ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Παραλληλοποίηση και βελτιστοποίηση εφαρμογών για παράλληλα συστήματα μεγάλης κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Δ. Σιακαβάρας

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Αθήνα, Νοέμβριος 2012

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Παραλληλοποίηση και βελτιστοποίηση εφαρμογών για παράλληλα συστήματα μεγάλης κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος Δ. Σιακαβάρας

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Νοεμβρίου 2012.

.....	.....	.....
Ν. Κοζύρης	Α. Παγουρτζής	Ν. Παπασπύρου
Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2012.

.....

Δημήτριος Δ. Σιακαβάρας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ένα από τα πιο δύσκολα προβλήματα στα συστήματα παράλληλης είναι η ανάπτυξη παράλληλου λογισμικού το οποίο κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν έπειτα από έναν αριθμό επεξεργαστών επειδή το κόστος επικοινωνίας γίνεται συγκρίσιμο με την ωφέλιμη υπολογιστική εργασία. Σκοπός της παρούσας διπλωματικής είναι η παραλληλοποίηση του αλγορίθμου συζυγών κλίσεων ο οποίος χρησιμοποιείται για την αριθμητική επίλυση συστημάτων γραμμικών διαφορικών εξισώσεων της μορφής $Ax = b$, όπου ο πίνακας A είναι αραιός. Δοκιμάζουμε διάφορους τρόπους διανομής του πίνακα και των διανυσμάτων του αλγορίθμου με στόχο να βρούμε τον βέλτιστο συνδυασμό φόρτου υπολογισμών και επικοινωνίας σε όλους τους επεξεργαστές και να πετύχουμε όσο το δυνατόν καλύτερη κλιμάκωση για μεγάλο αριθμό επεξεργαστών. Η πλατφόρμα στην οποία εκτελούμε τα πειράματα μας είναι μία συστοιχία από κόμβους πολυεπεξεργαστών με μοιραζόμενη μνήμη (SMP) και το δίκτυο διασύνδεσης είναι Gigabit Ethernet.

Λέξεις κλειδιά: παράλληλος προγραμματισμός, MPI, OpenMP, SMP συστοιχία, αλγόριθμος συζυγών κλίσεων, συστήματα γραμμικών εξισώσεων, αραιοί πίνακες

Abstract

One of the most difficult problems in parallel computing is to develop parallel software that has effective scalability. Several applications do not scale further than a number of processors because communication overhead becomes comparable with computational work. The goal of this diploma thesis is the parallelization of the conjugate gradient algorithm which is used for the numerical solution of systems of linear equations in the form $Ax = b$ in which the matrix A is sparse. We try different ways of distributing the matrix and vectors of the algorithm in order to find the best combination of computation and communication load between the processors and to achieve the best scalability for a large number of processors. Our execution platform is a cluster of SMP nodes and the interconnection network is Gigabit Ethernet.

Keywords: parallel programming, MPI, OpenMP, SMP cluster, conjugate gradient algorithm, systems of linear equations, sparse matrices

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Επισκόπηση

Σύμφωνα με το Νόμο του Moore [1], ο αριθμός των τρανζίστορ που μπορούν να τυπωθούν σε ένα chip διπλασιάζεται κάθε δύο χρόνια, με αποτέλεσμα να πολλαπλασιάζεται και η επεξεργαστική ισχύς. Όμως, καθώς τα τρανζίστορ πολλαπλασιάζονται η κατανάλωση ενέργειας και η θερμότητα που παράγεται πάνω στο chip έχουν κάνει πολύ δύσκολη την εξέλιξη των επεξεργαστών. Η επίδοση των σειριακών υπολογιστών έχει φτάσει στα όρια της. Έτσι η κοινωνία της πληροφορικής αναγκάζεται να στραφεί στη λύση των παράλληλων υπολογιστών.

Σε ένα σειριακό υπολογιστή μία βελτίωση σε επίπεδο υλικού οδηγεί σε αντίστοιχη βελτίωση και σε επίπεδο λογισμικού. Έτσι οι προγραμματιστές και οι προμηθευτές λογισμικού μπορούσαν να βασίζονται στο γεγονός ότι σε ταχύτερους υπολογιστές τα προγράμματά τους θα έτρεχαν πιο γρήγορα. Στους παράλληλους υπολογιστές κάτι τέτοιο δεν ισχύει. Σε πρώτη ανάλυση μπορεί να φαίνεται ότι το να αυξήσουμε τους επεξεργαστές είναι μία ιδανική λύση για να πετύχουμε επιτάχυνση στα προγράμματά μας. Ωστόσο προκύπτουν πολλά ζητήματα κατά την παραλληλοποίηση των εφαρμογών (π.χ. η πρόσβαση πολλών επεξεργαστών σε κοινή μνήμη προκαλεί συμφόρηση στο διάδρομο δεδομένων). Με άλλα λόγια αν δεν δοθεί ιδιαίτερη προσοχή κατά την παραλληλοποίηση μπορεί να μειωθεί η απόδοση πολλών εφαρμογών.

Οι παράλληλοι υπολογιστές έχουν επιφέρει σημαντικές βελτιώσεις σε εφαρμογές που αφορούν πολλούς τομείς της επιστήμης (γεωλογία, χημεία, βιολογία), της βιομηχανίας (αυτοκινητοβιομηχανία, φαρμακοβιομηχανία) και σε άλλους τομείς, π.χ. μηχανές αναζήτησης στο Web.

Η μεγαλύτερη πρόκληση για τους παράλληλους υπολογιστές είναι η δημιουργία κατάλληλου υλικού και λογισμικού το οποίο θα βοηθάει να γράφονται σωστά προγράμματα για αυτά τα συστήματα τα οποία θα τρέχουν αποδοτικά (όσον αφορά σε ταχύτητα αλλά και σε κατανάλωση ενέργειας) καθώς ο αριθμός των επεξεργαστών αυξάνεται.

1.2 Παράλληλες Αρχιτεκτονικές

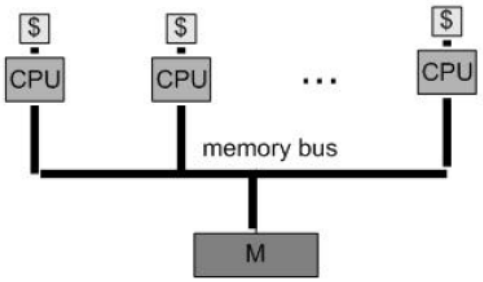
Σύμφωνα με την ταξινόμηση του Flynn [2] τα υπολογιστικά συστήματα κατηγοριοποιούνται με βάση το επίπεδο παραλληλισμού που προσφέρουν όσον αφορά τις εντολές και τα δεδομένα. Έχουμε έτσι δύο περιπτώσεις όσον αφορά τη ροή των εντολών: SI (single instruction), MI (multiple instruction), και δύο περιπτώσεις ροής δεδομένων: SD (single data), MD (multiple data). Κάθε υπολογιστικό σύστημα θα έχει ένα τύπο ροής εντολών και έναν τύπο ροής δεδομένων, έτσι έχουμε τέσσερις περιπτώσεις:

- **SISD:** Ο κλασσικός σειριακός υπολογιστής, δεν παρέχεται καμία παραλληλία, ούτε σε επίπεδο εντολής ούτε δεδομένων.
- **SIMD:** Ένας παράλληλος υπολογιστής που εκτελεί το ίδιο κομμάτι κώδικα πάνω σε διαφορετικά δεδομένα.
- **MISD:** Διαφορετικά κομμάτια κώδικα εκτελούνται πάνω στα ίδια δεδομένα. Ασυνήθιστη αρχιτεκτονική, χρησιμοποιείται σε ελάχιστα, μη εμπορικά, μηχανήματα.
- **MIMD:** Παράλληλος υπολογιστής όπου κάθε επεξεργαστής εκτελεί ξεχωριστό κομμάτι κώδικα σε ξεχωριστά δεδομένα. Η πιο ευρέως χρησιμοποιούμενη αρχιτεκτονική. Τυπικά παραδείγματα είναι τα clusters, και τα συστήματα με πολλαπλούς επεξεργαστές.

Όπως αναφέρθηκε προηγουμένως η αρχιτεκτονική που χρησιμοποιείται στο μεγαλύτερο ποσοστό των παράλληλων υπολογιστικών συστημάτων είναι η MIMD. Θα εξετάσουμε τις αρχιτεκτονικές αυτές με βάση την οργάνωση της μνήμης τους. Υπάρχουν τρεις κατηγορίες: αρχιτεκτονικές κοινής μνήμης, αρχιτεκτονικές κατανεμημένης μνήμης και υβριδικές αρχιτεκτονικές.

1.2.1 Αρχιτεκτονική κοινής μνήμης

Στις αρχιτεκτονικές κοινής μνήμης δυο ή περισσότεροι επεξεργαστές μοιράζονται την κεντρική μνήμη του συστήματος μέσω ενός κοινού διαύλου. Ο κάθε επεξεργαστής έχει ξεχωριστή ιεραρχία κρυφών μνημών. Εδώ είναι σημαντικό να τονίσουμε ότι σε τέτοια συστήματα μπορούν να τρέξουν διεργασίες με διαφορετικό εικονικό χώρο διευθύνσεων, ακόμα και αν αυτές μοιράζονται το φυσικό χώρο διευθύνσεων. Οι επεξεργαστές επικοινωνούν μεταξύ τους μέσω καθολικών μεταβλητών, και όλοι οι επεξεργαστές μπορούν να προσπελάσουν οποιοδήποτε block μνήμης μέσω εντολών φόρτωσης και αποθήκευσης. Η οργάνωση των συστημάτων αρχιτεκτονικής κοινής μνήμης φαίνεται στο σχήμα 1.1.



Σχήμα 1.1: Οργάνωση συστήματος αρχιτεκτονικής κοινής μνήμης.

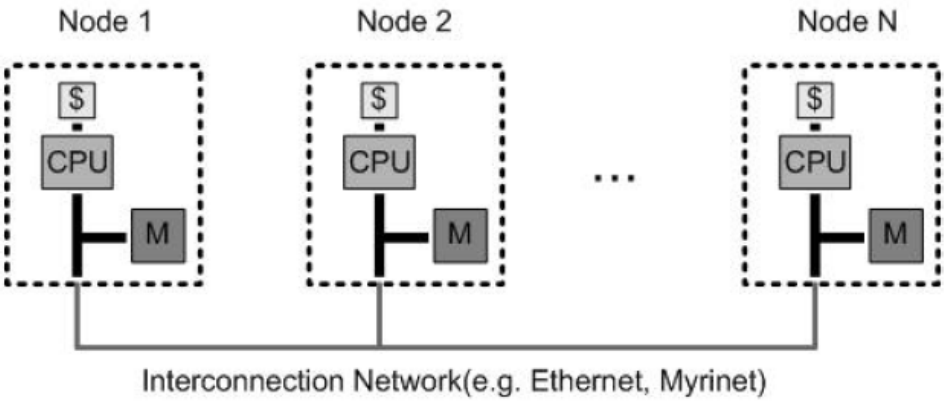
Οι αρχιτεκτονικές κοινής μνήμης χωρίζονται σε δύο κατηγορίες ανάλογα με την οργάνωση της κοινής μνήμης: Στην πρώτη κατηγορία ο χρόνος πρόσβασης στην μνήμη είναι σταθερός, ανεξάρτητος του επεξεργαστή που έστειλε το αίτημα και ανεξάρτητος του block που προσπελαύνεται. Αυτά τα συστήματα ονομάζονται Uniform Memory Access (UMA). Στην δεύτερη κατηγορία ανήκουν τα συστήματα Non-Uniform Memory Access (NUMA) στα οποία η κοινή μνήμη κατανέμεται σε πολλαπλούς κόμβους. Κάθε κόμβος είναι τοπικός σε ένα σύνολο επεξεργαστών και η τοπική πρόσβαση είναι, γενικά, ταχύτερη από την απομακρυσμένη πρόσβαση.

Αφού οι επεξεργαστές μοιράζονται δεδομένα και μπορούν να προσπελαύνουν τον ίδιο φυσικό χώρο διευθύνσεων υπάρχει ο κίνδυνος συγκρούσεων. Αυτό σημαίνει ότι μπορεί δύο επεξεργαστές να χρειάζεται να επεξεργαστούν ταυτόχρονα μία κοινή μεταβλητή. Σε αυτό το σημείο γίνεται απαραίτητη η χρήση ενός μηχανισμού συγχρονισμού των επεξεργαστών. Ένας τέτοιος μηχανισμός είναι η χρήση κλειδωμάτων για τις κοινές μεταβλητές. Μόνο ένας επεξεργαστής μπορεί να αποκτήσει το κλείδωμα και όλοι οι άλλοι επεξεργαστές είναι αναγκασμένοι να τον περιμένουν να τελειώσει.

Η αρχιτεκτονική κοινής μνήμης παρέχει μεγάλη ευκολία στον προγραμματιστή καθώς η πρόσβαση στην κοινή μνήμη γίνεται απλά με εντολές φόρτωσης, αποθήκευσης στη μνήμη. Όμως μπορεί να χρησιμοποιηθεί μόνο για λίγους συνδεδεμένους επεξεργαστές (30 το πολύ), αφού περιορίζεται σε μεγάλο βαθμό από το εύρος ζώνης του διαδρόμου και της μνήμης.

1.2.2 Αρχιτεκτονική κατανεμημένης μνήμης

Στις αρχιτεκτονικές κατανεμημένης μνήμης κάθε επεξεργαστής έχει ιδιωτική μνήμη και ιεραρχία κρυφών μνημών. Οι επεξεργαστές συνδέονται μεταξύ τους με ένα δίκτυο διασύνδεσης (π.χ. ethernet, myrinet). Κάθε επεξεργαστής έχει πρόσβαση μόνο στην τοπική μνήμη. Η επικοινωνία μεταξύ των επεξεργαστών γίνεται μέσω της ανταλλαγής μηνυμάτων πάνω από το δίκτυο διασύνδεσης. Για αυτό το σκοπό παρέχονται στον προγραμματιστή ειδικές εντολές αποστολής/λήψης δεδομένων από/προς οποιονδήποτε επεξεργαστή. Ένα παράδειγμα συστήματος κατανεμημένης μνήμης παρουσιάζεται στο σχήμα 1.2.



Σχήμα 1.2: Οργάνωση συστήματος αρχιτεκτονικής κατανεμημένης μνήμης.

Αυτή η αρχιτεκτονική χρησιμοποιείται κατά κύριο λόγο σε συστάδες υπολογιστών (clusters), όπου πολλοί σειριακοί υπολογιστές συνδέονται μέσω των διασυνδέσεων εισόδου/εξόδου σε κάποιο δίκτυο (π.χ. Ethernet). Ο κάθε υπολογιστής τρέχει ξεχωριστό αντίγραφο του λειτουργικού συστήματος.

Ένα μειονέκτημα των clusters, και κατ' επέκταση της αρχιτεκτονικής κατανεμημένης μνήμης, είναι το κόστος συντήρησης και διαχείρισης του συστήματος. Για ένα cluster με n διασυνδεδεμένους υπολογιστές το λειτουργικό κόστος είναι περίπου ίσο με το κόστος για τη λειτουργία n ξεχωριστών μηχανημάτων. Αντίθετα σε ένα σύστημα κοινής μνήμης στο οποίο n επεξεργαστές μοιράζονται την ίδια μνήμη το κόστος συντήρησης και διαχείρισης είναι ίσο με το αντίστοιχο κόστος για ένα μηχάνημα.

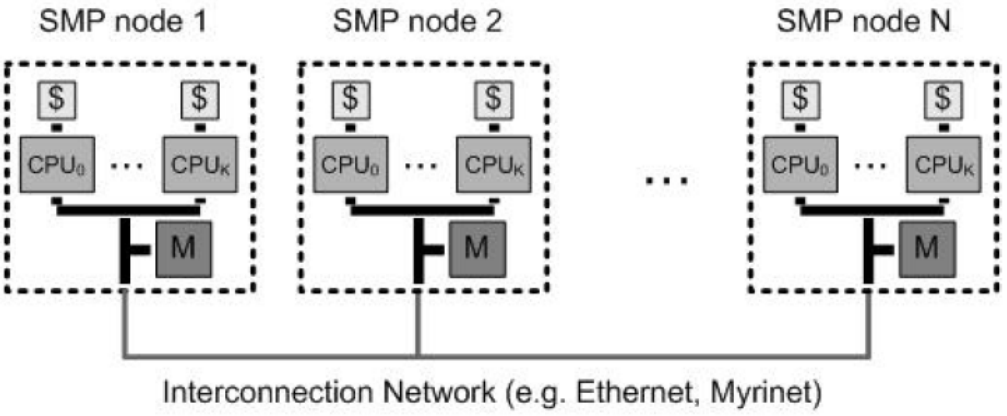
Ένα ακόμα μειονέκτημα των clusters είναι το εύρος ζώνης (bandwidth) και η ταχύτητα απόκρισης (latency). Στα clusters οι επεξεργαστές συνήθως συνδέονται μεταξύ τους μέσω των διεπαφών εισόδου/εξόδου, ενώ οι πυρήνες σε ένα πολυπύρηνο σύστημα μέσω της διεπαφής της μνήμης. Η διεπαφή της μνήμης έχει υψηλότερο εύρος ζώνης και ταχύτητα απόκρισης, επιτρέποντας ταχύτερη επικοινωνία.

Όσον αφορά τον προγραμματιστή, η παραλληλοποίηση ενός σειριακού προγράμματος για ένα σύστημα κατανεμημένης μνήμης απαιτεί ιδιαίτερο κόπο. Όλο το βάρος για την παραλληλοποίηση της εφαρμογής και του διαμοιρασμού δεδομένων ανάμεσα στους επεξεργαστές είναι ευθύνη του. Κάθε επικοινωνία ανάμεσα στους επεξεργαστές πρέπει να αναγνωριστεί και να υλοποιηθεί ρητά από τον προγραμματιστή.

Πέρα από τα παραπάνω μειονεκτήματα της αρχιτεκτονικής κατανεμημένης μνήμης, υπάρχει και ένα βασικό πλεονέκτημα. Τα συστήματα αυτά κλιμακώνουν πάρα πολύ καλά καθώς αυξάνεται ο αριθμός των διαθέσιμων επεξεργαστών. Έτσι υπάρχουν συστήματα με εκατοντάδες, ακόμα και χιλιάδες επεξεργαστές.

1.2.3 Υβριδική αρχιτεκτονική

Η υβριδική αρχιτεκτονική συνδυάζει τις δύο προαναφερθείσες αρχιτεκτονικές. Οι κόμβοι οι οποίοι συνδέονται μέσω του δικτύου διασύνδεσης αποτελούνται από πολυπύρηνους επεξεργαστές κοινής μνήμης. Η τυπική οργάνωση ενός συστήματος υβριδικής αρχιτεκτονικής φαίνεται στο σχήμα 1.3. Η υβριδική αρχιτεκτονική συνδυάζει τα πλεονεκτήματα των δύο παραπάνω αρχιτεκτονικών και για αυτό χρησιμοποιείται κατά κόρον στα σύγχρονα clusters και στους σύγχρονους υπέρ-υπολογιστές.



Σχήμα 1.3: Οργάνωση συστήματος υβριδικής αρχιτεκτονικής.

1.3 Παράλληλα Προγραμματιστικά Μοντέλα

Η σημαντικότερη πρόκληση που αντιμετωπίζουν οι παράλληλοι υπολογιστές και η οποία καθορίζει την εξέλιξη τους είναι η ευκολία που παρέχεται στους προγραμματιστές ώστε να γράφουν παράλληλα προγράμματα τα οποία είναι κομψά και αποδοτικά. Στόχος των παράλληλων προγραμματιστικών μοντέλων είναι η δημιουργία ενός περιβάλλοντος προγραμματισμού και εκτέλεσης προγραμμάτων στο οποίο το λειτουργικό σύστημα αναλαμβάνει να εκτελέσει το μεγαλύτερο μέρος των λειτουργιών επιτρέποντας τον προγραμματιστή να επικεντρωθεί στο κομμάτι της συγγραφής αποδοτικού κώδικα, χωρίς να χρειάζεται να ασχοληθεί με λεπτομέρειες υλοποίησης.

Αναλογικά με τις τρεις αρχιτεκτονικές παράλληλων υπολογιστών, και τα προγραμματιστικά μοντέλα είναι τρία και κατηγοριοποιούνται με βάση τον τρόπο επικοινωνίας ανάμεσα στις διεργασίες. Έτσι έχουμε: μοντέλο κοινού χώρου διευθύνσεων, μοντέλο ανταλλαγής μηνυμάτων, υβριδικό μοντέλο. Κάθε προγραμματιστικό μοντέλο είναι κατάλληλο για την αντίστοιχη αρχιτεκτονική, ωστόσο είναι δυνατόν να χρησιμοποιηθεί και σε οποιαδήποτε άλλη, π.χ. σε αρχιτεκτονική κοινής μνήμης μπορούμε να χρησιμοποιήσουμε το μοντέλο ανταλλαγής μηνυμάτων, με μειωμένες όμως αποδόσεις.

1.3.1 Μοντέλο κοινού χώρου διευθύνσεων

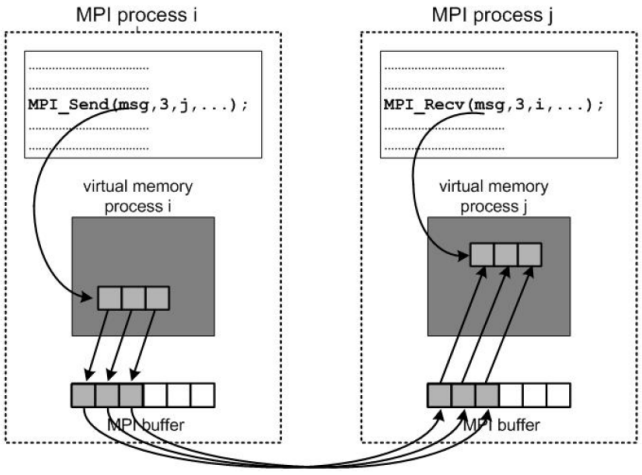
Στο μοντέλο κοινού χώρου διευθύνσεων οι διεργασίες επικοινωνούν μέσω μεταβλητών καθολικής εμβέλειας στον κοινό χώρο διευθύνσεων. Λόγω της ταυτόχρονης πρόσβασης πολλών επεξεργαστών σε κοινή μνήμη προκύπτουν προβλήματα όσον αφορά το συγχρονισμό των προσβάσεων αυτών. Για την επίλυση αυτών των ζητημάτων χρησιμοποιούνται αλγόριθμοι με κλειδώματα και σεμαφόρους.

