ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επικάλυψη Υπολογισμών και Επικοινωνίας σε Συστοιχίες από Πολυνηματικούς Επεξεργαστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλας Ι. Ιωάννου

Επιβλέπων : Νεκτάριος Κοζύρης
Αν. Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2008

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επικάλυψη Υπολογισμών και Επικοινωνίας σε Συστοιχίες από Πολυνηματικούς Επεξεργαστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλας Ι. Ιωάννου

Επιβλέπων : Νεκτάριος Κοζύρης
Αν. Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10η Ιουλίου 2008

……..	……..	……..
Ν. Κοζύρης	Κ. Σαγώνας	Ν. Παπασπύρου
Αν. Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2008

……..

Νικόλας Ι. Ιωάννου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ένα από τα πιο σημαντικά προβλήματα που παρουσιάζεται στα παράλληλα συστήματα επεξεργασίας αποτελεί η εύρεση νέων μεθόδων για την καλύτερη αξιοποίηση των διαθέσιμων υπολογιστικών πόρων. Σκοπός αυτής της διπλωματικής εργασίας μας είναι η καλύτερη αξιοποίηση παράλληλων αρχιτεκτονικών Simultaneous Multithreading (SMT) για την εκτέλεση δύσκολων υπολογισμών με εφαρμογή επικαλυπτόμενης δρομολόγησης και χρήση μετασχηματισμού υπερκόμβων (tiling). Στην προσέγγιση μας, επιλέγουμε ένα χρονικό μετασχηματισμό που εκμεταλλεύεται την εγγενή επικάλυψη μεταξύ των φάσεων επικοινωνίας και εκτέλεσης υπολογισμών μεταξύ διαδοχικών, ατομικών στιγμιότυπων του βρόχου (tile). Με αυτό τον τρόπο επιτυγχάνεται η πιο αποδοτική εκτέλεση των στιγμιότυπων του βρόχου, με την προϋπόθεση ότι κάποιο μέρος της κάθε φάσης επικοινωνίας μπορεί να επικαλυφθεί αποδοτικά με ατομικούς υπολογισμούς σε κάθε στιγμιότυπο. Εφαρμόζουμε την παραπάνω δρομολόγηση για την παράλληλη εκτέλεση στιγμιότυπων βρόχου υπολογισμού διακριτών Μερικών Διαφορικών Εξισώσεων. Προσπαθούμε να εκμεταλλευτούμε την παραλληλία σε επίπεδο νημάτων (TLP) του αλγόριθμου δρομολόγησης τοποθετώντας τους δύο ετερογενής φόρτους εργασίας επικοινωνίας και επεξεργασίας σε διαφορετικούς λογικούς επεξεργαστές που ανήκουν στον ίδιο φυσικό πολυνηματικό επεξεργαστή. Για το σκοπό αυτό υλοποιήσαμε διάφορες εκδόσεις παράλληλου προγράμματος (πολυνηματικές και μη), χρησιμοποιώντας μεταξύ άλλων το μετασχηματισμό υπερκόμβων (tiling) που αποτελεί τον πιο διαδεδομένο μετασχηματισμό κώδικα για την αναδιάταξη των επαναλήψεων σε φωλιασμένους βρόχους. Από τα πειραματικά αποτελέσματα που πήραμε σε μία συστοιχία από Xeons με τεχνολογία Hyper-Threading χρησιμοποιώντας διάφορες πρωτογενείς εντολές MPI_Send, φαίνεται ότι μειώνεται σημαντικά ο ολικός παράλληλος χρόνος εκτέλεσης όταν οι δύο ετερογενές φόρτοι εργασίας επικοινωνίας και υπολογισμού, που παρουσιάζουν υψηλή παραλληλία TLP, τοποθετηθούν στους δύο λογικούς επεξεργαστές ενός φυσικού πολυνηματικού επεξεργαστή.

Λέξεις κλειδιά: επικάλυψη επικοινωνίας, πολυνηματική αρχιτεκτονική SMT, TLP, εμφωλευμένοι βρόχοι, μετασχηματισμός υπερκόμβων, πρωτογενείς εντολές MPI.

Abstract

One of the most difficult problems in parallel computing is finding new methods to better utilize the available resources. The goal of this diploma thesis is to exploit parallel architectures of Simultaneous Multithreading for better performance in heavy computations with the implementation of an overlapping schedule and tiling transformation. In our schedule, we select a time transformation to execute tiles much more efficiently by exploiting the inherent overlapping between communication and computation phases among successive, atomic tile executions. This schedule reduces the overall execution time under the assumption that some part from every communication phase can be efficiently overlapped with atomic, pure tile computations. We implement this schedule for the parallel execution of nested loops that compute discrete Partial Differential Equations. We try to exploit the thread level parallelism (TLP) of the schedule algorithm by placing the two heterogeneous workloads of communication and computation onto different logical processors within one physical package. For this we have implemented several versions of a parallel program (some of which multithreaded), using amongst others the tiling transformation which is one of the most widely used code transformations to recompose nested loop iterations. Our experimental results in a cluster of Xeons with Hyper-Threading Technology by using *MPI_Send* primitives, show that there is a significant reduction in overall parallel execution time when the two heterogeneous workloads of computation and communication (which present high TLP) are placed for execution onto the two different logical processors of one physical processor with Hyper-Threading Technology.

Keywords: communication overlapping, multithreading architecture SMT, thread level parallelism TLP, nested loops, tiling transformation, MPI send-receive primitives.

Ευχαριστίες

Η διπλωματική εργασία αυτή πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη.

Καταρχήν θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Νεκτάριο Κοζύρη, για την εποπτεία του κατά την εκπόνηση της εργασίας μου, αλλά και για τη συμβολή του στη διαμόρφωσή μου ως μηχανικού από τις διδασκαλίες του.

Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του εργαστηρίου και ιδιαίτερα στον Μεταδιδακτορικό Ερευνητή Γιώργο Γκούμα για τη συνεχή καθοδήγηση και ενθάρρυνση που μου προσέφερε για την ολοκλήρωση της διπλωματικής μου εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω το οικογενειακό και φιλικό μου περιβάλλον και κυρίως τους γονείς μου, η υποστήριξη των οποίων με βοήθησε τόσο στην εκπόνηση της εργασίας όσο και στην ολοκλήρωση των προπτυχιακών μου σπουδών.

Πίνακας Περιεχομένων

Κεφάλαιο 1

Εισαγωγή

1.1 Γενικά

Για πάνω από δύο δεκαετίες, έχουμε παρατηρήσει μια εκρηκτική εξέλιξη στην επίδοση και στις δυνατότητες των υπολογιστικών συστημάτων. Κύριο ρόλο σε αυτή την εξαιρετικά πετυχημένη άνοδο έχει παίξει η πρόοδος στον τομέα του VLSI, η οποία επιτρέπει στους χρόνους ρολογιού να αυξάνονται και σε ολοένα και αυξανόμενο αριθμό από συστατικά να χωράνε σε ένα τσιπ. Κεντρικό άξονα αυτής της τεχνολογικής ανέλιξης αποτελεί η αρχιτεκτονική υπολογιστών, η οποία μεταφράζει την ωμή τεχνολογική δυνατότητα σε μεγαλύτερη επίδοση και σε επέκταση των δυνατοτήτων του υπολογιστικού συστήματος. Η αρχιτεκτονική υπολογιστών για να επιτύχει αυτούς τους στόχους έχει καταφύγει στην παραλληλία. Μεγαλύτερος όγκος υπολογιστικών πόρων σημαίνει ότι περισσότερες λειτουργίες μπορούν να εκτελούνται ταυτόχρονα, παράλληλα. Η παράλληλη αρχιτεκτονική υπολογιστών αφορά την οργάνωση αυτών των υπολογιστικών πόρων ώστε να λειτουργούν μαζί αρμονικά. Υπολογιστές όλων των τύπων έχουν χρησιμοποιήσει τον παραλληλισμό ολοένα και πιο αποτελεσματικά για να κερδίσουν σε επίδοση, και το επίπεδο στο οποίο εκμεταλλεύονται τον παραλληλισμό συνεχίζει να αυξάνεται. Ένα ακόμη σημαντικό στοιχείο αποτελεί η τεχνολογία αποθήκευσης. Τα δεδομένα τα οποία χειριζόμαστε με ακόμη μεγαλύτερο ρυθμό πρέπει να κρατούνται κάπου μέσα στο μηχάνημα. Συνεπώς, η ιστορία της παράλληλης επεξεργασίας είναι βαθιά περιπλεγμένη με την τοπικότητα των δεδομένων και την επικοινωνία. Ο αρχιτέκτονας υπολογιστών πρέπει να ξεδιαλύνει αυτές τις μεταβαλλόμενες συσχετίσεις για να σχεδιάσει τα διάφορα επίπεδα ενός υπολογιστικού συστήματος για να επιτύχει τη μεγιστοποίηση της επίδοσης και της ευκολίας προγραμματισμού μέσα στα όρια που επιβάλλονται από την τεχνολογία και το οικονομικό κόστος τη δεδομένη χρονική στιγμή.

Η παράλληλη επεξεργασία είναι μια συναρπαστική προοπτική για να κατανοήσει κανείς την αρχιτεκτονική υπολογιστών γιατί εφαρμόζεται σε όλα τα επίπεδα σχεδίασης, αλληλεπιδρά με ουσιαστικά όλες τις άλλες αρχές της αρχιτεκτονικής, και παρουσιάζει μοναδική εξάρτηση με την υποκείμενη τεχνολογία. Συγκεκριμένα, τα βασικά θέματα της τοπικότητας, του εύρους ζώνης, του latency, και του συγχρονισμού εγείρονται σε πολλά από τα στάδια σχεδίασης των παράλληλων υπολογιστικών συστημάτων. Οι επιπτώσεις και οι

συνέπειες από τη σχεδίαση πρέπει να αντιμετωπιστούν μέσα στο πλαίσιο πραγματικών φόρτων εργασίας.

Η παράλληλη αρχιτεκτονική υπολογιστών, όπως και οποιοδήποτε είδος σχεδίασης, περιλαμβάνει στοιχεία μορφοποίησης και λειτουργίας. Αυτά τα στοιχεία αντικατοπτρίζονται επιτυχημένα στον παρακάτω ορισμό [1]:

> Ένα παράλληλος υπολογιστής είναι μία "συλλογή από μονάδες υπολογισμού που επικοινωνούν και συνεργάζονται για να επιλύουν μεγάλα προβλήματα γρήγορα"

Ωστόσο, αυτός ο απλός ορισμός εγείρει μία πληθώρα ερωτημάτων. Για πόσο μεγάλη συλλογή από μονάδες υπολογισμού μιλάμε; Πόσο ισχυρές πρέπει να είναι οι επιμέρους μονάδες υπολογισμού, και μπορεί ο αριθμός τους να αυξηθεί με απλό τρόπο; Πώς αυτά τα στοιχεία επικοινωνούν και συνεργάζονται; Πώς τα δεδομένα μεταδίδονται μεταξύ επεξεργαστών, τι είδος αλληλοσύνδεσης διατίθεται, και τι λειτουργίες διατίθενται για να τοποθετηθούν κατά σειρά οι πράξεις που εκτελέστηκαν σε διαφορετικούς επεξεργαστές; Ποιες είναι οι πρωτογενείς αφαιρέσεις που παρέχει το hardware και το software στον προγραμματιστή; Και τέλος, πως όλα αυτά μεταφράζονται σε απόδοση; Για την απάντηση αυτών των ερωτημάτων, πρέπει να καταλάβουμε ότι οι μικρές, μέτριες και μεγάλες συλλογές από υπολογιστικές μονάδες έχουν καθεμία από αυτές σημαντικό ρόλο να καλύψουν στη σύγχρονη τεχνολογία υπολογιστών. Συνεπώς, είναι σημαντικό να κατανοήσουμε το σχεδιασμό παράλληλων συστημάτων κλιμακωτά, από το μικρό στο πολύ μεγάλο. Κάποια από τα προβλήματα σχεδίασης αφορούν καθ' όλη την κλίμακα σχεδίασης · άλλα αφορούν συγκεκριμένες περιπτώσεις, όπως είναι το εσωτερικό του τσιπ, το εσωτερικό του κουτιού, ή ένα πολύ μεγάλο μηχάνημα. Δεν είναι υπερβολή να πει κανείς ότι τα παράλληλα συστήματα καταλαμβάνουν ένα πλούσιο και ποικίλο χώρο σχεδίασης. Αυτή η ποικιλία κάνει το χώρο αυτό ενδιαφέρων, αλλά σημαίνει επίσης ότι πρέπει να αναπτύξουμε ένα καθαρό πλαίσιο από το οποίο να γίνονται κατανοητές οι πολλές εναλλακτικές σχεδίασης.

Η παράλληλη αρχιτεκτονική είναι ένας τομέας που αλλάζει ραγδαία. Ιστορικά, τα παράλληλα μηχανήματα έχουν παρουσιάσει καινοτόμες δομές οργάνωσης, συχνά συνδεδεμένες με συγκεκριμένα υπολογιστικά μοντέλα, καθώς οι αρχιτέκτονες αναζητούσαν την αποκόμιση της μέγιστης επίδοσης από μία συγκεκριμένη τεχνολογία. Σε πολλές περιπτώσεις, οι ακραίες οργανωτικές δομές δικαιολογούνται με τη δικαιολογία ότι οι εξελίξεις στην τεχνολογία βάσης έχουν ημερομηνία λήξης. Αυτές οι αρνητικές προβλέψεις φαίνεται να υπέρβαλλαν, καθώς οι πυκνότητα των λογικών κυκλωμάτων και οι ταχύτητες εναλλαγής έχουν συνεχίσει να βελτιώνονται και μετριότερος παραλληλισμός έχει εφαρμοσθεί σε χαμηλότερα επίπεδα για να διατηρήσει τη συνέχιση της βελτίωσης στην απόδοση των επεξεργαστών. Παρόλα αυτά, η απαίτηση των εφαρμογών για υπολογιστική ισχύ συνεχίζει να

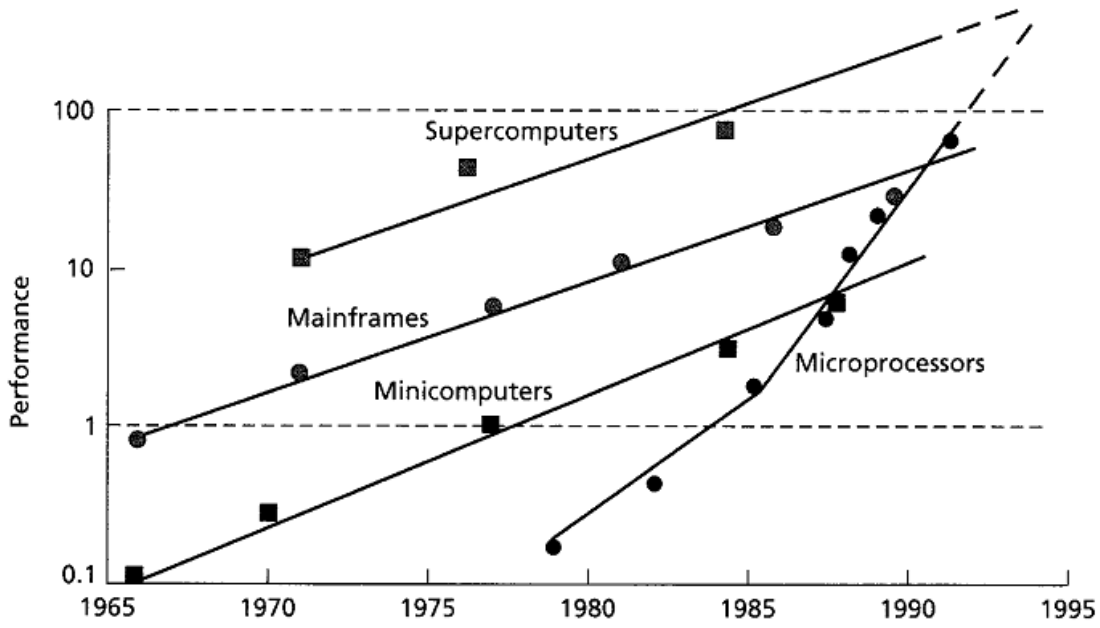
υπερτερεί από αυτή που μπορούν οι επεξεργαστές να προσφέρουν από μόνοι τους, και τα πολυεπεξεργαστικά συστήματα λαμβάνουν μία αυξανόμενα σημαντική θέση στην mainstream αγορά υπολογιστών. Το τι έχει αλλάξει είναι η καινοτομία αυτών των καινούργιων αρχιτεκτονικών. Ακόμη και παράλληλα συστήματα μεγάλης κλίμακας φτιάχνονται σήμερα έχοντας τα ίδια βασικά συστατικά όπως οι σταθμοί εργασίας και οι προσωπικοί υπολογιστές. Υπόκεινται στις ίδιες αρχές μηχανικής και στις συνέπειες του ισοζυγίου κόστους – επίδοσης. Ακόμη, για να αποφέρουν το μέγιστο σε επίδοση, ένα παράλληλο σύστημα πρέπει να απομυζά τα μέγιστα από καθένα των επιμέρους συστατικών του. Συνεπώς, για την κατανόηση των μοντέρνων παράλληλων αρχιτεκτονικών είναι απαραίτητη η σε βάθος αντιμετώπιση των συνεπειών μηχανικής σχεδίασης, και όχι απλά η μελέτη της περιγραφικής ταξινομίας πιθανών δομών παράλληλων συστημάτων.

1.2 Γιατί Επιλέγουμε Παράλληλη Αρχιτεκτονική

Η αρχιτεκτονική υπολογιστών, η τεχνολογία, και οι εφαρμογές εξελίσσονται μαζί και παρουσιάζουν έντονες αλληλεπιδράσεις μεταξύ τους. Η παράλληλη αρχιτεκτονική υπολογιστών δεν αποτελεί εξαίρεση. Μία νέα διάσταση προστίθεται στο χώρο σχεδίασης – ο αριθμός των επεξεργαστών – και ο σχεδιασμός είναι ακόμη πιο ισχυρά καθοδηγούμενος από τη ζήτηση για επίδοση μέσα σε λογικά οικονομικά κόστη. Όση και να είναι η επίδοση ενός μονοπύρηνου επεξεργαστή σε μια δεδομένη χρονική στιγμή, μπορεί πάντα ,θεωρητικά, να επιτευχθεί μεγαλύτερη επίδοση χρησιμοποιώντας μία πληθώρα τέτοιων επεξεργαστών μαζί. Το πόση επιπλέον επίδοση επιτυγχάνεται και με ποιο επιπλέον κόστος εξαρτάται από ένα αριθμό παραγόντων.

Για την καλύτερη κατανόηση αυτή την αλληλεπίδραση, ας θεωρήσουμε τα χαρακτηριστικά επίδοσης δομικών υπολογιστικών συστημάτων. Στο Σχήμα 1.1[1] παρουσιάζεται η αύξηση στην επίδοση των επεξεργαστών στη διάρκεια του χρόνου για πληθώρα κλάσεων υπολογιστικών συστημάτων [2]. Η διακεκομμένες επεκτάσεις των γραμμών αντιπροσωπεύουν μία πρόβλεψη της συνέχισης των τάσεων. Παρόλο που θα έπρεπε να είμαστε προσεχτικοί στην εξαγωγή ποσοτικών συμπερασμάτων από τόσο περιορισμένα δεδομένα, μπορούμε να κάνουμε αρκετές σημαντικές παρατηρήσεις από τη γραφική.

[1] Το Σχήμα αυτό πάρθηκε από ένα σημαίνων paper που προσπαθούσε να εξηγήσει τις δραματικές αλλαγές που λαμβάνουν χώρα στη βιομηχανία υπολογιστών [2]. Η μετρική της επίδοσης είναι λίγο δύσκολη γιατί αφορά μεγάλο χρονικό διάστημα και πληθώρα τμημάτων αγοράς. Η μελέτη παίρνει δεδομένα από benchmarks γενικής χρήσης, όπως είναι το SPEC, το οποίο είναι ευρέως χρησιμοποιούμενο για την αποτίμηση της επίδοσης σε τεχνικές υπολογιστικές εφαρμογές [27]. Μετά την έκδοση του paper, η μικροεπεξεργαστές συνέχισαν να ακολουθούν την προβλεφθείσα πορεία ενώ οι mainframes και οι supercomputers εισήλθαν σε περιόδους κρίσης και άρχισαν να ανακάμπτουν μόνο μετά που άρχισαν να χρησιμοποιούν πολλαπλούς μικροεπεξεργαστές CMOS για να ικανοποιήσουν το τμήμα της αγοράς τους.



Σχήμα 1.1: Τάσεις στην επίδοση στη διάρκεια του χρόνου υπολογιστικών συστημάτων micros, minicomputers, mainframes και supercomputers.

Πρώτον, η επίδοση επεξεργαστών τεχνολογίας CMOS ενός-τσιπ υψηλής ολοκλήρωσης, αυξάνεται σταδιακά και υποσκελίζει εκείνη των μεγαλύτερων, πιο ακριβών εναλλακτικών λύσεων. Η απόδοση των μικροεπεξεργαστών αυξάνεται με ρυθμό περίπου 50% ανά χρόνο. Τα πλεονεκτήματα της χρήσης μικρών, φθηνών, χαμηλής κατανάλωσης, μαζικής παραγωγής επεξεργαστών ως τα δομικά στοιχεία υπολογιστικών συστημάτων με πολλούς επεξεργαστές είναι ξεκάθαρα. Ωστόσο, μέχρι πρότινος η επίδοση των επεξεργαστών που εξυπηρετούσαν καλύτερα την παράλληλη αρχιτεκτονική ήταν πολύ πίσω από εκείνη του γρηγορότερου μονοπύρηνου επεξεργαστή. Αυτό δεν ισχύει πλέον. Παρόλο που τα παράλληλα συστήματα έχουν κατασκευαστεί σε διάφορες κλίμακες από τις πρώτες μέρες των υπολογιστών, η προσέγγιση είναι πιο εφαρμόσιμη σήμερα από ποτέ άλλοτε επειδή το βασικό υπολογιστικό μπλοκ ταιριάζει πολύ καλύτερα για τις ανάγκες της παράλληλης επεξεργασίας.

Η δεύτερη και ίσως πιο σημαντική παρατήρηση είναι ότι οι αλλαγές στο χώρο της αρχιτεκτονικής υπολογιστών, ακόμη και η δραματικές αλλαγές, αποτελούν κανόνα και όχι εξαίρεση. Η συνεχείς αλλαγές επιφέρουν βαθιές επιπτώσεις στον τρόπο μελέτης της αρχιτεκτονικής υπολογιστών γιατί ένας δεν πρέπει να κατανοήσει μόνο πως έχουν τα πράγματα σήμερα αλλά και πώς μπορούν να εξελιχτούν και γιατί.

1.2.1 Τάσεις στις Προγραμματιστικές Εφαρμογές

Η απαίτηση για ακόμη μεγαλύτερη επίδοση στις προγραμματιστικές εφαρμογές είναι ένα σύνηθες χαρακτηριστικό σε όλους τους τομείς της τεχνολογίας των υπολογιστών. Η πρόοδος στις δυνατότητες του hardware δίνει τη δυνατότητα στις εφαρμογές για νέες λειτουργίες, οι

οποίες αυξάνουν σε απαιτήσεις και τοποθετούν ακόμη μεγαλύτερες απαιτήσεις στην αρχιτεκτονική. Αυτή η κυκλική εξάρτηση καθοδηγεί την τρομακτική εξέλιξη στον σχεδιασμό, στη μηχανική, και στις διαδικασίες κατασκευής που αποτελούν τον πυρήνα της συντηρούμενης εκθετικής αύξησης της επίδοσης στην απόδοση των μικροεπεξεργαστών. Την παράλληλη αρχιτεκτονική την καθοδηγεί σε ακόμη μεγαλύτερους ρυθμούς εξέλιξης εφόσον αυτή επικεντρώνεται στις πιο απαιτητικές των εφαρμογών. Με μία αύξηση στην απόδοση των επεξεργαστών της τάξης του 50% κάθε χρόνο, ένα μηχάνημα παράλληλης επεξεργασίας με εκατό επεξεργαστές μπορεί να θεωρηθεί ως ένα μηχάνημα που παρέχει σε εφαρμογές υπολογιστική ισχύ που θα είναι ευρέως διαθέσιμη σε 10 χρόνια, ενώ με χίλιους επεξεργαστές αντανακλά ένα χρονικό ορίζονται 20 χρόνων.

Οι απαιτήσεις των εφαρμογών επίσης καθοδηγούν τους πωλητές υπολογιστικών συστημάτων να παρέχουν μια ευρεία γκάμα προϊόντων που παρουσιάζουν αυξανόμενα επίπεδα επίδοσης και χωρητικότητας με σταδιακά αυξανόμενο κόστος. Ο μεγαλύτερος όγκος μηχανημάτων και ο μεγαλύτερος αριθμός χρηστών βρίσκεται στο low-end, ενώ οι πιο απαιτητικές εφαρμογές εξυπηρετούνται από το high-end τμήμα της αγοράς. Ένα αποτέλεσμα αυτής της "πλατφόρμας με μορφή πυραμίδας" είναι ότι η πίεση για υψηλή επίδοση είναι μεγαλύτερη στο τμήμα high-end και ασκείται από μία σημαντική μειονότητα εφαρμογών. Πριν την επικράτηση των μικροεπεξεργαστών, η αυξημένη επίδοση επιτυγχανόταν με χρήση εξωτικών κυκλωματικών τεχνολογιών και αρχιτεκτονικές μηχανών. Σήμερα, για την αποκόμιση επίδοσης σημαντικά μεγαλύτερης από τον καλύτερο μικροεπεξεργαστή, βασική επιλογή αποτελεί η χρήση πολλαπλών επεξεργαστών, και οι πιο απαιτητικές εφαρμογές γράφονται ως παράλληλα προγράμματα. Συνεπώς, οι παράλληλες αρχιτεκτονικές και τα παράλληλα προγράμματα υπόκεινται στις πιο έντονες απαιτήσεις για επίδοση.

Ένα θεμελιώδης σημείο αναφοράς για αμφότερους των αρχιτεκτόνων υπολογιστών και των μηχανικών λογισμικού είναι το πώς η χρήση παραλληλισμού βελτιώνει την επίδοση του προγράμματος. Μπορούμε να ορίσουμε το *speedup* (επιτάχυνση) σε *n* επεξεργαστές ως εξής:

$$Speedup(n \text{ επεξεργαστές}) \equiv \frac{\text{Επίδοση}(n \text{ επεξεργαστές})}{\text{Επίδοση}(1 \text{ επεξεργαστής})}$$

Για ένα συγκεκριμένο πρόβλημα, η απόδοση ενός μηχανήματος για την περίπτωση αυτή είναι απλά το αντίστροφο του χρόνου που χρειάζεται για τη λύση του προβλήματος, έτσι έχουμε την ακόλουθη σημαντική ειδική περίπτωση:

$$Speedup_{\text{συγκεκριμένο πρόβλημα}}(n \text{ επεξεργαστές}) = \frac{\text{Χρόνος}(1 \text{ επεξεργαστής})}{\text{Χρόνος}(n \text{ επεξεργαστές})}$$

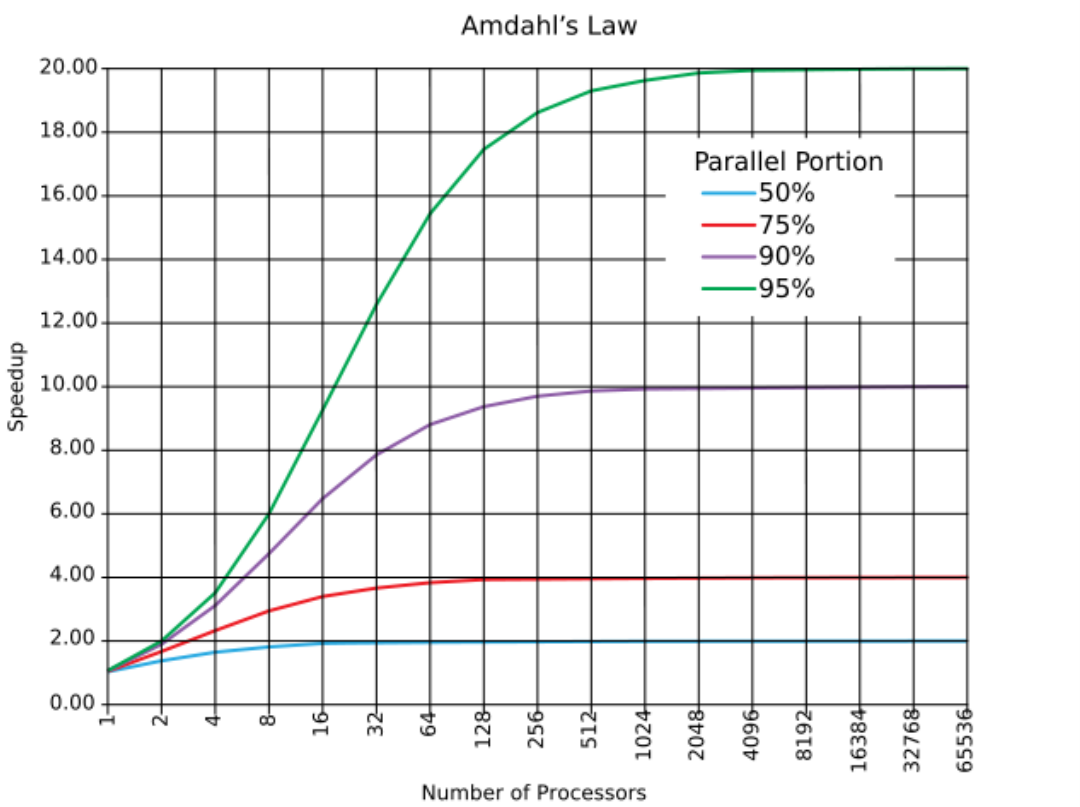
Δεν πρέπει να παρεξηγήσει κανείς τις παραπάνω σχέσεις θεωρώντας ότι η επίδοση όταν έχουμε *n* επεξεργαστές είναι *n* φορές εκείνης του ενός επεξεργαστή. Κάτι τέτοιο δεν ισχύει στη γενική περίπτωση. Για να κατανοήσουμε καλύτερα την επίδοση ενός προγράμματος πρέπει να δούμε το *νόμο του Amdahl* [3]. Ο νόμος αυτός αναφέρεται στην αύξηση σε επίδοση που μπορεί να επιτευχθεί από μία βελτίωση σε κάποιο τμήμα ενός προγράμματος, που επηρεάζει ένα ποσοστό *P* του ολικού προγράμματος, και η βελτίωση αυτή παρουσιάζει ένα speedup *S* (για παράδειγμα, εάν μία βελτίωση μπορεί να επιταχύνει μέχρι και το 30% του ολικού υπολογισμού, τότε το *P* είναι 0.3 · εάν η βελτίωση κάνει το τμήμα αυτό του υπολογισμού δύο φορές πιο γρήγορο, τότε το *S* θα είναι 2). Το ολικό speedup εφαρμογής αυτής της βελτίωση σε ένα πρόγραμμα δίνεται από τη σχέση:

$$Speedup \equiv \frac{1}{(1-P)+\frac{P}{S}}$$

Στην περίπτωση της παράλληλης επεξεργασίας, που μας ενδιαφέρει, ο *νόμος του Amdahl* μετατρέπεται ως εξής: Εάν *P* είναι το ποσοστό του ολικού προγράμματος που μπορεί να παραλληλοποιηθεί (δηλαδή μπορεί να τροποποιηθεί για να επωφελείται από την παράλληλη επεξεργασία), και (1-*P*) είναι το μέρος του προγράμματος που δεν μπορεί να παραλληλοποιηθεί (παραμένει σειριακό), τότε το μέγιστο speedup που μπορεί να επιτευχθεί χρησιμοποιώντας *N* επεξεργαστές δίνεται από τη σχέση:

$$Speedup_{parallel} \equiv \frac{1}{(1-P)+\frac{P}{N}}$$

Στο όριο, καθώς το *N* τείνει στο άπειρο, το μέγιστο speedup τείνει στο $1/(1-P)$. Στην πράξη, η αναλογία απόδοσης/κόστους μειώνεται δραματικά καθώς το *N* αυξάνει ακόμη και εάν το ποσοστό του (1-*P*) είναι μικρό. Για παράδειγμα, εάν το *P* είναι 90%, και το (1-*P*) συνεπώς 10%, τότε το συνολικό πρόγραμμα μπορεί να επιταχυνθεί από ένα παράγοντα το πολύ 10, όσο μεγάλος και αν είναι ο αριθμός των επεξεργαστών που χρησιμοποιούμε. Στο Σχήμα 1.2 βλέπουμε μία γραφική για διάφορες περιπτώσεις προγραμμάτων με διαφορετικά ποσοστά επαλληλίας και για διάφορους αριθμούς επεξεργαστών.



