL expansion example [28], the proposed method requires 11 times (resp. 33 times) less memory compared to the classical global storage because a large number of coefficients c_{ijk} are zero [24].

Since K_0 corresponds to the stiffness matrix of a system having the mean material properties, we can define in the same way, K_i, i > 0, as the stiffness matrix corresponding to a certain spatial fluctuation of the material properties given by the eigenfunction $\varphi_i(x)$. Since the mean of these fluctuations is zero, assuming that they are bounded within a certain range, the entries of K_0 are expected to be dominant in magnitude. Furthermore, it is easily seen from eq. (3.8.21) that $c_{ojk}\propto\delta_{jk}$ since $\xi_0\equiv 1$ and the Ψ_j 's are orthogonal to each other. This means that K_0 has a contribution only in the K_{jj} blocks that are on the main diagonal of K, as seen in eq. (3.8.24). These arguments confirm the diagonal dominance of K which should be taken into consideration during the solution process. Furhtermore, all matrices K_i all have the same non-zero structure, which can simplify storage handling.

3.9 POLYNOMIAL CHAOS EXPANSION

The polynomial chaos is a particular basis of the space of random variables $L^2\left(\Theta,F,P\right)$ based on Hermite polynomials of standard normal variables. The one-dimensional Hermite polynomials are defined by

$$h_n\left(x\right)=\left(-1\right)^n\frac{d^n\left[e^{-\frac{1}{2}x^2}\right]}{dx^n}e^{\frac{1}{2}x^2} \tag{3.9.1}$$

Hermite polynomials of independent standard normal variables are orthogonal to each other with respect to the inner product of $L^2\left(\Theta,F,P\right)$ defined in eq. (3.3.1), that is:

$$E\left[h_m\left(\xi_i\left(\theta\right)\right)h_n\left(\xi_i\left(\theta\right)\right)\right]=0,\ m\neq n \tag{3.9.2}$$

Multidimensional Hermite polynomials can be defined as products of Hermite polynomials of independent standard normal variables. To further specify their construction, let us consider the following integer sequences:

$$a=\left\{a_1,\dots,a_p\right\},\ a_j\geq 0 \tag{3.9.3}$$

$$i=\left\{i_1,\dots,i_p\right\},\ i_j>0 \tag{3.9.4}$$

The multidimensional Hermite polynomial associated with the sequences (i,α) is written as:

$$\Psi_{i,a}\left(\theta\right)=\prod_{k=1}^{p}h_{a_k}\left(\xi_{i_k}\left(\theta\right)\right) \tag{3.9.5}$$

It turns out that the set $\left\{\Psi_{i,a}\right\}$ of all polynomials associated with all possible sequences (i, α) of any length p forms a basis in $L^2\left(\Theta,F,P\right)$.

For further convenience, let us denote by $\Gamma_p\left(\xi_{i_1}\left(\theta\right),\dots,\xi_{i_p}\left(\theta\right)\right)$ the set of basis polynomials $\left\{\Psi_{i,a}\left(\theta\right)\mid\sum_{k=1}^{p}a_k=p\right\}$ and by Γ_p the space they span. Γ_p is a subspace of $L^2\left(\Theta,F,P\right)$, usually called homogeneous chaos of order p. The subspaces Γ_p are orthogonal to each other in $L^2\left(\Theta,F,P\right)$. This is easily proven by the fact that they are spanned by two sets of $\Psi_{i,a}$ having null intersection. Thus the following relationship, known as the Wiener Chaos decomposition, holds:

$$\bigoplus_{k=0}^{\infty}\Gamma_k=L^2\left(\Theta,F,P\right) \tag{3.9.6}$$

where $\oplus$ denotes the operator of orthogonal summation of subspaces in linear algebra. Consequently the expansion of any random variable u(θ) in the polynomial chaos can be written as:

$$u(\theta)=u_0\Gamma_0+\sum_{i_1=1}^{\infty}u_{i_1}\Gamma_1\left(\xi_{i_1}(\theta)\right)+\sum_{i_1=1}^{\infty}\sum_{i_2=1}^{\infty}u_{i_1i_2}\Gamma_2\left(\xi_{i_1}(\theta)\xi_{i_2}(\theta)\right)+\dots \qquad (3.9.7)$$

In this expression $u_0, u_{i_1}, u_{i_1i_2}$ are the coordinates of u(θ) associated with 0-th, first and second order homogeneous chaoses respectively. The lower order homogeneous chaos have the following closed-form expression:

$$\Gamma_0=1 \qquad (3.9.8)$$

$$\Gamma_1(\xi_i)=\xi_i \qquad (3.9.9)$$

$$\Gamma_2\left(\xi_{i_1},\xi_{i_2}\right)=\xi_{i_1}\xi_{i_2}-\delta_{i_1i_2} \qquad (3.9.10)$$

$$\Gamma_3\left(\xi_{i_1},\xi_{i_2},\xi_{i_3}\right)=\xi_{i_1}\xi_{i_2}\xi_{i_3}-\xi_{i_1}\delta_{i_2i_3}-\xi_{i_2}\delta_{i_3i_1}-\xi_{i_3}\delta_{i_1i_2} \qquad (3.9.11)$$

The polynomial chaos can be related to the (non-orthogonal) basis associated with the Neumann series expansion, as seen on eq. (3.8.12). For this purpose, let us introduce the orthogonal projection π_p of $L^2(\Theta,F,P)$ onto Γ_p. It can be shown that the following relationship holds:

$$\pi_p\left(\xi_{i_1}^{a_1}(\theta)\dots\xi_{i_p}^{a_p}(\theta)\right)=\Psi_{i,a} \qquad (3.9.12)$$

For computational purposes, finite dimensional polynomial chaoses are constructed by means of a finite number M of orthonormal Gaussian random variables. In this work, these variables are selected from the KL expansion of the input random field. The polynomial basis formed by means of these M random variables is denoted by $\Gamma_p\left(\xi_{1,}\dots,\xi_M\right)$ and it is called homogeneous chaos of dimension M and order p.

Following eq. (3.9.5), the basis $\Gamma_p\left(\xi_{1,}\dots,\xi_M\right)$ is generated as follows. To each set of M integers $\{\alpha_1,\dots,\alpha_M\}$ ranging from 0 to p and summing up to p, the following basis vector is associated:

$$\Psi_\alpha=\prod_{i=1}^{M}h_{a_i}(\xi_i) \qquad (3.9.13)$$

This formula allows for a systematic construction of the polynomial chaoses of any order. It can be shown that the dimension of $\Gamma_p\left(\xi_{1,}\dots,\xi_M\right)$ is the binomial factor $\begin{pmatrix}M+p-1\\p\end{pmatrix}$. The lower-dimensional polynomial chaoses (up to M = 4) have been tabulated in [24] for different orders (up to p = 4). As an example, Table 3.9.1 gives the two-dimensional polynomial chaoses for different orders.

When truncating eq. (3.9.7) after order p, the total number of basis polynomials P is given by:

$$P = \sum_{k=0}^{p} \binom{M+k-1}{k} \tag{3.9.14}$$

Table 3.9.2 gives an evaluation of P for certain values of M and p. It is seen that P is increasing extremely fast with both parameters. Remembering that each scalar response quantity u (which was a single number in the deterministic finite element method) is now represented by P coefficients, it is easily seen that SSFEM will require a large amount of computation. This may be worthwhile, considering that the whole probabilistic structure of u is (approximately) contained in these P coefficients.

j	p	j-th basis polynomial Ψ_j
0	p=0	1
1	p=1	ξ_1
2		ξ_2
3	p=2	$\xi_1^2 - 1$
4		$\xi_1 \xi_2$
5		$\xi_2^2 - 1$
6	p=3	$\xi_1^3 - 3\xi_1$
7		$\xi_2 \left(\xi_1^2 - 1 \right)$
8		$\xi_1 \left(\xi_2^2 - 1 \right)$
9		$\xi_2^3 - 3\xi_2$
10	p=4	$\xi_1^4 - 6\xi_1^2 + 3$
11		$\xi_2 \left(\xi_1^3 - 3\xi_1 \right)$
12		$\left(\xi_1^2 - 1 \right)\left(\xi_2^2 - 1 \right)$
13		$\xi_1 \left(\xi_3^2 - 3\xi_2 \right)$
14		$\xi_2^4 - 6\xi_2^2 + 3$

Table 3.9.1 – Two dimensional polynomial chaoses

From a practical point of view, the choice of M is dictated by the discretization of the input random fields. In the original SSFEM, the Karhunen-Loève expansion is used under the assumption that the input field is Gaussian. The choice of M is thus directly related to the accuracy requested in this random field discretization. The higher M, the better higher frequency random fluctuations of the input will be taken into account. Conversely, parameter p governs the order of non-linearity captured in describing the solution process. Typical values used are M = 4 and p = 2, 3.

M	p=1	p=2	p=3	p=4
2	3	6	10	15
4	5	15	35	70
6	7	28	83	210

Table 3.9.2 – Number of basis polynomials P (M: number of basis random variables, p: order of homogeneous chaos expansion)

3.10 SSFEM WITH A LOG-NORMAL INPUT RANDOM FIELD

The use of Gaussian random fields is quite common in the context of probabilistic mechanics. However these fields are not well suited to modeling material properties like Young's modulus or yield stress which are by their nature positive values. Indeed, for large coefficients of variation, realizations of the field could include negative outcomes that are physically meaningless. In contrast, the log-normal field appears attractive in this sense. A lognormal field can be defined by a transformation of a Gaussian field g(x) as:

$$l(x) = e^{g(x)} \tag{3.10.1}$$

The Karhunen-Loève expansion of a log-normal field, although possible, is of no practical interest since the probabilistic structure of the random variables $\{\xi_i\}$ appearing in the expansion cannot be determined. In order to be able to include log-normal fields in the SSFEM approach, it was proposed in [29] to expand them into the polynomial chaos.

Let us first consider a single lognormal random variable obtained as follows:

$$l = e^{\mu_g + \sigma_g \xi} \tag{3.10.2}$$

where ξ is a standard normal variable. The polynomial chaos expansion of l l reads:

$$l = \sum_{i=0}^{\infty} l_i \Psi_i(\xi) \tag{3.10.3}$$

where $\Psi_i(\xi)$ is the i-th Hermite polynomial in this case. Due to the orthogonality properties of the Ψ_i s, the coefficients l_i can be obtained as:

$$l_i = \frac{E\left[e^{\mu_g+\sigma_g\xi}\Psi_i\left(\xi\right)\right]}{E\left[\Psi_i^2\right]} = \frac{E\left[\Psi_i\left(\xi+\sigma_g\right)\right]}{E\left[\Psi_i^2\right]} e^{\mu_g+\frac{1}{2}\sigma_g^2} \tag{3.10.4}$$

However, the fraction in the above equation turns out to be equal to $\frac{\sigma_g^i}{i!}$ after some algebra, whereas the exponential term is nothing but the mean value of l, denoted by μ_l. Thus the expansion of any log-normal random variable into the (one-dimensional) polynomial chaos reduces to:

$$l = \mu_l \sum_{i=0}^{\infty} \frac{\sigma_g^i}{i!} \Psi_i\left(\xi\right) \tag{3.10.5}$$

Let us now consider the approximate log-normal field l(x) defined by exponentiating the following truncated Karhunen-Loève expansion of a Gaussian random field g(x):

$$l\left(x\right) = e^{\mu_g(x)+\sum_{i=1}^{M} g_i(x)\xi_i} = e^{\mu_g(x)+g(x)^T\xi} \tag{3.10.6}$$

The polynomial chaos expansion now reads:

$$l\left(x\right) = \sum_{i=0}^{\infty} l_i\left(x\right)\Psi_i\left(\xi\right) \tag{3.10.7}$$

Closed-form expressions of the coefficients $l_i\left(x\right)$ are given in the next section. To use SSFEM in conjunction with a lognormal input random field is now straightforward: the procedure described in Section 3.4 applies, where eq. (3.4.2) is replaced by eq. (3.10.3). The stochastic equilibrium equation of eq. (3.8.6) now writes:

$$\left(\sum_{i=0}^{\infty} K_i \Psi_i\left(\theta\right)\right) U\left(\theta\right) = F \tag{3.10.8}$$

After truncation to the first P terms, the Galerkin minimization of error leads to a system of linear equations similar to eq. (3.8.23). The coefficients c_{ijk} are now replaced by:

$$d_{ijk} = E\left[\Psi_i \quad \Psi_j \quad \Psi_k\right] \tag{3.10.9}$$

The polynomial chaos expansion of the input random field introduces a new approximation in SSFEM, which probably decreases the accuracy of the method. This accuracy has not been stated by Ghanem and his co-workers. Whether an adequate accuracy could be obtained with a manageable number of terms in the series expansion is of crucial importance. Unfortunately, no comparison with other methods (e.g. Monte Carlo simulation) are provided in [29, 30]. Regarding reliability problems, the accuracy in the tails of PDFs is probably also affected by the use of the polynomial chaos expansion of the input random field.

It is usual that more than one material property governs the evolution of a system (ie. Young's modulus and Poisson's ratio). In a probabilistic context, all these quantities have to be modeled as random fields if they are statistically independent.

This is completed in the following manner: each field is discretized using different sets of standard normal variables, say $\left[\xi_1 \ldots \xi_M\right]$ for the first one, $\left[\xi_{M+1} \ldots \xi_{M'}\right]$ for the second, etc. All these variables are then merged in a single array, the size of which determines the dimension of the polynomial chaos expansion of the response. This technique was applied in the heat conduction example presented in [30]. Except from the point of view of data management, using multiple input random fields seems not a difficult task. However, multiplying by 2 the length of vector ξ increases dramatically the size of the polynomial chaos basis (see for instance table 3.9.2), which basically controls the computation time.

3.11 KARHUNEN-LOÈVE EXPANSION OF LOG-NORMAL RANDOM FIELDS

Let us consider the following truncated Karhunen-Loève expansion of a Gaussian random field g(x):

$$\widehat{g}(x,\theta)=\mu_g(x)+\sum_{i=1}^{M} g_i(x)\xi_i(\theta) \tag{3.11.1}$$

Gathering the random variables $\xi_i(\theta)$ in a vector ξ and the deterministic functions $g_i(x)$ in a vector g(x), we can define the following approximate log-normal random field:

$$l(x)=e^{\widehat{g}(x)}=e^{\mu_g(x)+g(x)^T\xi} \tag{3.11.2}$$

Its coefficients in the polynomial chaos expansion are obtained as in eq. (3.10.4) by:

$$l_i(x)=\frac{E\left[e^{\mu_g(x)+g(x)^T\xi}\Psi_i\right]}{E\left[\Psi_i^2\right]} \tag{3.11.3}$$

The first coefficient corresponding to $\Psi_0 \equiv 1$ is the mean value of l(x), i.e.:

$$l_0(x)=\mu_l(x)=e^{\mu_g(x)+\frac{1}{2}\sum_{i=1}^{M} g_i(x)^2}=e^{\mu_g(x)+\frac{1}{2}\sigma_{\widehat{g}}^2(x)} \tag{3.11.4}$$

where $\sigma_{\widehat{g}}(x)$ is the standard deviation of $\widehat{g}(x)$. The other coefficients simplify after some algebra to:

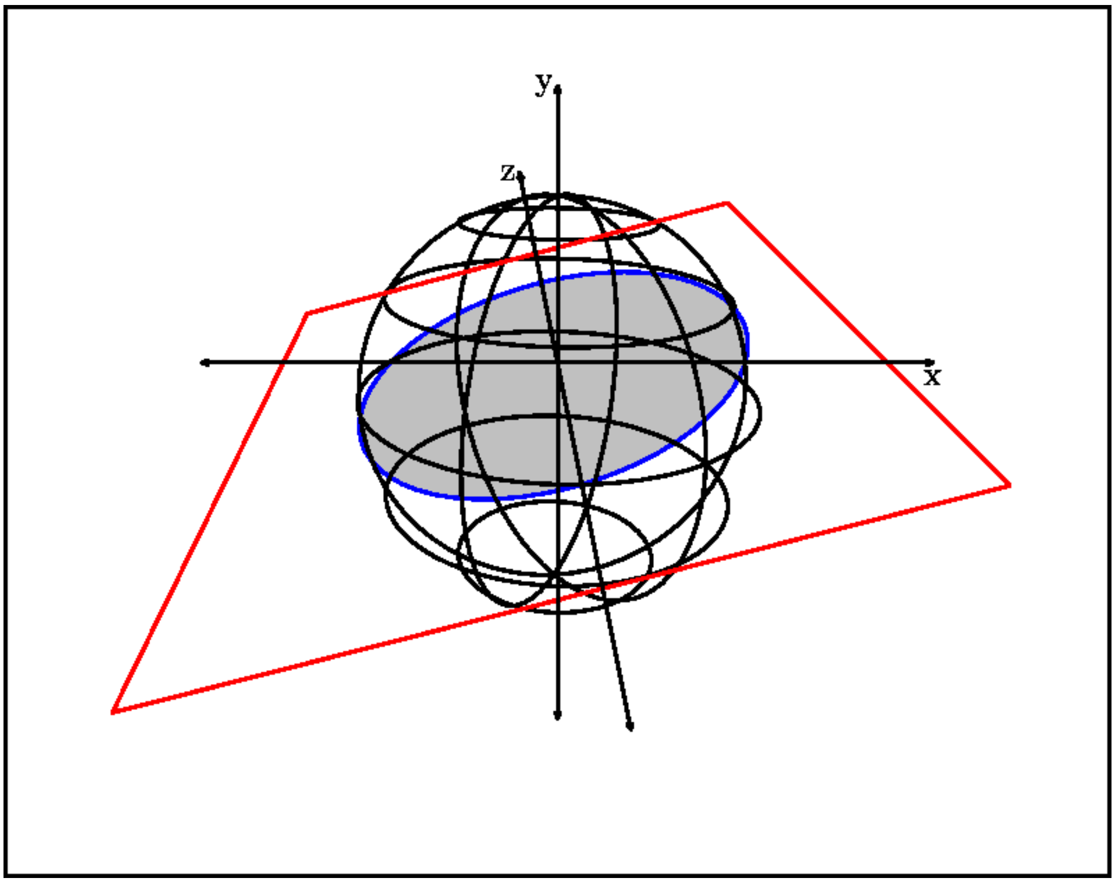
$$l_i(x)=\mu_l(x)\frac{E\left[\Psi_i\left(\xi+g(x)\right)\right]}{E\left[\Psi_i^2\right]} \tag{3.11.5}$$

Referring to representation (3.9.13) of the polynomials $\Psi_{\iota}(\xi)$, the fraction in the above equation can be written as:

$$\frac{E\left[\Psi_i\left(\xi+g(x)\right)\right]}{E\left[\Psi_i^2\right]}=\frac{\prod_{j=1}^{M} g_j(x)^{a_j}}{\prod_{j=1}^{M} a_j!} \tag{3.11.6}$$

Finally, letting M tend to ∞, the polynomial chaos expansion of the lognormal field can be written as:

$$l(x)=\mu_l(x)+\sum_{i=1}^{\infty} l_i(x)\Psi_i(\xi)\equiv\mu_l(x)\sum_{a}\frac{\prod_{j=1}^{M} g_j(x)^{a_j}}{\prod_{j=1}^{M} a_{j!}}\Psi_a(\xi) \tag{3.11.7}$$


y
z
x

4.1 SOLUTION METHODS

In order to solve the system of equations that occurs from the space and time discretization procedures that have been described in chapters 2 and 3, when applied on structural mechanics problems, a number of solution methods can be used. These methods are divided into direct and iterative, with respect to the procedure that is followed in order to compute the solution and to global and domain decomposition if the problem domain is being solved as a whole or is divided into subdomains.

In the following sections, both direct and iterative methods, suitable for structural static, dynamic and stochastic engineering problems, will be described with special emphasis on domain decomposition methods.

4.2 DIRECT SOLUTION WITH CHOLESKY FACTORIZATION

The most efficient direct solution methods are based on the classic Gauss elimination method. In principal, the different variants of the Gauss elimination method are derived by permutating the various steps of the actual elimination procedure. The Cholesky method [31], which is widely utilized in computational mechanics, is derived by separating all computations related to the right hand side of the linear system from the rest of the computations needed to solve the linear systems of the form:

$$Ax = b \tag{4.2.1}$$

The first step of the Cholesky method consists of all the necessary computations for performing the Gauss elimination process. This process is called factorization and aims to transform the symmetric coefficient matrix *A* to a product and is independent of the current right hand side. This factorization process can be written as follows:

$$A = LL^T \tag{4.2.2}$$

where *L* is a lower-triangular matrix. Moreover, a variant of the Cholesky method [32] is used where *A* is factorized as:

$$A = LDL^T \tag{4.2.3}$$

where *L* is once again a lower-triangular matrix with unit diagonals and *D* is a diagonal matrix. After the factorization process, the system can be solved by following the below procedure:

- Evaluation of the intermediate vector x1 with a forward substitution:

$$Lx_1 = b \Leftrightarrow x_1 = L^{-1}b \tag{4.2.4}$$

- Trivial evaluation of the intermediate vector x2:

$$Dx_2 = x_1 \Leftrightarrow x_2 = D^{-1}x_1 \tag{4.2.5}$$

- Evaluation of the unknown vector x with a backward substitution:

$$L^T x = x_2 \Leftrightarrow x = L^{-T} x_2 \tag{4.2.6}$$

The most computationally intensive part of this process is the factorization process as per eqs. (4.2.2) or (4.2.3). The rest of the computations described by eqs. (4.2.4) to (4.2.6) are much less computationally intensive and can be repeated for the solution of multiple right hand sides. As a result, the Cholesky method has the advantage of solving equations with multiple right hand sides with negligible additional computational cost.

Moreover, the Cholesky method can have different implementations, depending on the coefficient matrix storage format. Usually, matrix *A* is stored either in sparse format where only the non-zero terms of the matrix are stored or in skyline format where, for every column, all terms found between the first non-zero term and the diagonal are stored. For the first case, a sparse solver is utilized whereas for the second case, a skyline solver is the best choice. For each of these solvers, there is an optimum numbering in order to minimize the bandwidth and the subsequently required computations.

The computations required for both skyline and sparse solvers are quite coupled in nature since they involve a multitude of terms that do not exhibit any special locality pattern and can be either near to or far from each other. Attempting to parallelize these solvers demands high communication bandwidth and is quite difficult to implement. This communication overhead does not influence very much the performance of such parallel implementations in shared memory parallel architectures but in distributed memory environments, the performance penalty of this overhead is severe and has a big negative impact in scalability. In general, skyline and sparse solvers are popular in single node environments and are used extensively in commercial finite element software packages.

4.3 ITERATIVE SOLUTION WITH THE PRECONDITIONED CONJUGATE GRADIENT METHOD

Various iterative methods have been used for solving the discretized algebraic equations in computational mechanics and in structural computational mechanics. In particular, Jacobi, Gauss-Seidel, Successive Overrelaxation (SOR) and Steepest Descent are some early solution methods. However one of the most widely used iterative methods is the preconditioned conjugate gradient (PCG) method [33] which deals with solving linear systems of the form (4.2.1).

In order for the PCG method to converge, matrix A must be symmetric and positive semi-definite, meaning that its eigenvalues are greater or equal to zero. If matrix A is positive definite, meaning that its eigenvalues are greater than zero, then the PCG method is converging to the one and only solution of the linear system (4.2.1). However, in case matrix A has zero eigenvalues, the linear system (4.2.1) has either no solutions or is indefinite if the following stands:

$$b \in \text{range}(A) \Leftrightarrow \text{null}(A)^T b = 0 \tag{4.3.1}$$

In such a case, the linear system (4.2.1) has infinite solutions of the form:

$$x = \tilde{x} + \text{null}(A)a,\ a \in \mathbb{R}^d \tag{4.3.2}$$

where $\tilde{x}$ is one of the solutions of the linear system (4.2.1) and α is any vector with dimension equal to that of the null space of A.

The PCG algorithm is shown in Table 4.1, with $\tilde{A}^{-1}$ being a positive definite preconditioner. The method is considered to have converged when the following inequality stands:

$$\left\| b - Ax^k \right\| < \varepsilon \left\| b \right\| \tag{4.3.3}$$

where ε is a threshold user defined positive value, relevant to the required solution accuracy.

It is worth noting that the direction vectors p^k computed at each iteration are A-orthogonal which means that:

$$p^{k_1^T} A p^{k_2} = 0,\ k_1 \neq k_2 \tag{4.3.4}$$

In case of ill-conditioned linear systems and generally for cases that significant round-off errors are expected, the PCG algorithm can be applied with reorthogonalization which explicitly enforces the condition (4.3.4) at each iteration.

Initialization phase: $r^0 = b - Ax^0,\ z^0 = \tilde{A}^{-1}r^0,\ p^0 = z^0,\ q^0 = Ap^0,\ \eta^0 = \dfrac{p^{0^T} r^0}{p^{0^T} q^0}$

Repeat for k=1,2,… until convergence:

Solution estimate	$x^k = x^{k-1} + \eta^{k-1} p^{k-1}$
Residual vector	$r^k = r^{k-1} - \eta^{k-1} q^{k-1}$
Preconditioned residual vector	$z^k = \tilde{A}^{-1} r^k$

Search vector	Simple estimation	$p^k = z^k + \frac{z^{k^T} r^k}{z^{k-1^T} r^{k-1}} p^{k-1}$ (4.3.5)
	Using reorthogonalization	$p^k = z^k - \sum_{i=0}^{k-1} \frac{z^{k^T} q^i}{p^{i^T} q^i} p^i$
A matrix vector product		$q^k = Ap^k$ (4.3.6)
η estimation	Simple estimation	$\eta^k = \frac{z^{k^T} r^k}{p^{k^T} q^k}$
	Using reorthogonalization	$\eta^k = \frac{p^{k^T} r^k}{p^{k^T} q^k}$

Table 4.1 - The PCG algorithm

The number of iterations of the PCG method is directly related to the structure and span of the eigenvalue spectrum of matrix $\tilde{A}^{-1}A$. Another measure for estimating convergence speed of the PCG method is the condition number [34] which is defined as the ratio of the biggest in magnitude to the smallest eigenvalue of matrix $\tilde{A}^{-1}A$:

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}} \qquad (4.3.7)$$

The larger the condition number, the greater the number of iterations that are needed for the PCG method to converge.

Linear systems that feature a condition number of large magnitude are defined as ill-conditioned systems. In structural computational mechanics, structures combining very flexible and very stiff elements, shells of low width or structures comprised of highly heterogeneous materials produce linear systems that are generally ill-conditioned.

Moreover, the efficiency of the preconditioner itself is highly related to the condition number of matrix $\tilde{A}^{-1}A$. Ideally, the most suitable preconditioner is the inverse of matrix A so that $\tilde{A}^{-1}A = A^{-1}A = I$ and $\lambda_{\min} = \lambda_{\max} = 1$. In such a case, the condition number is equal to 1 and the PCG method converges in just one iteration. However, computing the inverse of matrix A is not computationally efficient, especially of large matrices, so, in practice, efficient preconditioners are approximations of the inverse of matrix A that are relatively cheap to compute.

4.3.1 PRECONDITIONED CONJUGATE PROJECTED GRADIENT

A variant of the PCG method is the preconditioned conjugate projected gradient (PCPG) method [35] which is suitable for solving linear systems of the form:

$$\begin{Bmatrix} P^T F\lambda = P^T d \\ G^T a = e \end{Bmatrix} \tag{4.3.8}$$

where the vector search space is projected to a different subspace using the following projector:

$$P = I - G\left(G^T G\right)^{-1} G^T \tag{4.3.9}$$

with matrix G imposing a set of constraint equations.

The PCPG alogorithm is described as follows:

Initialization phase: $\lambda^0 = G\left(G^T G\right)^{-1} e,\; w^0 = P^T\left(d - F\lambda_0\right)$ (4.3.10)

Repeat for k=1,2,… until convergence:

Preconditioned projected residual vector	$y^k = P\tilde{F}^{-1} w^k$ (4.3.11)
Search vector (with re-orthogonalization)	$p^k = y^k - \sum_{i=0}^{k-1} \frac{y^{k^T} F p^i}{p^{i^T} F p^i} p^i$ (4.3.12)
η estimation	$\eta^k = \frac{p^{k^T} w^k}{p^{k^T} F p^k}$
Solution estimate	$\lambda^{k+1} = \lambda^k + \eta^k p^k$
Residual vector	$w^{k+1} = w^k - \eta^k P^T F p^k$ (4.3.13)

Table 4.2 - The PCPG algorithm

4.3.2 IMPLEMENTATION FOR MULTIPLE RIGHT-HAND SIDES

By utilizing the PCG algorithm in order to solve the linear system (4.2.1), it is possible to significantly reduce the computational burden for solving a problem with multiple right-hand sides [36].

Let the following sequence of n_a linear problems:

$$Ax_i = b_i,\ i = 1,\ldots,n_a \tag{4.3.14}$$

and p_l with i=1..np to be the set of search vectors produced during solutions i=1..j with j < n_a. For the solution of analysis j+1, the PCG algorithm of Table 4.1 is followed using reorthogonalization and using the following first solution estimate:

$$x_{j+1}^0 = P_{np} x_p \tag{4.3.15}$$

with

$$P_{np} = \begin{bmatrix} p_1 & \cdots & p_{n_p} \end{bmatrix},\ x_p = \left(Q_{n_p}^T P_{n_p}\right)^{-1} P_{n_p}^T b_{j+1},\ Q_{n_p} = AP_{n_p} = \begin{bmatrix} Ap_1 & \cdots & Ap_{n_p} \end{bmatrix} = \begin{bmatrix} q_1 & \cdots & q_{n_p} \end{bmatrix}$$

Estimating x_p is trivial since $\left(Q_{n_p}^T P_{n_p}\right)$ has values only in its diagonal due to the fact that the search vectors are computed using reorthogonalization, ensuring A-orthogonality. Moreover, the search vector estimation step can be carried out using all or a fraction of the vectors stored from all the accumulated solutions.

4.3.3 PCG – THE TWO-LEVEL TECHNIQUE

A special technique used when solving linear systems with the PCG method is the two-level technique [37]. This technique is based on the definition of an arbitrary matrix C whose columns are a linear combination of the linear system (4.2.1). By setting:

$$C^T\left(b - Ax\right) = 0 \tag{4.3.16}$$

an approximation of the initial linear system is formed related to linear combinations defined from the columns of matrix C. In order to satisfy the above equation, the solution x is split as follows:

$$x = \hat{x} + Cx_c \tag{4.3.17}$$

Thus, the linear system (4.2.1) can be reformed using one set of redundant equations as follows:

$$\begin{Bmatrix} A\hat{x} + ACx_c = b \\ C^T A\hat{x} + C^T ACx_c = C^T b \end{Bmatrix} \tag{4.3.18}$$

By substituting the solution of the second equation of the redundant linear system to the first one, Eq. (4.3.18) becomes:

$$P_c^T A\hat{x} = P_c^T b \tag{4.3.19}$$

where

$$P_c = I - C\left(C^T AC\right)^{-1} C^T A \tag{4.3.20}$$

In case that $C^T AC$ is not positive-definite but singular with zero eigenvalues, Eq. (4.3.20) becomes:

$$P_c = I - C\left(C^T AC\right)^{+} C^T A \tag{4.3.21}$$

where $\left(C^T AC\right)^{+}$ is the generalized inverse of $C^T AC$.

The computation of matrix P_c of Eq. (4.3.20), or Eq. (4.3.21) if $C^T AC$ is singular, implicitly involves the need for solving a problem of the form

$$\left(C^T AC\right)x = b \tag{4.3.22}$$

at each PCG iteration when solving Eq. (4.3.19). In fact, the problem (4.3.22) is a second-level reduced form of problem (4.2.1), which explains why this solution technique is characterized as a "two-level" technique.

The main advantage of this technique is the exhibition of a global coupling effect of all substructure computations due to the $C^T AC$ term. Therefore, it provides a mechanism for propagating the error globally and ensuring the numerical scalability with respect to the number of substructures. If the number of columns of C is kept sufficiently small, the problem (4.3.22) becomes a coarse problem.

4.4 DOMAIN DECOMPOSITION

In the following sections, basic aspects of domain decomposition methods (DDM) will be presented in order to provide a solid foundation for the presentation of the primal and dual DDM (P-DDM and D-DDM respectively) used for the solution of static, dynamic and stochastic problems. Moreover, a specialized DDM family of methods will be presented, custom tailored for the solution of porous media problems as formulated in chapter 2.

4.4.1 SUBDOMAINS AND MAPPING OPERATORS

Subdomain mapping operators for DDM [38] can be implemented for mapping either the displacements and applied loads or the Lagrange multipliers of the subdomains.

If u and f represent the displacement and applied loads vectors of the global domain and u^s and f^s are vectors which refer to the corresponding quantities for every subdomain, the following equations hold:

$$u^s = \begin{bmatrix} u^{(1)^T} & \dots & u^{(N_s)^T} \end{bmatrix}^T \tag{4.4.1}$$

$$f^s = \begin{bmatrix} f^{(1)^T} & \dots & f^{(N_s)^T} \end{bmatrix}^T \tag{4.4.2}$$

$$u^s = Lu \tag{4.4.3}$$

$$f = L^T f^s \tag{4.4.4}$$

where N_s is the number of non-overlapping subdomains of the global domain and L is the so-called global to local mapping operator which is a Boolean matrix.

Eqs. (4.4.1)-(4.4.4), when applied to interface degrees of freedom (dof) become:

$$u_b^s = \begin{bmatrix} u_b^{(1)^T} & \dots & u_b^{(Ns)^T} \end{bmatrix}^T \tag{4.4.5}$$

$$f_b^s = \begin{bmatrix} f_b^{(1)^T} & \dots & f_b^{(Ns)^T} \end{bmatrix}^T \tag{4.4.6}$$

$$u_b^s = L_b u_b \tag{4.4.7}$$

$$f_b = L_b^{\,T} f_b^s \tag{4.4.8}$$

The traction forces on the interface nodes of the disconnected subdomains are usually expressed as:

$$t = f^s - B^T \lambda \tag{4.4.9}$$

or

$$t_b = f_b^s - B_b^{\,T} \lambda \tag{4.4.10}$$

when applied to the interface dof, where λ is the vector of the Lagrange multipliers and B is the so-called Lagrange mapping operator. The form of the mapping operator depends on the definition of the Lagrange multipliers. In the case of redundant Lagrange multipliers, which are used in the present investigation, B is a signed Boolean matrix.

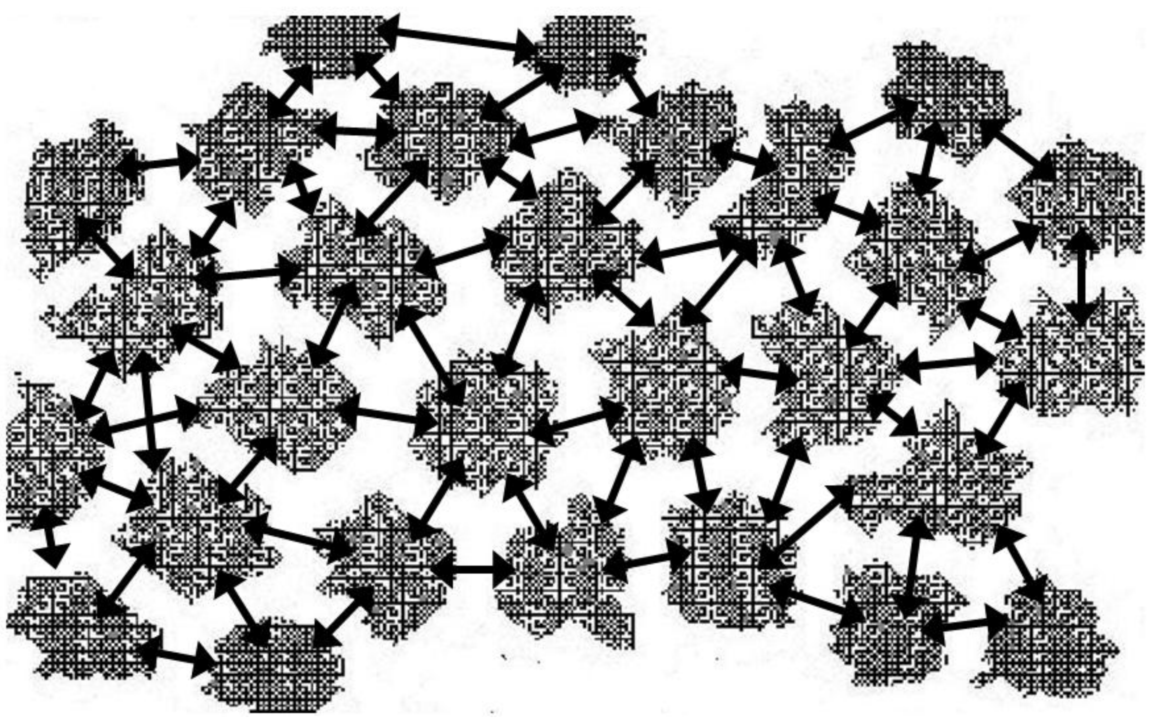


Figure 4.1 - A structural domain, split in subdomains. Arrows show the traction forces between the disconnected subdomains

The Lagrange mapping matrices may also be used to express the displacement compatibility condition at subdomain interfaces as:

$$Bu_s = \begin{bmatrix} B^{(1)} & \cdots & B^{(Ns)} \end{bmatrix} \begin{bmatrix} u^{(1)} \\ \vdots \\ u^{(Ns)} \end{bmatrix} = 0 \tag{4.4.11}$$

or

$$B_b u_b^s = \begin{bmatrix} B_b^{(1)} & \cdots & B_b^{(Ns)} \end{bmatrix} \begin{bmatrix} u_b^{(1)} \\ \vdots \\ u_b^{(Ns)} \end{bmatrix} = 0 \tag{4.4.12}$$

when applied to the interface dof.

Specific attention has to be paid when using these mapping operators in preconditioning steps of dual DDM. In the case of global to local mapping operator with respect to homogeneous problems, the global to local mapping operator in the preconditioning step can be written as:

$$L_p = L\left(M^s\right)^{-1} \tag{4.4.13}$$

or

$$L_{p_b} = L_b\left(M_b^s\right)^{-1} \tag{4.4.14}$$

Moreover, for cases of splitting displacements or forces to heterogeneous subdomains inside a preconditioning step, we get:

$$u_{b_{estimate}} = L_{p_b}^T u_b^s \tag{4.4.15}$$

$$f_{b_{estimate}}^s = L_{p_b} f_b \tag{4.4.16}$$

In the past, a number of modified versions of the Lagrange mapping operator that incorporate scaling effects have been used in preconditioning steps of the dual DDM. In the case of redundant Lagrange multipliers and homogeneous problems, the Lagrange mapping operator in the preconditioning step can be written as:

$$B_p = B\left(M^s\right)^{-1} \tag{4.4.17}$$

or

$$B_{p_b} = B_b\left(M_b^s\right)^{-1} \tag{4.4.18}$$

when applied to the interface dof, where M^s and M_b^s are diagonal matrices with diagonal entries the multiplicity of the corresponding dof, which correspond to the number of subdomains that this dof belongs to.

4.4.2 LOCAL PROBLEM SOLUTION

Domain decomposition methods require the repeated solution of many local subdomain problems corresponding to the dof of the subdomains. These local subdomain problems are typically solved using a direct method since their size is very small compared to the size of the global problem [38].

In the case of D-DDM, local subdomain problems are of the form:

$$K^{(s)}u^{(s)} = f^{(s)} - B^{(s)^T}\lambda \tag{4.4.19}$$

where $\mathrm{K}^{(s)}$ is the stiffness matrix of each subdomain.

In the case of P-DDM, similar local subdomain problems require repeated solution. These problems are of the form:

$$S^{(s)} u_b^{(s)} = \hat{f}_b^{(s)} - B_b^{(s)^T} \lambda \tag{4.4.20}$$

where $S^{(s)}$ is the Schur complement matrix (Eq. (4.4.22)) and $\hat{f}_b^{(s)}$ is the condensed force vector (Eq. (4.4.23)). The main difference of these local problems is the fact that they either refer to all of the subdomain dof or to the interface ones. In the latter case, the equations which are related to internal dof of the subdomains are eliminated first. In order to obtain the corresponding relations, the local subdomain problem of Eq. (4.4.19) is re-arranged so that it can be written in the form:

$$\begin{bmatrix} K_{bb}^{(s)} & K_{bi}^{(s)} \\ K_{ib}^{(s)} & K_{ii}^{(s)} \end{bmatrix} \begin{bmatrix} u_b^{(s)} \\ u_i^{(s)} \end{bmatrix} = \begin{bmatrix} f_b^{(s)} \\ f_i^{(s)} \end{bmatrix} - \begin{bmatrix} B_b{}^T \\ 0 \end{bmatrix} \lambda \tag{4.4.21}$$

with subscripts b and i denoting the restriction of the matrices to interface (boundary) and internal d.o.f., respectively. With this re-arrangement, the following matrices and vectors are defined:

$$S^{(s)} = K_{bb}^{(s)} - K_{bi}^{(s)} \left(K_{ii}^{(s)}\right)^{-1} K_{ib}^{(s)} \tag{4.4.22}$$

$$\hat{f}_b^{(s)} = f_b^{(s)} - K_{bi}^{(s)} \left(K_{ii}^{(s)}\right)^{-1} f_i^{(s)} \tag{4.4.23}$$

In the case of implicit dynamics, matrices $K^{(s)}$, $K_{bb}^{(s)}$, $K_{ii}^{(s)}$ and $S^{(s)}$ are coefficient matrices of the integrated dynamic equilibrium equations and as such, they are always positive definite. This means that the corresponding matrices of the subdomains have no null space and the structure they refer to (adequately constrained or not) have no rigid body modes.

4.4.3 INTERFACE PROBLEM SOLUTION

Domain decomposition methods require the solution of an interface problem in the form of the linear eq. (4.2.1) where A, x and b are the left-hand side matrix, the solution vector and the right-hand side vector, respectively. Usually, the left-hand side matrix is symmetric and positive definite or semi-definite. Furthermore, the above equation is typically solved iteratively with the standard PCG method. The use of the PCG method also requires the definition of a positive definite preconditioning matrix $\tilde{A}^{-1}$, as an approximation of the inverse or generalized inverse of A.

In the particular case of a semi-definite left-hand side matrix A (i.e. for an unconstrained subdomain of a structural mechanics problem), a solution of the interface problem exists under the condition:

$$b \in \operatorname{range}(A) \Leftrightarrow \operatorname{null}(A)^T b = 0 \tag{4.4.24}$$

When the above condition holds, the interface problem has infinite solutions of the form:

$$x = \hat{x} + \operatorname{null}(A) a, \; a \in \mathbb{R}^d \tag{4.4.25}$$

where $\tilde{x}$ is a particular solution of the local interface and a is any vector with dimension equal to the dimension d of the null-space of A. In the case of a semi-definite matrix A, the PCG succeeds by computing one of the above infinite solutions of Eq. (4.4.25).

4.4.4 RIGID BODY MODES

By splitting the displacements $u^{(s)}$ of a subdomain into $u_r^{(s)}$ and $u_d^{(s)}$, the following stands:

$$u^{(s)} = u_r^{(s)} + u_d^{(s)} \tag{4.4.26}$$

Displacements $u_r^{(s)}$ are caused by the rigid body modes of the subdomain while displacements $u_d^{(s)}$ are due to the stiffness $K^{(s)}$ and are related to it with an equation similar to that of eq. (4.4.19). However the stiffness matrix for these displacements does not coincide with the one of eq. (4.4.19) since the latter also takes displacements $u_r^{(s)}$ into account. In order to exclude these displacements, a set of artificial constraints is imposed so that displacements $u_r^{(s)}$ are equal to 0. In practice, this can be accomplished by magnifying the corresponding diagonals of the stiffness matrix by orders of magnitude, constituting these dof practically rigid.

The stiffness matrix that occurs from this procedure now represents a statically determined subdomain thus being positive definite and invertible, with its inverse being equal to the generalized inverse $K^{(s)^+}$, and displacements $u_d^{(s)}$ being equal to:

$$u_d^{(s)} = K^{(s)^+} \left(f^{(s)} - B^{(s)^T} \lambda_b \right) \tag{4.4.27}$$

while displacements $u_r^{(s)}$ are equal to:

$$u_r^{(s)} = R^{(s)} a^{(s)} \tag{4.4.28}$$

where $R^{(s)}$ is a matrix consisting of the rigid body modes of the subdomain and is equal to the null space of the stiffness matrix $K^{(s)}$ and $a^{(s)}$ is a vector representing the contribution of each rigid body mode at the displacement vector.

This implies that the loads $f^{(s)} - B^{(s)^T}\lambda_b$ are self-equilibrated which means that:

$$R^{(s)^T}\left(f^{(s)} - B^{(s)^T}\lambda_b\right) = 0 \tag{4.4.29}$$

By combining eqs. (4.4.26), (4.4.27) and (4.4.28) we get:

$$u^{(s)} = K^{(s)^+}\left(f^{(s)} - B^{(s)^T}\lambda_b\right) + R^{(s)}a^{(s)} \tag{4.4.30}$$

Eqs. (4.4.29) and (4.4.30) in block form are written as:

$$R^{s^T}\left(f^{s} - B^{s^T}\lambda_b\right) = 0 \tag{4.4.31}$$

$$u^{s} = K^{s^+}\left(f^{s} - B^{T}\lambda\right) + R^{s}a \tag{4.4.32}$$

where

$$K^{s^+} = \begin{bmatrix} K^{(1)^+} & & \\ & \ddots & \\ & & K^{(Ns)^+} \end{bmatrix},\ R^{s} = \begin{bmatrix} R^{(1)} & & \\ & \ddots & \\ & & R^{(Ns)} \end{bmatrix},\ a = \begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(Ns)} \end{bmatrix} \tag{4.4.33}$$

For the case of eq. (4.4.20), a corresponding block form is derived and using the rationale for the formulation of eqs. (4.4.31) and (4.4.32) we get:

$$R_b^{s^T}\left(\hat{f}_b^{s} - B_b^{T}\lambda\right) = 0 \tag{4.4.34}$$

$$u_b^{s} = S^{s^+}\left(\hat{f}_b^{s} - B_b^{T}\lambda\right) + R_b^{s}a \tag{4.4.35}$$

with

$$S^{s^+} = \begin{bmatrix} S^{(1)^+} & & \\ & \ddots & \\ & & S^{(Ns)^+} \end{bmatrix},\ \hat{f}_b^{s} = \begin{bmatrix} \hat{f}_b^{(1)} \\ \vdots \\ \hat{f}_b^{(Ns)} \end{bmatrix} \tag{4.4.36}$$

and R_b^s denoting the restriction of R^s to the interface dof.

4.5 SOLUTION METHODS FOR STATIC PROBLEMS

In the following sections, a series of domain decomposition methods for solving static problems will be presented.

4.5.1 P-DDM: THE PRIMAL SUBSTRUCTURING METHOD (PSM)

The basic DDM is the primal substructuring method, abbreviated in the following as PSM. In the context of this DDM, the internal dof of the subdomains are eliminated first. The PSM interface displacement problem is thus obtained by combining Eqs. (4.4.7), (4.4.8) and (4.4.20) in order to form the equation

$$\hat{S}u_b = \hat{f}_b \tag{4.5.1}$$

where

$$\hat{S} = L_b^T S^s L_b,\ \hat{f}_b = L_b^T \hat{f}_b^s \tag{4.5.2}$$

$$\hat{f}_b^s = \begin{bmatrix} \hat{f}_b^{(1)^T} & \cdots & \hat{f}_b^{(Ns)^T} \end{bmatrix}^T \tag{4.5.3}$$

$$S^s = \begin{bmatrix} S^{(1)} & & \\ & \ddots & \\ & & S^{(Ns)} \end{bmatrix} \tag{4.5.4}$$

The solution of the linear system of Eq. (4.5.1) is usually performed with the PCG method as stated in section 5.3, since the left-hand side matrix is symmetric and positive definite or semi-definite.

In the past, several strategies have been proven efficient for preconditioning the underlying iterative solver for these types of problems with a common choice being the preconditioner:

$$\tilde{A}^{-1} = L_{p_b}^T S^{s^+} L_{p_b} \tag{4.5.5}$$

which is used in the so-called Neumann–Neumann PSM [39]. More precisely, the preconditioner (4.5.5) is implemented as follows:

$$\tilde{A}^{-1} = L_{p_b}^T N_b^s K^{s^+} N_b^{s^T} L_{p_b} \tag{4.5.6}$$

where N_b^s is a Boolean matrix which extracts the interface dof from subdomain dof vectors, as:

$$u_b^s = N_b^s u^s,\ f_b^s = N_b^s f^s \tag{4.5.7}$$

4.5.2 D-DDM: THE FINITE ELEMENT TEARING AND INTECONNECTIING (FETI) METHOD

The FETI method [35], is a dual DDM that has been implemented for a number of problems in computational mechanics. Since its introduction, it has attracted a lot of attention and is considered as a fast domain decomposition algorithm suitable for both serial and parallel computing environments.