Το θετικό αυτού του μοντέλου όσον αφορά τη σκοπιά του προγραμματιστή είναι η ευκολία που του παρέχει. Μέσα στο πρόγραμμα οι προσπελάσεις στις καθολικές μεταβλητές γίνονται με απλές εντολές ανάγνωσης / εγγραφής στην μνήμη και η εκάστοτε υλοποίηση του προγραμματιστικού μοντέλου έχει την ευθύνη για το συγχρονισμό των ταυτόχρονων προσβάσεων.

Υπάρχουν πολλές υλοποιήσεις του προγραμματιστικού μοντέλου κοινού χώρου διευθύνσεων, η πιο ευρέως χρησιμοποιούμενη ωστόσο είναι η OpenMP την οποία και χρησιμοποιούμε στα πειράματα μας.

1.3.2 Μοντέλο ανταλλαγής μηνυμάτων

Στην αρχιτεκτονική κατανεμημένης μνήμης όπως αναφέρθηκε σε προηγούμενη παράγραφο ο κάθε επεξεργαστής έχει ιδιωτική μνήμη. Έτσι στο μοντέλο ανταλλαγής μηνυμάτων οι επεξεργαστές (ή ισοδύναμα οι διεργασίες) επικοινωνούν μέσω της ανταλλαγής μηνυμάτων πάνω στο δίκτυο διασύνδεσης. Από προγραμματιστικής σκοπιάς, υπάρχουν εντολές αποστολής και λήψης μηνυμάτων. Στην πιο απλή μορφή της μία εντολή αποστολής δέχεται ένα buffer δεδομένων και ένα αναγνωριστικό του παραλήπτη. Μία εντολή λήψης ορίζει ένα buffer στο οποίο θα αποθηκευθούν τα ληφθέντα δεδομένα και το αναγνωριστικό του αποστολέα. Έτσι κάθε εντολή αποστολής αντιστοιχίζεται με μία εντολή λήψης όπως φαίνεται στο σχήμα 1.4.



Σχήμα 1.4: Λειτουργία αποστολής / λήψης δεδομένων στο μοντέλο ανταλλαγής μηνυμάτων.

Υπάρχουν διάφορα είδη επικοινωνίας μέσω ανταλλαγής μηνυμάτων. Μία κατηγοριοποίηση είναι σε σύγχρονη και ασύγχρονη (synchronous asynchronous). Στη σύγχρονη επικοινωνία απαιτείται κάποιου είδους ¨χειραψία¨ ανάμεσα στις διεργασίες που διαμοιράζονται δεδομένα. Αυτή η επικοινωνία ονομάζεται και blocking καθώς οι συμμετέχουσες διεργασίες αναστέλλουν οποιαδήποτε λειτουργία τους μέχρι την ολοκλήρωση της επικοινωνίας. Στην ασύγχρονη επικοινωνία η διεργασία αποστολέας στέλνει τα δεδομένα και συνεχίζει την εκτέλεση της χωρίς να περιμένει τη λήψη τους από την διεργασία παραλήπτη. Το χρονικό σημείο στο οποίο θα γίνει η λήψη δεν έχει καμία σημασία. Η ασύγχρονη επικοινωνία ονομάζεται και non-blocking.

Η επικοινωνία διαχωρίζεται επίσης σε επικοινωνία Σημείο-προς-Σημείο (point-to-point, P2P) και συλλογική επικοινωνία (collective). Στην επικοινωνία σημείο-προς-σημείο συμμετέχουν μόνο δυο διεργασίες. Μία εξ' αυτών έχει το ρόλο του αποστολέα και η άλλη του παραλήπτη. Στη συλλογική επικοινωνία παίρνουν μέρος περισσότερες από δύο διεργασίες οι οποίες συνήθως ανήκουν στην ίδια ομάδα ή γκρουπ. Παράδειγμα τέτοιας επικοινωνίας είναι το σενάριο στο οποίο μία διεργασία στέλνει ένα κομμάτι δεδομένων σε όλες τις υπόλοιπες, ή όταν όλες οι διεργασίες χρειάζεται να στείλουν δεδομένα σε μία συγκεκριμένη.

Οι υλοποιήσεις του μοντέλου ανταλλαγής μηνυμάτων στις πιο πολλές περιπτώσεις προσφέρουν στον προγραμματιστή μία βιβλιοθήκη συναρτήσεων για ενσωμάτωση στον πηγαίο κώδικα.Το MPI το οποίο δημιουργήθηκε από το MPI Forum είναι το στάνταρ πρότυπο που καθορίζει το interface των συναρτήσεων που πρέπει να προσφέρει κάθε υλοποίηση αυτού του μοντέλου.

1.3.3 Υβριδικό μοντέλο

Το υβριδικό προγραμματιστικό μοντέλο συνδυάζει τα δύο προηγούμενα. Αυτό το μοντέλο είναι το πιο κατάλληλο για συστοιχίες πολυεπεξεργαστών (clusters). Μέσα στους κόμβους οι διεργασίες επικοινωνούν μέσω της διαμοιραζόμενης μνήμης, ενώ διεργασίες που βρίσκονται σε διαφορετικό κόμβο επικοινωνούν μέσω του δικτύου διασύνδεσης, με ανταλλαγή μηνυμάτων.

Κεφάλαιο 2

Ο αλγόριθμος Conjugate Gradient (CG)

2.1 Παρουσίαση

Ο αλγόριθμος Conjugate Gradient (συζυγών κλίσεων) [3] είναι ένας αλγόριθμος για την αριθμητική επίλυση συστημάτων γραμμικών διαφορικών εξισώσεων της μορφής $Ax = b$, όπου ο πίνακας A είναι πραγματικός και συμμετρικός. Η μέθοδος Conjugate Gradient είναι μία επαναληπτική μέθοδος, κι έτσι μπορεί να εφαρμοστεί σε συστήματα με μεγάλους αραιούς πίνακες, όπου είναι αδύνατον να χρησιμοποιήσουμε κλασσικές μεθόδους, όπως την αποσύνθεση Cholesky (Cholesky decomposition). Τέτοια συστήματα συνήθως προκύπτουν κατά την επίλυση μερικών διαφορικών εξισώσεων. Ο αλγοριθμος CG σε ψευδοκώδικα παρουσιάζεται στον κώδικα 2.1.

Κώδικας 2.1: Ο αλγοριθμος CG σε ψευδοκώδικα.

$r_0 = b - A \cdot x_0$
$p_0 = r_0$
$k = 0$

repeat
  $\alpha_k = \frac{r_k^T \cdot r_k}{p_k^T \cdot A \cdot p}$
  $x_{k+1} = x_k + \alpha_k \cdot p_k$
  $r_{k+1} = r_k - \alpha \cdot A \cdot p_k$
  if r_{k+1} is sufficiently small **then** exit loop **endif**
  $\beta_k = \frac{r_k^T \cdot r_{k+1}}{r_k^T \cdot r_k}$
  $p_{k+1} = r_{k+1} + \beta_k \cdot p_k$
  $k = k + 1$
end repeat
The result is x_{k+1}

Ο λόγος για τον οποίο επιλέξαμε να ασχοληθούμε με την ανάλυση του αλγορίθμου CG είναι ότι χρησιμοποιείται ευρέως σε πολλούς επιστημονικούς τομείς, όπου προκύπτουν συστήματα γραμμικών διαφορικών εξισώσεων με αραιούς πίνακες, όπως είναι η βιολογία, η μετεωρολογία και οι τηλεπικοινωνίες. Επιπρόσθετα, από τη σκοπιά των παράλληλων υπολογιστών παρουσιάζει τις παρακάτω τεχνικές προκλήσεις:

1. **Εξάρτηση από τη μορφή της εισόδου:** Ένα ιδιαίτερο χαρακτηριστικό του CG είναι ότι η ταχύτητα εκτέλεσης του εξαρτάται όχι μόνο από το μέγεθος του αραιού πίνακα αλλά σε μεγάλο βαθμό και από τη δομή του. Αυτό το πρόβλημα γίνεται ιδιαίτερα έντονο στον παράλληλο CG όπου η δομή του πίνακα καθορίζει τη διάσπαση του σε υποπίνακες.

2. **Έντονες προσβάσεις στην κύρια μνήμη:** Μία από τις πράξεις που περιλαμβάνει ο CG είναι ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα. Όπως θα δούμε σε επόμενη παράγραφο η πράξη αυτή έχει πολύ έντονες προσβάσεις στην μνήμη. Έτσι η κλιμακωσιμότητα του CG περιορίζεται σε μεγάλο βαθμό.

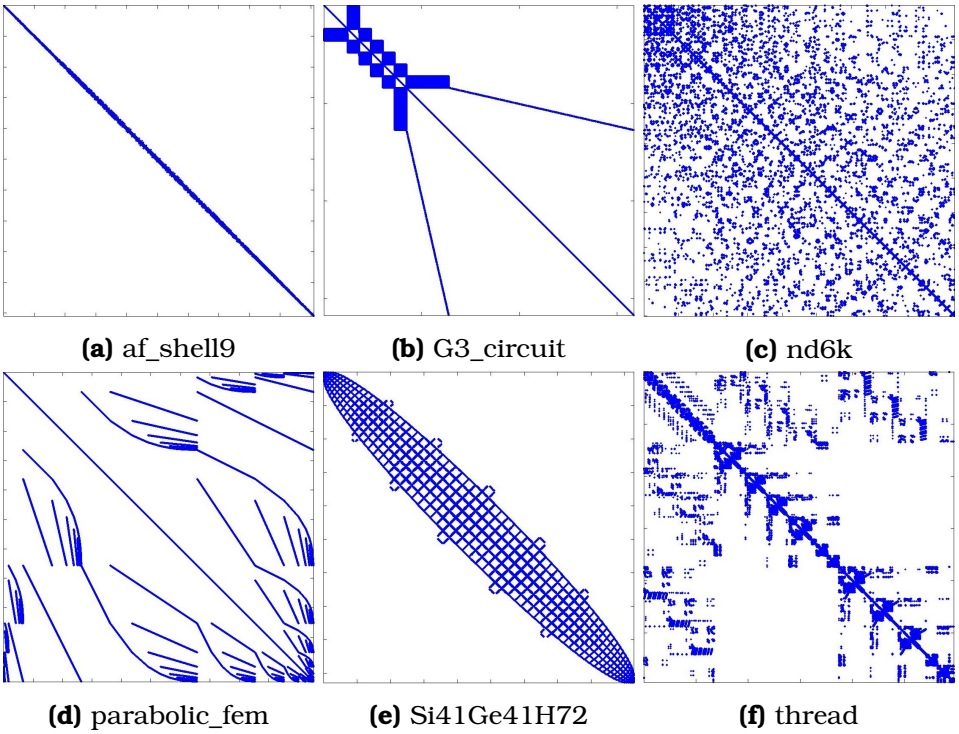
3. **Imbalance:** Ο διαμερισμός του πίνακα και των διανυσμάτων που συμμετέχουν στον CG είναι μια μεγάλη πρόκληση. Όλοι οι επεξεργαστές πρέπει να έχουν ίσο φόρτο εργασίας ώστε να αποφευχθεί το λεγόμενο imbalance, όπου πολλοί επεξεργαστές μένουν ανενεργοί περιμένοντας έναν άλλο να τελειώσει κάποιον υπολογισμό. Κάποιες φορές είναι πρακτικά αδύνατο να βρεθεί ο καλύτερος δυνατός διαμερισμός των δεδομένων, αφού εξαρτάται κατά κύριο λόγο από τη δομή του αραιού πίνακα. Πρόβλημα imbalance προκύπτει και στην επικοινωνία ανάμεσα στους επεξεργαστές, όταν κάποιοι επεξεργαστές έχουν να στείλουν πολύ περισσότερα δεδομένα από κάποιους άλλους.

4. **Επικοινωνία ανάμεσα στους επεξεργαστές:** Μεγάλο μέρος του χρόνου εκτέλεσης του παράλληλου CG σπαταλάται σε ανταλλαγή δεδομένων ανάμεσα στους επεξεργαστές. Η μείωση αυτού του χρόνου είναι ένα πολύ δύσκολο ζήτημα καθώς δεν υπάρχει συγκεκριμένο "μοτίβο" επικοινωνίας αλλά το μέγεθος και το είδος των μηνυμάτων που ανταλλάσσονται εξαρτώνται από τη δομή του αραιού πίνακα.

2.2 Αραιοί πίνακες

Αναφέρθηκε παραπάνω ότι η μέθοδος CG μπορεί να χρησιμοποιηθεί και σε συστήματα με μεγάλους αραιούς πίνακες. Οι αραιοί πίνακες είναι πίνακες που περιέχουν μεγάλο αριθμό μηδενικών στοιχείων. Δεν υπάρχει κάποιο συγκεκριμένο κριτήριο για να χαρακτηριστεί ένας πίνακας ως αραιός ή πυκνός, αλλά ένας πίνακας μπορεί να θεωρηθεί αραιός όταν υπάρχουν πλεονεκτήματα από μια τέτοια, ειδική, αντιμετώπιση (π.χ. μείωση απαιτούμενου χώρου αποθήκευσης). Για παράδειγμα ένα ποσοτικό κριτήριο είναι το εξής: ένας πίνακας NxM χαρακτηρίζεται ως αραιός αν ο αριθμός των μη μηδενικών στοιχείων του είναι τάξεις μεγέθους μικρότερος από

$N \cdot M$. Παραδείγματα αραιών πινάκων που προκύπτουν από πραγματικές εφαρμογές παρουσιάζονται στο Σχημα 2.1



(a) af_shell9 **(b)** G3_circuit **(c)** nd6k

(d) parabolic_fem **(e)** Si41Ge41H72 **(f)** thread

Σχήμα 2.1: Παραδείγματα αραιών πινάκων από πραγματικές εφαρμογές.

Οι αραιοί πίνακες εμφανίζονται σε πληθώρα επιστημονικών εφαρμογών, κυρίως κατά τη μελέτη χαλαρά συνδεδεμένων συστημάτων (loosely coupled systems). Μία από τις κυριότερες εφαρμογές των αραιών πινάκων είναι στην επίλυση ΜΔΕ, όπου το πρόβλημα διακριτοποιείται χρησιμοποιώντας τεχνικές όπως η μέθοδος πεπερασμένων στοιχείων (Finite Element Method - FEM) [3], η οποία συνήθως οδηγεί σε μεγάλους αραιούς πίνακες, αλλά και η μέθοδος CG, όπως αναφέραμε προηγουμένως.

Επιπρόσθετα, η θεωρία των αραιών πινάκων είναι στενά συνδεδεμένη με τη θεωρία των γράφων. Οι αραιοί πίνακες μπορούν να χρησιμοποιηθούν για την αναπαράσταση μεγάλων γράφων χρησιμοποιώντας λίστες γειτνίασης. Χαρακτηριστικό παράδειγμα τέτοιου γράφου είναι ο παγκόσμιος ιστός (World Wide Web - WWW) [4], όπου μία κατευθυνόμενη ακμή από τον κόμβο Α στον κόμβο Β αναπαριστά την παρουσία ενός συνδέσμου από τη σελίδα Α στη σελίδα Β. Από την άλλη μεριά, αλγόριθμοι γράφων χρησιμοποιούνται σε προβλήματα αραιών πινάκων, π.χ. για διαμερισμό των δεδομένων του αραιού πίνακα [5, 6]

2.2.1 Πολλαπλασιασμός αραιού πίνακα με διάνυσμα

Όπως αναφέρθηκε σε προηγούμενη παράγραφο μία από τις πράξεις του CG είναι ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα (SpMxV - Spare Matrix x Vector). Αυτήν είναι και η πράξη του που παρουσιάζει το μεγαλύτερο ερευνητικό ενδιαφέρον καθώς αποτελεί το μεγαλύτερο μέρος του χρόνου εκτέλεσης. Ο SpMxV συναντάται σε πληθώρα υπολογιστικών προβλημάτων και αποτελεί ένα από τα επτά σημαντικότερα υπολογιστικά προβλήματα των επόμενων δεκαετιών [7]. Εκτός από τον CG ο SpMxV είναι η βασική πράξη και για άλλες μεθόδους επαναληπτικής επίλυσης συστημάτων, όπως η Generalized Minimum Residual (GMRES).

Στην πράξη SpMxV ένας αραιός πίνακας NxM πολλαπλασιάζεται με ένα πυκνό διάνυσμα (μεγέθους M) και το αποτέλεσμα που προκύπτει είναι ένα νέο πυκνό διάνυσμα (μεγέθους N): $y = A \cdot x$. Μία γενική έκφραση για τα στοιχεία του y είναι:

$$y_i = \sum_{j=1}^{M} A_{ij} x_j, \quad 1 \leq i \leq N$$

Όπως αναφέραμε και προηγουμένως, η πράξη SpMxV είναι εξαιρετικά απαιτητική σε εύρος ζώνης μνήμης. Συγκεκριμένα σε $O(n^2)$ δεδομένα εκτελεί $O(n^2)$ υπολογισμούς, σε αντίθεση με άλλα αριθμητικά προβλήματα, όπως είναι ο πολλαπλασιασμός πινάκων, όπου σε $O(n^2)$ δεδομένα εκτελούνται $O(n^3)$ υπολογισμοί. Επίσης ο SpMxV παρουσιάζει και μία πληθώρα εγγενών προβλημάτων επίδοσης στις σύγχρονες αρχιτεκτονικές υπολογιστών [8, 9], τα οποία οδηγούν σε πολύ μέτριες επιδόσεις.

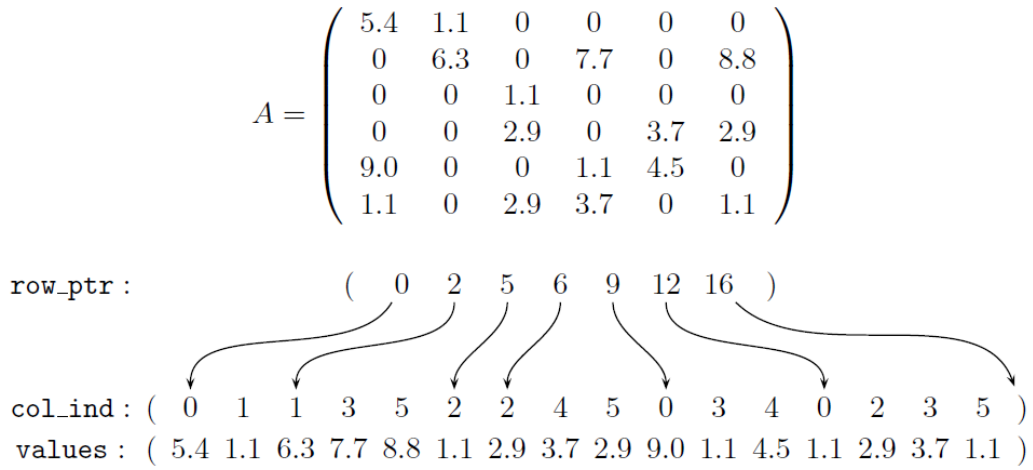
2.2.2 Σχήματα αποθήκευσης αραιών πινάκων

Οι αραιοί πίνακες που προκύπτουν σε πραγματικές επιστημονικές εφαρμογές είναι πολύ μεγάλοι και είναι ασύμφορο, κάποιες φορές αδύνατο, να χρησιμοποιηθούν οι κλασικοί αλγόριθμοι που χρησιμοποιούνται για τους πυκνούς πίνακες, για την αποθήκευση και επεξεργασία τους. Σε αυτήν την περίπτωση θα απαιτούνταν τεράστια υπολογιστική ισχύ και χωρητικότητα μνήμης. Έτσι για την αποθήκευση και επεξεργασία μεγάλων αραιών πινάκων χρησιμοποιούνται ειδικά σχήματα αποθήκευσης και επεξεργασίας.

Τα σχήματα αποθήκευσης αραιών πινάκων είναι δομές δεδομένων που εκμεταλλεύονται την αραιή δομή του πίνακα ώστε να μειώσουν τον απαιτούμενο χώρο στη μνήμη και, κατ' επέκταση, να βελτιώσουν τους αλγορίθμους που χρησιμοποιούνται για την επεξεργασία τους. Γενικά, τα σχήματα αυτά αποθηκεύουν μόνο την απαραίτητη πληροφορία, δηλαδή τα μη μηδενικά αριθμητικά στοιχεία του πίνακα. Ωστόσο, είναι απαραίτητο να αποθηκεύεται επιπλέον πληροφορία σχετική με τη θέση αυτών των στοιχείων. Έτσι τα δεδομένα των αραιών πινάκων διαχωρίζονται σε δύο κατηγορίες: *δεδομένα δομής:* δεδομένα που αναπαριστούν τη δομή του πίνακα και *δεδομένα τιμών:* δεδομένα που αναπαριστούν τις μη μηδενικές αριθμητικές τιμές του πίνακα. Στη συνέχεια παρουσιάζονται ορισμένα από τα πιο δημοφιλή σχήματα αποθήκευσης.

Το σχήμα CSR

Ένα από τα πιο δημοφιλή σχήματα αποθήκευσης αραιών πινάκων είναι το CSR (Compressed Sparse Row) [3, 10]. Για την αναπαράσταση του πίνακα το CSR χρησιμοποιεί τρεις πίνακες: (α) τον πίνακα values, στον οποίο αποθηκεύονται όλα τα μη μηδενικά στοιχεία του πίνακα (β) τον πίνακα row_ptr, στον οποίο αποθηκεύεται η θέση του πρώτου μη μηδενικού στοιχείου κάθε γραμμής και (γ) τον πίνακα col_ind, στον οποίο αποθηκεύεται η στήλη για κάθε μη μηδενικό στοιχείο. Έτσι τα δεδομένα δομής στο σχήμα CSR είναι οι πίνακες row_ptr και col_ind, ενώ τα δεδομένα τιμών είναι ο πίνακας values. Ένα παράδειγμα του σχήματος CSR για έναν πίνακα $6x6$ παρουσιάζεται στο σχήμα 2.2.