Σχήμα 1.2: Το speedup ενός προγράμματος που χρησιμοποιεί πολλαπλούς επεξεργαστές σε παράλληλη επεξεργασία σε σχέση με το ποσοστό του προγράμματος που είναι παραλληλοποιήσιμο.

Από τη γραφική παράσταση στο Σχήμα 1.2 μπορούμε να εξάγουμε χρήσιμα συμπεράσματα. Πρώτον, το speedup ενός προγράμματος που χρησιμοποιεί πολλαπλούς επεξεργαστές σε παράλληλη επεξεργασία περιορίζεται από το σειριακό μέρος του προγράμματος. Δηλαδή η χρήση πολλαπλών επεξεργαστών και η δημιουργία των καλύτερων παράλληλων αρχιτεκτονικών δεν είναι πανάκεια. Σε συνδυασμό με την ανάπτυξη της παράλληλης αρχιτεκτονικής πρέπει να υπάρχει ανάλογη ανάπτυξη στη δημιουργία παράλληλων εφαρμογών που μπορούν να εκμεταλλευτούν τις παράλληλες αυτές αρχιτεκτονικές. Η ανάπτυξη του ενός απ' τους δύο παράγοντες δε σημαίνει τίποτα χωρίς το άλλο. Για παράδειγμα, εάν τρέξουμε ένα καθαρά σειριακό πρόγραμμα σε ένα σύστημα με χίλιους επεξεργαστές θα έχουμε την ίδια επίδοση με εάν το τρέχαμε σε ένα επεξεργαστή! Αντίστοιχα εάν τρέξουμε ένα πρόγραμμα 100% παραλληλοποιήσιμο σε ένα σειριακό επεξεργαστή δε θα δούμε καμία βελτίωση στην επίδοση!

1.3 Συγκερασμός των Παράλληλων Αρχιτεκτονικών

Ιστορικά, τα παράλληλα συστήματα έχουν αναπτυχθεί μέσα στα πλαίσια διαφορετικών αρχιτεκτονικών σχολών, και τα πιο πολλά κείμενα για το αντικείμενο αυτό έχουν οργανωθεί

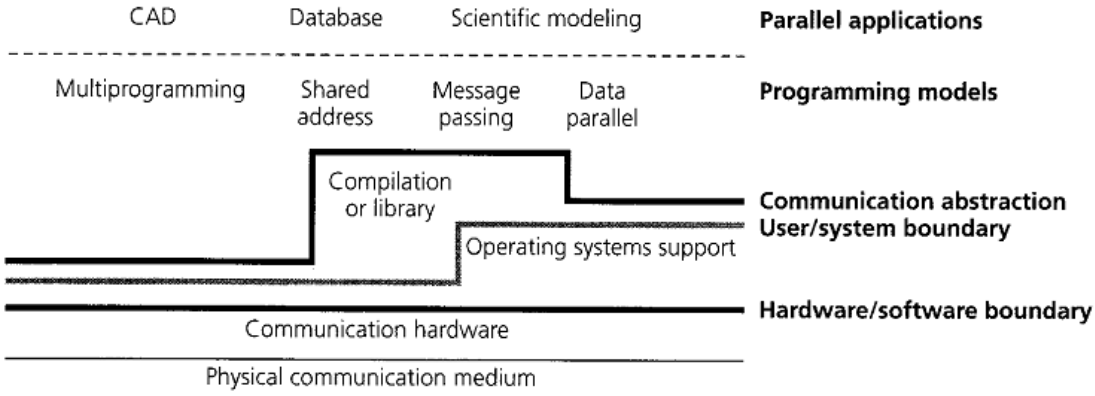
γύρω από αυτούς τους σχεδιασμούς. Ωστόσο, μετά από μελέτη του πώς εξελίχθηκε η παράλληλη αρχιτεκτονική, είναι ξεκάθαρο ότι οι σχεδιασμοί παράλληλων συστημάτων έχουν επηρεαστεί σε μεγάλο βαθμό από τις ίδιες τεχνολογικές τάσεις και από παρόμοιες απαιτήσεις εφαρμογών. Δεν είναι παράξενο λοιπόν που έχει παρατηρηθεί μεγάλο ποσοστό συγκερασμού στον τομέα αυτό.

1.3.1 Αρχιτεκτονική Επικοινωνίας Παράλληλων Συστημάτων

Με δεδομένο ότι ένας παράλληλος υπολογιστής δεν είναι τίποτε άλλο παρά μια συλλογή από μονάδες υπολογισμού που επικοινωνούν και συνεργάζονται για τη γρήγορη επίλυση μεγάλων προβλημάτων, μπορούμε λογικά να δούμε την παράλληλη αρχιτεκτονική ως επέκταση της συμβατικής αρχιτεκτονικής υπολογιστών για την επίλυση των ζητημάτων της επικοινωνίας και συνεργασίας μεταξύ ξεχωριστών υπολογιστικών μονάδων. Στην ουσία, η παράλληλη αρχιτεκτονική επεκτείνει τις βασικές αρχές της αρχιτεκτονικής υπολογιστών προσθέτοντας μια αρχιτεκτονική επικοινωνίας. Η αρχιτεκτονική υπολογιστών δύο ξεχωριστές όψεις. Η μία όψη αφορά τον ορισμό κύριων εννοιών αφαίρεσης, ειδικά τα σύνορα μεταξύ hardware/software και user/system. Η αρχιτεκτονική καθορίζει το σύνολο των λειτουργιών στο σύνορο και τους τύπους δεδομένων στους οποίους αυτές εφαρμόζονται. Η άλλη όψη αφορά τη δημιουργία της κατάλληλης δομής οργάνωσης που υλοποιεί αυτές τις αφαιρετικές έννοιες για να έχουμε υψηλή επίδοση με επωφελές κόστος. Μία αρχιτεκτονική επικοινωνίας παρουσιάζει επίσης αυτές τις δύο όψεις. Ορίζει τις βασικές λειτουργίες επικοινωνίας και συγχρονισμού, και φτιάχνει τις απαραίτητες οργανωτικές δομές για την υλοποίηση αυτών των λειτουργιών.

Το πλαίσιο μέσα στο οποίο μπορεί κανείς να κατανοήσει την επικοινωνία μέσα σε ένα σύστημα παράλληλης επεξεργασίας παρουσιάζεται στο Σχήμα 1.3. Το πρώτο στρώμα είναι το προγραμματιστικό μοντέλο, που αποτελεί το πώς αντιλαμβάνονται τη μηχανή οι προγραμματιστές στην εγγραφή κώδικα για εφαρμογές. Κάθε προγραμματιστικό μοντέλο ορίζει το πώς τα μέρη του προγράμματος που τρέχει παράλληλα επικοινωνούν πληροφορία μεταξύ τους και τι λειτουργίες συγχρονισμού είναι διαθέσιμες για το συντονισμό αυτών των ενεργειών. Οι εφαρμογές γράφονται μέσα στα πλαίσια ενός προγραμματιστικού μοντέλου. Στην απλούστερη περίπτωση, το μοντέλο αποτελείται από τον πολύ-προγραμματισμό ενός μεγάλου αριθμού ανεξάρτητων ακολουθιακών προγραμμάτων · καμία επικοινωνία ή λειτουργία συνεργασίας δε λαμβάνει χώρα σε προγραμματιστικό επίπεδο. Οι πιο ενδιαφέρουσες περιπτώσεις αφορούν πραγματικά παράλληλα προγραμματιστικά μοντέλα, όπως είναι τα *shared address space*, *message passing*, και *data parallel* programming. Μπορούμε να περιγράψουμε διαισθητικά τα μοντέλα αυτά ως ακολούθως:

- Το προγραμματιστικό μοντέλο του *shared address* μπορεί να παρομοιαστεί με ένα πίνακα ανακοινώσεων, όπου μπορείς να επικοινωνήσεις με ένα ή περισσότερους συναδέλφους με την ανάρτηση πληροφορίας σε γνωστές, μοιραζόμενες θέσεις. Οι διάφορες δραστηριότητες μπορούν να οργανωθούν με το να σημειώνεται το ποιος εκτελεί κάποιο έργο
- Το προγραμματιστικό μοντέλο *message passing* είναι όμοιο με τηλεφωνικές κλήσεις ή ταχυδρομικά γράμματα, τα οποία άγουν πληροφορία από συγκεκριμένο αποστολέα σε συγκεκριμένο παραλήπτη. Υπάρχει ένα καλά ορισμένο γεγονός κατά το οποίο πληροφορία αποστέλλεται ή λαμβάνεται, και αυτά τα γεγονότα είναι η βάση με την οποία οργανώνονται οι διάφορες δραστηριότητες. Όμως δεν υπάρχει καμία μοιραζόμενη τοποθεσία προσβάσιμη από όλους.
- Το μοντέλο *data parallel* είναι μία πιο πειθαρχημένη μορφή επικοινωνίας, όπου διάφοροι πράκτορες εκτελούν μία ενέργεια σε διαφορετικά στοιχεία ενός συνόλου δεδομένων ταυτόχρονα και στη συνέχεια ανταλλάσσουν πληροφορίες καθολικά πριν συνεχίσουν την πρόοδό τους. Η καθολική ανά-οργάνωση των δεδομένων μπορεί να επιτευχθεί διαμέσου προσβάσεων σε κοινές θέσεις ή με μηνύματα αφού το προγραμματιστικό μοντέλο ορίζει μόνο το ολικό αποτέλεσμα των παράλληλων βημάτων.



Σχήμα 1.3: Τα διάφορα στρώματα αφαίρεσης στην παράλληλη αρχιτεκτονική υπολογιστών.

Ένα προγραμματιστικό μοντέλο υλοποιείται σε όρους πρωτογενών κλήσεων επικοινωνίας user-level του συστήματος, στις οποίες αναφερόμαστε ως αφαίρεση στο επίπεδο της επικοινωνίας. Τυπικά, το προγραμματιστικό μοντέλο ενσωματώνεται σε μία παράλληλη γλώσσα προγραμματισμού ή σε ένα προγραμματιστικό περιβάλλον, έτσι υπάρχει μία αντιστοίχηση μεταξύ των γενικών δομικών στοιχείων της γλώσσας και των συγκεκριμένων πρωτογενών κλήσεων του συστήματος. Αυτές οι πρωτογενείς κλήσεις στο user-level μπορεί να παρέχονται κατευθείαν από το hardware, το λειτουργικό σύστημα, ή από software ρητά φτιαγμένο για το συγκεκριμένο μηχάνημα που αντιστοιχεί τα αφαιρετικά στοιχεία σε επίπεδο επικοινωνίας στις πραγματικές πρωτογενείς κλήσεις του hardware. Η απόσταση μεταξύ των

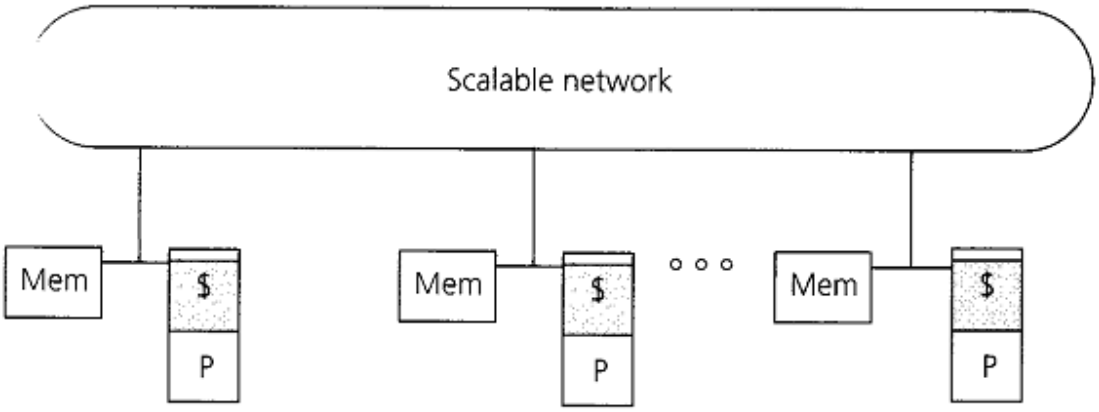
γραμμών στο Σχήμα 1.3 δείχνει ότι η χαρτογράφηση από το ένα στρώμα στο επόμενο μπορεί να είναι πολύ απλή ή πολύ περίπλοκη. Για παράδειγμα, η πρόσβαση σε μία κοινή τοποθεσία επιτυγχάνεται κατευθείαν με τη χρήση εντολών load και store σε ένα μηχάνημα στο οποίο όλοι οι επεξεργαστές χρησιμοποιούν την ίδια φυσική μνήμη · αντίθετα, η μεταφορά μηνυμάτων σε ένα τέτοιο μηχάνημα μπορεί να περικλείει μία βιβλιοθήκη ή μια κλήση συστήματος για να γραφεί το μήνυμα σε ένα buffer ή για να διαβαστεί.

Η αρχιτεκτονική επικοινωνίας καθορίζει το σύνολο των λειτουργιών επικοινωνίας που διατίθενται στα προγράμματα χρήστη, τη διάταξη αυτών των λειτουργιών, και τους τύπους δεδομένων στους οποίους εφαρμόζονται, παρόμοια με τον τρόπο που μία αρχιτεκτονική συνόλου εντολών (instruction set) κάνει για ένα επεξεργαστή. Σημειώστε ότι ακόμη και σε συμβατικά σύνολα εντολών, κάποιες λειτουργίες επιτυγχάνονται από ένα συνδυασμό hardware και software, όπως είναι η εντολή load που στηρίζεται στη μεσολάβηση του λειτουργικού συστήματος στην περίπτωση που υπάρξει σφάλμα σελίδας.

Στη συνέχεια θα επικεντρωθούμε στο προγραμματιστικό μοντέλο του *message passing*, με το οποίο ασχολούμαστε στα πλαίσια της διπλωματικής εργασίας.

1.3.2 Προγραμματιστικό Μοντέλο Αποστολής Μηνυμάτων (Message Passing)

Μία σημαντική κλάση μηχανημάτων παράλληλης επεξεργασίας, αποτελούν οι αρχιτεκτονικές *message-passing*. Οι αρχιτεκτονικές αυτές χρησιμοποιούν ως δομικά μπλοκ πλήρη υπολογιστικά συστήματα –συμπεριλαμβανομένου μικροεπεξεργαστή, μνήμης, και συστήματος I/O- και παρέχουν τη δυνατότητα επικοινωνίας μεταξύ επεξεργαστών ως ρητές λειτουργίες I/O. Το μπλοκ διάγραμμα για ένα μηχάνημα *message-passing* είναι παρουσιάζεται στο Σχήμα 1.4. Αυτό το στυλ σχεδιασμού έχει πολλά κοινά στοιχεία με δίκτυα από workstation, ή clusters, αν εξαιρέσει κανείς ότι το πακέτο των κόμβων είναι τυπικά πολύ πιο σφικτό, με την έννοια ότι δεν υπάρχει οθόνη ή πληκτρολόγιο ανά κόμβο, και ότι το δίκτυο έχει πολύ υψηλότερες δυνατότητες απ’ ότι ένα τυπικό τοπικό δίκτυο. Η σχέση μεταξύ του επεξεργαστή και του δικτύου τείνει να είναι πολύ πιο στενή από τις παραδοσιακές δομές I/O, οι οποίες υποστηρίζουν διασύνδεση με συσκευές που είναι πολύ πιο αργές από τον επεξεργαστή, αφού το *message-passing* είναι ουσιαστικά επικοινωνία επεξεργαστή-με-επεξεργαστή.

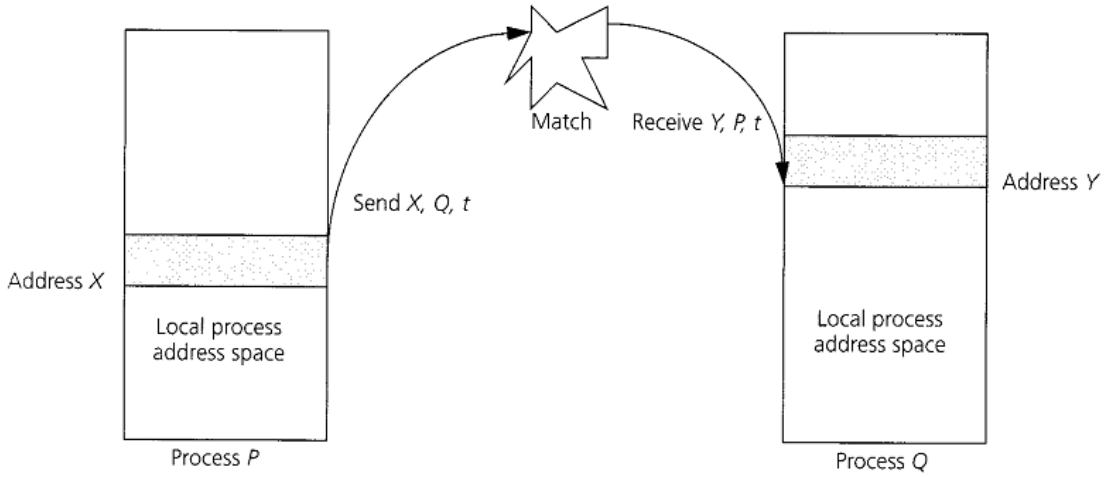


Σχήμα 1.4: Παράδειγμα μπλοκ διαγράμματος για το προγραμματιστικό μοντέλο *message passing*. (Όπου: P επεξεργαστής, $ κρυφή μνήμη)

Στο *message-passing*, υπάρχει μία σημαντική απόσταση μεταξύ του προγραμματιστικού μοντέλου και των πραγματικών πρωτογενών κλήσεων του hardware, με την επικοινωνία σε επίπεδο χρήστη να εκτελείται διαμέσου του λειτουργικού συστήματος ή με κλήσεις βιβλιοθήκης που εκτελούν πολλές ενέργειες χαμηλού επιπέδου, περιλαμβανομένου της κατά πράξη επικοινωνίας. Συνεπώς, αρχίζουμε την εξέταση του προγραμματιστικού μοντέλου *message-passing* με τη μελέτη της αφαίρεσης σε επίπεδο επικοινωνίας και στη συνέχεια με την εξέλιξη του hardware που υποστηρίζει αυτό το είδος αφαίρεσης.

Οι πιο κοινά χρησιμοποιούμενες λειτουργίες επικοινωνίας σε επίπεδο χρήστη σε συστήματα *message-passing* δεν είναι τίποτε άλλο από παραλλαγές των βασικών λειτουργιών send και receive. Στην πιο απλή του μορφή, το send ορίζει ένα τοπικό buffer δεδομένων το οποίο πρόκειται να αποσταλεί και μία διεργασία λήψης (τυπικά στον απομακρυσμένο επεξεργαστή). Η λειτουργία receive ορίζει μία διεργασία αποστολής και ένα τοπικό buffer δεδομένων μέσα στο οποίο θα τοποθετηθούν τα ληφθέντα δεδομένα. Μαζί, οι ταιριαστές λειτουργίες send και receive προκαλούν μία μετάδοση δεδομένων από τη μία διεργασία στην άλλη, όπως φαίνεται στο Σχήμα 1.5. Στα πλείστα συστήματα *message-passing*, η λειτουργία send επιτρέπει επίσης να επισυναφθεί σε ένα μήνυμα ένα αναγνωριστικό ή μια ετικέτα, και η λειτουργία receive καθορίζει ένα αντίστοιχο κανόνα για ταίριασμα των ετικετών και αναγνωριστικών. Συνεπώς, το πρόγραμμα χρήστη ονοματίζει τοπικές διευθύνσεις και εγγραφές σε ένα αφαιρετικό χώρο ετικετών διεργασίας. Ο συνδυασμός ενός send με το αντίστοιχό του receive πετυχαίνει ένα ζευγαρωμένο συγχρονισμένο συμβάν και μία αντιγραφή από μνήμη-σε-μνήμη, όπου κάθε άκρο καθορίζει τη διεύθυνση των τοπικών του δεδομένων. Υπάρχουν πολλές δυνατές παραλλαγές αυτού του συγχρονισμένου συμβάντος, που εξαρτώνται στο κατά πόσο το send ολοκληρώνεται όταν έχει εκτελεστεί το receive, ή

όταν το buffer είναι διαθέσιμο για να επαναχρησιμοποιηθεί, ή όταν έχει γίνει αποδεχτή η αίτηση. Παρόμοια, η λειτουργία receive μπορεί ενδεχομένως να περιμένει μέχρι να συμβεί ένα αντίστοιχο με αυτή send ή απλά να εκδώσει το receive. Καθεμία από αυτές τις παραλλαγές έχει μερικώς διαφορετική σημασία και διαφορετικές απαιτήσεις υλοποίησης.



Σχήμα 1.5: Η αφαιρετική προσέγγιση σε επίπεδο user-level των λειτουργιών send/receive στο μοντέλο *message-passing*

Το μοντέλο *message-passing* έχει χρησιμοποιηθεί εδώ και καιρό σαν ένα μέσο επικοινωνίας και συγχρονισμού μεταξύ αυθαίρετων συλλογών από συνεργαζόμενες ακολουθιακές διεργασίες, ακόμη και σε ένα επεξεργαστή. Σημαντικά παραδείγματα περιλαμβάνουν γλώσσες προγραμματισμού, όπως είναι η CSP και η Occam, και κοινές λειτουργίες λειτουργικών συστημάτων, όπως είναι τα socket. Τα παράλληλα προγράμματα που χρησιμοποιούν το μοντέλο *message-passing* είναι συνήθως αρκετά δομημένα. Στις περισσότερες περιπτώσεις, όλοι οι κόμβοι εκτελούν ταυτόσημα αντίγραφα ενός προγράμματος, με τον ίδιο κώδικα και τις ίδιες ιδιωτικές μεταβλητές. Συνήθως, οι διεργασίες μπορούν να αριθμήσουν η μία την άλλη χρησιμοποιώντας απλή γραμμική διάταξη των διεργασιών που αποτελούν το πρόγραμμα.

Κεφάλαιο 2

Πολυνηματικές Αρχιτεκτονικές

2.1 Γενικά

Ο αριθμός των τρανζίστορ σε ένα τσιπ αυξάνεται με εκθετικό ρυθμό, διπλασιάζεται κάθε περίπου δύο χρόνια, έχοντας ήδη ξεπεράσει το ένα δισεκατομμύριο τρανζίστορ (Nvidia® Gt200™). Το προφανές ερώτημα που γεννάται είναι πώς θα χρησιμοποιήσει κανείς αυτά τα τρανζίστορ. Μία δυνατότητα είναι η προσθήκη περισσότερης μνήμης (είτε κρυφής είτε «κύριας») στο τσιπ ˙ όμως, υπάρχει ένα όριο στην αύξηση της επίδοσης που μπορεί να επιφέρει η προσθήκη μνήμης και μόνο. Μία άλλη προσέγγιση θα ήταν να αυξήσουμε τον αριθμό των συστημάτων που υλοποιούνται σε ένα τσιπ, φέρνοντας έτσι διάφορες συσκευές υποστήριξης που μέχρι τώρα είναι εκτός τσιπ (όπως επιταχυντές γραφικών, ελεγκτές Ε/Ε και διεπαφές), πετυχαίνοντας έτσι μείωση του κόστους επικοινωνίας. Κάτι τέτοιο μειώνει το κόστος ενός συστήματος, αλλά έχει επίσης μειωμένο αντίκτυπο στην επίδοση.

Τα σύγχρονα υπολογιστικά συστήματα για να πετύχουν υψηλά επίπεδα απόδοσης εκμεταλλεύονται δύο μορφές παραλληλίας: την παραλληλία σε επίπεδο εντολής (*instruction level parallelism* ILP, από ένα πρόγραμμα ή νήμα) και επαλληλία σε επίπεδο νήματος (*thread level parallelism* TLP, είτε από πολυνηματικά παράλληλα προγράμματα είτε από ξεχωριστά προγράμματα σε πολυπρογραμματιστικό περιβάλλον). Οι *υπερβαθμωτοί επεξεργαστές* (superscalars) εκμεταλλεύονται την ILP εκτελώντας πολλαπλές (ανεξάρτητες) εντολές από ένα πρόγραμμα στον ίδιο κύκλο μηχανής. Οι *πολυεπεξεργαστές* από την άλλη εκμεταλλεύονται την επαλληλία TLP εκτελώντας εντολές διαφορετικών νημάτων σε κάθε διαφορετικό επεξεργαστικό πυρήνα. Δυστυχώς, και οι δύο μέθοδοι παραλληλοποίησης διαμοιράζουν στατικά τους υπολογιστικούς τους πόρους και δεν έχουν τη δυνατότητα να προσαρμοστούν δυναμικά στις ενδεχόμενες αυξομειώσεις σε TLP και ILP κατά την εκτέλεση ενός φόρτου εργασίας.

Μία αρχιτεκτονική ικανή να καταναλώσει παραλληλία οποιουδήποτε τύπου - επαλληλία σε επίπεδο εντολών (instruction level parallelism ILP, από ένα πρόγραμμα ή νήμα) και επαλληλία σε επίπεδο νημάτων (thread level parallelism TLP, είτε από πολυνηματικά

® Nvidia είναι σήμα κατατεθέν της Nvidia Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες

™ GT200 είναι σήμα κατατεθέν της Nvidia Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες

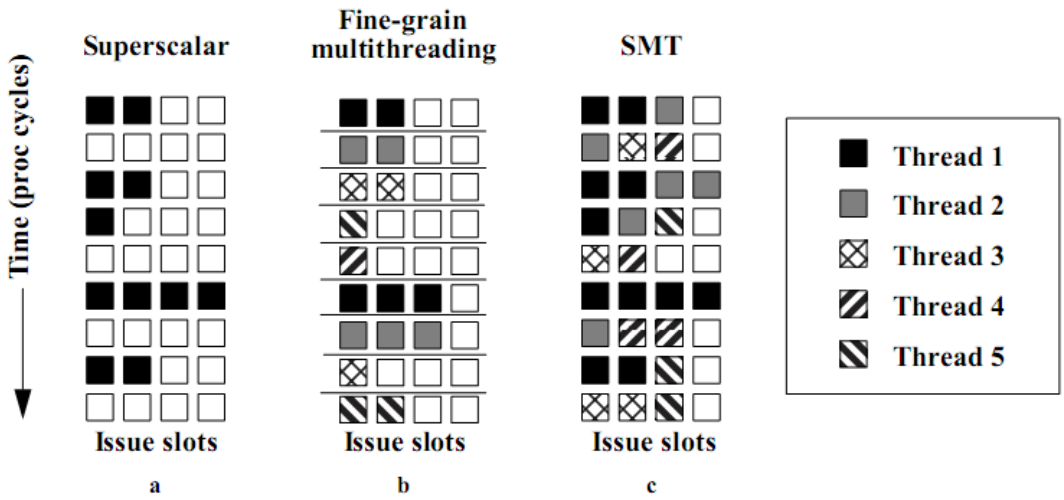
παράλληλα προγράμματα είτε από ξεχωριστά προγράμματα σε πολυπρογραμματιστικό περιβάλλον) – για να διατηρήσει την χρήση του επεξεργαστή σε υψηλά επίπεδα και να αυξήσει την διεκπεραιωτική του ικανότητα, είναι η αρχιτεκτονική του *Simultaneous Multithreading* (SMT). Στη συνέχεια της παραγράφου παρουσιάζουμε την αρχιτεκτονική SMT με περισσότερη λεπτομέρεια και μια πρώτη υλοποίησή της με την τεχνολογία Hyper-Threading της Intel.