While its predecessor, the PSM, performs iterations in order to compute the interface displacement vector u_b of the structure, the FETI method iterates on the Lagrange multiplier vector λ. The Lagrange multipliers, which represent the interaction forces between the subdomains, are dual with respect to the interface displacements and this explains the name dual substructuring method, in comparison to PSM.

In the context of the FETI method, the nodal force vector f of the structure is first split to the subdomains:

$$f^s = L_p f \tag{4.5.8}$$

Combining eqs. (4.4.11), (4.4.31) and (4.4.32), the following system of equations is obtained:

$$\begin{bmatrix} F_I & -G \\ -G^T & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ a \end{bmatrix} = \begin{bmatrix} d \\ -e \end{bmatrix} \tag{4.5.9}$$

where

$$F_I = BK^{s^+}B^T,\ G = BR^s,\ d = BK^{s^+} f^s,\ e = R^{s^T} f^s \tag{4.5.10}$$

In order to decouple the linear system (4.5.9), the following projector is introduced:

$$P = I - QG\left(G^T QG\right)^{-1} G^T \tag{4.5.11}$$

For homogeneous problems, operator P defined in eq. (4.5.11) is usually implemented with Q=I. However, for heterogeneous problems matrix Q might be set otherwise as described at the end of this section.

Computations involving projector P require the solution of linear problems of the form:

$$\left(G^T QG\right)x = b \tag{4.5.12}$$

which constitutes the so-called ''coarse-grid'' problem of the FETI method. This name is explained by the fact that $G^T QG$ is a sparse matrix, with the typical sparsity pattern of a finite element stiffness matrix, if one considers each subdomain as a finite element node having the same number of dof as the number of its zero energy modes. This coarse problem ensures the exchange of information between remote subdomains of the structure at each iteration of the underlying iterative solver used for the interface problem, thus guaranteeing fast convergence.

Premultiplying the first of the two matrix equations in eq. (4.5.9) with $\left(G^T Q G\right)^{-1} G^T Q$, it follows that for a given Lagrange multiplier vector λ, the vector of the zero energy mode amplitudes a is equal to:

$$a = -\left(G^T Q G\right)^{-1} G^T Q\left(d - F_I \lambda\right) \tag{4.5.13}$$

Furthermore, using eqs. (4.4.32) and (4.5.13), it follows that the jump $\Delta u_b = B u^s$ of the displacement field at subdomain interfaces is equal to:

$$\Delta u_b = B u_s = d - F_I \lambda + G a = P^T\left(d - F_I \lambda\right) \tag{4.5.14}$$

Based on eq. (4.5.14), the linear system (4.5.9) is equivalent to the following system where the unknown vectors λ and a are decoupled:

$$P^T F_I \lambda = P^T d \tag{4.5.15}$$

$$G^T a = e \tag{4.5.16}$$

In order to solve eqs. (4.5.15) and (4.5.16) for the Lagrange multiplier vector λ, the latter is being split as follows:

$$\lambda = \lambda_0 + P\bar{\lambda} \tag{4.5.17}$$

where vector λ_0 should satisfy eq. (4.5.16) and is chosen equal to:

$$\lambda_0 = Q G\left(G^T Q G\right)^{-1} e \tag{4.5.18}$$

Based on eqs. (4.5.17) and (4.5.18), the system of eqs. (4.5.15) and (4.5.16) can be written as the following interface problem:

$$\left(P^T F_I P\right)\bar{\lambda} = P^T\left(d - F_I \lambda_0\right) \tag{4.5.19}$$

In order to calculate the total displacement field u, the following steps are followed:

- The Lagrange multiplier vector $\bar{\lambda}$ is computed by solving the interface problem (4.5.19).
- The Lagrange multiplier vector λ is evaluated from eq. (4.5.17).
- The amplitudes a of the subdomain rigid body modes are computed from eq. (4.5.13).

- Subdomain displacement fields u^s are computed from eq. (4.4.32).
- The total displacement field u of the structure is finally given by $u = L_p^T u^s$

The two most widely used preconditioners for the FETI method are:

$$\tilde{F}_I^{D^{-1}} = B_{p_b} S^s B_{p_b}^T \tag{4.5.20}$$

$$\tilde{F}_I^{L^{-1}} = B_{p_b} K_{bb}^s B_{p_b}^T \tag{4.5.21}$$

the Dirichlet and the lumped preconditioners.

The Dirichlet preconditioner is typically used in fourth-order problems. Moreover, in second-order problems, the lumped preconditioner is usually more efficient in terms of the total solution time. In some second-order problems however, namely in highly heterogeneous structures and in problems where subdomains of bad aspect ratio are generated, the Dirichlet preconditioner may outperform the lumped one. Variant forms of the Dirichlet preconditioner using approximate expressions for K_{ii}^s of the Schur complement S^s may also be used.

Accordingly, these proconditioners can be used as values for matrix Q of projector P in case of heterogeneous problems.

4.5.3 P-DDM FOR STATIC ANALYSIS WITH D-DDM BASED PRECONDITIONERS: THE PFETI METHOD

In this section, we introduce a new category of preconditioners for the PSM originally proposed by Fragakis and Papadrakakis [38]. An iterative solver is applied for the solution of the interface problem (4.5.1) in order to compute the interface displacement vector u_b, given an interface force vector $\hat{f}_b$. A good preconditioner for the PSM must treat the kth residual $r^k = \hat{f}_b - \hat{S}u_b^k$ as applied forces on the interface nodes of the structure and return in $z^k = \tilde{A}^{-1} r^k$ a good estimate of the interface displacements of the structure for the applied forces r^k. For instance, if the PSM preconditioning step is performed with any solver, like for example the FETI method, the iterative solver will immediately converge in the first iteration. The PFETI method consists of using as preconditioner of the PSM, a crude approximation of the FETI solution and as such, the first estimate for the interface displacements of the structure obtained from the first iteration of the FETI method is chosen.

For example, consider the FETI algorithm with an applied forces vector f equal to:

$$f = N_b^T r^k \tag{4.5.22}$$

Since all forces in the load vector of eq. (4.5.22) are applied on the interface nodes of the structure, we have $f_b = r^k$ and $f_i^s = 0$. Furthermore, the interface forces f_b may be split to the subdomains with the equation:

$$f_b^s = L_{p_b} f_b = L_{p_b} r^k \tag{4.5.23}$$

From eqs. (4.4.23) and (4.5.22), it follows that $\hat{f}_b^s = f_b^s = L_{p_b} r^k$. The components of e and d of eq. (4.5.10) thus become $e = R^{s^T} f^s = R_b^{s^T} r^k$ and $d = BK^{s^+} f^s = B_b S^{s^+} L_{p_b} r^k$. Moreover, with respect to interface values, matrix F_I may be written as $F_I = BK^{s^+} B^T = B_b S^{s^+} B_b{}^T$. Futhermore, the initial Lagrange multiplier vector λ_0 is equal to:

$$\lambda_0 = QG\left(G^{\mathrm{T}}QG\right)^{-1} R_b^{s^T} r^k \tag{4.5.24}$$

Combining eqs. (4.5.13) and (4.5.24), initial zero energy mode amplitude a_0 is equal to:

$$a_0 = -\left(G^TQG\right)^{-1} G^TQ\left(d - F_I\lambda_0\right) = -\left(G^TQG\right)^{-1} G^TQB_bS^{s^+}\left(L_{p_b}r^k - B_b^T\lambda_0\right)$$
(4.5.25)

Combining the above equation with eqs. (4.4.15) and (4.4.35), the interface displacements u_{b_0} estimated from the initialization of the FETI method are:

$$\begin{aligned}
u_{b_0} &= L_{p_b}^T u_b^s \\
&= L_{pb}^T\left(S^{s^+}\left(\hat{f}_b^s - B_b^T\lambda_0\right) + R_b^s a_0\right) \\
&= L_{pb}^T\left(S^{s^+}\left(r^k - B_b^T\lambda_0\right) - R_b^s\left(G^TQG\right)^{-1} G^TQB_bS^{s^+}\left(L_{p_b}r^k - B_b^T\lambda_0\right)\right) \\
&= L_{pb}^T\left(I - R_b^s\left(G^TQG\right)^{-1} G^TQB_b\right)S^{s^+}\left(L_{p_b}r^k - B_b^T\lambda_0\right) \\
&= L_{pb}^T\left(I - R_b^s\left(G^TQG\right)^{-1} G^TQB_b\right)S^{s^+}\left(I - B_b^TQG\left(G^TQG\right)^{-1} R_b^{s^T}\right)L_{p_b}r^k
\end{aligned}$$
(4.5.26)

From the above equation, the PSM preconditioner is deduced:

$$\tilde{A}^{-1} = L_{p_b}^T H_b^T S^{s^+} H_b L_{p_b} \tag{4.5.27}$$

where

$$H_b = I - B_b^TQG\left(G^TQG\right)^{-1} R_b^{sT} \tag{4.5.28}$$

4.6 SOLUTION METHODS FOR DYNAMIC AND POROUS MEDIA PROBLEMS

In the following sections, a series of domain decomposition methods for solving dynamic problems will be presented. These problems are considered to be temporally discretized with an implicit scheme as the one shown in section 2.10

4.6.1 D-DDM WITH NO COARSE PROBLEM FOR IMPLICIT DYNAMICS

The main difference of the D-DDM for implicit dynamics compared to the initial FETI variant of dual substructuring method used for static problems, is the absence of the coarse problem related to the rigid body modes of the subdomains, since all subdomains develop internal strains and stresses due to inertial forces. The interface problem to be solved is:

$$F_I \lambda = d \tag{4.6.1}$$

with the preconditioners utilized for the underlying iterative interface problem solver being the same as the ones used at the dual substructuring method [40].

The dual substructuring method is designed to solve problems which may contain floating subdomains, that is subdomains with zero energy modes which exhibit singular coefficient matrices. In that case, the local problems described by eq. (4.4.19) are ill-posed and in order to guarantee their solvability, it is required that eq. (4.4.31) stands. This solvability condition forms a natural coarse problem which, as stated above, is absent in this D-DDM dynamic version because it is applicable to linear systems that have non-singular coefficient matrices and thus it does not require the solvability condition. However, this lack of a coarse problem constitutes this D-DDM non-scalable as the error propagates slowly when the number of subdomains is high. In the following sections, a family of D-DDM is discussed which use the two-level technique in order to impose a coarse problem which ensures scalability.

4.6.2 D-DDM FAMILY WITH AN ARTIFICIAL COARSE PROBLEM FOR IMPLICIT DYNAMICS

This family of D-DDM is based on a FETI variant which imposes an artificial coarse problem. The implicit dynamic versions of D-DDM are constructed by applying the two-level technique on the standard D-DDM with no coarse problem [41]. By combining eqs. (4.6.1), (4.3.19) and (4.3.20), the following interface problem is needed to be solved:

$$PF_I \lambda = Pd \tag{4.6.2}$$

where

$$P = I - C\left(C^T F_I C\right)^{-1} C^T F_I \tag{4.6.3}$$

As in the D-DDM with no coarse problem, the preconditioners used for the implementation of this version with the PCG method are the same with the standard D-DDM and are given by eqs. (4.5.20) and (4.5.21).

The introduction of matrix C is equivalent to imposing a set of optional admissible constraints. These constraints adhere to the following equation:

$$C^T \sum_{s=1}^{s=N_s} B^{(s)} u^{(s)^f} = 0 \tag{4.6.4}$$

where $u^{(s)^f}$ are the exact solutions of the local problems (4.4.19) and are called optional because they are not required for the solution of these local problems. In order to impose these constraints, a starting vector is chosen equal to:

$$\lambda^0 = C\left(C^T F_I C\right)^{-1} C^T d \tag{4.6.5}$$

Due to the nature of these constraints, any properly chosen matrix C with linearly independent columns will exhibit superior convergence properties compared to the D-DDM with no coarse problem.

The application of a PCG algorithm for the solution of the interface problem resulted from the D-DDM with no coarse problem is quite straightforward. In the case of D-DDM family with an artificial coarse problem, based on optional admissible constraints, special considerations have to be implemented in order to evaluate the projector P. For reasons of completeness, a typical projection step occurring in every iteration, which includes the evaluation of an inner product of the projector with a vector, is described as follows:

$$y = Pz = \left(I - C\left(C^T F_I C\right)^{-1} C^T F_I\right) z = z - \left(C\left(C^T W\right)^{-1} W^T\right) z = z - Xz \tag{4.6.6}$$

where:

$$W = F_I C = F_I \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix} \tag{4.6.7}$$

and $c_1 \ldots c_n$ are the columns of matrix C. Evaluation of matrix W is equivalent to solving a linear system with *n* right-hand sides.

In order to evaluate the Xz inner product of eq. (4.6.6), the following steps have to be performed:

Evaluate $a = W^T z$ (4.6.8)

Solve $\left(C^T W\right) x = a$ (4.6.9)

Evaluate $b = Cx$

After a closer look at the algorithm, it is evident that the W matrix is needed for every projection step of Eq. (4.6.2) and is involved in two operations in Eq. (4.6.6). As a result, it is prudent to store this matrix after its calculation and use it in subsequent stages of the projection step since it is required to be computed at each projection step twice.

It is evident that storage and computational costs related to the projector P are directly related to both the size and the efficient computation of matrix C.

4.6.3 P-DDM FAMILY WITH AN ARTIFICIAL COARSE PROBLEM FOR IMPLICIT DYNAMICS

This P-DDM family belongs to the PFETI family of methods and consists of the PSM, preconditioned with an estimate for the interface unknowns of the domain. This estimate is obtained from the first iteration of the D-DDM family with an artificial coarse problem based on optional admissible constraints [36, 37]. Thus, this P-DDM family is in fact a PSM algorithm with the following preconditioner:

$$\tilde{A}^{-1} = L_{p_b}^T \left(S^{s^{-1}} - S^{s^{-1}} B_b^T C \left(C^T F_I C \right)^{-1} C^T B_b S^{s^{-1}} \right) L_{p_b} \tag{4.6.10}$$

As in the case of the D-DDM family with an artificial coarse problem based on optional admissible constraints, any properly chosen matrix C with linearly independent columns will exhibit superior convergence properties compared to the PSM because of the implicit introduction of a coarse problem.

Implementation-wise, the preconditioning step of a primal DDM when solved with a PCG algorithm is denoted by a matrix-vector multiplication of the following form:

$$z^k = \tilde{S}^{-1} r^k \tag{4.6.11}$$

The preconditioner shown in Eq. (4.6.10) is never constructed explicitly. Instead, the following calculations take place at the preconditioning step:

Evaluate $\chi = L_{p_b} r^k$

Solve $S^s x = \chi$

Evaluate $d = B_b x$

Evaluate $e = C^T b$

Solve $\left(C^T W\right) y = e$

Evaluate $l = Cy$

Evaluate $p = B^T l$

Solve $S^s q = p$

Evaluate ${z_1}^k = L_{n_i}^T x$

Evaluate ${z_2}^k = L_{n_i}^T q$

Evaluate $z^k = {z_1}^k - {z_2}^k$

It can be easily seen that these steps are a super-set of the calculations performed for the projector evaluation of the D-DDM family with an artificial coarse problem based on optional admissible constraints, with the exception of the calculation in Eq. (4.6.8). As result, storage and computational costs related to the calculation of the preconditioner are directly related to both the size and the efficient computation of matrix C.

4.6.4 D-DDM-S AND P-DDM-S: SOLID BASED D-DDM AND P-DDM FOR ONE-PHASE AND POROUS MEDIA PROBLEMS

For structural dynamics problems, matrix C can be set equal to [37, 42]:

$$C = QG_I \tag{4.6.12}$$

where:

$$G_I = \left[B^{(1)} R^{(1)} \quad \cdots \quad B^{(Ns)} R^{(Ns)} \right] \tag{4.6.13}$$

$$R^{(s)} = \text{null}\left(\widehat{K}^{(s)}\right) \tag{4.6.14}$$

and $\widehat{K}^{(s)}$ is the coefficient matrix of a subdomain for the corresponding static structural problem with all its displacement boundary conditions removed. Calculating the null space of $\widehat{K}^{(s)}$ can be done using geometric-algebraic algorithms, which are very cost-effective and robust. Matrix Q can be set equal to unity or according to eqs. (4.5.20), (4.5.21) , depending on the nature of the problem (homogeneous, heterogeneous, fourth-order problems).

The same principal can be applied to porous media problems with $\widehat{K}^{(s)}$ being the coefficient matrix of the soil skeleton, thus constituting the D-DDM-S method. Its primal counterpart is the P-DDM-S method which consists in applying the PSM with the following preconditioner:

$$\tilde{A}^{-1} = L_{p_b}^T \left(S^{s^{-1}} - S^{s^{-1}} B_b^T QG \left(G^T Q^T F_I QG \right)^{-1} G^T Q^T B_b S^{s^{-1}} \right) L_{p_b} \tag{4.6.15}$$

4.6.5 D-DDM-P AND P-DDM-P: PERMEABILITY BASED D-DDM AND P-DDM FOR POROUS MEDIA PROBLEMS

An alternative to the use of the soil skeleton stiffness matrix null space for the construction of matrix *C,* is the usage of the permeability matrix null space [42]. In this case, matrix *C* becomes equal to:

$$C = E_I \tag{4.6.16}$$

where:

$$E_I = \left[B^{(1)} R_H{}^{(1)} \quad \cdots \quad B^{(Ns)} R_H{}^{(Ns)} \right] \tag{4.6.17}$$

$$R_H{}^{(s)} = \text{null}\left(\widehat{H}^{(s)} \right) \tag{4.6.18}$$

and $\widehat{H}^{(s)}$ is the permeability coefficient matrix of subdomains with all their pore boundary conditions removed. This choice of matrix C constitutes the D-DDM-P method.

The null space of $\widehat{H}^{(s)}$ is equivalent to the null space of a structural problem with one dof per node and is equal to a vector with equal values at each position (ie. unity). This equivalence stems from the fact that the stiffness matrix has the same form as the permeability matrix assuming that the structural problem has one dof per node. In particular, if $S = \frac{\partial}{\partial x}$ and $D = E$ the generic expression of the stiffness matrix becomes

$$\mathrm{K} = \int B^{\mathrm{T}} DBd\Omega = \int (SN)^T ESNdx = \int \left(\frac{\partial}{\partial x} N \right)^T E \frac{\partial}{\partial x} Ndx \tag{4.6.19}$$

where *E* is a scalar denoting the Young’s modulus. The above expression is identical to the expression of Eq. (14) for the permeability matrix $\widehat{H}^{(s)}$.

The primal counterpart of D-DDM-P is the P-DDM-P method which consists in applying the PSM with the following preconditioner:

$$\tilde{A}^{-1} = L_{p_b}^{T} \left(S^{s^{-1}} - S^{s^{-1}} B_b^{T} E_I \left(E_I^{\,T} F_I E_I \right)^{-1} E_I^{\,T} B_b S^{s^{-1}} \right) L_{p_b} \tag{4.6.20}$$

Both D-DDM-S and D-DDM-P methods, along with their corresponding primal counterparts P-DDM-S and P-DDM-P, are variants of the DDM family with an artificial coarse problem based on optional admissible constraints, varying on the selection of matrix C. This selection defines a number of properties of the method, such as the size of the coarse problem, the projector evaluation time, storage requirements, number of iterations and computational time.

All these properties are directly related to the size of matrix C. With $n_{dof}{}^{(s)}$ being the number of dof per subdomain, the size of matrix C, for a 3D continuum problem in the case of D-DDM-S and P-DDM-S, is $n_{dof}{}^{(s)} \times 6$ per subdomain, where as in the case of D-DDM-P and P-DDM-P, the corresponding size per subdomain is only $n_{dof}{}^{(s)} \times 1$. This means that the size of the porous coarse problem with 200 subdomains grows by three orders of magnitude for the D-DDM-S and P-DDM-S variants, as opposed to only two orders of magnitude when using D-DDM-P and P-DDM-P. This difference of one order of magnitude has a significant impact on the efficiency of the D-DDM-P and P-DDM-P methods compared to the D-DDM-S and P-DDM-S methods.

Eq. (4.6.7) shows that the projector evaluation time is directly proportional to the size of matrix C. Since matrix W has to be stored in order to accelerate the projector evaluation for each iteration, the size of matrix W is also directly proportional to the storage requirements. Finally, eqs. (4.6.8) and (4.6.9) show that the required time per iteration is also proportional to the size of the chosen matrix C, since both calculations are time-consuming.

4.7 SOLUTION METHODS FOR STOCHASTIC PROBLEMS

The most straightforward technique of solving stochastic partial differential equations (PDE) are the widely applicable non-intrusive Monte Carlo (MC) methods. They can handle any type of problems (linear, nonlinear, dynamic) as well as any kind of uncertainty in the load or in the system properties. In particular, when dealing with deterministic external loading, MC methods feature the solution of successive linear systems with multiple left-hand sides, since only the coefficient matrix K changes in every simulation. On the other hand, recently proposed approaches, such as stochastic collocation and Galerkin methods, are intrusive and are using tensor product spaces for the spatial and stochastic discretizations. In the case where the uncertain input parameters are modeled via the Karhunen-Loeve (KL) expansion and the system response is projected on a polynomial chaos (PC) basis, the method is called spectral stochastic finite element method (SSFEM). SSFEM approach applies a Galerkin minimization in order to transform a stochastic PDE into a coupled set of deterministic PDEs.

In MC methods due to the fact that the solution process has to start from the beginning, a new stiffness matrix needs to be formed at each simulation. Thus, the repeated solutions of the system of equations for each newly formed solution becomes a major computational task that hinders the stochastic assessment of large-scale problems with MC methods. In SSFEM approach the solution of stochastic problems has to be performed on augmented linear equation systems which can be orders of magnitude larger than the corresponding deterministic ones [25, 24] and thus, as in MC methods, for large-scale problems the solution of such augmented algebraic systems can become quite challenging due to the increased memory and computational resources required.

In the following sections, a set of custom-tailored solution methods will be presented, combining iterative solution methods and domain decomposition, providing superior numerical performance.

4.7.1 THE MC-PCG METHOD FAMILY

In high performance computing environments which feature computing systems with multicore processors and distributed memory architectures, iterative schemes are more advantageous since they manage to harness the computational power of such environments while being custom tailored to the particular properties of the equilibrium equations arising in the context of the numerical simulation used. In order to efficiently solve the resulting algebraic equations of a MC simulation, an iterative solver based on the PCG algorithm (see Chapter 4.3) is implemented.

The PCG algorithm equipped with a preconditioner following the rationale of incomplete Cholesky preconditioning features an error matrix E_i. This matrix is dependent on the discarded elements of the lower triangular matrix produced by the incomplete Cholesky factorization procedure, which do not satisfy a specified magnitude or position criterion [43]. Considering the near-by problems of the form:

$$\left(K_0 + \Delta K_i\right)u_i = f,\ i = 1 \ldots n_{sim} \tag{4.7.1}$$

if matrix E_i is taken as K_i, the preconditioning matrix becomes the initial matrix $\tilde{A} = K_0$. The PCG algorithm equipped with the latter preconditioner throughout the entire solution process constitutes the MC-PCG-Skyline method for the solution of the nsim near-by problems of eq. (4.7.1).

With the preconditioning matrix $\tilde{A} = K_0$ remaining the same during the successive Monte Carlo simulations, the repeated solutions required for the evaluation of the preconditioned residual vector $z^k = \tilde{A}^{-1} r^k$ can be treated as problems with multiple right-hand sides, since this vector needs to be evaluated at each PCG iteration k of each simulation i. In order for this evaluation to be efficient, a solution scheme capable of solving efficiently problems with multiple right-hand sides is required.

The original MC-PCG-K_0 algorithm proposed in [44] uses a Cholesky direct solver for performing the proconditioning step, where K_0 is factorized to LL^T at the beginning of the Monte Carlo simulation procedure. Subsequently, each evaluation of the preconditioned residual vector is carried out by a forward substitution, a vector operation and a backward substitution. Another implementation for obtaining the preconditioned residual vector z^k is proposed [45] where the dual domain decomposition FETI method is applied to perform the repeated solutions in parallel computing environment and is called the MC-PCG-FETI method. In the present work, each evaluation of the preconditioned residual vector is carried out using a PFETI solver [38], optimized for multiple right-hand sides [36] (see Chapter 4.3.2), adhering to the rationale of the PCG method, where the preconditioning step is performed with the FETI method. The latter method is called the MC-PCG-PFETI method.

4.7.2 THE SSFEM-PCG METHOD FAMILY

The augmented systems that are generated when using SSFEM are perfect candidates for iterative solvers since iterative solvers are flexible enough to be custom tailored to their particular properties while being suitable for high performance computing environments.

Current literature has provided solution procedures for solving eq. (3.8.25) that address small to medium problems. However, as the problem size grows, such a solution can become quite challenging due to the enormous memory and computational resources required. Contemporary efficient solution techniques are based on iterative solvers like the block Gauss-Jacobi [46], the CG [47, 28] and the block PCG [48].

In this work, specialized preconditioners that take advantage of the properties of the augmented SSFEM linear systems are used. In particular, two solution preconditioners are used and compared for efficiency for the case of Gaussian distribution. The first one is of the form:

$$\tilde{A} = \begin{bmatrix} a_1 K_0 & 0 & \dots & 0 \\ 0 & a_2 K_0 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & a_n K_0 \end{bmatrix} 11 \qquad (4.7.2)$$

where $a_i, i = 1 \dots n$ are the coefficients as calculated from the polynomial chaos bases as shown in described in chapter 3. For each evaluation of the preconditioned residual vector, the same K_0 matrix needs to be inverted *n* times and, as in the case of the MC-PCG-K$_0$ method, this matrix inversion is implemented as a linear system solution. Since matrix $\tilde{A}$ is block diagonal, the solution process can be pipelined as the successive solution of *n* linear systems with multiple right-hand sides. The PCG algorithm equipped with the latter preconditioner and utilizing the PFETI method for solving the successive linear systems, introduced during the evaluation of the preconditioned residual vector, constitutes the SSFEM-PCG-B method for the solution of the augmented linear system that is the outcome from the SSFEM implementation.

The second preconditioner is based on the rationale of the SSOR-based preconditioners. In particular, the augmented matrix K is decomposed into a diagonal component D, and a strictly lower triangular component L of the form:

$$L = \begin{bmatrix} 0 & 0 & \dots & 0 \\ K_{21} & 0 & \dots & 0 \\ \vdots & K_{m2} & \ddots & 0 \\ K_{n1} & K_{n2} & \dots & 0 \end{bmatrix} \tag{4.7.3}$$

Using this decomposition, the aforementioned preconditioner is of the form:

$$\tilde{A} = (D-L)D^{-1}\left(D-L^T\right) \Leftrightarrow \tilde{A}^{-1} = \left(D-L^T\right)^{-1} D (D-L)^{-1} \tag{4.7.4}$$

In this work, evaluation of the preconditioned residual vector of the PCG algorithm is implemented as follows:

- Solve $(D-L)z_1^k = r^k$ (4.7.5)
- Evaluate $z_2^k = Dz_1^k$
- Solve $\left(D-L^T\right)z^k = z_2^k$ (4.7.6)

The linear system (4.7.5) is lower triangular and its solution involves the implementation of a forward substitution algorithm in block form, as follows:

$$\begin{array}{ccccccccc} a_{11}K_0x_1 & & & & & & & = & b_1 \\ a_{21}K_1x_2 & + & a_{22}K_0x_2 & & & & & = & b_2 \\ \vdots & & \vdots & & \ddots & & & & \vdots \\ a_{m1}K_1x_m & + & a_{m2}K_2x_2 & + & \dots & + & a_{mm}K_mx_m & = & b_m \end{array}$$

where $r^k = \begin{bmatrix} b_1^T & b_2^T & \dots & b_m^T \end{bmatrix}^T$, $z_1^k = \begin{bmatrix} x_1^T & x_2^T & \dots & x_m^T \end{bmatrix}^T$ and $a_{xy}K_z$ are the various block matrices as they occur from the formulation of the SSFEM augmented system. The evaluation of these block equations are executed in a sequential manner and are implemented as the successive solution of the following linear systems:

$$\begin{aligned} K_0 x_1 &= \frac{b_1}{a_{11}} \\ K_0 x_2 &= \frac{b_2 - a_{21} K_1 x_1}{a_{22}} \\ &\vdots \\ K_0 x_m &= \frac{b_m - \sum_{i=1}^{m-1} a_{mi} K_i x_i}{a_{mm}} \end{aligned} \tag{4.7.7}$$

Similarly, the linear system (4.7.6) is upper triangular and its solution involves the implementation of a backward substitution algorithm in block form, as follows:

$$\begin{array}{ccccccccc} a_{11}K_0 y_1 & + & a_{12}K_2 y_2 & + & \dots & + & a_{1m}K_m y_m & = & c_1 \\ & & a_{22}K_0 y_2 & + & \dots & + & a_{2m}K_n y_m & = & c_2 \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{mm}K_0 y_m & = & c_m \end{array}$$

where $z_2^k = \begin{bmatrix} c_1^T & c_2^T & \dots & c_m^T \end{bmatrix}^T$ and $z^k = \begin{bmatrix} y_1^T & y_2^T & \dots & y_m^T \end{bmatrix}^T$

The evaluation of these block equations are executed as the successive solution of the following linear systems:

$$\begin{aligned} K_0 y_m &= \frac{c_m}{a_{mm}} \\ K_0 y_{m-1} &= \frac{c_{m-1} - a_{m-1,m-1} K_{m-1} x_{m-1}}{a_{m-1,m-1}} \\ &\vdots \\ K_0 y_1 &= \frac{c_1 - \sum_{i=1}^{m-1} a_{1i} K_i x_i}{a_{11}} \end{aligned} \tag{4.7.8}$$

The PCG algorithm equipped to the latter SSOR preconditioner and utilizing the PFETI method for solving the linear systems occurring from the aforementioned forward and backward substitutions, constitutes the SSFEM-PCG-S method for the solution of the augmented linear system that occurs from the SSFEM.

It is worth noting that in contrast with the common SSOR preconditioner applied in an iterative solver, the evaluation of the preconditioned residual vector of the SSFEM-PCG-S method is parallel and scalable due to the fact that all block matrices that take part on the matrix-vector multiplications operations of the forward and backward substitutions are already decomposed into subdomains. This means that each operation of these forward and backward substitutions, including both the matrix-vector multiplications and the solution process, are carried out in parallel, exhibiting the scalability of the PFETI method [38].

Implementing the A matrix-vector product

The augmented systems that are generated from the application of the SSFEM involve large coefficient matrices that feature a block form. Each block is comprised of a linear combination of stiffness matrix realizations that have identical structure and bandwidth with the deterministic matrix K_0.

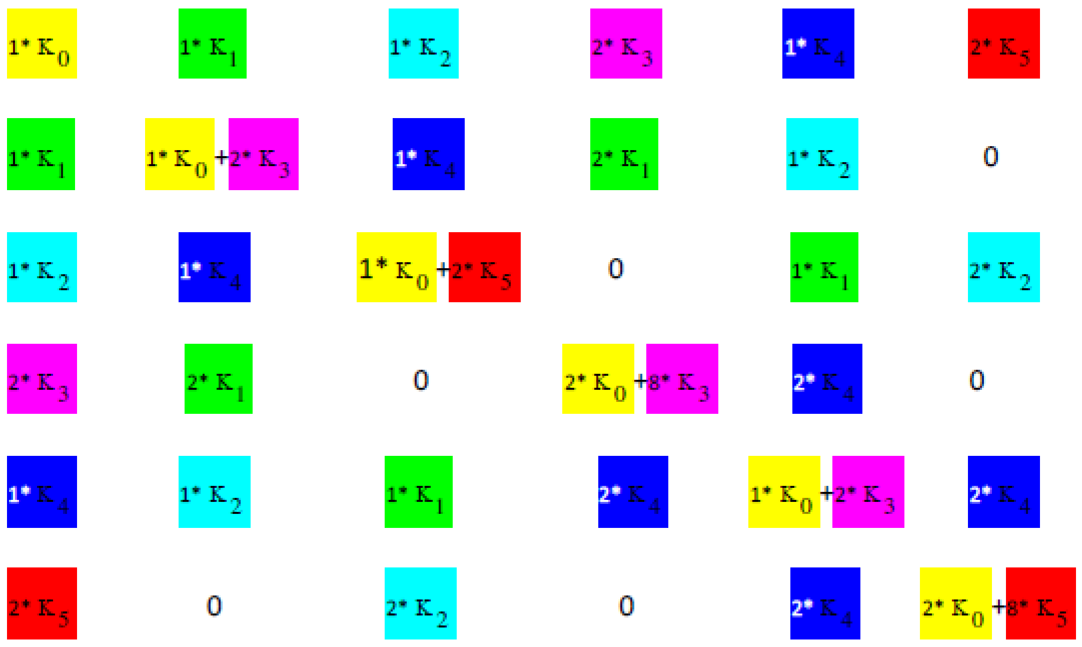


Figure 4.2 - Topology of K for the lognormal case (M=2, p=2)

Both the SSFEM-PCG-B and SSFEM-PCG-S methods need the evaluation of the A matrix-vector product at each iteration i. In this work, this evaluation is performed as a series of matrix-vector multiplications where the evaluated vectors are linearly combined in order to form the resulting vector. An example of this process for the case of a SSFEM augmented linear system as shown in Figure 4.2, where the A matrix-vector product $q^k = Ap^k$ for iteration k is evaluated in 6 consecutive steps, is shown below:

Initialization step:

$$p^k = \begin{bmatrix} p_0^{k^{\mathrm{T}}} & p_1^{k^{\mathrm{T}}} & p_2^{k^{\mathrm{T}}} & p_3^{k^{\mathrm{T}}} & p_4^{k^{\mathrm{T}}} & p_5^{k^{\mathrm{T}}} \end{bmatrix}^{\mathrm{T}}$$

$$_0q^k = \begin{bmatrix} {}_0q_0^{k^{\mathrm{T}}} & {}_0q_1^{k^{\mathrm{T}}} & {}_0q_2^{k^{\mathrm{T}}} & {}_0q_3^{k^{\mathrm{T}}} & {}_0q_4^{k^{\mathrm{T}}} & {}_0q_5^{k^{\mathrm{T}}} \end{bmatrix}^{\mathrm{T}} = 0$$

Step i=1:

$$\begin{aligned}
{}_iq_0^k &= {}_{i-1}q_0^k + K_0 p_0^k \\
{}_iq_1^k &= {}_{i-1}q_1^k + K_0 p_1^k \\
{}_iq_2^k &= {}_{i-1}q_2^k + K_0 p_2^k \\
{}_iq_3^k &= {}_{i-1}q_3^k + K_0 \cdot 2 p_3^k \\
{}_iq_4^k &= {}_{i-1}q_4^k + K_0 p_4^k \\
{}_iq_5^k &= {}_{i-1}q_5^k + K_0 \cdot 2 p_5^k
\end{aligned}$$

Step i=2:

$$\begin{aligned}
{}_iq_0^k &= {}_{i-1}q_0^k + K_1 p_0^k \\
{}_iq_1^k &= {}_{i-1}q_1^k + K_1 p_0^k + K_1 \cdot 2 p_3^k \\
{}_iq_2^k &= {}_{i-1}q_2^k + K_1 p_4^k \\
{}_iq_3^k &= {}_{i-1}q_3^k + K_1 \cdot 2 p_1^k \\
{}_iq_4^k &= {}_{i-1}q_4^k + K_1 p_2^k
\end{aligned}$$

Step i=3:

$$\begin{aligned}
{}_iq_0^k &= {}_{i-1}q_0^k + K_2 p_2^k \\
{}_iq_1^k &= {}_{i-1}q_1^k + K_2 p_4^k \\
{}_iq_2^k &= {}_{i-1}q_2^k + K_2 p_1^k + K_2 \cdot 2 p_5^k \\
{}_iq_4^k &= {}_{i-1}q_4^k + K_2 p_1^k \\
{}_iq_5^k &= {}_{i-1}q_5^k + K_2 \cdot 2 p_2^k
\end{aligned}$$

Step i=4:

$$\begin{aligned}
{}_iq_0^k &= {}_{i-1}q_0^k + K_3 \cdot 2 p_3^k \\
{}_iq_1^k &= {}_{i-1}q_1^k + K_3 \cdot 2 p_1^k \\
{}_iq_3^k &= {}_{i-1}q_3^k + K_3 \cdot 2 p_0^k + K_3 \cdot 8 p_3^k \\
{}_iq_4^k &= {}_{i-1}q_4^k + K_3 \cdot 2 p_4^k
\end{aligned}$$

Step i=5:
$$\begin{aligned}
{}_{i}q_0^k &= {}_{i-1}q_0^k + K_4 p_4^k \\
{}_{i}q_1^k &= {}_{i-1}q_1^k + K_4 p_2^k \\
{}_{i}q_2^k &= {}_{i-1}q_2^k + K_4 p_1^k \\
{}_{i}q_3^k &= {}_{i-1}q_3^k + K_4 \cdot 2 p_4^k \\
{}_{i}q_4^k &= {}_{i-1}q_4^k + K_4 p_0^k + K_4 \cdot 2 p_5^k \\
{}_{i}q_5^k &= {}_{i-1}q_5^k + K_4 \cdot 2 p_4^k
\end{aligned}$$

Step i=6:
$$\begin{aligned}
{}_{i}q_0^k &= {}_{i-1}q_0^k + K_5 \cdot 2 p_5^k \\
{}_{i}q_2^k &= {}_{i-1}q_2^k + K_5 \cdot 2 p_2^k \\
{}_{i}q_5^k &= {}_{i-1}q_5^k + K_5 \cdot 2 p_0^k + K_5 \cdot 8 p_5^k
\end{aligned}$$

with ${}_{6}q^k = q^k$. By examining these steps, it is evident that the A matrix-vector product evaluation is computationally intensive since for a 6x6 matrix, 34 matrix-vector (MV) products are being computed. In this work, a caching scheme has been developed for the log-normal case, in order to minimize the computational burden of this evaluation, which is described in the following section.

A caching scheme for the log-normal case

Stochastic problems modeled with input random fields featuring a log-normal distribution, produce coefficient matrices that have three major differences when compared to the ones produced from a problem modeled with a Gaussian distribution. The coefficient matrix sparsity in a log-normal case is

i. much denser when compared to the Gaussian ones,
ii. each block position might be the result of a linear combination of the stochastic matrices while in Gaussian coefficient matrices, each block position is occupied by a stochastic matrix multiplied by an integer coefficient and
iii. the structure of a number of block diagonal matrices is the result of a linear combination of the deterministic matrix and some of the stochastic matrices, while for Gaussian coefficient matrices, each block position is occupied only by the deterministic matrix multiplied by an integer coefficient.

P	**size multiplier**	**non-zero blocks (Gauss)**	**non-zero block (log-normal)**
2	6	12	30
3	10	24	100

4	15	40	324

Table 4.3 - Sparsity of the resulting augmented stochastic Karhunen-Loeve expansion for M=2

These differences affect considerably the performance of the solvers, with respect to the number of iterations necessary for convergence and the amount of computations required to perform each PCG iteration. The computational effort is further magnified due to a more cumbersome implementation of the A matrix-vector product evaluations that are needed in each iteration of the SSFEM-PCG-B and SSFEM-PCG-S methods.

As shown in the previous section, the augmented matrix is not formulated as a whole and each product is being evaluated by multiplying every block matrix with its corresponding block vector and accumulating the partial results to the corresponding position of the resulting vector. This means that if a block position of the coefficient matrix comprises a linear combination of *n* terms, *n* matrix-vector products, must be evaluated, just for the complete calculation of this position's contribution to the result vector. For a Gaussian field however, only one matrix-vector product for each block position needs to be evaluated.

In order to alleviate this additional computational effort, a caching scheme has been applied where each unique linear combination that occurs in every block position is being pre-calculated and stored in order to reduce the computation cost of each A matrix-vector evaluation at each iteration of the PCG-B and PCG-S methods.

In order to demonstrate this technique, we consider M=2, p=2 for the Log-normal case as depicted in Figure 4.2 where the linear combination $1K_0 + 2K_3$ is found two times. At the uncached case, we would need to perform 4 matrix-vector operations and 4 linear scaling operations in order to compute the contribution of these two terms at the final term, as shown in the previous section. However, for the cached case where the linear combination $1K_0 + 2K_3$ is stored as a separate matrix, only 2 matrix-vector operations need to be performed.

The following tables compare the amount of matrix-vector products needed for a Kahrunen-Loeve expansion of M=4 and a polynomial chaos expansion p varying from 4 to 6.

P	matrices stored	non-zero blocks	total mv products
4	70	3090	4937
5	126	10158	19542

6	210	29448	70952

Table 4.4 - Metrics for the normal scheme of the SSFEM-PCG family

P	matrices stored	non-zero blocks	total mv products
4	710	3090	3090
5	2109	10158	10158
6	6064	29448	29448

Table 4.5 - Metrics for the cached scheme of the SSFEM-PCG family

This caching scheme requires one to two orders of magnitude more computer memory resources for storing the corresponding stiffness matrices but it can offer significant performance benefits as it is shown in the numerical examples section of this work.

A full block preconditioning scheme for the log-normal case

The existence of linear combinations of the deterministic matrix with stochastic ones at the block diagonal of the coefficient matrix can really deteriorate the convergence rate of the block diagonal preconditioner of the SSFEM-PCG-B method. This is more pronounced at large input covariances where the magnitude of the stochastic matrices is comparable to the magnitude of the deterministic one. This is also the case for the SSFEM-PCG-S method where the solution of a linear system at the end of each block row of the preconditioner is required. If only the deterministic part is taken into account, convergence will deteriorate for large input covariances.

Instead of using the PFETI solver, optimized for multiple right-hand sides as per the SSFEM-PCG-B and SSFEM-PCG-S solvers, a PCG solver is used for the full linear combination having the aforementioned PFETI solver as its preconditioner, as in the case of the MC-PCG-K_0 solver described earlier.

As in the case of the Monte Carlo simulations, the repeated solutions required for the preconditioning step of the MC-PCG-K_0 algorithm can be treated as problems with multiple right-hand sides, since the entries in the residual vector are updated at each PCG iteration m of each block diagonal part of the coefficient matrix.

In order to illustrate this technique, we consider the augmented stiffness matrix of Figure 4.2. For the SSFEM-PCG-B method, the preconditioner is of the form:

$$\tilde{A} = \begin{bmatrix} K_0 & & & & & \\ & K_0 + 2K_3 & & & & \\ & & K_0 + 2K_5 & & & \\ & & & K_0 + 8K_3 & & \\ & & & & K_0 + 2K_3 & \\ & & & & & 2K_0 + 8K_5 \end{bmatrix}$$

This means that for each iteration k of the SSFEM-PCG-B method, the preconditioned residual vector involves the solution of the following linear systems:

$$\begin{aligned} &K_0 z_0^k = r_0^k \\ &\left(K_0 + 2K_3\right) z_1^k = r_1^k \\ &\left(K_0 + 2K_5\right) z_2^k = r_2^k \\ &\left(K_0 + 8K_3\right) z_3^k = r_3^k \\ &\left(K_0 + 2K_3\right) z_4^k = r_4^k \\ &\left(2K_0 + 8K_5\right) z_5^k = r_5^k \end{aligned}$$

with $z^k = \begin{bmatrix} z_0^{k^{\mathrm{T}}} & z_1^{k^{\mathrm{T}}} & z_2^{k^{\mathrm{T}}} & z_3^{k^{\mathrm{T}}} & z_4^{k^{\mathrm{T}}} & z_5^{k^{\mathrm{T}}} \end{bmatrix}^{\mathrm{T}}$ & $r^k = \begin{bmatrix} r_0^{k^{\mathrm{T}}} & r_1^{k^{\mathrm{T}}} & r_2^{k^{\mathrm{T}}} & r_3^{k^{\mathrm{T}}} & r_4^{k^{\mathrm{T}}} & r_5^{k^{\mathrm{T}}} \end{bmatrix}^{\mathrm{T}}$.

The PCG algorithm equipped with the preconditioner of the SSFEM-PCG-B method and utilizing the MC-PCG-K$_0$ method for solving the linear systems occurring at the preconditioned residual vector evaluation, constitutes the SSFEM-PCG-BF method for the solution of the augmented linear system that occurs from the SSFEM.

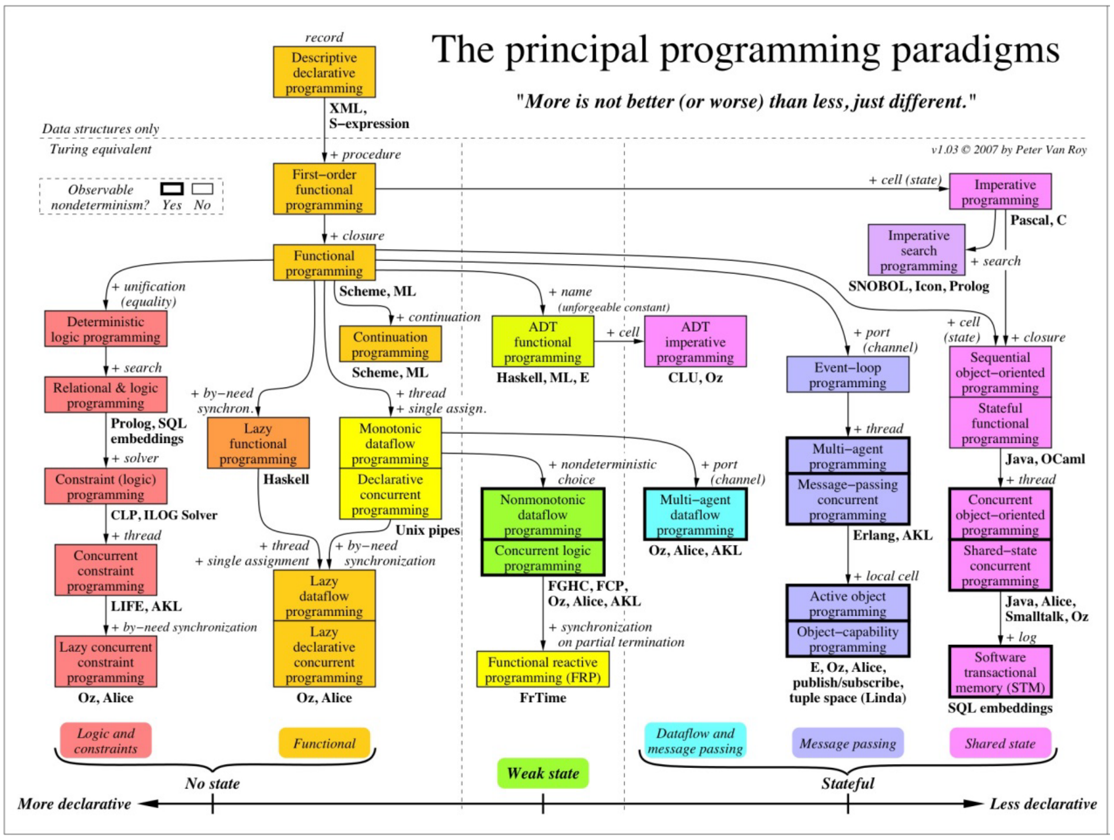
In the same fashion, the PCG-S method requires the successive solution of systems (4.7.5) and (4.7.6) which for the case considered are of the form:

$$\begin{aligned} &K_0 x_1 = \tilde{b}_1 \\ &\left(K_0 + 2K_3\right) x_2 = \tilde{b}_2 \\ &\left(K_0 + 2K_5\right) x_3 = \tilde{b}_3 \\ &\left(K_0 + 8K_3\right) x_4 = \tilde{b}_4 \\ &\left(K_0 + 2K_3\right) x_5 = \tilde{b}_5 \\ &\left(2K_0 + 8K_5\right) x_6 = \tilde{b}_6 \end{aligned} \quad \text{and} \quad \begin{aligned} &\left(2K_0 + 8K_5\right) y_6 = \tilde{c}_6 \\ &\left(K_0 + 2K_3\right) y_5 = \tilde{c}_5 \\ &\left(K_0 + 8K_3\right) y_4 = \tilde{c}_4 \\ &\left(K_0 + 2K_5\right) y_3 = \tilde{c}_3 \\ &\left(K_0 + 2K_3\right) y_2 = \tilde{c}_2 \\ &K_0 y_1 = \tilde{c}_1 \end{aligned}$$

with vectors $\tilde{b}_i$ and $\tilde{c}_i$ begin equal to the right hand sides as shown in eqs. (4.7.7) and (4.7.8) respectively.

The PCG algorithm equipped with the block SSOR preconditioner of the SSFEM-PCG-S method and utilizing the MC-PCG-K_0 method for solving the linear systems occurring at the preconditioned residual vector evaluation, constitutes the SSFEM-PCG-SF method for the solution of the augmented linear system that occurs from the SSFEM.