Σχήμα 2.2: Παράδειγμα του σχήματος CSR

Οι πίνακες values και col_ind έχουν μέγεθος ίσο με τον αριθμό των μη μηδενικών στοιχείων, ενώ ο πίνακας row_ptr έχει μέγεθος ίσο με τον αριθμό των γραμμών συν ένα. Το σχήμα CSR επιφέρει μία βελτίωση στην πράξη SpMxV αλλά το μειονέκτημα του είναι ότι δεν εκμεταλλεύεται μπλοκ συνεχόμενων στοιχείων του πίνακα όπως κάνουν τα σχήματα αποθήκευσης που θα δούμε παρακάτω. Η υλοποίηση του υπολογιστικού πυρήνα SpMxV για το σχήμα CSR παρουσιάζεται στον Κώδικα 2.2. Ο εξωτερικός βρόχος διατρέχει όλες τις γραμμές μέσω του πίνακα row_ptr, ενώ ο εσωτερικός βρόχος υπολογίζει κάθε στοιχείο του διανύσματος εξόδου.

Κώδικας 2.2: Υλοποίηση της πράξης SpMxV για το σχήμα CSR.

```
for (i=0; i<nrows; i++)
        for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
                y[i] += values[j]*x[col_ind[j]];
```

Το σχήμα CSR-DU

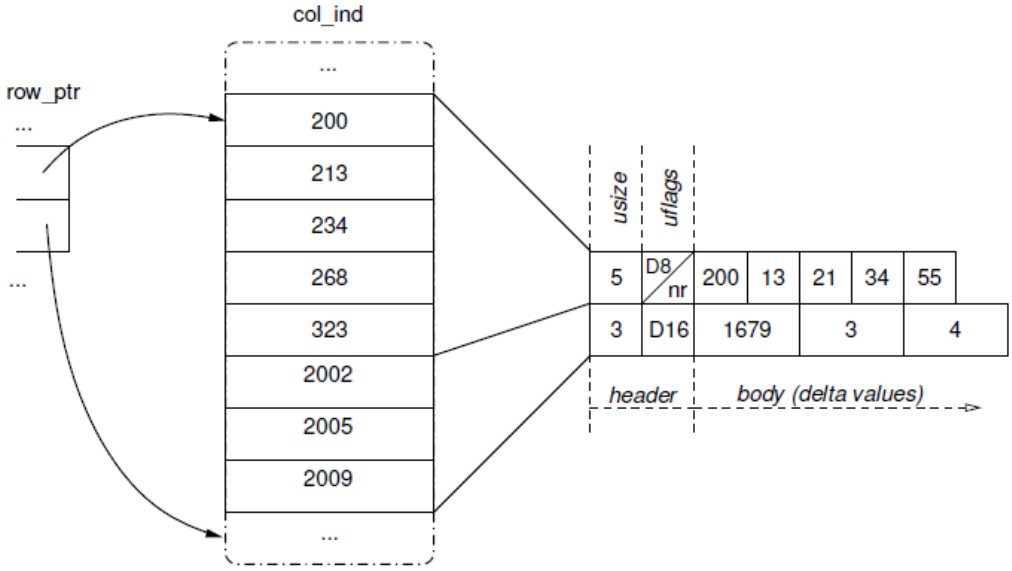
Όπως αναφέραμε προηγουμένως, ένα σημαντικό μειονέκτημα του σχήματος CSR είναι ότι δεν εκμεταλλεύεται συνεχόμενα μπλοκ μη μηδενικών στοιχείων. Οι αραιοί πίνακες, που προκύπτουν σε πραγματικές εφαρμογές, εμφανίζουν συνήθως κάποιες κανονικότητες στη δομή τους και έχουν προταθεί διάφορα σχήματα αποθήκευσης που προσπαθούν να τις εκμεταλλευτούν. Ένα από αυτά είναι και το σχήμα CSR-DU.

Το CSR-DU προσπαθεί να εκμεταλλευτεί πυκνές περιοχές του πίνακα, δηλαδή περιοχές με στοιχεία που βρίσκονται κοντά, χωρίς απαραίτητα να είναι συνεχόμενα. Οι περιοχές αυτές μπορούν να συνεισφέρουν σημαντικά στη μείωση του όγκου των δεδομένων δομής, χρησιμοποιώντας κωδικοποίηση δέλτα (delta encoding) στις στήλες των στοιχείων (πίνακας col_ind). Κατά την εφαρμογή της κωδικοποίησης δέλτα, οι δείκτες στηλών αντικαθιστώνται με τη διαφορά του τρέχοντος δείκτη από τον προηγούμενο. Για τα στοιχεία μιας γραμμής, η διαφορά αυτή είναι θετική και μικρότερη ή ίση από την τιμή του δείκτη. Συνεπώς, οι τιμές δέλτα μπορούν να αποθηκευτούν σε μικρότερου μεγέθους ακέραιους, οδηγώντας σε μείωση του όγκου δεδομένων.

Ο πίνακας χωρίζεται σε περιοχές, οι οποίες ονομάζονται units και έχουν μεταβλητό αριθμό στοιχείων. Για κάθε μία από αυτές τις περιοχές, υπολογίζουμε τη μέγιστη τιμή δέλτα και επιλέγουμε το ελάχιστο μέγεθος ακεραίου που μπορεί να αναπαραστήσει αυτή την τιμή για όλες τις τιμές δέλτα της περιοχής.

Το σχήμα CSR-DU λοιπόν χωρίζει τα δεδομένα δομής σε περιοχές, τις οποίες αποθηκεύει σε έναν πίνακα bytes που ονομάζεται ctl. Η κάθε περιοχή περιορίζεται σε στοιχεία μιας γραμμής και αποτελείται από δύο μέρη.Το πρώτο μέρος είναι η *επικεφαλίδα*, στην οποία αποθηκεύονται οι ιδιότητες της περιοχής, και το δεύτερο μέρος είναι το *σώμα*, όπου αποθηκεύονται κωδικοποιημένες οι τιμές δέλτα. Η επικεφαλίδα περιέχει δύο πεδία μεγέθους ενός byte: (α) το πεδίο usize, που αναπαριστά τον αριθμό των στοιχείων της περιοχής και (β) το πεδίο uflags, ένα διάνυσμα bit (bit-vector) που κωδικοποιεί τα υπόλοιπα χαρακτηριστικά της περιοχής. Αφού το πεδίο usize έχει μέγεθος ένα byte, ο μέγιστος αριθμός στοιχείων για κάθε περιοχή είναι: $2^8 = 256$. Το μέγεθος των δέλτα τιμών (1,2,4 bytes) του σώματος κωδικοποιείται στο uflags, μαζί με πληροφορία για το αν η συγκεκριμένη περιοχή ξεκινά μία νέα γραμμή.

Στο σχήμα 2.3 παρουσιάζεται ένα παράδειγμα του σχήματος CSR-DU, στο οποίο μία σειρά 8 στοιχείων χωρίζεται σε δύο περιοχές.



Σχήμα 2.3: Παράδειγμα του σχήματος CSR-DU

Μία παραλλαγή του σχήματος CSR-DU αποθηκεύει, σε έναν ακέραιο μεταβλητού μεγέθους στην επικεφαλίδα, την απόσταση από την προηγούμενη περιοχή. Έτσι αποφεύγεται το πρόβλημα όπου η πρώτη τιμή δέλτα είναι αρκετά μεγαλύτερη από τις υπόλοιπες και επιβάλλει μεγάλο μέγεθος αποθήκευσης στις υπόλοιπες τιμές. Αυτό συμβαίνει και στην δεύτερη περιοχή του παραδείγματος μας. Μία ακόμα παραλλαγή επιτρέπει την αποδοτική αποθήκευση περιοχών με συνεχόμενα στοιχεία. Σε αυτήν την περίπτωση δεν αποθηκεύουμε τις τιμές δέλτα, αλλά μόνο τον αριθμό των συνεχόμενων στοιχείων της περιοχής.

Η υλοποίηση της πράξης SpMxV για το σχήμα αποθήκευσης CSR-DU παρουσιάζεται στον κώδικα 2.3.

Το σχήμα CSX

Το σχήμα CSR-DU, στοχεύει στην αξιοποίηση περιοχών, στις οποίες τα μη μηδενικά στοιχεία βρίσκονται κοντά. Ωστόσο, λαμβάνει υπόψιν μόνο οριζόντιες περιοχές, που περιλαμβάνουν δηλαδή στοιχεία της ίδιας γραμμής. Η ιδέα των περιοχών θα μπορούσε να επεκταθεί, ώστε να υποστηρίζονται πολλαπλοί τύποι περιοχών, κάθε ένας από τους οποίους θα αντιστοιχεί σε διαφορετικές κανονικότητες που παρουσιάζονται στον πίνακα. Σε αυτήν την περίπτωση, ο πυρήνας SpMxV μπορεί να υλοποιηθεί σε δύο επίπεδα. Στο πρώτο επίπεδο διατρέχονται οι περιοχές, για κάθε μία από τις οποίες (στο δεύτερο επίπεδο) χρησιμοποιείται ειδικευμένη ρουτίνα πολλαπλασιασμού.

Το CSX θεωρεί περιοχές στοιχείων με σταθερή απόσταση, γενικεύοντας την προσέγγιση του σχήματος CSR-DU. Συνεπώς, στοιχεία της μορφής: (α, α+δ, α+2δ, . .

Κώδικας 2.3: Υλοποίηση της πράξης SpMxV για το σχήμα CSR-DU.

```
do {
        usize = ctl_get_u8(ctl);
        uflags = ctl_get_u8(ctl);
        if ( flags_new_row(uflags) ) {
                y_indx++;
                x_indx = 0;
        }
        switch ( flags_type(uflags) ) {
                case CSR_DU_U8:
                for (i=0; i<usize; i++) {
                        x_indx += ctl_get_u8(ctl);
                        y[y_indx] += *(values++) * x[x_indx];
                }
                break;

                case CSR_DU_U16:
                for (i=0; i<usize; i++) {
                        x_indx += ctl_get_u16(ctl);
                        y[y_indx] += *(values++) * x[x_indx];
                }
                break;

                case CSR_DU_U32:
                . . .
        }
} while (values < values_end);
```

Κώδικας 2.4: Υλοποίηση SpMxV για οριζόντιες περιοχές στοιχείων με σταθερή απόσταση.

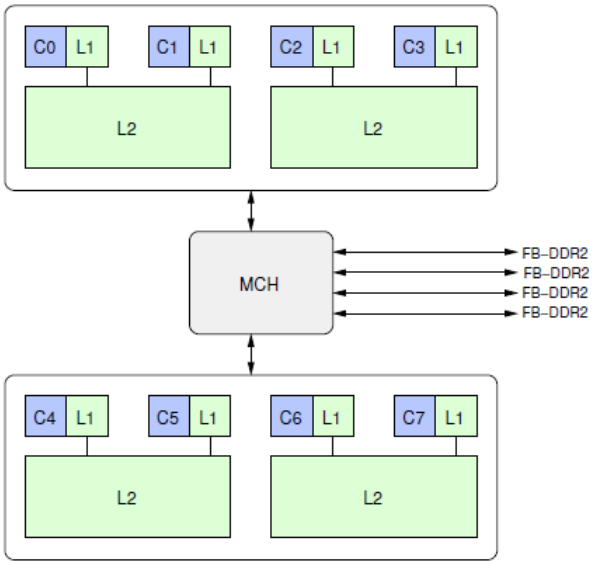
```
xi = x_indx;
yi = y_indx;
for (i=0; i < size; i++) {
        y[yi] += *(values++) * x[xi];
        xi += DELTA;
}
```

.) κωδικοποιούνται χρησιμοποιώντας μόνο το πρώτο στοιχείο, τη σταθερή απόστασή τους και τον αριθμό τους. Η υλοποίηση της πράξης SpMxV για οριζόντιες περιοχές τέτοιων στοιχείων παρουσιάζεται στον κώδικα 2.4. Οι περιοχές αυτές επεκτείνονται ώστε να υποστηρίζουν και διαφορετικές κατευθύνσεις, συγκεκριμένα την κάθετη, τη διαγώνια και την αντί-διαγώνια.

2.3 Πειραματική πλατφόρμα

2.3.1 Υλικό

Η εκτέλεση των πειραμάτων έγινε σε cluster που αποτελείται από δεκαέξι κόμβους, καθένας εκ των οποίων αποτελείται από δύο επεξεργαστές τεσσάρων πυρήνων. Οι κόμβοι συνδέονται σε δίκτυο μεταξύ τους μέσω ενός δικτύου Ethernet. Οι δεκατρείς κόμβοι έχουν επεξεργαστές Intel Clovetown και οι υπόλοιποι τρεις Intel Harpertown. Η αρχιτεκτονική είναι κοινή σε όλους τους κόμβους, φαίνεται στο σχήμα 2.4, η μόνη διαφορά τους είναι το μέγεθος της κρυφής μνήμης δευτέρου επιπέδου (L2 cache). Στον Clovertown έχουμε 8MB (2x4MB), ενώ στον Harpertown 12MB (2x6MB). Και οι δύο επεξεργαστές λειτουργούν σε συχνότητα 2 GHz, ενώ περιέχουν δυο ιδιωτικές κρυφές μνήμες L1 μεγέθους 32 KB (εντολών και δεδομένων).



Σχήμα 2.4: Η αρχιτεκτονική κάθε κόμβου.

2.3.2 Λογισμικό

Σε όλους τους κόμβους είναι εγκατεστημένο το λειτουργικό σύστημα Linux 64-bit στην έκδοση 2.6.38 του πυρήνα. Για την μεταγλώττιση των προγραμμάτων χρησιμοποιήσαμε την έκδοση 4.6.2 του gcc. Για την παραλληλοποίηση του αλγορίθμου χρησιμοποιήθηκαν δύο εργαλεία: η υλοποίηση OpenMPI (έκδοση 1.4.3) του προτύπου ανταλλαγής μηνυμάτων MPI και η βιβλιοθήκη έτοιμων συναρτήσεων OpenMP για τη γλώσσα C.

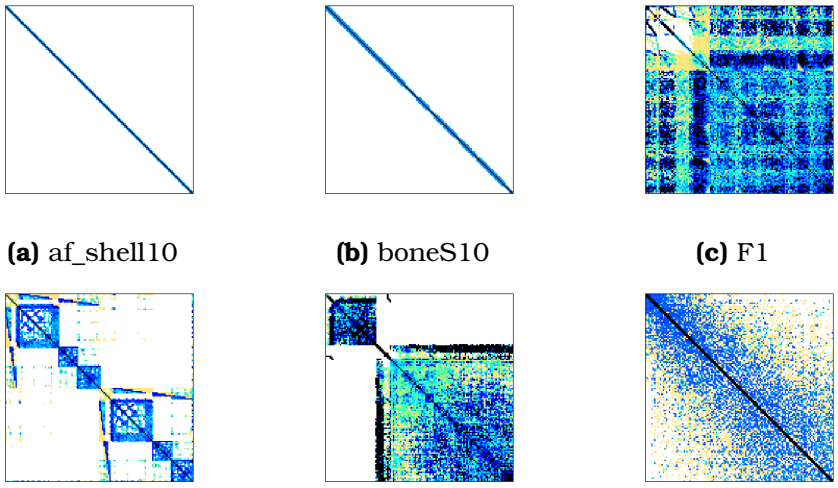
Επιπρόσθετα, περιορίσαμε τις διεργασίες σε συγκεκριμένους επεξεργαστές χρησιμοποιώντας την παράμετρο εκτέλεσης του OpenMPI, –bind-to-core και την κλήση συστήματος sched_setaffinity(). Τέλος χρησιμοποιήσαμε τη βιβλιοθήκη CBLAS, η οποία παρέχει ρουτίνες για βασικές πράξεις γραμμικής άλγεβρας (π.χ. πολλαπλασιασμός δύο διανυσμάτων), και τη βιβλιοθήκη METIS [12, 13] για την αναδιάταξη και το διαμερισμό των πινάκων.

2.3.3 Σύνολο αραιών πινάκων

Για το σκοπό των πειραμάτων χρησιμοποιήσαμε ένα σύνολο έξι αραιών πινάκων οι οποίοι έχουν προκύψει από πραγματικές εφαρμογές. Και οι έξι πίνακες είναι συμμετρικοί και πραγματικοί. Στον πίνακα 2.1 αναφέρονται τα βασικά χαρακτηριστικά τους, ενώ στο σχήμα 2.5 φαίνεται η δομή τους.

όνομα	**N** $(\cdot 10^3)$	**nnz** $(\cdot 10^6)$	**μέγεθος**(MB)
af_shell10	1,508.0	52.6	608.5
boneS10	914.8	55.4	638.2
F1	343.7	26.8	308.4
inline_1	503.7	36.8	423.2
ldoor	952.2	46.5	536.0
nd24k	72.0	28.7	328.9

Πίνακας 2.1: Σύνολο πινάκων που χρησιμοποιήθηκαν στη εκτέλεση των πειραμάτων. Οι στήλες περιέχουν τα χαρακτηριστικά των πινάκων: N είναι ο αριθμός των στηλών και γραμμών του πίνακα σε χιλιάδες, nnz ο αριθμός των μη μηδενικών στοιχείων σε εκατομμύρια και το μέγεθος είναι το μέγεθος του πίνακα όταν αποθηκεύεται με το σχήμα CSR.



(a) af_shell10 **(b)** boneS10 **(c)** F1

(d) inline_1 **(e)** ldoor **(f)** nd24k

Σχήμα 2.5: Η δομή των αραιών πινάκων που χρησιμοποιήθηκαν στην πειραματική εκτέλεση.

2.4 Υλοποίηση του αλγορίθμου CG

2.4.1 Σειριακός

Η υλοποίηση του σειριακού αλγορίθμου CG παρουσιάζεται στον κώδικα 2.5 ενώ στον πίνακα 2.2 παρουσιάζεται εν συντομία η λειτουργία των συναρτήσεων που χρησιμοποιήθηκαν.

Κώδικας 2.5: Υλοποίηση σειριακού αλγορίθμου CG.

```
spm_vec_init_value(x, 0.01);   // x_0

spm_csr_mulv(A, x, z);         // z = A · x_0
spm_vec_axpyz(-1, z, b, r);    // r_0 = b − A · x_0
spm_vec_copy(r, p);            // p_0 = r_0

for (k = 0; k < MAX_ITER; k++){

        spm_csr_mulv(A, p, z); // z = A · p_k

        rr = cblas_ddot(N, rv, 1, rv, 1);
        zp = cblas_ddot(N, zv, 1, pv, 1);
        alpha = rr / zp;   // α = (r_k^T · r_k) / (p_k^T · A · p_k)

        cblas_daxpy(N, alpha, pv, 1, xv, 1); // x_{k+1} = x_k + α · p_k
        cblas_daxpy(N, -alpha, zv, 1, rv, 1); // r_{k+1} = r_k − α · A · p_k

        rr_new = cblas_ddot(N, rv, 1, rv, 1);
        beta = rr_new / rr;        // β_k = (r_{k+1}^T · r_{k+1}) / (r_k^T · r_k)

        spm_vec_axpyz(beta, p, r, p); // p_{k+1} = r_{k+1} + β_k · p_k
}
```

2.4.2 Παράλληλος

Η παραλληλοποίηση του CG είναι μία, σχετικά, εύκολη διαδικασία. Ωστόσο, υπάρχουν δύο σημαντικά θέματα, τα οποία πρέπει να ληφθούν υπόψη: Ο διαμερισμός των δεδομένων (data partitioning) και η ισοκατανομή του φόρτου εργασίας και επικοινωνίας (Load / Communication balance).

Όνομα	Περιγραφή
spm_vec_init_value(x, 0.01);	Αρχικοποιεί το διάνυσμα x στην τιμή 0.01
spm_csr_mulv(A, x, z);	Υλοποιεί τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα. Το διάνυσμα x πολλαπλασιάζεται με τον αραιό πίνακα A και το αποτέλεσμα αποθηκεύεται στο διάνυσμα z.
spm_vec_axpyz(-1, z, b, r);	Το πρώτο όρισμα (εδώ το -1), που είναι ένας ακέραιος, πολλαπλασιάζεται με το διάνυσμα του δεύτερου ορίσματος (εδώ το z) και στο αποτέλεσμα προστίθεται το διάνυσμα του τρίτου ορίσματος (εδώ το b). Το συνολικό αποτέλεσμα αποθηκεύεται στο διάνυσμα r.
spm_vec_copy(r, p);	Τα στοιχεία του διανύσματος r αντιγράφονται στα στοιχεία του διανύσματος p.
cblas_ddot(N, rv, 1, rv, 1);	Ρουτίνα της βιβλιοθήκης CBLAS. Υπολογίζει το εσωτερικό γινόμενο του δεύτερου και τέταρτου ορίσματος (εδώ r * r). Το N είναι η διάσταση των διανυσμάτων και τα άλλα δύο ορίσματα έχουν να κάνουν με την εσωτερική υλοποίηση της συνάρτησης και δε θα μας απασχολήσουν.
cblas_daxpy(N, alpha, pv, 1, xv, 1);	Ρουτίνα της βιβλιοθήκης CBLAS. Πολλαπλασιάζει το διάνυσμα στο τρίτο όρισμα με τον ακέραιο στο δεύτερο όρισμα, στο αποτέλεσμα προστίθεται το διάνυσμα στο πέμπτο όρισμα και στο ίδιο διάνυσμα αποθηκεύεται το τελικό αποτέλεσμα.

Πίνακας 2.2: Περιγραφή των συναρτήσεων που χρησιμοποιούνται στον κώδικα 2.5

Διαμερισμός δεδομένων (Data partitioning)

Η επιλογή κατάλληλου τρόπου διαμοιρασμού των στοιχείων του πίνακα A είναι κρίσιμη για την επίδοση του παράλληλου CG. Υπάρχουν διάφορα σχήματα διαμερισμού δεδομένων για αραιούς πίνακες, εμείς στα πειράματα μας χρησιμοποιήσαμε τον διαμερισμό ανά γραμμές χοντρού κόκκου [14], όπου ο κάθε επεξεργαστής α-

ναλαμβάνει ένα μπλοκ συνεχόμενων γραμμών του πίνακα, και τον διαμερισμό με χρήση ενός αλγορίθμου για διαμερισμό γράφων (METIS). Θα αναφερθούμε εκτενέστερα και στις δύο μεθόδους σε επόμενες παραγράφους.