2.2 Αρχιτεκτονική Simultaneous Multithreading (SMT)

Το simultaneous multithreading συνδυάζει χαρακτηριστικά δύο άλλων τύπων επεξεργαστών: των wide-issue superscalar και των multithreaded επεξεργαστών. Από τους μεν πρώτους παίρνει την ικανότητα να εκδίδει πολλές εντολές ανά κύκλο και από τους δεύτερους την ικανότητα να εκτελεί διάφορα προγράμματα (ή νήματα) ταυτόχρονα. Το αποτέλεσμα είναι ένα επεξεργαστής που μπορεί να εκδώσει πολλές εντολές από διάφορα νήματα σε κάθε κύκλο.

Η διαφορά μεταξύ superscalar, multithreaded και simultaneous multithreading φαίνεται στο Σχήμα 2.1, όπου παρουσιάζεται ένα παράδειγμα ακολουθίας εκτέλεσης για τις τρεις αρχιτεκτονικές. Στα σχήματα αυτά, κάθε γραμμή αναπαριστά ένα κύκλο εκτέλεσης και τα τέσσερα κουτάκια δείχνουν τις τέσσερις πιθανές εντολές που μπορεί ο επεξεργαστής να εκδώσει ανά κύκλο. Ένα γεμάτο κουτί δείχνει ότι ο επεξεργαστής μπόρεσε να βρει μια εντολή για να εκδώσει σε αυτή τη θυρίδα, ενώ ένα άδειο κουτί δείχνει αχρησιμοποίητη ή σπατάλη (wasted) θυρίδας. Διαχωρίζουμε τις θυρίδες εντολών που σπαταλώνται σε δύο τύπους. *Οριζόντια σπατάλη* (horizontal waste) παρατηρείται όταν κάποια, αλλά όχι όλες, οι θυρίδες μπορούν να χρησιμοποιηθούν σε κάθε κύκλο. Αυτό είναι τυπικό δείγμα έλλειψης ILP σε ένα πρόγραμμα. *Κατακόρυφη σπατάλη* (vertical waste) παρατηρείται όταν ένας κύκλος μένει πλήρως αχρησιμοποίητος. Αυτό μπορεί να προκληθεί από μία εντολή με μεγάλη καθυστέρηση (όπως μία ανάγνωση μνήμης) που κατακρατά την έκδοση περεταίρω εντολών.

Ένας τυπικός superscalar, όπως ο DEC Alpha 21164, ο HP PA-8000, ο Intel Pentium Pro,ο MIPS R10000 και ο PowerPC 604 φαίνεται στο Σχήμα 2.1α. Σύμφωνα με τη λειτουργία των υπερβαθμωτών, εκτελεί ένα και μόνο πρόγραμμα ή νήμα, από το οποίο προσπαθεί να βρει πολλαπλές εντολές για να εκδώσει σε κάθε κύκλο. Σε περίπτωση που αποτύχει, οι θυρίδες έκδοσης εντολών μένουν αχρησιμοποίητες, κάτι που προκαλεί τόσο οριζόντια όσο και κάθετη σπατάλη. Οι αρχιτεκτονικές multithreaded (π.χ. ο Tera), από την άλλη, περιέχουν κατάσταση υλικού – μετρητή προγράμματος και καταχωρητές – για αρκετά νήματα. Σε κάθε δεδομένο κύκλο εκτέλεσης ο επεξεργαστής εκτελεί



Σχήμα 2.1: Σύγκριση του partitioning των θυρίδων εντολών σε διάφορες αρχιτεκτονικές

εντολές από ένα εκ των νημάτων. Στον επόμενο κύκλο, εκτελεί context switching σε context άλλου νήματος, έτσι ώστε να μπορεί να εκτελέσει εντολές από το νέο νήμα. Το context switching μπορεί να γίνει σε ένα μόνο κύκλο, επειδή οι πολλαπλές καταστάσεις υλικού(hardware contexts) που περιέχουν αντικαθιστούν την ανάγκη για αποθήκευση και επαναφορά της κατάστασης του επεξεργαστή. Το context switching νημάτων και η επακόλουθη ακολουθία εκτέλεσης φαίνονται στο Σχήμα 2.1b. Τα διάφορα σχεδιαστικά μορφήματα αναπαριστούν διαφορετικά νήματα, τα οποία εκδίδουν εντολές σε διαφορετικούς κύκλους. Στο σχήμα διαφαίνεται το κύριο πλεονέκτημα του πολυνηματισμού, και συγκεκριμένα η καλύτερη ανοχή σε εντολές με μεγάλες καθυστερήσεις, γιατί μπορεί να δρομολογήσει κάποιο άλλο νήμα κατά τη διάρκεια που ένα νήμα έχει μπλοκάρει. Αξίζει να σημειωθεί, όμως, ότι παρόλο που στην αρχιτεκτονική αυτή δεν παρουσιάζεται κατακόρυφη σπατάλη, έχει αυξηθεί η οριζόντια σπατάλη, αφού υπάρχει μετατροπή κάποιας από την πρότινος κατακόρυφη σπατάλη σε οριζόντια. Συνεπώς, καθώς το παράθυρο εκτέλεσης πλαταίνει οι πολυνηματικές αρχιτεκτονικές θα έχουν τελικά την ίδια μοίρα με τους υπερβαθμωτούς, δηλαδή δε θα έχουν τη δυνατότητα να βρουν αρκετό ILP σε ένα νήμα που εκτελείται μόνο του για να αξιοποιήσουν επαρκώς τον επεξεργαστή.

Η αρχιτεκτονική simultaneous multithreading, όπως φαίνεται στο Σχήμα 2.1c, συνδυάζει τα καλύτερα στοιχεία των αρχιτεκτονικών multithreaded και superscalar. Όπως τους superscalar, ένας SMT μπορεί εκμεταλλευτεί την επαλληλία σε επίπεδο εντολών(ILP) ενός νήματος εκδίδοντας πολλαπλές εντολές ανά κύκλο ˙ όμοια με τους multithreaded, μπορεί να καλύψει διεργασίες με μεγάλη καθυστέρηση εκτελώντας εντολές από άλλα νήματα. Η διαφορά του SMT είναι ότι μπορεί να κάνει και τα δύο ταυτόχρονα, δηλαδή στον ίδιο κύκλο. Σε κάθε κύκλο ένας SMT επεξεργαστής επιλέγει εντολές για εκτέλεση από όλα τα ενεργά νήματα. Ο επεξεργαστής χρονοδρομολογεί δυναμικά τα νήματα σε όλους τους υλικούς

πόρους του, παρέχοντας έτσι την καλύτερη πιθανότητα για την υψηλότερη αξιοποίηση του υλικού. Αν ένα νήμα έχει υψηλό επίπεδο επαλληλίας εντολών, τότε αυτός ο τύπος επαλληλίας ικανοποιείται · εάν τρέχουν πολλά νήματα, με το καθένα να έχει χαμηλό επίπεδο επαλληλίας εντολών, μπορούν να εκτελούνται μαζί, εξισορροπώντας έτσι το χαμηλό επίπεδο ILP του κάθε νήματος. Συνεπώς, η αρχιτεκτονική simultaneous multithreading έχει τη δυνατότητα να επαναφέρει θυρίδες εκτέλεσης που χάθηκαν λόγω οριζόντιας *και* κατακόρυφης σπατάλης.

Το αποτέλεσμα που προκύπτει από την SMT αρχιτεκτονική είναι η καλύτερη απόδοση για μια ποικιλία από εφαρμογές. Εάν πρόκειται για ένα σύνολο από ανεξάρτητα προγράμματα, υπάρχει βελτίωση του συνολικού throughput του μηχανήματος. Όταν ένα πρόγραμμα δεν έχει εντολές έτοιμες για εκτέλεση, εντολές μπορούν να βρεθούν σε ένα από τα άλλα προγράμματα. Αντίστοιχα, προγράμματα που είναι παραλληλοποιήσημα, είτε από τον compiler είτε από τον προγραμματιστή, έχουν τα ίδια κέρδη σε throughput · αλλά εδώ το αποτέλεσμα είναι *μείωση* του χρόνου εκτέλεσης για την εφαρμογή. Τέλος προγράμματα που πρέπει να εκτελεστούν σαν ένα και μόνο νήμα, έχουν όλους τους πόρους μηχανής στη διάθεσή τους και πετυχαίνουν περίπου τα ίδια επίπεδα απόδοσης όπως όταν εκτελούνται σε ένα μονο-νηματικό επεξεργαστή.

2.2.1 Ένα μοντέλο υλοποίησης της αρχιτεκτονικής SMT

Η αρχιτεκτονική SMT όπως προτείνεται στο [4], μπορεί να εκτελέσει εντολές μέχρι και από οκτώ νήματα και αποτελεί επέκταση ενός υπερβαθμωτού επεξεργαστή στον οποίο προστίθενται πολλοί από τους αναγκαίους και ήδη υπάρχοντες μηχανισμούς για την εκμετάλλευση της TLP μέσω πολυνημάτωσης. Για την υποστήριξη του SMT χρειάζονται βασικά δύο αλλαγές στον υπερβαθμωτό επεξεργαστή: στο μηχανισμό εύρεσης εντολών (instruction fetch mechanism) και στο σύνολο των καταχωρητών.

Πολύ σημαντικό είναι το γεγονός ότι δε χρειάζεται ειδικό υλικό για την χρονοδρομολόγηση εντολών από διαφορετικά νήματα στις λειτουργικές μονάδες. Συμβατικό υλικό δυναμικής χρονοδρομολόγησης που υπάρχει στους σημερινούς out-of-order superscalars μπορεί να παράγει πολυνηματική χρονοδρομολόγηση. Το κύκλωμα μετονομασίας καταχωρητών αφαιρεί δια-νηματικές διενέξεις ονομάτων καταχωρητών, με την αντιστοίχηση των ανά νήμα συγκεκριμένων καταχωρητών αρχιτεκτονικής πάνω στους καταχωρητές υλικού. Εντολές από όλα τα νήματα τοποθετούνται μαζί στις ουρές εντολών. Όταν οι τελεστές τους γίνουν διαθέσιμοι, προωθούνται στις λειτουργικές μονάδες, χωρίς να έχουν αναφορά από ποιο νήμα προήλθαν.

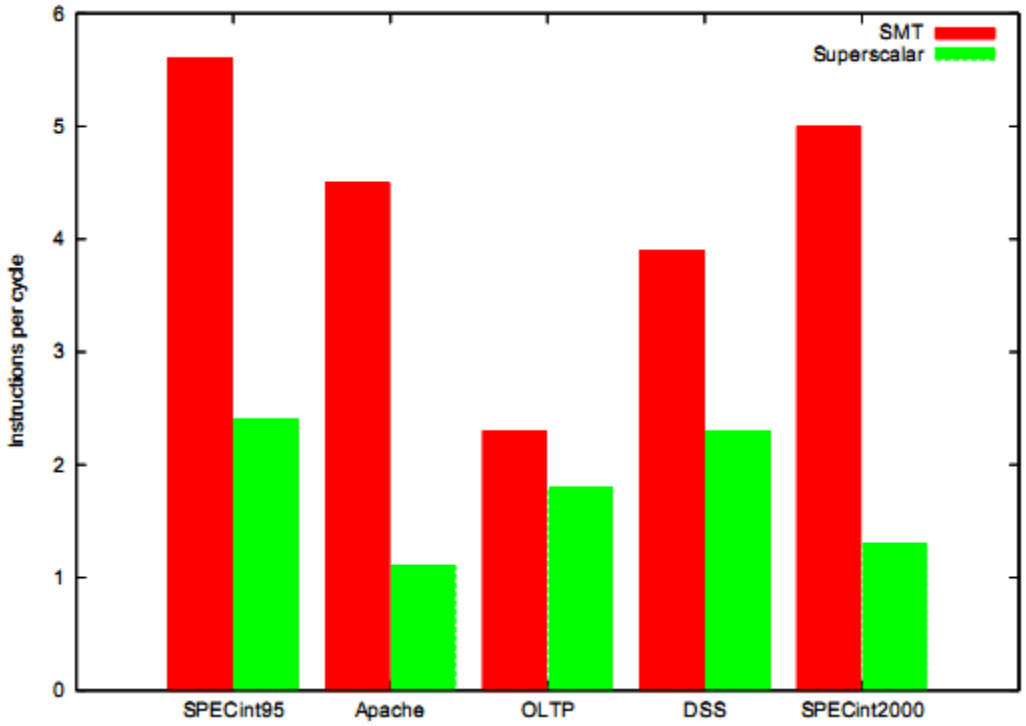
Αυτή η ελάχιστη επανασχεδίαση έχει δύο σημαντικές συνέπειες. Πρώτον, αφού οι περισσότεροι πόροι είναι διαθέσιμοι σε ένα νήμα που εκτελείται μόνο του, το SMT παρέχει

καλή μονο-νηματική επίδοση. Δεύτερον, επειδή οι αλλαγές για την εφαρμογή του SMT είναι ελάχιστες, η εμπορική μετάβαση από τους σημερινούς superscalars σε SMT πρέπει να είναι αρκετά ομαλή.

Ένα συνηθισμένο σύστημα πρόβλεψης διακλαδώσεων οδηγεί στην εύρεση οδηγεί στην εύρεση εντολών, μόνο που για να μπορεί ο επεξεργαστής να ανακτά εντολές από 8 νήματα, διαθέτει 8 μετρητές προγράμματος και 8 δείκτες στοίβας, ένα για κάθε νήμα. Σε κάθε κύκλο, ο μηχανισμός εύρεσης εντολών επιλέγει δύο νήματα (από τα νήματα για τα οποία δεν εκκρεμεί κάποια αστοχία στην κρυφή μνήμη εντολών) και φορτώνει μέχρι και 4 εντολές από κάθε νήμα. Ο μηχανισμός αυτός είναι ισοδύναμος με αυτό ενός υπερβαθμωτού επεξεργαστή με πλάτος εντολών εκτέλεσης 8, και χρειάζεται μόνο 2 θύρες στην κρυφή μνήμη εντολών. Για την αποδοτική λειτουργία του μηχανισμού εύρεσης εντολών, ανατίθενται προτεραιότητες στα νήματα. Η τεχνική που εφαρμόζεται, αναθέτει την υψηλότερη προτεραιότητα στη νήματα τα οποία έχουν το μικρότερο πλήθος εντολών στα εξής στάδια: αποκωδικοποίηση εντολών, μετονομασία καταχωρητών, ουρά εντολών προς εκτέλεση. Με αυτό τον τρόπο αποφεύγεται η λιμοκτονία ενός νήματος ή η αποκλειστική χρήση του επεξεργαστή από ένα νήμα, επιτυγχάνοντας ομοιόμορφη κατανομή των εντολών ανάμεσα στα νήματα.

Μετά την εύρεση και αποκωδικοποίηση των εντολών ακολουθεί η *μετονομασία καταχωρητών* (register renaming). Σε κάθε νήμα διατίθενται 32 αρχιτεκτονική καταχωρητές, οι οποίοι αντιστοιχίζονται μέσω της διαδικασίας μετονομασίας καταχωρητών στους φυσικούς καταχωρητές, οι οποίοι έχουν ίδιο πλήθος με το σύνολο των αρχιτεκτονικών καταχωρητών (8x32=256) συν κάποιους επιπλέον καταχωρητές που χρειάζονται κατά τη λειτουργία μετονομασίας καταχωρητών. Επειδή, το μεγαλύτερο σύνολο καταχωρητών οδηγεί σε περισσότερο χρόνο προσπέλασης, τα στάδια ανάγνωσης και εγγραφής καταχωρητών επιμηκύνονται κατά δύο κύκλους μηχανής για την αποφυγή της αύξησης στη διάρκεια του κύκλου μηχανής.

Επιπρόσθετα του νέου μηχανισμού εύρεσης εντολών, του μεγαλύτερου συνόλου καταχωρητών και της μακρύτερης διασωλήνωσης, το μοντέλο αρχιτεκτονικής του SMT χρειάζεται και τον πολλαπλασιασμό τριών πόρων του επεξεργαστή: του μηχανισμού αποδέσμευσης εντολών (instruction retire) για κάθε νήμα, του μηχανισμού διακοπών, και ένα πρόσθετο πεδίο στις εγγραφές του buffer πρόβλεψης διακλαδώσεων (brunch prediction) το οποίο δηλώνει την ταυτότητα του νήματος. Αυτό συμβαίνει, διότι η διαδικασία μετονομασίας καταχωρητών εξαλείφει τις οποιεσδήποτε εξαρτήσεις καταχωρητών μεταξύ νημάτων, έτσι ώστε οι συνηθισμένες ουρές εντολών να μπορούν να χρησιμοποιηθούν για τη δυναμική δρομολόγηση εντολών από διαφορετικά νήματα. Οι εντολές των νημάτων περιμένουν μέσα στις ουρές, και μία εντολή μπορεί να εκτελεστεί μόλις τα ορίσματά της γίνουν διαθέσιμα.



Σχήμα 2.2: Σύγκριση επίδοσης μεταξύ ενός υπερβαθμωτού με ένα SMT επεξεργαστή

Στο μοντέλο αυτό, οι περισσότεροι πόροι του επεξεργαστή μοιράζονται δυναμικά ανάμεσα στα νήματα, ενώ ελάχιστοι πόροι μοιράζονται στατικά. Επομένως στην περίπτωση που εκτελείται ένα μόνο νήμα, οι πόροι είναι σχεδόν όλοι διαθέσιμοι. Έτσι το μοντέλο αυτό αρχιτεκτονικής SMT επιτρέπει να πετυχαίνουμε την αυξημένη επίδοση που παρουσιάζει το simultaneous multithreading, χωρίς να επηρεάζεται η επίδοση του σύγχρονου υπερβαθμωτού επεξεργαστή (και στην περίπτωση εκτέλεσης ενός μόνο νήματος) στον οποίο βασίστηκε ο σχεδιασμός. Μία θεωρητική γραφική πρόβλεψης της επίδοσης του συστήματος SMT σε σχέση με τον αντίστοιχο υπερβαθμωτό επεξεργαστή δίνεται στο Σχήμα 2.2.

2.3 Τεχνολογία Hyper-Threading

Η τεχνολογία Hyper-Threading™ αποτελεί την πρώτη εμπορική υλοποίηση της τεχνολογίας του simultaneous multithreading (SMT) στην Αρχιτεκτονική της Intel®. Η τεχνολογία Hyper-Threading κάνει ένα φυσικό επεξεργαστή να φαίνεται σαν δύο λογικοί επεξεργαστές · οι φυσικοί πόροι εκτέλεσης είναι μοιραζόμενοι ενώ το αρχιτεκτονικό state αντιγράφεται για τους δύο λογικούς επεξεργαστές. Από την πλευρά του λογισμικού και της αρχιτεκτονικής, αυτό σημαίνει ότι τα λειτουργικά συστήματα και οι εφαρμογές χρήστη μπορούν δρομολογούν

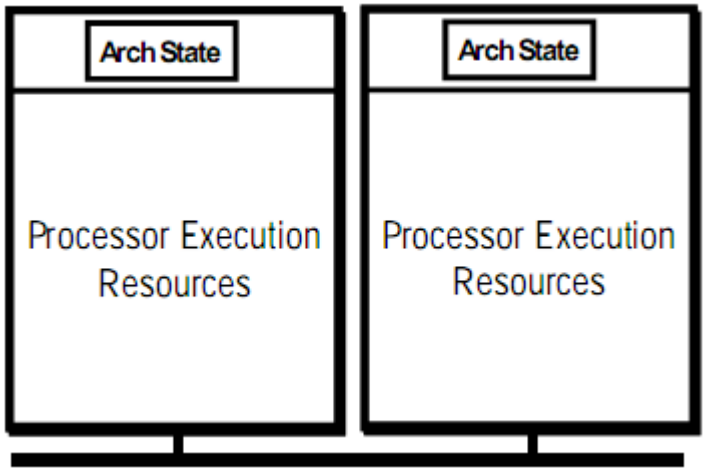
™ Hyper-Threading είναι σήμα κατατεθέν της Intel Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες

® Intel είναι σήμα κατατεθέν της Intel Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες

διεργασίες ή νήματα όπως θα έκαναν σε πολλαπλούς φυσικούς επεξεργαστές. Από μικροαρχιτεκτονική σκοπιά, αυτό μεταφράζεται σε ύπαρξη και εκτέλεση εντολών και από τους δύο λογικούς επεξεργαστές σε μοιραζόμενους υπολογιστικούς πόρους. Στη συνέχεια κάνουμε μία εκτενή παρουσίαση της αρχιτεκτονικής του Hyper-Threading [5], η βαθιά κατανόηση της οποίας είναι απαραίτητη για την απόκτηση της μέγιστης επίδοσης από ένα σύστημα με αυτή την τεχνολογία.

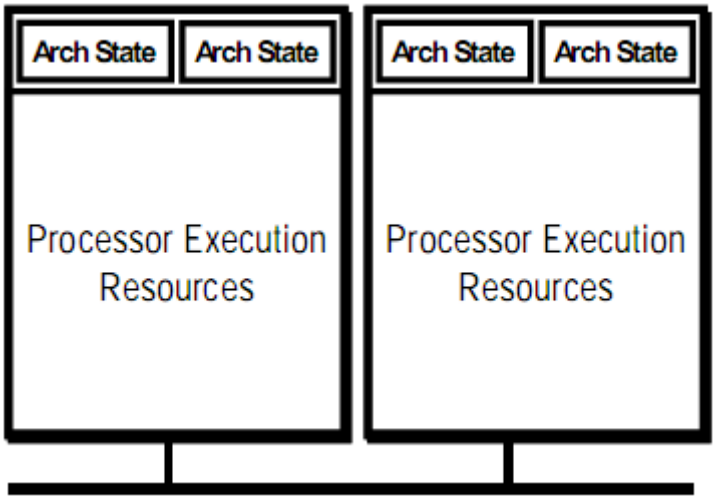
2.3.1 Αρχιτεκτονική του Hyper-Threading

Η τεχνολογία Hyper-Threading κάνει ένα φυσικό επεξεργαστή να εμφανίζεται σαν πολλαπλοί λογικοί επεξεργαστές ([6],[7]). Για να είναι κάτι τέτοιο εφικτό, υπάρχει ένα αντίγραφο της αρχιτεκτονικής κατάστασης για κάθε λογικό επεξεργαστή, και οι λογικοί επεξεργαστές μοιράζονται ένα μοναδιαίο σύνολο από φυσικές μονάδες εκτέλεσης. Από προγραμματιστικής ή αρχιτεκτονικής σκοπιάς, αυτό μεταφράζεται στο ότι λειτουργικά συστήματα και προγράμματα χρήστη μπορούν να δρομολογούν διεργασίες ή νήματα στους λογικούς επεξεργαστές όπως θα έκαναν σε συμβατικούς φυσικούς επεξεργαστές σε ένα πολυεπεξεργαστικό σύστημα. Από μικροαρχιτεκτονικής πλευράς, αυτό σημαίνει ότι οι εντολές από λογικούς επεξεργαστές θα υφίστανται και θα εκτελούνται ταυτόχρονα σε κοινές μονάδες εκτέλεσης.



Σχήμα 2.3: Επεξεργαστές χωρίς τεχνολογία Hyper-Threading

Σαν ένα παράδειγμα, στο Σχήμα 2.3 παρουσιάζεται ένα πολυεπεξεργαστικό σύστημα με δύο φυσικούς επεξεργαστές οι οποίοι δεν έχουν τη δυνατότητα του Hyper-Threading. Στο Σχήμα 2.4, από την άλλη, παρουσιάζεται ένα πολυεπεξεργαστικό σύστημα με δύο φυσικούς επεξεργαστές που έχουν τη δυνατότητα του Hyper-Threading. Με δύο αντίγραφα της αρχιτεκτονικής κατάστασης σε κάθε φυσικό επεξεργαστή, το σύστημα φαίνεται να έχει τέσσερις λογικούς επεξεργαστές.



Σχήμα 2.4: Επεξεργαστές με τεχνολογία Hyper-Threading

Η πρώτη υλοποίηση της τεχνολογίας Hyper-Threading γίνεται διαθέσιμη στην οικογένεια επεξεργαστών Intel Xeon για διπύρηνους και πολυεπεξεργαστικούς διακομιστές, με δύο λογικούς επεξεργαστές ανά φυσικό επεξεργαστή. Με την πιο αποδοτική χρήση των υπαρχόντων υπολογιστικών πόρων, η οικογένεια επεξεργαστών Intel Xeon μπορεί να αυξήσει σημαντικά την επίδοση χωρίς πρακτικά να υπάρχει αύξηση κόστους. Αυτή η υλοποίηση του Hyper-Threading προσθέτει κάτω από 5% στο σχετικό μέγεθος ψηφίδας και στις ανάγκες ισχύος, αλλά μπορεί να παρέχει κέρδη στην απόδοση που είναι πολύ μεγαλύτερα του 5%.

Ο κάθε λογικός επεξεργαστής διατηρεί ένα πλήρες σύνολο της αρχιτεκτονικής κατάστασης. Η αρχιτεκτονική κατάσταση αποτελείται από καταχωρητές που συμπεριλαμβάνουν καταχωρητές γενικού σκοπού, καταχωρητές ελέγχου, καταχωρητές advanced programmable interrupt controller (APIC), και μερικούς καταχωρητές μηχανικής κατάστασης. Από προγραμματιστικής πλευράς, από τη στιγμή που η αρχιτεκτονική κατάσταση έχει αντιγραφεί, ο επεξεργαστής εμφανίζεται σαν δύο επεξεργαστές. Ο αριθμός των τρανζίστορ που χρειάζονται για να αποθηκευτεί η αρχιτεκτονική κατάσταση είναι ένα εξαιρετικά μικρό ποσοστό του συνόλου των τρανζίστορ του επεξεργαστή. Οι λογικοί επεξεργαστές μοιράζονται σχεδόν όλους τους άλλους πόρους του φυσικού επεξεργαστή, όπως είναι οι κρυφές μνήμες, οι μονάδες εκτέλεσης, το κύκλωμα πρόβλεψης διακλαδώσεων, το κύκλωμα ελέγχου (control logic) και τους διαύλους.

Ο καθένας από τους λογικούς επεξεργαστές έχει τον δικό του ελεγκτή διακοπών ή APIC. Η διακοπές που αποστέλλονται σε ένα συγκεκριμένο λογικό επεξεργαστή χειρίζονται μόνο από αυτόν το λογικό επεξεργαστή.

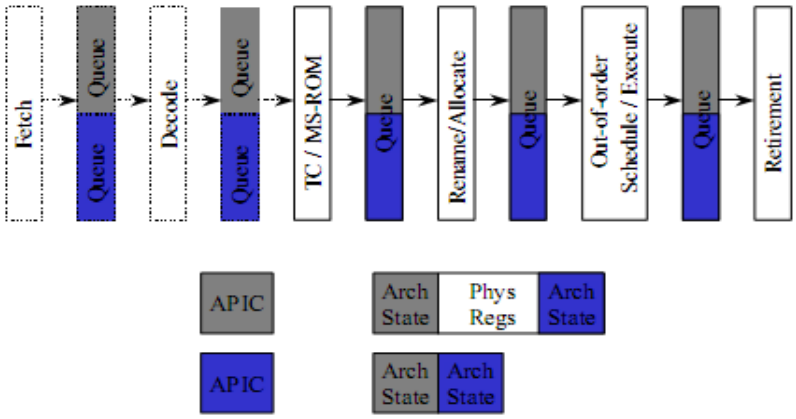
2.3.2 Μια πρώτη υλοποίηση στη οικογένεια επεξεργαστών Intel Xeon

Διάφοροι στόχοι βρίσκονταν στο επίκεντρο των επιλογών μικροαρχιτεκτονικής σχεδίασης που έγιναν για την υλοποίηση της τεχνολογίας Hyper-Threading στην οικογένεια επεξεργαστών Intel Xeon. Ένας τέτοιος στόχος ήταν η ελαχιστοποίηση του κόστους σε μέγεθος ψηφίδας που θα προκαλούσε η υλοποίηση αυτής της τεχνολογίας. Εφόσον οι λογικοί επεξεργαστές μοιράζονται τη μεγάλη πλειονότητα των μικροαρχιτεκτονικών πόρων και μόνο μερικές δομές χρειάστηκε να αντιγραφούν, το συνολικό κόστος σε μέγεθος ψηφίδας της πρώτης αυτής υλοποίησης ανέρχεται σε λιγότερο από το 5% της συνολικής ψηφίδας του φυσικού επεξεργαστή.

Ένας δεύτερος στόχος ήταν η εξασφάλιση του ότι σε περίπτωση που ο ένας λογικός επεξεργαστής μπλοκάρει τότε ο άλλος λογικός επεξεργαστής δε θα επηρεαστεί και θα είναι σε θέση να συνεχίσει τη δουλειά του. Ένας λογικός επεξεργαστής μπορεί να μπλοκάρει για μια πληθώρα από λόγους, όπως είναι η εξυπηρέτηση αστοχιών κρυφής μνήμης, ο χειρισμός λανθασμένων προβλέψεων διακλάδωσης, ή η αναμονή για εκτέλεση για τα αποτελέσματα εκτέλεσης προηγούμενων εντολών. Η ανεξάρτητη λειτουργία του ενός λογικού επεξεργαστή όταν ο άλλος έχει μπλοκάρει εξασφαλίστηκε με το να γίνεται η διαχείριση των ουρών buffering με τέτοιο τρόπο ώστε κανένας από τους δύο λογικούς επεξεργαστές να μην μπορεί να χρησιμοποιήσει όλες τις καταχωρήσεις όταν δύο ενεργά νήματα[2] βρίσκονταν σε εκτέλεση. Αυτό επιτυγχάνεται είτε περιορίζοντας είτε διαμερίζοντας τον αριθμό των ενεργών καταχωρήσεων που μπορεί το κάθε νήμα να έχει στις ουρές buffering.