5 PROGRAMMING PARADIGMS


The principal programming paradigms
"More is not better (or worse) than less, just different."
v1.03 © 2007 by Peter Van Roy
record
Descriptive declarative programming
XML, S−expression
Data structures only
Turing equivalent
+ procedure
First−order functional programming
Observable nondeterminism? Yes No
+ cell (state)
Imperative programming
Pascal, C
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ closure
Functional programming
Scheme, ML
+ unification (equality)
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by−need synchronization
Lazy concurrent constraint programming
Oz, Alice
+ continuation
Continuation programming
Scheme, ML
+ by−need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Unix pipes
+ thread
+ single assignment
+ by−need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, Oz
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
FGHC, FCP, Oz, Alice, AKL
+ synchronization on partial termination
Functional reactive programming (FRP)
FrTime
+ port (channel)
Multi−agent dataflow programming
Oz, Alice, AKL
+ port (channel)
Event−loop programming
+ thread
Multi−agent programming
Message−passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object−capability programming
E, Oz, Alice, publish/subscribe, tuple space (Linda)
+ cell (state)
+ closure
Sequential object−oriented programming
Stateful functional programming
Java, OCaml
+ thread
Concurrent object−oriented programming
Shared−state concurrent programming
Java, Alice, Smalltalk, Oz
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Weak state
Dataflow and message passing
Message passing
Shared state
No state
Stateful
More declarative
Less declarative

5.1 THE OBJECT-ORIENTED PARADIGM

Object-oriented technology is built upon a sound engineering foundation, whose elements are collectively called the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. None of these principles are new but, in object-oriented programming, these elements are brought together in a synergistic way.

Object-oriented analysis and design is fundamentally different than traditional structured design approaches since it requires a different way of thinking about problem decomposition. Moreover, it produces software architectures that are very different from those produced by following structured design patterns. These differences arise from the fact that structured design methods build upon structured programming, whereas object-oriented design builds upon the object model.

Looking back upon the history of software engineering, two trends are noticeable

• The shift in focus from programming-in-the-small to programming-in-the-large

• The evolution of high-order programming languages

Most new industrial-strength software systems are larger and more complex than their predecessors. This growth in complexity has prompted a significant amount of useful applied research in software engineering, particularly with regard to decomposition, abstraction, and hierarchy with the development of more expressive programming languages to having complemented these advances. The trend set is to move away from languages that tell the computer what to do (imperative languages) and lean towards languages that describe the key abstractions in the problem domain (declarative languages). In successive generations, the kind of abstraction mechanism each language supported changed.

First-generation languages were used primarily for scientific and engineering applications, and the vocabulary of this problem domain was almost entirely mathematics. Languages such as FORTRAN 1 were thus developed to allow the programmer to write mathematical formulas, thereby freeing the programmer from some of the intricacies of assembly or machine language. This first generation of high-order programming languages therefore represented a step closer to the problem space, and a step further away from the underlying machine.

Among second-generation languages, the emphasis was upon algorithmic abstractions. By this time, machines were becoming more and more powerful, and the economics of the computer industry meant that more kinds of problems could be automated, especially for business applications. Now, the focus was largely upon telling the machine what to do: read these personnel records first, sort them next, and then print this report. Again, this new generation of high-order programming languages moved a step closer to the problem space, and further away from the underlying machine.

By the late 1960s, especially with the advent of transistors and then integrated circuit technology, the cost of computer hardware had dropped dramatically, yet processing capacity had grown almost exponentially. Larger problems could now be solved, but these demanded the manipulation of more kinds of data. Thus, languages such as ALGOL 60 and, later, Pascal evolved with support for data abstraction. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. This generation of high-order programming languages again moved developers a step closer to the problem domain, and further away from the underlying machine.

In the 1970s, programming language research was very active, resulting in the creation of literally a couple of thousand different programming languages and their dialects. To a large extent, the drive to write larger and larger programs highlighted the inadequacies of earlier languages. As a result, many new language mechanisms were developed to address these limitations. Despite the fact that few of these languages survived, many of the concepts that they introduced found their way into successors of earlier languages. Characteristic examples are Smalltalk (a revolutionary successor to Simula), Ada (a successor to ALGOL 68 and Pascal, with contributions from Simula, Alphard, and CLU), CLOS (which evolved from Lisp, LOOPS, and Flavors), C++ (derived from a marriage of C and Simula), and Eiffel (derived from Simula and Ada).

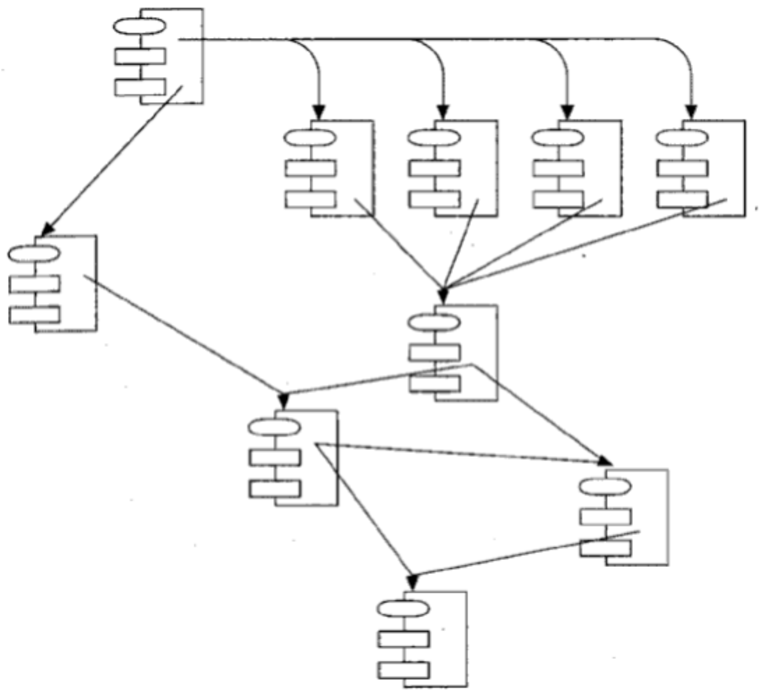


Figure 5.1 - Topology of OO languages for small- to moderate sized applications

The importance of data abstraction to mastering complexity is fundamental: “The nature of abstractions that may be achieved through the use of procedures is well suited to the description of abstract operations, but is not particularly well suited to the description of abstract objects. This is a serious drawback, for in many applications, the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem” [49]. This realization had two important consequences:

- Data-driven design methods emerged, which provided a disciplined approach to the problems of doing data abstraction in algorithmically oriented languages.
- Theories regarding the concept of a type appeared, which eventually found their realization in languages such as Pascal.

The natural conclusion of these ideas first appeared in the language Simula and was improved upon during the period of the language generation gap, resulting in the relatively recent development of several languages such as -Smalltalk, Object Pascal, C++, CLOS, Ada, and Eiffel which are object-oriented.

Figure 5.1 illustrates the topology of these languages for small- to moderate sized applications. The physical building block in these languages is the module which represents a logical collection of classes and objects instead of subprograms, as in earlier languages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [50]. For this reason, the physical structure of a small to moderate-sized object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented languages. Additionally, there is little or no global data but, instead, data and operations are united in such a way that the fundamental logical building blocks of a system are not algorithms but classes and objects.

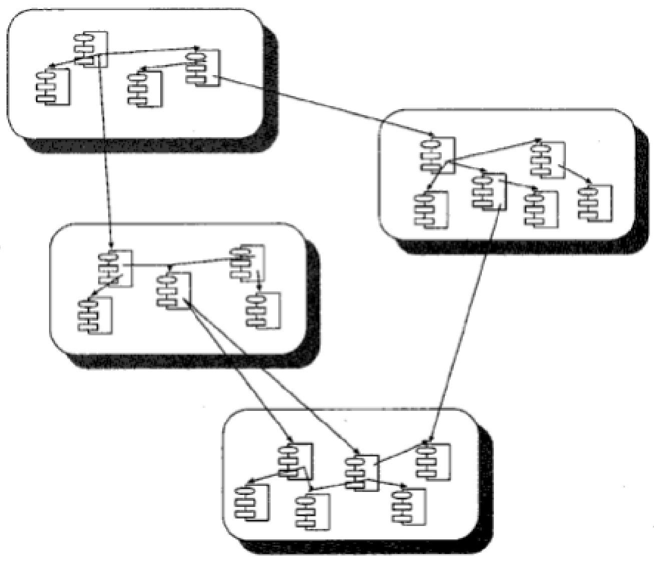


Figure 5.2 - Topology of large scale applications built with OO languages

For very complex systems, classes, objects, and modules provide an essential yet insufficient means of abstraction. However, the object model is scalable and in large systems, clusters of abstractions can be built in layers on top of one another. At any given level of abstraction, meaningful collections of objects can be composed that collaborate to achieve a higher level behavior. Examining any given cluster with respect to its implementation, more sets of cooperative abstractions can be found as shown in Figure 5.2.

Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks. Similarly, object-oriented design methods have evolved to help developers exploit the expressive power of object-oriented programming languages, using the class and object as basic building blocks.

Actually, the object model has been influenced by a number of factors, not just object-oriented programming. The object model has proven to be a unifying concept in computer science, applicable not just to programming languages, but also to the design of user interfaces, databases, and even computer architectures. The reason for this widespread appeal is simply that an object orientation helps to cope with the complexity inherent in many different kinds of systems.

Object-oriented analysis and design thus represents an evolutionary development, not a revolutionary one; it does not break with advances from the past, but builds upon proven ones. Unfortunately, most programmers today are formally and informally trained only in the principles of structured design. Despite the fact that countless useful software systems have been developed using these techniques, there are limits to the amount of complexity that can be tackled with, using only algorithmic decomposition. Furthermore, languages such as C++ and Java are used as if they were only traditional, algorithmically oriented languages, not only the language expressibility is underexploited but we usually end up worse off than if we had used an older language such as C or Pascal.

In the following sections, we will talk about the goals served from the FE code that was developed during this work and will dive into the details of the object oriented design properties and principals.

5.2 GOALS OF A FINITE ELEMENT COMPUTER CODE

During the implementation of finite element codes at an academic or research level, engineers tend to create programs that focus mainly on their research area. Since such codes have been built just to serve research purposes, there is no emphasis on the number of finite elements supported nor to the kind of problems (linear, non-linear, dynamic, stochastic, optimization, etc) being solved. On the contrary, programs are built as simple as possible, almost serving as a pilot or a proof of concept for the idea that the engineer has conceived. Moreover, when special solution techniques are involved, special manipulation of the data produced by the FEM (loads, stiffness matrices, mass matrices, etc) may be considered to such an extent that, basic infrastructure methods such as global stiffness matrix assembly may need to be rebuilt from scratch, in order to provide the appropriate data to the implemented solvers.

Matters get more complicated when dealing with solvers that operate in parallel environments. In order to fully exploit the power of today's parallel processing environments, programs must take into account both shared and distributed memory paradigms and also consider the utilization of external processing hardware such as GPUs or FPGAs. Finally, the data structures involved for each solver prove to be quite different which requires an underlying message passing infrastructure that is flexible enough to accommodate various data structures.

The main goal of the FE code that was implemented in this work is to provide a unified, abstract and extendable infrastructure that will provide future engineers with all the building blocks required to implement new features with respect to elements, problems and solvers, while providing performance and scalability.

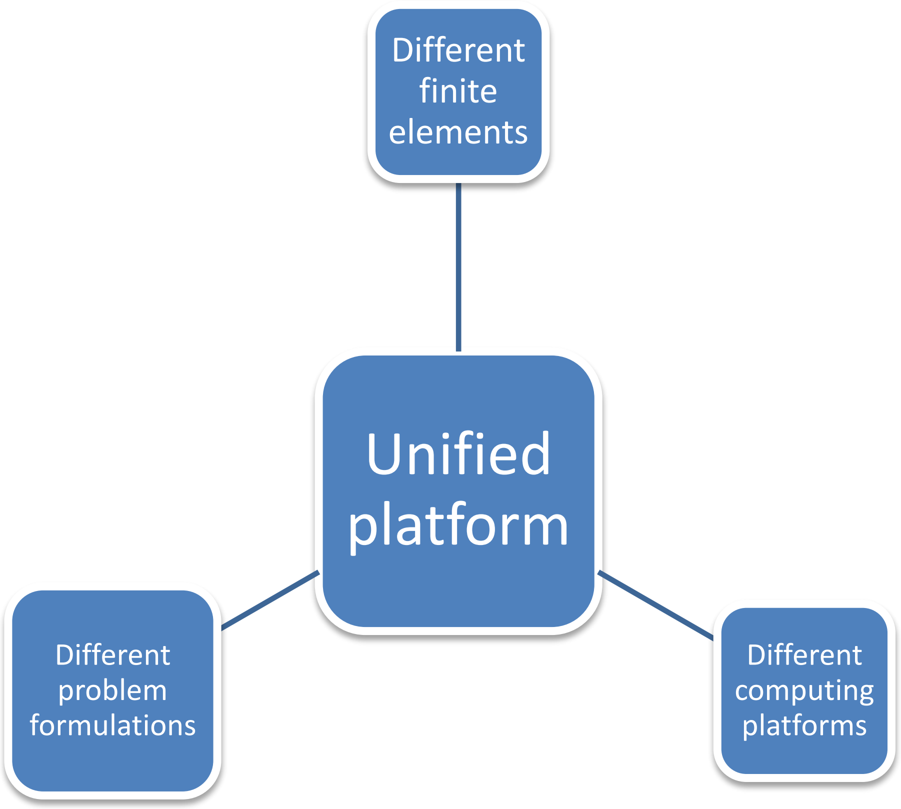

Different finite elements
Unified platform
Different problem formulations
Different computing platforms

Figure 5.3 - A unified FEM platform

5.3 ASPECTS OF OBJECT-ORIENTED PROGRAMMING

In this section, various object-oriented properties will be presented along with design decisions for the development and implementation of the finite element code used in this work.

5.3.1 MODULARITY

The act of partitioning a program into individual components can reduce its complexity to some degree. Moreover, it creates a number of well defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program [51]. In some languages, such as Smalltalk, there is no concept of a module, and so the class forms the only physical unit of decomposition. In many others, including Object Pascal, C++, CLOS, and Ada, the module is a separate language construct, and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system while these abstractions are placed in modules to produce the system's physical architecture. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity.

Moreover, modularization consists of dividing a program into modules which can be compiled separately but may have connections with other modules. As such, “The connections between modules are the assumptions which the modules make about each other” [52]. Most languages that support the module as a separate concept also distinguish between the interface of a module and its implementation.

Particular languages support modularity in diverse ways. For example, modules in C++ are nothing more than separately compiled files. The traditional practice in the C/C++ community is to place module interfaces in files named with an h suffix; these are called header files. Module implementations are placed in files named with a c suffix. Dependencies among files can then be asserted using the #include macro. This approach is entirely one of convention; it is neither required nor enforced by the language itself. Object Pascal is a little more formal about the matter. In this language, the syntax for units (its name for modules) distinguishes between module interface and implementation. Dependencies among units may be asserted only in a module's interface. Ada goes one step further. A package (its name for modules) has two parts: the package specification and the package body. Unlike Object Pascal, Ada allows connections among modules to be asserted separately in the specification and body of a package. Thus, it is possible for a package body to depend upon modules that are otherwise not visible to the package's specification.

Deciding upon the right set of modules for a given problem is almost as hard a problem as deciding upon the right set of abstractions and according to [53] because the solution may not be known when the design stage starts, decomposition into smaller modules may be quite difficult. For older applications (such as compiler writing), this process may become standard, but for new ones (such as defense systems or spacecraft control), it may be quite difficult.

Modules serve as the physical containers in which we declare the classes and objects of our logical design. For tiny problems, the developer might decide to declare every class and object in the same package. One better solution is to group logically related classes and objects in the same module, and expose only those elements that other modules absolutely must see. This kind of modularization is beneficial but can be taken to extremes. Let’s consider an application that runs on a distributed set of processors and uses a message passing mechanism to coordinate the activities of different programs. In a large system, it is common to have several hundred or even a few thousand kinds of messages. A naive strategy might be to define each message class in its own module. Considering this approach, it turns out to be a poor design decision because it creates a documentation nightmare and makes it extremely difficult for any users to find the classes they need. Furthermore, when decisions change, hundreds of modules must be modified or recompiled. This example shows how information hiding can have adverse effects [54].

Arbitrary modularization can prove worse when compared to complete lack of modularization. In traditional structured design, modularization is primarily concerned with the meaningful grouping of subprograms, using the criteria of coupling and cohesion. In object-oriented design, the problem is subtly different: the task is to decide where to physically package the classes and objects from the design's logical structure, which are distinctly different from subprograms.

Experience indicates that there are several useful technical as well as non technical guidelines that can help to achieve an intelligent modularization of classes and objects. “The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently. Each module's structure should be simple enough that it can be understood fully; it should be possible to change the implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules; the case of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed” [55]. There is a pragmatic edge to these guidelines. In practice, the cost of recompiling the body of a module is relatively small: only that unit needs to be recompiled and the application to be relinked. However, the cost of recompiling the interface of a module is relatively high. Especially with strongly typed languages, one must recompile the module interface, its body, all other modules that depend upon this interface, the modules that depend upon these modules, and so on. Thus, for very large programs (assuming that our development environment does not support incremental compilation), a change in a single module interface might result in many minutes of recompilation. Obviously, such recompilations should not happen too frequently since they hinder productivity and for this reason, a module's interface should be as narrow as possible, yet still satisfy the needs of all using modules. A developer should aim to hide as much as possible in the implementation of a module; incrementally shifting declarations from a modules implementation to its interface is far less painful and destabilizing than ripping out extraneous interface code.

The developer must therefore balance two competing technical concerns, the desire to encapsulate abstractions, and the need to make certain abstractions visible to other modules. "System details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling module programs" [56]. In other words, a developer should aim to build modules that are cohesive by grouping logically related abstractions and loosely coupled by minimizing the dependencies among modules. From this perspective, modularity can be defined as the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Thus, the principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Two additional technical issues can affect modularization decisions. First, since modules usually serve as the elementary and indivisible units of software that can be reused across applications, a developer might choose to package classes and objects into modules in a way that makes their reuse convenient. Second, many compilers generate object code in segments, one for each module. Therefore, there may be practical limits on the size of individual modules. With regard to the dynamics of subprogram calls, the placement of declarations within modules can greatly affect the locality of reference and thus, the paging behavior of a virtual memory system. Poor locality happens when subprogram calls occur across segments and lead to cache misses and page thrashing that ultimately slow down the whole system.

Several competing non-technical needs may also affect modularization decisions. Typically, work assignments in a development team are given on a module-by-module basis, and so the boundaries of modules may be established to minimize the interfaces among different parts of the development organization. Senior designers are usually given responsibility for module interfaces and more junior developers complete their implementation. On a larger scale, the same situation applies with subcontractor relationships. Abstractions may be packaged so as to quickly stabilize the module interfaces agreed upon among the various companies. Changing such interfaces usually involves troublesome procedures among team members in order to update their code bases and, as a result, this often leads to conservatively designed interfaces. Modules also usually serve as the unit of documentation and configuration management. Security may also be an issue: most code may be considered unclassified, but other code that might be classified secret or higher is best placed in separate modules.

Taking into consideration all these different requirements is difficult but focus should be given to the most important point: finding the right classes and objects and then organizing them into separate modules are largely independent design decisions. The identification of classes and objects is part of the logical design of the system, but the identification of modules is part of the system's physical design. One cannot make all the logical design decisions before making all the physical ones, or vice versa; rather, these design decisions happen iteratively.

Design decision

During the design process of this code, a separation of the solver and mesh/preprocessor data structures was deemed necessary. The rationale of this design decision was the fact that a solver is merely a tool for the solution of a set of equations that define a certain problem formulation, providing a set of values that have a physical meaning that corresponds with the physical meaning of the mesh. If the equations derived from different problem formulations have similar mathematical properties, there is no need to program specific solvers for each formulation. The same solver can be used, provided that a translation mechanism can generate the appropriate set of equations given the problem formulation and the mesh

For the implementation of this design, separate classes for the solver and for the mesh were generated, following a modular paradigm. These classes are loosely coupled via a translator class, providing the functionality of solving different meshes and problem formulations using any of the programmed solvers. Due to this modular design, the production of different finite elements, problem formulations and solvers is completely independent from each other, providing the opportunity for separate developers to program new solvers and finite elements.

5.3.2 ABSTRACTION

Abstraction is one of the fundamental ways to cope with complexity. “Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" [57]. Abstraction is also "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary" [58].

Moreover, “a concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it” [59]. Combining these different viewpoints, an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide defined conceptual boundaries, relative to the perspective of the viewer.

An abstraction focuses on the outside view of an object, and so serves to separate an object's essential behavior from its implementation. This behavior/implementation division is called an abstraction barrier [60] achieved by applying the principle of least commitment, through which the interface of an object provides its essential behavior, and nothing more. An additional principle is also used which is called the principle of least astonishment, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction. Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented design. "There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence" [61]. From the most to the least useful, these kinds of abstractions include the following:

- Entity abstraction – An object that represents a useful model of a problem domain or solution-domain entity
- Action abstraction – An object that provides a generalized set of operations[1], all of which perform the same kind of function
- Virtual machine abstraction – An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
- Coincidental abstraction – An object that packages a set of operations that have no relation to each other

Developers should aim to build entity abstractions because they directly parallel the vocabulary of a given problem domain.

A client is any object that uses the resources of another object, known as the server. We can characterize the behavior of an object by considering the services that it provides to other objects, as well as the operations that it may perform upon other objects. This view forces us to concentrate upon the outside view of an object, and leads us to what the so-called contract model of programming [62] where the outside view of each object defines a contract upon which other objects may depend, and which in turn must be carried out by

[1] The terms operation, method, and member function evolved from three different programming cultures (Ada, Smalltalk, and C++, respectively). They all mean virtually the same thing and are used interchangeably in this work.

the inside view of the object itself, often in collaboration with other objects. This contract thus establishes all the assumptions a client object may make about the behavior of a server object. In other words, this contract encompasses the responsibilities of an object, namely, the behavior for which it is held accountable [63].

Individually, each operation that contributes to this contract has a unique signature comprising all of its formal arguments and return type. The entire set of operations that a client may perform upon an object, together with the legal orderings in which they may be invoked, is its protocol. A protocol denotes the ways in which an object may act and react, and thus constitutes the entire static: and dynamic outside view of the abstraction.

Central to the idea of an abstraction is the concept of invariance. An invariant is some logical condition whose truth must be preserved. For each operation associated with an object, preconditions which are invariants assumed by the operation, as well as post conditions which are invariants satisfied by the operation, are defined. Violating an invariant breaks the contract associated with an abstraction. If a precondition is violated, this means that a client has not satisfied its part of the bargain, and hence the server cannot proceed reliably. Similarly, if a post condition is violated, this means that a server has not carried out its part of the contract, and so its clients can no longer trust the behavior of the server. An exception is an indication that some invariant has not been or cannot be satisfied. Certain languages permit objects to throw exceptions so as to abandon processing and alert some other object to the problem, which in turn may catch the exception and handle the problem.

All abstractions have static as well as dynamic properties. For example, a file object takes up a certain amount of space on a particular memory device; it has a name, and it has contents. These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: a file object may grow or shrink in size, its name may change, its contents may change. In a procedure-oriented style of programming, the activity that changes the dynamic value of objects is the central part of all programs: things happen when subprograms are called and statements are executed. In a rule-oriented style of programming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on. In an object-oriented style of programming, things happen whenever we operate upon an object (in Smalltalk terminology, when we send a message to an object). Thus, invoking an operation upon an object elicits some reaction from the object. What operations we can meaningfully perform upon an object and how that object reacts constitute the entire behavior of the object.

Design decision

During the design process of this code, the implementation of different solvers was deemed necessary. These solvers had differences, not only on the underlying mathematics but on their implementation as well. Specifically, both single-domain and multi-domain solvers were implemented in order to investigate their efficiency, accuracy and scalability on a variety of physical problems of varying magnitude.

Due to the very different nature of these solvers, it would be impossible to use traditional programming techniques and be completely agnostic of the nature and internals of each solver. However, using abstraction, we were able to use any solver from the other code modules without knowing, either the identity or the specific implementation details of each solver.

5.3.3 ENCAPSULATION

Encapsulation enforces that "no part of a complex system should depend on the internal details of any other part" [64]. Whereas abstraction "helps people to think about what they-are doing, encapsulation allows program changes to be reliably made with limited effort" [65].

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the observable behavior of an object, whereas encapsulation focuses upon the implementation that gives rise to this behavior. Encapsulation is most often achieved through information hiding which is the process of hiding all properties of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the, implementation of its methods.

Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. For example, considering the structure of a finite element code, it is feasible to understand how a finite element works at a high level of abstraction, ignoring details such as the integration scheme or if the element is based on Cartesian or natural coordinates. In such cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

It is suggested that "for abstraction to work, implementations must be encapsulated" [66]. In practice, this means that each class must have two parts: an interface and an implementation. The interface of a class captures only its outside view, encompassing abstraction of the behavior common to all instances of the class. The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. The interface of a class is the one place where all of the assumptions that a client may make about any instances of the class are asserted; the implementation encapsulates details about which no client may make assumptions. To summarize, encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Design decision

During the design process of this code, solving FEM models comprised of different finite elements was deemed necessary. Specifically, a variety of continuum and engineering finite elements should be supported in order to assess the accuracy and the efficiency of these finite elements when solving various geotechnical and soil-structure interaction problems.

In order to support this feature, a generalized method of assembling the global and subdomain stiffness matrix was developed, utilizing the encapsulation feature of object oriented programming. This feature made the code extensible for the implementation of future finite elements without these extensions breaking the already existing code base. Moreover, the finite element libraries developed were programmed by different team members, boosting productivity.

5.3.4 HIERARCHY

While abstraction can prove to be a useful tool in OO programming, it is not sufficient for taming the complexity of large software systems. Encapsulation helps managing this complexity by hiding the inside view of abstractions while modularity provides a way to cluster logically related abstractions.

A set of abstractions often forms a hierarchy, and by identifying these hierarchies in a design, understanding of the problem is greatly simplified. Considering this property, hierarchy can be defined as a ranking or ordering of abstractions.

The most important hierarchies in a complex system are its class structure ("is a" hierarchy) and its object structure (the "part of' hierarchy). Inheritance is the most important "is a” hierarchy, and is an essential element of object systems. In its essence, inheritance defines a relationship among classes; one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses. Typically, a subclass augments or redefines the existing structure and behavior of its superclasses.

Semantically, inheritance denotes an "is-a" relationship. For example, a truss “is a" kind of finite element, a skyline matrix "is a" kind of symmetrical 2D matrix and PCG "is a” solution algorithm. Inheritance thus implies a generalization/specialization hierarchy, where a subclass specializes the more general structure or behavior of its superclasses. This property can serve as a criterion for inheritance; if B "is not a" kind of A, then B should not inherit from A.

As the inheritance hierarchy evolves, the structure and behavior that are common for different classes will tend to migrate to common superclasses. As such, inheritance can be considered as being a generalization/specialization hierarchy. Superclasses represent generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden. In this manner, inheritance allows to state abstractions with an economy of expression. Indeed, neglecting the "is a" hierarchies that exist can lead to bloated, inelegant designs. “Without inheritance, every class would be a free-standing unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he chooses. Any consistency across classes is the result of discipline on the part of the programmers. Inheritance makes it possible to define new software in the same way we introduce any concept to a newcomer, by comparing it with something that is already familiar" [67].

There is a healthy tension among the principles of abstraction, encapsulation, and hierarchy. "Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction" [68]. For a given class, there are usually two kinds of clients: objects that invoke operations upon instances of the class, and subclasses that inherit from the class. Therefore, with inheritance, encapsulation can be violated in one of three ways:

- The subclass might access an instance variable of its superclass,
- Call a private operation of its superclass, or
- Refer directly to superclasses of its superclass.

Different programming languages trade off support for encapsulation and inheritance in different ways, but among the languages referenced in this work, C++, C# and Java offer perhaps the greatest flexibility. Specifically, the interface of a class may have three parts: private parts, which declare members that are accessible only to the class itself, protected parts, which declare members that are accessible only to, the class and its subclasses, and public parts, which are accessible to all clients.

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of” hierarchies describe aggregation relationships. Aggregation is not a concept unique to object-oriented programming languages. Indeed, any language that supports record-like structures supports aggregation. However, the combination of inheritance with aggregation can prove to be powerful as aggregation permits the physical grouping of logically related structures and inheritance allows these common groups to be easily reused on different abstractions.

Aggregation raises the issue of ownership. The abstraction of a subdomain permits different finite elements to be used over time, but replacing a finite element in a subdomain does not change the identity of the subdomain as a whole, nor does removing a subdomain necessarily destroys all of its finite elements. In other words, the lifetime of a subdomain and its finite elements are independent.

Design decision

One of the differences between various finite elements is the evaluation of its stiffness matrix. Specifically, the mechanical behavior of a finite element as a part of a finite element model is affected by the aforementioned matrix. This matrix defines how this element is stressed when various loads and displacements are being imposed on the model.

Depending on the nature of each finite element (i.e. if it is two- or three-dimensional), there are certain processes that are common between families of finite elements. For such cases, utilizing the notion of hierarchy and inheritance was deemed appropriate in order to define a generic behavior for these elements while specializing specific behavior of each element during the development process of this code.

5.3.5 TYPING

The concept of a type derives primarily from the theories of abstract data types. "A type is a precise characterization of structural or behavioral properties which a collection of entities all share" [69]. In this work, the terms type and class will be used interchangeably. Although the concepts of a type and a class are similar, typing is considered as a separate element of the object model because the concept of a type places a very different emphasis upon the meaning of abstraction. Specifically, typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

Typing allows the expression of abstractions so that the programming language they are implemented can be made to enforce design decisions. This kind of enforcement is essential for programming-in-the-large [70].

The idea of conformance is central to the notion of typing. For example, consider units of measurement in engineering; when distance is divided by time, a value denoting speed is expected. In the same manner, multiplying temperature by a unit of force doesn't make sense, but multiplying mass by force does. These are both examples of strong typing, wherein the rules of a domain prescribe and enforce certain legal combinations of abstractions.

A given programming language may be strongly- typed, weakly typed, or even untyped, yet still be called object-oriented. For example, Eiffel is strongly-typed, meaning that type conformance is strictly enforced; operations cannot be called upon an object unless the exact signature of that operation is defined in the object's class or superclasses. In strongly typed languages, violation of type conformance can be detected at the time of compilation. Smalltalk, on the other hand, is an untyped language: a client can send any message to any class (although a class may not know how to respond to the message). Violations of type conformance may not be known until execution, and usually manifest themselves as execution errors. Languages such as C# or Java are hybrid: they have tendencies toward strong typing, but it is possible to ignore or suppress the typing rules.

Strong typing permits the usage of a programming language to enforce certain design decisions, therefore helps to regulate the complexity of an evolving software system. On the other hand, strong typing introduces semantic dependencies such that even small changes in the interface of a base class require recompilation of all subclasses.

There are a number of important benefits to be derived from using strongly typed languages [71]:

- Without type checking, a program in most languages can 'crash' in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.
- Most compilers can generate more efficient object code if types are declared.

Untyped languages offer greater flexibility, but "in almost all cases, the programmer in fact knows what sorts of objects are expected as the arguments of a message, and what sort of object will be returned" [72], even with untyped languages. In practice, the safety offered by strongly typed languages usually more than compensates for the flexibility lost by not using an untyped language, especially for programming-in-the large.

The concepts of strong typing and static typing are entirely different. Strong typing refers to type consistency, whereas static typing - also known as static binding or early binding - refers to the time when names are bound to types. Static binding means that the types all variables and expressions are fixed at the time of compilation; dynamic binding (also called late binding) means that the types of all variables and expressions are not known until runtime. Because strong typing and binding are independent concepts, a language may be both strongly and statically typed strongly typed yet support dynamic binding (Object Pascal and C++), or untyped yet support dynamic binding.

Dynamic binding provides a feature called polymorphism; it represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some common set of operations [73]. The opposite of polymorphism is monomorphism, which is found in all languages that are both strongly typed and statically bound, such as Ada.

Polymorphism exists when the features of inheritance and dynamic binding interact. It is perhaps the most powerful feature of object-oriented programming languages next to their support for abstraction, and it is what distinguishes object-oriented programming from more traditional programming with abstract data types.

5.4 APPLIED OBJECT ORIENTED PROGRAMMING

This section deals with the analysis of certain parts of the developed code where object-oriented programming notions were utilized, driving the overall design process. Certain issues that have been dealt with will be analyzed with respect to implementing a finite element code in a multi-processor environment using structured programming and how these issues are dealt with using object-oriented programming notions.

Specifically, the design decisions for issues like memory management, multi-processing and algebraic computations will be presented through specific applications.

5.4.1 THE SUBDOMAIN ENTITY

Designing the subdomain class was done through enumerating all of the necessary properties, methods and member functions that would define the behavior of this class inside the developed code. The subdomain class is quintessential with respect to defining geometrical and connectivity properties of a finite element mesh. This happens due to the fact that the subdomain class is vital for both single-domain and multi-domain (domain decomposition) solvers which are utilized in order to solve models in a multi-processor environment. When dealing with single-domain solvers, a single subdomain class contains all the necessary information needed to describe the model while for multi-domain solvers, a set of subdomain classes are instantiated, containing the model information of each subdomain along with their interconnection data.

The subdomain class properties include:

- Elements: Every subdomain class should have knowledge about the finite elements that it includes and how these are interconnected.
- Nodes: The nodes contained by the elements that comprise the subdomain are crucial in order to define subdomain boundaries, interconnection with other subdomains along with constraints and force or displacement loads.

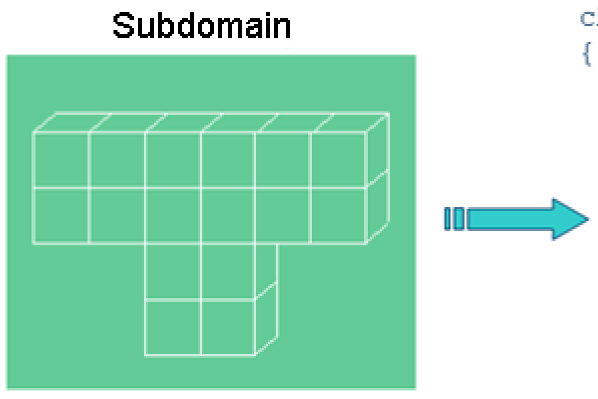
- Neighboring subdomains: Utilizing the above properties, each subdomain maintains a list of neighboring subdomains in order to provide information for the calculation of various domain-decomposition solvers (ie. lagrange multipliers)

All of the above subdomain class properties were defined by using the notion of hierarchy and more specifically, the notion of aggregation where the subdomain class aggregates instances of the appropriate related classes.

With the same rationale, a set of subdomain class methods includes:

- Stiffness matrix calculation: Each subdomain should be able to iterate through all of its elements and calculate its stiffness matrix
- Rigid body modes calculation: Should the subdomain be not adequately constrained, it should be able to calculate its rigid body modes which can then be utilized if necessary by the appropriate single- or multi-domain solvers.

A simplified interface of the above subdomain as utilized in the code developed is found below:



```
class Subdomain
{
  public:
  ...
  // Calculation functions
    void CalcRigidModes();
    void CalcK();
  ...
  // Physical properties
  private:
    Vector<int> neighborSubs;
    NodeSet itsNodes;
    ElementSet itsElementSet;
    Vector<double> forces;
  ...
};
```

Figure 5.4 - The subdomain class

5.4.2 ITERATIVE SOLVERS AND DOMAIN DECOMPOSITION

Domain decomposition solver implementation was the corner stone of the design process of this code. This was the case because:

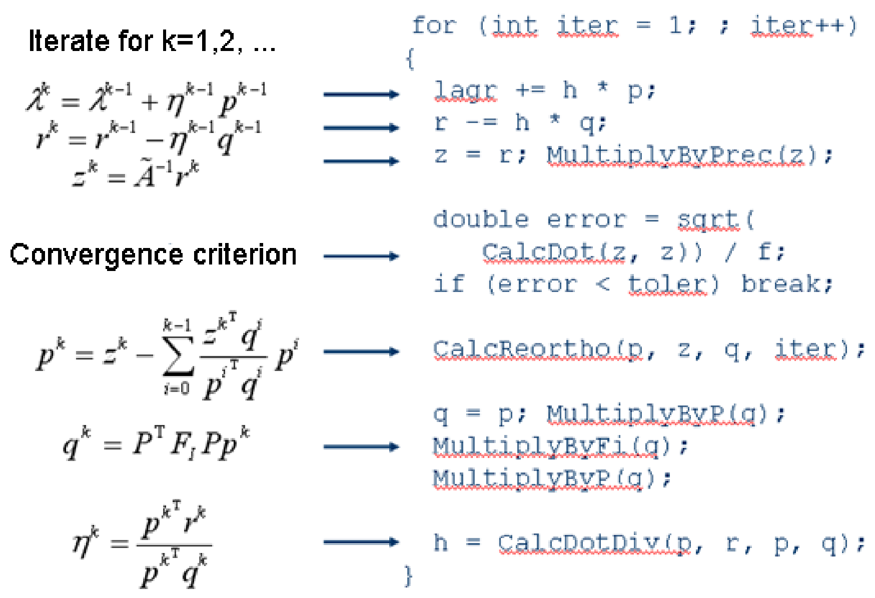
- Domain-decomposition solvers are well suited for large-scale structural engineering problems and inherently support parallel processing environments.

- They are complicated in implementation, especially for parallel processing environments since they involve inter-processor communication and synchronization which makes the implementation of the necessary subdomain and domain partitioning infrastructure necessary. Such an implementation can be used for various domain-decomposition solvers as well and without any modifications when designed appropriately.
- Their implementation requires the development of basic linear algebra building blocks that are useful for other less complicated solvers as well.

The formulation of the various DD methods and their underlying solvers was presented on Chapter 4. It is evident that these methods involve a lot of matrix and vector computations along with message passing between processors when executed on a distributed memory parallel processing environment. Implementing such solvers in structured programming environments can produce code that is less readable and succinct, making the data structures involved in each process to be almost cryptic for someone that has not been involved in authoring the code.

Such issues can prove to be very cumbersome for developers, with debugging being one of the most obvious problems. Unreadable "spaghetti" code can cause trouble for experienced programmers who can spend great amount of time in order to debug even common logical errors. One other important issue deals with code extensibility as unreadable code can prove to be very difficult to extend. This difficulty does not always have to do with the complexity of the task undertaken by the code but rather with the accidental complexity arising from its implementation details and the underlying mathematical or algebraic computations. It is evident that such properties can make code maintenance a very cost inefficient process since every attempt to modify the code can result in a series of debugging and extension processes that will prove to be cumbersome and time consuming.

The implementation of the PCPG iterations for the solution of a model using the FETI solver, along with a one-to-one correspondence with the theoretical algorithm is found below:



$$\lambda^k = \lambda^{k-1} + \eta^{k-1} p^{k-1}$$
$$r^k = r^{k-1} - \eta^{k-1} q^{k-1}$$
$$z^k = \tilde{A}^{-1} r^k$$

$$p^k = z^k - \sum_{i=0}^{k-1} \frac{z^{k^{\mathrm{T}}} q^i}{p^{i^{\mathrm{T}}} q^i} p^i$$
$$q^k = P^{\mathrm{T}} F_I P p^k$$
$$\eta^k = \frac{p^{k^{\mathrm{T}}} r^k}{p^{k^{\mathrm{T}}} q^k}$$

```
for (int iter = 1; ; iter++)
{
  lagr += h * p;
  r -= h * q;
  z = r; MultiplyByPrec(z);

  double error = sqrt(
     CalcDot(z, z)) / f;
  if (error < toler) break;

  CalcReortho(p, z, q, iter);

  q = p; MultiplyByP(q);
  MultiplyByFi(q);
  MultiplyByP(q);

  h = CalcDotDiv(p, r, p, q);
}
```

Figure 5.5 - The PCPG iteration loop

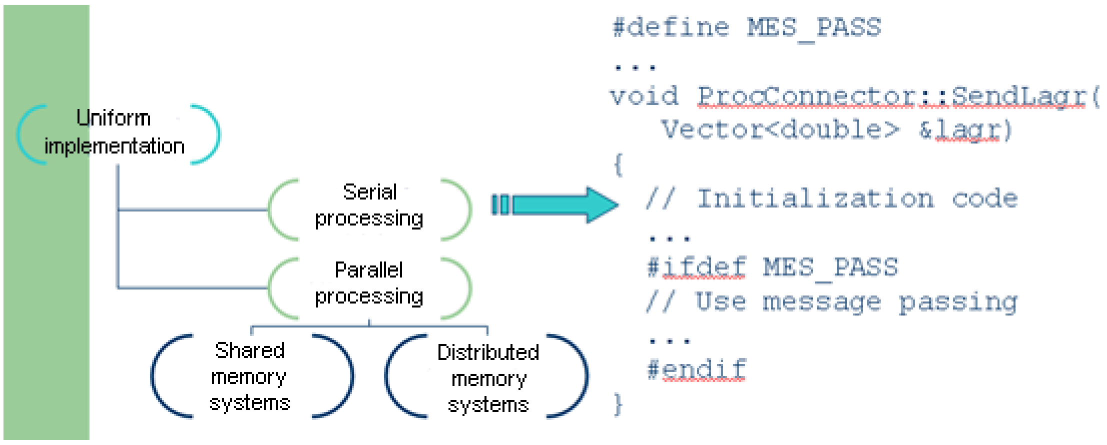
5.4.3 HOMOGENEOUS SINGLE- AND MULTI-CORE PROGRAMMING

In order to tackle the large-scale problems that were investigated in this work, it was deemed necessary for the code to run in parallel processing computing systems. Parallel processing is based on the simultaneous utilization of many processing units in order to evaluate a series of computations in less time. Such systems are divided into shared and distributed memory systems. Shared memory systems comprise many processors which have access to a common address space, i.e.: PCs based on multi-core processors or GPUs that have hundreds of streaming processors that have access to a global memory. On the other hand, distributed memory systems comprise processing units that have access to their own address space, i.e.: PCs interconnected via a LAN.

In order for processors inside a distributed memory system to have access to a memory space other than their own, interprocessor communication is taking place through message passing interfaces. Such interfaces are PVM (Parallel Virtual Machine), MPI (Message passing interface) or custom messaging systems, tailored for the specific algorithms that are executing in this parallel environment. The administration of the whole messaging system is the responsibility of the developer since this messaging process is strictly coupled with the algorithms that need to be executed.

On the other hand, due to the scoping semantics of a shared memory parallel system, the developer has just to make sure that each processor is fully loaded with computational work and to ensure that no race conditions occur during the concurrent execution of tasks. Specifically, a race condition occurs on a software system when the output of a certain method is dependent on the sequence or timing of other methods.

In order to deal with this duality, the concept of encapsulation was used where the implementation details for each processing system was "hidden" from the other classes of this code, constituting them agnostic to the underlying computer infrastructure. A special connector class was constructed, responsible for the transfer of information between nodes.



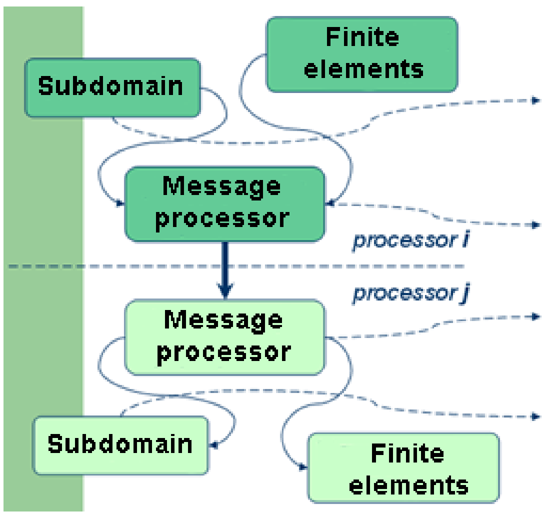
```
#define MES_PASS
...
void ProcConnector::SendLagr(
    Vector<double> &lagr)
{
  // Initialization code
  ...
  #ifdef MES_PASS
  // Use message passing
  ...
  #endif
}
```

Figure 5.6 - A unified message passing scheme

In order to ensure a transparent implementation with respect to the execution of this code in single- and multi-processing environments, a message administrator class was constructed which aims to aggregate information being sent from other objects in order to send it with a single message to the appropriate processor. This class provides better performance since message passing initialization overhead is being minimized and abstracts information locality from the other classes.

This abstraction is also extended between this class and the ProcConnector class. Specifically, the message administrator is agnostic to the nature of the parallel processing environment (shared memory or distributed); it just gathers all information to be shared among processors and lets ProcConnector to take care of the actual transmission, if it is deemed necessary.

A figure of several objects interacting with the message administration class in order to transmit information is found below:



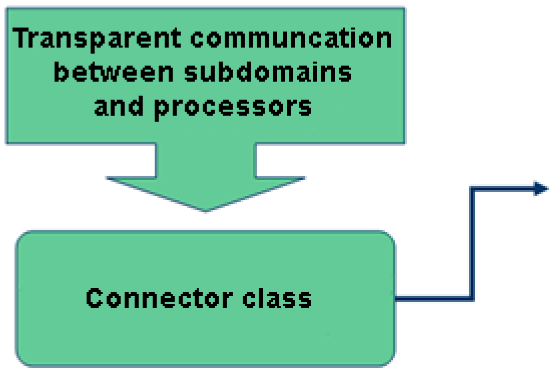
```
...
// Pack structural objects
someSubdomain.AddToBuf(
  someBuf);
...
// Send packed buffer
messageAdmin.SendBuf(someBuf);
...
// Receive packed buffer
messageAdmin.GetBuf(someBuf);
...
// Unpack structural objects
someSubdomain.GetFromBuf(
  someBuf);
...
```

Figure 5.7 - Exchanging information using the message processor class

Another example illustrating the transparent implementation with respect to the execution of this code in single- and multi-processing environments involves the evaluation of dot products on iterative solution algorithms. Specifically, in PCG and PCPG methods, dot products including information from all cores are being evaluated at each iteration. This evaluation occurs as follows:

- Partial dot products involving information contained in the memory domain of each core are being evaluated in parallel.
- Messages are being exchanged between cores in order to evaluate partial dot products that involve information contained in more than one cores.
- All the above partial dot products are being summed in parallel, following a tree-like pattern in order to evaluate the full dot product.

This process is shown below:



```
...
// Calculation of dot product
someSolver.CalcDot(p, q);
...

...
// Implement dot product
dpcProducts.CalcResult(v1,
  v2);
dpcProducts.SendReceive();
...
```

Figure 5.8 - Dot products

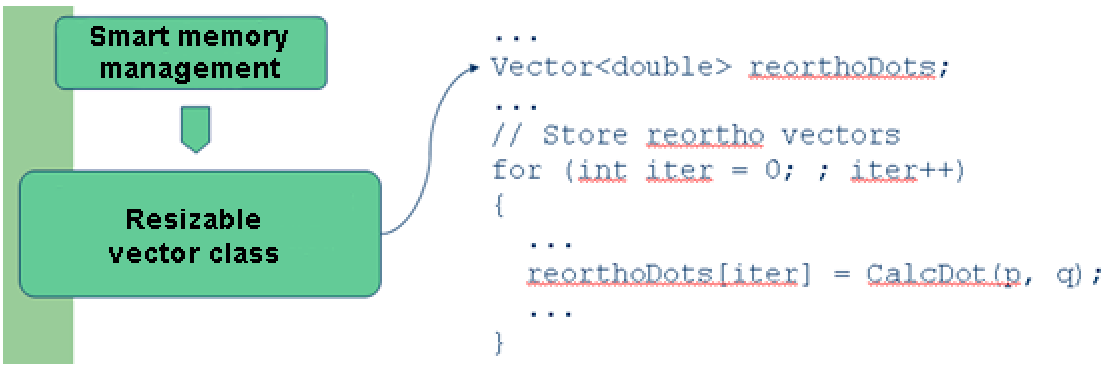
5.4.4 THE VECTOR AND ARRAY CLASSES

The array and vector classes are the foundation of the code developed in this work since all data structures used in the solver classes are based on them. Moreover, they are the main data structures containing information that occupy large memory areas and are subject to communication between cores.

Referring to FORTRAN which is the language of choice for people with engineering background, memory management is very cumbersome. FORTRAN features dynamic memory allocation at its later versions and, for reasons of portability, FORTRAN developers are reluctant to its usage. However, even with the usage of such a feature, it is very difficult to design a transparent system architecture with respect to memory allocation issues.