Σημαντικό ρόλο παίζει και η επιλογή του σχήματος διαμερισμού των διανυσμάτων. Για τα διανύσματα r και x η επιλογή είναι εύκολη, κάθε επεξεργαστής αναλαμβάνει τα στοιχεία των διανυσμάτων που αντιστοιχούν στις γραμμές του πίνακα που του έχουν ανατεθεί. Η διάσπαση του διανύσματος p είναι και το πιο περίπλοκο ζήτημα όσον αφορά τα διανύσματα. Το πρόβλημα έγκειται στο γεγονός ότι οι προσβάσεις που κάνει ο κάθε επεξεργαστής στα στοιχεία του p εξαρτώνται από τη δομή του υπό-πίνακα και είναι αδύνατο να προβλεφθούν. Η πιο απλή λύση είναι να κρατάει ο κάθε επεξεργαστής ένα πλήρες αντίγραφο του p, έτσι όμως έχουμε έντονη επικοινωνία ανάμεσα στους επεξεργαστές. Η δεύτερη λύση που εξετάζουμε είναι ο κάθε επεξεργαστής να λαμβάνει ακριβώς τα στοιχεία που χρειάζεται, κάτι που απαιτεί αρκετό χρόνο preprocessing αλλά μειώνει σε πολύ μεγάλο βαθμό, όπως θα δούμε, την επικοινωνία. Επίσης μπορεί να προκαλέσει imbalance στην επικοινωνία. Παρακάτω θα εξετάσουμε και τις δύο λύσεις με περισσότερη λεπτομέρεια.

Ισοκατανομή φόρτου εργασίας και επικοινωνίας (Load / Communication balance)

Το σημαντικότερο ίσως ζήτημα που προκύπτει κατά την παραλληλοποίηση του CG είναι η ισοκατανομή του φόρτου εργασίας και επικοινωνίας ανάμεσα στους επεξεργαστές. Όσον αφορά το κομμάτι των υπολογισμών η λύση είναι να αναλαμβάνουν όλοι οι επεξεργαστές περίπου ίσο αριθμό μη μηδενικών στοιχείων και γραμμών του πίνακα A. Για την ισοκατανομή του φόρτου επικοινωνίας τη λύση δίνει ο αλγόριθμος METIS, αφού κατά τον διαμερισμό του πίνακα A λαμβάνει υπόψη και το φόρτο επικοινωνίας.

Παρατηρώντας τον σειριακό κώδικα βλέπουμε ότι μέσα στον βρόχο επανάληψης, ο οποίος αποτελεί και το ουσιαστικό κομμάτι του αλγορίθμου, έχουμε την αλληλουχία πράξεων:

$\rightarrow$ spmv: $z = A \cdot p$

$\rightarrow$ ddot: $rr = r \cdot r$

$\rightarrow$ ddot: $zp = z \cdot p = p \cdot A \cdot p$

$\rightarrow$ daxpy: $x = x + \alpha \cdot p$

$\rightarrow$ daxpy: $r = r - \alpha \cdot z = r - \alpha \cdot A \cdot p$

$\rightarrow$ ddot: $rr\_new = r \cdot r$

$\rightarrow$ daxpy: $p = r + \beta \cdot p$

Όσον αφορά στο κομμάτι της επικοινωνίας ανάμεσα στους επεξεργαστές, οι περιπτώσεις όπου απαιτείται ανταλλαγή μηνυμάτων είναι: (α) ένα reduction μετά από κάθε τοπικό εσωτερικό γινόμενο, και (β) ένα gather όπου ο κάθε επεξεργαστής λαμβάνει τα απαραίτητα στοιχεία του διανύσματος p. Έτσι προκύπτει η παρακάτω αλληλουχία πράξεων:

→ spmv: $z = A \cdot p$

→ ddot: $rr = r \cdot r$

→ **reduce:** όλα τα τοπικά $r \cdot r$

→ ddot: $zp = z \cdot p = p \cdot A \cdot p$

→ **reduce:** όλα τα τοπικά $z \cdot p$

→ daxpy: $x = x + \alpha \cdot p$

→ daxpy: $r = r - \alpha \cdot z = r - \alpha \cdot A \cdot p$

→ ddot: $rr\_new = r \cdot r$

→ **reduce:** όλα τα τοπικά $r \cdot r$

→ daxpy: $p = r + \beta \cdot p$

→ **gather:** Κάθε επεξεργαστής στέλνει και λαμβάνει τα στοιχεία του διανύσματος p που χρειάζεται.

Ο κώδικας που υλοποιεί τον παράλληλο αλγόριθμο CG είναι ο κώδικας 2.6. Ο κώδικας αυτός παραμένει αναλλοίωτος σε όλες τις υλοποιήσεις που θα παρουσιάσουμε παρακάτω. Αυτό που αλλάζει είναι ο διαμερισμός του πίνακα A και ο τρόπος διανομής του διανύσματος p στο σύνολο των επεξεργαστών.

Κώδικας 2.6: Υλοποίηση παράλληλου αλγορίθμου CG.

```
for (i = 0; i < MAX_ITER; i++){

        spm_csr_mulv(Al, p, zl); //z = Ap

        lrr = cblas_ddot(l_nr_rows, rlv, 1, rlv, 1);
        MPI_Allreduce(&lrr, &rr, 1, . . .);

        lzp = cblas_ddot(l_nr_rows, zlv, 1, plv, 1);
        MPI_Allreduce(&lzp, &zp, 1, . . .);

        alpha = rr / zp;

        cblas_daxpy(l_nr_rows, alpha, plv, 1, xlv, 1); //x = x + alpha*p
```

```
        cblas_daxpy(l_nr_rows, -alpha, zlv, 1, rlv, 1); //r = r - alpha*A*p

        lrr_new = cblas_ddot(l_nr_rows, rlv, 1, rlv, 1);

        MPI_Allreduce(&lrr_new, &rr_new, 1, . . .);
        beta = rr_new / rr;

        spm_vec_axpyz(beta, pl, rl, pl); // p = r + beta*p

        MPI_Allgatherv(plv, l_nr_rows, . . .);

}
```

2.4.3 Μετρήσεις πειραματικής διαδικασίας

Με βάση όσα είπαμε προηγουμένως το βασικό κομμάτι του παράλληλου αλγόριθμου CG αποτελείται από πέντε βασικά μπλοκ: spmv, ddot, daxpy όσον αφορά το υπολογιστικό μέρος, και reduce, gather όσον αφορά το κομμάτι της επικοινωνίας ανάμεσα στους επεξεργαστές. Αντίστοιχα, οι χρόνοι που θα χρησιμοποιήσουμε για την αξιολόγηση της εκάστοτε υλοποίησης είναι:

1. **spmv:** Ο χρόνος που σπαταλάται στην εκτέλεση του πολλαπλασιασμού πίνακα με διάνυσμα.
2. **ddot:** Ο χρόνος για την εκτέλεση των τριών εσωτερικών γινομένων.
3. **daxpy:** Ο χρόνος για την εκτέλεση των δυο πράξεων πολλαπλασιασμού διανύσματος με σταθερά και πρόσθεση σε διάνυσμα.
4. **reduce:** Ο χρόνος που σπαταλάται στο reduction των επιμέρους εσωτερικών γινομένων για τον υπολογισμό του συνολικού.
5. **gather:** Ο χρόνος που απαιτείται ώστε όλοι οι επεξεργαστές να στείλουν και να λάβουν τα κομμάτια του διανύσματος p που απαιτούνται.
6. **total:** Ο συνολικός χρόνος εκτέλεσης του αλγορίθμου, που προκύπτει ως άθροισμα των πέντε παραπάνω χρόνων.

2.5 CG Scatter

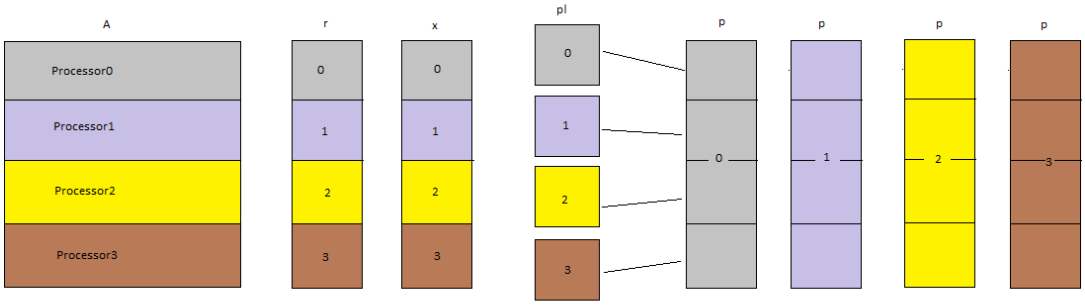
2.5.1 Υλοποίηση

Σε αυτήν την πρώτη υλοποίηση του παράλληλου αλγορίθμου CG έχουμε κάνει τις εξής επιλογές:

→ Ο πίνακας A διασπάται, με τη μέθοδο του διαμερισμού ανά γραμμές χοντρού κόκκου, σε οριζόντια μπλοκ γραμμών με περίπου ίσο αριθμό μη μηδενικών στοιχείων. Σε κάθε επεξεργαστή ανατίθεται και ένα μπλοκ. Στον ίδιο επεξεργαστή ανατίθενται τα αντίστοιχα μπλοκ των διανυσμάτων r, x. Έτσι σε κάθε επεξεργαστή έχουμε έναν τοπικό υπό-πίνακα Al και δυο τοπικά διανύσματα rl, xl.

→ Κάθε επεξεργαστής αναλαμβάνει να υπολογίσει το αντίστοιχο μπλοκ του διανύσματος p και να το στείλει σε όλους τους άλλους. Έτσι έχουμε σε κάθε επεξεργαστή ένα τοπικό διάνυσμα pl και ένα τοπικό πλήρες αντίγραφο του p.

Ο διαμερισμός των δεδομένων για την υλοποίηση CG Scatter παρουσιάζεται γραφικά στο σχήμα 2.6.



Σχήμα 2.6: Ο διαμερισμός των δεδομένων για την υλοποίηση CG Scatter

Το πρόβλημα σε αυτήν την υλοποίηση, όπως θα φανεί και στα πειράματα, είναι ότι απαιτείται πολύ μεγάλος όγκος επικοινωνίας. Κάθε επεξεργαστής στέλνει σε όλους τους υπόλοιπους το τοπικό διάνυσμα pl. Έτσι αν έχουμε p επεξεργαστές και το μέγεθος του διανύσματος p είναι N κάθε επεξεργαστής θα αναλάβει $\frac{N}{p}$ στοιχεία. Σε κάθε επανάληψη του κυρίως βρόχου κάθε επεξεργαστής θα στέλνει p-1 μηνύματα μεγέθους $\frac{N}{p}$. Οπότε συνολικά ο όγκος της επικοινωνίας που απαιτείται σε κάθε επανάληψη είναι $p \times \frac{N}{p} \times (p-1) = N \times (p-1)$. Στον πίνακα 2.3 παρουσιάζεται ο συνολικός όγκος επικοινωνίας που πραγματοποιείται σε κάθε επανάληψη του κυρίως βρόχου.

Όνομα	**N** $(\cdot 10^3)$	**Όγκος επικοινωνίας σε MB**			
		Αριθμός επεξεργαστών			
		2	4	8	16
af_shell10	1,508.0	11.51	34.52	80.54	172.58
boneS10	914.8	6.98	20.94	48.86	104.7
F1	343.7	2.62	7.87	18.36	39.34
inline_1	503.7	3.84	11.53	26.9	57.65
ldoor	952.2	7.26	21.79	50.85	108.97
nd24k	72.0	0.55	1.65	3.85	8.24
		Αριθμός επεξεργαστών			
		32	64	96	128
af_shell10	1,508.0	356.67	724.85	1093.03	1461.21
boneS10	914.8	216.38	439.75	663.11	886.47
F1	343.7	81.31	165.24	249.18	333.11
inline_1	503.7	119.13	242.11	365.09	488.06
ldoor	952.2	225.21	457.68	690.15	922.62
nd24k	72.0	17.03	34.61	52.19	69.76

Πίνακας 2.3: Ο όγκος επικοινωνίας για κάθε εκτέλεση του κυρίως βρόχου του προγράμματος, για κάθε πίνακα και κάθε διαφορετικό αριθμό επεξεργαστών, στην υλοποίηση Scatter. Τα στοιχεία του p έχουν μέγεθος 8 bytes συνεπώς ο όγκος επικοινωνίας σε κάθε περίπτωση υπολογίζεται ως $\frac{N*(p-1)*8}{2^{20}}$.

Από την άλλη η υλοποίηση αυτήν αναμένουμε ότι δε θα παρουσιάζει πρόβλημα imbalance αφού τα μη μηδενικά στοιχεία μοιράζονται εξίσου στο σύνολο των επεξεργαστών (ισορροπία όσον αφορά στο κομμάτι των υπολογισμών) και ο κάθε επεξεργαστής έχει περίπου ίσο αριθμό στοιχείων του p που πρέπει να στείλει (ισορροπία όσον αφορά στο κομμάτι της επικοινωνίας).

2.5.2 Πειραματική αξιολόγηση

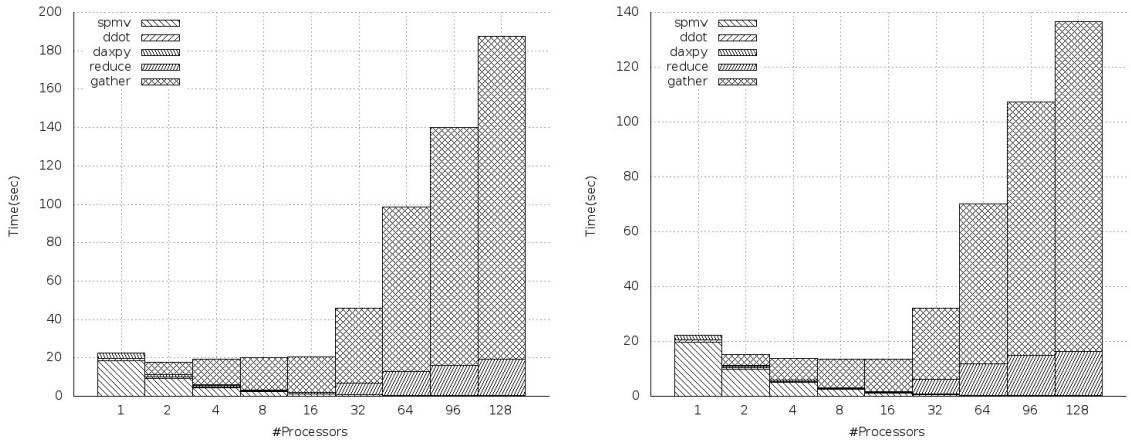
Στο σχήμα 2.7 φαίνεται για κάθε πίνακα και κάθε διαφορετικό αριθμό επεξεργαστών η κατανομή του συνολικού χρόνου. Στον σειριακό αλγόριθμο ο spmv καταλαμβάνει το μεγαλύτερο μέρος του χρόνου εκτέλεσης αλλά όσο αυξάνονται οι επεξεργαστές, το gather γίνεται ο κυρίαρχος όρος.

Στο σχήμα 2.8 παρουσιάζονται όλοι οι χρόνοι για όλους τους πίνακες. Βλέπουμε ότι οι λειτουργίες spmv, ddot, daxpy κλιμακώνουν σε αρκετά ικανοποιητικό βαθμό. Το gather, όπως ήταν αναμενόμενο, αυξάνεται εκθετικά με την αύξηση των επεξεργαστών. Επίσης οι πίνακες με μεγαλύτερο αριθμό γραμμών έχουν και υψηλότερο gather.

Το reduce έχει μία συμπεριφορά που χρειάζεται να εξετάσουμε αναλυτικότερα.

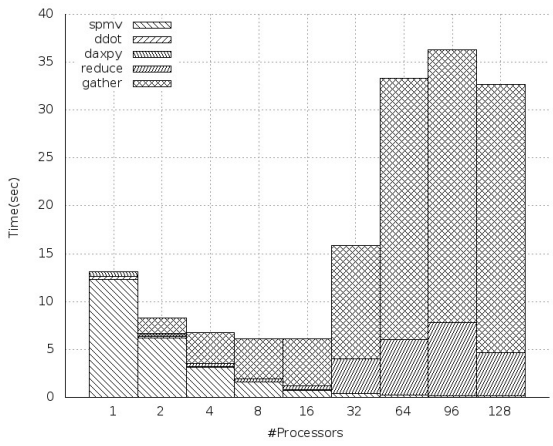
Το reduce είναι μία λειτουργία της οποίας ο χρόνος εκτέλεσης δεν εξαρτάται από το μέγεθος και τη δομή του πίνακα, αλλά μόνο από τον αριθμό των επεξεργαστών. Έτσι θα περιμέναμε ότι για κάθε πίνακα θα είχαμε περίπου ίσο reduce για ίδιο αριθμό επεξεργαστών, κάτι το οποίο δε συμβαίνει όπως φαίνεται και στο σχήμα 2.8d. Ενδεχόμενο imbalance, είτε στο κομμάτι των υπολογισμών είτε στο κομμάτι της επικοινωνίας, ανάμεσα στις διεργασίες θα εξηγούσε την αύξηση του μέσου reduce. Στο σχήμα 2.9 παρουσιάζεται για κάθε λειτουργία η διαφορά μεταξύ των διεργασιών που έχουν τον μέγιστο και ελάχιστο χρόνο, ως ένα μέτρο του imbalance για κάθε ξεχωριστή λειτουργία. Βλέπουμε ότι σε όλους του πίνακες το imbalance στο gather ακολουθεί το imbalance του reduce. Αυτό μας οδηγεί στο συμπέρασμα ότι έχουμε πρόβλημα imbalance στο gather το οποίο αποτυπώνεται και στο reduce αφού διεργασίες με μικρό gather έχουν μεγάλο χρόνο αναμονής στο reduce και αντίθετα. Όμως, όπως αναφέρθηκε προηγουμένως, ο φόρτος υπολογισμών και επικοινωνίας είναι περίπου ίσος για κάθε διεργασία, άρα οδηγούμαστε στην υπόθεση ότι το imbalance οφείλεται σε συμφόρηση του διαύλου της κοινής μνήμης που χρησιμοποιείται για αποστολή / λήψη μηνυμάτων ανάμεσα σε επεξεργαστές που ανήκουν στον ίδιο κόμβο. Αυτός ο ισχυρισμός ενισχύεται από το γεγονός ότι το imbalance γίνεται αρκετά μεγάλο αφού ξεπεράσουμε τους 16 επεξεργαστές, αφού δηλαδή αρχίσουμε να τοποθετούμε περισσότερες από μία διεργασίες σε κάθε κόμβο.

Αξιοσημείωτο είναι επίσης το γεγονός ότι παρατηρούμε πολύ μεγάλη αύξηση στον ελάχιστο χρόνο reduce. Θα περίμενε κανείς ότι τουλάχιστον μία διεργασία (αυτή με τον υψηλότερο χρόνο gather) θα είχε πολύ μικρό reduce. Όμως όπως παρατηρούμε στο σχήμα 2.10 ακόμα και το ελάχιστο reduce είναι αρκετά υψηλό για περισσότερες από 16 διεργασίες. Επίσης βλέπουμε ότι το ελάχιστο reduce αυξάνεται ανάλογα με τον αριθμό των γραμμών του πίνακα (μεγαλύτεροι πίνακες έχουν υψηλότερο ελάχιστο reduce). Οι δύο τελευταίες επισημάνσεις μας οδηγούν στο συμπέρασμα ότι και η αύξηση του reduce οφείλεται στη συμφόρηση των διαύλων κοινής μνήμης των κόμβων.

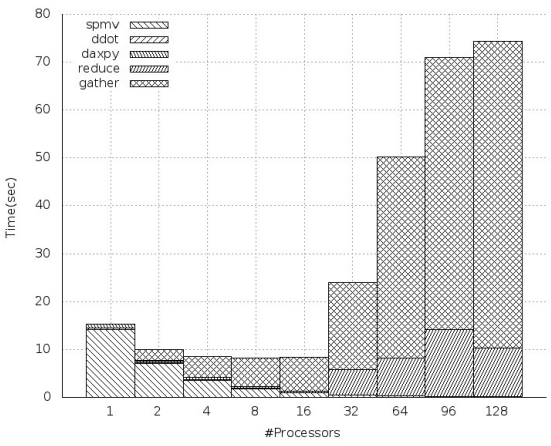


(a) af_shell10

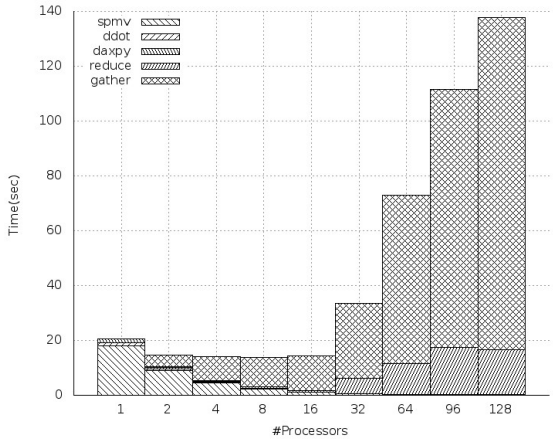
(b) boneS10



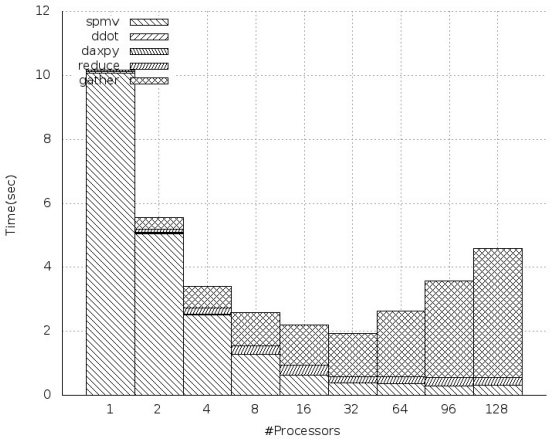
(c) F1



(d) inline_1

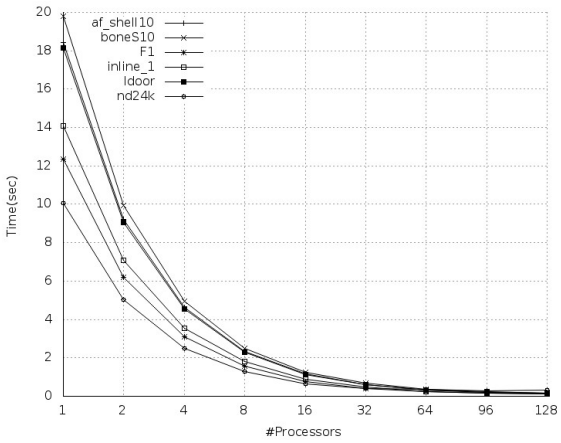


(e) ldoor

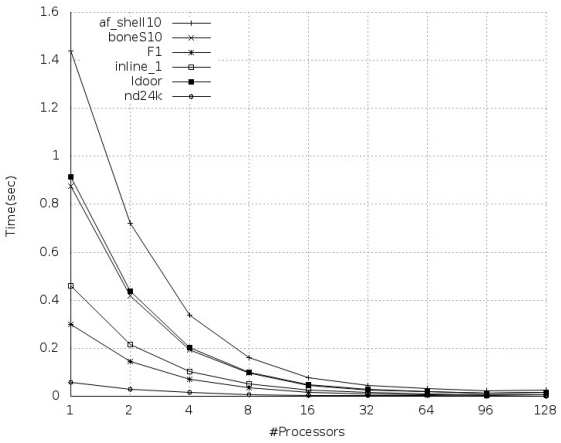


(f) nd24k

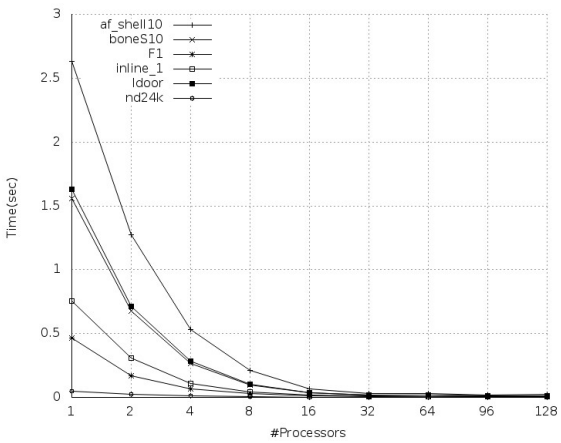
Σχήμα 2.7: Κατανομή του συνολικού χρόνου εκτέλεσης για κάθε πίνακα στην υλοποίηση CG Scatter.