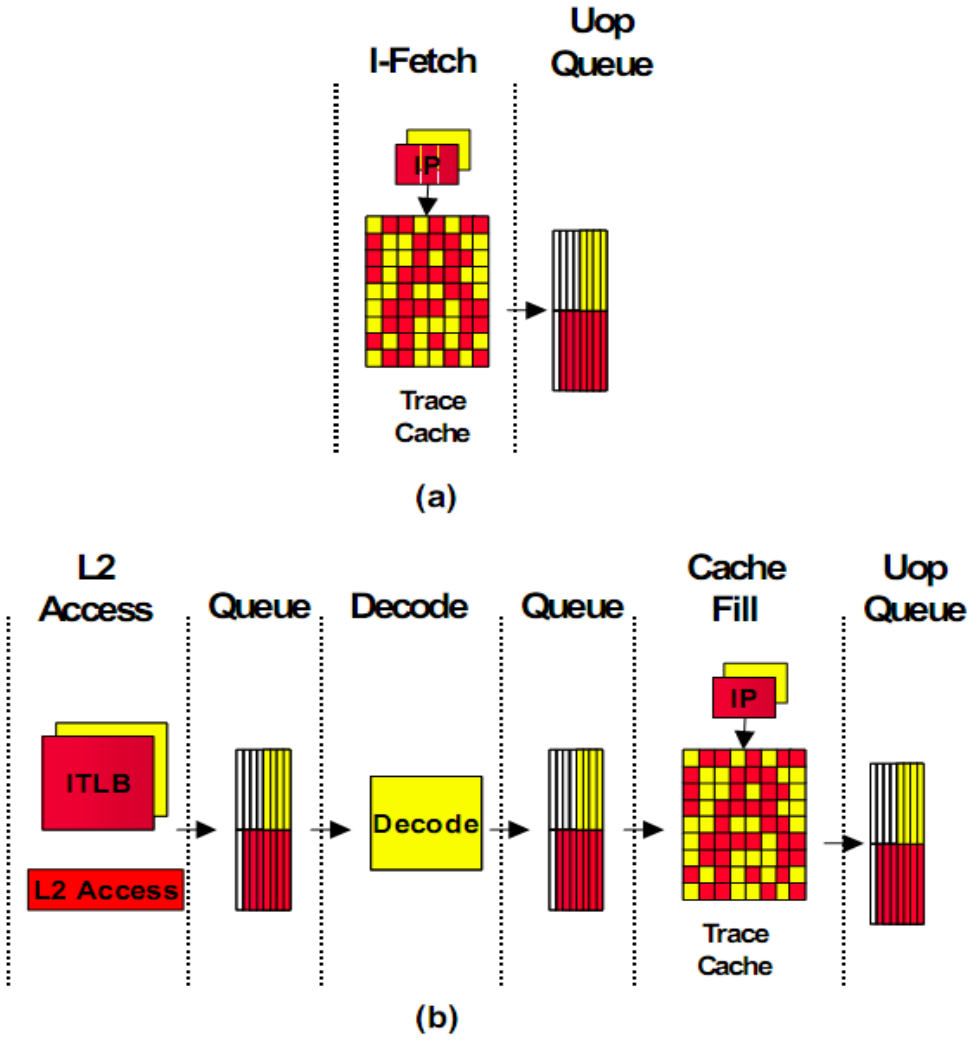
Ένας τρίτος στόχος ήταν να μπορεί ένας επεξεργαστής που έχει την τεχνολογία Hyper-Threading να εκτελεί ένα νήμα που εκτελείται μόνο του με την ίδια ταχύτητα που το εκτελεί ένας επεξεργαστής που δεν έχει αυτή την τεχνολογία. Αυτό σημαίνει ότι η διαμέριση των πόρων μεταξύ των λογικών επεξεργαστών πρέπει να καταργείται όταν μόνο ένα νήμα είναι ενεργό. Μία high-level άποψη της μικροαρχιτεκτονικής διασωλήνωσης δίνεται στο Σχήμα 2.5. Όπως παρουσιάζεται, οι ουρές buffering διαχωρίζουν τα κύρια λογικά μπλοκ της διασωλήνωσης. Οι ουρές buffering είναι είτε διαμερίζονται είτε αντιγράφονται για να εξασφαλίσουν ανεξάρτητη πρόοδο μέσα από κάθε λογικό μπλοκ.

[2] Στα ενεργά νήματα περιλαμβάνονται ο άεργος βρόχος του λειτουργικού συστήματος επειδή τρέχει μία ακολουθία κώδικα που διαρκώς ελέγχει τις ουρές διεργασιών. Ο άεργος βρόχος του λειτουργικού συστήματος μπορεί να καταναλώνει σημαντικούς υπολογιστικούς πόρους.



Σχήμα 2.5: Η διασωλήνωση στον επεξεργαστή Intel Xeon

Στις ακόλουθες παραγράφους θα συζητήσουμε για τη διασωλήνωση, για την υλοποίηση των κύριων λειτουργιών, και θα περιγράψουμε με λεπτομέρεια τους διάφορους τρόπους με τους οποίους μοιράζονται ή αντιγράφονται οι υπολογιστικοί πόροι.



Σχήμα 2.6: Λεπτομέρεια front-end pipeline (a) Trace Cache Hit (b) Trace Cache Miss

2.3.2.1 Τμήμα Front End

Το front end της διασωλήνωσης είναι υπεύθυνο για να μεταφέρει εντολές σε μεταγενέστερα στάδια του pipe. Όπως παρουσιάζεται στο Σχήμα 1.6a, οι εντολές γενικά προέρχονται από το Execution Trace Cache (TC), το οποίο βρίσκεται βασικά στην κρυφή μνήμη εντολών επιπέδου 1. Στο Σχήμα 2.6b φαίνεται ότι μόνο όταν υπάρχει μία αστοχία TC η μηχανή φορτώνει και αποκωδικοποιεί εντολές από την κρυφή μνήμη επιπέδου 2. Κοντά στο TC βρίσκεται η ROM Μικροκώδικα, η οποία αποθηκεύει αποκωδικοποιημένες εντολές για τις μακρύτερες και πιο περίπλοκες εντολές IA-32.

2.3.2.2 Execution Trace Cache (TC)

Στο TC αποθηκεύονται αποκωδικοποιημένες εντολές, ονομαζόμενες μικρό-εντολές ή "uops". Οι πλείστες εντολές σε ένα πρόγραμμα φορτώνονται και εκτελούνται από το TC. Δύο σύνολα από δείκτες-επόμενης-εντολής ανεξάρτητα παρακολουθούν την πρόοδο από τα δύο προγραμματιστικά νήματα που βρίσκονται σε εκτέλεση. Οι δύο λογικοί επεξεργαστές διαιτητεύουν την πρόσβαση στο TC σε κάθε κύκλο ρολογιού. Εάν και οι δύο λογικοί επεξεργαστές θέλουν να έχουν πρόσβαση στο TC την ίδια στιγμή, η πρόσβαση παραχωρείται σε ένα από αυτούς και μετά στον άλλον σε εναλλακτικούς κύκλους ρολογιού. Για παράδειγμα, εάν ένας κύκλος χρησιμοποιείται για το φόρτωμα μιας γραμμής κρυφής μνήμης για τον ένα λογικό επεξεργαστή, ο επόμενος κύκλος θα χρησιμοποιηθεί για το φόρτωμα γραμμής για τον άλλο λογικό επεξεργαστή, δεδομένου ότι και οι δύο λογικοί επεξεργαστές αιτήθηκαν πρόσβαση στο trace cache. Εάν ένας λογικός επεξεργαστής έχει μπλοκάρει ή δεν μπορεί να χρησιμοποιήσει το TC, τότε ο άλλος λογικός επεξεργαστής μπορεί να χρησιμοποιήσει όλο το εύρος ζώνης του trace cache, για κάθε κύκλο.

Οι καταχωρήσεις του TC έχουν ετικέτα με πληροφορία για το νήμα στο οποίο ανήκουν και κατανέμονται δυναμικά ανάλογα με τις ανάγκες. Το TC είναι 8-way set associative, και οι καταχωρήσεις αντικαθίστανται με βάση το λιγότερο-πρόσφατα-χρησιμοποιούμενο (least-recently-used LRU) αλγόριθμο που χρησιμοποιεί και τα 8 ways. Η μοιραζόμενη φύση του TC επιτρέπει στον ένα λογικό επεξεργαστή να έχει περισσότερες καταχωρήσεις από τον άλλο εάν είναι απαραίτητο.

2.3.2.3 Microcode ROM

Όταν εμφανιστεί μία περίπλοκη εντολή, το TC στέλνει ένα δείκτη εντολής-μικροκώδικα στη ROM Μικροκώδικα. Στην συνέχεια ο ελεγκτής της ROM φορτώνει τις απαραίτητες uops και επιστρέφει τον έλεγχο στο TC. Δύο δείκτες εντολών μικροκώδικα χρησιμοποιούνται για να

ελέγχουν τις ροές ανεξάρτητα σε περίπτωση που και οι δύο λογικοί επεξεργαστές εκτελούν περίπλοκες εντολές IA-32.

Οι καταχωρήσεις της ROM Μικροκώδικα είναι μοιραζόμενες μεταξύ των δύο λογικών επεξεργαστών. Η πρόσβαση σε αυτήν γίνεται εναλλακτικά μεταξύ των λογικών επεξεργαστών ακριβώς όπως γίνεται στην περίπτωση του TC.

2.3.2.4 Πρόβλεψη Διακλάδωσης και ITLB

Εάν υπάρξει αστοχία TC, τα bytes των εντολών πρέπει να φορτωθούν από την κρυφή μνήμη L2 και αποκωδικοποιούνται σε uops πριν τοποθετηθούν στο TC. Το Instruction Translation Lookaside Buffer (ITLB) λαμβάνει αιτήσεις από το TC για την παροχή νέων εντολών, και μεταφράζει την διεύθυνση του δείκτη επόμενης εντολής σε μία φυσική διεύθυνση. Μία αίτηση αποστέλλεται στην κρυφή μνήμη L2 και τα bytes των εντολών επιστρέφονται. Αυτά τα bytes τοποθετούνται σε streaming buffers, τα οποία κρατούν τα bytes μέχρι να αποκωδικοποιηθούν.

Τα ITLBs υπάρχουν σε δύο αντίτυπα. Κάθε λογικός επεξεργαστής έχει το δικό του ITLB και το δικό του σύνολο από δείκτες εντολών για να παρακολουθεί την πρόοδο της φόρτωσης εντολών για τους δύο λογικούς επεξεργαστές. Το κύκλωμα που φορτώνει εντολές και είναι υπεύθυνο για την αποστολή αιτημάτων στην κρυφή μνήμη L2 διαιτητεύει με βάση τη λογική first-come first-serve, διατηρώντας όμως τουλάχιστον μία θυρίδα αίτησης για κάθε λογικό επεξεργαστή. Με αυτό τον τρόπο, και οι δύο λογικοί επεξεργαστές μπορούν να έχουν συναλλαγές φόρτωσης εντολών που να εκκρεμούν, ταυτόχρονα.

Σε κάθε λογικό επεξεργαστή αντιστοιχεί ένα σύνολο από δύο streaming buffers των 64-byte για την αποθήκευση bytes εντολών σε προετοιμασία του σταδίου αποκωδικοποίησης εντολών. Τα ITLBs και τα streaming buffers είναι μικρές δομές, για αυτό και το κόστος σε μέγεθος ψηφίδας από την αντιγραφή αυτών των δομών είναι πολύ περιορισμένο.

Οι δομές που χρειάζονται για την πρόβλεψη διακλάδωσης είτε αντιγράφονται είτε μοιράζονται. Το return stack buffer, που προβλέπει τον τελικό προορισμό των εντολών return, αντιγράφεται εις διπλούν επειδή αποτελεί μία πολύ μικρή δομή και επειδή είναι πιο αποδοτικό υπάρχει πρόβλεψη των ζευγών call/return για κάθε νήμα ανεξάρτητα. Το buffer που κρατά το ιστορικό διακλάδωσης που χρησιμοποιείται για να δούμε τον ολικό ιστορικό πίνακα είναι επίσης ανεξάρτητο για κάθε λογικό επεξεργαστή. Ο μεγάλος πίνακας γενικού ιστορικού, όμως, είναι μία μοιραζόμενη δομή με καταχωρήσεις που έχουν ετικέτα με το ID ενός λογικού επεξεργαστή.

2.3.2.5 IA-32 Instruction Decode

Οι εντολές IA-32 είναι δύσκολες στην αποκωδικοποίηση επειδή αποτελούνται από μεταβλητό αριθμό από bytes και παρέχουν πολλές διαφορετικές επιλογές. Ένα σημαντικό μέγεθος κυκλωματική λογικής και κυκλώματος ενδιάμεσης κατάστασης είναι απαραίτητο για την αποκωδικοποίηση αυτών των εντολών. Ευτυχώς, όμως, το TC παρέχει τις περισσότερες από τις uops, και η αποκωδικοποίηση είναι απαραίτητη μόνο για εντολές που αστοχούν το TC.

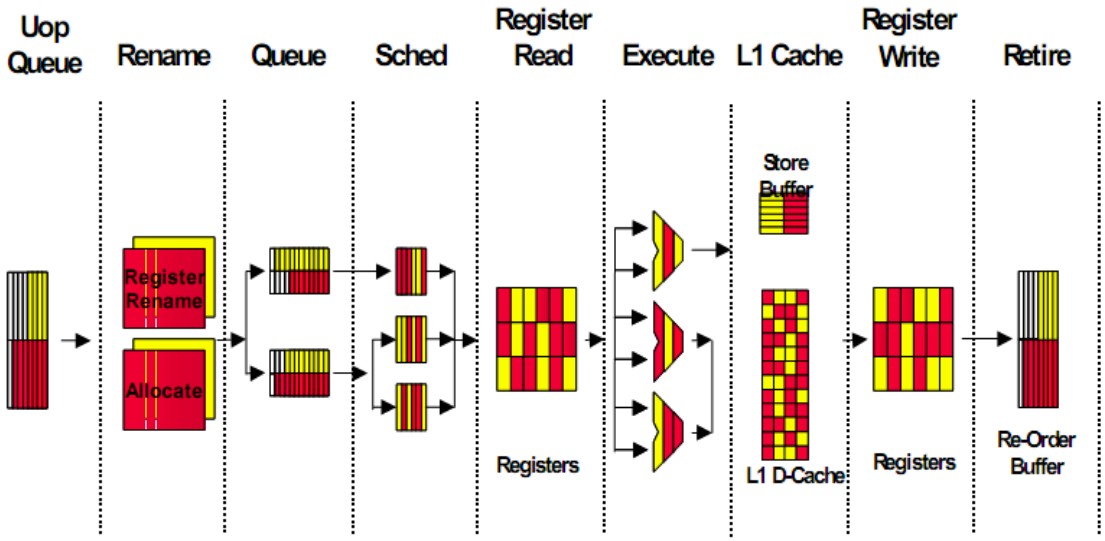
Το κύκλωμα αποκωδικοποίησης δέχεται bytes εντολών από τα streaming buffers και τα αποκωδικοποιεί σε uops. Όταν και τα δύο νήματα αποκωδικοποιούν εντολές ταυτόχρονα, τα streaming buffers εναλλάσσονται μεταξύ των νημάτων ώστε και τα δύο νήματα να μοιράζονται το ίδιο κύκλωμα αποκωδικοποίησης. Το κύκλωμα αποκωδικοποίησης χρειάζεται να κρατά δύο αντίτυπα όλων των καταστάσεων που απαιτούνται για την αποκωδικοποίηση των εντολών IA-32, ένα για κάθε λογικό επεξεργαστή, παρόλο που αποκωδικοποιεί εντολές για ένα λογικό επεξεργαστή τη φορά. Γενικά, γίνεται αποκωδικοποίηση πολλών εντολών για τον ένα λογικό επεξεργαστή πριν δώσει τον έλεγχο στον άλλο λογικό επεξεργαστή. Η απόφαση για πιο χοντροκομμένη διαμέριση στην εναλλαγή μεταξύ των δύο λογικών επεξεργαστών έγινε με γνώμονα τη μείωση του κόστους σε μέγεθος ψηφίδας και τη μείωση της πολυπλοκότητας. Βέβαια, στην περίπτωση που μόνο ένας λογικός επεξεργαστής χρειάζεται το κύκλωμα αποκωδικοποίησης, το πλήρες εύρος ζώνης αποκωδικοποίησης αφιερώνεται σε αυτόν. Οι εντολές που έχουν αποκωδικοποιηθεί γράφονται στο TC και προωθούνται, ακολούθως, στην ουρά uop.

2.3.2.6 Ουρά Uop

Μετά τη φόρτωση των uops από το trace cache ή από τη ROM Μικροκώδικα, ή από προώθηση από το κύκλωμα αποκωδικοποίησης, τοποθετούνται στην “ουρά uop”. Αυτή η ουρά διαχωρίζει το Front End από την out-of-order Μηχανή Εκτέλεσης στην ροή διασωλήνωσης. Η ουρά uop έχει διαμεριστεί με τέτοιο τρόπο ώστε κάθε λογικός επεξεργαστής να έχει μισές από τις καταχωρήσεις. Αυτή η διαμέριση επιτρέπει και στους δύο λογικούς επεξεργαστές να έχουν ανεξάρτητη πρόοδο χωρίς να επηρεάζονται από καθυστερήσεις του front-end (π.χ., αστοχίες TC) ή καθυστερήσεις εκτέλεσης.

2.3.3.7 Μηχανή εκτέλεσης Out-Of-Order

Η μηχανή εκτέλεσης out-of-order αποτελείται από λειτουργίες παραχώρησης, μετονομασίας καταχωρητών, χρονοδρομολόγησης και εκτέλεσης, όπως παρουσιάζεται στο Σχήμα 2.7. Αυτό το μέρος της μηχανής αναδιατάσσει εντολές και τις εκτελεί μόλις οι είσοδοί (τελεστές) τους είναι έτοιμοι, χωρίς να κρατάει την αρχική σειρά του προγράμματος.



Σχήμα 2.7: Λεπτομέρεια της διασωλήνωσης της μηχανής εκτέλεσης out-of-order

2.3.3.7.1 Allocator

Η μηχανή εκτέλεσης out-of-order έχει διάφορους buffers για να μπορεί να εκτελεί τις λειτουργίες αναδιάταξης, tracing και sequencing. Το κύκλωμα allocator παίρνει uops από την ουρά uop και παραχωρεί πολλά από τα σημαντικά buffers μηχανής που χρειάζονται για την εκτέλεση της κάθε uop, συμπεριλαμβανομένου των 126 καταχωρήσεων του re-order buffer, τους 128 φυσικούς καταχωρητές για ακέραιους και τους 128 κινητής υποδιαστολής, της 48 καταχωρήσεις του load buffer και τις 24 καταχωρήσεις του store buffer. Κάποιοι από αυτούς τους σημαντικούς buffers διαμερίζονται με τέτοιο τρόπο ώστε ο κάθε λογικός επεξεργαστής να μπορεί να χρησιμοποιεί το πολύ τις μισές καταχωρήσεις.

Συγκεκριμένα, κάθε λογικός επεξεργαστής μπορεί να χρησιμοποιήσει μέχρι 63 καταχωρήσεις του re-order buffer, 24 load buffers, και 12 καταχωρήσεις του store buffer.

Εάν υπάρχουν uops και για τους δύο λογικούς επεξεργαστές στην ουρά uop, ο allocator θα εναλλάσσει την επιλογή uops από τον ένα λογικό επεξεργαστή στον άλλο σε κάθε κύκλο ρολογιού για την παραχώρηση πόρων. Εάν ένας λογικός επεξεργαστής έχει χρησιμοποιήσει το μέγιστο ενός αναγκαίου πόρου, όπως είναι οι καταχωρήσεις του store buffer, ο allocator σηματοδοτήσει το σήμα “stall” για αυτόν το λογικό επεξεργαστή και θα συνεχίσει να αναθέτει πόρους στον άλλο λογικό επεξεργαστή. Επιπλέον, εάν η ουρά uop περιέχει μόνο uops για τον ένα λογικό επεξεργαστή μόνο, ο allocator θα προσπαθήσει να αναθέτει τους πόρους για αυτόν το λογικό επεξεργαστή σε κάθε κύκλο για να βελτιστοποιήσει το εύρος ζώνης ανάθεσης, παρόλο που τα όρια στο πλήθος των πόρων που μπορεί να αποκτήσει εξακολουθούν να υφίστανται.

Με το να περιορίζεται ο μέγιστος αριθμός των κρίσιμων buffer που μπορούν να χρησιμοποιηθούν, η μηχανή βοηθά στην επιβολή δίκαιης κατανομής πόρων και παρεμποδίζει τη δημιουργία αδιεξόδων (deadlocks).

2.3.3.7.2 Μετονομασία Καταχωρητών

Το κύκλωμα μετονομασίας καταχωρητών μετονομάζει (αντιστοιχεί) τους αρχιτεκτονικούς καταχωρητές IA-32 στους φυσικούς καταχωρητές της μηχανής. Αυτό επιτρέπει στους 8 καταχωρητές ακεραίων γενικής χρήσης IA-32 να επεκταθούν δυναμικά για να μπορούν να χρησιμοποιήσουν τους διαθέσιμους 128 φυσικούς καταχωρητές. Το κύκλωμα μετονομασίας χρησιμοποιεί έναν Πίνακα Ψευδωνύμων Καταχωρητών (Register Alias Table RAT) για να ελέγχει την τελευταία κατάσταση του κάθε αρχιτεκτονικού καταχωρητή και να μπορεί να ενημερώνει τις επόμενες εντολές για το που θα πάρουν τους τελεστές εισόδου τους.

Εφόσον ο κάθε λογικός επεξεργαστής πρέπει να διατηρεί και να παρακολουθεί την πλήρη αρχιτεκτονική του κατάσταση, υπάρχουν δύο RAT, ένα για κάθε λογικό επεξεργαστή. Η λειτουργία μετονομασίας των καταχωρητών εκτελείται παράλληλα με τη λειτουργία του κυκλώματος allocator που περιγράφτηκε παραπάνω, έτσι ώστε το κύκλωμα μετονομασίας καταχωρητών να δουλεύει πάνω στις ίδιες uops πάνω στις οποίες ο allocator αναθέτει πόρους.

Όταν οι uops έχουν ολοκληρώσει τις διαδικασίες ανάθεσης και μετονομασίας καταχωρητών, τοποθετούνται σε δύο σύνολα από ουρές, μία για λειτουργίες μνήμης (load και store) και μία για όλες τις άλλες λειτουργίες. Αυτές οι δύο ουρές ονομάζονται ουρά εντολών μνήμης και ουρά γενικών εντολών, αντίστοιχα. Οι δύο αυτές ουρές και πάλι διαμοιράζονται με τέτοιο τρόπο ώστε οι uops από κάθε λογικό επεξεργαστή να μπορούν να χρησιμοποιήσουν το πολύ τις μισές καταχωρήσεις των ουρών.

2.3.3.7.3 Δρομολόγηση Εντολών

Οι δρομολογητές (schedulers) βρίσκονται στην καρδιά της μηχανής εκτέλεσης out-of-order. Πέντε δρομολογητές uop χρησιμοποιούνται για να δρομολογούν διαφορετικούς τύπους uop για τις διάφορες μονάδες εκτέλεσης. Συνολικά, μπορούν να τροφοδοτούν με μέχρι και έξι uop σε κάθε κύκλο ρολογιού. Οι δρομολογητές αποφασίζουν πότε οι uop είναι έτοιμες για εκτέλεση με βάση την κατάσταση ετοιμότητας των καταχωρητών τελεστών εισόδου από τους οποίους εξαρτώνται και τη διαθεσιμότητα των μονάδων εκτέλεσης.

Οι ουρές εντολών μνήμης και γενικών λειτουργιών προωθούν uops στις πέντε ουρές δρομολόγησης όσο πιο σύντομα μπορούν, εναλλάσσοντας μεταξύ uops για τους δύο λογικούς επεξεργαστές σε κάθε κύκλο, όπου κρίνεται απαραίτητο.

Σε κάθε δρομολογητή αντιστοιχεί μία ουρά δρομολόγησης των οκτώ με είκοσι καταχωρήσεων από την οποία επιλέγει uops για να τροφοδοτήσει τις μονάδες εκτέλεσης. Οι δρομολογητές επιλέγουν uops ανεξάρτητα από το εάν ανήκουν στον ένα λογικό επεξεργαστή ή στον άλλο. Η λειτουργία των δρομολογητών αγνοεί πλήρως την ύπαρξη δύο λογικών επεξεργαστών. Οι uops κρίνονται μόνο με βάση τη ετοιμότητα των εισόδων τους και τη διαθεσιμότητα των πόρων εκτέλεσης. Για παράδειγμα, οι δρομολογητές θα μπορούσαν να εκδώσουν δύο uops από τον ένα λογικό επεξεργαστή και δύο από τον άλλον στον ίδιο κύκλο ρολογιού. Για την αποφυγή αδιεξόδων και την διατήρηση δίκαιης κατανομής πόρων, υπάρχει ένας περιορισμένος αριθμός ενεργών εισόδων που μπορεί κάθε λογικός επεξεργαστής να έχει σε κάθε μία από τις ουρές δρομολόγησης. Το όριο αυτού του περιορισμού εξαρτάται από το μέγεθος της ουράς δρομολόγησης.

2.3.3.7.4 Μονάδες Εκτέλεσης

Ο πυρήνας εκτέλεσης και η ιεραρχία μνήμης αγνοούν επίσης σε μεγάλο βαθμό την ύπαρξη δύο λογικών επεξεργαστών. Αφού οι καταχωρητές πηγής και προορισμού έχουν ήδη αντιστοιχιστεί σε φυσικούς καταχωρητές σε ένα σύνολο από κοινούς φυσικούς καταχωρητές, οι uops απλά προσπελαύνουν το φυσικό αρχείο καταχωρητών για να πάρουν τους καταχωρητές προορισμού τους, και γράφουν τα αποτελέσματά τους πίσω στο φυσικό αρχείο καταχωρητών. Η σύγκριση μεταξύ των αριθμών των φυσικών καταχωρητών και μόνο επιτρέπει στο κύκλωμα προώθησης να προωθεί τα αποτελέσματα σε άλλες uop που βρίσκονται σε εκτέλεση χωρίς να χρειάζεται να έχει γνώση για τους λογικούς επεξεργαστές.

Μετά την εκτέλεση, οι uops τοποθετούνται στο buffer αναδιάταξης. Το buffer αυτό διαχωρίζει στο στάδιο εκτέλεσης από το στάδιο απόσυρσης των εντολών. Το buffer αναδιάταξης χωρίζεται με τέτοιο τρόπο ώστε ο κάθε λογικός επεξεργαστής να μπορεί να χρησιμοποιήσει τις μισές του εγγραφές.

2.3.3.7.5 Απόσυρση Εντολών

Το κύκλωμα απόσυρσης των uop εκτελεί λειτουργία commit της αρχιτεκτονικής κατάστασης με τη σειρά προγράμματος. Το κύκλωμα αυτό παρακολουθεί πότε uops από τους δύο λογικούς επεξεργαστές είναι έτοιμες για απόσυρση, και μετά αποσύρει τις uops σε σειρά προγράμματος για κάθε λογικό επεξεργαστή εναλλάσσοντας μεταξύ των δύο λογικών επεξεργαστών. Το κύκλωμα απόσυρσης θα αποσύρει uops του ενός λογικού επεξεργαστή, μετά του άλλου, εναλλάσσοντας μπρος και πίσω μεταξύ των δύο. Εάν ο ένας λογικός επεξεργαστής δεν έχει καμία εντολή για απόσυρση τότε όλο το διαθέσιμο εύρος ζώνης αφιερώνεται στον άλλο λογικό επεξεργαστή.

Όταν οι εντολές store έχουν αποσυρθεί, τα δεδομένα προς αποθήκευση πρέπει να γραφούν πάνω στην κρυφή μνήμη δεδομένων επιπέδου ένα. Το κύκλωμα επιλογής εναλλάσσεται μεταξύ των δύο λογικών επεξεργαστών εκτελεί commit των δεδομένων αποθήκευσης στην κρυφή μνήμη.

2.3.3.8 Υποσύστημα Μνήμης

Το υποσύστημα μνήμης περιλαμβάνει το DTLB, την χαμηλού latency κρυφή μνήμη επιπέδου 1 (L1), την επιπέδου 2 ενοποιημένη κρυφή μνήμη (L2), και την ενοποιημένη κρυφή μνήμη επιπέδου 3 (Η κρυφή μνήμη επιπέδου 3 είναι μόνο διαθέσιμη στον επεξεργαστή Intel Xeon MP). Η πρόσβαση στο υποσύστημα μνήμης αγνοεί επίσης σε μεγάλο βαθμό την ύπαρξη δύο λογικών επεξεργαστών. Οι δρομολογητές στέλνουν uops load ή store χωρίς αναφορά στους λογικούς επεξεργαστές και το υποσύστημα μνήμης τις χειρίζεται καθώς φτάνουν.

2.3.3.9 DTLB

Το DTLB μεταφράζει διευθύνσεις σε φυσικές διευθύνσεις. Έχει 64 πλήρως συσχετιστικές (fully associative) εγγραφές · κάθε εγγραφή μπορεί να αντιστοιχίζει μία σελίδα είτε 4K είτε 4MB. Παρόλο που το DTLB είναι μία δομή μοιραζόμενη από τους δύο λογικούς επεξεργαστές, κάθε εγγραφή περιέχει ετικέτα με το ID του λογικού επεξεργαστή. Κάθε λογικός επεξεργαστής έχει επίσης ένα προσωρινό καταχωρητή για την εξασφάλιση δίκαιης κατανομής και προόδου στις αστοχίες DTLB.