FORTRAN legacy code usually features the declaration of a very large vector which is then subdivided to sections with the usage of indices. These indices were used in order to hold a reference as to where information is stored. Such an implementation can prove problematic since a vast amount of memory is allocated which may not be used completely or may not be sufficient. In the latter case, the program will crash at runtime.

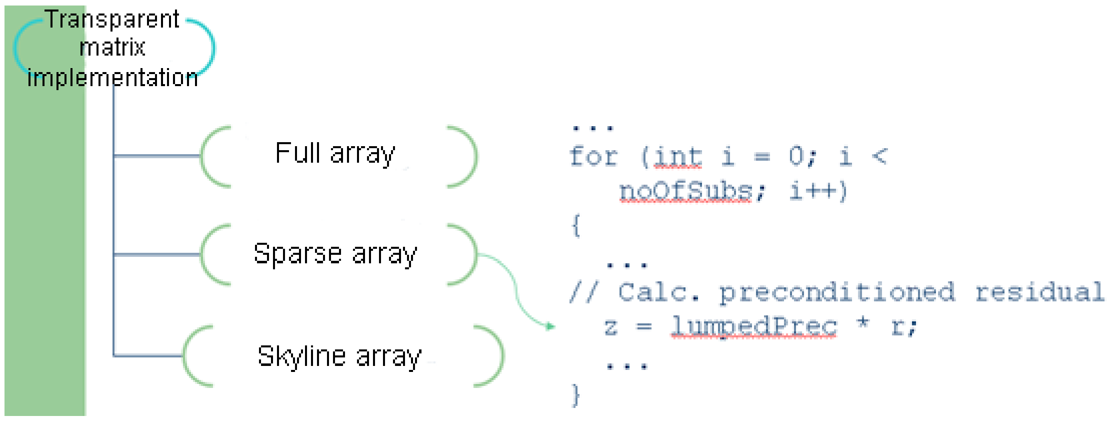
In the code developed in this work, a vector class capable of resizing itself is implemented and is used throughout the code as shown below:



```
...
Vector<double> reorthoDots;
...
// Store reortho vectors
for (int iter = 0; ; iter++)
{
   ...
   reorthoDots[iter] = CalcDot(p, q);
   ...
}
```

Figure 5.9 - The resizable vector class

During the implementation of finite element solver algorithms, coefficient matrices might be stored using various storage techniques, besides storing them element-by-element. Since finite element matrices usually are symmetric and have a lot of zero elements, sparse storage or skyline storage schemes may prove to be more memory efficient. Despite their different storage scheme, these matrices still retain their properties with respect to operators like addition, multiplication, etc. The implementation of these operators should be done in a way that is agnostic to the storage scheme, as seen below.



```
...
for (int i = 0; i <
    noOfSubs; i++)
{
   ...
// Calc. preconditioned residual
   z = lumpedPrec * r;
   ...
}
```

Figure 5.10 - The matrix multiplication operator

6 PARALLEL PROGRAMMING

6.1 SERIAL HARDWARE

In order to solve large-scale problems in Computational Mechanics, it is necessary to develop computer programs that can utilize parallel computing environments. However, in order to write efficient parallel programs, knowledge of the underlying hardware and system software is needed. Moreover, it is very useful to have some knowledge of different types of parallel software and application programming interfaces (APIs), so in this chapter a brief overview of various topics in hardware and software will be presented along with a methodology for developing parallel programs.

Parallel hardware and software have grown out of conventional serial hardware and software, that is hardware and software that runs (more or less) a single job at a time. So in order to better understand the current state of parallel systems, some aspects of serial systems will be presented.

6.1.1 THE VON NEUMANN ARCHITECTURE

The classical von Neumann architecture consists of main memory, a central processing unit (CPU) or processor or core, and an interconnection between the memory and the CPU. Main memory consists of a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location and the contents of the location. These contents can be either instructions or raw data.

The central processing unit is divided into a control unit and an arithmetic and logic unit (ALU). The control unit is responsible for deciding which instructions in a program should be executed, and the ALU is responsible for executing the actual instructions. Data in the CPU and information about the state of an executing program are stored in special, very fast storage called registers. The control unit has a special register called the program counter, where the address of the next instruction to be executed is stored.

Instructions and data are transferred between the CPU and memory via the interconnect. This has traditionally been a bus, which consists of a collection of parallel wires and some hardware controlling access to the wires. A von Neumann machine executes a single instruction at a time, and each instruction operates on only a few pieces of data as seen in Figure 6.1.

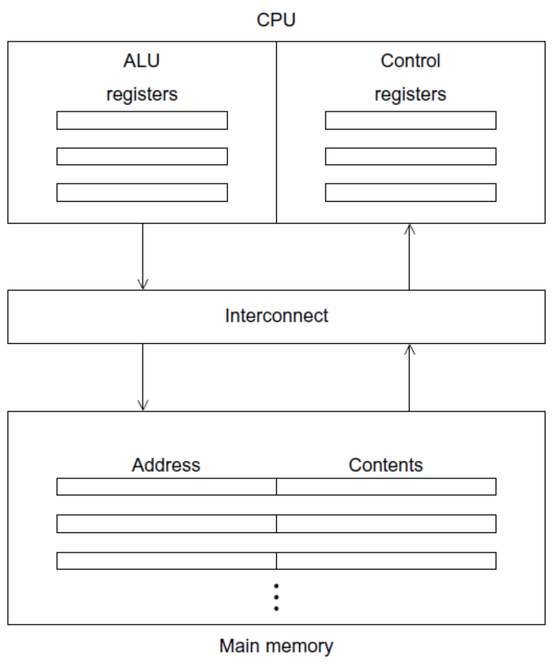


Figure 6.1 - The von Neumann architecture

Fetching or reading from memory occurs when data or instructions are transferred from memory to the CPU, while data are written to memory or stored when data are transferred from the CPU to memory. The separation of memory and CPU is the so-called von Neumann bottleneck, since the interconnect determines the rate at which instructions and data can be accessed. The potentially vast quantity of data and instructions needed to run a program is effectively isolated from the CPU. Contemporary CPUs are capable of executing instructions more than one hundred times faster than they can fetch items from main memory.

In order to better understand this process, let's assume that a large company has a single factory (the CPU) in one town and a single warehouse (main memory) in another with a single two-lane road joining the warehouse and the factory. All the raw materials used in manufacturing the products are stored in the warehouse. Also, all the finished products are stored in the warehouse before being shipped to customers. If the rate at which products can be manufactured is much larger than the rate at which raw materials and finished products can be transported, then it is likely that there will be a huge traffic jam on the road, and the employees and machinery in the factory will either be idle for extended periods or they will have to reduce the rate at which they produce finished products.

In order to address the von Neumann bottleneck and improve CPU performance, computer engineers and computer scientists have experimented with many extensions and modifications to the basic von Neumann architecture. Some of these modifications will be presented in the next section.

6.1.2 PROCESSES, MULTI-TASKING AND THREADS

The operating system (OS) is a major piece of software with the purpose to manage hardware and software resources on a computer. It determines which programs can run and when they can run and controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.

When a user runs a program, the operating system creates a process which is an instance of a computer program that is being executed. A process consists of the following entities:

- The executable machine language program.
- A block of memory which will include:
  - the executable code, a call stack that keeps track of active functions
  - a heap and
  - some other memory locations.
- Descriptors of resources that the operating system has allocated to the process, like file descriptors or canvas descriptors.
- Security information, i.e.: information specifying which hardware and software resources the process can access.
- Information about the state of the process such as whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process' memory.

Most modern operating systems are multitasking which means that the operating system provides support for the apparent simultaneous execution of multiple programs. This is possible even on a system with a single core where each process runs for a small interval of time (typically a few milliseconds), often called a time slice. After one running program has executed for a time slice, the operating system can run a different program. A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time.

In a multitasking OS, if a process needs to wait for a resource i.e.: it needs to read data from external storage, it will block. This means that it will stop executing and the operating system can run another process. However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user.

Threading provides a mechanism for dividing programs into independent tasks so that, when one thread is blocked another thread can be run. Furthermore, switching between threads is much faster compared to switching between processes. This is because threads are more light weight than processes. Threads are contained within processes and they can use the same executable while sharing the same memory and the same I/O devices. In fact, two threads belonging to one process can share all of the process' resources with the exception of needing a record of their own program counter and their own call stacks, so that they can execute independently of each other.

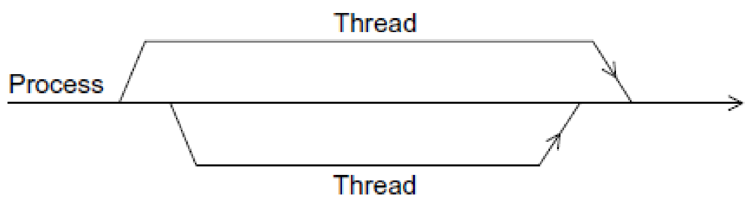


Figure 6.2 - A process and two threads

If a process is the "master" thread of execution and threads are started and stopped by the process, it is possible to form the analogy of the process and its subsidiary threads as lines. When a thread is started, it forks off the process and when a thread terminates, it joins the process as shown in Figure 6.2.

6.1.3 CACHING

Caching is one of the most widely used methods of addressing the von Neumann bottleneck. To understand the ideas behind caching, recall our example. A company has a factory (CPU) in one town and a warehouse (main memory) in another, and there is a single, two-lane road joining the factory and the warehouse. There are a number of possible solutions to the problem of transporting raw materials and finished products between the warehouse and the factory. One is to widen the road. Another is to move the factory and/or the warehouse or to build a unified factory and warehouse. Caching exploits both of these ideas. Rather than transporting a single instruction or data item, an effectively wider interconnection is used which can transport more data or more instructions in a single memory access. Moreover, rather than storing all data and instructions exclusively in main memory, blocks of data and instructions are stored in a special memory region that is effectively closer to the registers in the CPU.

In general a cache is a collection of memory locations that can be accessed in less time than some other memory locations. In this work, references to caches have the meaning of CPU cache, which is a collection of memory locations that the CPU can access more quickly than it can access main memory. A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Once a cache is available, an obvious problem is to decide which data and instructions should be stored in this cache. The universally used principle is based on the idea that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs typically execute the next instruction with branching tending to be relatively rare. Similarly, after a program has accessed one memory location, it often accesses a memory location that is physically nearby. An extreme example of this is in the use of arrays. Consider the loop:

```
float z[1000];

. . .

sum = 0.0;

for (i = 0; i < 1000; i++)

sum += z[i];
```

Arrays are allocated as blocks of contiguous memory locations which means that the location storing z[1] immediately follows the location z[0]. Thus, as long as i < 999, the read of z[i] is immediately followed by a read of z[i+1].

The principle that an access of one location is followed by an access of a nearby location is called locality. After accessing one memory location (instruction or data), a program will typically access a nearby location (spatial locality) in the near future (temporal locality).

In order to exploit the principle of locality, the system uses an effectively wider interconnect to access data and instructions where a memory access will effectively operate on blocks of data and instructions instead of individual instructions and individual data items. These blocks are called cache blocks or cache lines. A typical cache line stores 8 to 16 times as much information as a single memory location. In our example, if a cache line stores 16 floats, then when we first go to add sum += z[0], the system might read the first 16 elements of z, z[0], z[1], . . . , z[15] from memory into cache. As a result, next 15 additions will use elements of z that are already in the cache.

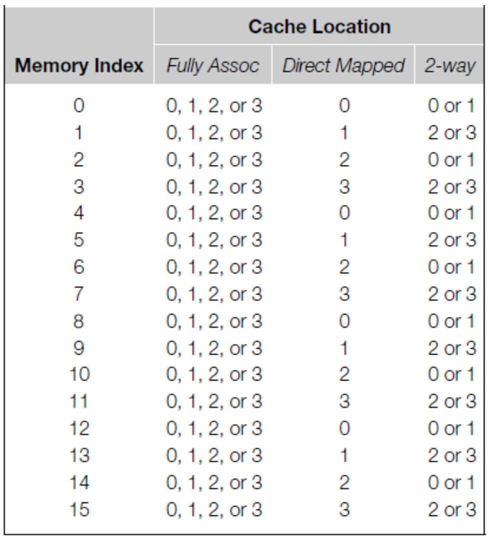
Conceptually, it is often convenient to think of a CPU cache as a single monolithic structure. However, in practice, the cache is usually divided into levels: the first level (L1) is the smallest and the fastest, and higher levels (L2, L3, . . .) are larger and slower. Most contemporary computer systems, have at least two levels while having three levels is quite common. Caches usually store copies of information in slower memory, and, if we think of a lower-level (faster, smaller) cache as a cache for a higher level, this usually applies. So, for example, a variable stored in a level 1 cache will also be stored in level 2. However, some multilevel caches do not duplicate information that is available in another level. For these caches, a variable in a level 1 cache might not be stored in any other level of the cache, but it would be stored in main memory.

When the CPU needs to access an instruction or data, it works its way down the cache hierarchy: First it checks the level 1 cache, then the level 2, and so on. Finally, if the information needed is not in any of the caches, it accesses the main memory. When a cache is checked for information and the information is available, it is called a cache hit or just a hit. If the information is unavailable, it is called a cache miss or a miss. Hit or miss is often modified by the level. For example, when the CPU attempts to access a variable, it might have an L1 miss and an L2 hit.

When the CPU attempts to read data or instructions and there is a cache read miss, it will read from memory the cache line that contains the needed information and store it in the cache. This may stall the processor, while it waits for the slower memory. The processor may stop executing statements from the current program until the required data or instructions have been fetched from memory. Considering the previous loop, when z[0] is read, the processor may stall while the cache line containing z[0] is transferred from memory into the cache.

When the CPU writes data to a cache, the value in the cache and the value in main memory are different or inconsistent. There are two basic approaches to dealing with this inconsistency. In write-through caches, the line is written to main memory when it is written to the cache. In write-back caches, the data isn't written immediately. Rather, the updated data in the cache is marked dirty, and when the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Another issue in cache design is deciding where lines should be stored. This decision varies from system to system with one extreme being a fully associative cache, in which a new line can be placed at any location in the cache and the other extreme being a direct mapped cache, in which each cache line has a unique location in the cache to which it will be assigned. Intermediate schemes are called n-way set associative where each cache line can be placed in one of n different locations in the cache, i.e.: in a two way set associative cache, each line can be mapped to one of two locations.



	Cache Location		
Memory Index	*Fully Assoc*	*Direct Mapped*	*2-way*
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 6.1 - Assignments of a 16-line main memory to a 4-line cache

As an example, suppose that the main memory consists of 16 lines with indexes 0–15, and the cache consists of 4 lines with indexes 0–3. In a fully associative cache, line 0 can be assigned to cache location 0, 1, 2, or 3. In a direct mapped cache, lines might be assigned by looking at their remainder after division by 4. This means that lines 0, 4, 8, and 12 would be mapped to cache index 0, lines 1, 5, 9, and 13 would be mapped to cache index 1, and so on. In a two way set associative cache, the cache can be grouped into two sets, indexes 0 and 1 form one set—set 0—and indexes 2 and 3 form another— set 1. This way, the remainder of the main memory index modulo 2 can be used, and cache line 0 would be mapped to either cache index 0 or cache index 1 as in Table 6.1.

When more than one line in memory can be mapped to several different locations in a cache (fully associative and n-way set associative), the line that should be replaced or evicted must also be known. In the previous example, if for instance, line 0 is in location 0 and line 2 is in location 1, the storage location of line 4 must be decided. The most commonly used scheme for taking this decision is called least recently used where the cache has a record of the relative order in which the blocks have been used, and if line 0 were used more recently than line 2, then line 2 would be evicted and replaced by line 4.

A caching example

The CPU cache usage and administration is controlled by the system hardware and an application cannot directly determine which data and which instructions are in the cache. However, knowing the principle of spatial and temporal locality allows an indirect control over caching. As an example, C stores two-dimensional arrays in “row-major” order. This means that, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. In row-major storage, we row 0 is stored first, then row 1, and so on. In the following two code segments, we the first pair of nested loops is expected to have much better performance than the second, since it is accessing the data in the two-dimensional array in contiguous blocks.

```
double A[MAX][MAX], x[MAX], y[MAX];

. . .

/* Initialize A and x, assign y = 0 */

. . .

/* First pair of loops */

for (i = 0; i < MAX; i++)

for (j = 0; j < MAX; j++)

y[i] += A[i][j]————————————————————————x[j];

. . .

/* Assign y = 0 */

. . .

/* Second pair of loops */

for (j = 0; j < MAX; j++)

for (i = 0; i < MAX; i++)

y[i] += A[i][j] * x[j];
```

To better understand this, suppose MAX is four, and the elements of A are stored in memory as follows:

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

So, for example, A[0][1] is stored immediately after A[0][0] and A[1][0] is stored immediately after A[0][3]. Let's suppose that none of the elements of A are in the cache when each pair of loops starts executing and that a cache line consists of four elements of A, with A[0][0] being the first element of a cache line. Finally, let's suppose that the cache is direct mapped and it can only store eight elements of A, or two cache lines.

Both pairs of loops attempt to first access A[0][0]. Since it is not in the cache, this will result in a cache miss, and the system will read the line consisting of the first row of A, A[0][0], A[0][1], A[0][2], A[0][3], into the cache. The first pair of loops then accesses A[0][1], A[0][2], A[0][3], all of which are in the cache, and the next miss in the first pair of loops will occur when the code accesses A[1][0]. Continuing in this fashion, we see that the first pair of loops will result in a total of four misses when it accesses elements of A, one for each row. Note that since this cache can only store two lines or eight elements of A, one of the lines already in the cache will have to be evicted, when the first element of row two and the first element of row three are read. However, once a line is evicted, the first pair of loops won't need to access the elements of that line again.

After reading the first row into the cache, the second pair of loops then needs to access A[1][0], A[2][0], A[3][0], none of which are in the cache. As a result, the next three accesses of A will also result in misses. Furthermore, because the cache is small, the reads of A[2][0] and A[3][0] will require lines already in the cache to be evicted. Since A[2][0] is stored in cache line 2, reading its line will evict line 0, and reading A[3][0] will evict line 1. After finishing the first pass through the outer loop, access to A[0][1] is needed which was evicted with the rest of the first row. It is evident that every time an element of A is read, a cache miss occurs with the second pair of loops resulting in 16 misses.

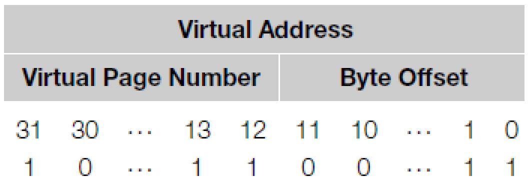
Based on the above remarks, the first pair of nested loops is expected to be much faster than the second. In fact, if the code is executed on a contemporary computer, with MAX = 1000, the first pair of nested loops is approximately three times faster than the second pair.

6.1.4 VIRTUAL MEMORY

Caches make it possible for the CPU to quickly access instructions and data that are in main memory. However, if we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory. This is especially true with multitasking operating systems; in order to switch between programs and create the illusion that multiple programs are running simultaneously, the instructions and data that will be used during the next time slice should be in main memory. Thus, in a multitasking system, even if the main memory is very large, many running programs must share the available main memory. Furthermore, this sharing must be done in such a way that each program's data and instructions are protected from corruption by other programs.

Virtual memory was developed so that main memory can function as a cache for secondary storage. It exploits the principle of spatial and temporal locality by keeping in main memory only the active parts of the many running programs; those parts that are idle are kept in a block of secondary storage called swap space. Like CPU caches, virtual memory operates on blocks of data and instructions. These blocks are commonly called pages, and since secondary storage access can be hundreds of thousands of times slower than main memory access, pages are relatively large—most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.

Trying to assign physical memory addresses to pages during the compilation of a program can prove to be problematic because with such a correspondence, each page of the program can only be assigned to one block of memory and, with a multitasking operating system, it is very likely that many programs will request to use the same block of memory. In order to circumvent this problem, when a program is compiled, its pages are assigned to virtual page numbers. When the program is executed, a table is created that maps the virtual page numbers to physical addresses and, when the program refers to a virtual address, this page table is used to translate the virtual address into a physical address. If the creation of the page table is managed by the operating system, it can ensure that the memory used by one program doesn't overlap the memory used by another.



Virtual Address									
Virtual Page Number					Byte Offset				
31	30	⋯	13	12	11	10	⋯	1	0
1	0	⋯	1	1	0	0	⋯	1	1

Table 6.2 - Virtual address divided into virtual page number and byte offset

One major drawback to the use of a page table is that it can double the time needed to access a location in main memory. Let's assume that an instruction in main memory is waiting for execution. The executing program will have the virtual address of this instruction but before finding the instruction in memory, the virtual address needs to be translated into an actual physical address. In order to do this, the page in memory that contains the instruction must be found. Now the virtual page number is stored as a part of the virtual address. As an example, suppose our addresses are 32 bits and our pages are 4 kilobytes = 4096 bytes. Each byte in the page can be identified with 12 bits, since 2^{12} = 4096. Thus, we can use the low order 12 bits of the virtual address to locate a byte within a page, and the remaining bits of the virtual address can be used to locate an individual page as in Table 6.2.

The virtual page number can be computed from the virtual address without going to memory. However, once the virtual page number is pinpointed, access to the page table is needed for the translation into a physical page. If the required part of the page table is not in cache, it needs to be loaded from memory. After that, virtual address can be translated to a physical address in order to read the required instruction.

Although multiple programs can use main memory at more or less the same time, using a page table has the potential to significantly increase each program's overall run time. In order to address this issue, processors have a special address translation cache called a translation-lookaside buffer (TLB) which caches a small number of entries (typically 16–512) from the page table in very fast memory. Using the principle of spatial and temporal locality, most of our memory references are expected to be in pages whose physical address is stored in the TLB, and the number of memory references that require accesses to the page table in main memory will be substantially reduced.

The terminology for the TLB has a direct correspondence with the terminology for caches; looking for an address whose virtual page number is in the TLB is a TLB hit while in the opposite case it is a miss. On the other hand, the terminology for the page table has an important difference from the terminology for caches. Attempting to access a page that's not in memory, that is, it does not have a valid physical address and the page is only stored on disk, is called a page fault.

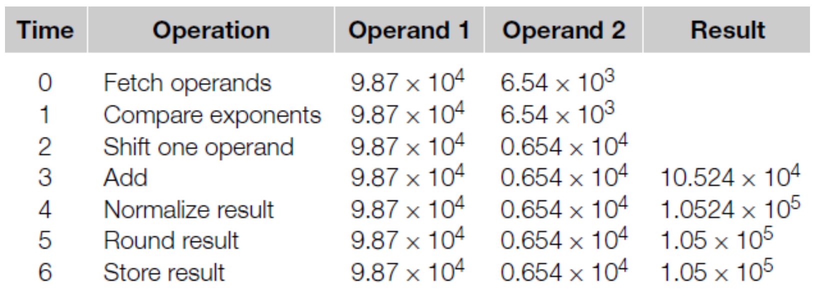
The relative slowness of disk accesses has a couple of additional consequences for virtual memory. First, with CPU caches write-misses can be handled with either a write-through or write-back scheme. With virtual memory, however, disk accesses are so expensive that they should be avoided whenever possible, so virtual memory always uses a write-back scheme. This can be handled by keeping a bit on each page in memory that indicates whether the page has been updated. If it has been updated, when it is evicted from main memory, it will be written to disk. Moreover, since disk accesses are so slow, management of the page table and the handling of disk accesses can be done by the operating system. As a result, even though an application cannot directly control virtual memory, unlike CPU caches which are handled by system hardware, virtual memory is usually controlled by a combination of system hardware and operating system software.

6.1.5 INSTRUCTION LEVEL PARALLELISM

Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions. There are two main approaches to ILP: pipelining, in which functional units are arranged in stages, and multiple issue, in which multiple instructions can be simultaneously initiated. Both approaches are used in virtually all modern CPUs.

Pipelining

The principle of pipelining is similar to a factory assembly line: while one team is bolting a car's engine to the chassis, another team can connect the transmission to the engine and the driveshaft of a car that is already been processed by the first team, and a third team can bolt the body to the chassis in a car that has been processed by the first two teams. As an example involving computation, the steps for adding the floating point numbers $9.87*10^4$ and $6.54*10^3$ are shown below:



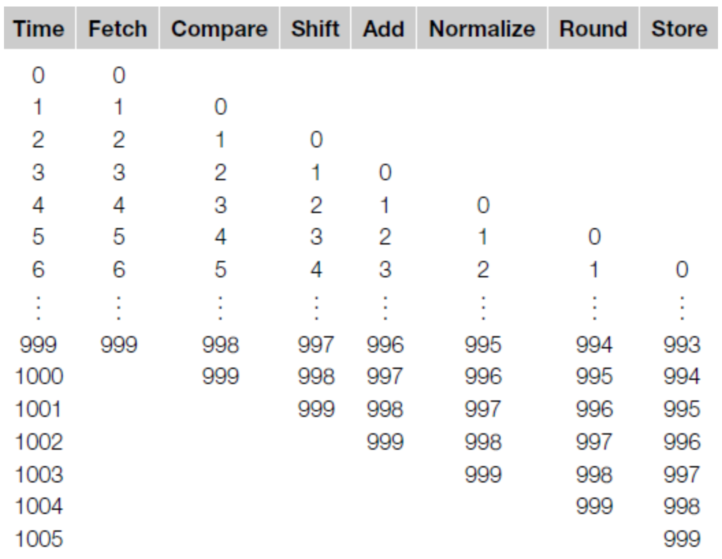
Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

In this example, base 10 is used along with a three digit mantissa or significand with one digit to the left of the decimal point. Normalizing shifts the decimal point one unit to the left and rounding rounds to three digits. If each of the operations takes one nanosecond, the addition operation will take seven nanoseconds. The execution of the loop in the following code:

```
float x[1000], y[1000], z[1000];

...

for (i = 0; i < 1000; i++)

z[i] = x[i] + y[i];
```

will take something like 7000 nanoseconds.

As an alternative, assume that the floating point adder is divided into seven separate pieces of hardware or functional units with the first unit fetching two operands, the second comparing exponents, and so on. Moreover, assume that the output of one functional unit is the input to the next, so, for example, the output of the functional unit that adds the two values is the input to the unit that normalizes the result. For this case, a single floating point addition will also take seven nanoseconds. However, when the loop is executed, x[1] and y[1] can be fetched while the exponents of x[0] and y[0] are being compared. Using this notion, it is possible to simultaneously execute seven different stages in seven different additions, as shown in Table 6.3. From this table, it is evident that after time 5, the pipelined loop produces a result every nanosecond instead of every seven nanoseconds which brings the total time to execute the for loop from 7000 nanoseconds down to 1006 nanoseconds, an improvement of almost a factor of seven.



Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Table 6.3 - Pipelined Addition: numbers in the table are subscripts of operands/results

In general, a pipeline with k stages will not get a k-fold improvement in performance. For example, if the times required by the various functional units are different, then the stages will effectively run at the speed of the slowest functional unit. Furthermore, delays such as waiting for an operand to become available can cause the pipeline to stall.

Multiple issue

Pipelines improve performance by taking individual pieces of hardware or functional units and connecting them in sequence. Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program. For example, if two complete floating point adders are available, the time it takes to execute the loop:

```
for (i = 0; i < 1000; i++)

z[i] = x[i] + y[i];
```

can be reduced to half.

While the first adder is computing z[0], the second can compute z[1]; while the first is computing z[2], the second can compute z[3]; and so on. If the functional units are scheduled at compile time, the multiple issue system is said to use static multiple issue. If they're scheduled at run-time, the system is said to use dynamic multiple issue. A processor that supports dynamic multiple issue is called superscalar.

In order to make use of multiple issue, the system must find instructions that can be executed simultaneously. One of the most important techniques to find such instructions is called speculation. In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess. As a simple example, consider the following code:

```
z = x + y;

if (z > 0)

w = x;

else

w = y;
```

The system might predict that the outcome of z = x + y will give z a positive value, and, as a consequence, it will assign w = x. As another example, in the following code:

```
z = x + y;

w = *a p; /* a p is a pointer */
```

the system might predict that p does not refer to z, therefore it can simultaneously execute the two assignments. As both examples make clear, speculative execution must allow for the possibility that the predicted behavior is incorrect. In the first example, we will need to go back and execute the assignment w = y if the assignment z = x + y results in a value that's not positive. In the second example, if a p does point to z, we'll need to reexecute the assignment w = *a p.

If the compiler does the speculation, it will usually insert code that tests whether the speculation was correct, and, if not, takes corrective action. If the hardware does the speculation, the processor usually stores the result(s) of the speculative execution in a buffer. When it is known that the speculation was correct, the contents of the buffer are transferred to registers or memory. If the speculation was incorrect, the contents of the buffer are discarded and the instruction is re-executed.

While dynamic multiple issue systems can execute instructions out of order, in current generation systems the instructions are still loaded in order and the results of the instructions are also committed in order, which means that the results of instructions are written to registers and memory in the program-specified order. Optimizing compilers, on the other hand, can reorder instructions which can have important consequences for shared-memory programming.

6.1.6 HARDWARE MULTITHREADING

ILP can be very difficult to exploit for programs with a long sequence of dependent statements. For example, in a direct calculation of the Fibonacci numbers shown below:

```
f[0] = f[1] = 1;

for (i = 2; i <= n; i++)

f[i] = f[i-1] + f[i-2];
```

there is essentially no opportunity for simultaneous execution of instructions. Thread-level parallelism (TLP) attempts to provide parallelism through the simultaneous execution of different threads, providing a coarser-grained parallelism than ILP. The program units that are being simultaneously executed (threads) are larger or coarser than the finer-grained units (individual instructions).

Hardware multithreading provides a means for systems to continue doing computational work when the task being currently executed has stalled, i.e.: if the current task has to wait for data to be loaded from memory. Instead of looking for parallelism in the currently executing thread, it is more efficient to simply run another thread. In order for this to be efficient, the system must support very rapid switching between threads, which is the case for contemporary OS.

In fine-grained multithreading, the processor switches between threads after each instruction, skipping threads that are stalled. While this approach has the potential to avoid wasted machine time due to stalls, it has the drawback that a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction. Coarse-grained multithreading attempts to avoid this problem by only switching threads that are stalled waiting for a time-consuming operation to complete, i.e.: a load from main memory. Such a strategy permits switching threads not to be instantaneous. However, the processor may be idle on shorter stalls with thread switching also causing delays.

Simultaneous multithreading (SMT) is a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units. By designating "preferred" threads which are threads that have many instructions ready to execute, thread slowdown can be greatly reduced.

6.2 PARALLEL HARDWARE

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. However, since this form of parallelism is not usually visible to the programmer, they are both treated as extensions to the basic von Neumann model. In the following sections, a series of parallel hardware that are exploitable by the programmer are defined.

6.2.1 SIMD SYSTEMS

In parallel computing, Flynn's taxonomy [74] is frequently used to classify computer architectures according to the number of instruction streams and the number of data streams it can simultaneously manage. A classical von Neumann system is therefore a single instruction stream, single data stream (SISD) system, since it executes a single instruction at a time and it can fetch or store one item of data at a time.

Single instruction, multiple data, or SIMD, systems are parallel systems which operate on multiple data streams by applying the same instruction to multiple data items. Thus, an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle. As an example, suppose that a "vector addition" is being executed where we have two arrays x and y, each with n elements, and the elements of y are added to the elements of x, as show below:

```
for (i = 0; i < n; i++)

x[i] += y[i];
```

Assuming that our SIMD system has n ALUs, x[i] and y[i] could be loaded into the ith ALU which could add y[i] to x[i] and store the result in x[i]. If the system has m ALUs and m < n, the additions can be executed in blocks of m elements at a time, i.e.: if m=4 and n=15, elements 0 to 3 can be added first, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in the example, only three elements of x and y need to be added, so one of the four ALUs will be idle.

The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system. For example, suppose that the previous addition must be evaluated only if y[i] is positive, as shown below:

```
for (i = 0; i < n; i++)

if (y[i] > 0.0) x[i] += y[i];
```

In this case, each element of y must be loaded into an ALU and determine whether it is positive. If y[i] is positive, the addition is evaluated, otherwise, the ALU storing y[i] will be idle while the other ALUs carry out the addition. Note also that in a “classical” SIMD system, the ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding. Moreover, the ALUs have no instruction storage, so an ALU cannot delay execution of an instruction by storing it for later execution.

Taking these examples under consideration, it is evident that SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. Parallelism that is obtained by dividing data among the processors and having the processors all apply the same instructions to their subsets of the data, is called data-parallelism. SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often do not perform very well on other types of parallel problems. In contemporary computer systems, graphics processing units (GPUs) and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

Vector processors can operate on arrays or vectors of data, while conventional CPUs operate on individual data elements or scalars. Contemporary systems have the following characteristics:

- Vector registers. These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.
- Vectorized and pipelined functional units where the same operation is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors.
- Vector instructions. These are instructions that operate on vectors rather than scalars. If the vector length is vector length, these instructions have the great virtue that a simple loop such as the one shown below:
  ```
  for (i = 0; i < n; i++)
  x[i] += y[i];
  ```
  requires only a single load, add, and store for each block of vector length elements, while a conventional system requires a load, add, and store for each element.
- Interleaved memory. The memory system consists of multiple “banks” of memory which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather. In strided memory access, the program accesses elements of a vector located at fixed intervals, i.e: accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four. Scatter/gather in this context, is writing (scatter) or

reading (gather) elements of a vector located at irregular intervals, i.e.: accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

Vector processors have the virtue that for many applications, they are very fast and very easy to use. Vectorizing compilers are quite good at identifying code that can be vectorized and loops that cannot be vectorized, providing information about why a loop couldn't be vectorized. The user can thereby make informed decisions about whether it's possible to rewrite the loop so that it will vectorize. Vector systems have very high memory bandwidth, and every data item that's loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line. On the other hand, they do not handle irregular data structures as well as other parallel architectures, and there seems to be a very finite limit to their scalability, that is, their ability to handle ever larger problems. It is difficult to see how systems could be created that would operate on ever longer vectors. Current generation systems scale by increasing the number of vector processors, not the vector length. Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

Graphics processing units

Real-time graphics application programming interfaces (APIs) use points, lines, and triangles to internally represent the surface of an object. They use a graphics processing pipeline to convert the internal representation into an array of pixels that can be sent to a computer screen. Several of the stages of this pipeline are programmable. The behavior of the programmable stages is specified by functions called shader functions. The shader functions are typically quite short—often just a few lines of C code. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream. Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism, and in the current generation all GPUs use SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.

Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual. GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread. The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function. A drawback here is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams. GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power. A special reference to GPUs is made on Chapter 7.

6.2.2 MIMD SYSTEMS

Multiple instruction, multiple data (MIMD) systems support multiple simultaneous instruction streams operating on multiple data streams. MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU. Furthermore, unlike SIMD systems, MIMD systems are usually asynchronous, that is, the processors can operate at their own pace. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, processors executing the same sequence of instructions may be executing different statements of this sequence at a point in time.

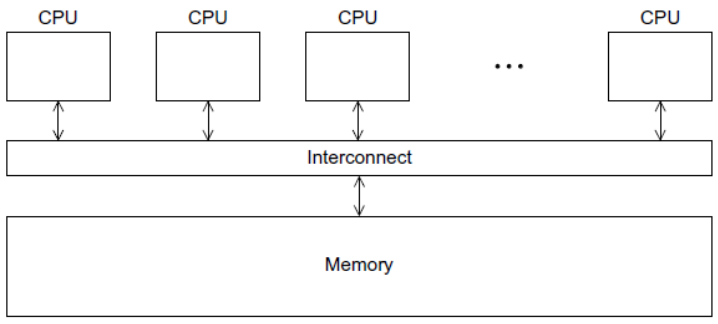


Figure 6.3 - A shared memory system

There are two principal types of MIMD systems: shared-memory systems and distributed-memory systems. In a shared-memory system, a collection of autonomous processors is connected to a memory system via an interconnection network and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures. On the other hand, in a distributed-memory system, each processor is paired with its own private memory and the processor-memory pairs communicate over an interconnection network. In distributed-memory systems the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor as seen in Figure 6.3 and Figure 6.4

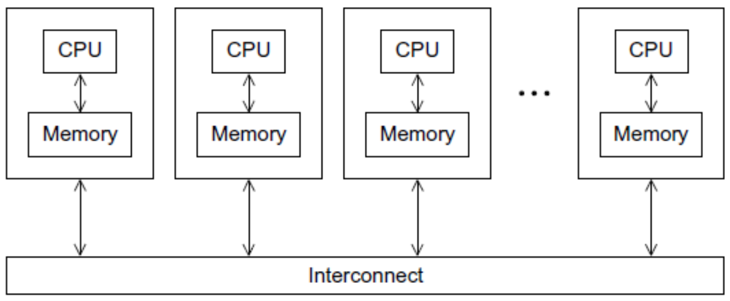


Figure 6.4 - A distributed memory system

Shared memory systems

The most widely available shared-memory systems use one or more multicore processors. A multicore processor has multiple CPUs or cores on a single chip where the cores, typically have private level 1 caches, while other caches may or may not be shared between the cores.

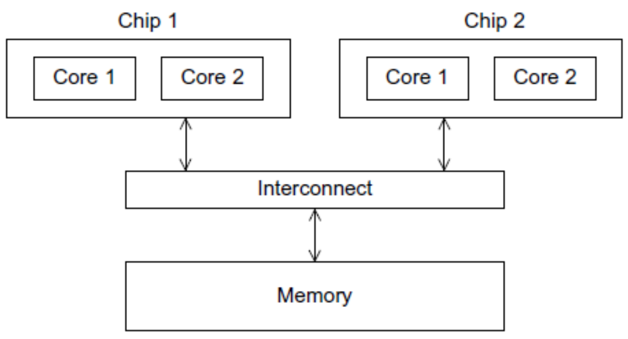


Figure 6.5 - A UMA multicore system

In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each others' blocks of main memory through special hardware built into the processors, as seen in Figure 6.5 and Figure 6.6. In the first type of system, the time to access all the memory locations will be the same for all the cores, while in the second type a memory location to which a core is directly connected can be accessed more quickly than a memory location that must be accessed through another chip. The first type of system is called a uniform memory access (UMA) system, while the second type is called a nonuniform memory access (NUMA) system. UMA systems are usually easier to program, since the programmer does not need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

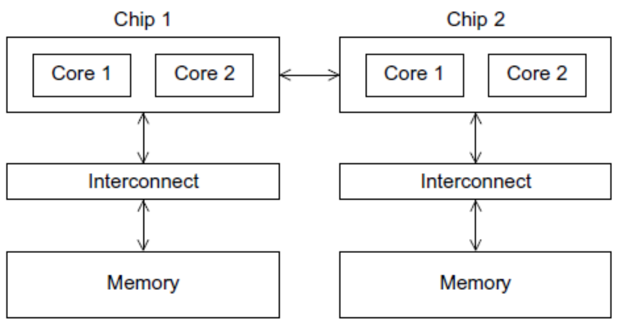


Figure 6.6 - A NUMA multicore system

Distributed memory systems

The most widely available distributed-memory systems are called clusters. They are composed of a collection of commodity systems like PCs, connected by a commodity interconnection network like Ethernet. In fact, the nodes of these systems, the individual computational units joined together by the communication network, are usually shared-memory systems with one or more multicore processors. To distinguish such systems from pure distributed-memory systems, they are sometimes called hybrid systems.

The grid provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be heterogeneous, that is, the individual nodes may be built from different types of hardware.

6.2.3 INTERCONNECTION NETWORKS

The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program.

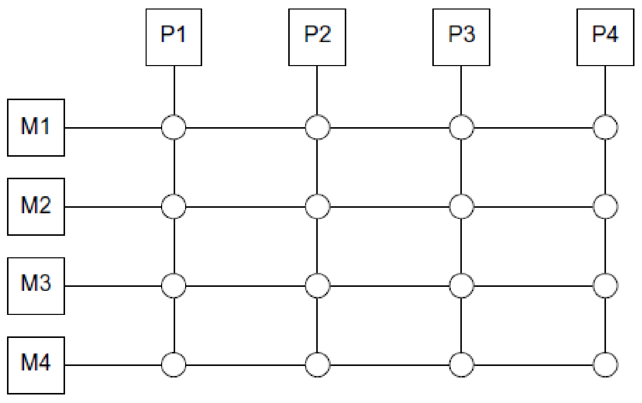


Figure 6.7 - A crossbar switch connecting four processors (Pi) and four memory modules (Mj)

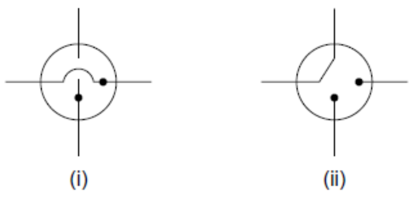


Figure 6.8 - Configuration of internal switches in a crossbar

Shared memory interconnects

Currently the two most widely used interconnects on shared-memory systems are buses and crossbars. A bus is a collection of parallel communication wires together with some hardware that controls access to it, with these communication wires to be shared by the devices that are connected to it. Buses have low cost and flexibility since multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, the likelihood of contention for use of the bus increases, as the number of devices connected to the bus increases, with the expected performance of the bus to be decreasing. Therefore, if a large number of processors are connected to a bus, the processors are expected to wait for access to main memory frequently.

As the size of shared-memory systems increases, buses are rapidly being replaced by switched interconnects. Switched interconnects use switches to control the routing of data among the connected devices. A crossbar is illustrated in Figure 6.7 where the lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

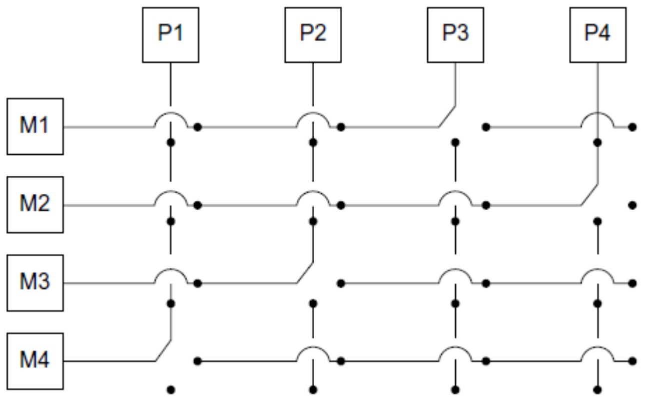


Figure 6.9 - Simultaneous memory accesses by the processors

The individual switches can assume one of the two configurations shown in Figure 6.8. With these switches and at least as many memory modules as processors, there will only be a conflict between two cores attempting to access memory. This conflict will occur when the two cores attempt to simultaneously access the same memory module. Figure 6.9 shows the configuration of the switches if P1 writes to M4, P2 reads from M3, P3 reads from M1, and P4 writes to M2.

Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

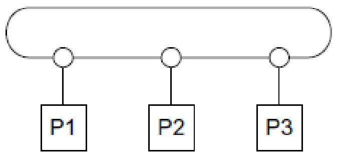


Figure 6.10 - A ring

Distributed memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects. In a direct interconnect each switch is directly connected to a processor-memory pair and the switches are connected to each other. Figure 6.10 shows a ring while Figure 6.11 shows a two-dimensional toroidal mesh. As before, the circles are switches, the squares are processors, and the lines are bidirectional links. A ring is superior to a simple bus since it allows multiple simultaneous communications.

The toroidal mesh will be more expensive than the ring, because the switches are more complex—they must support five links instead of three—and if there are p processors, the number of links is 3p in a toroidal mesh, while it's only 2p in a ring. However, the number of possible simultaneous communications patterns is greater with a mesh than with a ring.

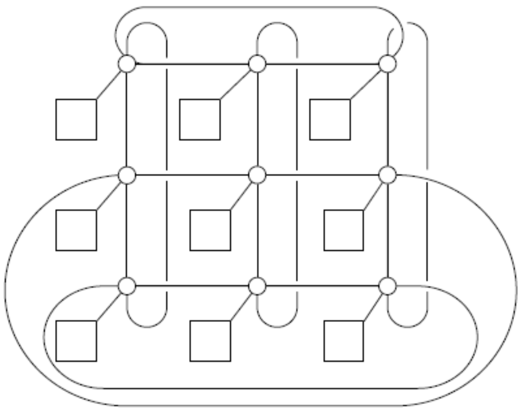


Figure 6.11 - A toroidal mesh

One measure of "number of simultaneous communications" or "connectivity" is bisection width. To understand this measure, assume a parallel system divided into two halves with each half containing half of the processors or nodes. Bisection width is the quantity of simultaneous communications that can take place "across the divide" between the two halves. In Figure 6.12 a ring with eight nodes is divided into two groups of four nodes and it is evident that only two communications can take place between the halves. However, in Figure 6.13, the nodes are divided into two parts so that four simultaneous communications can take place. Since bisection width is supposed to give a "worst-case" estimate, it is assumed to be two, not four.

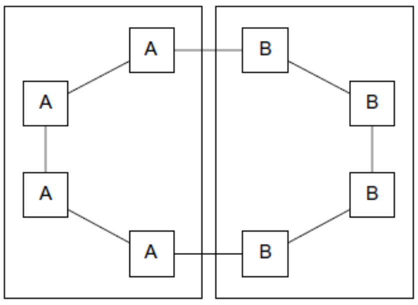


Figure 6.12 - Ring bisection, two communications between the two halves

An alternative way of computing the bisection width is to remove the minimum number of links needed to split the set of nodes into two equal halves. The number of links removed is the bisection width. Assuming a square two-dimensional toroidal mesh with $p = q^2$ nodes where q is even, the nodes can be split into two halves by removing the "middle" horizontal links and the "wraparound" horizontal links as in Figure 6.14. This suggests that the bisection width is at most $2q = 2\sqrt{p}$. In fact, this is the smallest possible number of links and the bisection width of a square two-dimensional toroidal mesh is $2\sqrt{p}$.

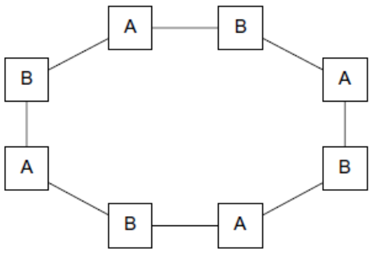


Figure 6.13 - Ring bisection of four communications

The bandwidth of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second. Bisection bandwidth is often used as a measure of network quality and it is similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links. For example, if the links in a ring have a bandwidth of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

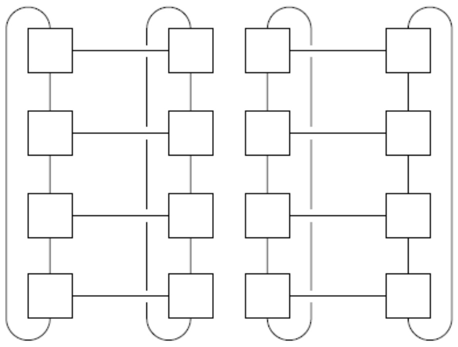


Figure 6.14 - A bisection of a toroidal mesh

The ideal direct interconnect is a fully connected network in which each switch is directly connected to every other switch as in Figure 6.15 with a bisection width of $\frac{p^2}{4}$. However, it's impractical to construct such an interconnect for systems with more than a few nodes, since it requires a total of $\frac{p^2}{2} + \frac{p}{2}$ links, and each switch must be capable of connecting to p links. It is therefore a “theoretical best possible” interconnect and not a practical one, only used as a basis for evaluating other interconnects.

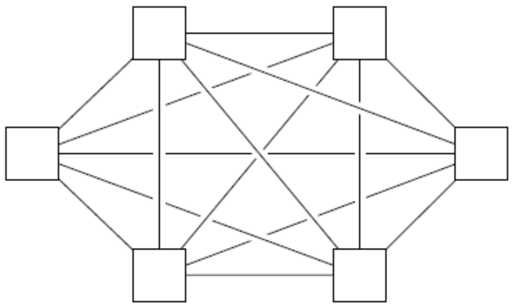


Figure 6.15 - A fully connected network

The hypercube is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively. A one-dimensional hypercube is a fully-connected system with two processors. A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches. Similarly, a three-dimensional hypercube is built from two two-dimensional hypercubes as in Figure 6.16. Thus, a hypercube of dimension d has $p = 2^d$ nodes, and a switch in a d-dimensional hypercube is directly connected to a processor and d switches. The bisection width of a hypercube is $\frac{p}{2}$, so it has more connectivity than a ring or toroidal mesh, but the switches must be more powerful, since they must support $1 + d = 1 + \log_2(p)$ wires, while the mesh switches only require five wires. This constitutes a hypercube with p nodes to be more expensive to construct than a toroidal mesh.