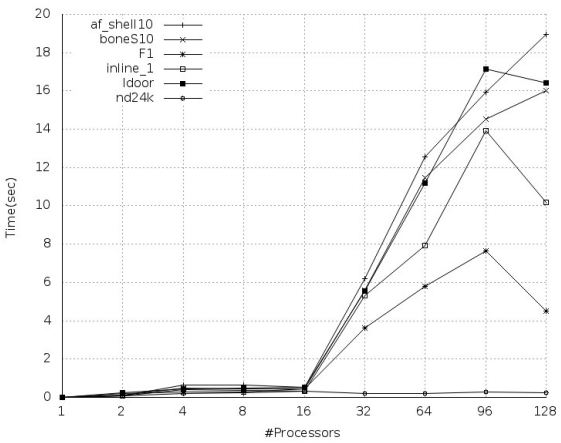
(a) spmv



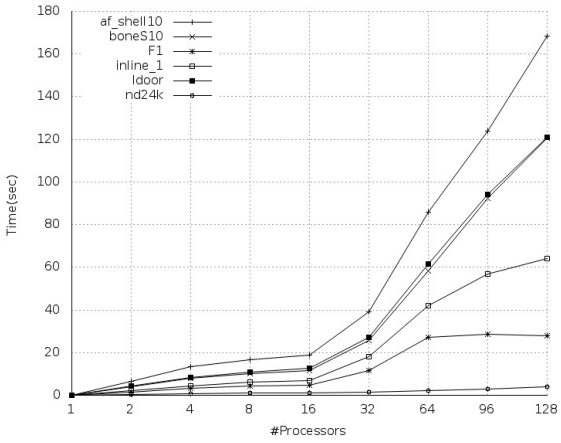
(b) ddot



(c) daxpy

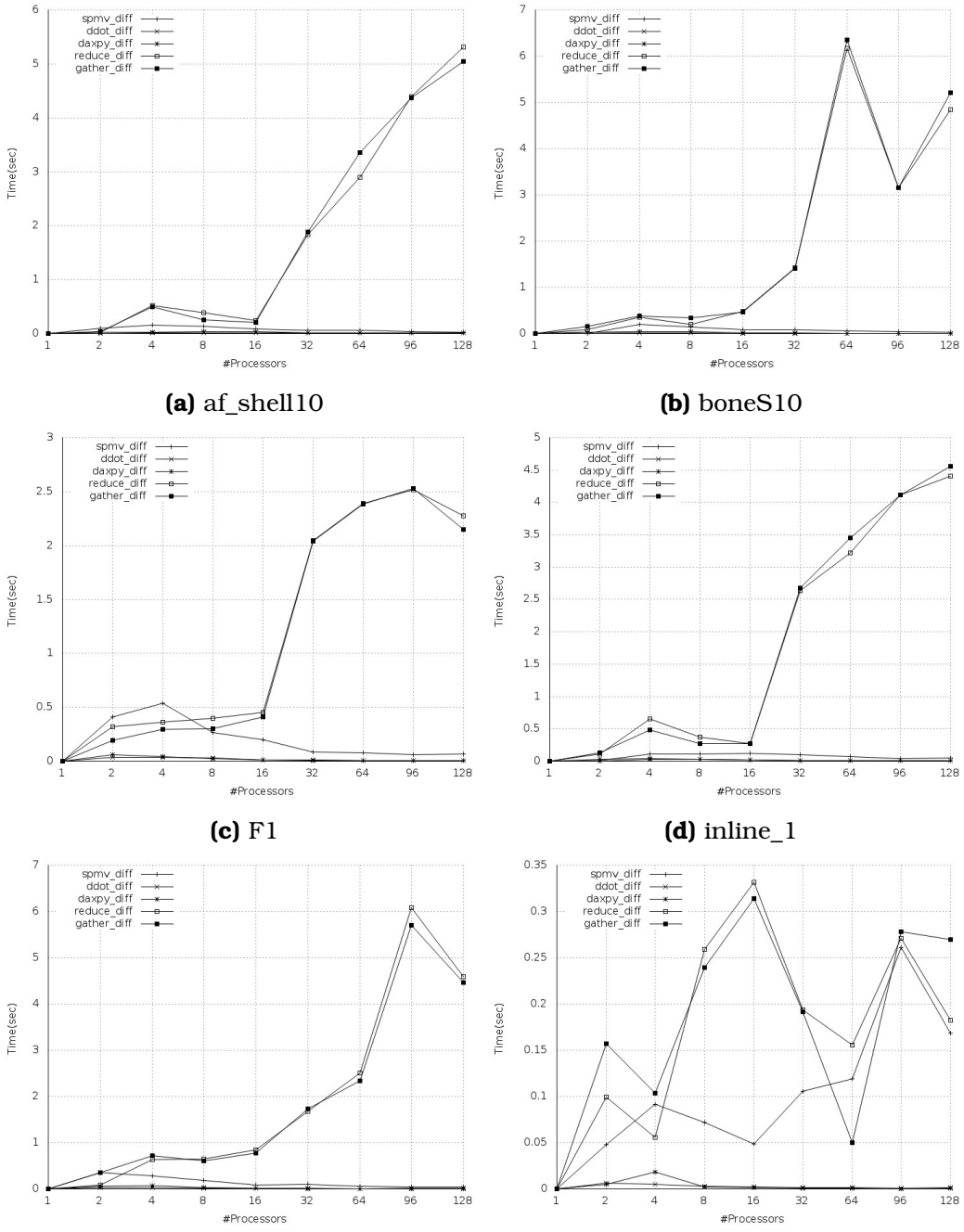


(d) reduce



(e) gather

Σχήμα 2.8: Χρόνοι εκτέλεσης για κάθε πίνακα στην υλοποίηση CG Scatter.

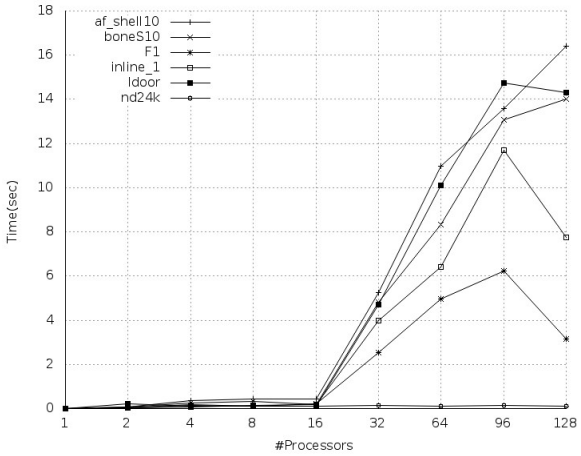


(a) af_shell10 **(b)** boneS10

(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

Σχήμα 2.9: Διαφορά μεγίστου - ελαχίστου για κάθε χρόνο για κάθε πίνακα στην υλοποίηση CG Scatter.

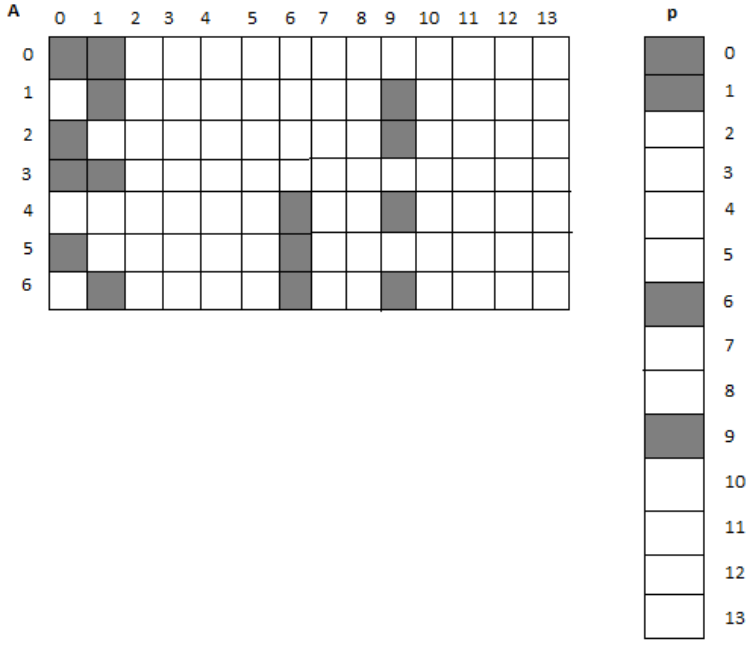


Σχήμα 2.10: Το ελάχιστο reduce για όλους τους πίνακες στην υλοποίηση CG Scatter.

2.6 CG processor-to-processor (P2P)

2.6.1 Υλοποίηση

Όπως είδαμε προηγουμένως το κυριότερο πρόβλημα της Scatter υλοποίησης είναι ο μεγάλος όγκος επικοινωνίας που απαιτείται ανάμεσα στους επεξεργαστές, και αφορά το κομμάτι του διαμοιρασμού του διανύσματος p. Κάθε επεξεργαστής, αφού υπολογίσει το μπλοκ του διανύσματος p που του έχει ανατεθεί, το στέλνει σε όλους τους υπόλοιπους επεξεργαστές. Όμως στην πράξη spmv που συμμετέχει το διάνυσμα p χρησιμοποιείται μόνο ένα μικρό κομμάτι των στοιχείων του σε κάθε επεξεργαστή, λόγω της αραιής δομής του τοπικού πίνακα Al. Για παράδειγμα στο σχήμα 2.11, όπου ένας αραιός πίνακας $7x14$ (με σκούρο χρώμα φαίνονται τα μη μηδενικά στοιχεία του) πολλαπλασιάζεται με ένα διάνυσμα 14 στοιχείων, μόνο τα 4 από τα 14 στοιχεία του διανύσματος χρησιμοποιούνται. Με βάση τα παραπάνω καταλήγουμε στο συμπέρασμα ότι μεγάλο μέρος των μηνυμάτων που ανταλλάσσονται μεταξύ των επεξεργαστών στην υλοποίηση scatter δεν είναι απαραίτητο.



Σχήμα 2.11: Πολλαπλασιασμός αραιού πίνακα με διάνυσμα. Βλέπουμε ότι μόνο ένα μικρό μέρος των στοιχείων του διανύσματος χρησιμοποιούνται.

Το επόμενο βήμα λοιπόν είναι η υλοποίηση P2P (Processor-to-Processor). Σε αυτήν την υλοποίηση κάθε επεξεργαστής κρατάει πληροφορίες σχετικά με το ποια

στοιχεία του p του ανήκουν και ποια στοιχεία του πρέπει να στείλει ή να παραλάβει και σε ποιους επεξεργαστές. Έχουμε λοιπόν τις πληροφορίες που αφορούν την αποστολή δεδομένων από κάποιο επεξεργαστή προς τους άλλους όπου έχουμε τις παρακάτω τοπικές μεταβλητές:

→ Μία μεταβλητή *p_nr_rows_send*, όπου αποθηκεύεται ο συνολικός αριθμός των στοιχείων του p που θα στείλει προς άλλους επεξεργαστές.

→ Δύο πίνακες *p_rows_to_send[]* και *p_rows_receivers[]*. Και οι δύο έχουν μέγεθος *p_nr_rows_send*. Στον πρώτο πίνακα κρατάμε τον αριθμό της στήλης κάθε στοιχείου του p που πρέπει να στείλει ο επεξεργαστής και στον δεύτερο πίνακα το αναγνωριστικό του επεξεργαστή στον οποίο θα αποσταλεί το αντίστοιχο στοιχείο.

→ Έναν πίνακα *p_nr_rows_to_send_to_each_proc[]*, μεγέθους ίσου με τον αριθμό των επεξεργαστών. Σε κάθε θέση i του εν λόγω πίνακα αποθηκεύεται ο αριθμός των στοιχείων του p που ο επεξεργαστής θα στείλει στον i-οστό επεξεργαστή.

Στις πληροφορίες που κρατάει ο επεξεργαστής και αφορούν την παραλαβή στοιχείων του διανύσματος p έχουμε:

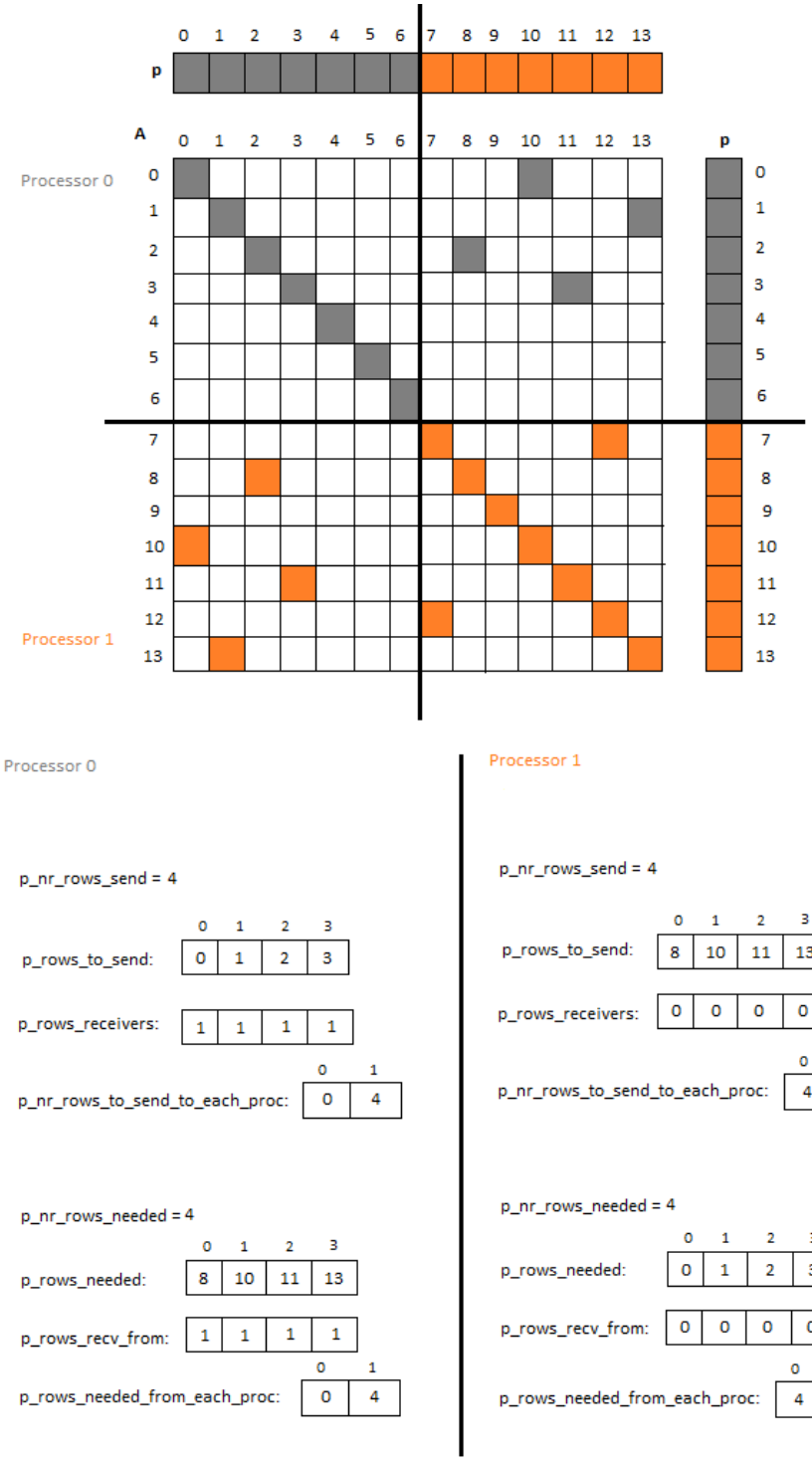
→ Μία μεταβλητή *p_nr_rows_needed*, όπου αποθηκεύεται ο αριθμός των στοιχείων του p που πρέπει να παραλάβει από άλλους επεξεργαστές.

→ Δύο πίνακες *p_rows_needed[]* και *p_rows_recv_from[]*. Και οι δύο έχουν μέγεθος *p_nr_rows_needed*. Στον πρώτο πίνακα κρατάμε τον αριθμό της στήλης κάθε στοιχείου του p που πρέπει να παραλάβει ο επεξεργαστής και στον δεύτερο πίνακα το αναγνωριστικό του επεξεργαστή από τον οποίο θα αποσταλεί το αντίστοιχο στοιχείο.

→ Έναν πίνακα *p_nr_rows_needed_from_each_proc[]*, με μέγεθος ίσο με τον αριθμό των επεξεργαστών. Σε κάθε θέση i του εν λόγω πίνακα αποθηκεύεται ο αριθμός των στοιχείων του p που ο επεξεργαστής θα παραλάβει από τον i-οστό επεξεργαστή.

Στο σχήμα 2.12 φαίνονται οι πληροφορίες που κρατάνε οι δύο επεξεργαστές για το διαμερισμό ενός πίνακα $14x14$. Για τον ίδιο πίνακα ο όγκος επικοινωνίας που θα απαιτούνταν για την υλοποίηση Scatter είναι $N \times (p-1) = 14 \times (2-1) = 14$ στοιχεία, αντίθετα τώρα ο απαιτούμενος όγκος είναι 8 στοιχεία (4 από τον processor 0 προς τον processor 1 και 4 προς την αντίθετη κατεύθυνση). Έχουμε δηλαδή 42% μείωση φόρτου επικοινωνίας.

Στο πίνακα 2.4 παρουσιάζεται για κάθε πίνακα και για περισσότερους από 16 επεξεργαστές ο όγκος επικοινωνίας που απαιτείται για την εκτέλεση κάθε επανάληψης του βρόχου της υλοποίησης P2P, καθώς και η επί τοις εκατό μείωση του σε σχέση με τον αντίστοιχο όγκο επικοινωνία της υλοποίησης Scatter.



Σχήμα 2.12: Δεδομένα που αποθηκεύονται σε κάθε επεξεργαστή και αφορούν το διαμοιρασμό του διανύσματος p.

Όνομα	Αριθμός επεξεργαστών	Όγκος επικοινωνίας P2P (MB)	Μείωση σε σχέση με το Scatter (%)
af_shell10	16	0.6	99.65
	32	1.24	99.65
	64	2.53	99.65
	96	3.81	99.65
	128	5.1	99.65
boneS10	16	1.6	98.47
	32	3.31	98.47
	64	6.73	98.47
	96	10.16	98.47
	128	12.69	98.57
F1	16	15.99	59.35
	32	19.64	75.85
	64	22.71	86.26
	96	24.09	90.33
	128	24.97	92.5
inline_1	16	9.85	82.91
	32	13.06	89.04
	64	15.7	93.52
	96	17.21	95.29
	128	18.23	96.26
ldoor	16	10.34	90.51
	32	12.85	94.29
	64	15.65	96.58
	96	16.92	97.55
	128	17.72	98.08
nd24k	16	3.74	54.61
	32	5.24	69.23
	64	6.24	81.97
	96	6.67	87.22
	128	7	89.97

Πίνακας 2.4: Ο όγκος επικοινωνίας για κάθε εκτέλεση του κυρίως βρόχου του προγράμματος, για κάθε πίνακα και κάθε διαφορετικό αριθμό επεξεργαστών, στην υλοποίηση P2P.