2.3.3.10 Κρυφές Μνήμες L1,L2 και L3

Η κρυφή μνήμη δεδομένων L1 είναι μία μνήμη 4-way set associative με γραμμές των 64-byte. Είναι μία κρυφή μνήμη write-through, που σημαίνει ότι οι εγγραφές πάντα αντιγράφονται και στην κρυφή μνήμη L2. Η κρυφή μνήμη L1 έχει virtual διευθυνσιοδότηση και περιέχει φυσικές ετικέτες.

Οι κρυφές μνήμες L2 και L3 είναι 8-way set associative με γραμμές των 128-byte. Οι κρυφές μνήμες αυτές είναι φυσικά διευθυνσιοδοτημένες. Και οι δύο λογικοί επεξεργαστές, ανεξάρτητα με το ποιου λογικού επεξεργαστή οι uops έφεραν δεδομένα στην κρυφή μνήμη αρχικά, μπορούν να μοιράζονται από κοινού όλες τις εγγραφές σε όλα τα επίπεδα κρυφής μνήμης.

Επειδή οι λογικοί επεξεργαστές μπορούν να μοιράζονται δεδομένα στην κρυφή μνήμη, υπάρχει το ενδεχόμενο των διενέξεων κρυφής μνήμης, που μπορεί να οδηγήσει σε παρατήρηση μειωμένης επίδοσης. Υπάρχει, όμως, και το ενδεχόμενο για ύπαρξη κοινών δεδομένων στην κρυφή μνήμη. Για παράδειγμα, ένας λογικός επεξεργαστής μπορεί να

φορτώσει εντολές οι δεδομένα στην κρυφή μνήμη, που χρειάζονται από τον άλλο επεξεργαστή · αυτό είναι κάτι συνηθισμένο σε κώδικα εφαρμογών για διακομιστές. Στο μοντέλο χρήσης παραγωγού-καταναλωτή, ένας λογικός επεξεργαστής μπορεί να παράγει δεδομένα που ο άλλος λογικός επεξεργαστής θέλει να χρησιμοποιήσει. Σε τέτοιες περιπτώσεις, υπάρχει πιθανότητα για καλά οφέλη στην επίδοση.

2.3.3.11 Δίαυλος

Οι αιτήσεις μνήμης των λογικών επεξεργαστών που δεν μπορούν να ικανοποιηθούν από την ιεραρχία κρυφής μνήμης χειρίζονται από το δίαυλο επικοινωνίας. Το κύκλωμα του διαύλου περιλαμβάνει τον τοπικό APIC ελεγκτή διακοπών, καθώς επίσης και μνήμη συστήματος εκτός τσιπ και χώρο I/O. Το κύκλωμα του διαύλου αντιμετωπίζει επίσης θέματα συνάφειας (snooping) διευθύνσεων που θα τοποθετηθούν στην κρυφή μνήμη από αιτήσεις που προήλθαν από άλλους agents εξωτερικού διαύλου, καθώς επίσης και εισερχόμενες αιτήσεις διακοπών διαμέσου του τοπικού APIC.

Από πλευράς υπηρεσιών, οι αιτήσεις από τους λογικούς επεξεργαστές αντιμετωπίζονται με βάση τη λογική first-come, με μοιραζόμενο αποθηκευτικό χώρο για τις ουρές και το buffering. Δε δίνεται προτεραιότητα σε ένα από τους δύο λογικούς επεξεργαστές.

Η διάκριση μεταξύ αιτημάτων από τους λογικούς επεξεργαστές διατηρείται με αξιοπιστία από τις ουρές διαύλου παρόλα αυτά. Οι αιτήσεις στον τοπικό APIC και οι πόροι μεταβίβασης διακοπών είναι μοναδικοί και ανεξάρτητοι για κάθε λογικό επεξεργαστή. Το κύκλωμα διαύλου επίσης εκτελεί μέρος των λειτουργιών barrier fence και των λειτουργιών ordering της μνήμης, οι οποίες τοποθετούνται στις ουρές των αιτήσεων διαύλου με βάση τη λογική από ποιο λογικό επεξεργαστή προήλθαν.

Για σκοπούς αποσφαλμάτωσης, και σαν βοήθεια στους μηχανισμούς διασφάλισης ανεξάρτητης προόδου σε υλοποιήσεις clustered πολυεπεξεργαστών, το ID του λογικούς επεξεργαστή αποστέλλεται εμφανώς στον εξωτερικό δίαυλο του επεξεργαστή στη φάση αίτησης των συναλλαγών. Άλλες αιτήσεις διαύλου, όπως μεταφορά (eviction) γραμμών κρυφής μνήμης ή λειτουργίες prefetch, κληρονομούν το ID του λογικού επεξεργαστή της αίτησης που παρήγαγε τη συναλλαγή.

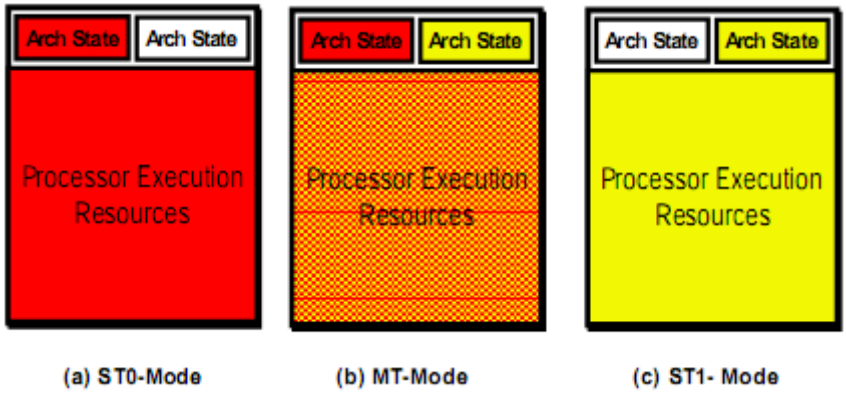
2.3.3.11 Λειτουργίες Single-Task και Multi-Task

Για τη βελτιστοποίηση της απόδοσης όταν μόνο ένα προγραμματιστικό νήμα διατίθεται για εκτέλεση, υπάρχουν δύο τρόποι λειτουργίας που αναφέρονται ως single-task (ST) ή multi-task(MT). Στη περίπτωση του MT, υπάρχουν δύο ενεργοί λογικοί επεξεργαστές και κάποιοι

από τους υπολογιστικούς πόρους διαμερίζονται όπως περιγράφτηκε προηγουμένως. Υπάρχου δύο εναλλακτικές στη λειτουργία ST: single-task στον λογικό επεξεργαστή 0 (ST0) και single-task στον λογικό επεξεργαστή 1 (ST1). Στη λειτουργία ST0 ή ST1, μόνο ο ένας εκ των δύο λογικών επεξεργαστών είναι ενεργός, και οι πόροι που είχαν διαμεριστεί κατά τη λειτουργία MT έχουν επανέλθει για να δώσουν στον ένα ενεργό λογικό επεξεργαστή όλους τους πόρους. Η αρχιτεκτονική της Intel IA-32 περιέχει την εντολή HALT που σταματά τη λειτουργία εκτέλεσης ενός επεξεργαστή και κανονικά αφήνει τον επεξεργαστή να μπει σε λειτουργία χαμηλής ισχύος. Η εντολή HALT είναι μία προνομιούχα εντολή, κάτι που σημαίνει ότι μόνο το λειτουργικό σύστημα ή άλλες διεργασίες ring-0 μπορούν να την εκτελέσουν. Εφαρμογές επιπέδου χρήστη δεν μπορούν να εκτελέσουν την εντολή αυτή.

Σε ένα επεξεργαστή με την τεχνολογία Hyper-Threading, η εκτέλεση της εντολής HALT μεταβάλει τη λειτουργία του επεξεργαστή από την MT στην ST0 ή στην ST1, ανάλογα με το ποιος λογικός επεξεργαστής εκτέλεσε την εντολή HALT. Για παράδειγμα, εάν ο λογικός επεξεργαστής εκτέλεσε την HALT, μόνο ο λογικός επεξεργαστής 1 θα είναι ενεργός · ο φυσικός επεξεργαστής θα μπει έτσι στη λειτουργία ST1 και οι μοιραζόμενοι πόροι θα επανενωθούν για να δώσουν στο λογικό επεξεργαστή 1 πλήρη πρόσβαση σε όλους τους υπολογιστικούς πόρους. Εάν οι εναπομείναντες λογικοί επεξεργαστές εκτελέσουν εκ νέου την εντολή HALT, ο επεξεργαστής θα μεταβεί σε λειτουργία χαμηλής ισχύος.

Κατά τις λειτουργίες ST0 ή ST1, μία διακοπή που αποστέλλεται στον σταματημένο επεξεργαστή θα προκαλέσει μετάβαση στην κατάσταση MT. Το λειτουργικό σύστημα είναι υπεύθυνο για τη διαχείριση των μεταβάσεων λειτουργίας MT (όπως περιγράφεται στην επόμενη παράγραφο).



Σχήμα 2.8: Ανάθεση υπολογιστικών πόρων

Στο Σχήμα 2.8 συνοψίζεται η συζήτησή μας. Σε ένα επεξεργαστή με την τεχνολογία Hyper-Threading, οι υπολογιστικοί πόροι ανατίθενται σε έναν και μόνο λογικό επεξεργαστή εάν ο φυσικός επεξεργαστής βρίσκεται στην κατάσταση ST0 ή ST1. Σε λειτουργία MT, οι υπολογιστικοί πόροι μοιράζονται μεταξύ των δύο λογικών επεξεργαστών.

2.3.3.12 Λειτουργικά Συστήματα και Εφαρμογές

Ένα σύστημα με επεξεργαστές που χρησιμοποιούν την τεχνολογία Hyper-Threading φαίνεται στο λειτουργικό σύστημα και στις εφαρμογές λογισμικού σαν να έχει τον διπλάσιο αριθμό επεξεργαστών σε σχέση με τους φυσικούς που έχει. Τα λειτουργικά συστήματα διαχειρίζονται τους λογικούς επεξεργαστές όπως θα έκαναν και με τους φυσικούς επεξεργαστές, δρομολογώντας τρέχουσες ρουτίνες ή νήματα στους λογικούς επεξεργαστές. Όμως, για την καλύτερη δυνατή επίδοση, το λειτουργικό σύστημα θα έπρεπε να εφαρμόζει δύο βελτιστοποιήσεις.

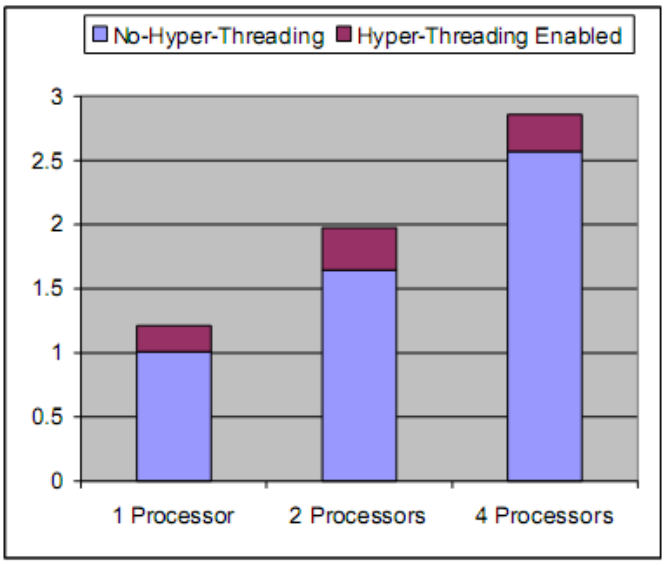
Η πρώτη είναι να χρησιμοποιεί στην εντολή HALT εάν ο ένας λογικός επεξεργαστής είναι ενεργός ενώ ο άλλος όχι. Η εντολή HALT θα επιτρέπει στον επεξεργαστή να μεταβαίνει στην κατάσταση ST0 ή ST1. Ένα λειτουργικό σύστημα που δε χρησιμοποιεί αυτή την βελτιστοποίηση θα εκτελεί στον αδρανή λογικό επεξεργαστή μία ακολουθία από εντολών που επανειλημμένα ελέγχουν για διαθέσιμες εργασίες. Αυτός ο καλούμενος "άεργος βρόχος" (idle loop) μπορεί να καταναλώνει σημαντικούς πόρους εκτέλεσης που θα μπορούσαν εναλλακτικά να χρησιμοποιηθούν για να αυξήσουν την απόδοση του άλλου ενεργού λογικού επεξεργαστή.

Η δεύτερη προτεινόμενη βελτιστοποίηση αφορά τη δρομολόγηση προγραμματιστικών νημάτων στους λογικούς επεξεργαστές. Γενικά, για καλύτερη επίδοση, το λειτουργικό σύστημα θα έπρεπε να δρομολογεί νήματα σε λογικούς επεξεργαστές που βρίσκονται σε διαφορετικούς φυσικούς επεξεργαστές προτού δρομολογήσει πολλαπλά νήματα στον ίδιο φυσικό επεξεργαστή. Αυτή η βελτιστοποίηση επιτρέπει στα προγραμματιστικά νήματα να χρησιμοποιούν διαφορετικές φυσικούς υπολογιστικούς πόρους όταν είναι διαθέσιμοι.

2.3.3.13 Απόδοση

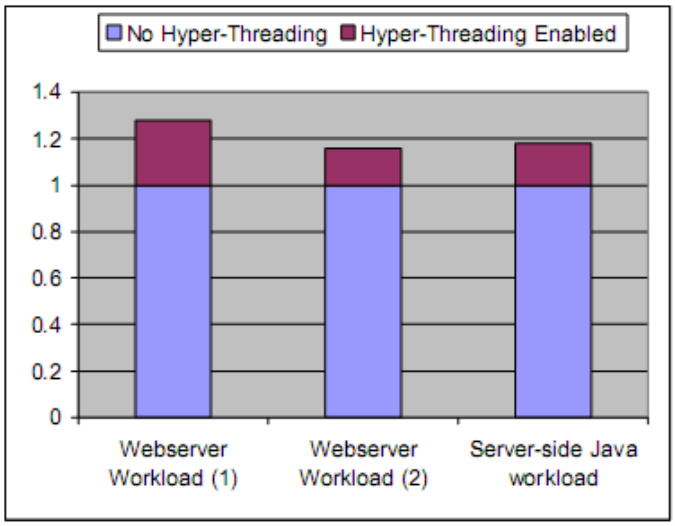
Η οικογένεια επεξεργαστών Intel Xeon παρέχει την υψηλότερη επίδοση συστήματος διακομιστή από οποιοδήποτε άλλο IA-32 αρχιτεκτονικό επεξεργαστή της Intel που έχει παρουσιαστεί μέχρι σήμερα. Προκαταρτικές μετρήσεις benchmark τεστ παρουσιάζουν αύξηση επιδόσεων μέχρι και 65% σε high-end εφαρμογές διακομιστών σε σύγκριση με προηγούμενης γενιάς Pentium® III Xeon επεξεργαστές σε πλατφόρμες διακομιστών 4-way. Σημαντικό μέρος αυτής της αύξησης στην επίδοση μπορεί να καταλογιστεί στην τεχνολογία Hyper-Threading.

® Pentium είναι σήμα κατατεθέν της Intel Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες



Σχήμα 2.9: Αύξηση της επίδοσης λόγω Hyper-Threading σε φόρτο εργασίας OLTP

Στο Σχήμα 2.9 παρουσιάζεται η επίδοση online επεξεργασίας δοσοληψιών, κλιμακώνοντας από σύστημα ενός μέχρι και σύστημα τεσσάρων επεξεργαστών με ενεργοποιημένη την τεχνολογία Hyper-Threading. Αυτή η γραφική έχει κανονικοποιηθεί στην επίδοση του μονο-επεξεργαστικού συστήματος. Μπορεί να δει κανείς ότι υπάρχει σημαντικό συνολικό κέρδος στην επίδοση αποδιδόμενο στην τεχνολογία Hyper-Threading, που ανέρχεται στο 21% στις περιπτώσεις των συστημάτων ενός και δύο επεξεργαστών.



Σχήμα 2.10: Επίδοση σε benchmark εφαρμογών διακομιστών

Στο Σχήμα 2.10 φαίνεται το κέρδος λόγω της τεχνολογίας Hyper-Threading σε εκτέλεση benchmarks επικεντρωμένα σε εφαρμογές διακομιστών. Οι φόρτοι εργασίας που επιλέχθηκαν ήταν δύο διαφορετικά benchmarks που ήταν σχεδιάστηκαν να ασκούν χαρακτηριστικά διακομιστών δεδομένων και Web και ένας φόρτος εργασίας που

επικεντρώνεται στην άσκηση της πλευράς διακομιστή ενός περιβάλλοντος Java. Σε αυτές τις περιπτώσεις το κέρδος σε επίδοση κυμάνθηκε από 16 μέχρι 23%.

Κεφάλαιο 3

Αλγοριθμικά Μοντέλα – Μετασχηματισμοί Βρόχων

3.1 Γενικά

Το φλέγον ζήτημα στο πρόβλημα της αυτόματης παραλληλοποίησης βρόχων και της αποδοτικής αντιστοίχησής τους σε διαφορετικές παράλληλες αρχιτεκτονικές αυτό είναι η ανακούφιση του κόστους επικοινωνίας με τον αποδοτικό έλεγχο της λεπτότητας στο ισοζύγιο μεταξύ υπολογισμού και επικοινωνίας. Σε συστήματα κατανεμημένης μνήμης, οι ρητές αποστολές μηνυμάτων προκαλούν ένα επιπλέον κόστος επικοινωνίας λόγω αναγκαίων καθυστερήσεων στην έναρξη της επικοινωνίας και καθυστερήσεις στην αποστολή των δεδομένων.

Για την ελαχιστοποίηση του κόστους επικοινωνίας, ο Shang[8], ο Hollander[9] και άλλοι, έχουν παρουσιάσει μεθόδους για διαίρεση του χώρου δεικτοδότησης σε ανεξάρτητα σύνολα επαναλήψεων, τα οποία ανατίθενται σε διαφορετικούς επεξεργαστές. Ωστόσο, σε πολλές περιπτώσεις, η διαμέριση του χώρου δεικτοδότησης σε ανεξάρτητα σύνολα δεν είναι δυνατή, συνεπώς οι ανταλλαγές δεδομένων μεταξύ επεξεργαστών επιβάλλουν επιπλέον καθυστερήσεις επικοινωνίας. Για όσο αφορά παραλληλισμό μικρής λεπτότητας, πολλές μέθοδοι έχουν προταθεί για την συγκέντρωση γειτονικών αλυσίδων επαναλήψεων, διατηρώντας όμως τη βέλτιστη δρομολόγηση υπερεπιπέδου ([10],[11]).

Όσον αφορά τον αδρομερής παραλληλισμός (coarse grain parallelism), που ενδιαφέρει στην περίπτωσή μας, οι ερευνητές ασχολούνται με το πρόβλημα της ανακούφισης του κόστους επικοινωνίας με την εφαρμογή του μετασχηματισμού tiling (ή υπερκόμβος). Κάτω από αυτό το σχήμα, τα γειτονικά σημεία επαναλήψεων ομαδοποιούνται για την κατασκευή ενός μεγαλύτερο κόμβου επεξεργασίας που μπορεί να εκτελεστεί ατομικά χωρίς παρεμβάσεις. Οι ανταλλαγές δεδομένων επίσης ομαδοποιούνται και αποστέλλονται σαν ένα μήνυμα για κάθε γειτονικό επεξεργαστή, στο τέλος της εκτέλεσης ενός ατομικού υπερκόμβος.

Σε αυτή την παράγραφο παρουσιάζουμε το αλγοριθμικό μοντέλο που χρησιμοποιούμε στα πλαίσια της διπλωματικής εργασίας, καθώς επίσης και μια αναφορά στο μετασχηματισμό tile και σε διάφορες θεωρητικές έννοιες που κρίνεται απαραίτητο να ειπωθούν.

3.2 Σημειογραφία

Στα πλαίσια της εργασίας μας, χρησιμοποιείται η παρακάτω σημειογραφία: N είναι το σύνολο των φυσικών, Z το σύνολο των ακεραίων, n ο αριθμός των εμφωλευμένων βρόχων του αλγορίθμου και m ο αριθμός των διανυσμάτων εξάρτησης του αλγορίθμου. Ο χώρος επαναλήψεων του βρόχου ορίζεται ως $J^{n+1} \subset Z$: $J^{n+1} = \{\vec{j}(j_1, j_2, ..., j_{n+1}) \in Z^{n+1} \wedge l_i \le j_i \le u_i,\ i = 1...n+1\}$. Κάθε σημείο σε αυτό το $n+1-$ διάστατο χώρο είναι ένα ξεχωριστό στιγμιότυπο του σώματος του βρόχου. Ένα διάνυσμα εξαρτήσεων ορίζεται ως: $d_i = (d_{i1}, ..., d_{in}), 1 \le i \le m$. Ο πίνακας εξαρτήσεων D ενός αλγορίθμου Α είναι ένα σύνολο όλων των διανυσμάτων εξάρτησης αυτού του αλγορίθμου: $D = \{d_1, d_2, ..., d_m\}$. Σημειώστε ότι όλα τα διανύσματα εξάρτησης θεωρούνται ομοιόμορφα και σταθερά, π.χ. είναι ανεξάρτητα των στιγμιότυπων των υπολογισμών.

3.3 Αλγοριθμικό Μοντέλο

Το αλγοριθμικό μας μοντέλο αφορά εφαρμογές οι οποίες περιλαμβάνουν τέλεια εμφωλευμένους βρόχους $(n+1)$-διαστάσεων με σταθερές εξαρτήσεις στη ροή εκτέλεσης. Ο χώρος των επαναλήψεων J^{n+1} είναι ορθογώνιος, συνεπώς ισχύει $J^{n+1} = \{\vec{j}(j_1, j_2, ..., j_{n+1}) \in Z^{n+1} \wedge l_i \le j_i \le u_i,\ i = 1...n+1\}$, όπου $l_i, u_i \in Z$ είναι τα κάτω και άνω όρια του i-οστού βρόχου, αντίστοιχα. Οι εξαρτήσεις του προβλήματος εκφράζονται μέσω σταθερών διανυσμάτων $(n+1)$-διαστάσεων της μορφής $\vec{d}_i, i = 1...m$. Ορίζουμε το $\vec{d}_{ij}$ ως το j-οστό στοιχείο του διανύσματος $\vec{d}_i$. Στην κλάση των προβλημάτων που εξετάζονται ισχύει $\vec{d}_{ij} \ge 0, i = 1...m$ και $j = 1...n+1$. Ο πίνακας εξαρτήσεων του αλγορίθμου, D, είναι ένας πίνακας $(n+1) \times m$ που έχει ως στήλες τα διανύσματα εξαρτήσεων του αλγορίθμου. Ισχύει ότι $rank(D) = n+1$, που σημαίνει ότι ο αλγόριθμος έχει $n+1$ γραμμικώς ανεξάρτητα διανύσματα εξάρτησης. Σημειώστε ότι, εάν $rank(D) < n+1$, τότε ο χώρος επαναλήψεων θα μπορούσε να διαμεριστεί σε ανεξάρτητους υποχώρους και να παραλληλοποιηθεί χωρίς τη χρήση του tiling που δίνεται στο [12]. Ορίζουμε το διάνυσμα $\vec{d} = (d'_1, d'_2, ..., d'_{n+1})$, όπου $d'_i = \max(d_{il}), l = 1...m$, που εκφράζει το μέγιστο μήκος εξάρτησης ανά διάσταση. Σε αντιδιαστολή με τα [13],[14], εμείς θεωρούμε υπολογισμούς εντός πυρήνα (in-core), π.χ. όλα τα σύνολα δεδομένων (data sets) που ανατίθενται σε κάθε διεργασία χωράνε στην κύρια μνήμη, συνεπώς δεν υπολογίζουμε χρόνους προσπέλασης δευτερεύουσας αποθήκευσης.

Γενικά, οι αλγόριθμοι έχουν τη μορφή του Αλγόριθμου 1, όπου U είναι ένας πίνακας (n+1)-διαστάσεων και F είναι μία γραμμική συνάρτηση.

Αλγόριθμος 1: Αλγοριθμικό Μοντέλο

1 **for** $j_1 \leftarrow l_1$ **to** u_1 **do**

2 …

3 **for** $j_n \leftarrow l_n$ **to** u_n **do**

4 **for** $j_{n+1} \leftarrow l_{n+1}$ **to** u_{n+1} **do**

5 $U[\vec{j}] = F(U[\vec{j} - \vec{d}_1], ..., U[\vec{j} - \vec{d}_m]);$

3.3.1 Παράδειγμα εφαρμογής: Εξίσωση Διάχυσης

Διάχυση είναι το φυσικό φαινόμενο της διακίνησης στο εσωτερικό ενός υγρού (ή αερίου), όπως για παράδειγμα είναι η διακίνηση μολυσμένων σωματιδίων στην ατμόσφαιρα. Τα φαινόμενα διάχυσης μελετούνται συχνά στη μετεωρολογία. Η εξίσωση διάχυσης είναι η μερική διαφορική εξίσωση (ΜΔΕ) που καθορίζει την κίνηση ενός βαθμωτού καθώς διαχέεται από ένα γνωστό πεδίο ταχύτητας (το υλικό στο οποίο λαμβάνει χώρα η διάχυση). Η εξίσωση διάχυσης για ένα βαθμωτό μέγεθος v (π.χ. πυκνότητα σωματιδίων ή θερμοκρασία) εκφράζεται μαθηματικά ως εξής:

$$\frac{\partial v}{\partial t} = \vec{a} \nabla v$$

όπου $\vec{a}$ είναι το διάνυσμα πεδίου, π.χ. το διάνυσμα ταχύτητας του υλικού. Σε δύο χωρικές διαστάσεις η παραπάνω εξίσωση είναι ισοδύναμη με την ακόλουθη:

$$\frac{\partial v}{\partial t} = a_x \frac{\partial v}{\partial x} + a_y \frac{\partial v}{\partial y} \qquad (1)$$

Εάν θέλουμε να εξετάσουμε το φαινόμενο της διάχυσης σε ένα χώρο $X \times Y$ για χρονικό διάστημα T, μπορούμε να διαμερίσουμε το αρχικό χωρίο σε ένα διακριτό ομοιόμορφο πλέγμα χρησιμοποιώντας χρονικό βήμα Δt και χωρικό βήματα Δx και Δy. Μετά, μπορούμε να διαμερίσουμε σε διακριτά κομμάτια την παραπάνω ΜΔΕ χρησιμοποιώντας διάφορα πεπερασμένα διαφορετικά σχήματα. Για παράδειγμα, εάν εφαρμόσουμε το σχήμα *Euler-Forward* [15], η χρονική παράγωγος μπορεί να αντικατασταθεί από το ακόλουθο κλάσμα διαφορών: $\frac{\partial v}{\partial t} = \frac{v_{ij}^{n+1} - v_{ij}^{n}}{\Delta t}$. Η φυσική του προβλήματος μας επιτρέπει να εφαρμόσουμε διαφορικά σχήματα τύπου *upwind* [15] για τις χωρικές μερικές παραγώγους, τα οποία περιλαμβάνουν υπολογισμούς με "προηγούμενα" χωρικά σημεία του πλέγματος. Η

στρατηγική διαμέρισης σε διακριτά μέρη ευνοεί την απευθείας εφαρμογή ορθογώνιου tiling στη συνέχεια. Συνεπώς, στην περίπτωσή που εξετάζουμε μπορούμε να αντικαταστήσουμε τις χωρικές μερικές παραγώγους ως εξής: $\frac{\vartheta v}{\vartheta x} = \frac{v_{ij}^{n} - v_{(i-1)j}^{n}}{\Delta x}$. Εάν αντικαταστήσουμε τις παραπάνω φόρμουλες στην Εξίσωση (1) λαμβάνουμε:

$$v_{ij}^{n+1} = (1 + 2a\frac{\Delta t}{\Delta x})v_{ij}^{n} - a\frac{\Delta t}{\Delta x}(v_{(i-1)j}^{n} + v_{i(j-1)}^{n}) \qquad (2)$$

όπου για σημειογραφική απλότητα υποθέσαμε ένα ομοιόμορφο υπολογιστικό πλέγμα στις δύο χωρικές διαστάσεις ($\Delta x = \Delta y$) και $a_x = a_y = a$. Σημειώστε ότι τα $v_{ij}^{0}, v_{0j}^{n}, v_{i0}^{n}$ είναι γνωστό από τις αρχικές συνθήκες και συνοριακές συνθήκες του προβλήματος ΜΔΕ. Η Εξίσωση (2) μπορεί εύκολα να επιλυθεί για όλα τα σημεία στο διακριτό πλέγμα υπολογισμού $T' \times X' \times Y'$, όπου $T' = T / \Delta t, X' = X / \Delta x$ και $Y' = Y / \Delta y$ για τον εμφωλευμένο βρόχο που παρουσιάζεται στον Αλγόριθμο 2.

Αλγόριθμος 2: Εμφωλευμένος Βρόχος για Εξίσωση Διάχυσης 2-D

```
1  for j1 ← 0 to T' do
2      for j2 ← 0 to X' do
3          for j3 ← 0 to Y' do
4              U[j1+1][j2][j3] = (1 + 2·a·dt/dx)·U[j1][j2][j3] −
               a·dt/dx·(U[j1][j2−1][j3] + U[j1][j2][j3−1]);
```

Ο πίνακας εξαρτήσεων του παραπάνω αλγόριθμού είναι $D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ και $d' = (1,1,1)$. Η διαδικασία διακριτοποίησης που ακολουθείται οδηγεί στο να έχει ο πίνακας εξαρτήσεων μη αρνητικά στοιχεία. Σημειώστε ότι διαφορετικά σχήματα διακριτοποίησης μπορεί να οδηγήσουν σε μακρύτερες εξαρτήσεις. Ο αναγνώστης μπορεί να βρει περισσότερες πληροφορίες στο [15].