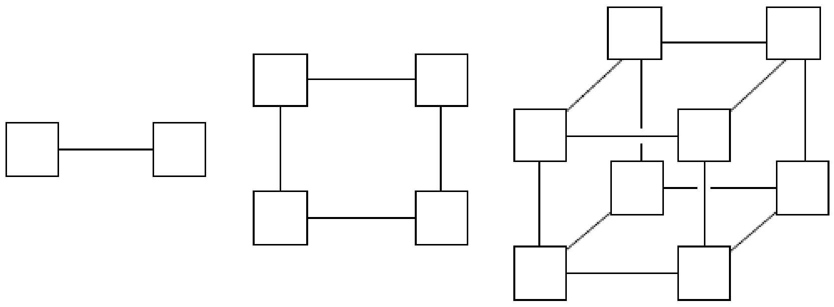


Figure 6.16 - 1D, 2D and 3D hypercubes

Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network as in Figure 6.17.

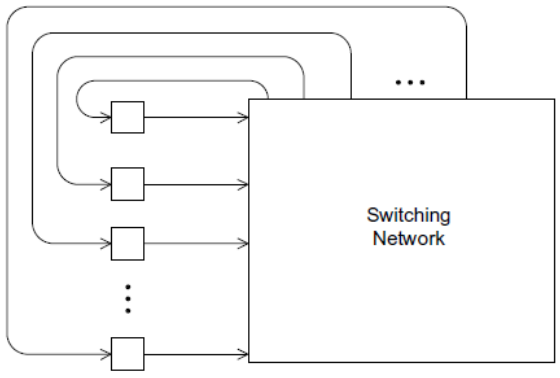


Figure 6.17 - A generic indirect network

The crossbar and the omega network are relatively simple examples of indirect networks. The diagram of a distributed-memory crossbar in Figure 6.18 has unidirectional links. Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor. An omega network is shown in Figure 6.19. The switches are two-by-two crossbars as in Figure 6.20. Unlike the crossbar, there are communications that cannot occur simultaneously, i.e.: in Figure 6.19, if processor 0 sends a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega network is less expensive than the crossbar. The omega network uses $\frac{1}{2} p \log_2(p)$ of the 2x2 crossbar switches, so it uses a total of $2p\log_2(p)$ switches, while the crossbar uses p^2.

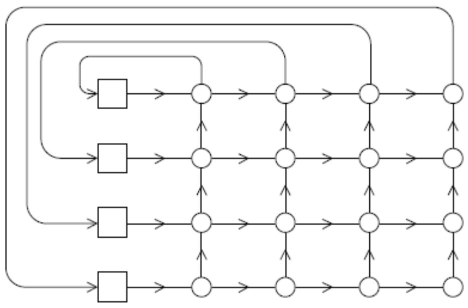


Figure 6.18 - A crossbar interconnect for distributed memory

It's a little bit more complicated to define bisection width for indirect networks. However, the principle is the same: the nodes must be divided into two groups of equal size and communication that can take place between the two halves is determined, or alternatively, the minimum number of links that need to be removed so that the two groups can't communicate. The bisection width of a p x p crossbar is p and the bisection width of an omega network is $\frac{p}{2}$.

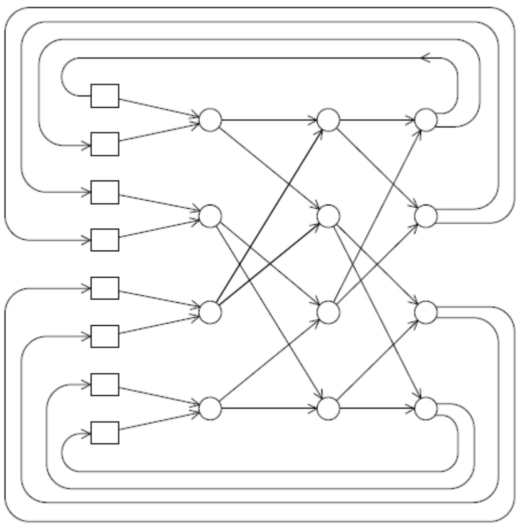


Figure 6.19 - An omega network

Latency and bandwidth

Any time data is transmitted, there is a need to know how long it will take for the data to reach its destination. This is true whether referring about transmitting data between main memory and cache, cache and register, hard disk and memory, or between two nodes in a distributed-memory or hybrid system. There are two figures that are often used to describe the performance of an interconnect, the latency and the bandwidth. The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is

$$T = l + \frac{n}{b} \qquad (6.2.1)$$

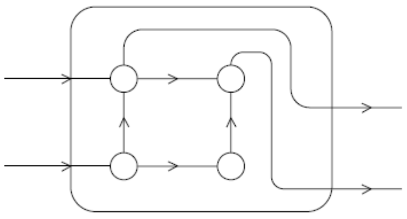


Figure 6.20 - A switch in an omega network

These terms are often used in different ways. Latency is sometimes used to describe total message transmission time or the time required for any fixed overhead involved in transmitting data. During a message transmission between two nodes in a distributed memory system, a message is not just raw data as it might include the data to be transmitted, a destination address, some information specifying the size of the message, some information for error correction, and so on. In such a case, latency might be the time it takes to assemble the message on the sending side, the time needed to combine the various parts and the time to disassemble the message on the receiving side, the time needed to extract the raw data from the message and store it in its destination.

6.2.4 CACHE COHERENCE

As seen in Section 6.1.3, CPU caches are managed by system hardware; programmers do not have direct control over them. This has several important consequences for shared-memory systems. To better illustrate these issues, assume a shared memory system with two cores, each of which has its own private data cache. Moreover, assume that x is a shared variable that has been initialized to 2, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1 with the following statements to be executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

The memory location for y0 will eventually get the value 2, and the memory location for y1 will eventually get the value 6. However, it is not so clear what value z1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z1, z1 will get the value 4 x 7=28. However, at time 0, x is in the cache of core 1, so unless for some reason x is evicted from core 0’s cache and then reloaded into core 1’s cache, it actually appears that the original value x = 2 may be used, and z1 will get the value 4 x 2 = 8.

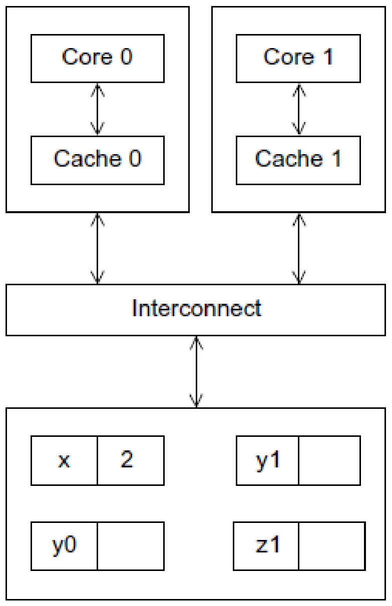


Figure 6.21 - A shared-memory system with two cores and two caches

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy. If it is using a write-through policy, the main memory will be updated by the assignment x = 7. However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of x in the cache of core 0 probably won't even be available to core 1 when it updates z1.

The programmer does not have direct control over when the caches are updated, so a program cannot execute these statements and know what will be stored in z1. The caches described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. This issue is called the cache coherence problem.

Snooping cache coherence

There are two main approaches to insuring cache coherence: snooping cache coherence and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect is a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and writeback caches. In principle, if the interconnect is shared with writethrough caches, there is no need for additional traffic on the interconnect since each core can simply "watch" for writes. With write-back caches, an extra communication is necessary since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive and snooping cache coherence requires a broadcast every time a variable is updated. In that sense, snooping cache coherence is not scalable because, for larger systems, it will cause performance to degrade. For example, assume a system with the basic distributed-memory architecture and a single address space for all the memories, i.e.: core 0 can access the variable x stored in core 1's memory, by simply executing a statement such as y = x. Such a system can scale to very large numbers of cores, in principle. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a directory which stores the status of each cache line. Typically, this data structure is distributed, so in this example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated. It is evident that there will be substantial additional storage required for the directory but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It is important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. Assume that a call to function f(i,j) will occur repeatedly, adding the computed values into a vector:

```
int i, j, m, n;

double y[m];

/* Assign y = 0 */

. . .
```

```
for (i = 0; i < m; i++)

for (j = 0; j < n; j++)

y[i] += f(i,j);
```

This can be parallelized by dividing the iterations in the outer loop among the cores. If we have core count cores, we might assign the first m/core count iterations to the first core, the next m/core count iterations to the second core, and so on, as shown below:

```
/* Private variables */

int i, j, itercount;

/* Shared variables initialized by one core */

int m, n, core count

double y[m];

itercount = m/corecount

/* Core 0 does this */

for (i = 0; i < itercount; i++)

for (j = 0; j < n; j++)

y[i] += f(i,j);

/* Core 1 does this */

for (i = itercount+1; i < 2*itercount; i++)

for (j = 0; j < n; j++)

y[i] += f(i,j);

. . .
```

Assume a shared-memory system with two cores, m = 8, doubles are eight bytes, cache lines are 64 bytes, and y[0] is stored at the beginning of a cache line. A cache line can store eight doubles, and y takes one full cache line. Since all of y is stored in a single cache line, each time one of the cores executes the statement y[i] += f(i,j), the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory. If n is large, a large percentage of the assignments y[i] += f(i,j) are expected to access main memory, despite the fact that core 0 and core 1 never access each others' elements of y. This is called false sharing, because the system is behaving as if the elements of y were being shared by the cores. False sharing does not cause incorrect results but it can degrade performance of a program by causing many more accesses to memory than necessary. This effect can be reduced by using temporary storage that is local to the thread or process and then copying the temporary storage to the shared storage.

6.2.5 SHARED-MEMORY VERSUS DISTRIBUTED-MEMORY

Newcomers to parallel computing sometimes wonder why all MIMD systems are not shared-memory, since most programmers find the concept of implicitly coordinating the work of the processors through shared data structures more appealing than explicitly sending messages. The principal hardware issue is the cost of scaling the interconnect. As processors are added to a bus, the chance that there will be conflicts over access to the bus increase dramatically, so buses are suitable for systems with only a few processors. Large crossbars are very expensive, so it is also unusual to find systems with large crossbar interconnects.

On the other hand, distributed-memory interconnects such as the hypercube and the toroidal mesh are relatively inexpensive, and distributed-memory systems with thousands of processors that use these and other interconnects have been built. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

6.3 PARALLEL SOFTWARE

Parallel hardware is now a commodity since virtually all desktop and server systems use multicore processors. However, the same does not apply to parallel software. With the exception of operating systems, database systems, and web servers, there is currently very little commodity software that makes extensive use of parallel hardware. This poses a problem because we can no longer rely on hardware and compilers to provide a steady increase in application performance. In order to continue having increases in application performance and application power, it is imperative to develop applications that exploit shared- and distributed-memory architectures. In this section some of the issues involved in writing software for parallel systems will be presented along with some terminology.

6.3.1 PROCESS AND THREAD COORDINATION

Assume the addition of two arrays, as shown below:

```
double x[n], y[n];

. . .

for (int i = 0; i < n; i++)
```

```
x[i] += y[i];
```

In order to parallelize this loop, assigning elements of the arrays to processes/threads is necessary. Assuming p processes/threads, process/thread 0 will be responsible for elements 0, … ,n/p-1, process/thread 1 will be responsible for elements n=p, … ,2n/p-1, and so on.

In this example, the programmer only needs to:

1. Divide the work among the processes/threads in such a way that
   a. each process/thread gets roughly the same amount of work, and
   b. the amount of communication required is minimized.

   The process of dividing the work among the processes/threads so that (a) is satisfied is called load balancing. The process of converting a serial program or algorithm into a parallel program is often called parallelization. Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be embarrassingly parallel. However, the vast majority of problems are much more difficult to parallelize and for these problems, coordination of the work of the processes/threads is needed. In these programs, we also usually need to
2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among the processes/threads.

These last two tasks are often interrelated. For example, in distributed-memory programs, the processes are implicitly synchronized when they communicate and in shared-memory programs, we often communicate among the threads by synchronizing them.

6.3.2 SHARED MEMORY

In shared-memory programs, variables can be shared or private. Shared variables can be read or written by any thread, and private variables can ordinarily be accessed by only one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

In many environments, shared-memory programs use dynamic threads. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests and when a new request arrives, it forks a worker thread; the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.

An alternative to the dynamic paradigm is the static thread paradigm where all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may free resources that are no longer needed and terminate. In terms of resource usage, this may be less efficient: if a thread is idle, its resources i.e.: stack, program counter, etc, cannot be freed. On the other hand, forking and joining threads can be fairly time-consuming operations so, if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It is also closer to the most widely used paradigm for distributed-memory programming so, part of the mindset that is used for one type of system is preserved for the other.

Non-determinism

In any MIMD system in which the processors execute asynchronously it is likely that there will be non-determinism. A computation is non-deterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they will complete statements varies from run to run, constituting the results of the program different from run to run. As a very simple example, assume two threads, one with id or rank 0 and the other with id or rank 1. Assume also that each is storing a private variable called my_x with thread 0's value for my_x being 7, and thread 1's being 19. Moreover assume that both threads execute the following code:

```
. . .
printf("Thread %d > my val = %dnn", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my val = 7
Thread 1 > my val = 19
```

but it could also be

```
Thread 1 > my val = 19
Thread 0 > my val = 7
```

Since threads are executing independently and interacting with the operating system, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

In many cases non-determinism does not pose a problem. In our example, since we have labelled the output with the thread's rank, the order in which the output appears probably does not matter. However, there are also many cases in which non-determinism, especially in shared-memory programs, can be disastrous because it can easily result in program errors. Here is a simple example with two threads. Assume each thread computes an int which is stored in a private variable called my_val. Moreover, assume the values stored in my_val need to be added into a shared-memory location x that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my val = Compute val(my rank);

x += my val;
```

An addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Clearly this is not what we want, and it is easy to imagine other sequences of events that result in an incorrect value for x. The non-determinism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location x. When threads or processes attempt to simultaneously access a resource, and the accesses can result in an error, we often say the program has a race condition because the threads or processes are said to be in a "horse race." This means that the outcome of the computation depends on which thread wins the race. In our example, the threads are in a race to execute x += my_val. In this case, unless one thread completes x += my_val before the other thread starts, the result will be incorrect. A block of code that can only be executed by one thread at a time is called a critical section and it is up to the programmer to insure mutually exclusive access to the critical section, meaning that it needs to be insured that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for insuring mutual exclusion is a mutual exclusion lock or mutex or lock. A mutex is a special type of object that is supported by the underlying hardware. The basic idea is that each critical section is protected by a lock. Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a mutex function, and, when it is done executing the code in the critical section, it should "relinquish" the mutex by calling an unlock function. While one thread "owns" the lock, meaning that it has returned from a call to the lock function but has not yet called the unlock function, any other thread attempting to execute the code in the critical section will wait in its call to the lock function.

Thus, in order to insure that the previous code functions correctly, it needs to be modified as shown below:

```
my_val = Compute_val(my_rank);

Lock(&add_my_val_lock);

x += my_val;

Unlock(&add_my_val_lock);
```

This insures that only one thread at a time can execute the statement x += my_val. Note that the code does not impose any predetermined order on the threads. Either thread 0 or thread 1 can execute x += my_val first. Also note that the use of a mutex enforces serialization of the critical section.

Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible. There are alternatives to mutexes. In busy-waiting, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable ok_for_1 that has been initialized to false. For that case, thread 1 won't update x until after thread 0 has updated it, as shown below:

```
my_val = Compute_val(my_rank);

if (my_rank == 1)

while (!ok_for_1); /* Busy-wait loop */

x += my_val; /* Critical section */

if (my_rank == 0)

ok_for_1 = true; /* Let thread 1 update x */
```

So until thread 0 executes ok for 1 = true, thread 1 will be stuck in the loop while (!ok for 1). This loop is called a “busy-wait” because the thread can be very busy waiting for the condition. Despite the fact that busy-wait is simple to understand and implement, it can be very wasteful of system resources because, even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered.

Semaphores are similar to mutexes, although the details of their behavior are slightly different, and there are some types of thread synchronization that are easier to implement with semaphores than mutexes. A monitor provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time.

There are a number of other alternatives that are currently being studied but that are not yet widely available. The one that has attracted the most attention is probably transactional memory [75]. In database management systems, a transaction is an access to a database that the system treats as a single unit. For example, transferring $1000 from a savings account to a checking account should be treated by a bank’s software as a transaction, so that the software cannot debit the savings account without also crediting the checking account. If the software was able to debit the savings account, but was then unable to credit the checking account, it would rollback the transaction meaning that the transaction would, either be fully completed or any partial changes would be erased. The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

Thread safety

In many cases, parallel programs can call functions developed for use in serial programs without any issues or problems. However, there are some notable exceptions with the most important being functions that make use of static local variables. Ordinary local variables which are variables declared inside a function, are allocated from the system stack. Since each thread has its own stack, ordinary local variables are private. However, a static variable that is declared in a function, persists from one call to the next. Thus, static variables are effectively shared among any threads that call the function leading to unexpected and unwanted consequences.

For example, the C string library function strtok splits an input string into substrings. When it’s first called, it’s passed a string, and on subsequent calls it returns successive substrings. This can be arranged through the use of a static char* variable that refers to the string that was passed on the first call. Now suppose two threads are splitting strings into substrings. Clearly, if, for example, thread 0 makes its first call to strtok, and then thread 1 makes its first call to strtok before thread 0 has completed splitting its string, then thread 0's string will be lost or overwritten, and, on subsequent calls it may get substrings of thread 1's strings.

A function such as strtok is not thread safe. This means that if it is used in a multithreaded program, there may be errors or unexpected results. When a block of code is not thread safe, it is usually because different threads are accessing shared data. Thus, even though many serial functions can be used safely in multithreaded programs, meaning that they are thread-safe, programmers need to be cautious of functions that were written exclusively for use in serial programs.

6.3.3 DISTRIBUTED MEMORY

In distributed-memory programs, the cores can directly access only their own, private memories. There are several APIs that are used but the most widely used is message-passing. Distributed-memory APIs is can be used with shared-memory hardware; it is perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads and a library or compiler can implement the communication that's needed. Distributed-memory programs are usually executed by starting multiple processes rather than multiple threads since, typical "threads of execution" in a distributed-memory program may run on independent CPUs with independent operating systems and there may be no software infrastructure for starting a single "distributed" process and having that process fork one or more threads on each node of the system.

Message-passing

A message-passing API provides (at a minimum) a send and a receive function. Processes typically identify each other by ranks in the range 0, 1, … , p-1, where p is the number of processes. As an example, process 1 might send a message to process 0 with the following pseudo-code:

```
char message[100];

. . .

my rank = Get_rank();

if (my_rank == 1) {

sprintf(message, "Greetings from process 1");

Send(message, MSG CHAR, 100, 0);

} else if (my rank == 0) {

Receive(message, MSG CHAR, 100, 1);
printf("Process 0 > Received: %snn", message);

}
```

Here the Get_rank function returns the calling process' rank, then the processes branch depending on their ranks. Process 1 creates a message and then sends it to process 0 with the call to Send. The arguments to the call are in order: the message, the type of the elements in the message (MSG CHAR), the number of elements in the message (100), and the rank of the destination process (0). On the other hand, process 0 calls Receive with the following arguments: the variable into which the message will be received (message), the type of the message elements, the number of elements available for storing the message, and the rank of the process sending the message. After completing the call to Receive, process 0 prints the message.

Several points are worth noting here:

1. The program segment is SPMD. The two processes are using the same executable, but carrying out different actions. In this case, what they do depends on their ranks.
2. The variable message refers to different blocks of memory on the different processes. Programmers often stress this by using variable names such as my message or local message.
3. Process 0 can write to stdout. This is usually the case: most implementations of message-passing APIs allow all processes access to stdout and stderr—even if the API doesn't explicitly provide for this.

There are several possibilities for the exact behavior of the Send and Receive functions with most message-passing APIs providing several different send and/or receive functions. The simplest behavior is for the call to Send to block until the call to Receive starts receiving the data. This means that the process calling Send will not return from the call until the matching call to Receive has started. Alternatively, the Send function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied. The most common behavior for the Receive function is for the receiving process to block until the message is received.

Typical message-passing APIs also provide a wide variety of additional functions. For example, there may be functions for various "collective" communications, such as a broadcast, in which a single process transmits the same data to all the processes, or a reduction, in which results computed by the individual processes are combined into a single result—for example, values computed by the processes are added. There may also be special functions for managing processes and communicating complicated data structures. The most widely used API for message passing is the Message-Passing Interface or MPI.

Message-passing is a very powerful and versatile API for developing parallel programs. Virtually all of the programs that are run on the most powerful computers in the world use message-passing. However, it is also very low level as there is a huge amount of detail that the programmer needs to manage and, in order to parallelize a serial program, it is usually necessary to rewrite the vast majority of the program. The data structures in the program may have to either be replicated by each process or be explicitly distributed among the processes. Furthermore, the rewriting process usually cannot be done incrementally. For example, if a data structure is used in many parts of the program, distributing it for the parallel parts and collecting it for the serial (unparallelized) parts will probably be prohibitively expensive. Therefore, message passing is sometimes called "the assembly language of parallel programming," and there have been many attempts to develop other distributed-memory APIs.

One-sided communication

In message-passing, one process, must call a send function and the send must be matched by another process' call to a receive function. Any communication requires the explicit participation of two processes. In one-sided communication, or remote memory access, a single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process. This can simplify communication, since it only requires the active participation of a single process. Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes. It can also reduce overhead by eliminating the overhead of one of the function calls (send or receive).

It should be noted that some of these advantages may be hard to realize in practice. For example, if process 0 is copying a value into the memory of process 1, 0 must have some way of knowing when it's safe to copy, since it will overwrite some memory location. Process 1 must also have some way of knowing when the memory location has been updated. The first problem can be solved by synchronizing the two processes before the copy, and the second problem can be solved by another synchronization or by having a "flag" variable that process 0 sets after it has completed the copy. In the latter case, process 1 may need to poll the flag variable in order to determine that the new value is available by repeatedly checking the flag variable until it gets the value indicating 0 has completed its copy. These issues can considerably increase the overhead associated with transmitting a value. A further difficulty is that since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.

Partitioned global address space languages

Since many programmers find shared-memory programming more appealing than message-passing or one-sided communication, a number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware. This is not a straightforward process; if simply a compiler was developed that treated the collective memories in a distributed-memory system as a single large memory, programs would have poor, or, at best, unpredictable performance because, each time a running process accessed memory, it might access local memory (memory belonging to the core on which it was executing) or remote memory (memory belonging to another core).

Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudo-code for a shared-memory vector addition:

```
shared int n = . . . ;

shared double x[n], y[n];

private int i, my_first_element, my_last_element;

my first element = . . . ;

my last element = . . . ;

/* Initialize x and y */

. . .

for (i = my_first_element; i <= my_last_element; i++)