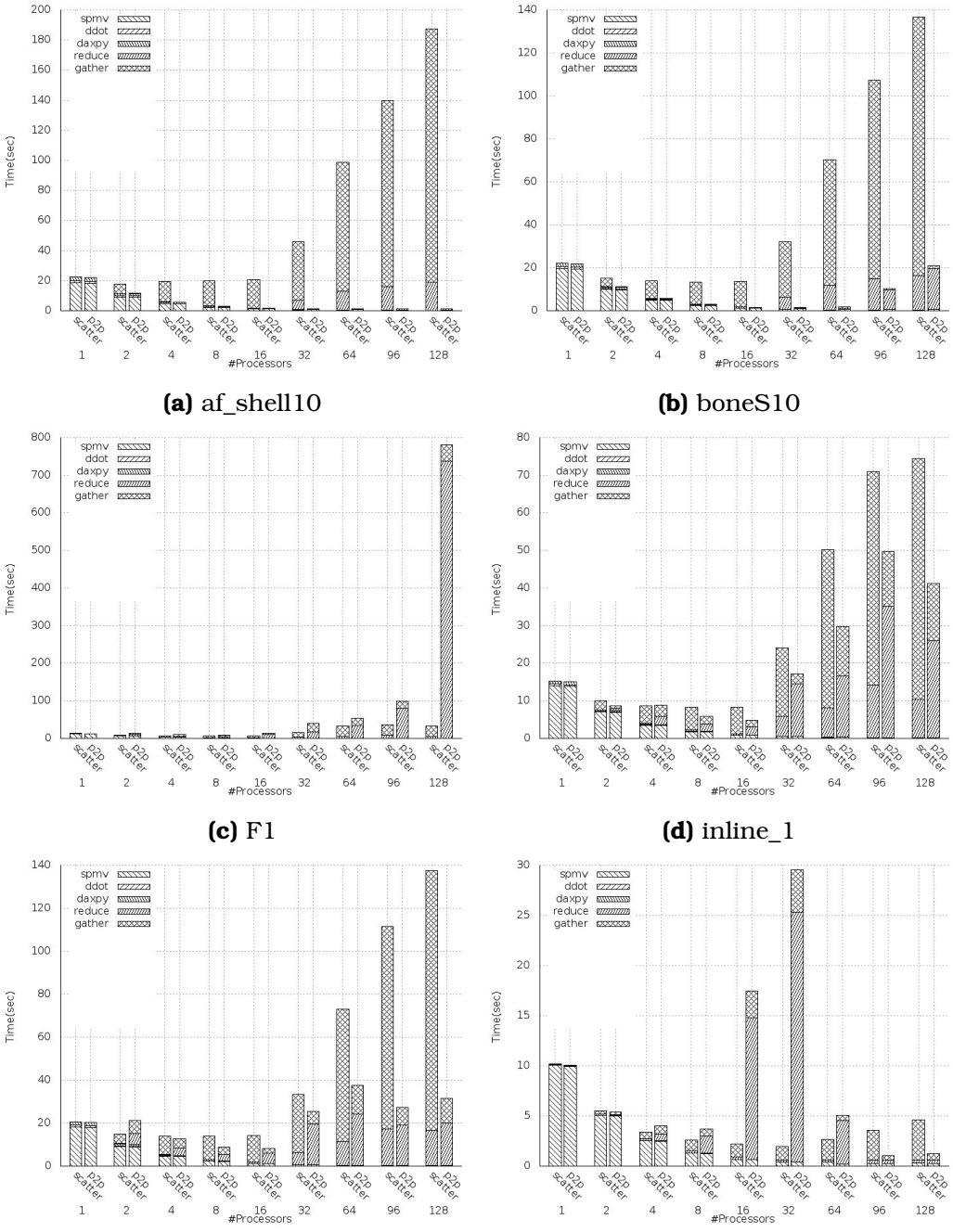
2.6.2 Πειραματική αξιολόγηση

Στο σχήμα 2.13 παρουσιάζονται οι μέσοι χρόνοι εκτέλεσης για την υλοποίηση P2P, σε αντιπαραβολή με τους αντίστοιχους χρόνους της Scatter. Οι χρόνοι spmv, ddot, daxpy δεν μεταβάλλονται αφού το κομμάτι των υπολογισμών σε κάθε επεξεργαστή παραμένει ίδιο με την υλοποίηση Scatter. Ο μέσος χρόνος gather μειώνεται σε κάθε περίπτωση, αφού έχουμε μειώσει τον όγκο της απαιτούμενης επικοινωνίας ανάμεσα στους επεξεργαστές. Αντίθετα το μέσο reduce, στους πίνακες F1, inline_1, ldoor, nd24k, παρουσιάζει πολύ μεγάλη αύξηση. Αυτό ισχυριζόμαστε ότι οφείλεται σε communication imbalance ανάμεσα στις διεργασίες.

Πράγματι, όπως φαίνεται στο σχήμα 2.14, όπου παρουσιάζονται οι διαφορές μεγίστου - ελαχίστου για κάθε πίνακα, οι πίνακες που παρουσιάζουν αύξηση στο μέσο reduce έχουν πολύ μεγάλη διακύμανση στους χρόνους gather μεταξύ των διεργασιών, δηλαδή πολύ μεγάλο communication imbalance. Για παράδειγμα ο πίνακας F1 στις 128 διεργασίες έχει 250 δευτερόλεπτα διαφορά ανάμεσα στο μέγιστο και το ελάχιστο gather, που σημαίνει ότι η διεργασία με το ελάχιστο θα ¨περιμένει¨ τουλάχιστον 250 δευτερόλεπτα κολλημένη στο reduce. Αντίθετα ο πίνακας af_shell10 παρουσιάζει ελάχιστη διακύμανση στο gather και παρουσιάζει πολύ υψηλή βελτίωση σε σχέση με την υλοποίηση Scatter.

Πέρα από το communication imbalance ανάμεσα στις διεργασίες η υλοποίηση P2P έχει ακόμα ένα πρόβλημα. Παρ' όλο που μειώνει σε πολύ μεγάλο ποσοστό τον απαιτούμενο όγκο επικοινωνίας εξακολουθεί να εμφανίζει πολύ μεγάλο ελάχιστο χρόνο reduce, γεγονός που σημαίνει πως παρά τη μείωση της επικοινωνίας εξακολουθεί να υπάρχει το πρόβλημα της συμφόρησης του διαύλου της κοινής μνήμης των κόμβων. Επιπρόσθετα μπορούμε να συμπεράνουμε ότι η εμφάνιση του communication imbalance επιδεινώνει αυτό το πρόβλημα. Όπως φαίνεται από το σχήμα 2.15 όλοι οι πίνακες, εκτός από τον af_shell10, ο οποίος όπως είδαμε προηγουμένως δεν εμφανίζει καθόλου imbalance, εμφανίζουν πολύ υψηλούς χρόνους ελάχιστου reduce με αποκορύφωμα τον πίνακα F1 ο οποίος έχει περίπου 40 και 520 δευτερόλεπτα ελάχιστο reduce, στις 96 και 128 διεργασίες αντίστοιχα.

Ενδιαφέρον παρουσιάζει ο πίνακας nd24k ο οποίος στη Scatter υλοποίηση δεν αντιμετώπιζε πρόβλημα με τη συμφόρηση του διαύλου κοινής μνήμης αλλά στην υλοποίηση P2P, για 16 και 32 διεργασίες, παρουσιάζει υψηλό gather imbalance (σχήμα 2.14φ) και για αυτό το λόγο έχει και ιδιαίτερα υψηλό ελάχιστο reduce. Επίσης ο πίνακας F1, για 96 και 128 διεργασίες, αντιμετωπίζει τεράστιο πρόβλημα imbalance στο gather (σχήμα 2.14ς) και ως εκ τούτου έχει και πολύ υψηλό ελάχιστο reduce.

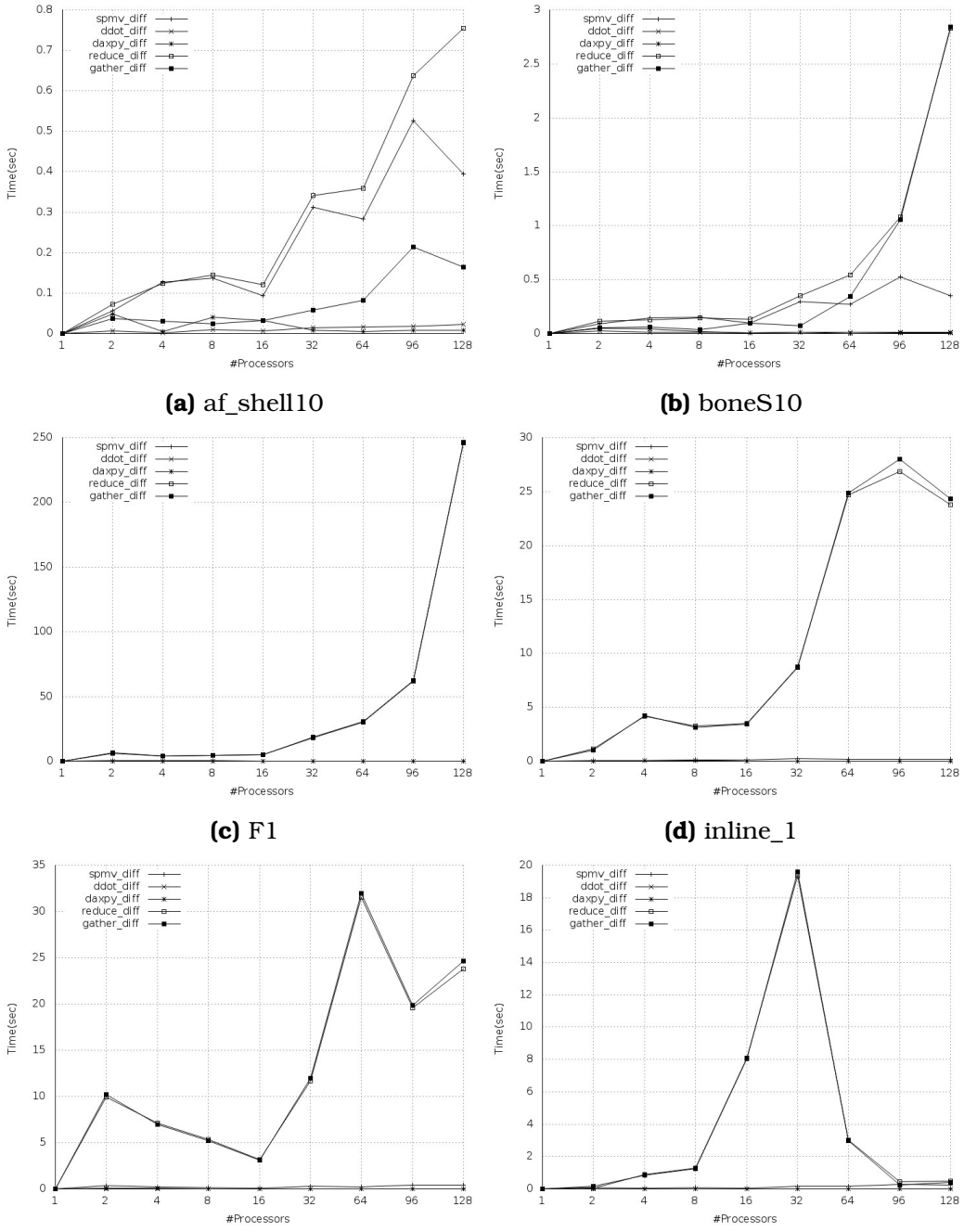


(a) af_shell10 **(b)** boneS10

(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

Σχήμα 2.13: Σύγκριση μέσων χρόνων των υλοποιήσεων Scatter και P2P.

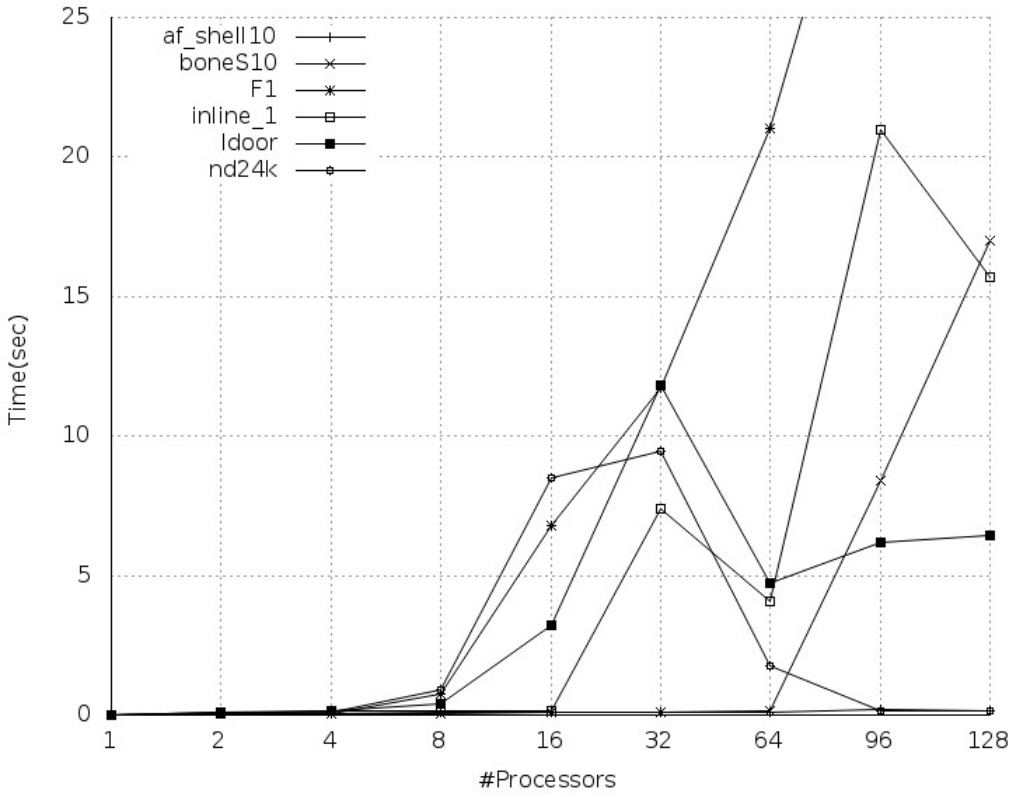


(a) af_shell10 **(b)** boneS10

(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

Σχήμα 2.14: Διαφορά μεγίστου - ελαχίστου για κάθε χρόνο για κάθε πίνακα στην υλοποίηση CG P2P.

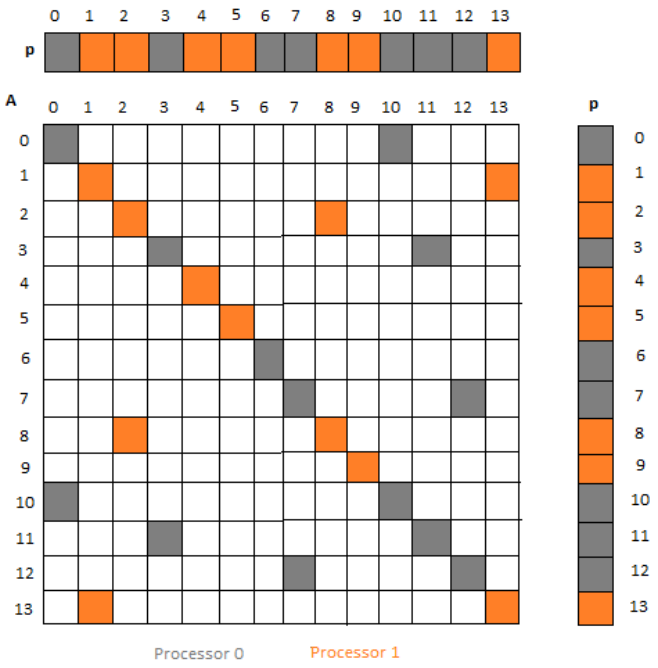


Σχήμα 2.15: Το ελάχιστο reduce για όλους του πίνακες στην υλοποίηση CG p2p.

2.7 CG metis partitioning

2.7.1 Υλοποίηση

Παρ' όλο που η υλοποίηση P2P βελτιώνει κάπως του χρόνους εκτέλεσης σε σύγκριση με την Scatter, έχει και αυτήν κάποια μειονεκτήματα. Το κυριότερο από αυτά είναι ότι ¨σπάει¨ τον αρχικό πίνακα σε μπλοκ συνεχόμενων γραμμών. Με αυτόν τον τρόπο έχουμε κάποια μηνύματα ανάμεσα σε επεξεργαστές τα οποία θα μπορούσαμε να αποφύγουμε αν ο πίνακας διαμεριζόταν σε μπλοκ γραμμών που δεν είναι κατ' ανάγκη συνεχόμενες. Για να γίνει πιο κατανοητό, ας επεκτείνουμε το παράδειγμα του πίνακα $14x14$ που χρησιμοποιήσαμε και στην προηγούμενη παράγραφο. Στο σχήμα 2.16 δίνεται ένα σχήμα διαμοιρασμού του πίνακα σε μπλοκ μη συνεχόμενων γραμμών. Παρατηρούμε ότι σε αυτήν την περίπτωση ο κάθε επεξεργαστής χρησιμοποιεί για την πράξη spmv μόνο στοιχεία του διανύσματος p που του ανήκουν οπότε έχουμε μηδενική επικοινωνία ανάμεσα στους επεξεργαστές. Στην περίπτωση του διαμοιρασμού ανά γραμμές χοντρού κόκκου ο συνολικός όγκος επικοινωνίας ήταν 8 στοιχεία του p (4 από τον processor 0 προς τον processor 1 και 4 προς την αντίθετη κατεύθυνση).

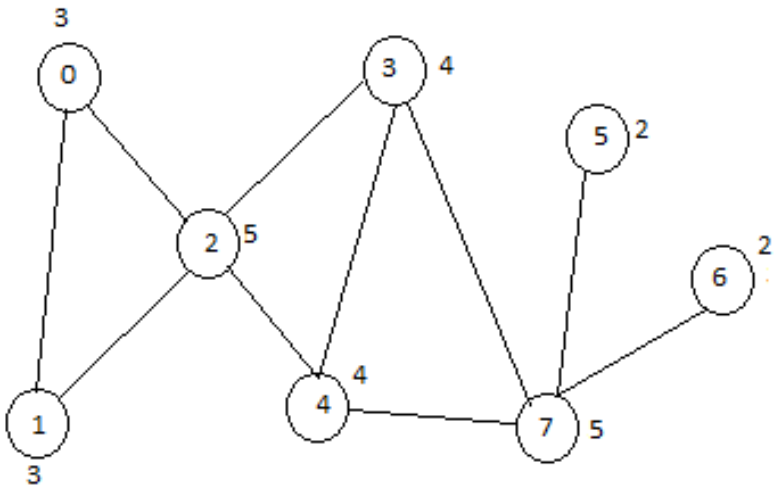


Σχήμα 2.16: Διαμοιρασμός του πίνακα $14x14$ του προηγούμενου παραδείγματος σε μπλοκ μη συνεχόμενων γραμμών.

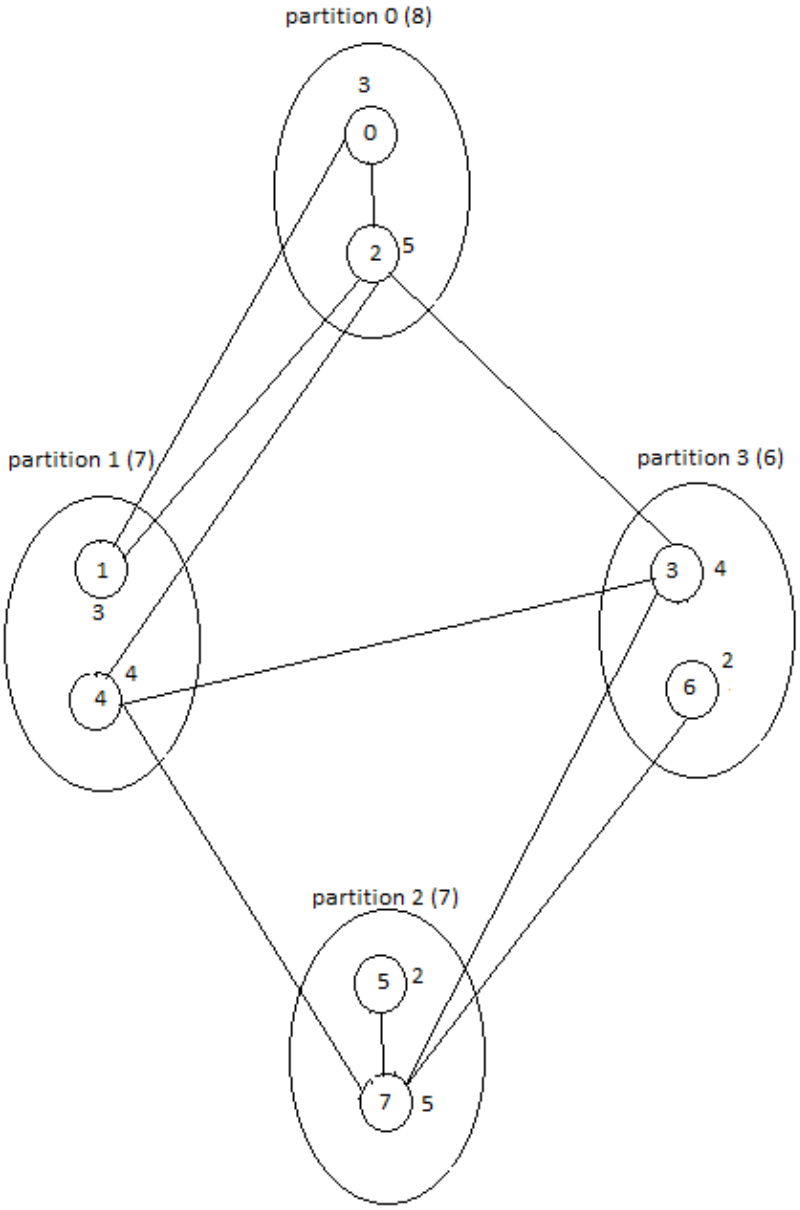
Ο διαμερισμός του πίνακα με τον τρόπο που αναφέρθηκε προηγουμένως δεν είναι εύκολη διαδικασία καθώς υπάρχουν αρκετοί παράμετροι που πρέπει να ληφθούν

υπόψη. Ένας από αυτούς είναι να διατηρηθεί ο ισομερής καταμερισμός των μη μηδενικών στοιχείων του πίνακα στο σύνολο των επεξεργαστών. Για το σκοπό αυτό θα χρησιμοποιήσουμε την βιβλιοθήκη METIS η οποία παρέχει ρουτίνες για διαμερισμό γράφων.

Η βιβλιοθήκη METIS παρέχει κάποιες ρουτίνες που χρησιμοποιούνται για το διαμερισμό γράφων, με κριτήριο τα βάρη των κόμβων να μοιραστούν ομοιόμορφα, κρατώντας παράλληλα μικρό τον αριθμό των ακμών που ενώνουν μία διαμέριση με κάποια άλλη. Ένα παράδειγμα ενός γράφου δίνεται στο σχήμα 2.17 και ο αντίστοιχος διαμερισμός του σε τέσσερα κομμάτια, με χρήση ρουτίνας από τη βιβλιοθήκη METIS στο σχήμα 2.18.