3.4 Μετασχηματισμός Υπερκόμβων

Σε ένα μετασχηματισμό υπερκόμβων ο χώρος επαναλήψεων J^{n+1} διαμερίζεται σε πανομοιότυπους παραλληλεπίπεδους χώρους ($n+1$)-διαστάσεων (tiles ή υπερκόμβους) που σχηματίζονται από $n+1$ ανεξάρτητες ομάδες από παράλληλα υπερεπίπεδα. Ο

μετασχηματισμός υπερκόμβων καθορίζεται από ένα τετραγωνικό πίνακα (n+1)-διαστάσεων, H. Κάθε γραμμή του πίνακα H είναι κάθετη σε μία ομάδα υπερεπιπέδων που σχηματίζουν τα tiles. Δυαδικά, ο μετασχηματισμός υπερκόμβων μπορεί να οριστεί από n+1 γραμμικώς ανεξάρτητα διανύσματα, τα οποία αποτελούν τις πλευρές των υπερκόμβων. Όμοια με τον πίνακα H, ο πίνακας P περιέχει τα διανύσματα πλευράς ενός υπερκόμβου ως διανύσματα στήλες. Ισχύει $P = H^{-1}$.

Τυπικά, ένας μετασχηματισμός υπερκόμβων ορίζεται ως:

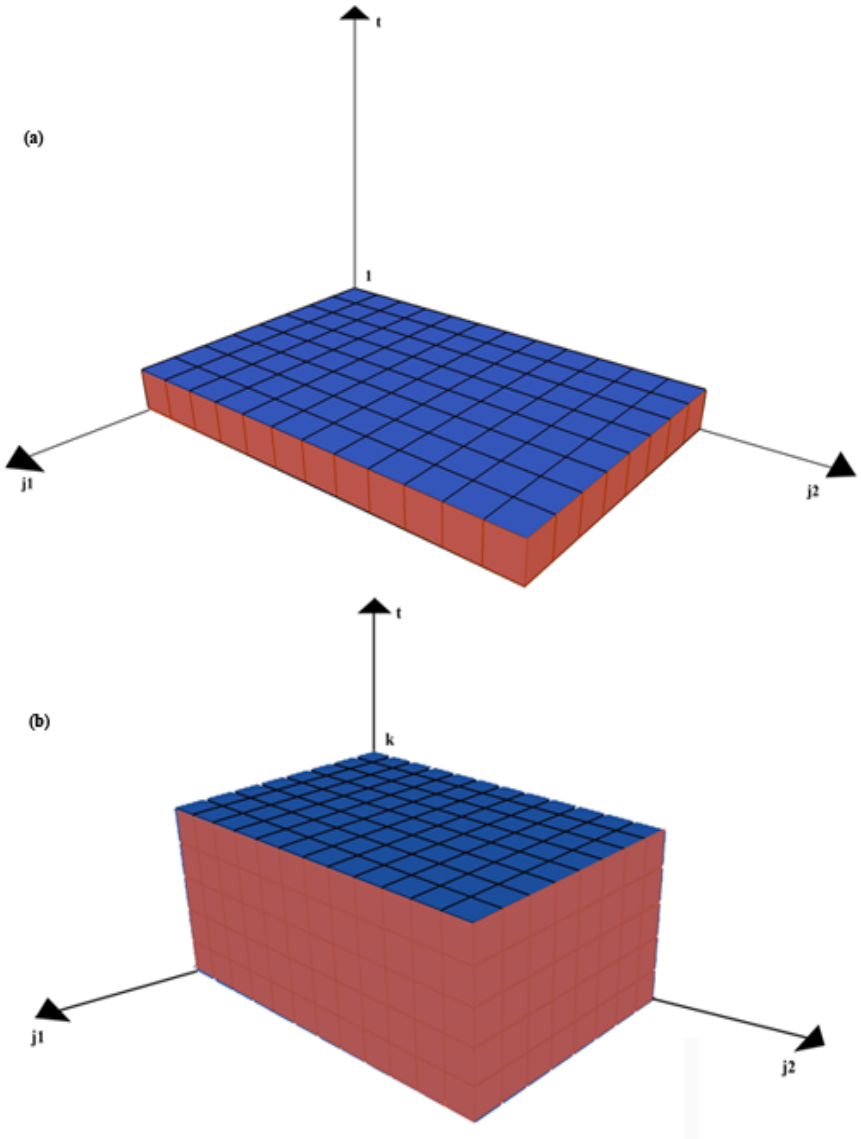
$$r : Z^{n+1} \to Z^{2n+2}, r(\vec{j}) = \begin{bmatrix} \lfloor H\vec{j} \rfloor \\ \vec{j} - H^{-1}\lfloor H\vec{j} \rfloor \end{bmatrix}$$

όπου $\lfloor H\vec{j} \rfloor$ καθορίζει τις συντεταγμένες του tile στο οποίο αντιστοιχεί ο δείχτης $\vec{j}(j_1, j_2, ..., j_{n+1})$ και το $\vec{j} - H^{-1}\lfloor H\vec{j} \rfloor$ δίνει τις συντεταγμένες του $\vec{j}$ μέσα σε αυτό το tile σε σχέση με την αρχή συντεταγμένων του tile. Συνεπώς ο αρχικός χώρος επαναλήψεων (n+1)-διαστάσεων μετασχηματίζεται σε ένα χώρο (2n+2)-διαστάσεων, τις διαστάσεις των tiles και τις διαστάσεις των δεικτών εσωτερικά των tiles. Οι δείκτες εσωτερικά των tiles πρέπει να εκτελούνται ακολουθιακά, ενώ τα tiles αυτά καθαυτά μπορούν να ανατίθενται σε διεργασίες και να εκτελούνται παράλληλα σύμφωνα με μια έγκυρη δρομολόγηση υπερεπιπέδου. Ο tiled χώρος J^S και ο πίνακας εξαρτήσεων του υπερκόμβου D^S ορίζονται ως ακολούθως: $J^S = \{\vec{j}^S(j_1^S, ..., j_{n+1}^S) \mid \vec{j}^S = \lfloor H\vec{j} \rfloor, \vec{j} \in J^{n+1}\}$, $D^S = \{\vec{d}^S \mid \vec{d}^S = \lfloor H(\vec{j}_0 + \vec{d}) \rfloor, \vec{d} \in D,$ $\vec{j}_0 \in J^{n+1} \mid 0 \le \lfloor H\vec{j}_0 \rfloor \le 1\}$, όπου $\vec{j}_0$ ορίζει τα σημεία δεικτοδότησης ου ανήκουν στο πρώτο πλήρες tile ξεκινώντας από την αρχή των αξόνων του χώρου επαναλήψεων J^{n+1}. Ο tiled χώρος μπορεί να γραφτεί επίσης ως: $J^S = \{\vec{j}^S \mid \vec{j}_i^S Z \wedge l_i^S \le j_i^S \le u_i^S, i = 1, .., n\}$. Κάθε άνυσμα $\vec{j}^S$ σε αυτό το χώρο ακεραίων (n+1)-διαστάσεων αποτελεί ένα ανεξάρτητο tile με συντεταγμένες ($j_1^S, j_2^S, ..., j_{n+1}^S$).

Με δεδομένο ένα αλγόριθμο με πίνακα εξαρτήσεων D, για να είναι ένας μετασχηματισμός tile έγκυρος, πρέπει να ισχύει $HD \ge 0$. Αυτό εξασφαλίζει ότι τα tiles είναι ατομικά και ότι η αρχική σειρά εκτέλεσης διατηρείται. Στην αντίθετη περίπτωση οποιαδήποτε σειρά εκτέλεσης των tile θα οδηγήσει σε αδιέξοδο (deadlock). Στα πλαίσια της διπλωματικής μας εργασίας θεωρούμε ότι όλα τα διανύσματα εξαρτήσεων δεν περιέχουν αρνητικά στοιχεία και είναι μικρότερα από το μέγεθος του tile. Αυτό μας επιτρέπει να εφαρμόζουμε ορθογωνικούς μετασχηματισμούς tiling που ορίζονται από τους διαγώνιους πίνακες $H = diag(h_1, h_2, ..., h_{n+1})$ και $P = diag(p_1, p_2, ..., p_{n+1})$. Επιπλέον, όλες οι εξαρτήσεις

περιέχονται εξολοκλήρου σε κάθε περιοχή ενός υπερκόμβου, που σημαίνει ότι $|HD| < 1$ ή εναλλακτικά ότι ο πίνακας εξαρτήσεων του υπερκόμβου, D^S περιέχει μόνο 0 και 1. Αυτή η υπόθεση είναι αρκετά λογική εφόσον τα διανύσματα εξαρτήσεων για συνηθισμένα προβλήματα είναι σχετικά μικρά, ενώ τα μεγέθη των tile συχνά τυγχάνει να είναι τάξεις μεγέθους μεγαλύτερα σε συστήματα με πολύ γρήγορους επεξεργαστές. Σε αυτή την περίπτωση το κάθε tile χρειάζεται να ανταλλάσσει δεδομένα μόνο με τους πλησιέστερούς του γείτονες.

Στο Σχήμα 3.1 παρουσιάζουμε ένα παράδειγμα εφαρμογής μετασχηματισμού υπερκόμβων, όπου το μέγεθος tile είναι k και το αρχικό μας χωρίο επαναλήψεων είναι 2+1 διαστάσεων.



Σχήμα 4.1: (a) Χωρίς tiling, (β) με tile μεγέθους k

3.5 Κόστος Επικοινωνίας – Κόστος Υπολογισμού

Ο αριθμός των σημείων δεικτοδότησης που περιέχεται σε ένα υπερκόμβο αντιπροσωπεύει το αντίστοιχο υπολογιστικό κόστος αυτού του υπερκόμβου (tile), και υπολογίζεται με $\det(P)$. Συνεπώς έχουμε $V_{comp} = \det(P)$ και για το ορθογωνικό tiling $V_{comp} = \det(P) = \prod_{i=1}^{n+1} p_i$. Το κόστος επικοινωνίας για ένα tile είναι ανάλογο του αριθμού των επαναλήψεων βρόχου στα οποία χρειάζεται αν αποσταλούν δεδομένα σε γειτονικούς κόμβους, ή με άλλα λόγια, το σύνολο των διανυσμάτων εξάρτησης που τέμνουν το σύνορο του συγκεκριμένου υπερκόμβου. Αυτό το κόστος μπορεί να υπολογιστεί από την έκφραση:

$$V_{comm}(H) = \frac{1}{\left|\det(H)\right|} \sum_{i=1}^{n} \sum_{k=1}^{n} \sum_{j=1}^{m} \mathrm{h_{i,k}} \mathrm{d_{k,j}} \qquad (3.1)$$

Πρακτικά αυτή η φόρμουλα υπολογίζει το άθροισμα όλων των δυνατών $h_i d_j$, το οποίο αντιπροσωπεύει τη συνεισφορά του κάθε διανύσματος εξάρτησης στο κόστος επικοινωνίας, για κάθε συνοριακή επιφάνεια του tile.

Εάν τα tiles κατά μήκος της ίδιας διάστασης αντιστοιχιστούν στον ίδιο επεξεργαστή, τα διανύσματα εξάρτησης που τέμνουν τη συνοριακή επιφάνεια του tile στην αντίστοιχη διάσταση δεν επιβάλλουν επιπλέον επικοινωνία μεταξύ των επεξεργαστών. Σε αυτή την περίπτωση, το κόστος επικοινωνίας υπολογίζεται από την έκφραση:

$$V_{comm}(H) = \frac{1}{\left|\det(H)\right|} \sum_{i \in \{1,\ldots,x-1,x+1,\ldots,n\},\, j \in \{1,\ldots,m\}} (H_{-x} D)_{i,j'} \qquad (3.2)$$

όπου το H_{-x} αντιστοιχεί στον πίνακα H με το διάνυσμα στήλη να είναι κάθετο στη συνοριακή επιφάνεια στη διάσταση του επεξεργαστή που έχει αντιστοιχιστεί το tile. Στο [16] έχει παρουσιαστεί μια τεχνική που υπολογίζει το διάνυσμα H που επιβάλλει το ελάχιστο βάρος επικοινωνίας για ένα δεδομένο μέγεθος tile.

Κεφάλαιο 4

Δρομολόγηση

4.1 Γενικά

Σε αυτό το κεφάλαιο παρουσιάζουμε αλγόριθμους δρομολόγησης για τα μοντέλα μετασχηματισμένων βρόχων όπως αυτά παρουσιάστηκαν στο κεφάλαιο 3. Συγκεκριμένα παρουσιάζουμε πρόγραμμα δρομολόγησης σύμφωνα με το [17], που μειώνει σημαντικά τον ολικό χρόνο εκτέλεσης του βρόχου κάτω από την προϋπόθεση ότι κάποιο μέρος της κάθε φάσης επικοινωνίας μπορεί να επικαλυφθεί αποδοτικά με ατομικούς, ακέραιους υπολογισμούς των tile. Το ολικό πρόγραμμα χρονοδρομολόγησης μοιάζει με ένα διασωληνωμένο μονοπάτι δεδομένων (datapath) όπου οι υπολογισμοί δεν είναι πλέον διαστρωματωμένοι με αποστολές και λήψεις σε μη-τοπικούς επεξεργαστές.

4.2 Δρομολόγηση Χωρίς Επικάλυψη

Στο [18], οι Hodzic και Shang έχουν παρουσιάσει μία μέθοδο για δρομολόγηση βρόχων που έχουν μετασχηματιστεί διαμέσου ενός μετασχηματισμού υπερκόμβων. Το αποδοτικότερο μέγεθος tile g που ελαχιστοποιεί τον ολικό χρόνο εκτέλεσης καθορίζεται από τις ακριβείς παραμέτρους της παράλληλης αρχιτεκτονικής, όπως είναι για παράδειγμα η λεπτότητα (grain) μεταξύ επικοινωνίας και υπολογισμού. Με δεδομένο το μέγεθος tile, υπολογίζουν τον αποδοτικότερο tile μετασχηματισμό H που μειώνει το κόστος επικοινωνίας για κάθε tile. Οι γραμμές του πίνακα H καθορίζουν το ακριβές σχήμα του tile. Τα σχετικά μεγέθη των πλευρών και του σχήματος των tile καθορίζονται από τα διανύσματα εξάρτησης του αλγορίθμου, όπου ο όγκος του tile(μέγεθος g) καθορίζεται από τις hardware παραμέτρους. Όταν το H έχει υπολογιστεί πλήρως, εφαρμόζεται στον αρχικό χώρο του πίνακα (index space). Ο χώρος υπολογισμού J^S που προκύπτει δρομολογείται με χρήση ενός υπερεπιπέδου χρόνου, Π. Όλα τα tile κατά μήκος μιας συγκεκριμένης διάστασης αντιστοιχούνται στον ίδιο επεξεργαστή. Η συνολική εκτέλεση των tiles περιλαμβάνει διαδοχικές φάσεις υπολογισμού διαστρωματωμένες με φάσεις επικοινωνίας. Ένας επεξεργαστής λαμβάνει τα δεδομένο που χρειάζεται για την εκτέλεση ενός στιγμιότυπου τη χρονική στιγμή i, εκτελεί τους υπολογισμούς και αποστέλλει στους γειτονικούς του επεξεργαστές τα συνοριακά δεδομένα, τα οποία θα χρησιμοποιηθούν για τους υπολογισμούς του στιγμιότυπου τη χρονική στιγμή i+1.

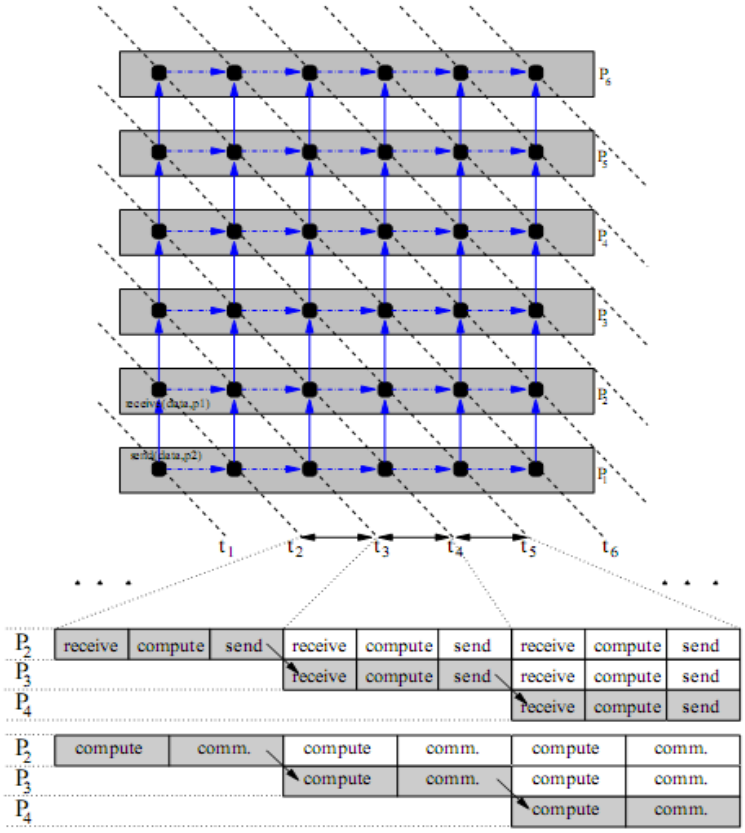
Συνεπώς ο ολικός χρόνος εκτέλεσης δίνεται από τη σχέση:

$$T = P(g)(T_{comp} + T_{comm}), \qquad (4.1)$$

όπου $T_{comm} = T_{startup} + T_{trasmit}$, $P(g)$ ο αριθμός των χρονικών υπερεπιπέδων που απαιτούνται για την εκτέλεση του αλγορίθμου, T_{comp} ο χρόνος εκτέλεσης ενός τετραγωνιδίου ($T_{comp} = gt_c$) και T_{comm} ο χρόνος επικοινωνίας. Είναι ξεκάθαρο ότι ο συνολικός χρόνος εκτέλεσης εξαρτάται από το μέγεθος του tile g, αφού επηρεάζει τον αριθμό των χρονο-επιπέδων (αύξηση του g οδηγεί σε μείωση του ολικού αριθμού των χρονο-επιπέδων), το υπολογιστικό κόστος (gt_c) και τον όγκο επικοινωνίας (V_{comm}).

Ας θεωρήσουμε τώρα την υλοποίηση της παραπάνω δρομολόγησης σε ένα περιβάλλον μεταφοράς μηνυμάτων. Σε αυτό το πλαίσιο ο χρόνος εκτέλεσης μιας φάσης υπολογισμού και μίας φάσης επικοινωνίας αποτελείται από: το χρόνο μετάδοσης των δεδομένων που θα ληφθούν ($T_{trasmit}$), το χρόνο ενεργοποίησης λήψης $T_{startup}$, το χρόνο υπολογισμού $T_{compute}$, το χρόνο ενεργοποίησης αποστολής $T_{startup}$ και το χρόνο μετάδοσης των προς αποστολή δεδομένων ($T_{trasmit}$) (Σχήμα 4.4).

Η συνολική παράλληλη εκτέλεση του βρόχου αποτελείται από ατομικούς υπολογισμούς tile διαστρωματωμένους με φάσεις επικοινωνίας για μετάδοση των αποτελεσμάτων στους γειτονικούς επεξεργαστές. Εφόσον ο χώρος υπολογισμού J^S έχει μόνο μοναδιαία διανύσματα εξάρτησης (δες παράγραφο 3.4), η καλύτερη γραμμική χρονοδρομολόγηση μπορεί εύκολα να αποδειχτεί ότι είναι: Π =[1 1 … 1]. Στο Σχήμα 4.1, παρουσιάζεται η μη επικαλυπτόμενη δρομολόγηση για ένα χωρίο υπολογισμού χρησιμοποιώντας έξι επεξεργαστές. Κάθε χρονικό βήμα μεταξύ διαδοχικών υπερεπιπέδων περιέχει ένα τρίπτυχο λήψης-υπολογισμού-αποστολής μη επικαλυπτόμενων υπό-φάσεων για κάθε tile. Όλα τα tiles κατά μήκος της ίδιας διάστασης αντιστοιχούνται στον ίδιο επεξεργαστή.



Σχήμα 4.1: Χρονοδρομολόγηση χωρίς επικάλυψη

Παράδειγμα 4.1

Θεωρήστε τον κάτωθι αλγόριθμο:

for i1=0 to 9999

for i2=0 to 999

A(i1,i2)=A(i1-1,i2-1)+A(i1-1,i2)+A(i1,i2-1)

endfor

endfor

$J^2 = \{(i_1,i_2) : 0 \le i_1 \le 9999, 0 \le i_2 \le 999\}$, D = {(1,1),(1,0),(0,1)}.

Υποθέστε πως για την αρχιτεκτονική που θα γίνει η εφαρμογή είναι $t_c \approx 1\mu sec$, $t_s = 100t_c$ (λογική υπόθεση εφόσον $t_c \ll t_s$) και $t_t = 0{,}8t_c$/byte (π.χ. Ethernet 10Mbps). Στη συνέχεια και σύμφωνα με την έκφραση (11) στο [18], για $g = ct_s/t_c$ που δίνει το βέλτιστο μέγεθος tile στις δύο διαστάσεις, έχουμε $g = 100$ ($c = 1$, ο αριθμός των γειτονικών επεξεργαστών). Ο όγκος επικοινωνίας που υπολογίζεται από τη φόρμουλα (3.2) είναι $V_{comm} = 20$ και το κάθε μέγεθος δεδομένων είναι b = 4bytes (float). Συνεπώς έχουμε $T_{comp} = gt_c = 100t_c$, $T_{comm} = T_{startup}+T_{transmit}$. Έχουμε δύο αρχικές καθυστερήσεις, μία για κάθε αποστολή και λήψη που εκτελείται, συνεπώς $T_{startup} = 2 \times t_s = 200t_c$ και $T = = bV_{comm}t_t = 20 \times 4 \times 0{,}8t_c$. Επιλέγουμε ως βέλτιστα τετράγωνα tile με πλευρά μήκους 10 ($H = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$). Ο χώρος των tile θα είναι επομένως:

$J^S = \{(i_1^S, i_2^S) : 0 \leq i_1^S \leq 999,\ 0 \leq i_2^S \leq 99\}$. Καθώς η μέγιστη τιμή που μπορεί να πάρει το i_1^S είναι 999, που είναι μεγαλύτερη από τη μέγιστη τιμή του i_2^S, αντιστοιχούμε τα tiles σε κάθε επεξεργαστή κατά μήκος του i_1^S. Το βέλτιστο διάνυσμα δρομολόγησης για αυτό τον αλγόριθμο είναι Π = (1,1) και το μήκος δρομολόγησης είναι P = Π(999,99) – Π(0,0) + 1 = 999 + 99 + 1 = 1099. Ο ολικός χρόνος εκτέλεσης που δίνεται από τη σχέση (4.1) είναι T = $1099(100t_c + 200t_c + 20x4x0{,}8t_c) = 1099x364t_c = 400036t_c = 0.4$secs. Σε αυτό το παράδειγμα, υποθέτουμε ότι το $T_{trasmit}$ είναι ο ολικός χρόνος μετάδοσης για ένα πλήρες ζεύγος αποστολής-λήψης. Θα μπορούσαμε να το είχαμε μοιράζει σε δύο κομμάτια, χωρίς να είχε, όμως, καμία επίδραση στα αποτελέσματα.

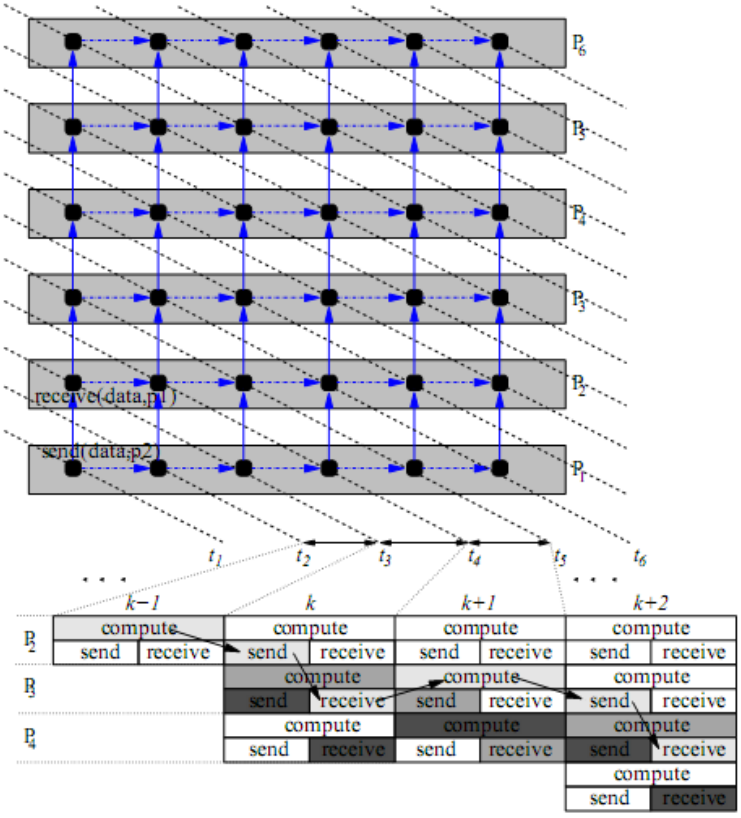
4.3 Δρομολόγηση με Επικάλυψη

Το γραμμικό πρόγραμμα δρομολόγησης που παρουσιάστηκε στην προηγούμενη παράγραφο πετυχαίνει μία μέτρια αξιοποίηση του επεξεργαστή. Όλοι οι επεξεργαστικοί κόμβοι ταυτόχρονα είτε εκτελούν υπολογισμούς είτε επικοινωνούν τα αποτελέσματα με τους γείτονές τους. Ωστόσο, αυτό που προκαλεί την τόσο ανεπαρκή αξιοποίηση της υπολογιστικής ισχύος, είναι οι ροές δεδομένων μεταξύ των διαδοχικών χρονικών βημάτων. Συγκεκριμένα, φαίνεται ότι τα υποστάδια υπολογισμού και επικοινωνίας κάθε χρονικού βήματος θα έπρεπε να σειριοποιηθούν για να διατηρηθεί η σωστή σειρά εκτέλεσης. Κάθε επεξεργαστής πρέπει πρώρα να λαμβάνει δεδομένα, στη συνέχεια να εκτελεί υπολογισμό και τέλος να αποστέλλει τα αποτελέσματα που θα χρησιμοποιηθούν στο επόμενο χρονικό βήμα από τους γειτονικούς του κόμβους (Σχήμα 4.3).

Θα ήταν ιδανικό εάν ένας κόμβος είχε τη δυνατότητα να λαμβάνει, να επεξεργάζεται και να αποστέλλει τα δεδομένα ταυτόχρονα. Οι σύγχρονοι υπολογιστές έχουν μηχανές DMA και διεπαφές δικτύου (network interfaces NICs) που μπορούν να λειτουργούν παράλληλα με τη CPU. Αυτό σημαίνει ότι κάποιος επικοινωνιακός φόρτος μπορεί να επικαλύπτεται με πραγματικούς κύκλους ρολογιού της CPU. Επιπλέον, non blocking πρωτογενείς κλήσεις μεταφοράς μηνυμάτων ανακουφίζουν τον επεξεργαστή από τις αναμονές για την ολοκλήρωση των αντίστοιχων λειτουργιών επικοινωνίας. Πράγματι, παρόλο που κάποιος φόρτος non blocking επικοινωνίας επιβαρύνει ακόμη τη CPU, το μεγαλύτερο μέρος του kernel buffering (TCP/IP stacks) και η φάση επικοινωνίας μπορούν να επικαλυφθούν ιδανικά με την εκτέλεση άλλων χρήσιμων υπολογισμών. Μία πιο εκτενής ματιά στη σωστή ροή δεδομένων (dataflow) στην περίπτωση της δρομολόγησης χωρίς επικάλυψη, φανερώνει την ακόλουθη ενδιαφέρουσα ιδιότητα: εάν τροποποιήσουμε ελάχιστα την αρχική γραμμική δρομολόγηση μπορούμε να επικαλύψουμε κάποιο χρόνο επικοινωνίας με εκτέλεση

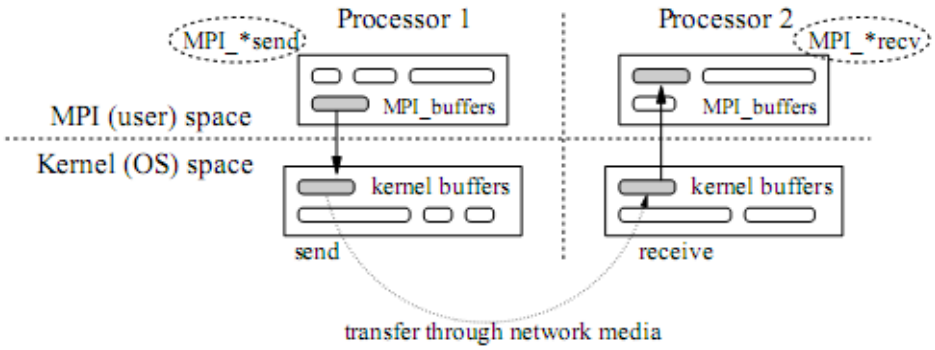
υπολογισμών. Αυτό σημαίνει ότι, σε κάθε χρονικό βήμα, ο επεξεργαστής πρέπει να αποστέλλει και να λαμβάνει δεδομένα που δεν είναι άμεσα εξαρτημένα με τα δεδομένα που επεξεργάζεται τη δεδομένη χρονική στιγμή. Ένα έγκυρο σχήμα εκτέλεσης θα ήταν το εξής: ο επεξεργαστής λαμβάνει δεδομένα από όλους τους γείτονές του για να τα χρησιμοποιήσει στο χρονικό βήμα *k*+1, αποστέλλει δεδομένα που υπολογίστηκαν το προηγούμενο χρονικό βήμα (*k*-1) και υπολογίζει τα αποτελέσματα του τρέχοντος χρονικού βήματος *k* (Σχήματα 4.3 και 4.2).