x[i] += y[i];
```

At first, two shared arrays are declared, then, on the basis of the process' rank, the elements of the array "belong" to each process are determined. After initializing the arrays, each process adds its assigned elements. If the assigned elements of x and y have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of x is assigned to core 0 and all of y is assigned to core 1, then the performance is likely to be vastly degraded since, each time the assignment x[i] += y[i] is executed, the process will need to refer to remote memory.

Partitioned global address space (PGAS) languages such as the ones presented in [76], provide some of the mechanisms of shared-memory programs and also provide the programmer with tools to avoid the problem mentioned above. Private variables are allocated in the local memory of the core on which the process is executing and the distribution of the data in shared data structures is controlled by the programmer..

Programming hybrid systems

It is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this “hybrid” API makes program development extremely difficult [77]. Most commonly, such systems are usually programmed using a single, distributed-memory API for both inter- and intra-node communication.

6.3.4 PARALLEL PROGRAM DESIGN

In general, parallelizing a serial program includes the division of the work among the processes/threads so that each process gets roughly the same amount of work and communication is minimized. In most cases, special consideration is needed for the processes/threads to synchronize and communicate.

Unfortunately, there is no universal parallelization process to be followed but an outline of steps as suggested in [78] can be as follows:

1. Partitioning. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. Communication. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. Agglomeration or aggregation. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. Mapping. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

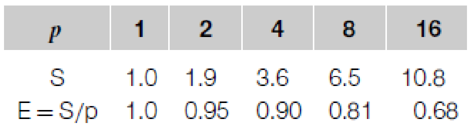
This is sometimes called Foster’s methodology.

6.4 PERFORMANCE

In the following sections, various metrics for measuring performance benefits when executing parallel codes in parallel hardware are presented.

6.4.1 SPEEDUP AND EFFICIENCY

The most efficient parallelization implementation is to equally divide the work among the cores while at the same time introducing no additional work for the cores. For that case, should a program is executed using p cores with one thread or process on each core, it will run p times faster than the serial program. If serial run-time is T_{serial} and the parallel run-time is $T_{parallel}$ then the maximum performance benefit is $T_{parallel} = \frac{T_{serial}}{p}$. In such cases, a parallel program is said to have linear speedup.



p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 6.4 - Speedups and efficiencies of a parallel program

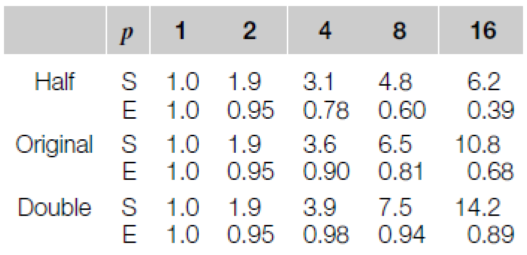
In practice, linear speedup is unlikely to be achieved because the use of multiple processes/threads almost invariably introduces some overhead. Shared memory programs will almost always have critical sections, which will require the use of some mutual exclusion mechanism such as a mutex. The calls to the mutex functions are overhead that is not present in the serial program and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, will not have these overheads. Thus, it will be very unusual for us to find that our parallel programs get linear speedup. Furthermore, it is likely that the overheads will increase as we increase the number of processes or threads with more threads probably leading to more threads that need to access a critical section. More processes will probably mean more data needs to be transmitted across the network. So if we define the speedup of a parallel program to be:

$$S = \frac{T_{serial}}{T_{parallel}} \qquad (6.4.1)$$

then linear speedup has S=p, which is unusual. Furthermore, as p increases, S is expected to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that $\frac{S}{p}$ will probably get smaller and smaller as p increases as show in Table 6.4 The value $\frac{S}{p}$, is sometimes called the efficiency of the parallel program. By substituting the formula for S, efficiency is equal to:

$$E = \frac{S}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{p \cdot T_{parallel}} \tag{6.4.2}$$

It is clear that $T_{parallel}$, S, and E depend on p, the number of processes or threads. Moreover, $T_{parallel}$, S, E, and T_{serial} all depend on the problem size. For example, if the problem size is doubled or halved for the whose speedups are shown in Table 6.4, the speedups and efficiencies shown in Table 6.5 occur.



	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Table 6.5 - Speedups and efficiencies of a parallel program on different problem sizes

In this example, increasing the problem size results to an increase in speedups and efficiencies, while a decrease in these quantities occurs when the problem size is decreased. Such a behavior is quite common.

Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead" such as mutual exclusion or communication. Therefore, if $T_{overhead}$ denotes this parallel overhead, it's often the case that

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead} \tag{6.4.3}$$

Furthermore, as the problem size is increased, $T_{overhead}$ often grows more slowly than T_{serial}. For such case, both speedup and efficiency will increase; since there is more work for the processes/threads to do, the relative amount of time spent coordinating the work of the processes/threads should be less.

A final issue to consider is what values of T_{serial} should be used when reporting speedups and efficiencies as it is stated that they should be the run-time of the fastest program on the fastest processor available. In practice, most authors use a serial program on which the parallel program was based and run it on a single processor of the parallel system.

6.4.2 AMDAHL'S LAW

Amdahl's law [79] states that unless virtually all parts of a serial program are parallelized, the possible speedup is going to be very limited, regardless of the number of cores available. In order to illustrate this, assume that 90% of a serial program is parallelized and that the parallelization is "perfect," meaning that, regardless of the number of cores p used, the speedup of this part of the program will be p. If the serial run-time is T_{serial} = 20 seconds then the run-time of the parallelized part will be $0.9 \times \frac{T_{serial}}{p} = \frac{18}{p}$ and the run-time of the "unparallelized" part will be $0.1 \times T_{serial} = 2$. The overall parallel run-time will be equal to:

$$T_{parallel} = 0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial} = \frac{18}{p} + 2 \tag{6.4.4}$$

and the speedup will be equal to:

$$S = \frac{T_{serial}}{0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial}} = \frac{20}{\frac{18}{p} + 2} \tag{6.4.5}$$

Now as p gets larger and larger, $0.9 \times \frac{T_{serial}}{p} = \frac{18}{p}$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{serial} = 2$, meaning that the denominator in S cannot be smaller than $0.1 \times T_{serial} = 2$. The fraction S must therefore be smaller than:

$$S \leq \frac{T_{serial}}{0.1 \times T_{serial}} = \frac{20}{2} = 10 \tag{6.4.6}$$

This means that even though 90% of the program is perfectly parallelized, a speedup better than 10 will never be achieved even if the program is run on a parallel computing environment with 100 or 1000 cores.

More generally, if a fraction r of the serial program remains unparallelized, then Amdahl's law implies that a speedup better than 1/r cannot be achieved. In the previous example, r=1-0.9=0.1=1/10, so the maximum is 10. If a fraction r of a serial program is "inherently serial," that is, cannot possibly be parallelized, then a speedup better than 1=r is not possible. Even if r is quite small like 1/100, having a system with thousands of cores at our disposal will not provide a speedup better than 100. Fortunately, Amdahl's law does not take into consideration the problem size. For many problems, an increase in problem size results in a decrease of the "inherently serial" fraction of the program as stated by Gustafson's law [80]. This is also backed up in practice where there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems.

6.4.3 SCALABILITY

Assume that a parallel program is run with a fixed number of processes/threads and a fixed input size, and an efficiency E is obtained. If the number of processes/threads that are used by the program are increased and there exists a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is said to be scalable.

In order to illustrate that, assume that T_{serial}=n microseconds and n is also the problem size. If $T_{parallel} = \frac{n}{p} + 1$, then efficiency is equal to:

$$E = \frac{n}{p\left(\frac{n}{p}+1\right)} = \frac{n}{n+p} \tag{6.4.7}$$

To determine if the program is scalable, the number of processes/threads is increased by a factor of k, and the factor x that needs to increase the problem size by is sought so that E is unchanged. If the number of processes/threads is kp and the problem size is xn, the following equation needs to be evaluated for x:

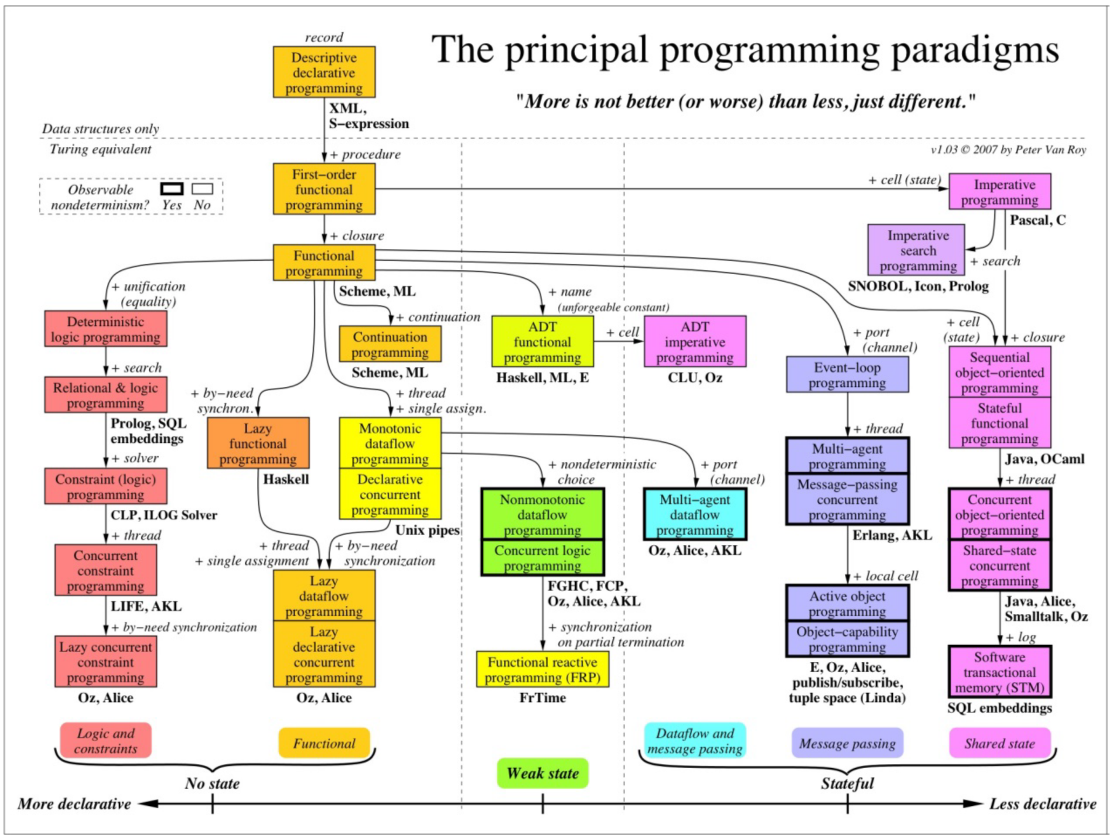
$$E = \frac{n}{n+p} = \frac{xn}{xn+kp} \tag{6.4.8}$$

If x=k, there will be a common factor of k in the denominator xn+kp=kn+kp=k(n+p) and the fraction can be reduced in order to get:

$$\frac{xn}{xn+kp} = \frac{kn}{k\left(n+p\right)} = \frac{n}{n+p} \tag{6.4.9}$$

This means that if the problem size is increased at the same rate that the number of processes/threads are increased, the efficiency will be unchanged, and the program is considered to be scalable.

When the number of processes/threads is increased while keeping the efficiency fixed without increasing the problem size, the program is said to be strongly scalable. If the efficiency is kept fixed by increasing the problem size at the same rate as the number of processes/threads is increased, then the program is said to be weakly scalable, as in the previous example.


The principal programming paradigms
"More is not better (or worse) than less, just different."
v1.03 © 2007 by Peter Van Roy
record
Descriptive declarative programming
XML, S−expression
Data structures only
Turing equivalent
+ procedure
Observable nondeterminism? Yes No
First−order functional programming
+ cell (state)
Imperative programming
Pascal, C
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ closure
Functional programming
Scheme, ML
+ unification (equality)
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by−need synchronization
Lazy concurrent constraint programming
Oz, Alice
+ continuation
Continuation programming
Scheme, ML
+ by−need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Unix pipes
+ thread
+ single assignment
+ by−need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, Oz
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
FGHC, FCP, Oz, Alice, AKL
+ synchronization on partial termination
Functional reactive programming (FRP)
FrTime
+ port (channel)
Multi−agent dataflow programming
Oz, Alice, AKL
+ port (channel)
Event−loop programming
+ thread
Multi−agent programming
Message−passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object−capability programming
E, Oz, Alice, publish/subscribe, tuple space (Linda)
+ cell (state)
+ closure
Sequential object−oriented programming
Stateful functional programming
Java, OCaml
+ thread
Concurrent object−oriented programming
Shared−state concurrent programming
Java, Alice, Smalltalk, Oz
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Weak state
Dataflow and message passing
Message passing
Shared state
No state
Stateful
More declarative
Less declarative

7.1 GPUS AS GENERAL PURPOSE PROCESSING PLATFORMS

With the evolution of high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multi-threaded, manycore processor with a vast computational potential and very high memory bandwidth.

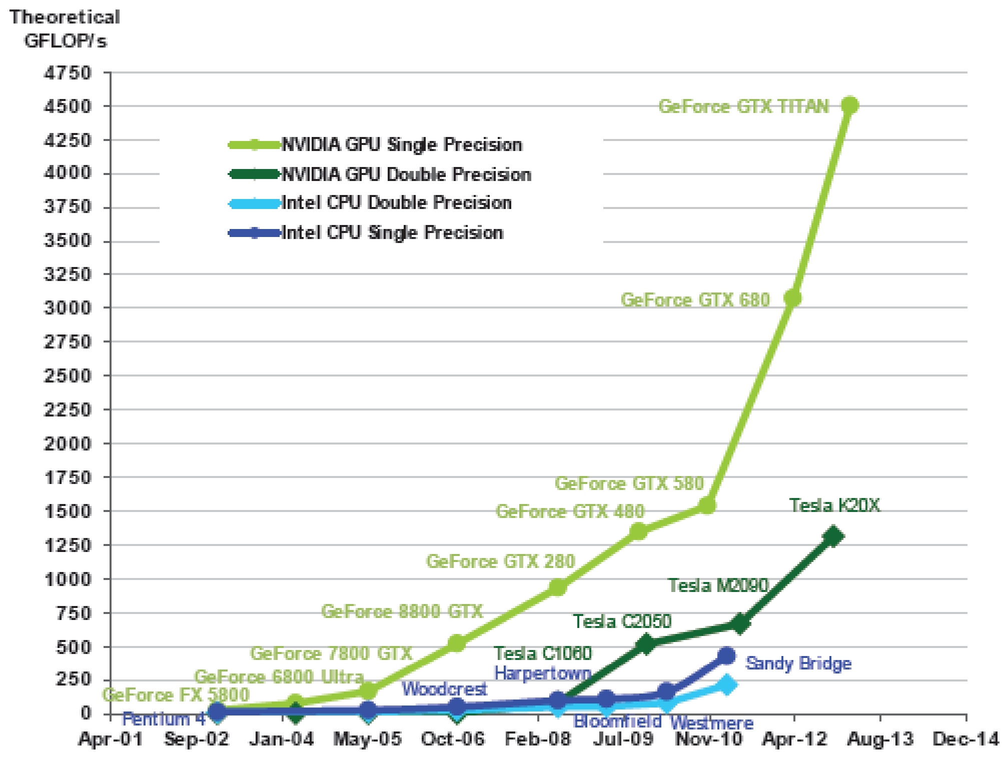


Figure 7.1 - Computational power of the NVIDIA GPUs

The reason behind this huge difference in floating-point capability between the CPU and the GPU is due to the GPU's design; the GPU is specialized for compute-intensive, highly parallel computation and designed so that more transistors are devoted to data processing rather than data caching and flow control.

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations (SIMD) with high arithmetic intensity, which is the ratio of arithmetic operations to memory operations. Following the SIMD paradigm, there is a lower requirement for flow control while memory access latency can be hidden with calculations instead of big data caches, because the code is executed on many data elements and has high arithmetic intensity,

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

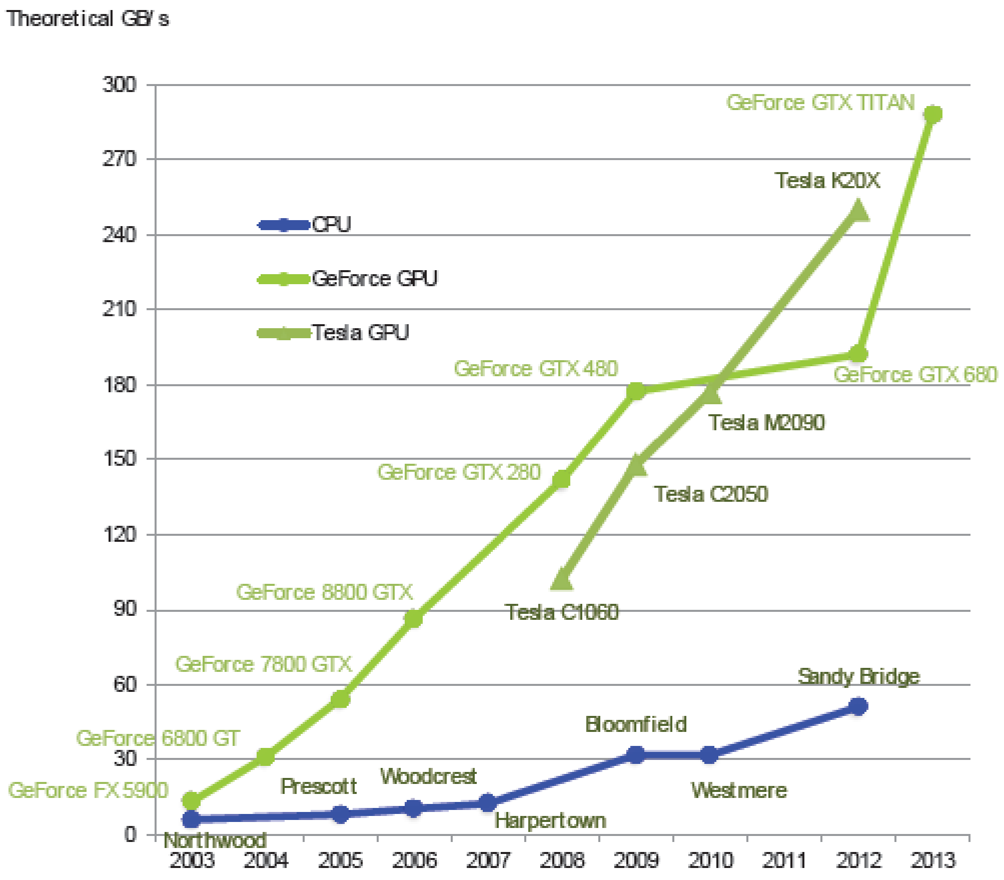


Figure 7.2 - GPU vs CPU memory bandwidth

The advent of multi-core CPUs and manycore GPUs means that mainstream processor chips are now parallel systems with their parallelism to continue scaling with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. It features three key abstractions:

- A hierarchy of thread groups
- A hierarchy of shared memories and

- Barrier synchronization

These abstractions are simply exposed to the programmer as a minimal set of language extensions and provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors and only the runtime system needs to know the physical multiprocessor count.

A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

7.2 CUDA THREADS

The GPU is following a variation of the SIMD paradigm, applying 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. It generally results from a forking execution into two or more concurrently running tasks. 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. All threads generated by a kernel define a grid and are organized in blocks. A grid consists of a number of blocks (all equal in size), and each block consists of a number of threads, as shown in Figure 7.3.

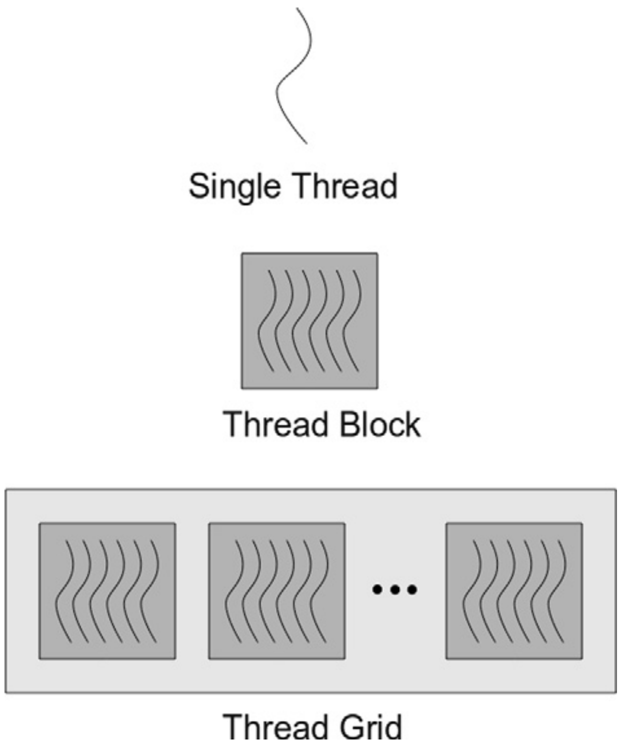


Figure 7.3 - Thread organization

Another type of thread grouping are warps. Warps are the units of thread scheduling in SMPs. Only one warp can be executed by a SMP at any given time. The number of threads in a warp is specific to the particular hardware implementation – it depends on how many threads the available hardware can process at the same time. The purpose of warps is to ensure high hardware utilization. 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 workload at all times in spite of the long-latency operations. The CUDA Programming Guide recommends that the number of threads per block should be chosen as a multiple of the warp size or better yet, a multiple of double the warp size from the viewpoint of performance [81].

7.3 CUDA MEMORY

CUDA devices have a variety of different memories that can be utilized by programmers in order to achieve high performance, as seen in Figure 7.4. 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 and off-chip. Constant memory provides interaction with the host too, but the device is only allowed to read from it and not write to it. However, it provides faster and more parallel data access paths for CUDA kernel execution than the global memory. Graphics processors are also equipped with texture memory which is used to accelerate frequently performed operations. CUDA allows the programmers to use some of the added capabilities of the separate texture unit hardware. Texture memory also provides a way to interact with the display capabilities of the GPU. There are also other types of memories, namely registers and shared memories, which cannot be accessed by the host. Data in these memories can be accessed in a highly parallel manner, but, due to their small size, specific attention is required not to exceed their limited capacities. 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. If the registers size is insufficient, then the data spills into local memory which is also private for each thread but is significantly slower.

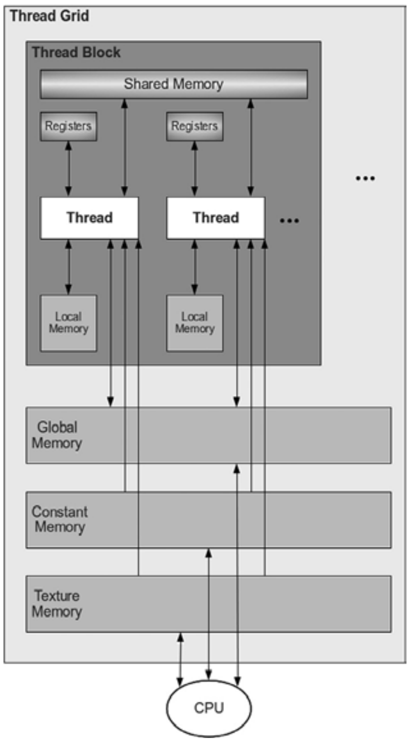


Figure 7.4 - Visual representation of CUDA memory model and scope

This is why overflowing the registers needs to be avoided. Moreover, shared memories are allocated to thread blocks instead of single threads, which allows all threads in a block to access variables in the shared memory locations allocated specifically for that block. Shared memories are as fast as registers while also allowing cooperation between threads of the same block.

7.4 BEST PRACTICES

As mentioned above, threads are grouped together as thread blocks, so that each block of threads is executed on the same SMP. Threads in the same block can communicate through the very fast shared memory. Threads in different blocks can communicate through the device memory, which is a lot slower than shared memory. For the aforementioned reason, device global memory access should be minimized and when it is required, it should be coalesced to attain high performance. Performance tuning requires taking into account kernel sizes, memory access timing, sizes of on-chip memory (small) as well as global memory on the GPU.

Memory coalescing is possible if consecutive threads access consecutive memory addresses. When utilizing memory coalescing, memory access by consecutive threads in a warp is combined by the hardware into several wide memory accesses. By organizing the memory properly, it allows the device to access the memory in a few coalesced reads/writes instead of many scattered (albeit simultaneous) accesses [81].

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 massing 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 [82]. The global memory's long access latency can be circumvented by proper utilization of the shared memory. When many threads need the same input, the corresponding data can be placed in the shared memory from where it can be accessed by threads in an efficient manner. Thus, a large number of global memory accesses can be avoided, leading to a significant increase in performance. This type of memory handling is called locality.

7.5 HYBRID CPU-GPU IMPLEMENTATION OF DDM

The Dual DDM FETI solver has been implemented in hybrid CPU–GPU workstations with the purpose of exploiting all available processing power and CPU 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. In particular, the 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 evaluation of vector d from Eq.(4.5.10), since it requires the strategy for the solution of the local subdomain problems and the amount of 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 CPU and GPU.

7.5.1 THE CHOLESKY DIRECT SOLVER

The Cholesky direct solver for computing the solution of Ku=f comprises the following steps:

- Factorization of matrix K to the form $K = LDL^T$
- Forward substitution so that $Lx_1 = f \Leftrightarrow x_1 = L^{-1}f$ and trivial solution of $Dx_2 = x_1 \Leftrightarrow x_2 = D^{-1}x_1$ since D is diagonal.
- Backward substitution so that $L^T u = x_2 \Leftrightarrow u = L^{-T}x_2$.

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:

$$D_j = K_{kk} - \sum_{k=1}^{j-1} L_{jk}^2 D_k \tag{7.5.1}$$

$$L_{ij} = \frac{1}{D_j}\left(K_{ij} - \sum_{k=1}^{j-1} L_{jk} L_{jk} D_k \right) \tag{7.5.2}$$

where the indices 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. (4.3.11) with the lumped type preconditioner of

Eq. (4.5.21), 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.

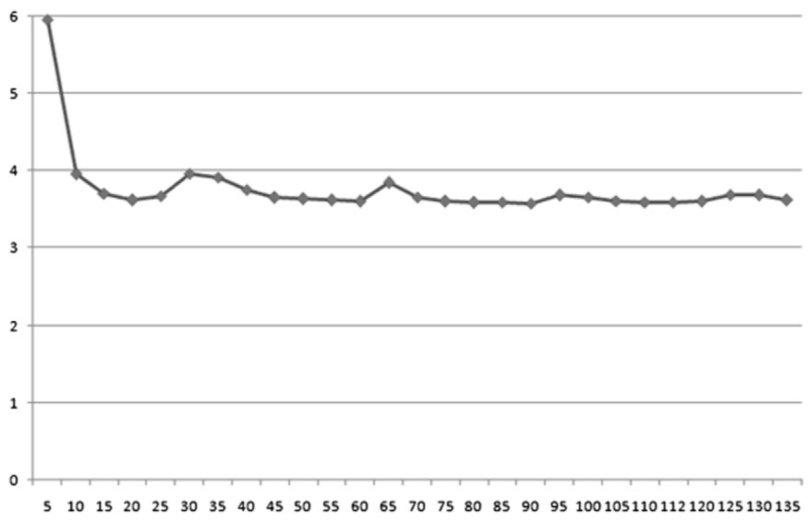


Figure 7.5 - Time in ms for factorizing a subdomain kernel. Horizontal axis represents the simultaneous factorizations computed at the GPU in parallel.

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 Figure 7.5, 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.

7.5.2 THE SOLUTION AT THE PROJECTION STEP

The projection matrix–vector multiplication encountered in Eqs. (4.3.10) and (4.3.13) involves the solution of

$$G^T G x_1 = x_2 \Leftrightarrow x_1 = \left(G^T G\right)^{-1} x_2 \tag{7.5.3}$$

at the initialization step and at each PCPG iteration, respectively, where x_1, x_2 are temporary vectors. This solution is usually performed with a direct solver since the order of the coefficient matrix GTG 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 performthis 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. (4.3.12) is applied with search vectors computed in previous PCPG iterations as well. This implementation is impractical when applied to the full problem $Ku = f$ due to excessive storage requirements. However, this methodology can be efficiently utilized for the projection step, where the size of the $G^T G$ 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.

7.5.3 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 (Eq. (4.3.12)), 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]+b[i]. On the other hand, the summation process is not that trivial.

On a sequential processor, the summation operation would be implemented by writing a simple loop with a single accumulator variable to construct the sum of all elements in sequence. On a parallel machine, using a single accumulator variable would create a global serialization point and lead to very poor performance. In order to overcome this problem, a parallel reduction strategy is implemented where each parallel thread sums a fixed-length sub-sequence of the input. Then, these partial sums are gathered by summing pairs of partial sums in parallel. Each step of this pair-wise summation divides the number of partial sums by half and ultimately produces the final sum after log2N steps as shown in Figure 7.6.

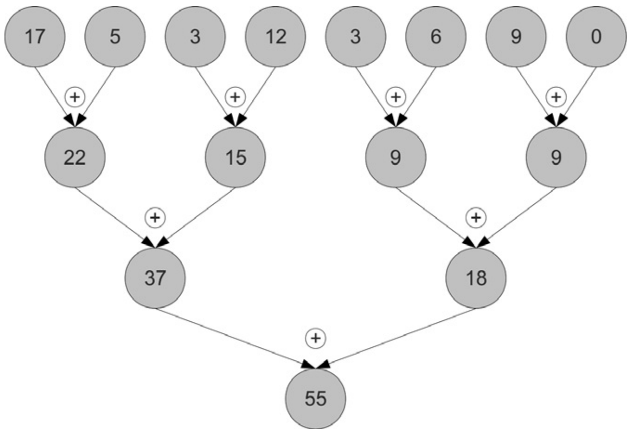


Figure 7.6 - Parallel summation using a tree-like structure

The dot product algorithm implemented in this work features a subsequence of length one which means that each thread is instructed to load one element of the input sequence. At the end of the reduction, the first thread of the block (thread 0) holds the sum of all elements initially loaded by the threads of this block. This is achieved in parallel by summing values in a tree-like pattern since the loop in the implemented kernel implicitly builds a summation tree over the input elements. The action of this loop for the simple case of a block of eight threads is illustrated in Figure 7.7 and each step of the loop corresponds to each successive level of the diagram with the arrow edges indicating from where partial sums are being read. In order to calculate the dot product, the partial sums of all the blocks in the grid must also be added.

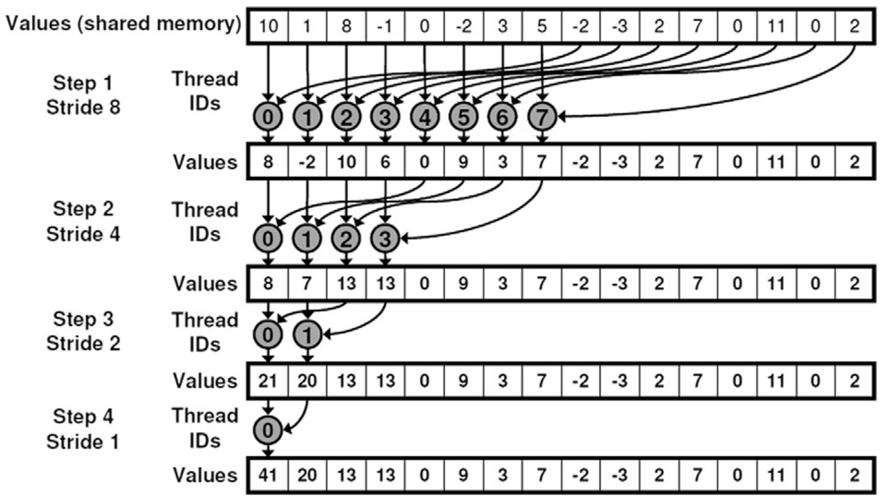


Figure 7.7 - Successive steps for summation in a simple case of an eight-threaded block

In this work this is achieved by having each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until the sequence is reduced to a single value. A more attractive alternative supported by the Tesla architecture and not by the GPUs used in this work, is to use atomicAdd(), an efficient atomic read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

7.5.4 SPARSE MATRIX-VECTOR MULTIPLICATION

At every PCPG iteration, the preconditioning step (Eq. (4.3.11)) 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 (4.3.6) of the PCG algorithm and in the solution of the projection step of Eq. (7.5.3) 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 [83]. 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 indices and data are accessed contiguously, therefore overcoming the principal deficiency of the approaches documented in [83]. 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.

7.6 DYNAMIC LOAD BALANCING

7.6.1 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 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 Figure 7.8. Thus, both CPU and GPU are constantly busy with calculations until the queue is emptied.

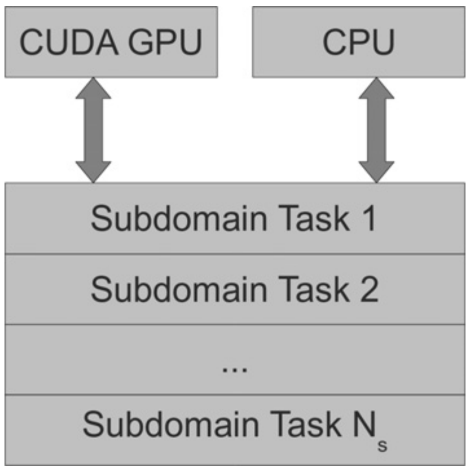


Figure 7.8 - The task queue contains numerical computations to be performed by available resources.

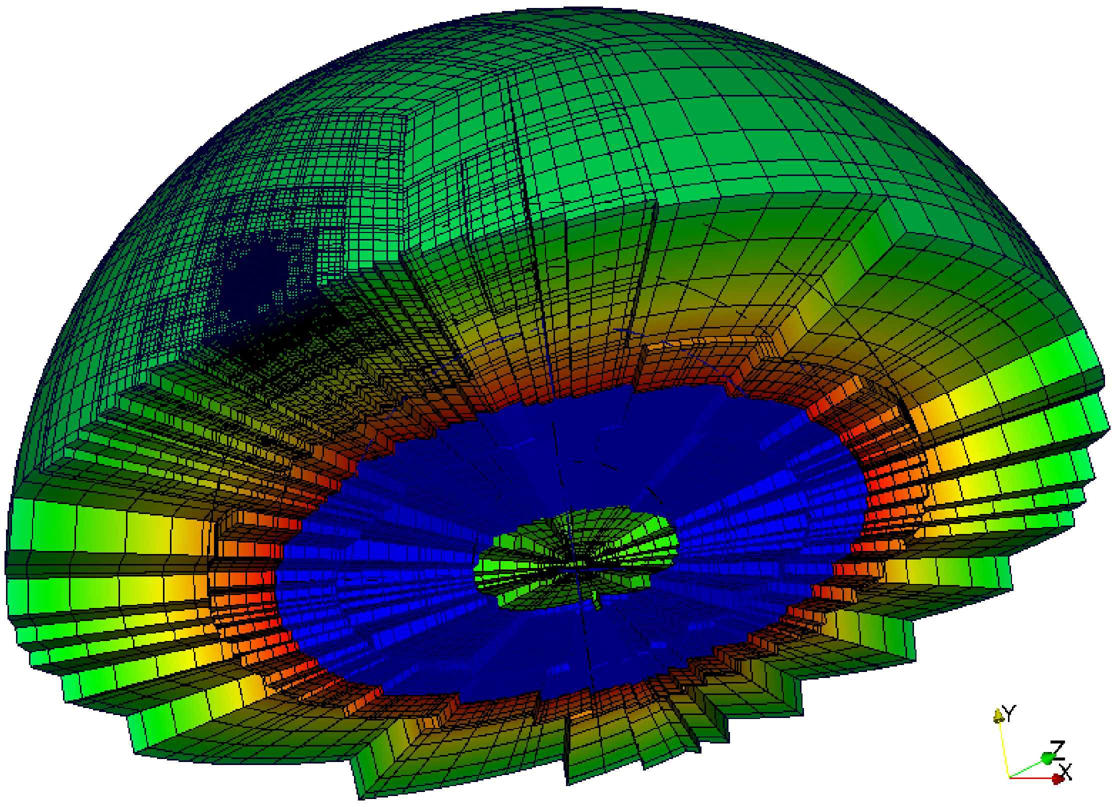
For the case of the PCG solver with diagonal preconditioner, the most time consuming operation is the sparse matrix–vector multiplication (SpMV) of Eq. (4.3.6) between the subdomain stiffness matrices and the corresponding search vectors. The same SpMV operation is required for the solution of the preconditioning step in Eq. (4.3.11) of the PCPG algorithm, while a similar operation is performed during the solution of the projection step of Eq. (7.5.3) 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.

7.6.2 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 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.


Y
Z
X

8.1 X-DDM-S AND X-DDM-P NUMERICAL RESULTS

The section presents a set of numerical examples that demonstrate the numerical efficiency of the proposed variants P-DDM-P, D-DDM-P, P-DDM-S and D-DDM-S methods described in chapter 4. In order to assess the efficiency of these methods, their performance is compared in two test cases. The first test case is a cubic soil consolidation problem subjected to a surface step load. The domain is discretized with 8 node hexahedral finite elements (2nd order quadrature) with 3 d.o.f. per node for the soil skeleton and 1 d.o.f. for the pore pressure. This problem is solved using the monolithic **u**-*p* formulation presented in Chapter 2. The boundary conditions of this test case along with the loading are shown in the 2D cut of Figure 8.1.

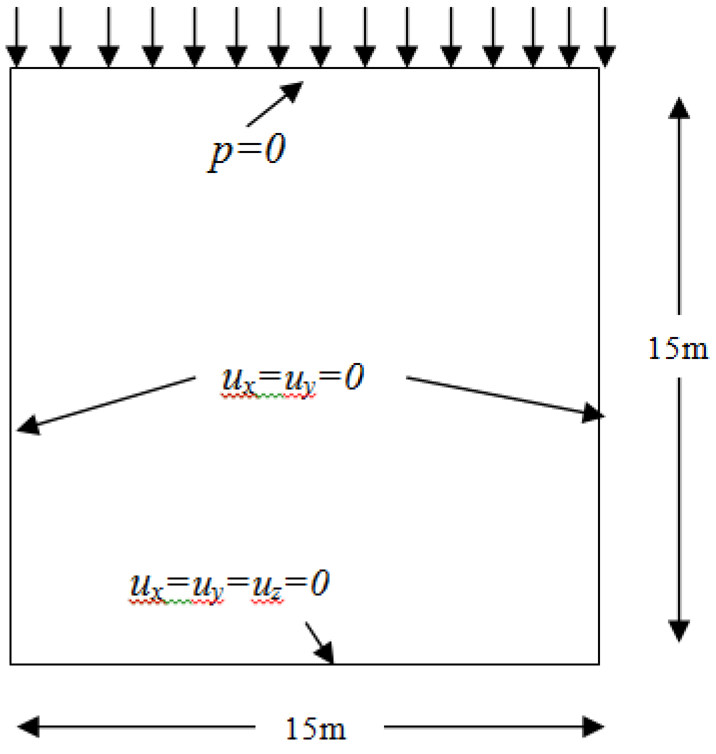


Figure 8.1 - Test case 1: A 2D cut showing boundary and loading conditions

The resulting linear system to be solved at each time increment has 115,320 d.o.f. and, in order to investigate the scalability of the various methods, a parametric analysis was carried out with respect to the number of subdomains. The solution was obtained for the first time increment of the Newmark algorithm and with number of subdomains ranging from 45 to 300. Two characteristic subdivisions are shown in Figure 8.2. The time step for this test case was 10^{-1} seconds.

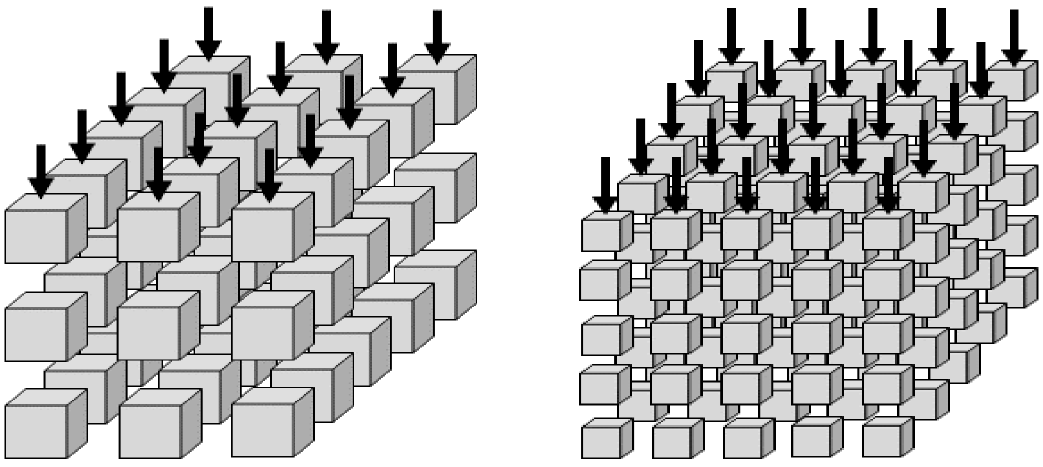


Figure 8.2 - Partitioning in 45 and 125 subdomains

The computer system used for both test cases is an Intel Pentium 4 531 3GHz workstation with 1MB L2 cache equipped with 8GB of DDR2/667 memory which is enough for performing these calculations without disk caching.

Young modulus of soil	3.9MPa
Poisson’s ratio	0.3
Void ratio	0.455
Permeability	$1.2 \cdot 10^{-8}$
Saturation	100%
Density of soil	1.6t/m^3
Bulk modulus of water	2.2GPa

Table 8.1 - Material properties of the soil for test cases 1 & 2

Figure 8.3 and Figure 8.4 depict the required number of iterations and the computing time to reach a solution tolerance of 10^{-4} of the DDM considered for the first time step of the

Newmark time integration algorithm and for different number of subdomains. The convergence criterion used is $\left\|\mathbf{f} - \mathbf{K}\mathbf{u}^{\mathbf{k}}\right\| / \left\|\mathbf{f}\right\|$ where **f** is the force vector, **K** is the stiffness matrix and $\mathbf{u}^{\mathbf{k}}$ is the unknowns vector of iteration k. The initialization time required for the computation of the projection matrix P (Eq. (4.6.3)) and for the subdomain factorization of the coefficient matrix is shown in Figure 8.5.