Σχήμα 2.17: Παράδειγμα γράφου με 8 κόμβους και βάρη στους κόμβους. Ο αριθμός μέσα σε κάθε κόμβο είναι το αναγνωριστικό του και ο αριθμός από πάνω το βάρος του.



Σχήμα 2.18: Διαμερισμός του γράφου του σχήματος 2.17 σε τέσσερα κομμάτια. Ο αριθμός σε παρένθεση πάνω από κάθε διαμέριση είναι το συνολικό βάρος της διαμέρισης.

Προκειμένου να χρησιμοποιήσουμε τις προαναφερθείσες ρουτίνες για το διαμερισμό του πίνακα A θα θεωρήσουμε ένα γράφο με τα εξής χαρακτηριστικά:

→ Ο αριθμός των κόμβων είναι ίσος με τον αριθμό των γραμμών του πίνακα (δηλαδή ίσος με τον αριθμό των στηλών αφού αναφερόμαστε σε τετραγωνικούς

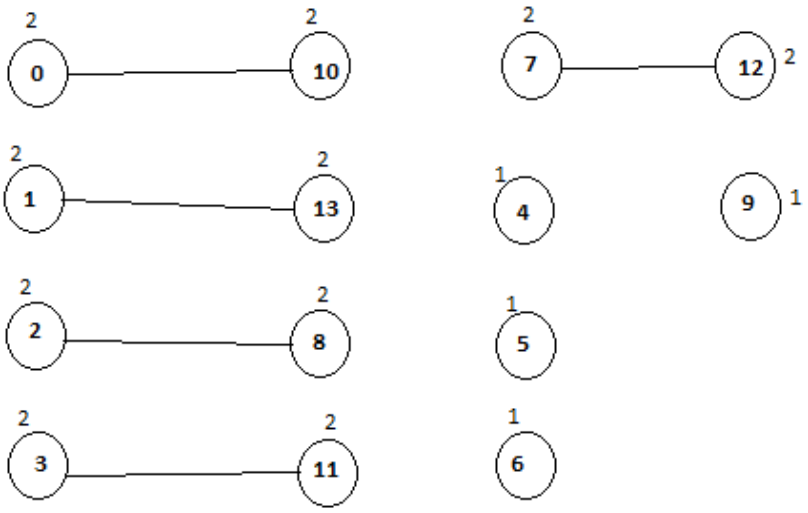
πίνακες).

→ Το βάρος κάθε κόμβου είναι ίσο με τον αριθμό των μη μηδενικών στοιχείων που περιέχει η αντίστοιχη γραμμή του πίνακα.

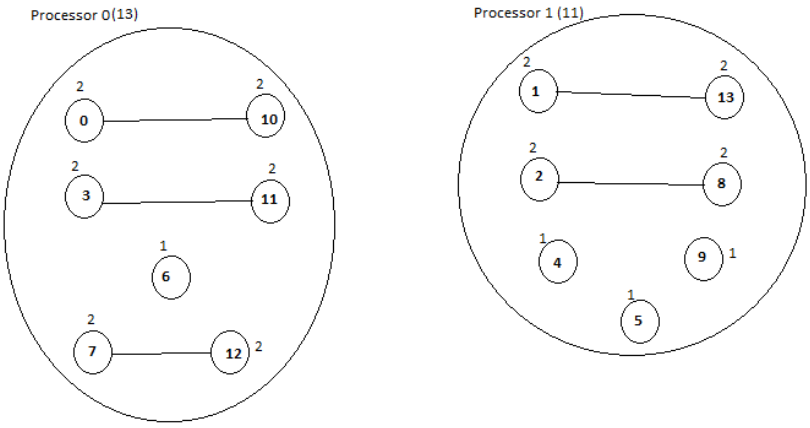
→ Κάθε ακμή από έναν κόμβο i σε κάποιον άλλο κόμβο j αναπαριστά την ύπαρξη μη μηδενικού στοιχείου στις θέσεις (i,j) και (j,i) του πίνακα (υπενθυμίζουμε ότι ο αλγόριθμος CG χρησιμοποιείται για συμμετρικούς πίνακες).

→ Κάθε ακμή (i,j) πρακτικά συμβολίζει την ανάγκη ανταλλαγής δύο στοιχείων του διανύσματος p ανάμεσα στον επεξεργαστή που έχει τη γραμμή i (θα πρέπει να παραλάβει το στοιχείο j του p) και τον επεξεργαστή που έχει τη γραμμή j του πίνακα (θα πρέπει να παραλάβει το στοιχείο i του p). Έτσι το βάρος κάθε ακμής θα είναι ίσο με δύο.

Με βάση τα παραπάνω το πρόβλημα του διαμερισμού του συμμετρικού τετραγωνικού πίνακα A ισοδυναμεί με τον διαμερισμό του αντίστοιχου γράφου. Ο ισοδύναμος γράφος του πίνακα του σχήματος 2.16 παρουσιάζεται στο σχήμα 2.19 ενώ στο σχήμα 2.20 φαίνεται ο διαμερισμός του γράφου σε δύο κομμάτια. Παρατηρούμε ότι έχουμε την ελάχιστη δυνατή επικοινωνία (δεν ανταλλάσσεται κανένα μήνυμα ανάμεσα στους δύο επεξεργαστές), ενώ ταυτόχρονα τα μη μηδενικά στοιχεία έχουν μοιραστεί σχεδόν ισόποσα.



Σχήμα 2.19: Ο ισοδύναμος γράφος του πίνακα του σχήματος 2.16.



Σχήμα 2.20: Ο διαμερισμός του γράφου του πίνακα του σχήματος 2.16 σε δύο κομμάτια.

Στον πίνακα 2.5 παρουσιάζεται ο όγκος επικοινωνίας που απαιτείται σε κάθε επανάληψη του κυρίως βρόχου του αλγορίθμου CG, όταν ο πίνακας A διαμερίζεται με χρήση της βιβλιοθήκης METIS, καθώς επίσης και η επί τοις εκατό μείωση του σε σχέση με τον αντίστοιχο όγκο επικοινωνίας της υλοποίησης P2P. Βλέπουμε ότι έχουμε σε κάθε περίπτωση πολύ μεγάλη μείωση άρα περιμένουμε να έχουμε πολύ καλά πειραματικά αποτελέσματα από την συγκεκριμένη υλοποίηση.

Όνομα	Αριθμός επεξεργαστών	Όγκος επικοινωνίας Metis (MB)	Μείωση σε σχέση με το P2P (%)
af_shell10	16	0.28	53.33
	32	0.43	65.32
	64	0.64	74.7
	96	0.81	78.74
	128	0.94	81.57
boneS10	16	0.26	83.75
	32	0.4	87.92
	64	0.64	90.49
	96	0.84	91.73
	128	1.03	91.88
F1	16	0.41	97.44
	32	0.7	96.44
	64	1.14	94.98
	96	1.49	93.81
	128	1.67	93.31
inline_1	16	0.27	97.26
	32	0.46	96.48
	64	0.75	95.22
	96	1	94.19
	128	1.2	93.42
ldoor	16	0.17	98.36
	32	0.31	97.59
	64	0.51	96.74
	96	0.65	96.16
	128	0.78	95.6
nd24k	16	0.71	81.02
	32	1.11	78.82
	64	1.63	73.88
	96	2.07	68.97
	128	2.49	64.43

Πίνακας 2.5: Ο όγκος επικοινωνίας για κάθε εκτέλεση του κυρίως βρόχου του προγράμματος, για κάθε πίνακα και κάθε διαφορετικό αριθμό επεξεργαστών, στην υλοποίηση Metis.

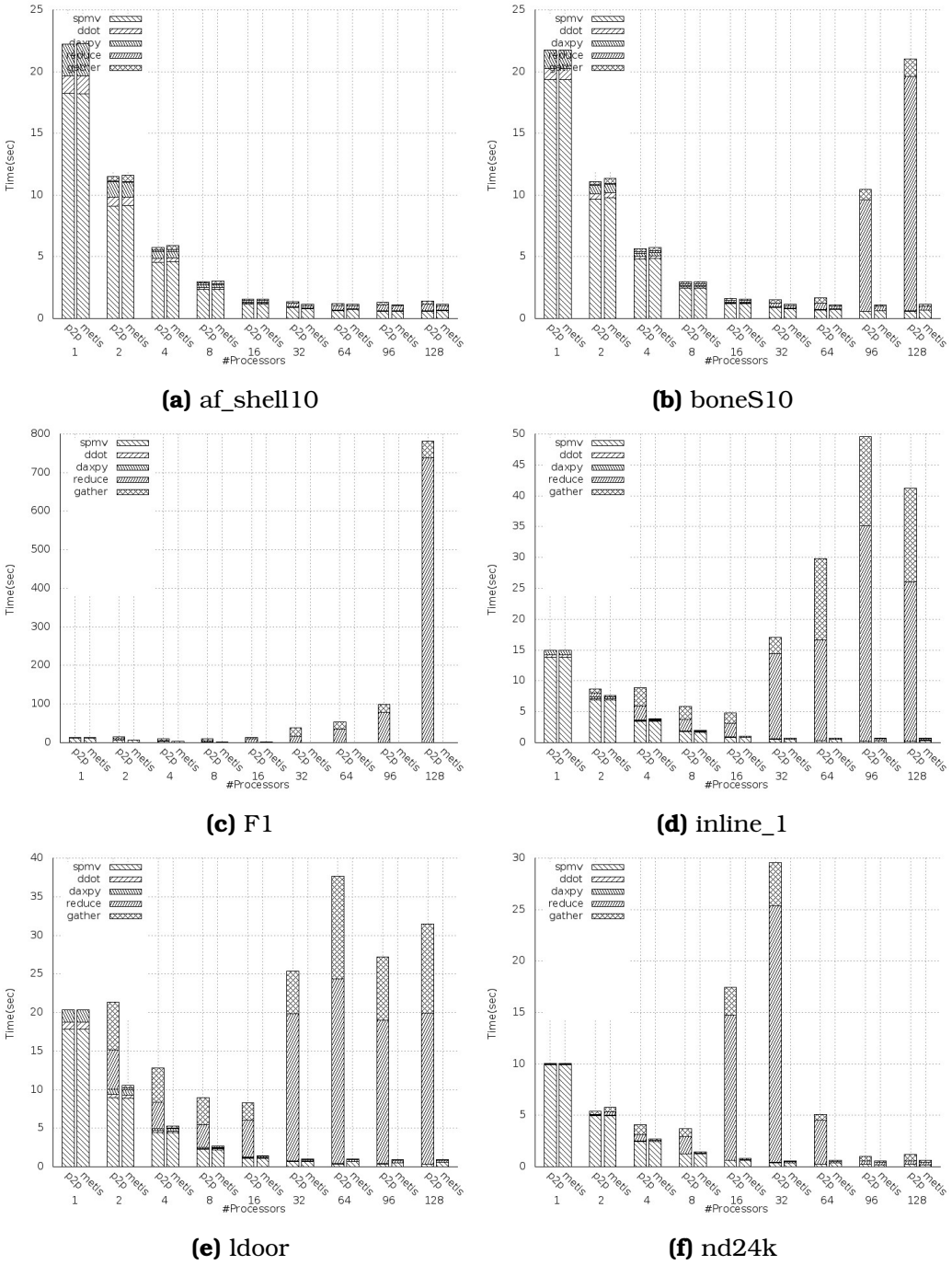
2.7.2 Πειραματική αξιολόγηση

Στο σχήμα 2.21 συγκρίνουμε τους μέσους χρόνους της υλοποίησης Metis με τους αντίστοιχους της P2P. Ο πίνακας af_shell10 δεν παρουσιάζει μεγάλη βελτίωση καθώς ήδη οι χρόνοι του ήταν πολύ χαμηλοί στην υλοποίηση P2P και δεν παρουσίαζε κάποιο πρόβλημα imbalance. Αντίθετα όλοι οι υπόλοιποι πίνακες που αντιμετώπιζαν πρόβλημα imbalance, παρουσιάζουν σημαντική βελτίωση στο κομμάτι του reduce και του gather.

Η υλοποίηση Metis όπως δείχνουν και τα πειραματικά αποτελέσματα ελαχιστοποιεί τον απαιτούμενο όγκο επικοινωνίας ανάμεσα στους επεξεργαστές, και επιπλέον αναθέτει περίπου ίσο φόρτο επικοινωνίας στον καθένα. Με αυτό τον τρόπο εξαλείφουμε το communication imbalance ανάμεσα στους επεξεργαστές, όπως επαληθεύεται και από το σχήμα 2.22. Σε αυτό παρουσιάζεται για κάθε πίνακα η διαφορά μεγίστου - ελαχίστου για κάθε χρόνο, δηλαδή το imbalance. Βλέπουμε ότι το gather imbalance πλέον είναι πολύ χαμηλό και το κυρίαρχο είναι πλέον το spmv imbalance, το οποίο όμως είναι επίσης αρκετά χαμηλό.

Όμως με την μείωση του όγκου επικοινωνίας το Metis επιτυγχάνει και την εξάλειψη του προβλήματος που αντιμετωπίσαμε προηγουμένως με την συμφόρηση του διαύλου της κοινής μνήμης στους κόμβους. Το ελάχιστο reduce πλέον έχει τη συμπεριφορά που περιμέναμε, δηλαδή εξαρτάται μόνο από τον αριθμό των επεξεργαστών και είναι πολύ χαμηλό. Το ελάχιστο reduce για όλους του πίνακες παρουσιάζεται στο σχήμα 2.23.

Σε αυτό το σημείο αξίζει να επισημάνουμε ότι στην υλοποίηση Metis καταφέραμε να περιορίσουμε την αύξηση του χρόνου που σπαταλάται σε επικοινωνία καθώς αυξάνονται οι επεξεργαστές και πλέον ο κυρίαρχος χρόνος έχει γίνει ο spmv (βλ. σχήμα 2.24). Η συμπεριφορά του spmv πλέον παρουσιάζει το μεγαλύτερο ενδιαφέρον. Στο σχήμα 2.24 βλέπουμε ότι οι μεγαλύτερες μειώσεις στο χρόνο του spmv σημειώνονται στις μεταβάσεις από 16 σε 32 και από 64 σε 96 επεξεργαστές. Και στις δύο αυτές μεταβάσεις διπλασιάζεται η συνολική L2 cache. Αυτό έρχεται σε πλήρη συμφωνία με το γεγονός ότι η λειτουργία spmv παρουσιάζει έντονες προσβάσεις στη μνήμη.

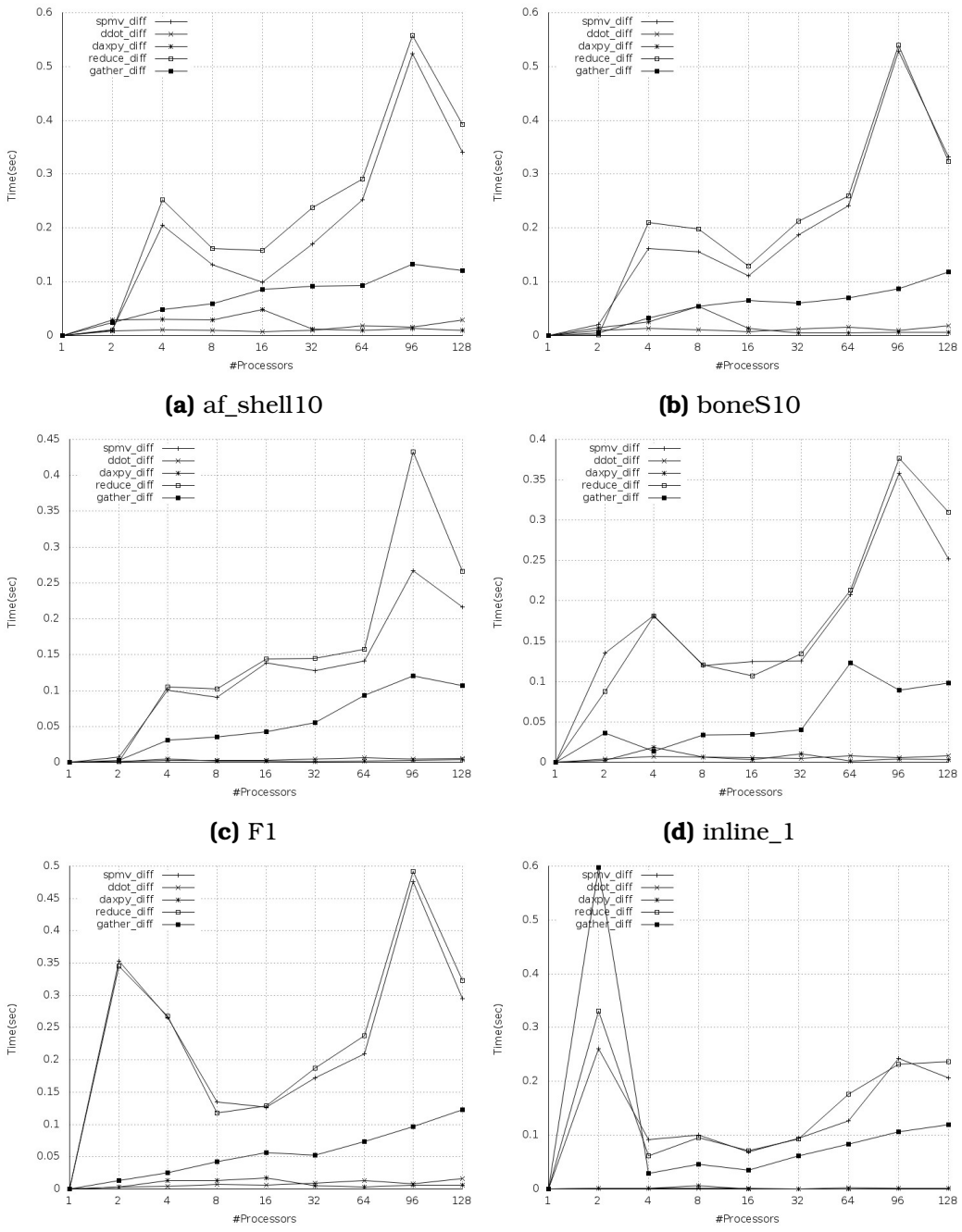


(a) af_shell10 **(b)** boneS10

(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

Σχήμα 2.21: Σύγκριση μέσων χρόνων των υλοποιήσεων P2P και Metis.

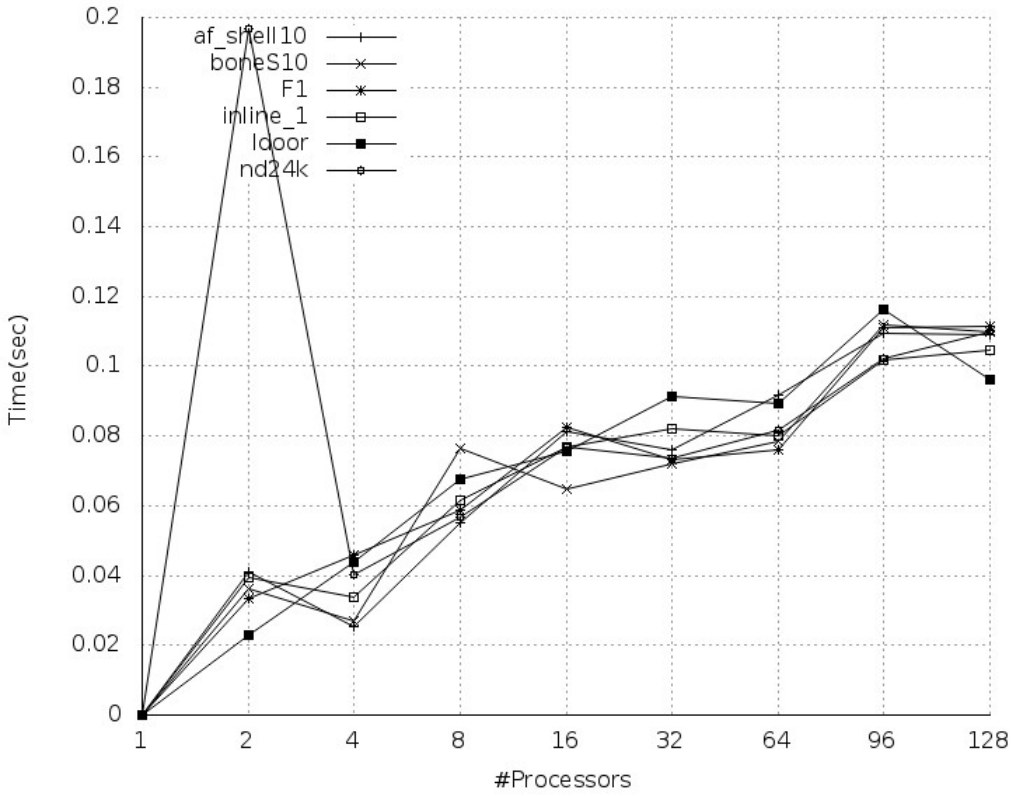


(a) af_shell10 **(b)** boneS10

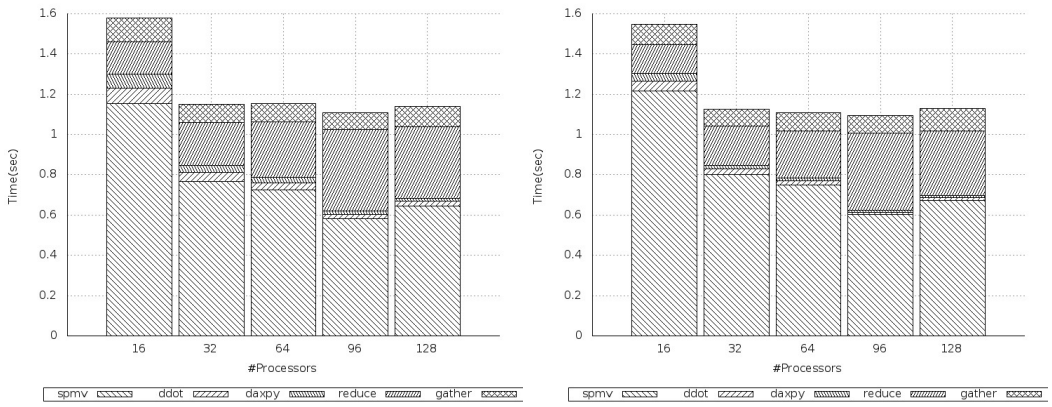
(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

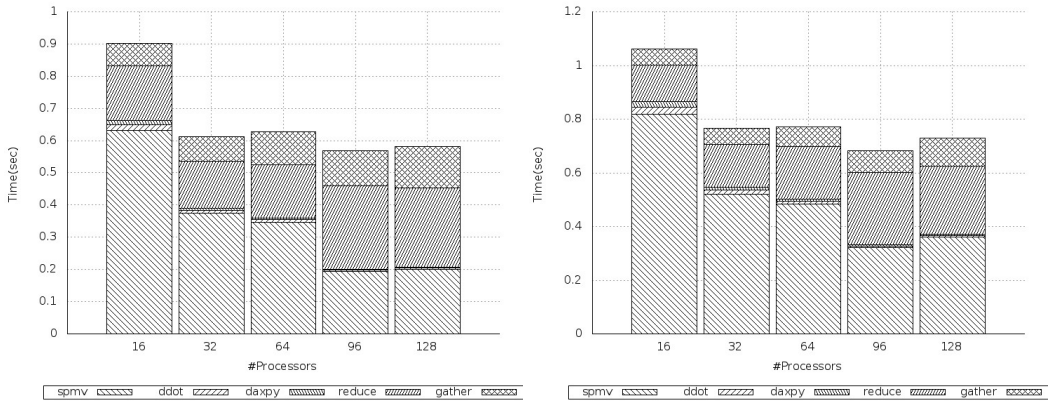
Σχήμα 2.22: Διαφορά μεγίστου - ελαχίστου για κάθε χρόνο για κάθε πίνακα στην υλοποίηση CG Metis.