Στο [19] έχει παρουσιαστεί ένα γραμμικό υπερεπίπεδο για τη βέλτιστη χρονοδρομολόγηση grid task γράφων Unit Execution Times-Unit Communication Times. Οι γράφοι πλέγματος (grid) μοιάζουν με χώρους επαναλήψεων με μοναδιαία διανύσματα εξάρτησης. Θεωρώντας το μοντέλο UET-UCT, είναι σαν να έχεις φάσεις επικοινωνίας που χρειάζονται ίσο χρόνο με τις φάσεις υπολογισμού. Στο [19], έχει επίσης αποδειχτεί ότι η βέλτιστη χωρική δρομολόγηση για UET-UCT είναι να αναθέσεις όλα τα σημεία κατά μήκος της μέγιστης διάστασης στον ίδιο επεξεργαστή. Η αναλογία της απαίτησης ίσων χρόνων επικοινωνίας και υπολογισμών με την περίπτωση που εξετάζουμε είναι προφανής. Εάν μπορούσαμε να πετύχουμε μία λεπτότητα επεξεργασίας προς επικοινωνία *g*, τέτοια ώστε ο χρόνος που απαιτείται για την επικοινωνία με τους γειτονικούς κόμβους να είναι ίσος με το χρόνο που απαιτείται από τη CPU για υπολογισμό, τότε θα μπορούσαμε να εφαρμόσουμε αυτή την ελαφρά τροποποιημένη γραμμική δρομολόγηση και την αντίστοιχη χωρική δρομολόγηση. Η βέλτιστη χρονοδρομολόγηση για χώρο tile $j^S(j_1^S, j_2^S, ..., j_n^S)$ σε αυτή την περίπτωση είναι $2j_1^S + 2j_2^S + ... + 2j_{i-1}^S + 2j_{i+1}^S + ... + 2j_n^S + j_i^S$, όπου i είναι η διάσταση κατά μήκος της οποίας τα tiles έχουν αντιστοιχιστεί στον ίδιο επεξεργαστή.



Σχήμα 4.2: Χρονοδρομολόγηση με επικάλυψη

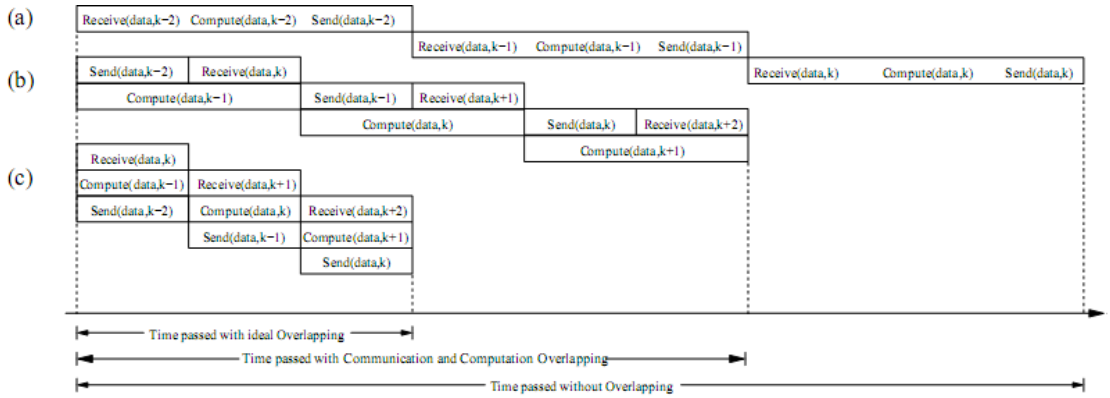
Στο Σχήμα 4.2 παρουσιάζεται η δρομολόγηση με επικάλυψη. Θεωρείστε, για παράδειγμα, τον επεξεργαστή P_3 τη χρονική στιγμή *k*: καθώς εκτελεί τον υπολογισμό για ένα tile, παράλληλα εκτελεί τα ακόλουθα: αποστέλλει τα αποτελέσματα που παράχθηκαν τη χρονική στιγμή *k*-1 και λαμβάνει δεδομένα από τους γείτονές του, τα οποία θα χρησιμοποιηθούν κατά τον υπολογισμό του επόμενου tile τη χρονική στιγμή *k*+1. Παρατηρήστε τα βέλη του Σχήματος 4.2. Αναπαριστούν την πραγματική ροή δεδομένων μεταξύ διαδοχικών χρονικών στιγμών (υπολογισμοί-αποστολές-λείψεις) σε μορφή διασωλήνωσης. Το αποτέλεσμα αυτής της δρομολόγησης είναι να δίνει τη δυνατότητα σε διαδοχικούς υπολογισμούς να επικαλύπτονται με φάσεις επικοινωνίας, συνεπώς να επιτυγχάνεται θεωρητικά αξιοποίηση του επεξεργαστή στο 100%.



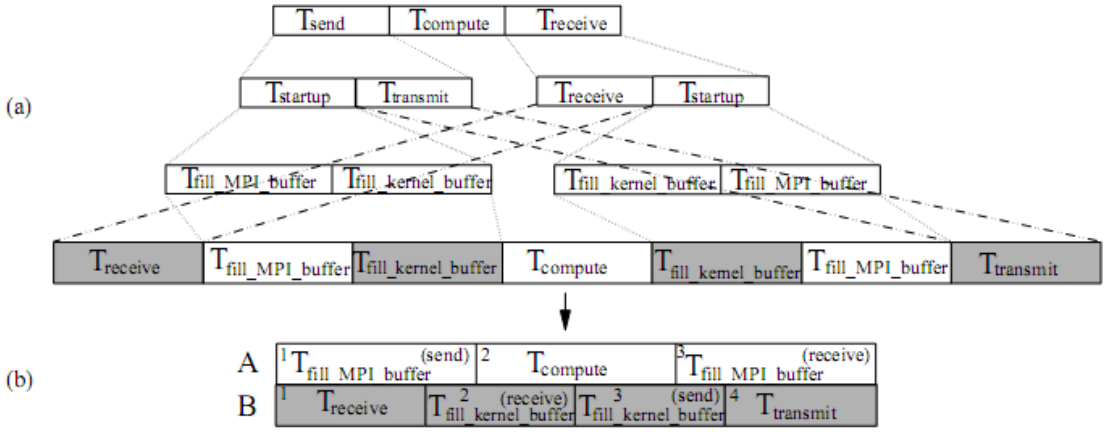
Σχήμα 4.5: MPI space (χρήστη) και kernel (OS) space

4.3.1 Εφαρμογή σε ένα Περιβάλλον Ανταλλαγής Μηνυμάτων

Σε ένα περιβάλλον ανταλλαγής μηνυμάτων, όπως το MPI, ένας επεξεργαστής πρώτα ξεκινά όλες τις λειτουργίες non-blocking λήψης, στη συνέχεια εκτελεί τον ατομικό υπολογισμό ενός tile και τέλος δρομολογεί όλες τις non-blocking λειτουργίες αποστολής. Καθώς εκτελεί τα στιγμιότυπα βρόχου υπολογισμού των tile, μπορεί να λαμβάνει δεδομένα από τους γείτονές του και να αποστέλλει δεδομένα που υπολογίστηκαν προηγουμένως. Εφόσον όλες οι πρωτογενείς κλήσεις είναι non-blocking, η έκδοση μίας κλήσης αποστολής, για παράδειγμα, απαιτεί την προσοχή του επεξεργαστή μόνο για να γεμίσει το MPI buffer αποστολής του συστήματος. Αφού αυτό ολοκληρωθεί, ο έλεγχος επιστρέφει στον επεξεργαστή ο οποίος εκτελεί το υπόλοιπο του προγράμματος, που είναι ο υπολογισμός του tile. Τα ίδια ισχύουν στην περίπτωση της κλήσης για λήψη δεδομένων. Όταν εκδοθεί μία εντολή receive, προετοιμάζεται ένα MPI buffer λήψης και ο έλεγχος επιστρέφει αμέσως στο πρόγραμμα για να συνεχίσει την εκτέλεση της επόμενής του εντολής. Από τη στιγμή που το μήνυμα φτάσει στο kernel, αντιγράφεται από το kernel buffer στο buffer λήψης και στη συνέχεια είναι έτοιμο να χρησιμοποιηθεί (Σχήμα 4.5). Στην πράξη, εφόσον τα υποκείμενα στρώματα λαμβάνουν το μήνυμα πριν την ακριβή κλήση της receive στο πρόγραμμα χρήστη, τοποθετούμε όλες τις receive στο τέλος κάθε τρίπτυχου send-compute-receive. Με παρόμοιο τρόπο, εκτελούμε όλα τα send στην αρχή της παραπάνω τριάδας, έτσι ώστε όλα τα send να ξεκινούν όσο πιο σύντομα γίνεται.



Σχήμα 4.3: Διάφορα επίπεδα επικάλυψης επεξεργασίας με επικοινωνία



Σχήμα 4.4: Ανάλυση ενός χρονικού βήματος

Φαίνεται ότι η αρχική προετοιμασία και των δύο MPI system buffer (receive και send) είναι ένα αναπόφευκτο υπολογιστικό κόστος. Ωστόσο, με τη βοήθεια μίας μηχανής DMA, ή κάποια άλλης υπολογιστικής μονάδας (π.χ. λογικός επεξεργαστής Hyper-Threading), η αντιγραφή των δεδομένων που θα σταλούν στο kernel buffer, ή η αντιγραφή των δεδομένων που λήφθηκαν στο MPI buffer λήψης από το kernel buffer μπορούν να επικαλυφθούν με υπολογισμούς. Επιπλέον, για μεγάλα μηνύματα, ο χρόνος μετάδοσης είναι επίσης αυξημένος, που μπορεί και αυτός να επικαλυφθεί. Ένα ιδανικό σχήμα εμφανίζεται στο Σχήμα 4.3b και η αντίστοιχη ανάλυση για κάθε χρονικό βήμα στο Σχήμα 4.4b. Εάν υποθέσουμε ιδανική επικάλυψη και των send με τα receive (π.χ. υποστήριξη DMA για multichannel I/O), τότε στο Σχήμα 4.3c φαίνεται η επιτυγχόμενη συμπίεση χρόνου.

Σύμφωνα με τα παραπάνω, έχουμε:

$$T = P(g)\max(A_1+A_2+A_3, B_1+B_2+B_3+B_4) \qquad (4.2)$$

Εφόσον το χρονικό υπερεπίπεδο είναι τέτοιο, ώστε επικρατεί είτε η εκτέλεση υπολογισμών είτε η επικαλυπτόμενη επικοινωνία. Όπως φαίνεται στο Σχήμα 4.4.

A_1: ο χρόνος που χρειάζεται το σύστημα MPI να γεμίσει το MPI buffer για τη λειτουργία αποστολής ($T^{(send)}_{fill\_MPI\_buffer}$),

A_2: ο χρόνος υπολογισμού ($T_{compute}$),

A_3: ο χρόνος που χρειάζεται το σύστημα MPI να γεμίσει το MPI buffer για τη λειτουργία λήψης ($T^{(receive)}_{fill\_MPI\_buffer}$),

B_1: ο χρόνος για λήψη των δεδομένων (πλευρά λήψης) ($T_{receive}$),

B_2: ο χρόνος που χρειάζεται το OS kernel να γεμίσει ένα kernel buffer για τη λειτουργία λήψης ($T^{(receive)}_{fill\_kernel\_buffer}$),

B_3: ο χρόνος που χρειάζεται το OS kernel να γεμίσει ένα kernel buffer για τη λειτουργία αποστολής ($T^{(send)}_{fill\_kernel\_buffer}$),

B_4: ο χρόνος που χρειάζεται για μετάδοση των δεδομένων ($T_{transmit}$).

Υποθέτουμε ότι η ολικός χρόνος μετάδοσης χωρίζεται σε δύο κομμάτια, το χρόνο διάδοσης της πλευράς αποστολέα και το χρόνο διάδοσης της πλευράς του παραλήπτη, B_4 και B_1, αντίστοιχα. Στο [17] αποδείχτηκε ότι όλα τα A_i, B_i εξαρτώνται από το μέγεθος g, και συνεπώς $A_i(g)$, $B_i(g)$. Στη περίπτωση με επικάλυψη, το βέλτιστο P(g) δίνεται από την έκφραση $2u_1^S + 2u_2^S + ... + 2u_{i-1}^S + 2u_{i+1}^S + ... + 2u_n^S + u_i^S$, όπου $(u_1^S, u_2^S, ..., u_n^S)$ οι συντεταγμένες του "τελευταίου" tile στο χώρο J^S, θεωρώντας ότι (0,0,…,0) είναι οι συντεταγμένες του πρώτου tile και ότι η μεγαλύτερη διάσταση είναι η i (δες το [19]). Έχουμε τις ακόλουθες δύο περιπτώσεις:

1. Ο μη αποφευκτός χρόνος έναρξης για όλα τα send και receive συν ο καθαρός χρόνος υπολογισμού είναι μεγαλύτερος από το χρόνο που χρειάζεται για την υπόλοιπη επικοινωνία και διάδοση:

   Εάν $A_1+A_2+A_3 > B_1+B_2+B_3+B_4$ τότε η (4.2) γίνεται:

   $$T = P(g)(A_1+A_2+A_3) \qquad (4.3)$$

   Από το λήμμα 1 στο [18], προκύπτει ότι $P(g) = P_0 g^{-1/n}$, συνεπώς έχουμε $T = P_0 g^{-1/n}(A_1 + A_3 + gt_c) \Rightarrow T = P_0(A_1(g) + A_3(g))g^{-1/n} + P_0 t_c g^{\frac{n-1}{n}}$. Παίρνουμε το βέλτιστο ολικό χρόνο όταν $T'(g) = 0$. Όσο για το g χρησιμοποιούμε πειραματικές τιμές, εφόσον δεν υπάρχει αναλυτική φόρμουλα για τις συναρτήσεις $A_1(g)$, $A_2(g)$.

2. Ο μη αποφευκτός χρόνος έναρξης για όλα τα send και receive συν ο καθαρός χρόνος υπολογισμού είναι μικρότερος από το χρόνο που χρειάζεται για την υπόλοιπη επικοινωνία και διάδοση:

   Εάν $A_1+A_2+A_3 < B_1+B_2+B_3+B_4$ τότε η (4.2) γίνεται:

   $$T = P(g)(B_1+B_2+B_3+B_4)$$

   Όπως και στο [18], ο χρόνος μετάδοσης είναι $B_1 = B_4 = bt_t V_0 g^{\frac{n-1}{n}}$, συνεπώς έχουμε

   $T = P_0 g^{-1/n}(B_2(g) + B_3(g) + 2bt_t V_0 g^{\frac{n-1}{n}}) \Rightarrow T = P_0(B_2(g) + B_3(g))g^{-1/n}$

   $+2P_0 bt_t V_0 g^{\frac{n-2}{n}}$. Παίρνουμε το βέλτιστο ολικό χρόνο όταν $T'(g) = 0$.

Παράδειγμα 4.2

Το διασωληνωμένο dataflow στην περίπτωση με επικάλυψη δουλεύει ως εξής (Σχήμα 4.2): Τα δεδομένα που έχουν υπολογιστεί από τον επεξεργαστή P_2 τη χρονική στιγμή k-1, αποστέλλονται στον επεξεργαστή P_3 κατά τη χρονική στιγμή k, λαμβάνονται από τον P_3 κατά την ίδια χρονική στιγμή k, και μετά εκτελείται υπολογισμός σε αυτά τα δεδομένα τη χρονική στιγμή k+1, από τον ίδιο επεξεργαστή. Στη συνέχεια, τη χρονική στιγμή k+2, ο επεξεργαστής P_3 αποστέλλει τα αποτελέσματα που υπολογίστηκαν την προηγούμενη χρονική στιγμή στον P_4, για να ληφθούν μέχρι το τέλος της χρονικής στιγμής k+2.

Παράδειγμα 4.3

Θεωρήστε τον αλγόριθμο στην παράγραφο 3.3 με βέλτιστο διάνυσμα δρομολόγησης να είναι όμως το (1,2). Όπως έχουν δείξει πειραματικές τιμές [17], μία ρεαλιστική υπόθεση μπορεί να είναι η ακόλουθη: με $\mathrm{T}_{fill\_MPI\_buffer} = 1/2\, t_s$, και $\mathrm{T}_{fill\_MPI\_buffer} + \mathrm{T}_{fill\_\mathrm{ker}nel\_buffer} = \mathrm{T}_{startup}$ έχουμε ένα send και ένα receive σε κάθε χρονική επανάληψη κατά μήκος της διάστασης i_2. Το μήκος δρομολόγησης τώρα είναι P = Π(999,99) - Π(0,0) + 1 = 999 + 2x99 + 1 = 1198. Αφού ισχύει $B_1+B_2+B_3+B_4 = 50t_c+50t_c+20x0.4x0.8t_c < A1+A_2+A_3 = 50t_c+50t_c+100t_c$, ο ολικός χρόνος εκτέλεσης είναι σε αυτή την περίπτωση $1198(25t_c+25t_c+100t_c)=179700tc=0.24secs$, πολύ λιγότερος παρά την περίπτωση χωρίς επικάλυψη (0.4*secs*). Εάν ρυθμίσουμε το g έτσι ώστε $A_1+A_2+A_3 = B_1+B_2+B_3+B_4$, έχουμε δηλαδή πλήρη επικάλυψη, θα μπορούσαμε να πετύχουμε πολύ καλύτερα αποτελέσματα. Είναι προφανές ότι μία μεγαλύτερη τιμή του g θα αύξανε το μέγεθος των δεδομένων που χρειάζεται να ανταλλαχθούν και θα μείωνε τον αριθμό των υπερεπιπέδων $P(g)$, ταυτόχρονα μειώνοντας τον τελεστή gt_c.

Κεφάλαιο 5

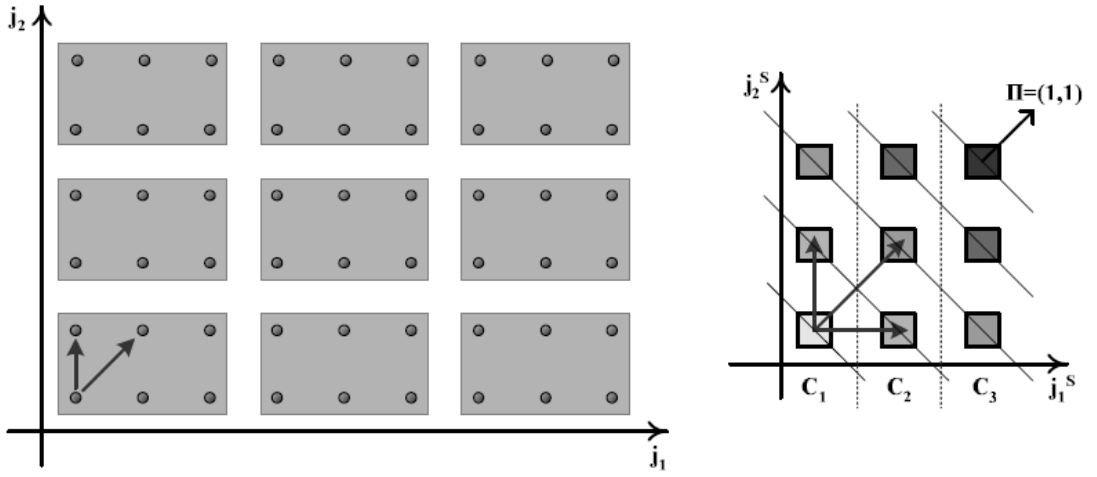
Προσομοίωση Εξίσωσης Διάχυσης με Χρήση Επικαλυπτόμενης Δρομολόγησης

5.1 Γενικά

Σε αυτό το κεφάλαιο παρουσιάζουμε την υλοποίηση μιας προσομοίωσης διακριτού υπολογισμού της εξίσωσης διάχυσης, όπως αυτή παρουσιάζεται στο Αλγοριθμικό Μοντέλο του Κεφαλαίου 3 (Αλγόριθμος 2 στην παράγραφο 3.3.1). Η εξίσωση αυτή παρουσιάζει μεγάλο επιστημονικό ενδιαφέρον καθώς εφαρμόζεται στα εξής προβλήματα: διάδοση θερμότητας σε μέσο με σταθερό συντελεστή θερμικής αγωγιμότητας, διάδοση ενός υγρού σε ισότροπο μέσο, διάδοση της στροβιλότητας σε μέσο, ο ρυθμός μεταβολής του αριθμού των νετρονίων σε ραδιενεργό υλικό (πυρηνικό αντιδραστήρα), κ.ά. Συνήθως τα προβλήματα αυτά χειρίζονται μεγάλο όγκο δεδομένων με πραγματικές τιμές όπου η ακρίβεια παίζει σημαντικό ρόλο. Η προσομοίωση τέτοιων φυσικών φαινομένων με εμπειρικά μέσα είναι αδύνατη και σαν μόνη δυνατή λύση παρουσιάζεται η προσομοίωσή τους με χρήση υπολογιστών. Η υπολογιστική δυνατότητα και οι αποθηκευτικές ανάγκες που χρειάζονται για μια κατάλληλη προσομοίωση τέτοιων προβλημάτων είναι τεράστια και τα μονο-επεξεργαστικά συστήματα μοιάζουν ανήμπορα να σηκώσουν ένα τέτοιο βάρος. Η χρήση των παράλληλων συστημάτων επεξεργασίας σε τέτοια συστήματα μοιάζει επιτακτική για να έχουμε αποτελέσματα μέσα σε λογικά χρονικά περιθώρια.

Σκοπός μας είναι να εκμεταλλευτούμε την πολυνηματική δυνατότητα που παρέχει η τεχνολογία Hyper-Threading με χρήση της επικαλυπτόμενης δρομολόγησης που παρουσιάζεται στην παράγραφο 4.3 μέσα στα πλαίσια μιας προσομοίωσης της εξίσωσης διάχυσης. Η προσέγγισή μας ακολουθεί την εξής πορεία: αρχικά υλοποιούμε ένα γραμμικό πρόγραμμα προσομοίωσης του αλγορίθμου για σημείο αναφοράς όσο αφορά το speedup των παράλληλων εκδόσεων και υποστηρίζει μεταβλητό χωρίο εισόδου. Σαν δεύτερο στάδιο έχουμε την υλοποίηση ενός απλού (χωρίς καμία βελτιστοποίηση) παράλληλου προγράμματος σε γλώσσα προγραμματισμού C με χρήση βιβλιοθηκών MPI που υποστηρίζει μεταβλητή τοπολογία επεξεργαστών και χωρίο εισόδου. Στη συνέχεια προσθέτουμε τροποποιήσεις στο πρόγραμμα του βήματος δύο (βασική έκδοση) για να δούμε τις επιδράσεις τους στην επίδοση. Η τρίτη μας υλοποίηση προσθέτει στη βασική έκδοση δυνατότητες tiling στα πλαίσια της παραγράφου 3.4. Στην τέταρτη και πέμπτη υλοποίηση προσθέτουμε επικαλυπτόμενη

δρομολόγηση, όπως αυτή παρουσιάζεται στα πλαίσια της παραγράφου 4.3. χωρίς και με tiling, αντίστοιχα (με είσοδο μεταβλητό μέγεθος tile). Στις τελευταίες δύο υλοποιήσεις σπάμε κάθε διεργασία MPI σε δύο νήματα, στην μία περίπτωση συμμετρικά και στην άλλη ασύμμετρα τα οποία θέλουμε να τρέχουν το καθένα στον ένα από τους δύο λογικούς επεξεργαστές ενός φυσικού επεξεργαστή με τεχνολογία Hyper-Threading. Καθεμία από τις δύο τελευταίες υλοποιήσεις διατίθεται με και χωρίς tiling. Να αναφέρουμε ότι όλες οι υλοποιήσεις έχουν φτιαχτεί για 1+1, 2+1 και 3+1 (χωρικές + χρονική) διαστάσεις. Να σημειώσουμε ότι τα προγράμματα αυτά με μικρή τροποποίηση μπορούν να εφαρμοστούν και σε υπολογισμούς άλλων εξισώσεων με απλή τροποποίηση του κυρίως βρόχου υπολογισμού. Παρουσιάζουμε στη συνέχεια τους τρόπους υλοποίησης των προσομοιώσεών μας.



Σχήμα 5.1: Παράδειγμα διαμέρισης χωρίου, tiling, και δρομολόγησης για εξίσωση διάχυσης 1 χωρικής διάστασης (+1 χρονικής, j_2).

5.2 Σειριακή Έκδοση

Η σειριακή έκδοση του προγράμματος προσομοίωσης αποτελεί απλή υλοποίηση του αλγορίθμου 2 της υποπαραγράφου 3.3.1. Το πρόγραμμα αυτό είναι πλήρως σειριακό και φτιάχτηκε για να αποτελεί σημείο αναφοράς επίδοσης στο speedup των παράλληλων προγραμμάτων. Η υλοποίηση δίνεται στον παρακάτω ψευδοκώδικα, όπου παρουσιάζουμε μόνο τον κυρίως βρόχο για απλότητα:

Ψευδοκώδικας 5.1: Σειριακό Πρόγραμμα 2+1 Διαστάσεων

```
1    while (loops < MAXLOOPS)
2    {
3      t = (++loops)%2;
4      /*Υπολογισμός διακριτής εξίσωσης διάχυσης*/
5      for(y = 1; y < Y; y++)
6        for(x = 1; x < X; x++)
7          U[t][y][x] = (1+2*a*dt/dx)*U[(t-1)%2][y][x] -
                        a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
8    }
```

Να σημειώσουμε ότι στη χρονική διάσταση αποθηκεύουμε μόνο την τρέχουσα και προηγούμενη χρονική στιγμή. Κάτι τέτοιο κρίθηκε απαραίτητο για να δίνεται η δυνατότητα επεξεργασίας όσο το δυνατό μεγαλύτερων χωρίων και εφαρμόζεται και στις παράλληλες υλοποιήσεις που παρουσιάζονται στη συνέχεια. Ο πυρήνας του ψευδοκώδικα 5.1 θα χρησιμοποιείται στη συνέχεια στα παράλληλα προγράμματα ως συνάρτηση *compute()*.

5.3 Βασική Παράλληλη Έκδοση MPI

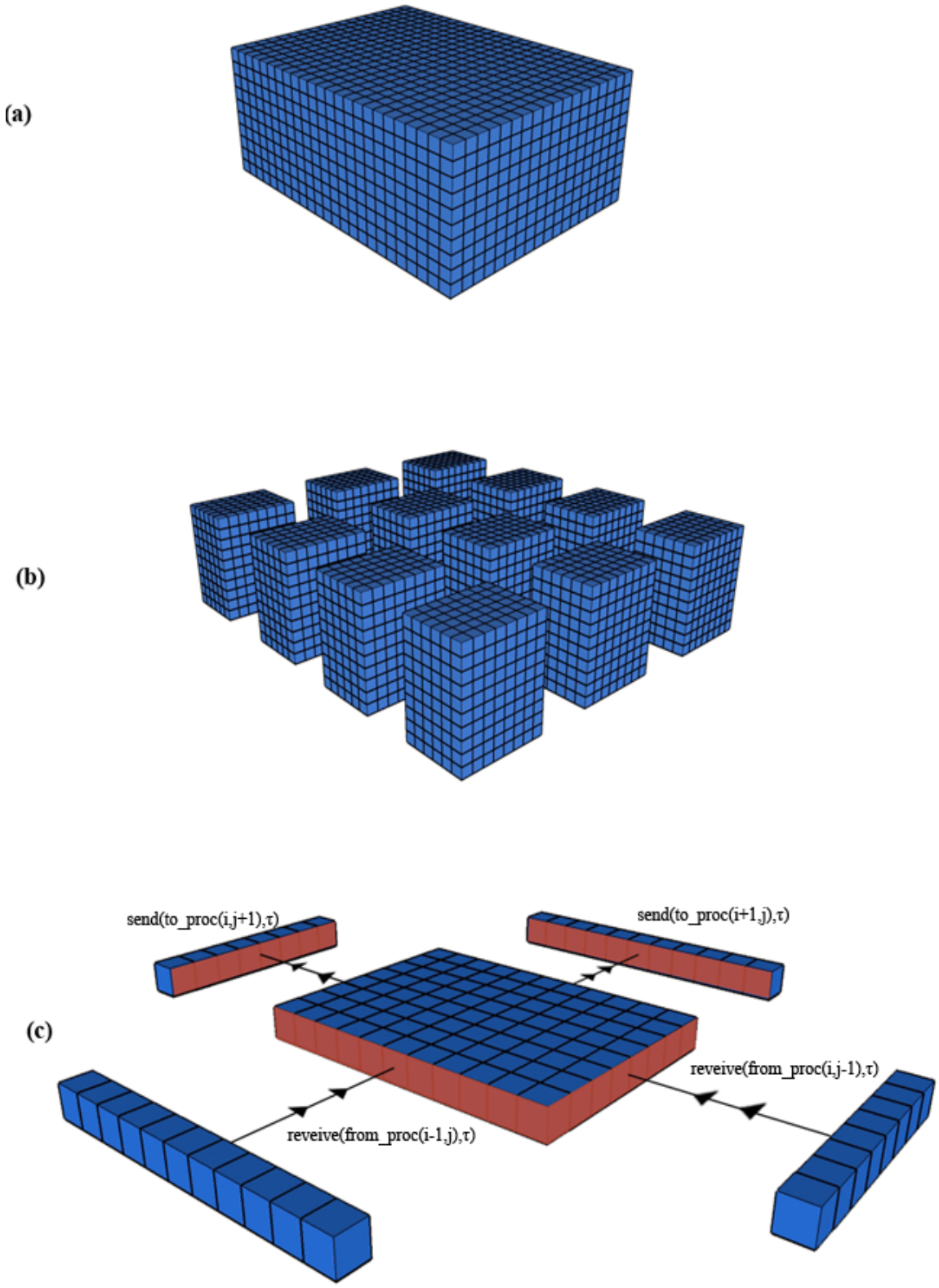
Για να μπορέσουμε να εκμεταλλευτούμε ένα παράλληλο σύστημα επεξεργασίας πρέπει να παράγουμε παράλληλα προγράμματα, δηλαδή προγράμματα βασισμένα σε παράλληλο αλγόριθμο. Όπως αναφέραμε και στην υποπαράγραφο 1.2.1(νόμος του Amdahl για παράλληλα συστήματα [3]), για να πετύχουμε κάτι τέτοιο πρέπει να το πρόγραμμα που θέλουμε να παραλληλοποιήσουμε να έχει σημαντικό κομμάτι του που να επιδέχεται παραλληλοποίησης. Στην περίπτωσή μας το μέρος του αλγορίθμου με το μεγαλύτερο υπολογιστικό κόστος αποτελεί ο κυρίως βρόχος υπολογισμού, ο οποίος επιδέχεται παραλληλισμού αυξανόμενης κλίμακας όσο αυξάνει το χωρίο προσομοίωσης. Με αυτά κατά νου παραλληλοποιούμε τον βασικό Αλγόριθμο 2 ως εξής: μοιράζουμε σε καθένα από τους επεξεργαστές της τοπολογίας ένα κομμάτι του πίνακα υπολογισμών U πάνω στον οποίο θα εκτελούν υπολογισμούς. Τα διανύσματα εξάρτησης που δημιουργούνται μεταξύ των γειτονικών επεξεργαστών επιβάλλουν επικοινωνία μεταξύ τους, σύμφωνα με την παράγραφο 3.4. Συγκεκριμένα πρέπει να αποστέλλονται και να λαμβάνονται σύνορα κατά μήκος της κάθε διάστασης εκεί που συνορεύουν γειτονικοί κόμβοι. Λόγω της φύσης της εξίσωσης διάχυσης, οι εξαρτήσεις είναι της εξής μορφής: ο κάθε επεξεργαστικός κόμβος έχει διάνυσμα εξάρτησης με την προηγούμενη χρονική διάσταση (που κρατά ο ίδιος, καμία επικοινωνία) και διανύσματα εξάρτησης κάθετα στις χωρικές διαστάσεις και με φορά προς τα πίσω (θεωρώντας ορθοκανονικό σύστημα συντεταγμένων). Λαμβάνει δηλαδή δεδομένα από γειτονικούς κόμβους προς τα πίσω και αποστέλλει δεδομένα προς τα εμπρός.