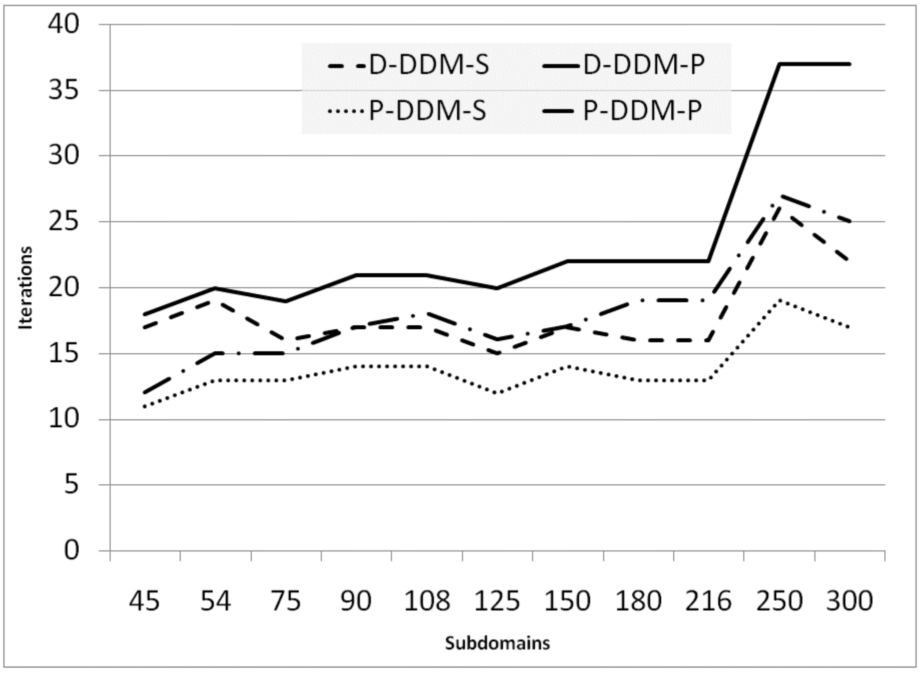


Figure 8.3 - Test case 1: Number of iterations for each time step for different number of subdomains

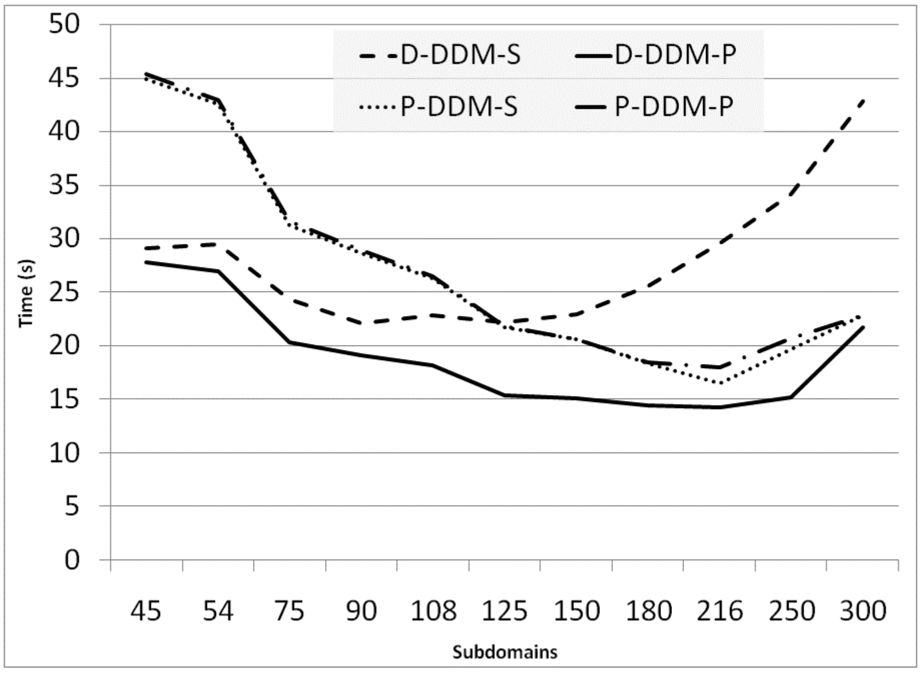


Figure 8.4 - Test case 1: Computation time in seconds per iteration for different number of subdomains

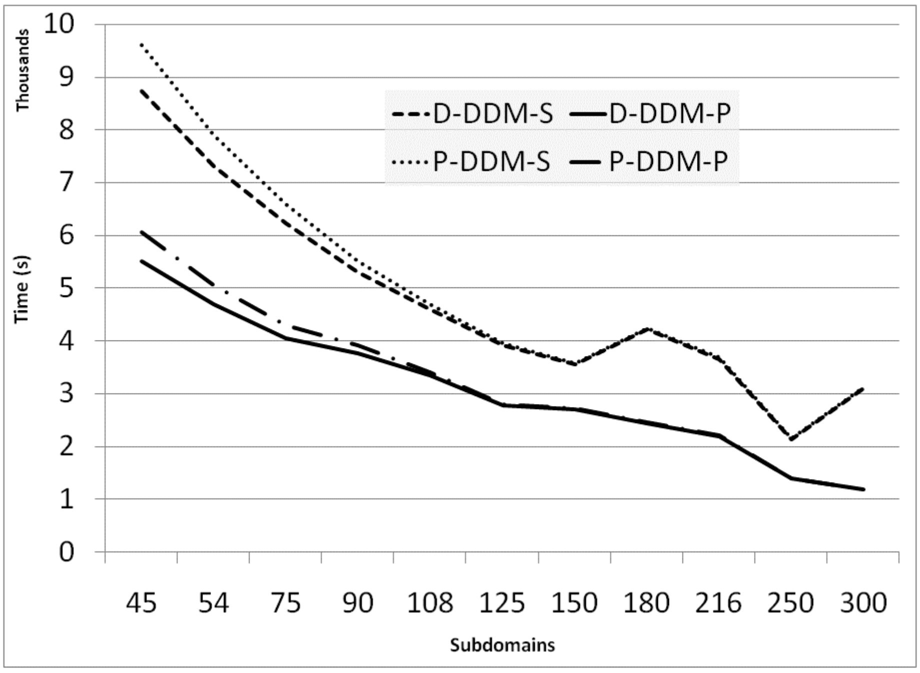
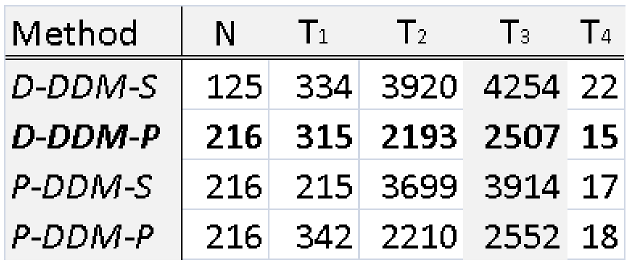


Figure 8.5 - Test case 1: Initialization time in seconds for different number of subdomains

The overall performance of the methods for the optimum number of subdomains is shown in Table 8.2. N is the optimum number of subdomains, T_1 stands for the required time in seconds for the first time step, T_2 corresponds to the initialization time, T_3 is the total time and T_4 denotes the time required per DDM iteration. Finally, Figure 8.6 demonstrates the best overall performance of the methods for one time step of the Newmark time integration algorithm with $\Delta t=10^{-2}$ and convergence tolerance of the DDM equal to 10^{-4}.



Method	N	T_1	T_2	T_3	T_4
D-DDM-S	125	334	3920	4254	22
D-DDM-P	**216**	**315**	**2193**	**2507**	**15**
P-DDM-S	216	215	3699	3914	17
P-DDM-P	216	342	2210	2552	18

Table 8.2 - Test case 1: Performance of the methods

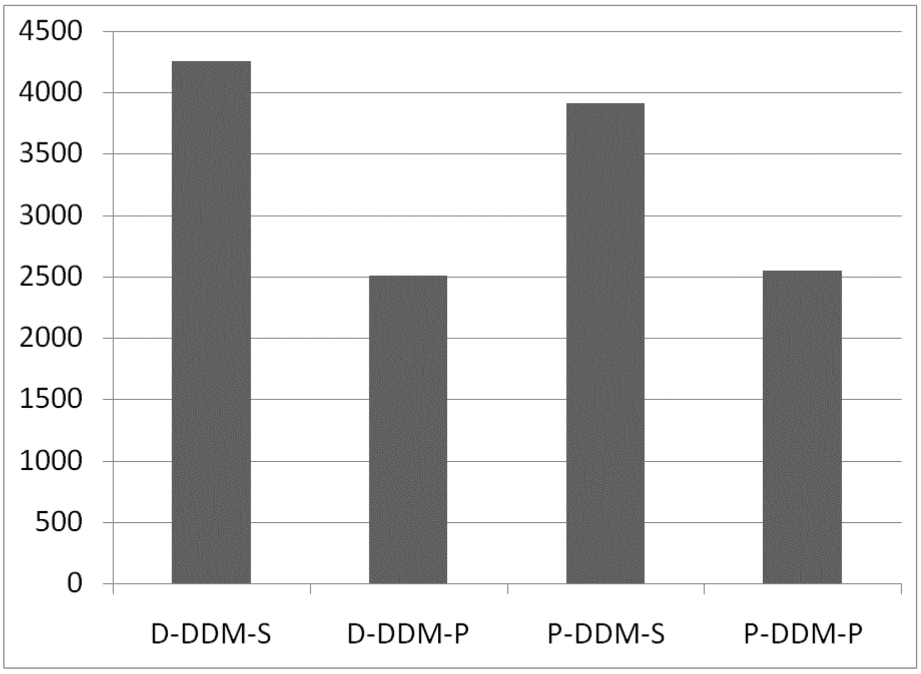


Figure 8.6 - Test case 1: Overall performance of the methods

The second test case is a soil foundation problem. The boundary conditions and the loading are shown in Figure 8.7. The domain is discretized with 8-node hexahedral elements, as in test case 1, resulting in 272,160 d.o.f. The domain was subdivided in 200 subdomains and the analysis was carried out for a total time span of 3 seconds with a time step of 10^{-2} seconds.

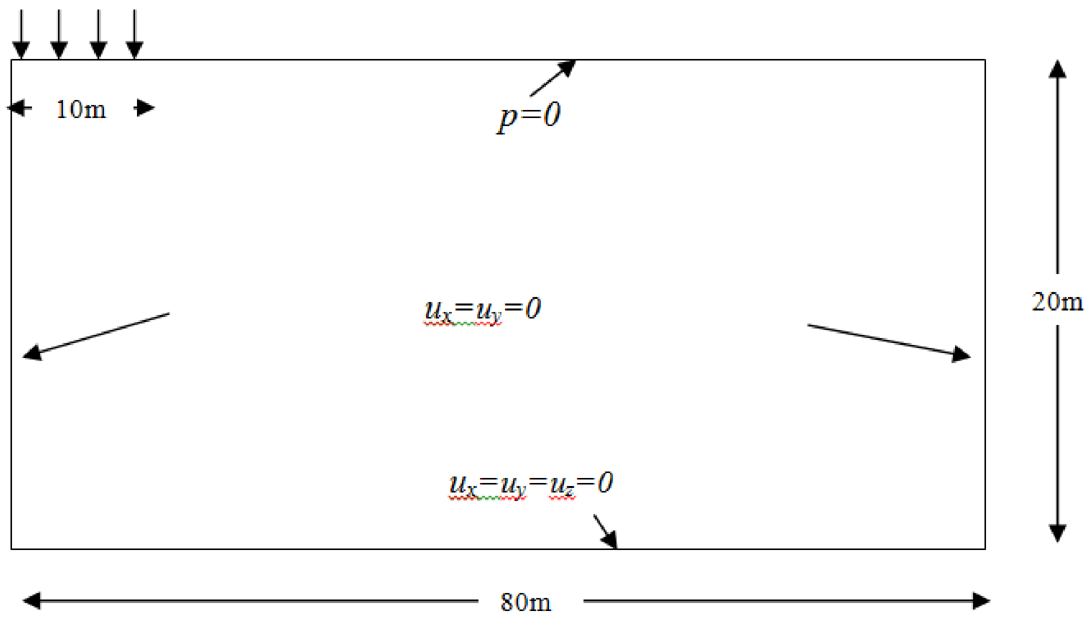


Figure 8.7 - Test case 2: A 2D cut showing boundary and loading conditions

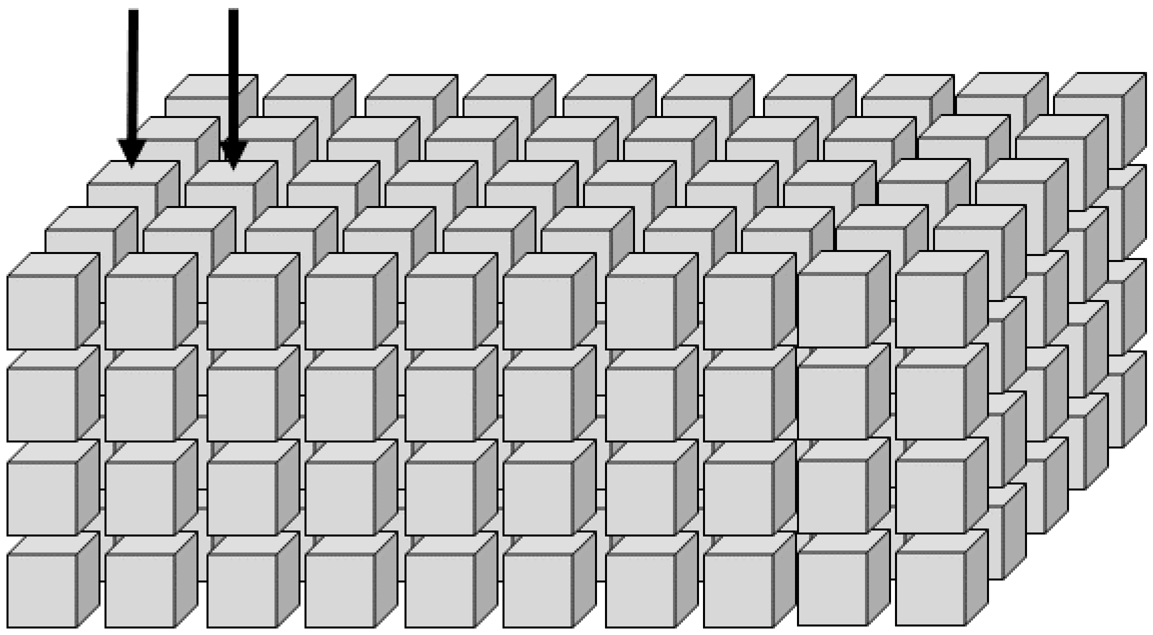


Figure 8.8 - Test case 2: Partitioning of the domain in 200 subdomains

The performance of the methods for this test case is presented in Table 8.3. N1 and N2 stand for the number of DDM iterations for the first time step of the Newmark time integration algorithm and for the solution of the total problem with 3 seconds duration, respectively, T1 corresponds to the initialization time in seconds, T2 is the total time and T3 denotes the time required per DDM iteration. The convergence tolerance of the DDM methods was set to 10^{-4}. The overall performance is also displayed graphically in Figure 8.9.

Method	N_1	N_2	T_1	T_2	T_3
D-DDM-S	9	519	7645	49510	81
D-DDM-P	**9**	**549**	**3073**	**31049**	**51**
P-DDM-S	7	507	7736	37287	58
P-DDM-P	7	531	3104	33447	57

Table 8.3 - Test case 2: Performance of the methods

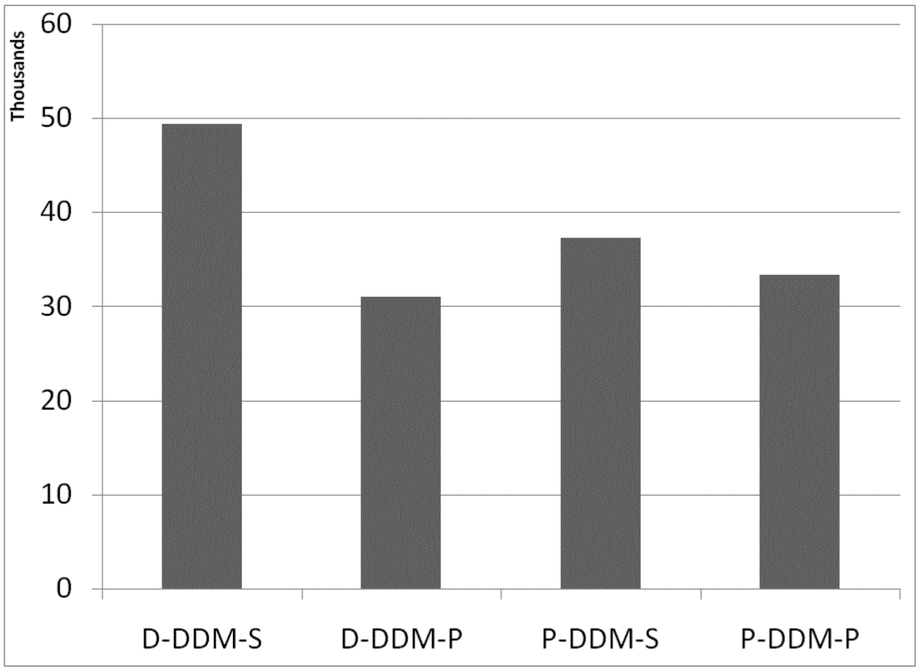


Figure 8.9 - Test case 2: Overall performance of the methods for the optimum number of subdomains

8.2 PERFORMANCE OF D-DDM-P IN HYBRID CPU/GPU ENVIRONMENT

This section provides a set of numerical examples that demonstrate the efficiency of the domain decomposition solvers implemented in GPU environments as presented on Chapter 7, using the dual method shown in section 4.6.5. Their performance is demonstrated in parametric studies of 3D linear porous media problems. Specifically, the cubic soil consolidation problem of section 8.1 is revisited and is solved in a hybrid CPU/GPU environment. The workstation used consists of an Intel Core 2 Quad Q6600 2.4 GHz, which has 4 physical cores/4 logical cores and 8 MB L2 cache, 3 GB RAM and one NVIDIA GTX285 GPU equipped with 1 GB GDDR3 memory. This configuration is adequate for performing all finite element calculations for this example in double precision without disk caching.

Number of subdomains	dof	Subdomain problems		Projection step problem	
		Iterations without	Iterations with	Iterations without	Iterations with
		Re-orthogonalization		Re-orthogonalization	
45	135	95	8	11	1
54	162	89	7	13	2
75	225	77	6	18	2
90	270	71	6	21	3
108	324	65	5	25	3
125	375	61	5	29	4
150	450	56	4	35	4
180	540	52	4	42	5
216	648	48	4	50	6
250	750	45	4	58	7
300	900	42	3	70	9

Table 8.4 - Example 1: Performance of PCG with and without re-orthogonalization

Figure 8.11a shows the resulting dof of each subdomain and of the corresponding interface problem for different number of subdomains. The required number of PCPG iterations for the solution of the interface problem, with Cholesky and PCG solvers for the subdomain problems, are presented in Figure 8.11b. In both PCG and PCPG algorithms, the convergence accuracy is e=10^{-4}. The results demonstrate the numerical scalability of the FETI method with PCPG iterations remaining almost constant regardless of the size of the interface problem.

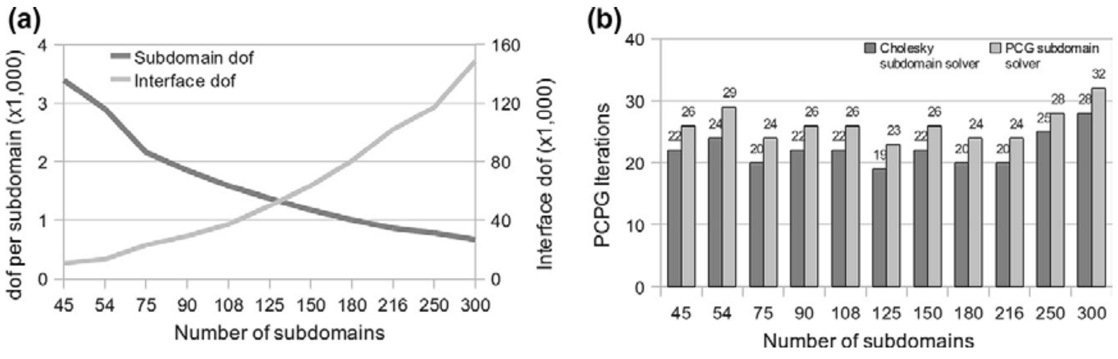


Figure 8.10 - Example 1: (a) Subdomain and interface dof for different number of subdomains. (b) Iteration numbers for the PCPG solution of the interface problem with Cholesky and PCG subdomain solvers.

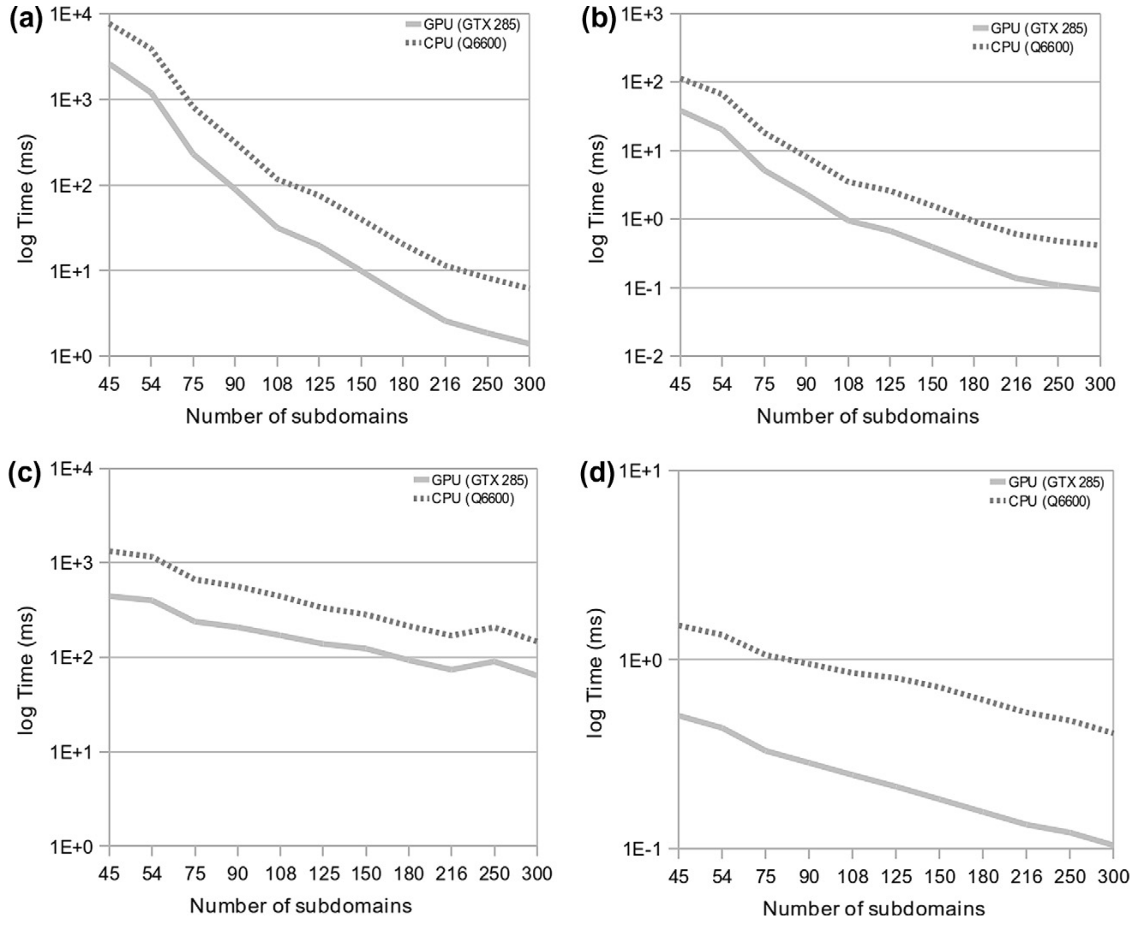


Figure 8.11 - Example 1: Computing time per subdomain. (a) Cholesky Factorization. (b) Forward/backward substitutions. (c) PCG solution. (d) SpMV multiplication

The convergence behavior of PCG for the solution of the subdomain problems and of the projection step, with and without reorthogonalization for treating the repeated solutions in the course of the PCPG iterations, is shown in Table 8.4. The solution accuracy of the projection step is increased to $e=10^{-7}$ since it affects the convergence properties of PCPG. It can be seen that the number of iterations is reduced by approximately one order of magnitude with the re-orthogonalization procedure.

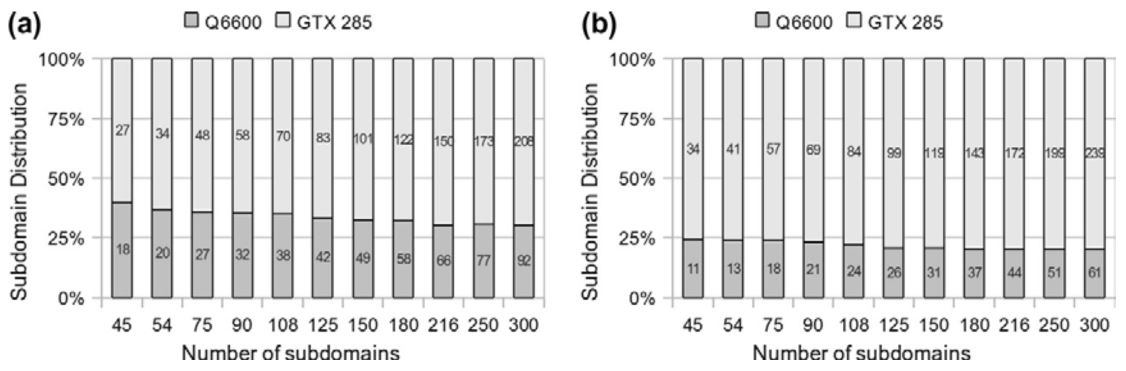


Figure 8.12 - Example 1: Optimum subdomain distribution between CPU and GPU. (a) Factorization and forward/backward substitutions of the Cholesky subdomain solver. (b) SpMV multiplications and PCG subdomain solver.

Figure 8.12 presents the computing time per subdomain required for the Cholesky factorization, the forward/backward substitutions and the PCG solution, as well as for performing one SpMV multiplication, for different number of subdomains. It can be seen that the required time of GTX285 is always faster that the corresponding time of Q6600 and follows the same trend in all four types of computations.

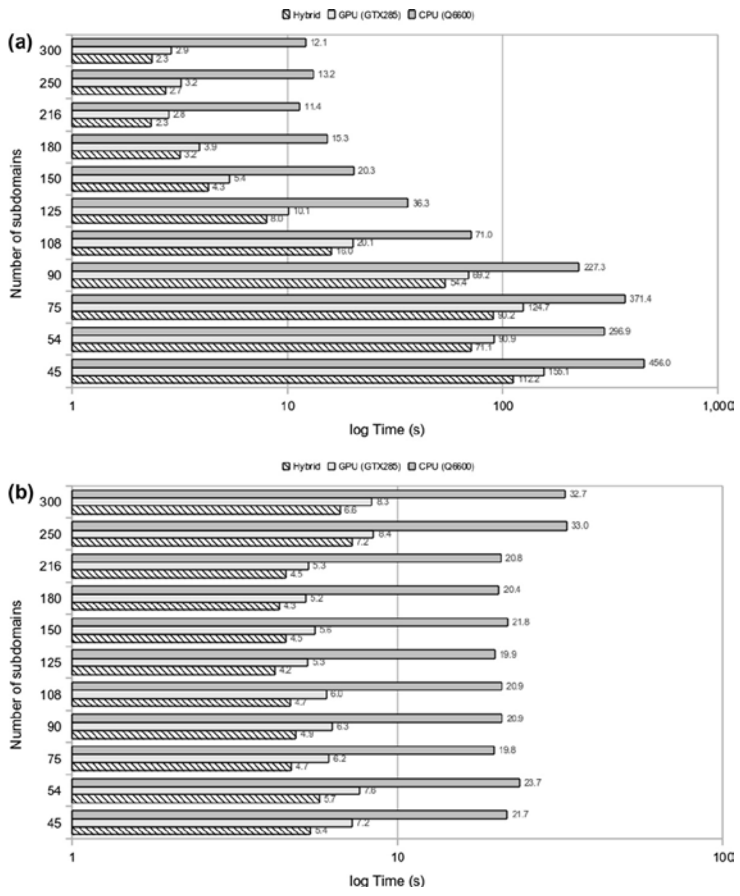


Figure 8.13 - Example 1: Total solution time of FETI for the Hybrid, GPU (GTX285) and CPU (Q6600) cases. (a) Cholesky subdomain solver (b) PCG subdomain solver

Figure 8.13 shows the optimum subdomain distribution between CPU and GPU, as the result of the dynamic load balancing implemented in this work, in order to keep both computing components of the workstation busy during the solution. Figure 8.13a corresponds to the load balance for the Cholesky solution of the subdomain problems for different number of subdomains, while Figure 8.13b depicts the optimum subdomain distribution for performing the SpMV multiplications of the preconditioning step of PCPG and of the PCG solution of the subdomain problems. The percentage for the optimum distribution between Q6600 and GTX285 is about 35%–75% for the Cholesky solver and 20%–80% for the SpMV multiplications.

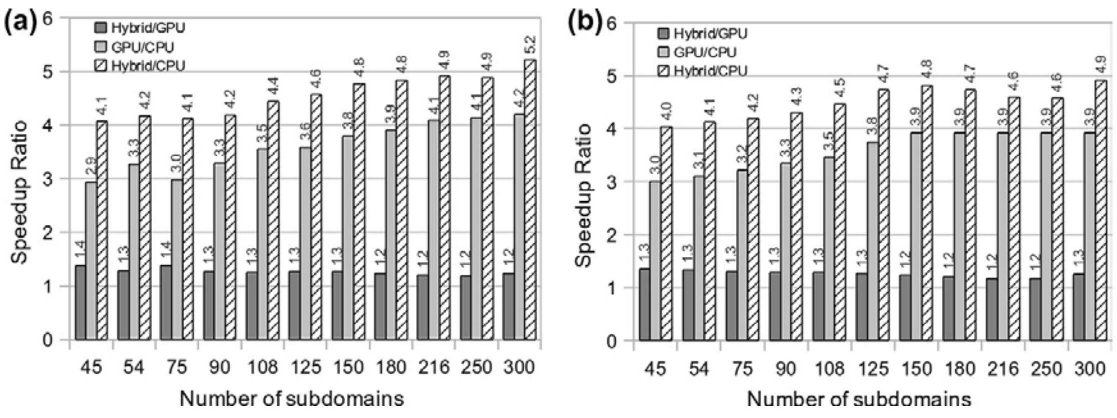


Figure 8.14 - Example 1: Performance speedup ratios for different combinations of CPU (Q6600) and GPU (GTX 285). (a) Cholesky subdomain solver. (b) PCG subdomain solver.

The performance of FETI with the Cholesky and PCG solvers for the subdomain problems is depicted in Figure 8.14a and b, respectively, for three cases: Q6600-only, GTX285-only and hybrid Q6600/GTX285. The optimum performance for all cases was achieved in the range of 180 subdomains which corresponds to a subdomain size of approximately 1000 dof and to an interface problem with approximately 80,000 dof.

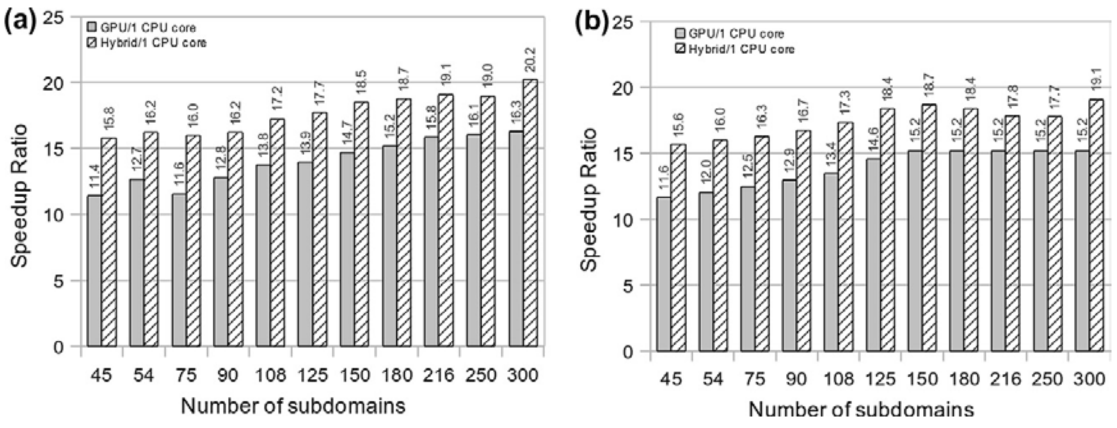


Figure 8.15 - Example 1: Performance speedup ratios per CPU core for different combinations of CPU (Q6600) and GPU (GTX 285). (a) Cholesky subdomain solver. (b) PCG subdomain solver.

A more illustrative indication of the performance of the method on the three workstation configurations used for this example is depicted in Figure 8.15 and Figure 8.16 where the relative performance speedup ratios are presented. Figure 8.15 shows the speedup ratios of GPU vs CPU, Hybrid vs GPU and Hybrid vs CPU, while in Figure 8.16, the corresponding speedup ratios are presented with respect to 1 CPU core, for the two versions of FETI. It can be seen that the hybrid implementation achieves speedups ranging from 4.1x to 5.2x compared to the CPU, depending on the number of subdomains, while the corresponding speedups of hybrid vs GPU are around 1.3x. These speedups are almost quadrupled when compared to 1 CPU core as indicated in Figure 8.16.

8.3 STOCHASTIC SOLVER RESULTS

This section provides a set of numerical examples for stochastic finite element and reliability analysis that demonstrate the efficiency of the solution methods presented in Section 4.7. For the case of Gaussian fields we will examine the performance of SSFEM-PCG-B and SSFEM-PCG-S. For log-normal fields we will test the performance of SSFEM-PCG-B, SSFEM-PCG-BF, SSFEM-PCG-S, SSFEM-PCG-SF and their variants with caching SSFEM-PCG-BC, SSFEM-PCG-BFC, SSFEM-PCG-SC, SSFEM-PCG-SFC, respectively. For the case of the MC method, we will examine the performance of the MC-PCG-Skyline, MC-PCG-FETI and MC-PCG-PFETI solvers. The computer platform used is an Intel Core i7 X980 with 6 physical cores at 3.33GHz with 24GB of RAM.

In order to assess the computational efficiency of the MC and SSFEM methods for the analysis of systems with uncertain properties, a soil cube of 10 x 10 x 20 meters under load in the center of its upper surface due to a large footing was considered, resulting to a finite element mesh of 10k dof approximately, as shown in Figure 8.27a. This mesh is decomposed into 16 subdomains featuring a cubic aspect ratio each, as shown in Figure 8.27b. A multi-parametric study has been carried out first, considering both Gaussian and log-normal stochastic fields.

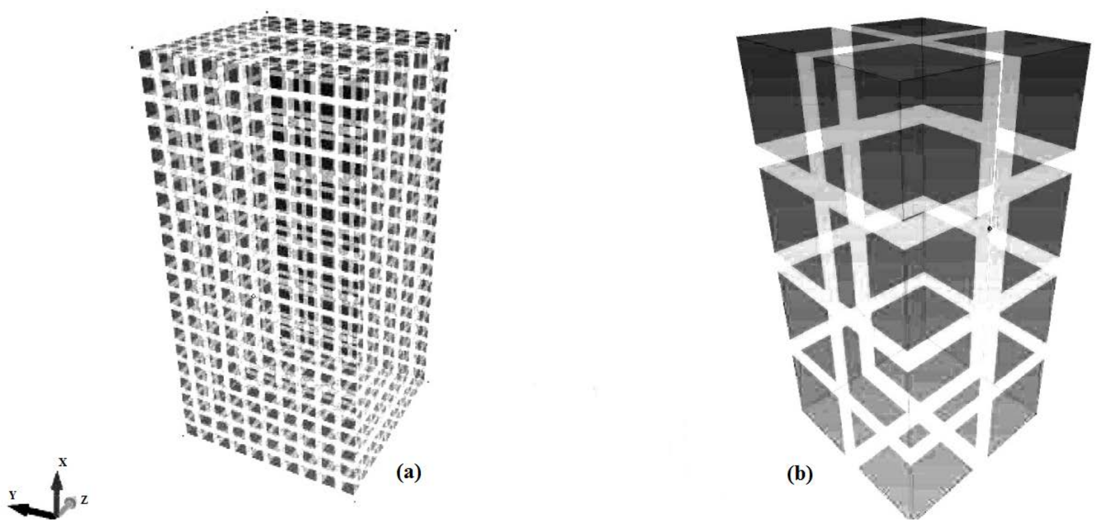


Figure 8.16 - Domain decomposition of a quarter of the deterministic soil problem with 10k dof. (a) Element mesh (b) Subdomain mesh

One dimensional stochastic fields are used to describe the spatial variation of the system's modulus of elasticity E around its mean as $E = E_0 (1 + f(x))$, where E_0 is the mean value of E and f(x) a zero mean homogeneous stochastic field with standard deviation σE. The covariance function of the random field f(x) is assumed to be exponential:

$$C(x_1, x_2) = \sigma_E{}^2 e^{-\frac{|\Delta x|}{b}} \quad (62) \qquad (8.3.1)$$

where Δx = x2-x1. Three test cases regarding coefficients σE are examined: (a) σE = 15% (Gaussian), (b) σE = 30% (log-normal) and (c) σE = 80% (lognormal). Moreover, four correlation length values are assumed: (a) b= 0.1a, (b) b= 1a, (c) b= 10a and (d) b= 100a, with a being the height of the cube.

For all these test cases, two separate problems are addressed: evaluation of the second moments of the response field and a reliability analysis with 0.1% probability of failure. Setting a = 20m, the correlation lengths that were examined for this example were 2m, 20m, 200m and 2000m.

8.3.1 SOLVER ASSESSMENT PROCEDURE

In order to set an objective basis for assessing the computational performance of the numerical algorithms discussed, a parametric study was conducted, regarding different values for standard deviation σE and correlation length b. For the computation of the second moments of the response field, the following procedure was followed:

Step 1: A series of Monte Carlo analyses of 100k simulations was carried out, using M = 1 as the order of the KL expansion, in order to estimate the necessary number of simulations for a convergence error of less than 1% for each value of σE and b examined. This error is computed as the normalized difference of the COV (%) at each simulation with respect to the COV (%) computed at the end of the 100k simulations.

Step 2: Assuming that the convergence behavior of the previous step remains invariant for increasing M, another series of Monte Carlo analyses was carried out, in the range of M = 2 to M = 12, in order to estimate the appropriate order of the KL expansion for a convergence error of less than 1%. In this case an "exact" solution was assumed at M = 12 in order to compute the relative error (%) for different M.

Step 3: Using the results of step 2, the same procedure as in step 2 was carried out performing SSFEM analyses, in order to estimate the appropriate order of the PC expansion required for convergence to the corresponding MC results.

Step 4: For the case of reliability analysis with 0.1% target probability of failure, the order of the PC expansion is being modified, with respect to step 3 (convergence in COV %), in order to reach a convergence error in the estimation of the probability of failure of less than 10%, compared to the corresponding MC results. The number of simulations for both MC and SSFEM is in this case 100k.

8.3.2 COMPUTATION OF THE SECOND MOMENTS OF THE RESPONSE FIELD

Figure 8.28, Figure 8.29 and Figure 8.30 show the convergence error for each field as per step 1 of the assessment procedure. Based on these figures, the number of simulations necessary for evaluating the second moments of the response field are shown in Table 8.6.

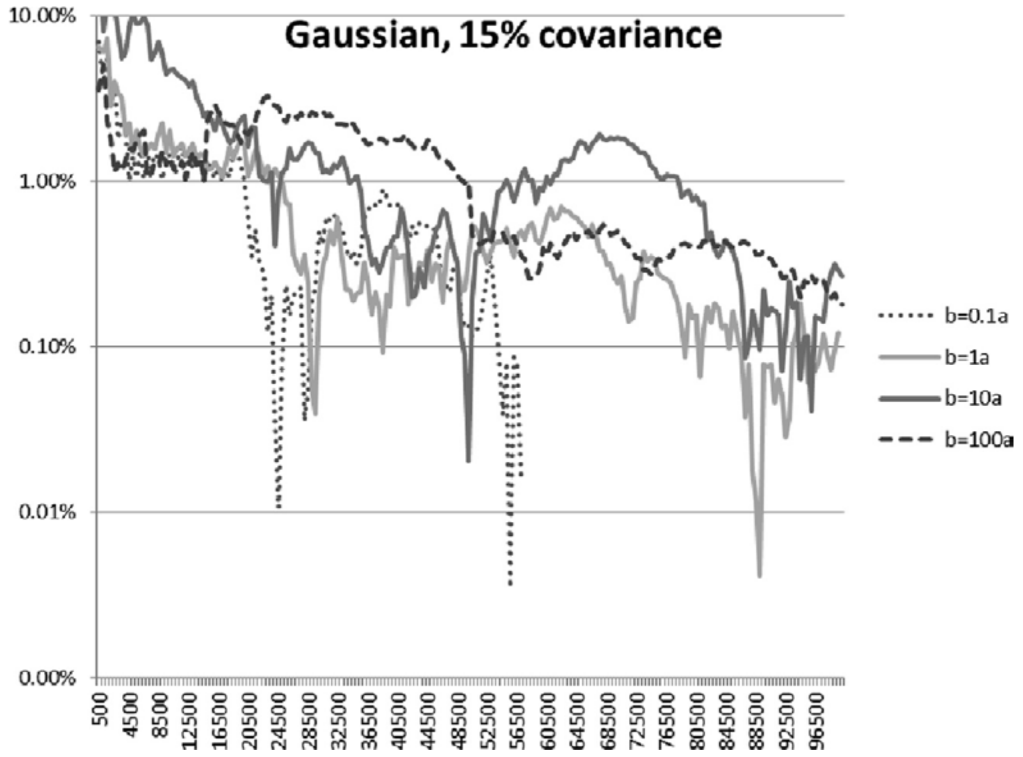


Figure 8.17 - Step 1: COV (%) convergence error of MC for the Gaussian field with σE = 15%

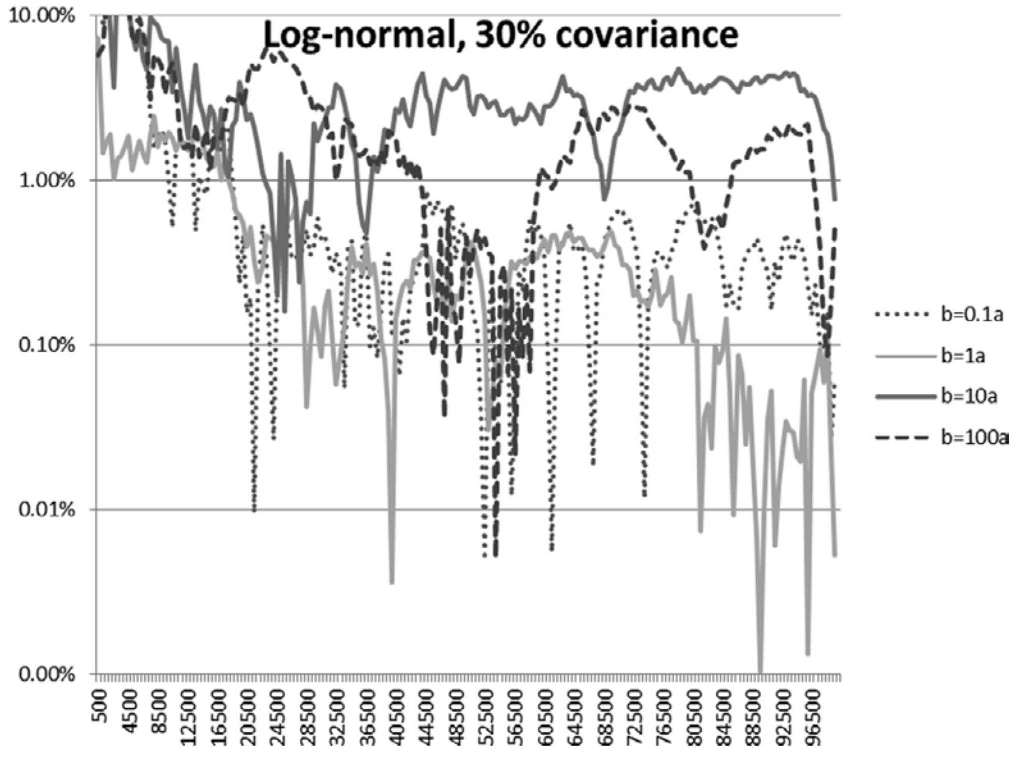


Figure 8.18 - Step 1: COV (%) convergence error of MC for the log-normal field with σE = 30%

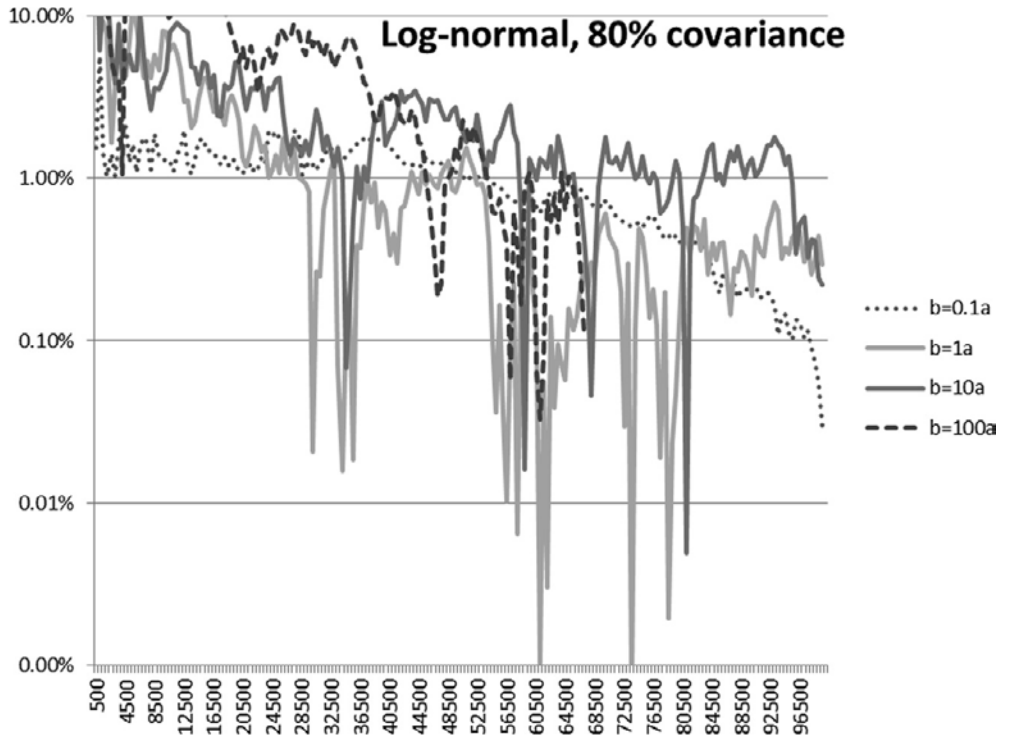


Figure 8.19 - Step 1: COV (%) convergence error of MC for the log-normal field with σE = 80%

Correlation length b	$\sigma_E = 15\%$	$\sigma_E = 30\%$	$\sigma_E = 80\%$
$0.1a$	20.000	10.000	53.000
$1a$	25.000	18.000	28.000
$10a$	23.000	23.000	34.000
$100a$	50.000	43.000	45.000

Table 8.5 - Required number of MC simulations for achieving a COV error less than 1%.

Correlation length b	$\sigma_E = 15\%$		$\sigma_E = 30\%$		$\sigma_E = 80\%$	
	M	Error (%)	M	Error (%)	M	Error (%)
$0.1a$	12	"exact"	10	0.43	4	0.75
$1a$	6	0.93	4	0.75	4	0.57
$10a$	2	0.36	2	0.85	4	0.26
$100a$	2	0.48	2	0.53	4	0.96

Table 8.6 - Step 2: COV (%) convergence errors for the various KL expansion orders

Following step 2, Figure 8.31, Figure 8.32 and Figure 8.33 show the convergence error for each field as per step 3 of the assessment procedure for the selection of PC expansion order (p) required for the SSFEM to converge at an error less than 1% using the KL expansion orders M shown in Table 8.7. This relative error is computed with respect to the corresponding MC simulations with the same parameter M.

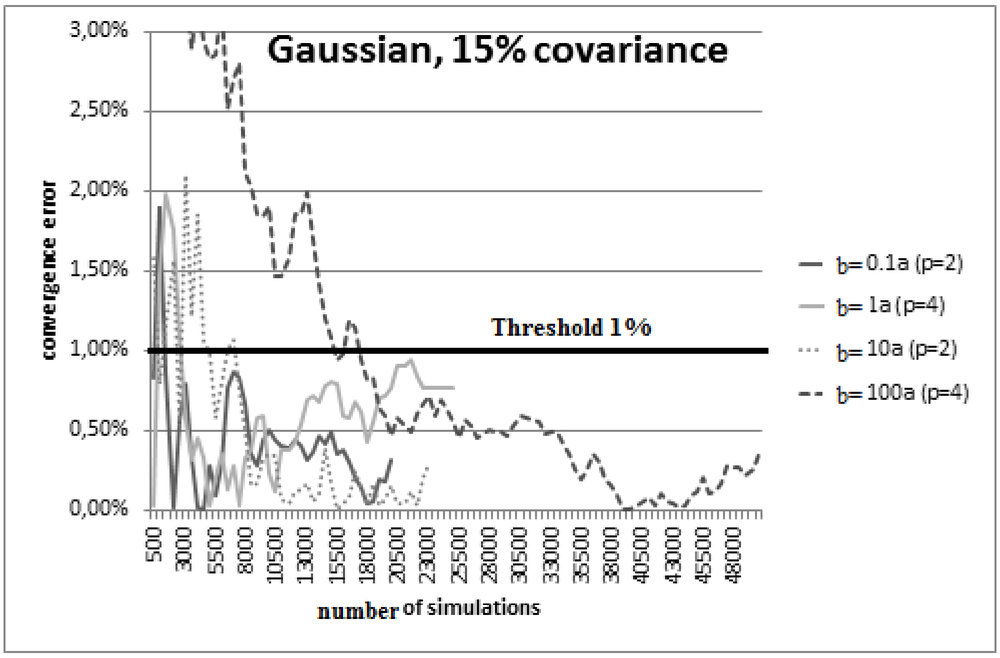


Figure 8.20 - Step 3: COV (%) convergence error of the SSFEM for the Gaussian field with σE=15% and p = 2,3,4.

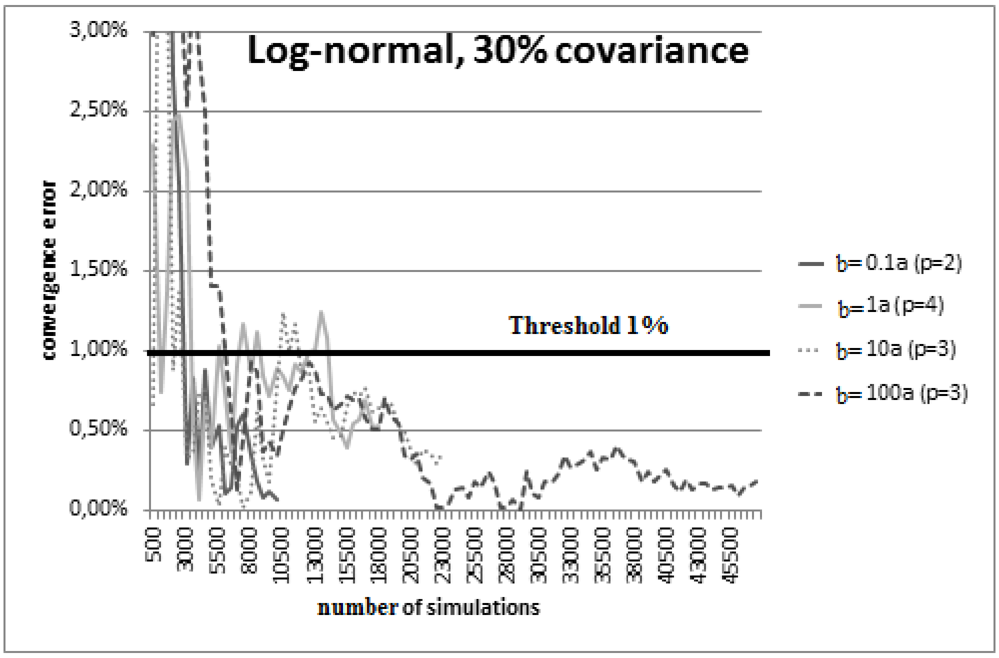


Figure 8.21 - Step 3: COV (%) convergence error of the SSFEM for the log-normal field with σE=30% and p = 2,3,4.

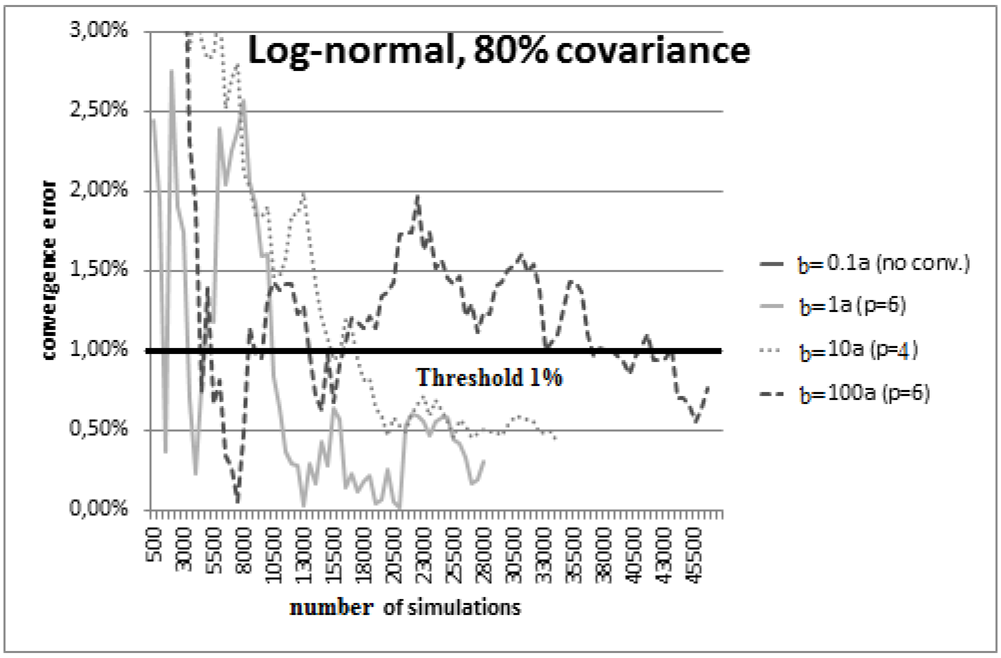


Figure 8.22 - Step 3: COV (%) convergence error of the SSFEM for the log-normal field with σE=80% and p = 2,3,4.

Figure 8.34, Figure 8.35 and Figure 8.36 depict some indicative graphs of the convergence behavior of the SSFEM in specific cases. Table 8.8 summarizes the convergence of the SSFEM (relative error %) with respect to MC, for all cases considered.

Correlation	$\sigma_E = 15\%$		$\sigma_E = 30\%$		$\sigma_E = 80\%$	
length b	p	Error (%)	p	Error (%)	p	Error (%)
$0.1a$	2	0.23	2	0.07	6	30.00
$1a$	4	0.09	4	0.69	6	0.52
$10a$	2	0.03	3	0.36	4	0.68
$100a$	4	0.36	3	0.74	6	0.88

Table 8.7 - Convergence errors for the SSFEM

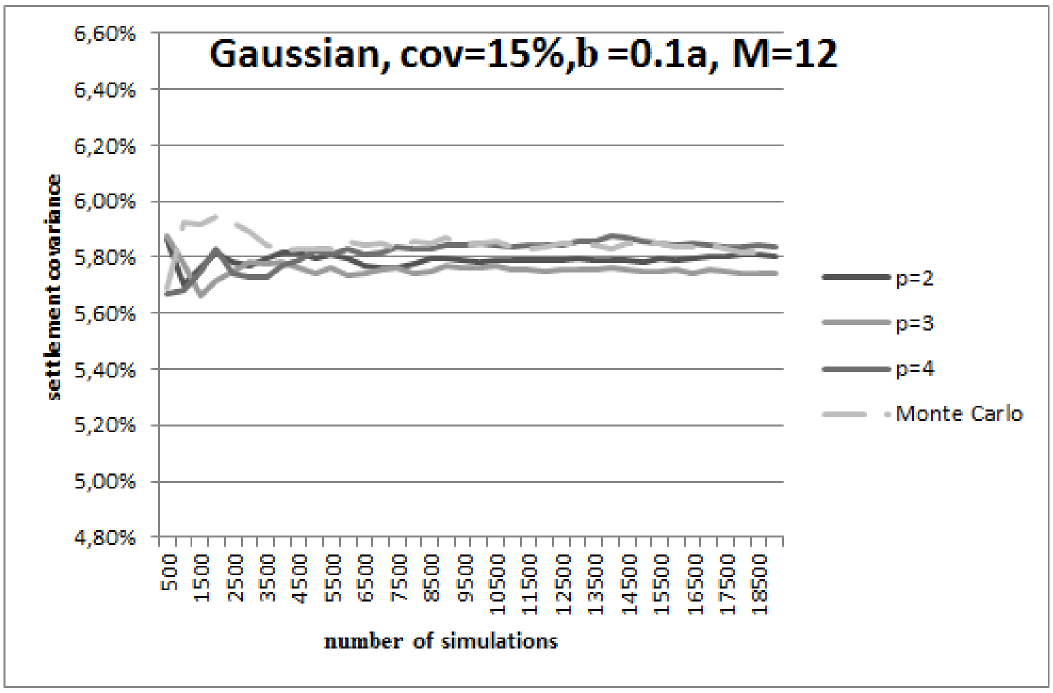


Figure 8.23 - Settlement covariance for σE = 15% , correlation length 2m, and M = 12 for MC and SSFEM.

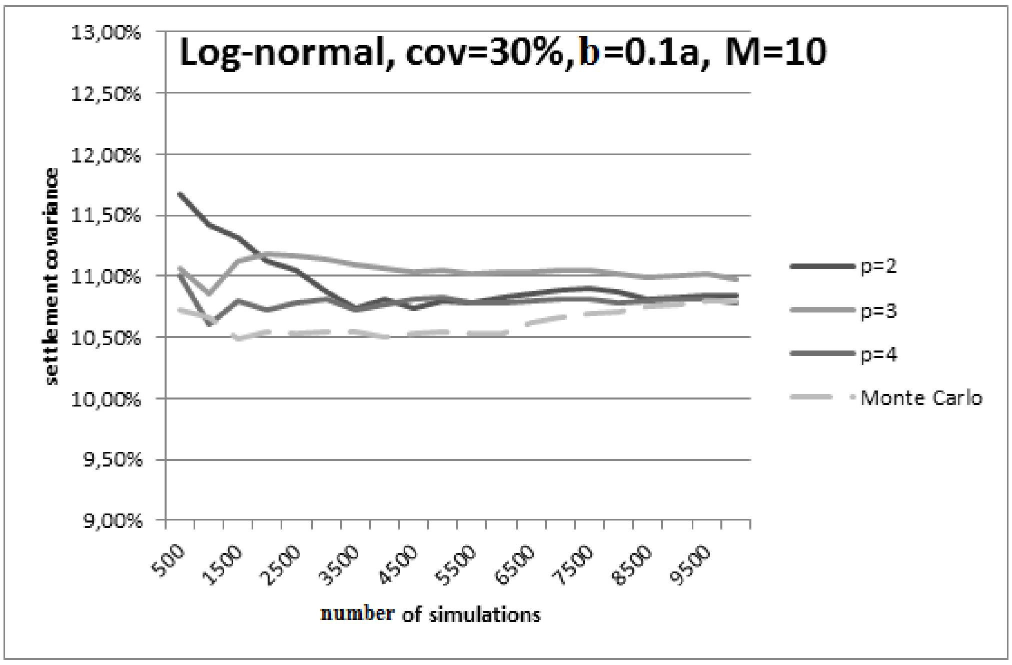


Figure 8.24 - Settlement covariance for σE = 30%, correlation length 2m, and M = 10 for MC and SSFEM

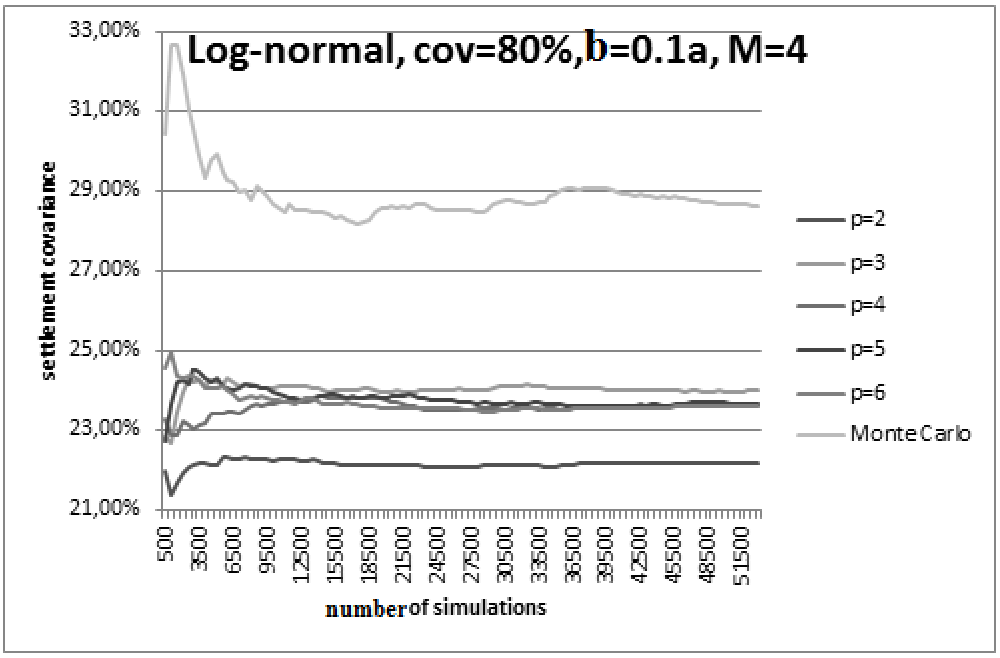


Figure 8.25 - Settlement covariance for σE = 80%, correlation length 2m and M = 4 for MC and SSFEM

It is worth noting that for the case of b=0.1a, the SSFEM failed to provide a solution within the acceptable error margin when compared to the MC solution. While increasing the p-order of the PC expansion, the SFFEM method was asymptotically converging to a solution which exhibited a 30% error when compared to the corresponding Monte Carlo solution.

8.3.3 RELIABILITY ANALYSIS

Utilizing the values of M obtained at step 2 of the solver assessment procedure for the computation of the second moments of the response field, we performed reliability analysis on the same test problem. Table 8.9 shows the settlement values which correspond to a probability of failure of 0.1%, as estimated by MC with 100k simulations, for various stochastic parameters (σE and b) considered. Table 8.10 shows the probability of failure for the corresponding limit state settlements of Table 8.9 using SSFEM with the KL and PC order expansions used for the second order moments analysis as shown in Table 8.7 and Table 8.8, respectively. The settlement values of Table 8.9 are used as reference values, i.e. as the limit states that correspond to a probability of failure 0.1% for all cases considered, while Table 8.11 shows the same probability of failure with the PC order expansion needed to reach almost the same accuracy with the "reference" MC solution. Values marked in bold correspond to analyses that needed an increase of the PC order expansion.

Correlation length b	MC		
	$\sigma_E = 15\%$	$\sigma_E = 30\%$	$\sigma_E = 80\%$
$0.1a$	0.019405	0.023468	0.076088
$1a$	0.024796	0.032947	0.118410
$10a$	0.028214	0.038950	0.166543
$100a$	0.029439	0.039841	0.168306

Table 8.8 - Settlements with 0.1% probability of failure

Correlation length b	SSFEM		
	$\sigma_E = 15\%$	$\sigma_E = 30\%$	$\sigma_E = 80\%$
$0.1a$	0.06 %	0.04 %	-
$1a$	0.09 %	0.09 %	0.07 %
$10a$	0.03 %	0.09 %	0.01 %
$100a$	0.07 %	0.10 %	0.10 %

Table 8.9 - Probability of failure as computed by SSFEM for the settlements of Table 8.9 for the limit states

Correlation length b	$\sigma_E = 15\%$		$\sigma_E = 30\%$		$\sigma_E = 80\%$	
	p	prob	p	prob	p	prob
$0.1a$	4	**0.09%**	**5**	**0.09%**	-	-
$1a$	4	0.09%	4	0.09%	**7**	0.09 %
$10a$	4	**0.09%**	3	0.09 %	**8**	**0.10%**
$100a$	**6**	**0.09%**	3	0.10 %	6	0.10 %

Table 8.10 - Probability of failure as computed by SSFEM for the settlements of Table 7 and the necessary PC order

For the case of b=0.1a and σE = 80%, a series of analyses were performed with various expansion orders M and p, going up to M = 12 and p = 6. However, SSFEM failed to converge to an acceptable solution resulting to a minimum convergence error of 20%.

8.3.4 PERFORMANCE OF THE PROPOSED SOLUTION PROCEDURES

Using all previous numerical data (number of simulations, KL expansion order and PC expansion order), a series of numerical tests were performed in order to assess the performance of the various solution techniques discussed and proposed in this work. For all cases considered the normalized solution accuracy was set to 10^{-7} while for the computation of the preconditioned residual vector, the required accuracy was set to 10^{-3}.

$\sigma_E = 15\%$	Correlation length b	$0.1a$	$1a$	$10a$	$100a$
	MC simulations	20.000	25.000	23.000	50.000
	PCG iterations	184.398	196.875	114.715	144.592
MC-PCG-Skyline	Time (s)-sequential	31.759	85.930	163.202	224.143
	Time (s)-parallel	4.670	12.637	24.000	32.962
MC-PCG-FETI	FETI iterations	455.016 (2.950.368)	134.198 (3.150.000)	22.015 (1.835.440)	39.060 (2.313.472)
	Time (s)-sequential	48.001	36.752	21.203	39.827
	Time (s)-parallel	7.059	5.405	3.118	5.857
MC-PCG-PFETI	PFETI iterations	425.281 (2.950.368)	124.133 (3.150.000)	20.157 (1.835.440)	35.761 (2.313.472)
	Time (s)-sequential	36.771	28.076	16.443	30.590
	Time (s)-parallel	5.407	4.129	2.418	4.498

Table 8.11 - Performance of the various MC-PCG-Skyline variants for the MC for evaluating the second moments of the response field for σE=15% in sequential and parallel implementation

Table 8.12, Table 8.13 and Table 8.14 show the performance of proposed MC-PCG-PFETI solver for the evaluation of the second order moments of the response field using the MC method, in comparison to MC-PCG-Skyline and MC-PCG-FETI and are visually depicted in Figure 8.37, Figure 8.38 and Figure 8.39. The PFETI and FETI iterations correspond to the sum of the PFETI and FETI iterations needed for all the MC simulations using the A-orthogonalization technique, while in parentheses the corresponding PFETI and FETI iterations without A-orthogonalization are given. These numbers show a drastic decrease of iterations ranging from one to two orders of magnitude as a result of the A-orthogonalization procedure.

Moreover, from these tables, it is evident that the PFETI variant outperforms the FETI one in all tests, showing a 1.25x speedup. This performance increase occurs for two reasons: (i) PFETI needs ~10% less iterations when compared to FETI. (ii) The cost for each reorthogonalization of the PFETI method is about 35% less when compared to the FETI method. This stems from the fact that the interface problem of the PFETI method is based on the boundary dof of each subdomain while the interface problem of the FETI method is based on the lagrange multipliers which, due to the existence of a considerable number of subdomains crosspoints, are significantly larger in quantity than the boundary dof.

$\sigma_E = 30\%$	Correlation length b:	$0.1a$	$1a$	$10a$	$100a$
	MC simulations	10.000	18.000	23.000	43.000
	PCG iterations	110.100	221.531	114.541	153.825
MC-PCG-Skyline	Time (s)-sequential	18.922	105.391	161.203	241.235
	Time (s)-parallel	2.783	15.499	23.706	35.476
MC-PCG-FETI	FETI iterations	314.475 (1.761.600)	107.198 (3.544.496)	23.442 (1.832.656)	37.087 (2.461.200)
	Time (s)-sequential	33.053	32.437	22.189	38.625
	Time (s)-parallel	4.861	4.770	3.263	5.680
MC-PCG-PFETI	PFETI iterations	294.235 (1.761.600)	99.478 (3.544.496)	21.777 (1.832.656)	34.375 (2.461.200)
	Time (s)-sequential	25.337	24.956	17.393	30.080
	Time (s)-parallel	3.726	3.670	2.558	4.423

Table 8.12 - Performance of the various MC-PCG-Skyline variants for the MC for evaluating the second moments of the response field for σE = 30% in sequential and parallel implementation

$\sigma_E = 80\%$	Correlation length:	$0.1a$	$1a$	$10a$	$100a$
	MC simulations	53.000	28.000	34.000	45.000
	PCG iterations	1.193.825	695.100	272.340	253.350
MC-PCG-Skyline	Time (s)-sequential	205.082	68.030	370.320	400.378
	Time (s) -parallel	30.159	10.004	54.459	58.879
MC-PCG-FETI	FETI iterations	3.413.444 (19.101.200)	1.624.400 (11.121.600)	72.507 (4.357.440)	56.799 (4.053.600)
	Time (s)-sequential	358.806	97.472	65.108	60.159
	Time (s)-parallel	52.765	14.334	9.575	8.847
MC-PCG-PFETI	PFETI iterations	3.265.860 (19.101.200)	1.530.760 (11.121.600)	67.320 (4.357.440)	52.650 (4.053.600)
	Time (s)-sequential	281.182	75.286	50.653	46.932
	Time (s) -parallel	41.350	11.071	7.449	6.902

Table 8.13 - Performance of the various MC-PCG-Skyline variants for the MC for evaluating the second moments of the response field for σE = 80% in sequential and parallel implementation

The Skyline variant seems to be more efficient for b=0.1a but this happens due to the relatively small size of the deterministic model. For large deterministic models, the Skyline variant is outperformed by domain decomposition methods, particularly in massively parallel computation environments.

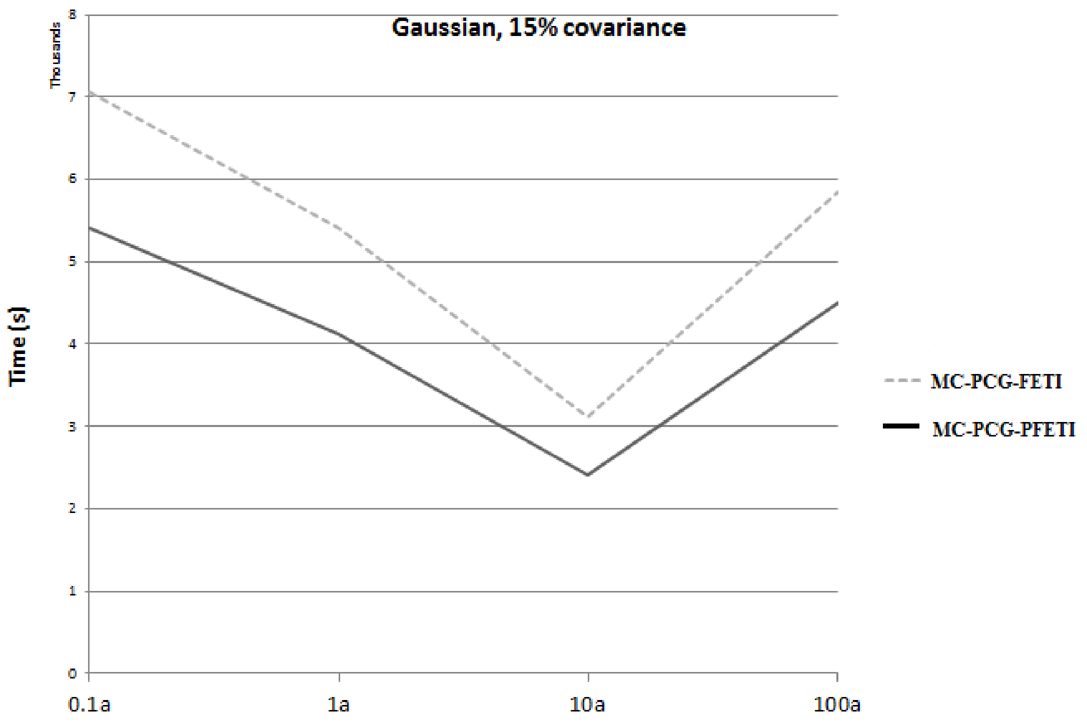


Figure 8.26 - Performance of the MC-PCG-PFETI and MC-PCG-FETI for Gaussian σE=15%

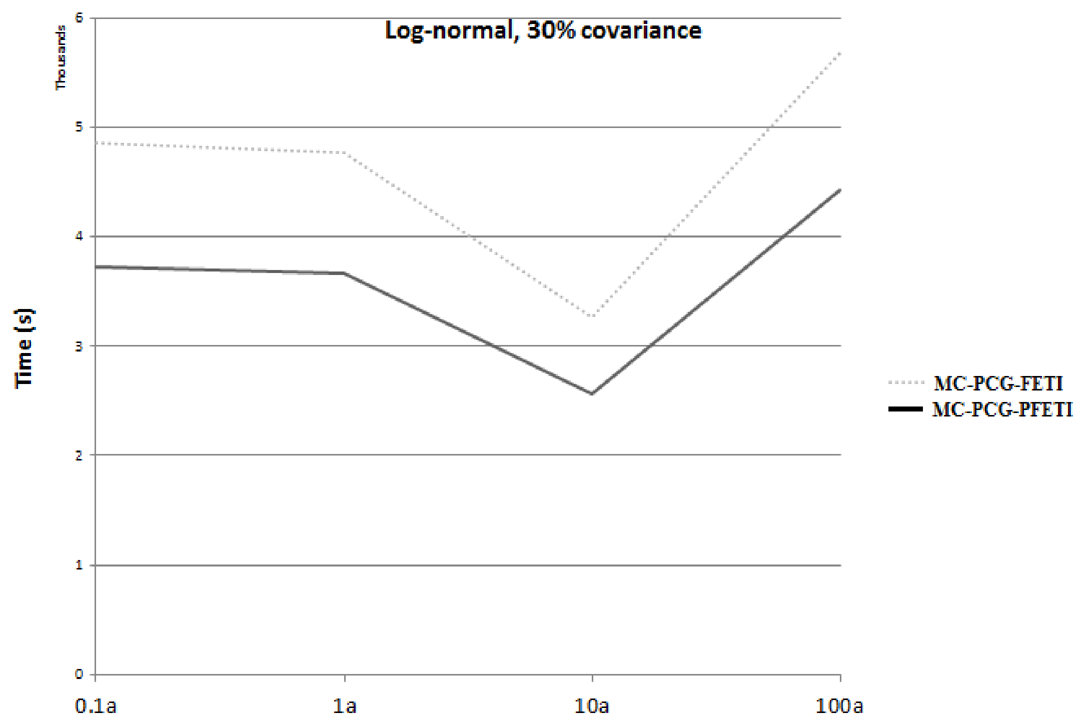


Figure 8.27 - Performance of the MC-PCG-PFETI and MC-PCG-FETI for lognormal σE = 30%

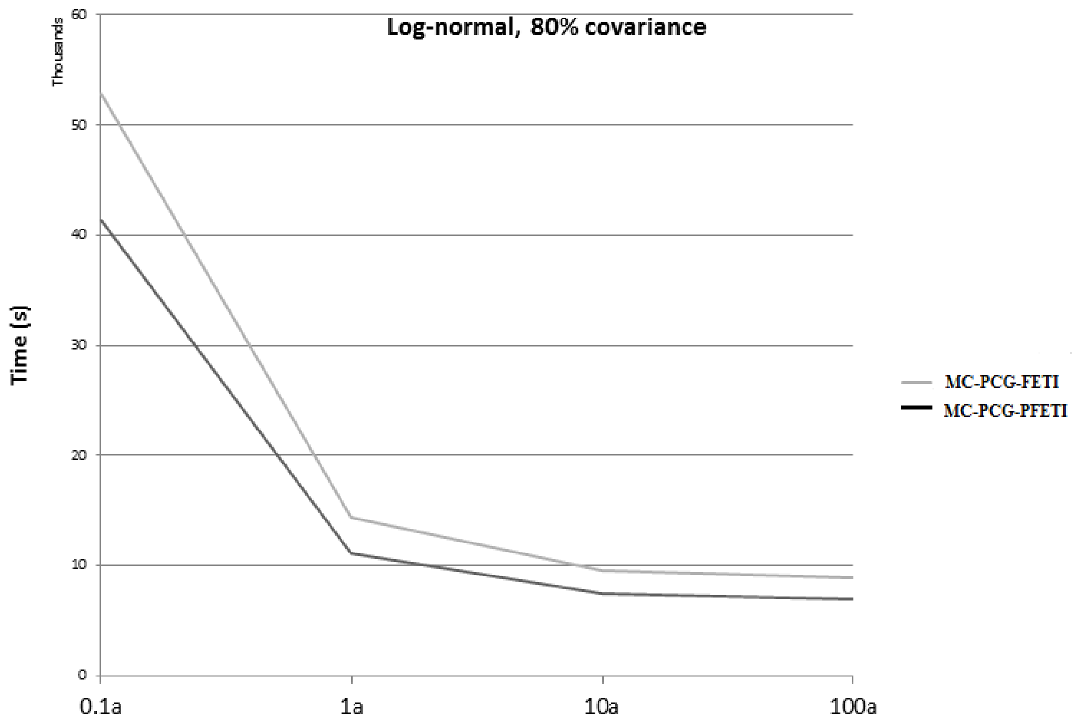


Figure 8.28 - Performance of the MC-PCG-PFETI and MC-PCG-FETI for lognormal σE = 80%

Table 8.15, Table 8.16 and Table 8.17 depict the performance of the proposed SSFEM solution methods. Table 8.15 shows the performance of SSFEM methods, using the information gathered from steps 1-3 with respect to the necessary number of simulations, KL expansion order (M) and PC expansion order (p) for the Gaussian case. For the Gaussian case, only the SSFEM-PCG-B and SSFEM-PCG-S solvers are used for the SSFEM. It can be seen that the proposed SSFEM-PCG-S outperforms SSFEM-PCG-B achieving a 2.8x speedup with respect to the SSFEM-PCG-B.

Gaussian $\sigma_E = 15\%$					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
SSFEM-PCG-B	MC Simulations	20.000	25.000	23.000	50.000
	PCG iterations	7	10	5	8
	PFETI iterations	650	702	96	123
	Total Time (s)-sequential	2.417	3.339	105	168
	Total Time (s)-parallel	422	586	18	29
SSFEM-PCG-S	PCG iterations	3	4	3	4
	PFETI iterations	377	269	74	74
	Total Time (s)-sequential	1.026	1.178	77	100
	Total Time (s)-parallel	179	204	13	17

Table 8.14 - Performance metrics for the Gaussian case (σE=15% covariance)

log-normal 30%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
SSFEM-PCG-B	MC Simulations	10.000	18.000	23.000	43.000
	PCG iterations	10	15	12	11
	PFETI iterations	551	367	120	98
	Total Time (s)-sequential	2.786	6.568	244	256
	Total Time (s)-parallel	488	1.149	43	43
SSFEM-PCG-BF	PCG iterations	10	17	13	11
	PFETI iterations	577	328	132	91
	Total Time (s)-sequential	2.930	7.191	260	273
	Total Time (s)-parallel	516	1.241	44	46
SSFEM-PCG-S	PCG iterations	4	5	4	4
	PFETI iterations	403	260	100	79
	Total Time (s)-sequential	1.872	4.317	172	142
	Total Time (s)-parallel	330	745	31	24
SSFEM-PCG-SF	PCG iterations	4	5	5	4
	PFETI iterations	577	263	108	85
	Total Time (s)-sequential	2.728	4.347	200	150
	Total Time (s)-parallel	480	752	35	25

Table 8.15 - Performance metrics for the log-normal case (σE = 30% covariance)

Table 8.16 shows performance metrics for the log-normal case with 30% covariance, where all solver variants are implemented within SSFEM. As in the case of the Gaussian field SSFEM-PCG-S outperforms SSFEM-PCG-B, achieving a 2.3x speedup when compared to the SSFEM-PCG-B.

log-normal 80%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
SSFEM-PCG-B	MC Simulations	53.000	28.000	34.000	45.000
	PCG iterations	48	89	33	58
	PFETI iterations	685	1.010	523	584
	Total Time (s)-sequential	266.702	272.836	14.171	321.161
	Total Time (s)-parallel	46.799	49.175	2.447	55.380
	Total Time cached(s)-sequential	112.142	114.721	9.443	134.402
	Total Time cached(s)-parallel	1988	2079	131	2393
SSFEM-PCG-BF	PCG iterations	47	117	47	82
	PFETI iterations	2.827	12.393	423	467
	Total Time (s)-sequential	273.786	280.083	19.072	452.675
	Total Time (s)-parallel	48.047	50.475	3.291	78.050
	Total Time cached(s)-sequential	12.246	12.563	1.237	1.885
	Total Time cached(s)-parallel	21.495	22.580	2.130	32.538
SSFEM-PCG-S	PCG iterations	16	186	36	59
	PFETI iterations	528	1.167	414	338
	Total Time (s)-sequential	177.970	368.937	28.790	651.255
	Total Time (s)-parallel	31.236	66.484	4.968	112.289
	Total Time cached(s)-sequential	74.930	154.271	18.473	276.205