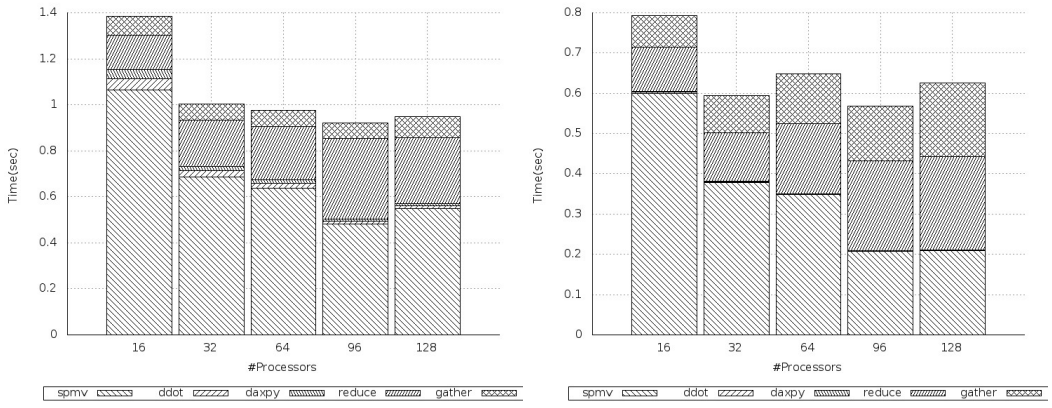
Σχήμα 2.23: Το ελάχιστο reduce για όλους του πίνακες στην υλοποίηση CG Metis.



(a) af_shell10 **(b)** boneS10



(c) F1 **(d)** inline_1



(e) ldoor **(f)** nd24k

Σχήμα 2.24: Μέσοι χρόνοι για κάθε πίνακα στην υλοποίηση CG Metis.

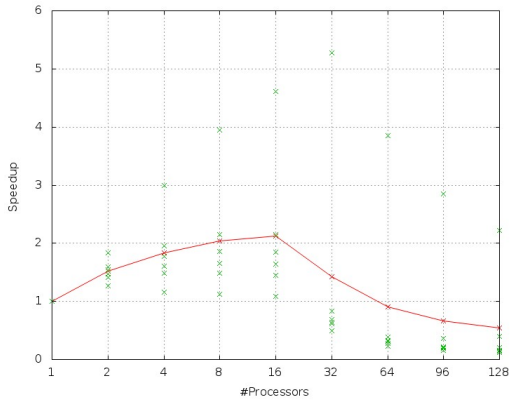
2.8 Συμπεράσματα

Οι πρώτες τρεις υλοποιήσεις που είδαμε ήταν οι Scatter, P2P, Metis. Στην πρώτη είδαμε ότι, παρόλο που κάθε διεργασία αναλάμβανε ίσο φόρτο υπολογισμών και επικοινωνίας, ο τεράστιος συνολικός όγκος επικοινωνίας που απαιτείτο προκαλούσε συμφόρηση στον δίαυλο κοινής μνήμης των κόμβων με αποτέλεσμα η αύξηση στο gather και στο reduce να υπερκαλύπτει όποια βελτίωση είχαμε από την παραλληλοποίηση του υπολογιστικού μέρους του αλγορίθμου.

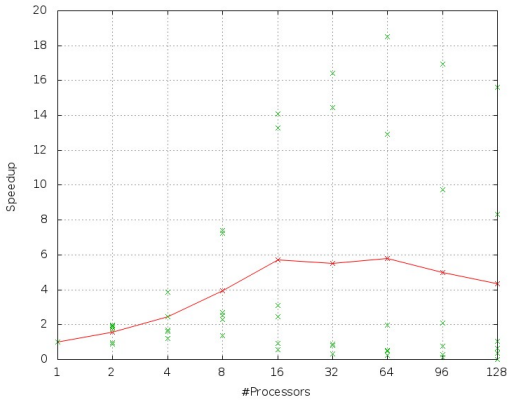
Στην υλοποίηση P2P καταφέραμε να μειώσουμε σε μεγάλο βαθμό τον απαιτούμενο όγκο επικοινωνίας, ωστόσο με αυτόν τον τρόπο δημιουργήθηκε ένα χάσμα ανάμεσα σε διεργασίες που έπρεπε να στείλουν μεγάλο όγκο δεδομένων και σε άλλες οι οποίες είχαν μόνο ένα μικρό αριθμό μηνυμάτων να στείλουν. Έτσι σε αρκετές περιπτώσεις δεν είχαμε την βελτίωση που θα περιμέναμε αναλογικά με το ποσοστό μείωσης του φόρτου επικοινωνίας. Επίσης, η μείωση αυτή σε κάποιους πίνακες δεν ήταν αρκετή ώστε να αποφύγουμε τη συμφόρηση των διαδρόμων μνήμης των κόμβων.

Τέλος, με την υλοποίηση Metis καταφέραμε να ξεπεράσουμε τα προβλήματα των προηγούμενων δύο. Από τη μία, ο ισομερής καταμερισμός του φόρτου επικοινωνίας ανάμεσα στις διεργασίες συντέλεσε στην εξάλειψη του gather imbalance και από την άλλη η περαιτέρω μείωση του συνολικού όγκου επικοινωνίας επέτρεψε την αποφυγή της συμφόρησης των διαδρόμων μνήμης των κόμβων.

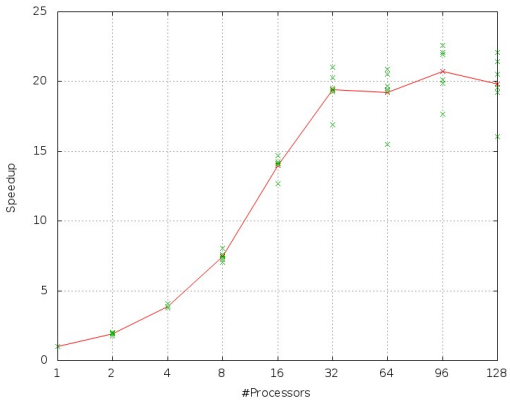
Στο σχήμα 2.25 παρουσιάζονται οι επιταχύνσεις κάθε υλοποίησης σε σχέση με τον σειριακό αλγόριθμο CG.



(a) Scatter



(b) P2P



(c) Metis

Σχήμα 2.25: Επιταχύνσεις παράλληλου αλγόριθμου CG για κάθε υλοποίηση.

Κεφάλαιο 3

Προχωρημένες τεχνικές βελτιστοποίησης παράλληλου αλγορίθμου CG

Στις υλοποιήσεις του προηγούμενου κεφαλαίου ο στόχος μας ήταν η μείωση κυρίως του κόστους επικοινωνίας καθώς αυτήν αποτελούσε το μεγαλύτερο μέρος του χρόνου εκτέλεσης, μετά από κάποιον αριθμό επεξεργαστών. Καταφέραμε να πετύχουμε σημαντική βελτίωση σε αυτό το κομμάτι και πλέον ο κυρίαρχος όρος, ακόμα και για 128 επεξεργαστές, είναι ο χρόνος που σπαταλάται σε υπολογισμούς και συγκεκριμένα στον πολλαπλασιασμό του αραιού πίνακα A με το διάνυσμα p (SpMxV). Σε αυτό το κεφάλαιο λοιπόν, στο οποίο παρουσιάζονται δύο υλοποιήσεις υβριδικού προγραμματισμού, θα επικεντρωθούμε στην μείωση του χρόνου spmv.

3.1 CG hybrid CSR

Στην υλοποίηση αυτή θα χρησιμοποιήσουμε το μοντέλο παράλληλου προγραμματισμού κοινής μνήμης για το εσωτερικό των κόμβων. Πιο συγκεκριμένα, θα χρησιμοποιήσουμε την βιβλιοθήκη OpenMP. Σε αυτή την υλοποίηση ο αρχικός πίνακας και τα διανύσματα διαμοιράζονται στους δεκαέξι κόμβους, όπου στη συνέχεια ένας αριθμός από νήματα (threads) ενεργεί πάνω στα κοινά αυτά δεδομένα και μας δίνει το επιθυμητο αποτέλεσμα.

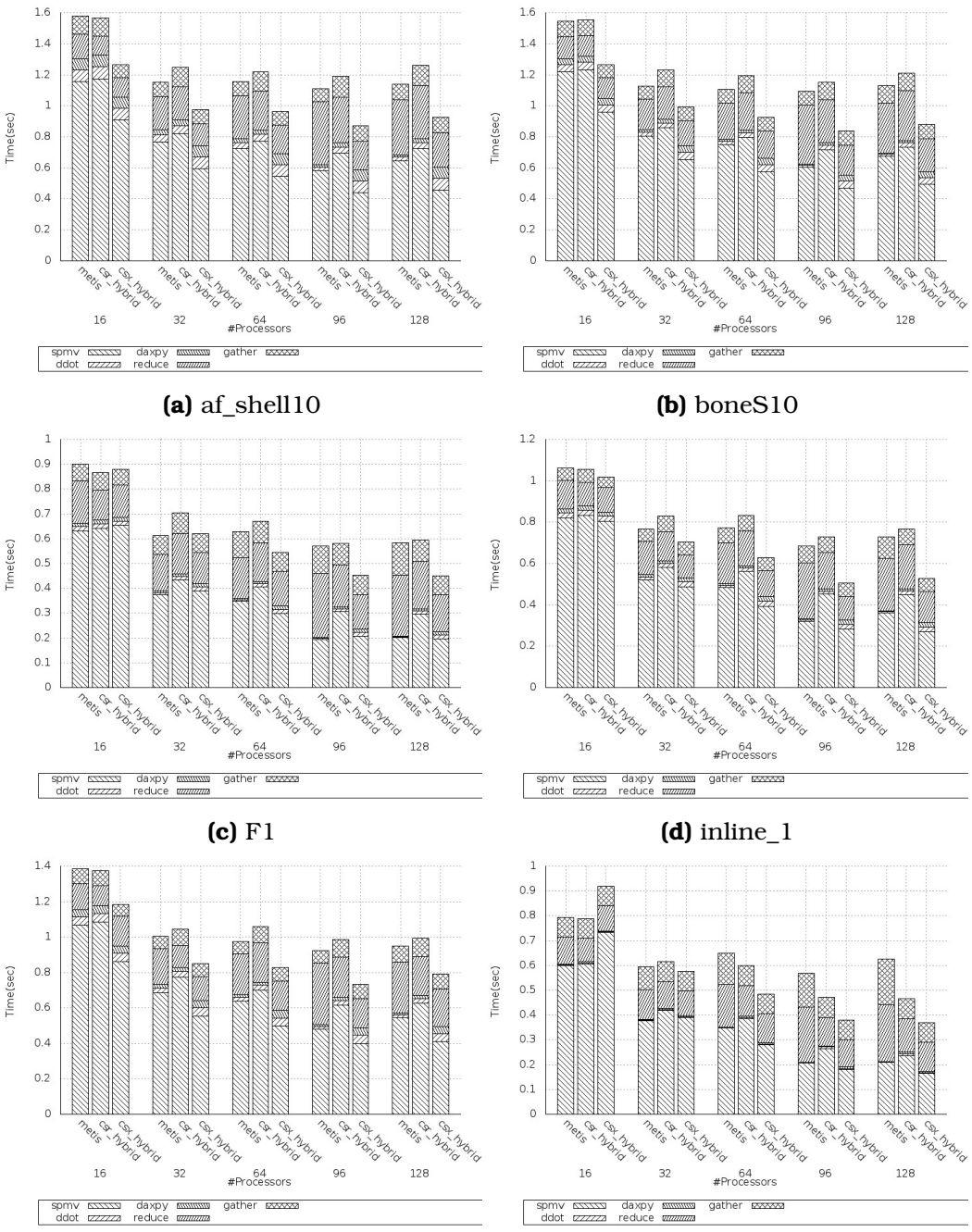
Με αυτόν τον τρόπο περιμένουμε ότι το κόστος επικοινωνίας, για περισσότερους από 16 επεξεργαστές, θα παραμείνει σταθερό, αφού δεν προσθέτουμε MPI διεργασίες, ενώ το κόστος των υπολογισμών θα συνεχίσει να μειώνεται. Ωστόσο, όπως έχουμε αναφέρει και σε προηγούμενα κεφάλαια ο πολλαπλασιασμός αραιού πίνακα με διάνυσμα παρουσιάζει πολύ έντονες προσβάσεις στην κοινή μνήμη με αποτέλεσμα τη μείωση της κλιμακωσιμότητας του καθώς αυξάνουμε τον αριθμό των επεξεργαστών. Έτσι δεν περιμένουμε σημαντικές βελτιώσεις, τουλάχιστον στο κομμάτι του spmv.

3.2 CG hybrid CSX

Όπως έχουμε αναφέρει και σε προηγούμενο κεφάλαιο η αναπαράσταση του αραιού πίνακα με το σχήμα CSR έχει το μειονέκτημα ότι δεν εκμεταλλεύεται πιθανές κανονικότητες (δηλαδή μπλοκ συνεχόμενων μη μηδενικών στοιχείων σε μία ή δύο διαστάσεις) που εμφανίζονται στον πίνακα ώστε να βελτιώσει την τοπικότητα των προσβάσεων που γίνονται στην κύρια μνήμη. Σε αυτήν την υλοποίηση θα χρησιμοποιήσουμε το σχήμα CSX για την αναπαράσταση του πίνακα με στόχο τη μείωση των απαιτούμενων προσβάσεων στην κοινή μνήμη.

3.3 Πειραματική αξιολόγηση υβριδικών υλοποιήσεων

Στο σχήμα 3.1 παρουσιάζεται η κατανομή των χρόνων εκτέλεσης των δύο υβριδικών υλοποιήσεων σε σύγκριση με την υλοποίηση Metis. Οι χρόνοι επικοινωνίας δε μεταβάλλονται καθώς αυξάνουμε τον αριθμό των επεξεργαστών στις δύο υβριδικές υλοποιήσεις. Όσον αφορά στο κομμάτι του spmv στη μείωση του οποίου στοχεύαμε, βλέπουμε ότι η υλοποίηση με το σχημα αναπαράστασης CSX παρουσιάζει εμφανώς καλύτερα αποτελέσματα από το σχήμα CSR το οποίοι ακόμα και σε σύγκριση με την υλοποίηση Metis εμφανίζει ελάχιστη ή και καθόλου βελτίωση. Σε κάποιες περιπτώσεις μάλιστα έχουμε ακόμα και αυξηση του χρόνου spmv.



(a) af_shell10 **(b)** boneS10

(c) F1 **(d)** inline_1

(e) ldoor **(f)** nd24k

Σχήμα 3.1: Σύγκριση μέσων χρόνων των υλοποιήσεων Metis, Hybrid CSR και Hybrid CSX.

Κεφάλαιο 4

Συμπεράσματα και μελλοντικές κατευθύνσεις

Στην παρούσα διπλωματική προσπαθήσαμε να αναλύσουμε τη συμπεριφορά του παράλληλου αλγορίθμου συζυγών κλίσεων και να τη βελτιώσουμε ώστε να κλιμακώνει ικανοποιητικά για μεγάλο αριθμό επεξεργαστών.

Αρχικά, παρουσιάστηκαν τρεις υλοποιήσεις. Η πιο απλή από τις τρεις, η Scatter, είδαμε ότι εμπεριείχε τεράστιο όγκο επικοινωνίας με αποτέλεσμα οι χρόνοι επικοινωνίας να υπερκαλύπτουν τη βελτίωση που επέφερε η παραλληλοποίηση στο κομμάτι των υπολογισμών. Επίσης, είχαμε πρόβλημα συμφόρησης του διαδρόμου της κοινής μνήμης των κόμβων λόγω του μεγάλου όγκου επικοινωνίας.

Η δεύτερη υλοποίηση, η P2P, μείωσε σε πολύ μεγάλο ποσοστό τον όγκο της απαιτούμενης επικοινωνίας ανάμεσα στους επεξεργαστές αλλά δημιουργήθηκε πολύ μεγάλο communication imbalance ανάμεσα στις διεργασίες. Επίσης, το πρόβλημα της συμφόρησης των διαδρόμων της μνήμης των κόμβων παρέμεινε και σε αυτήν την υλοποίηση.

Η τρίτη υλοποίηση, Metis, μείωσε ακόμα περισσότερο τον όγκο της απαιτούμενης επικοινωνίας και παρουσίασε σημαντική βελτίωση στους χρόνους επικοινωνίας. Το σημαντικότερο είναι ότι η υλοποίηση αυτή δεν παρουσίασε πρόβλημα imbalance, ούτε πρόβλημα συμφόρησης των διαδρόμων της κοινής μνήμης.

Στη συνέχεια, παρουσιάσαμε δύο υλοποιήσεις υβριδικού προγραμματισμού με στόχο τη βελτίωση των χρόνων που αφορούν το υπολογιστικό κομμάτι του αλγορίθμου. Στην πρώτη από αυτές απλά χρησιμοποιήσαμε τη βιβλιοθήκη OpenMP για τη δημιουργία των νημάτων που χρησιμοποιούν την κοινή μνήμη σε κάθε κόμβο. Δεν είδαμε κάποια βελτίωση καθώς η λειτουργία spmv δεν κλιμακώνει σε συστήματα κοινής μνήμης.

Στην τελευταία υλοποίηση χρησιμοποιήσαμε για την αναπαράσταση του αραιού πίνακα το σχήμα csx, για το οποίο έχει αποδειχθεί ότι η λειτουργία spmv κλιμακώνει αρκετά καλά [11]. Τα αποτελέσματα της υλοποίησης αυτής ήταν αρκετά ικανοποιητικά αφού πετύχαμε σημαντικές μειώσεις στους χρόνους εκτέλεσης της λειτουργίας

spmv.

Στη συνέχεια παρουσιάζουμε πιθανές μελλοντικές ερευνητικές κατευθύνσης:

- **Συμπίεση δεδομένων που ανταλλάσσονται μεταξύ των επεξεργαστών.** Είδαμε ότι ο παράλληλος αλγόριθμος CG απαιτεί την ανταλλαγή μεγάλου όγκου πληροφορίας ανάμεσα στους επεξεργαστές. Με χρήση αλγορίθμων συμπίεσης δεδομένων όπως ο LZO θα μπορούσαμε να συμπιέσουμε τα δεδομένα πριν την αποστολή και να τα αποσυμπιέζουμε κατά τη λήψη. Έτσι θα είχαμε κάποια βελτίωση στους χρόνους επικοινωνίας, όμως θα έπρεπε να υπερκεράσουμε το κόστος συμπίεσης/αποσυμπίεσης.

- **Επικάλυψη κόστους υπολογισμών και επικοινωνίας.** Κατά την αποστολή και λήψη των μηνυμάτων ανάμεσα στους επεξεργαστές οι τελευταίοι παραμένουν ανενεργοί περιμένοντας την ολοκλήρωση της επικοινωνίας. Θα μπορούσαμε να εκμεταλλευτούμε αυτό το χρόνο ώστε να εκτελέσουμε λειτουργίες που χρησιμοποιούν μόνο τοπικά δεδομένα όσο αναμένουμε να φτάσουν δεδομένα από άλλους επεξεργαστές. Επίσης μία υβριδική λύση επικάλυψης θα μπορούσε να χρησιμοποιεί ένα νήμα εκτέλεσης για το κομμάτι των υπολογισμών όσο ένα παράλληλο νήμα έχει αναλάβει την αποστολή και λήψη των δεδομένων από και προς άλλους επεξεργαστές.

Βιβλιογραφία

[1] G. Moore, "Cramming more components onto integrated circuits", Electronics Magazine 19 April 1965

[2] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 1972.

[3] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, USA, 2003.

[4] J.M. Kleinberg, S.R. Kumar, P. Raghavan, S. Rajagopalan, and A.S. Tomkins, The web as a graph: Measurements, models and methods. Lecture Notes in Computer Science, pages 1-17, 1999.

[5] Bruce Hendrickson and Tamara G. Kolda, Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. SIAM J. Sci. Comput., 21(6):2048-2072, 1999

[6] Brendan Vastenhouw and Rob H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev., 47(1):67-95, 2005

[7] K. Asanovic and et al., The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS2006183, EECS Department, University of California, Berkeley, December 2006.

[8] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris, "Performance Evaluation of the Sparse Matrix-vector Multiplication on Modern Architectures," The Journal of Supercomputing, Vol 50, No 1, 2009

[9] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, Optimization of sparse matrixvector multiplication on emerging multicore platforms. In Supercomputing'A 07, Reno, NV, November 2007

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia, 1994

[11] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, Nektarios Koziris, CSX: An Extended Compression Format fo SpMV on Shared Memory Systems, 2010

[12] George Karypis and Vipin Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs ,International Conference on Parallel Processing, pp. 113-122, 1995

[13] G. Karypis and V. Kumar, METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995

[14] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In SC 'A07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, NV, November 2007. 21, 59, 89, 104