	Total Time cached(s)-parallel	13.180	27.807	3.189	47.624
SSFEM-PCG-SF	PCG iterations	12	25	13	20
	PFETI iterations	461	449	273	228
	Total Time (s)-sequential	133.533	276.818	10.529	220.894
	Total Time (s)-parallel	23.430	49.887	1.818	38.096
	Total Time cached(s)-sequential	56.253	115.818	6.803	92.094
	Total Time cached(s)-parallel	9.879	2.087	1.175	15.888

Table 8.16 - Performance metrics for the log-normal case (σE = 80% covariance)

Table 8.17 presents the performance metrics for the log-normal case with 80% covariance. As previously, the SSFEM-PCG-S and SSFEM-PCG-SF variants outperform the SSFEM-PCG-B and SSFEM-PCG-BF methods, showing a speedup up to 2.8x. For this covariance of the log-normal case, the proposed caching scheme proves to be quite efficient, providing up to 3x speedup when compared to the corresponding uncached method.

Table 8.18, Table 8.19 and Table 8.20 compare the performance of the MC and SSFEM when using the most computationally efficient solution method for evaluating the second order moments of the response field. It can be seen that for the Gaussian input field that SSFEM outperforms Monte Carlo method. The same conclusion can be reached for the log-normal case with 30% covariance.

Gaussian 15%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
MC	PCG iterations	184.398	196.875	114.715	144.592
	PFETI iterations	425.281	124.133	20.157	35.761
	Time (s)-sequential	36.771	28.076	16.443	30.590
	Time (s)-parallel	5.407	4.129	2.418	4.498
SSFEM	PCG iterations	3	4	3	4
	PFETI iterations	377	269	74	74
	Time (s)-sequential	1.026	1.178	77	100
	Time (s)-parallel	179	204	13	17

Table 8.17 - Monte Carlo vs. SSFEM for the Gaussian case (15% covariance)

log-normal 30%					
Correlation length b:		$0.1a$	$1a$	$10a$	$100a$
MC	PCG iterations	110.100	221.531	114.541	153.825
	PFETI iterations	294.235	99.478	21.777	34.375
	Time (s)-sequential	25.337	24.956	17.393	30.080
	Time (s)-parallel	3.726	3.670	2.558	4.423
SSFEM	PCG iterations	4	5	4	4
	PFETI iterations	403	260	100	79
	Time (s)-sequential	1.568	2.884	162	132
	Time (s)-parallel	277	498	28	23

Table 8.18 - Monte Carlo vs. SSFEM for the log-normal case (30% covariance)

Table 8.20 shows performance metrics for the log-normal case with 80% covariance, where all solver variants are used for the SSFEM. In contrast to the log-normal case with 30% covariance, MC method outperforms SSFEM in all cases except for the b=10a correlation length. This is due to the small order of p = 4 required by the PC expansion, compared to the other cases which required an expansion of order p = 6. For the b= 0.1a case, SSFEM fails to converge.

Table 8.21, Table 8.22 and Table 8.23 present performance comparisons of the MC and SSFEM when using the most efficient solution method for carrying out a reliability analysis. It can be seen that for the Gaussian input field, SSFEM outperforms MC by more than 2 orders of magnitude, while for the log-normal input field with 30% covariance, SSFEM outperforms MC for all cases except for the case of 0.1a correlation length. However, for the log-normal input field with 80% covariance, SSFEM is inferior to the MC while being unable to converge for the case of 0.1a correlation length.

log-normal 80%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
MC	PCG iterations	1.194.328	695.100	272.340	253.350
	PFETI iterations	3.265.860	1.530.760	67.320	52.650
	Time (s)-sequential	281.182	75.286	50.653	46.932
	Time (s)-parallel	41.350	11.071	7.449	6.902
SSFEM	PCG iterations	-	89	13	20
	PFETI iterations	-	1.010	273	228
	Time (s)-sequential	-	114.721	6.803	92.094
	Time (s)-parallel	-	20.679	1.175	15.888

Table 8.19 - Monte Carlo vs. SSFEM for the log-normal case (80% covariance)

Gaussian 15%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
MC	PCG iterations	899.678	763.678	490.980	271.804
	PFETI iterations	2.022.934	469.789	469.789	72.059
	Time (s)-sequential	179.401	112.023	68.632	59.793
	Time (s)-parallel	26.382	16.474	10.093	8.793
SSFEM		$p = 4$	$p = 4$	$p = 4$	$p = 6$
	PCG iterations	3	4	5	6
	PFETI iterations	473	269	180	448
	Time (s)-sequential	6.797	1.178	291	1.276
	Time (s)-parallel	1.156	204	53	203

Table 8.20 - Reliability analysis: MC vs. SSFEM for the Gaussian case (σE = 15% covariance)

log-normal 30%					
Correlation length b		$0.1a$	$1a$	$10a$	$100a$
MC	PCG iterations	1.087.678	1.143.034	490.236	356.205
	PFETI iterations	287.4353	512.199	93.203	79.748
	Time (s)-sequential	250.318	128.625	74.246	69.648
	Time (s)-parallel	36.811	18.915	10.919	10.242
SSFEM		$p = 5$	$p = 4$	$p = 3$	$p = 3$
	PCG iterations	5	5	4	4
	PFETI iterations	621	260	100	79
	Time (s) -sequential	302.369	2.884	162	132
	Time (s)-parallel	54.508	498	28	23

Table 8.21 - Reliability analysis : MC vs. SSFEM for the log-normal case (σE = 30% covariance)

log-normal 80%					
Correlation length b:		$0.1a$	$1a$	$10a$	$100a$
MCS	PCG iterations	2.223.683	2.478.497	792.509	557.113
	PFETI iterations	6.057.138	5.441.639	195.228	116.438
	Time (s)-sequential	522.998	267.753	147.731	103.245
	Time (s)-parallel	76.912	39.375	21.725	15.183
SSFEM		-	$p = 7$	$p = 8$	$p = 6$
	PCG iterations	-	25	15	20
	PFETI iterations	-	449	309	228
	Time (s)-sequential	-	472.607	517.627	92.094
	Time (s)-parallel	-	79.033	86.419	15.888

Table 8.22 - Reliability analysis: MC vs. SSFEM for the log-normal case (σE = 80% covariance)

8.4 SOIL-STRUCTURE INTERACTION NUMERICAL RESULTS

This section provides a set of numerical examples involving porous media as presented in Chapter 2. In order to assess the influence of pore pressure on a soil-structure interaction problem under seismic loading, a parametric study is conducted on a finite element mesh of 600k dof which simulates a 5-storey steel frame building situated on top of a soil mass 60x60mx20m. The building is excited with and without taking the soil under consideration, with and without soil non linearity and for dry and fully saturated cases. For the non linear case, the Mohr-Coulomb plasticity model is considered. The finite elements used are 2-node beams for simulating the building and 8-node hexahedral hybrid elements (8 translational dof and 8 pore pressure dof) for modeling the soil volume.

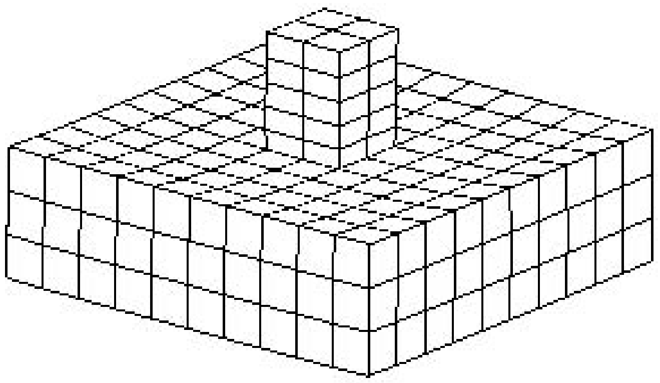


Figure 8.29 - A simplified view of the soil-structure model

The boundary nodes of the soil mass are connected with spring elements and harmonic dampeners in order to get more realistic results and to minimize seismic wave reflection and refraction phenomena. Loading conditions included the soil and structure’s own weight and a seismic load imposed as the equivalent force of accelerating the mass of the whole model as dictated by the seismic accelerogram of Figure 8.30 which is constructed from data of a real earthquake. The spectrum of the longitudinal axis of the earthquake is depicted on Figure 8.31. Each analysis is performed using implicit time integration as shown in Section 2.9, with a time step of 20ms in order to record the structure's behavior during the earthquake.

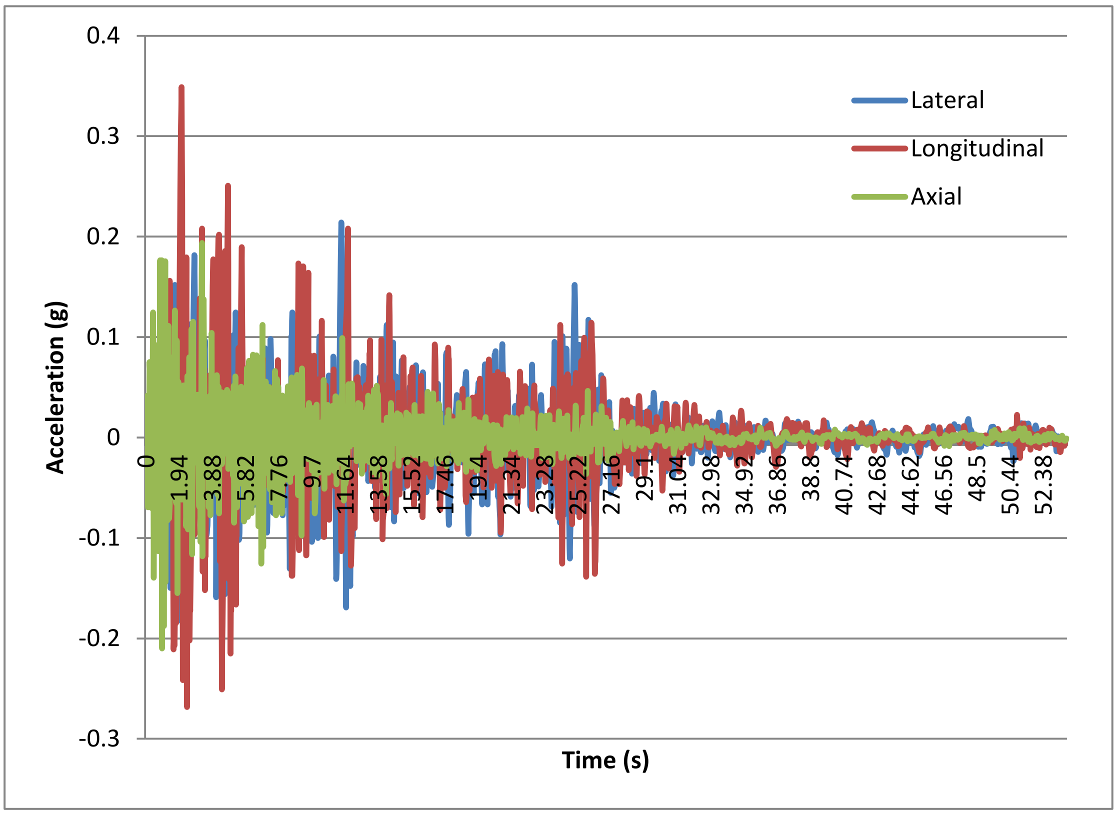


Figure 8.30 - Seismic accelerogram in all directions

The soil composition along with its properties is presented in Table 8.24. The maximum dampening ratio for the building was set equal to 10% while for the soil volume mesh, no damping was set as the harmonic dampeners were tuned to absorb most of the seismic energy.

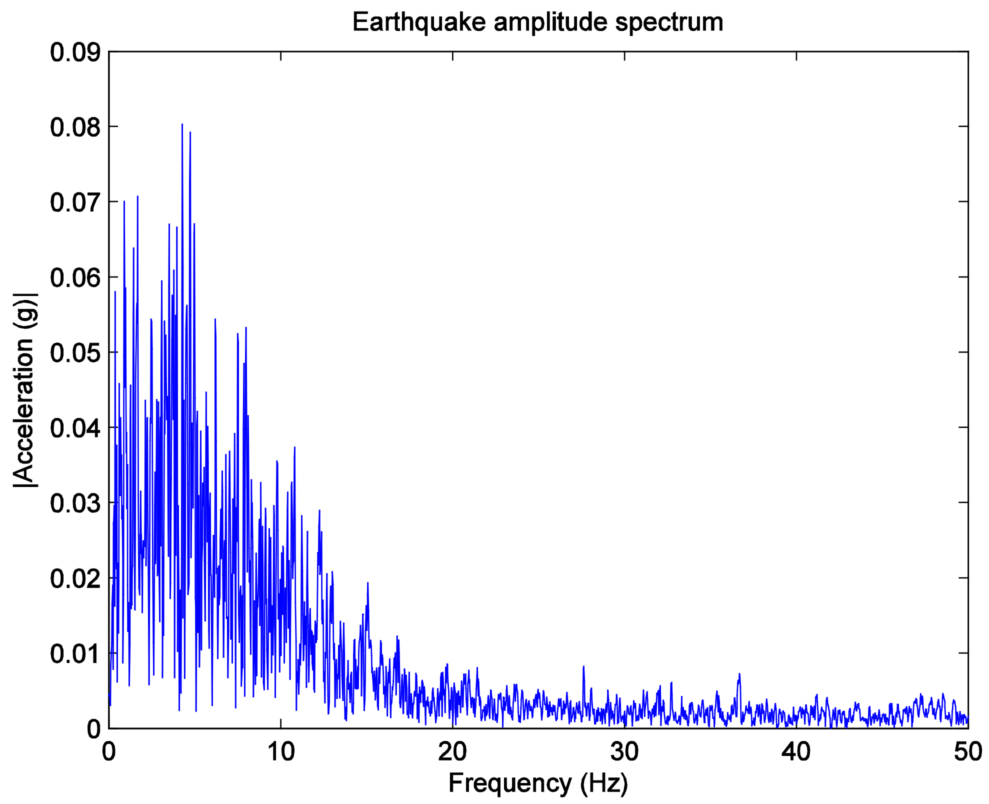


Figure 8.31 – Seismic spectrogram of the longitudinal axis

Soil	Young modulus (kPa)	v	Cohesion (kPa)	Friction (deg)	Dilation (deg)	Dry density (t/m^3)	Poro-sity	Permeabi-lity (m/s)
Clay (0-5m)	6000	0.25	17.5	20	0	1.9	0.35	1.02E-06
Sand (5-15m)	20000	0.3	0	35	5	1.8	0.455	3.06E-03
Dense sand (15-20m)	60000	0.35	300	30	0	1.85	0.3	1.02E-07

Table 8.23 – Soil composition properties

Table 8.24 depicts the maximum longitudinal displacements of the building roof and the soil along with the settlements after the seismic excitation, Figure 8.32 depicts the time history of the building longitudinal displacements and Figure 8.33 depicts the time history of the soil longitudinal displacements. It is evident that the omission of the soil participation in the seismic excitation underestimates the seismic response of the building. On the other hand, maximum longitudinal displacement for both the building and the soil is recorded for the dry soil case while maximum axial displacement and maximum soil settlements are recorded for the nonlinear saturated soil.

Foundation	**Max roof displacement (m)**	**Max soil displacement (m)**	**Max axial displacement (m)**	**Settlement (m)**
No soil	0.1313	0	0.1067	0
Dry soil	0.50494	0.11278	0.06578	0
Saturated soil	0.45885	0.10137	0.03569	0
Dry soil (nonlinear)	0.48531	0.10538	0.09729	0.04763
Saturated soil (nonlinear)	0.39891	0.08723	0.09755	0.07502

Table 8.24 – Maximum displacements and remaining settlements

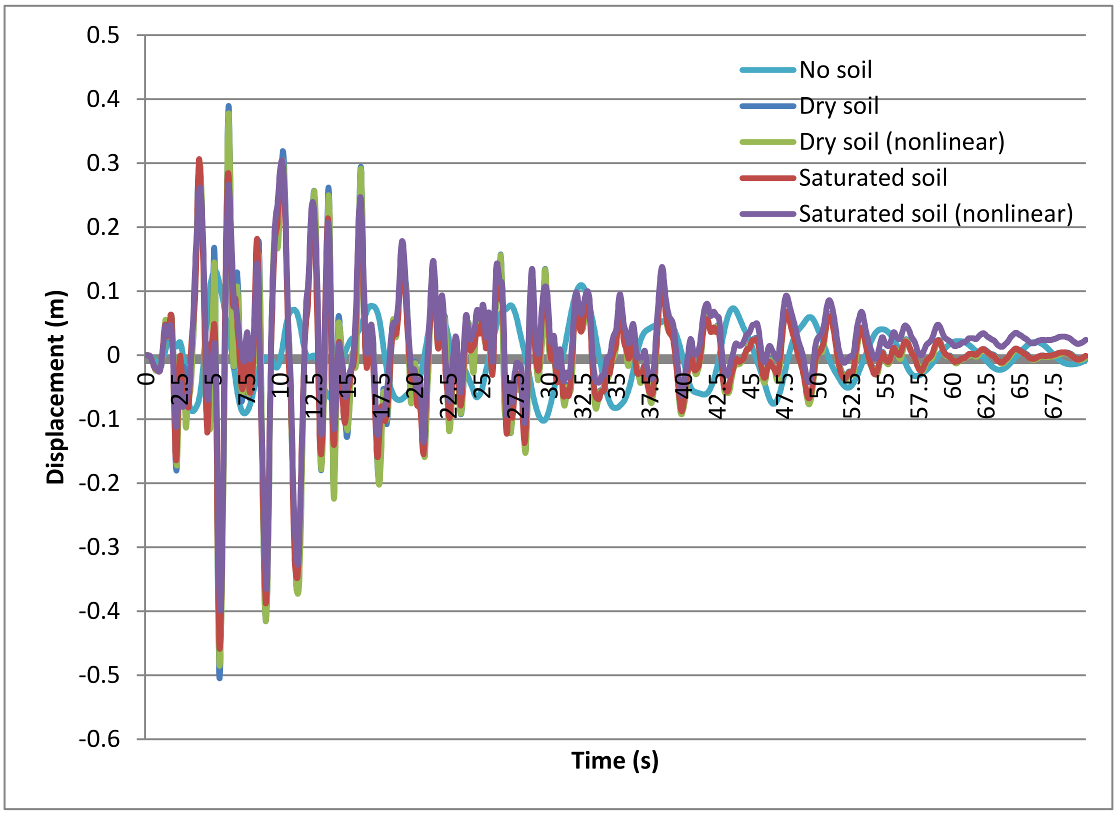


Figure 8.32 – Roof displacement history for all cases

The latter behavior is expected, since pore pressure buildup reduces shear resistance and results to liquefaction phenomena where the structure is steadily “sinking” in the soil. This is clearly depicted in Figure 8.34 where the time history of the settlements for each case is depicted.

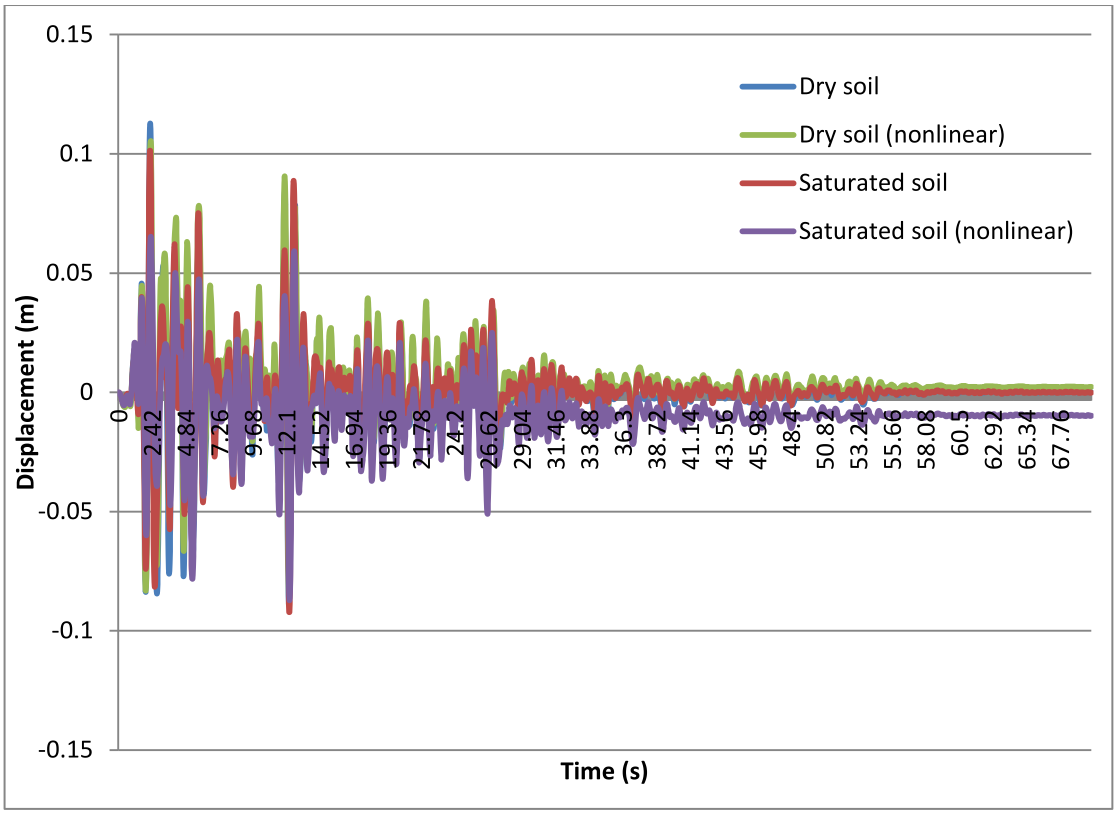


Figure 8.33 – Soil longitudinal displacement history

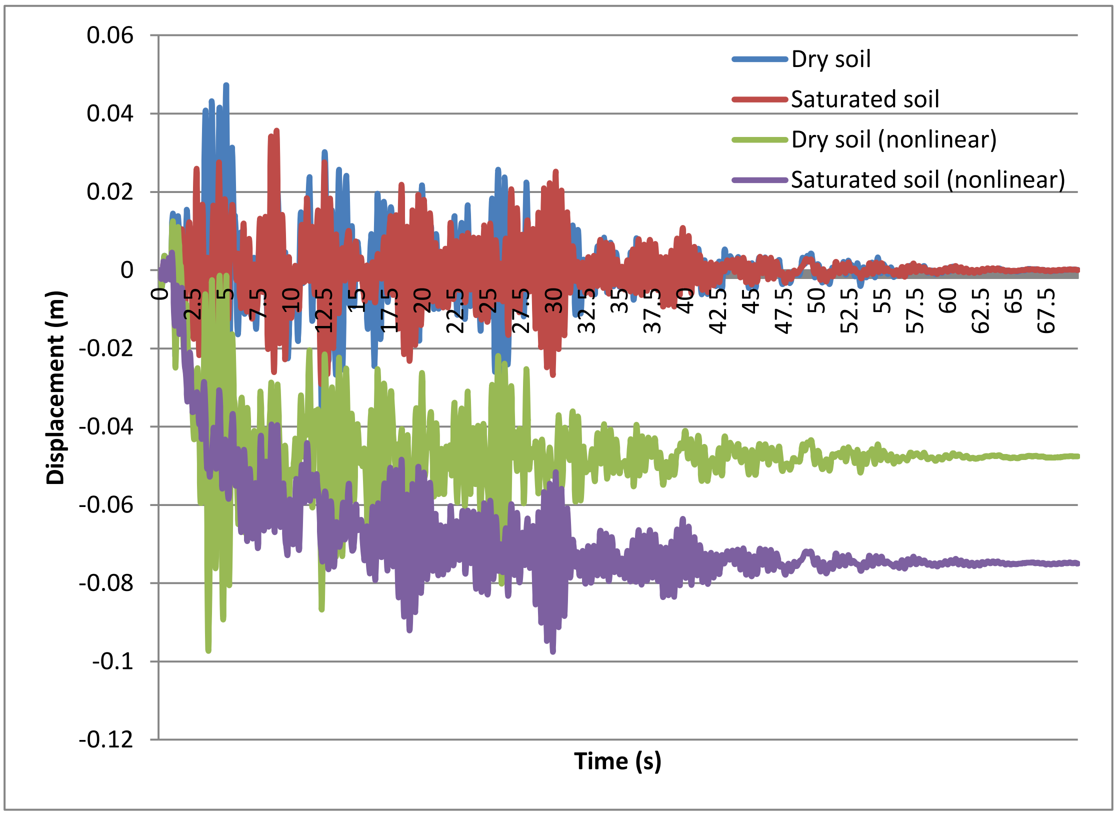


Figure 8.34 – Settlements time history

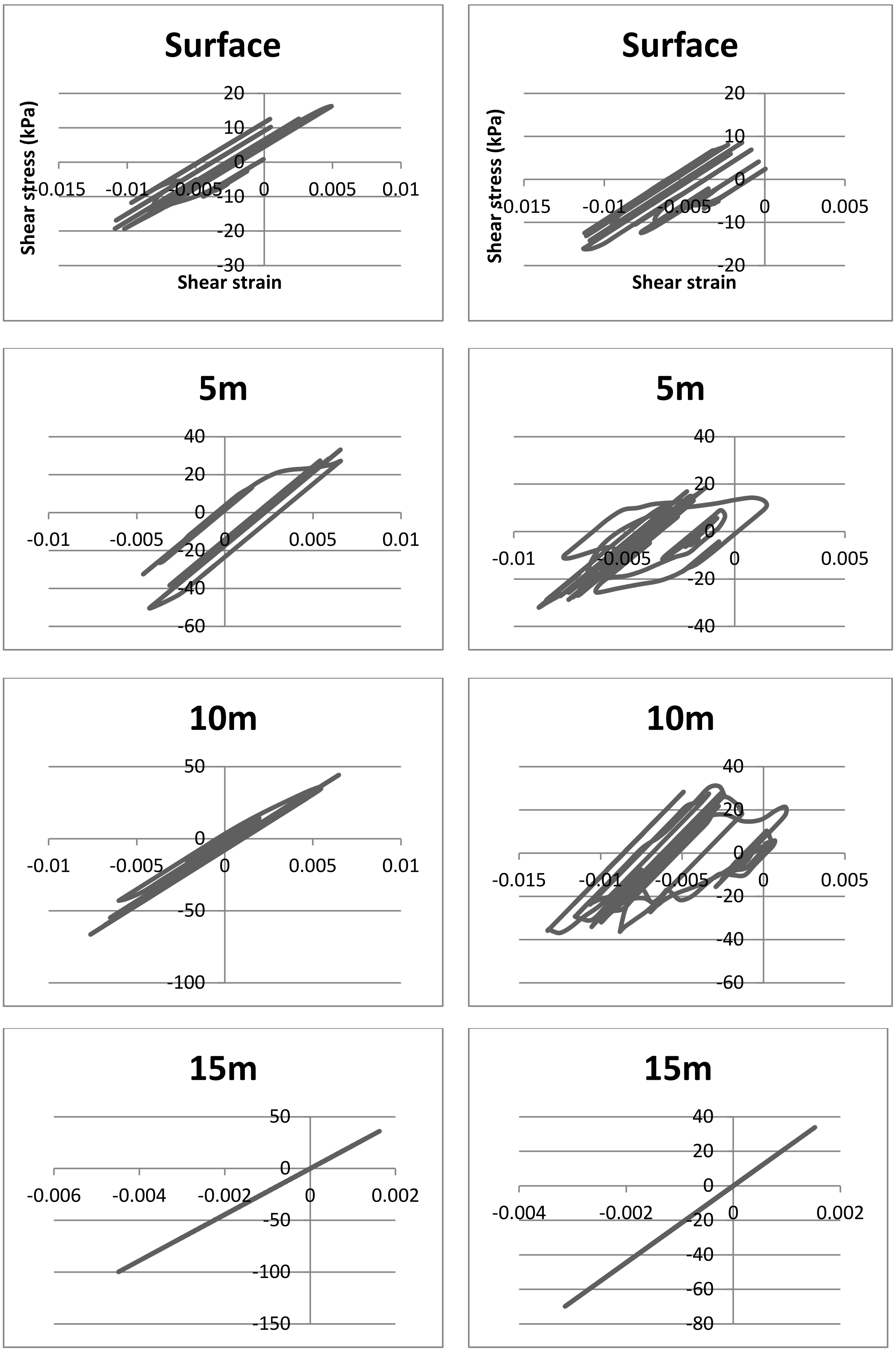


Figure 8.35 - Shear strain-stress curves (Left: Dry soil, Right: Saturated soil)

Figure 8.35 depict the shear stress-strain response of the soil under the building foundations, for the nonlinear soil cases. On the left, the curves for the dry soil are depicted and on the right, the curves for the saturated soil are shown. By observing the graphs for levels of 5 to 10m where the soil consists of sand, it is evident that the shear strength for the saturated case is reduced, suggesting mild liquefaction.

Figure 8.46 depicts a snapshot of the contour of the pore pressures during the earthquake. Pore pressure build up is evident at the foundation soil due to the cyclic loading occurring from the earthquake.

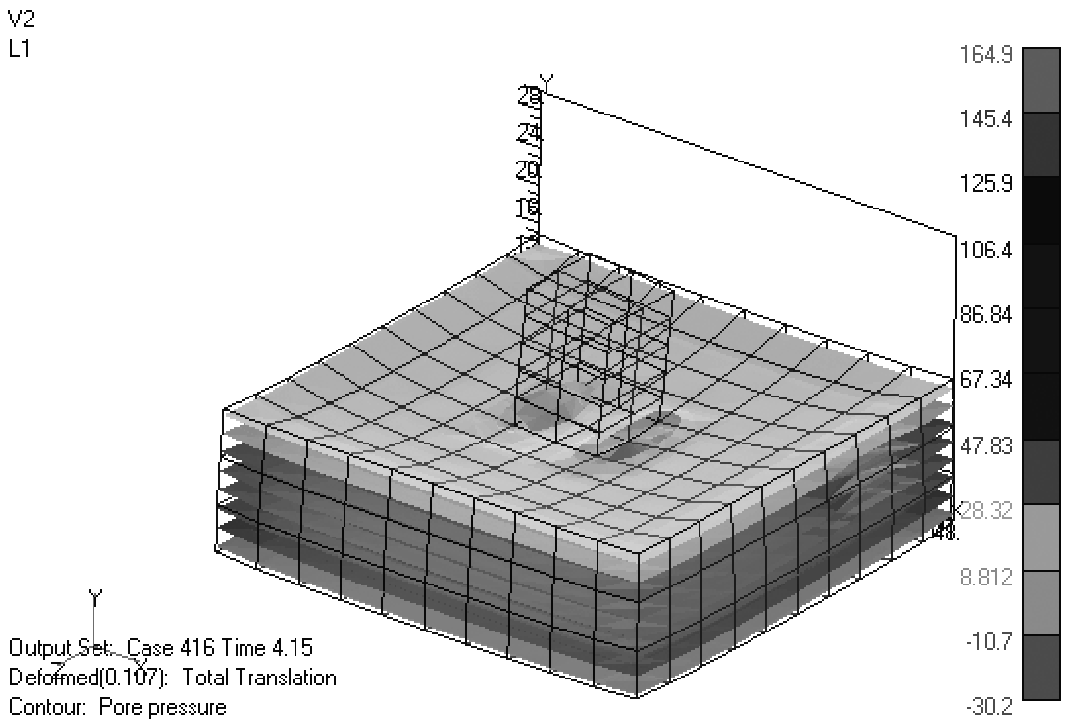


Figure 8.36 - Earthquake snapshot with a contour of pore pressures for the alluvial case

9 CONCLUDING REMARKS AND FUTURE WORK

This Thesis explores the solution of seismic soil-structure interaction problems. For the case of partially or fully saturated soils, whose pores are partially or fully filled with a fluid, their behavior is very different when compared to the behavior of single phase material. Intergranular forces will be affected by the pore pressures due to the fluid's presence, making the evaluation of these pore pressures of great importance for the accurate prediction of the medium's strains and stresses. This is especially applicable to dynamic phenomena such as those which occur in earthquakes as the pressures developed will, in general, be linked to the straining (or loading) history and must always be taken into account. In general, the simultaneous solution of both strains of the solid matrix and the transient fluid flow which occurs inside the medium's pores is required for seismic analysis.

Another issue for evaluating the seismic behavior of a structure involves the identification of the model parameters and the uncertainties associated with their estimation. Moreover, the intrinsic randomness of materials or loads is such that deterministic models using average characteristics lead to rough representations of real-life behavior. Stochastic mechanics is accounting for randomness and spatial variability of the mechanical properties of materials, leading to more accurate predictions at the expense of more computational resources.

Domain decomposition methods are used for the solution of both porous media and stochastic problems because they allow the exploitation of the natural parallelism offered by the subdivision of the physical domains to a number of subdomains. After the description of the basic primal and dual domain decomposition methods, the primal/dual domain decomposition method is introduced and a novel preconditioning technique custom tailored to porous media problems, applicable to dual and primal/dual domain decomposition methods is presented. Finally a set of novel domain decomposition methods for stochastic analysis is presented

The software systems that efficiently implement the above solution algorithms are large and complex compared to straightforward direct methods. This growth in complexity can be tackled with specialized software engineering techniques that promote decomposition, abstraction, and hierarchy. One of the most efficient paradigms that promotes the above properties is the object-oriented paradigm and it is presented here along with some of its applications for implementing a software code that can efficiently solve the problems presented using the solution techniques mentioned above.

Central processing units (CPUs) and Graphics processing units (GPUs) and their characteristic properties are thoroughly presented. In a massively parallel context, CPUs are quite different than GPUs. 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. On the other hand, CPU threads are more costly and massive parallelization generally involves distributed computing and message passing which are also thoroughly presented. The implementation of dual domain decomposition method in a hybrid CPU-GPU environment is also 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. An important aspect of the hybrid 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. 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.

Numerical examples are presented for all the above cases. The two families of domain decomposition methods, namely the primal domain decomposition (P-DDM) and the dual domain decomposition (D-DDM), for implicit dynamics were tested in two characteristic porous media problems. The numerical results demonstrated the efficiency of the proposed methods for solving large scale porous media problems. Furthermore, the beneficial effect of the use of the null space of the permeability matrix for the construction of the coarse problem was shown where an improvement of up to 40% was achieved on the computational efficiency of both initialization and iteration computing times. The same stands for the hybrid CPU/GPU implementation as the hybrid implementation achieves speedups ranging from 4.1x to 5.2x compared to the CPU, depending on the number of subdomains, while the corresponding speedups of hybrid vs GPU are around 1.3x.

For the stochastic problems considered, when comparing the novel solution techniques proposed for the MC procedure, a speedup of 1.25x was exhibited while for the SSFEM, a speedup of 3x was exhibited when utilizing the block-SSOR preconditioning combined with caching techniques with respect to the diagonally block preconditioning, making the SSFEM even more attractive for solving large scale stochastic problems in high performance computing environments. The efficiency comparison of the Monte Carlo method and SSFEM was based on the computation of the second order moments of the response field as well as on reliability analysis. For the first case, SSFEM proved to be more efficient when dealing with input fields exhibiting small to medium covariance. However, for the case of large covariance, the Monte Carlo outperformed the SSFEM in most cases with the latter being unable to converge in one of the problems cases.

For the soil structure interaction problem, a 4-storey building on a sandy soil with a clay surface was considered and the effect of soil and saturation was explored. It is concluded that the effect of pore pressures to the shear strength of the soil are existent only when considering a nonlinear analysis of the soil skeleton. Finally, not considering the influence of the soil to the structure, provides unrealistically low displacements for the building.

Below are some future considerations of the present work:

- Implementation of nonlinear fluid flow for porous media
- Implementation of SSFEM for nonlinear, dynamic problems
- Multi-GPU implementation of solution algorithms.
- 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 BIBLIOGRAPHY

[1] J. Boussinesq, Essai theorique sur l'equilibre d'elasticite des massif pulverulents, 1876.

[2] P. Fillunger, "Der Auftrieb in Talsperren," Austria, 1913, pp. 532-556.

[3] P. Fillunger, "Versuch uber die Zugfestigkeit bei allseitigem Wasserdruck," Austria, 1915, pp. 443-448.

[4] M. Levy, Quelques considerations sur la construction des grandes barrages, 1895.

[5] C. Lyell, Student's elements of geology, London, 1871.

[6] O. Reynolds, "Experiments showing dilatancy, a property of granular material," *Proc. R. Inst,* vol. 11, pp. 354-363, 1886.

[7] K. v. Terzaghi and L. Rendulic, "Die wirksame Flachenporositat des Betons," *Z. Ost. Ing.-u ArchitVer.,* vol. 86, pp. 1-9, 1934.

[8] K. v. Terzaghi, "The shearing resistance of saturated soils," *Proc. 1st ICSMFE,* vol. 1, pp. 54-56, 1936.

[9] M. A. Biot, "General theory of thee-dimensional consolidation," *J. Appl. Phys.,* vol. 12, pp. 155-164, 1941.

[10] O. C. Zienkiewicz and T. Shiomi, "Dynamic Behaviour of saturated porous media: The generalized Biot formulation and its numerical solution," *Int. J. Num. Anal. Geotech.,* vol. 8, pp. 71-96, 1984.

[11] R. S. Sandhu and E. L. Wilson, "Finite element analysis of flow in saturated porous elastic media," *ASCE EM,* vol. 95, pp. 641-652, 1969.

[12] J. Ghaboussi and E. L. Wilson, "Variational formulation of dynamics of fluid saturated porous elastic solids," *ASCE EM,* vol. 98, pp. 947-963, 1972.

[13] A. H. C. Chan, F. O. O. and D. Muir Wood, "A Fully Explicit u-w Scheme for Dynamic Soil and Pore Fluid Interaction," in *APCOM*, Hong Kong, 1991.

[14] O. C. Zienkiewicz, C. T. Chang and P. Bettess, "Drained, undrained, consolidating and dynamic behaviour assumptions in soils," *Geotechnique,* vol. 4, no. 30, pp. 385-395, 1980.

[15] O. C. Zienkiewicz and R. L. Taylor, The Finite Element Method - Volume I: Basic Formulation and Linear Problems, London: McGraw-Hill, 1989.

[16] O. C. Zienkiewicz and R. L. Taylor, The Finite Element Method - Volume II: Solid and Fluid Mechanics, Dynamics and Non-Linearity, London: McGraw-Hill, 1991.

[17] A. H. C. Chan, A unified Finite Element Solution to Static and Dynamic Geomechanics problems, Wales: Ph.D. Dissertation, University College of Swansea, 1988.

[18] M. Katona, "A general family of single-step methods for numerical time integration of structural dynamic equations," *NUMETA,* no. 1, pp. 213-225, 1985.

[19] M. G. Katona and O. C. Zienkiewicz, "A unified set of single step algorithms Part 3: The Beta-m method, a generalisation of the Newmark scheme," *Int. J. Num. Meth. Eng.,* no. 21, pp. 1345-1359, 1985.

[20] N. M. Newmark, "A method of computation for structural dynamics," in *Proc. ASCE*, 1959.

[21] J. Neveu, Introduction aux probabilites, Paris: Cours de l' Ecole Polytechnique, 1992.

[22] M. Loeve, Probability theory, New York: Springer Verlag, 1977.

[23] P.-D. Spanos and R.-G. Ghanem, "Stochastic finite element expansion for random media," *ASCE,* vol. 5, no. 115, pp. 1035-1053, 1989.

[24] R.-G. Ghanem and P.-D. Spanos, Stochastic finite elements - A spectral approach, Springer Verlag, 1991.

[25] R.-G. Ghanem and P.-D. Spanos, "Spectral stochastic finite element formulation for reliability analysis," *J. Eng. Mech.,* vol. 10, no. 117, pp. 2351-2372, 1991.

[26] R.-G. Ghanem and P.-D. Spanos, "Polynomial chaos in stochastic finite elements," *J. App. Mech, ASME,* pp. 197-202, 1990.

[27] R.-G. Ghanem, "Ingredients for a general purpose stochastic finite elements implementation," *Comp. Meth. Appl. Mech. Eng,* no. 168, pp. 19-34, 1999.

[28] R.-G. Ghanem and R. Kruger, "Numerical solution of spectral stochastic finite element systems," *Comp. Meth. Appl. Mech. Eng,* vol. 3, no. 129, pp. 289-303, 1996.

[29] R.-G. Ghanem, "The nonlinear gaussian spectrum of log-normal stochastic processes and variables," *J. Appl. Mech., ASME,* no. 66, pp. 964-973, 1999.

[30] R.-G. Ghanem, "Stochastic finite elements with multiple random non-Gaussian properties," *J. Eng. Mech., ASCE,* no. 125, pp. 26-40, 1999.

[31] R. A. Horn and C. R. Johnson, Matrix Analysis, Cambridge University Press, 1985.

[32] G. H. Golub and C. F. Van Loan, Matrix Computations (3rd ed.), Baltimore: Johns Hopkins, 1996.

[33] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards,* vol. 49, no. 6, 1952.

[34] Y. Saad, Iterative methods for sparse linear systems, Philadelphia, Pa.: Society for Industrial and Applied Mathematics, 2003.

[35] C. Farhat and F. X. Roux, "A method of finite element tearing and interconnecting and its parallel solution algorithm," *Internat. J. Numer. Meths. Engrg.,* vol. 32, 1992.

[36] 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," *Comp. Meth. App. Mech. Engrg.,* vol. 193, no. 42, pp. 4611-4662, 2004.

[37] C. Farhat, P.-S. Chen, F. Risler and F. X. Roux, "A unified framework for accelerating the convergence of iterative substructuring methods with Lagrange multipliers," *Int. J. Numer. Meth. Engng.,* vol. 42, pp. 257-288, 1998.

[38] 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," *Comp. Meth. Appl. Mech. Engrg.,* vol. 192, no. 35, pp. 3799-3830, 2003.

[39] J. Mandel, "Balancing domain decomposition," *Commun. Appl. Numer. Meth.,* vol. 9, pp. 233-241, 1993.

[40] C. Farhat and L. Crivelli, "A transient FETI methodology for large-scale parallel implicit computations in Structural Mechanics," *Int. J. Numer. Meth. Engng.,* vol. 37, pp. 1945-1975, 1994.

[41] C. Farhat, P.-S. Chen, F. Risler and F. Roux, "A unified framework for accelerating the convergence of iterative substructuring methods with Lagrange multipliers," *Int. J. Numer. Meth. Engng.,* vol. 42, pp. 257-288, 1998.

[42] G. Stavroulakis and M. Papadrakakis, "Advances on the domain decomposition solution of large scale porous media problems," *Comp. Meth. Appl. Mech. Engrg.,* vol. 196, pp. 1935-1945, 2009.

[43] M. Papadrakakis, Solving large-scale linear problems in solid and structural mechanics, John Wiley & Sons, 1993.

[44] D. Charmpis and M. Papadrakakis, "Improving the computational efficiency in finite element analysis of shells with uncertain properties," *Comp. Meth. Appl. Mech. Engrg.,* vol. 194, pp. 1447-1478, 2005.

[45] M. Papadrakakis and K. A., "Parallel solution methods for stochastic finite element analysis using Monte Carlo simulation," *Comp. Meth. Appl. Mech. Engrg,* vol. 168, pp. 305-320, 1999.

[46] D. Chung, G. M.A., L. Graham-Brady and F.-J. Lingen, "Efficient numerical strategies for spectral stochastic finite element models," *Int. J. Num. Meth. Engrg.,* vol. 64, pp. 1334-

1349, 2005.

[47] M. Pellissetti and R. Ghanem, "Iterative solution of systems of linear equations arising in the context of stochastic finite elements," *Adv. Engrg. Software,* vol. 31, pp. 607-616, 2000.

[48] D. Ghosh, P. Avery and C. Farhat, "A method to solve spectral stochastic finite element problems for large-scale systems," *Int. J. Numer. Meth. Engng,* vol. 0, pp. 1-6, 2008.

[49] K. Shankar, "Data Design: Types, Structures and Abstractions," in *Handbook of Software Engineering*, New York, Van Nostrand Reinhold, 1984.

[50] Macintosh MacApp 1.1.1 Programmer's reference, Cupertino, CA: Apple Computers, 1986.

[51] G. Myers, Composite/Structured design, New York, NY.: Van Nostrand Reinhold, 1978.

[52] B. Liskov, "A design methodology for reliable software systems," in *Tutorial on software design techniques*, New York, NY, IEEE computer society, 1980.

[53] M. Zelkowitz, "Perspectives on Software Engineering," *ACM computing surveys,* vol. 10, no. 2, p. 20, 1978.

[54] D. Parnas, P. Clements and D. Weiss, "The modular structure of complex systems," *IEEE transactions on software engineering,* Vols. SE-11, no. 3, p. 260, 1985.

[55] B. a. Parnas, A-7E Software.

[56] D. Parnas, P. Clements and D. Weiss, "Enhancing reusability with information hiding," in *Proceedings of the workshop on reusability in programming*, 1983.

[57] O. Dahl, E. Dijkstra and C. Hoare, Structured programming, London: Academic Press, 1972.

[58] M. Shaw, "Abstraction techniques in modern programming languages," *IEEE software,* vol. 1, no. 4, 1984.

[59] V. Berzins, M. Gray and D. Naumann, "Abstraction-based software development," *Communications of the ACM,* vol. 29, no. 5, 1986.

[60] H. Abelson and G. Sussman, Structure and interpretation of computer programs, Cambridge, MA.: The MIT press, 1985.

[61] E. Seidewitz and M. Stark, "Towards a general object-oriented software development methodology," in *Proceedings of the first international conference on Ada programming language applications for the NASA space station*, NASA Lyndon B. Johnson space center, TX., 1986.

[62] B. Meyer, Object-oriented software construction, New York, NY: Prentice Hall, 1988.

[63] R. Wirfs-Brock and B. Wilkerson, "Object-oriented design: A responsibility-driven approach," *SIGPLAN notices,* vol. 24, no. 10, 1989.

[64] D. Ingalls, "The Smalltalk-76 programming system design and implementation," in *Proceedings of the fifth annual ACM symposium on principles of programming languages*.

[65] J. Gannon, R. Hamlet and H. Mills, "Theory of modules," *IEEE transactions of software engineering,* Vols. SE-13, no. 7, 1987.

[66] B. Liskov, "Data abstraction and hierarchy," *SIGPLAN notices,* vol. 23, no. 5, 1988.

[67] B. Cox, Object-oriented programming: An evolutionary approach, Reading, MA.: Addison-Wesley, 1986.

[68] S. Danforth and C. Tomlinson, "Type theories and object-oriented programming," *ACH Computer surveys,* vol. 20, no. 1, 1988.

[69] S. Zilles, On conceptual modeling: Perspectives from artificial intelligence, databases, and programming languages, New York: Springer-Verlag, 1984.

[70] P. Wegner, "Dimensions of object-based language design," *SIGPLAN notices,* vol. 22, no. 12, 1987.

[71] L. Tesler, "The Smalltalk environment," *Byte,* vol. 6, no. 8, 1981.

[72] Borning and Ingalls, Type declaration.

[73] D. Thomas, "What is an Object?," *Byte,* vol. 14, no. 3, 1989.

[74] M. Flynn, "Very high-speed computing systems," in *Proc. IEEE 54*, 1966.

[75] J. Larus and C. Kozyrakis, "Transactional memory," *Commun. ACM,* vol. 51, no. 7, pp. 80-88, 2008.

[76] B. Chamberlain, D. Callahan and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl,* vol. 21, no. 3, pp. 291-312, 2007.

[77] O. L, "Effects of ordering strategies and programming paradigms on sparse," *SIAM Rev,* vol. 44, no. 3, pp. 373-393, 2002.

[78] I. Foster, Designing and Building Parallel Programs, Reading, MA.: Addison-Wesley, 1995.

[79] G. Amdahl, "Validity of the single processor approach to achieving large scale computing," in *Proceedings of the American Federation of Information Processing*, Atlantic City, NJ, 1967.

[80] J. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM,* vol. 5, no. 31, pp. 532-533, 1988.

[81] N. Corporation, "CUDA Programming Guide Version 3.2".

[82] D. Kirk and W. Hwu, Programming Massively Parallel Processors: A Hands-on approach, Morgan-Kauffman, 2010.

[83] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable parallel programming with CUDA, 2008.