Η πρώτη αυτή βασική υλοποίηση αποτελεί λεπτομερή παραλληλισμό (fine grain parallelism) και διαμερίζει τον $(n+1)$-διάστατο χώρο επαναλήψεων $u_1 \times u_2 \times ... \times u_{n+1}$ σε μία n-διάστατη τοπολογία επεξεργαστών $C_1 \times C_2 \times ... \times C_n = C$, με αποτέλεσμα η διάσταση u_i να τεμαχίζεται πρακτικά σε C_i κομμάτια. Αποτελεί λεπτομερή παραλληλισμό γιατί το στοιχειώδες κομμάτι (ή tile) στο οποίο σε κάθε βρόχο εκτελεί ο κάθε επεξεργαστής υπολογισμούς και για το οποίο αποστέλλει και λαμβάνει δεδομένα έχει χρονική διάσταση μεγέθους 1. Να σημειώσουμε ότι η τοπολογία των επεξεργαστών είναι ένα μεταβλητό μέγεθος που δίνεται σαν είσοδος στο πρόγραμμά μας.

Παράδειγμα 5.1

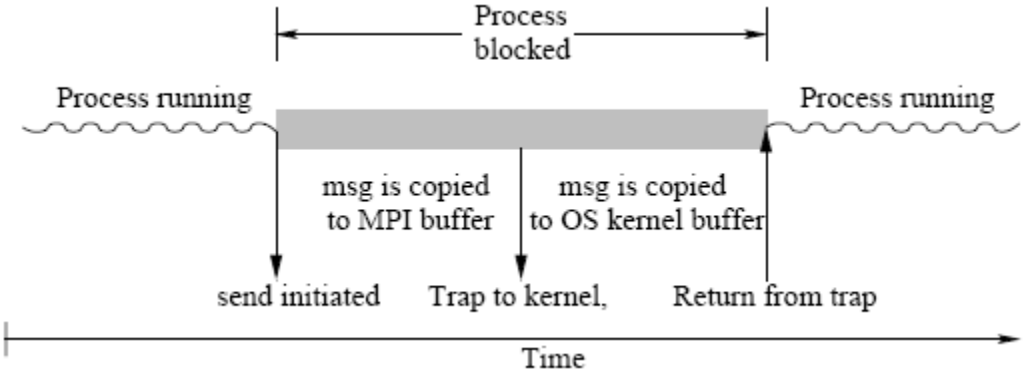
Έστω έχουμε ως είσοδο τον εξής χώρο επαναλήψεων 2 χωρικών διαστάσεων: $J^{2+1} = \{(j_1, j_2, j_3) : 0 \le j_1 \le 2999, 0 \le j_2 \le 999, 0 \le j_3 \le 999\}$, δηλαδή χωρίο 3000x1000 για 1000 επαναλήψεις, και τοπολογία 3×4 ($C_1 = 3, C_2 = 4$). Τότε ο καθένας από τους επεξεργαστές θα πάρει ένα μέρος του αρχικού χώρου επαναλήψεων, μεγέθους 1000x250 x 1000 επαναλήψεις. Στην περίπτωση αυτή ο πίνακας διανυσμάτων εξάρτησης είναι ο εξής: $D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, που προκύπτει από την εξίσωση υπολογισμού του Αλγορίθμου 2 και αντιστοιχεί σε διανύσματα εξάρτησης με τους κόμβους που βρίσκονται μία θέση προς τα πίσω σε κάθε διάσταση, εάν θεωρήσουμε ορθοκανονικό σύστημα συντεταγμένων.

Η συζήτησή μας αυτή παρουσιάζεται σχηματικά για τις 2+1 (χωρικές + χρονική) διαστάσεις στο Σχήμα 5.2a,b και c. Στο Σχήμα 5.2.a παρουσιάζεται το ολικό χωρίο επαναλήψεων που πρόκειται να διαμεριστεί μεταξύ των υπολογιστών. Στο 5.2b δείχνουμε το διαμερισμένο χωρίο σε μια τοπολογία 3x4 ($C_1 = 3, C_2 = 4$) επεξεργαστών, και τέλος στο 5.2c φαίνεται το στοιχειώδες κομμάτι υπολογισμού και η επικοινωνία μεταξύ γειτονικών επεξεργαστών. Η ροή του υπολογισμού ξεκινά από τον κόμβο που βρίσκεται στην αρχή των αξόνων και επεκτείνεται ακτινικά κατά μήκος των χωρικών και της χρονικής διάστασης σε αύξουσα σειρά. Υπάρχει συνεπώς μία αρχική καθυστέρηση σε σχέση με την έναρξη του κάθε κόμβου μέχρι να πάρει τα πρώτα συνοριακά στοιχεία. Αυτή η καθυστέρηση όμως είναι αμελητέα όταν ο χώρος των κοινών δεδομένων είναι πολύ μεγάλος και ο αριθμός των επαναλήψεων του βρόχου είναι επίσης μεγάλος, όπως ισχύει στην περίπτωσή μας.



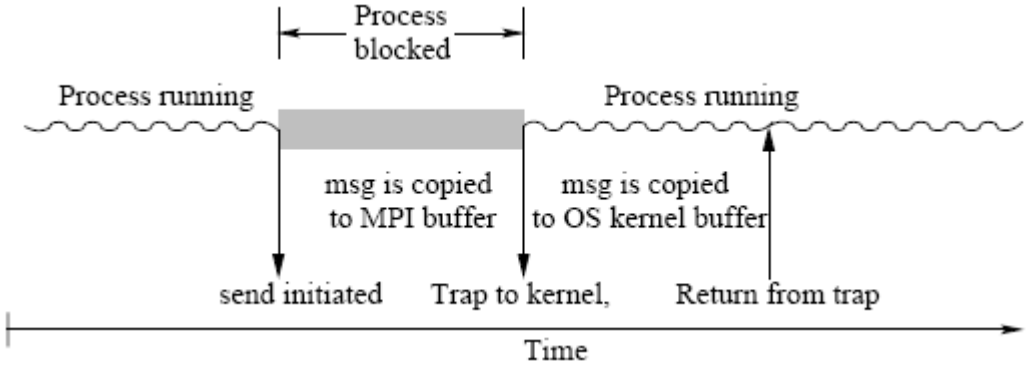
Σχήμα 5.2: (a) Αρχικό χωρίο επαναλήψεων 2 χωρικών διαστάσεων και 1 χρονικής. (b) Παράδειγμα διαμέρισης του χωρίου σε τοπολογία 3x4, (c) το στοιχειώδες κομμάτι στο οποίο εκτελούνται υπολογισμοί και η επικοινωνία σε κάθε βρόχο

Για την υλοποίηση της επικοινωνίας μεταξύ των επεξεργαστών χρησιμοποιούμε το περιβάλλον ανταλλαγής μηνυμάτων MPI. Στα πλαίσια του MPI μας δίνεται η δυνατότητα χρήσης δύο τύπων πρωτογενών κλήσεων επικοινωνίας (send και receive): των τυπικών blocking κλήσεων και των αντίστοιχων non-blocking. Όταν μια διεργασία καλέσει μια ρουτίνα blocking send, καθορίζει ένα buffer και τον προορισμό αποστολής αυτού. Καθώς το μήνυμα αποστέλλεται, η διεργασία που κάλεσε το send μπλοκάρει. Η εντολή που ακολουθεί της send δεν εκτελείται μέχρι το μήνυμα να έχει αποσταλεί πλήρως, όπως φαίνεται στο Σχήμα 5.2. Παρόμοια, μία κλήση blocking receive δεν επιστρέφει τον έλεγχο στην καλούσα διεργασία παρά μόνο όταν το μήνυμα έχει φτάσει και έχει τοποθετηθεί στο buffer μηνύματος, όπως έχει καθοριστεί από την παράμετρο της κλήσης receive.



Σχήμα 5.3: Πρωτογενής blocking κλήση send

Στην εναλλακτική περίπτωση των non-blocking πρωτογενών κλήσεων επικοινωνίας του MPI δεν ισχύουν τα παραπάνω. Εάν το send είναι non-blocking, επιστρέφει τον έλεγχο στην καλούσα διεργασία αμέσως, πριν το μήνυμα αποσταλεί. Το πλεονέκτημα αυτής της μεθόδου (Σχήμα 5.3) είναι ότι η καλούσα διεργασία μπορεί να συνεχίσει την εκτέλεση υπολογισμών παράλληλα με την αποστολή του μηνύματος, αντί ο επεξεργαστή να μπει σε άεργη κατάσταση.



Σχήμα 5.4: Πρωτογενής non-blocking κλήση send

Για τους παραπάνω λόγους επιλέξαμε να χρησιμοποιήσουμε τις πρωτογενείς non-blocking κλήσεις επικοινωνίας στην εφαρμογή μας. Ο ψευδοκώδικας της βασικής παράλληλης έκδοσης για 2+1 διαστάσεις δίνεται στη συνέχεια, όπου θεωρούμε ένα κεντρικό κόμβο που αποστέλλει και λαμβάνει μηνύματα:

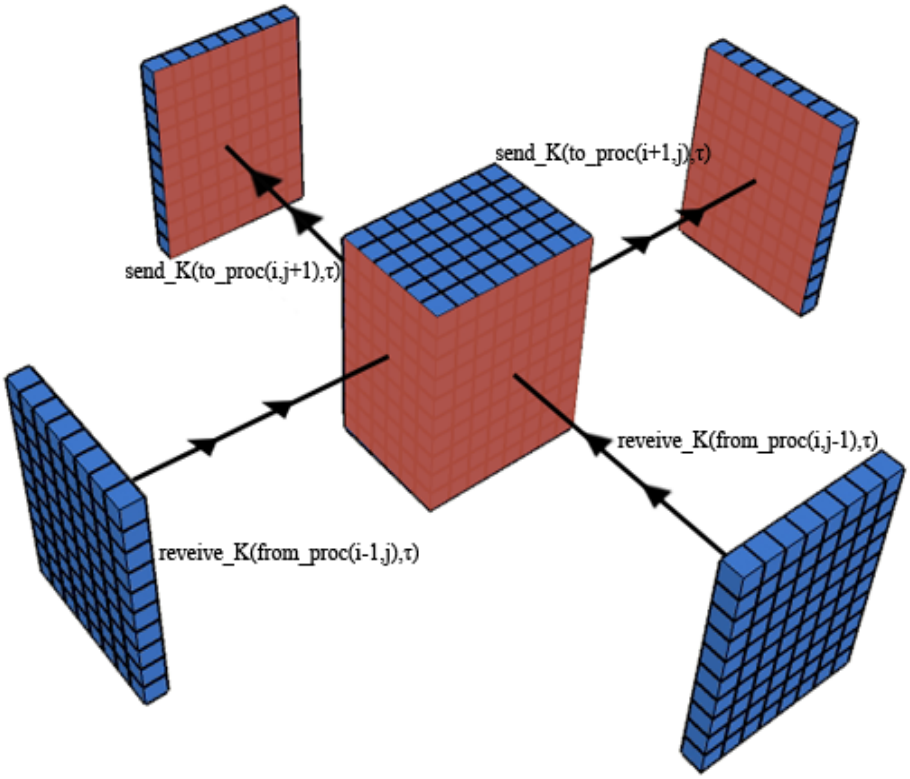
Ψευδοκώδικας 5.2: Βασικό Παράλληλο Πρόγραμμα 2+1 Διαστάσεων

```
1   /*Εκτέλεση απαραίτητων αρχικοποιήσεων, δέσμευση μνήμης, διαμέριση
2     χωρίου επαναλήψεων στις διεργασίες και ορισμός γειτόνων*/
3   Init2D();
4   while (loops < MAXLOOPS)
5   {
6      I_Receive((x-1,y), (x,y-1));
7      Wait();
8      UnpackData();
9      Compute();
10     SetDataToSend(()());
11     I_Send((x+1,y), (x,y+1));
12     Wait();
13  }
```

Το πρόγραμμα παίρνει σαν παραμέτρους το χωρίο επαναλήψεων καθώς επίσης και το μέγεθος του υπολογιστικού πλέγματος στο οποίο θα εκτελεστεί.

5.4 Παράλληλη Έκδοση με Tiling

Στη βασική έκδοση της προηγούμενης παραγράφου προσθέτουμε δυνατότητες tiling μεταβαλλόμενου ύψους (του χρονικού παραθύρου) ανάλογα με την είσοδο του χρήστη, σύμφωνα με την παράγραφο 3.4. Τώρα το πρόγραμμά μας μετατρέπεται σε πρόγραμμα αδρομερούς παραλληλισμού, όπου διαμερίζουμε το χώρο επαναλήψεων J^{n+1} σε ομάδες ύψους > 1 ως προς τη χρονική διάσταση, σχηματίζοντας τον αντίστοιχο tiled χώρο J^S. Έτσι σε κάθε βασικό βρόχο του αλγορίθμου έχουμε αποστολή, λήψη και υπολογισμό σε ομάδες δεδομένων που αντιστοιχούν σε χρονικό ύψος μεγαλύτερο του ένα. Η μετατροπή που επιφέρει η προσθήκη του tiling στο στοιχειώδες κομμάτι υπολογισμού του αλγορίθμου σε σχέση με το Σχήμα 5.5. Παρατηρούμε ότι εάν οι άλλες διαστάσεις παραμείνουν σταθερές, με το tiling έχουμε αύξηση του ανταλλασσόμενου όγκου δεδομένων μεταξύ γειτονικών επεξεργαστών σε κάθε επανάληψη του βρόχου. Ο συνολικά ανταλλασσόμενος όγκος δεδομένων, βέβαια, παραμένει ο ίδιος.



Σχήμα 5.5: Στοιχειώδες κομμάτι στην περίπτωση του tiling

Έστω k το επιθυμητό ύψος του tiling όπως δόθηκε είσοδος στο πρόγραμμα. Για να επιτύχουμε την προσθήκη δυνατοτήτων tiling εκτελούμε τροποποιήσεις που επικεντρώνονται στον κυρίως βρόχο του αλγορίθμου. Συγκεκριμένα η διάσταση του χρόνου διαμερίζεται σε ομάδες χρονικών βημάτων μεγέθους k. Σε κάθε επανάληψη του βρόχου ο κάθε επεξεργαστής εκτελούμε αποστολή, λήψη και υπολογισμό σε μία από αυτές της ομάδες. Ο όγκος δεδομένων που διακινούνται σε κάθε βρόχο είναι k-φορές μεγαλύτερος από τον αντίστοιχο όγκο στον βρόχο του βασικού παράλληλου προγράμματος, καθώς επίσης και k-φορές περισσότεροι υπολογισμοί. Αυτό έχει σαν συνέπεια το να χρειαζόμαστε buffer αποστολής και λήψης k-φορές μεγαλύτερο, καθώς επίσης και αύξηση του πίνακα U 2(n)-διαστάσεων κατά ένα παράγοντα $k/2$ στις $(k+1)(n)$-διαστάσεις. Η υλοποίηση της προσθήκης tiling στο βασικό παράλληλο πρόγραμμα συνοψίζεται στον παρακάτω ψευδοκώδικα:

Ψευδοκώδικας 5.3: Παράλληλο Πρόγραμμα με Tiling, 2 Χωρικών Διαστάσεων

```
1     /*Εκτέλεση απαραίτητων αρχικοποιήσεων, δέσμευση μνήμης, διαμέριση
2       χωρίου επαναλήψεων στις διεργασίες και ορισμός γειτόνων*/
3     Init2D_tile(k);
4     while (loops < MAXLOOPS/k) //Θεωρούμε MAXLOOPS%k=0
5     {
6        I_Receive(k, (x-1,y), (x,y-1));
```

```
7          Wait();
8          Unpack_k_Data();
9          Compute_k_Data();
10         Set_k_DataToSend();
11         I_Send(k, (x+1,y), (x,y+1));
12         Wait();
13     }
```

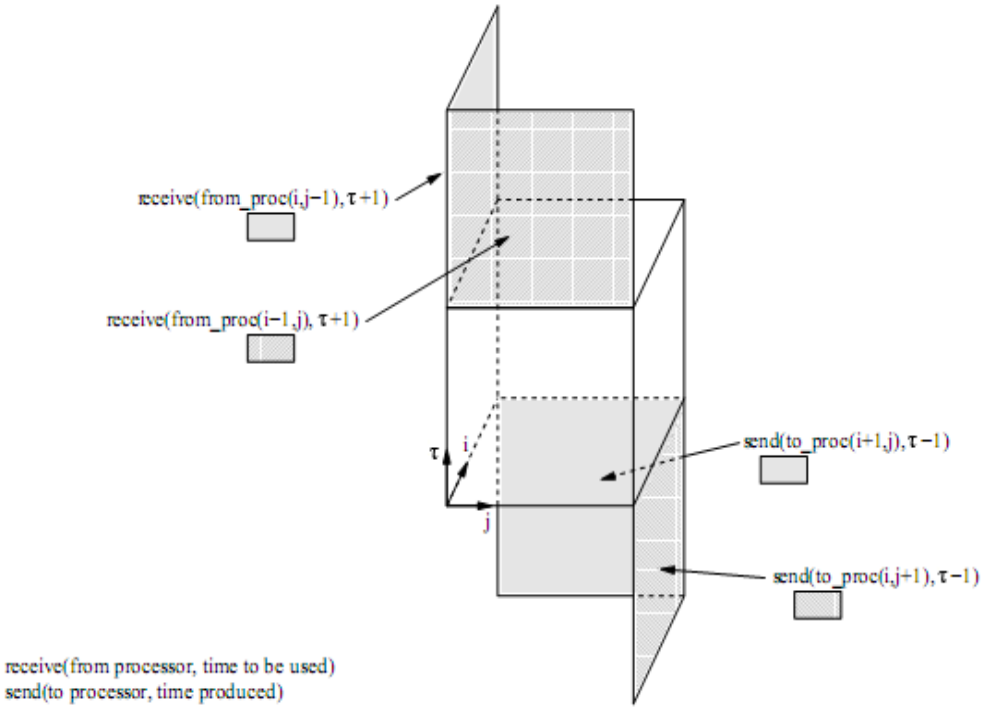
Στα αρνητικά της υλοποίησης αυτής συγκαταλέγεται η αυξημένη απαίτηση σε υπολογιστικούς πόρους αποθήκευσης κατά $O(k)$ σε σχέση με τη βασική έκδοση. Η αυξημένη απαίτηση αυτή έχει σαν αποτέλεσμα τον περιορισμό των χωρίων στα οποία μπορούμε να εκτελέσουμε την προσομοίωση ανάλογα με τη διαθέσιμη μνήμη που έχει το σύστημά μας. Ωστόσο, η υλοποίηση αυτή παρουσιάζει και αρκετά θετικά, καθώς μας δίνει τη δυνατότητα να τροποποιούμε την λεπτότητα παραλληλισμού για να πετύχουμε την καλύτερη δυνατή επίδοση. Για παράδειγμα μπορεί μια προσομοίωση στη βασική έκδοση να πάσχει από μεγάλο κόστος υπολογισμού σε σχέση με συγκριτικά μικρό κόστος επικοινωνίας. Διατηρώντας τις υπόλοιπες εισόδους σταθερές μπορούμε να αυξάνουμε σταδιακά το μέγεθος του tile για να επιτύχουμε εξισορρόπηση των δύο παραγόντων, όπως είναι επιθυμητό από αυτά που προτείνονται σε διάφορες έρευνες για το ζήτημα του tiling (π.χ. στο [17]).

5.5 Έκδοση με Επικαλυπτόμενη Δρομολόγηση

Στις μέχρι τώρα υλοποιήσεις δεν αναφερθήκαμε καθόλου σε θέματα δρομολόγησης των χρονικών βημάτων του κυρίως βρόχου του παράλληλου αλγορίθμου. Σιωπηρά ενσωματώσαμε την "αυτονόητη" δρομολόγηση που ταυτίζεται με εκείνη που παρουσιάζεται στην παράγραφο 4.2 και συνοψίζεται στο εξής: σε κάθε χρονικό βήμα (στιγμιότυπο του βρόχου), είτε με tiling είτε χωρίς, λαμβάνουμε πρώτα τα δεδομένα που θα χρειαστούμε στους υπολογισμούς του tile (είτε πρόκειται για στοιχειώδες μεγέθους 1 είτε για k (υπερκόμβος)) αυτού του βήματος, εκτελούμε τους υπολογισμούς του tile και αποστέλλουμε δεδομένα που μόλις υπολογίσαμε.

Σε αυτή την παράγραφο παρουσιάζουμε την υλοποίηση της δρομολόγησης με επικάλυψη όπως παρουσιάζεται στην παράγραφο 4.3 και στο παράδειγμα της παραγράφου 4.3.1. Η αλλαγή του αλγόριθμου δρομολόγησης αφορά και στην βασική έκδοση και στην έκδοση με tiling. Για την κατανόηση της δρομολόγησης με επικάλυψη όπως εφαρμόζεται στο πρόγραμμά μας παρουσιάζεται μέσω ενός παραδείγματος: Θεωρείστε ένα χωρίο 2 χωρικών διαστάσεων και ένα επεξεργαστή τοποθετημένο κεντρικά στο 2-διάστατο υπολογιστικό πλέγμα, έστω P_i. Κατά το χρονικό βήμα τ ο επεξεργαστής αυτός εκτελεί τα εξής: αποστέλλει τα αποτελέσματα που παράχθηκαν τη χρονική στιγμή τ-1 και λαμβάνει δεδομένα από τους

γείτονές του, τα οποία θα χρησιμοποιηθούν κατά τον υπολογισμό του επόμενου tile τη χρονική στιγμή $\tau+1$, όπως παρουσιάζεται στο Σχήμα 4.2. Αντίστοιχα εργάζονται και οι υπόλοιποι επεξεργαστές της τοπολογίας. Στο Σχήμα 5.5 παρουσιάζεται η λειτουργία ενός χρονικού βήματος για την περίπτωση των δύο χωρικών διαστάσεων. Παρατηρήστε ότι για την υλοποίηση χρειάζεται κάποιο επιπλέον buffering γιατί κρατάμε δεδομένα από τρία χρονικά βήματα (προηγούμενο, υφιστάμενο, επόμενο).



Σχήμα 5.6: Χρονισμός και επιπλέον buffering για την περίπτωση της δρομολόγησης με επικάλυψη

Για να είναι δυνατή η υλοποίηση μιας τέτοιας δρομολόγησης, ήταν απαραίτητη η προσθήκη ενός επιπλέον βήματος αρχικοποίησης πριν την έναρξη του βρόχου και ενός δυαδικού με αυτό τελικού βήματος μετά το τέλος του βρόχου. Κατά το βήμα της αρχικοποίησης εκτελείται μία λήψη, ένας υπολογισμός, και ακόμη μία λήψη. Η αρχικοποίηση αυτή είναι απαραίτητη για να δημιουργηθεί το απαραίτητο offset, έτσι ώστε όταν εισέρθουμε στον κυρίως βρόχο να έχουμε τα δεδομένα που χρειάζεται από τον υπολογισμό, καθώς κα τα δεδομένα που χρειάζονται για την αποστολή και να λάβουμε τα δεδομένα που θα χρειαστούν στον επόμενο υπολογισμό. Αντίστοιχα στο τελευταίο βήμα εκτελείται μία αποστολή, ένας υπολογισμός και ακόμη μία αποστολή. Τα όσα συζητήσαμε συνοψίζονται στον παρακάτω ψευδοκώδικα:

Ψευδοκώδικας 5.4: Παράλληλο Πρόγραμμα με Επικαλυπτόμενη Δρομολόγηση

```
1       Receive(1); //1η λήψη
2       Unpackdata(1);
```

```
3       Compute_tile(1); //1ος υπολογισμός
4       Setdatatosend(1);
5       Receive(2);
6       Unpackdata(2);
7       while(loops < MAXLOOPS -1)
8       {
9          Receive(τ+1);
10         Send(τ-1);
11         Compute(τ);
12         WaitAll();
13         Setdatatosend(τ);
14         Unpackdata(τ+1);
15      }
16      Send(Last-1);
17      Compute(Last);
18      Setdatatosend(Last);
19      Send(Last);
```

Να σημειώσουμε ότι παρόλο που η υλοποίηση αυτή αφήνει τη δυνατότητα να επικαλυφτεί η επεξεργασία με την αντίστοιχη επικοινωνία σε κάθε βρόχο, εντούτοις στο συγκεκριμένο πρόγραμμα που ο βρόχος εκτελείται από τον ίδιο επεξεργαστή δεν έχουμε στην πράξη επικάλυψη. Υπάρχει βέβαια μια βελτίωση γιατί, αν θυμηθούμε την εικόνα 4.4 στα πλαίσια της συζήτησής μας στην παράγραφο 4.2.1, θα δούμε ότι το χρονικό βήμα στο πρόγραμμά μας πλησιάζει αυτό που παρουσιάζεται στο Σχήμα 4.4b. Η διαφορά είναι όμως ότι μόνο το γέμισμα των kernel buffer αποστολής και λήψης γίνεται από άλλη διεργασία (OS kernel) ενώ όλες οι άλλες λειτουργίες(γέμισμα των MPI buffer, υπολογισμοί, εντολές MPI send και receive) εκτελούνται από την ίδια διεργασία που εκτελεί το πρόγραμμα. Συνεπώς, στην περίπτωση που το OS kernel εκτελείται στον ίδιο επεξεργαστή που εκτελεί και την διεργασία του προγράμματος δε θα παρατηρήσουμε καμία βελτίωση σε σχέση με το αρχικό βασικό πρόγραμμα ενώ εάν εκτελείται σε άλλο επεξεργαστή υπάρχει μικρό περιθώριο βελτίωσης. Πραγματικά οφέλη από την υλοποίησή μας αυτή μπορούμε να πάρουμε εάν εκμεταλλευτούμε την ανεξαρτησία των λειτουργιών επεξεργασίας και επικοινωνίας σε κάθε χρονικό βήμα που μας παρέχει η δρομολόγηση με επικάλυψη τοποθετώντας την κάθε λειτουργία σε διαφορετική μονάδα επεξεργασίας. Μια τέτοια δυνατότητα έχει ήδη εξερευνηθεί με επιτυχία στο [17] όπου η λειτουργία επικοινωνίας ανατίθεται σε DMA engine, ενώ εμείς στη συνέχεια εξερευνούμε τη δυνατότητα ανάθεσης της λειτουργίας των υπολογισμών στον ένα λογικό επεξεργαστή ενός φυσικού επεξεργαστή με τεχνολογία Hyper-Threading (βλέπε παράγραφο 2.3) και τη λειτουργία της επικοινωνίας στον άλλο λογικό επεξεργαστή.

5.6 Έκδοση με Επικαλυπτόμενη Δρομολόγηση με Χρήση Δύο Συμμετρικών Νημάτων (Pthreads)

Στα πλαίσια της συζήτησης που ξεκινήσαμε στο τέλος της προηγούμενης παραγράφου παρουσιάζουμε την πρώτη μας υλοποίηση που σπάει τη ροή του παράλληλου προγράμματος σε δύο νήματα. Σε αυτή την υλοποίηση τα νήματα αυτά είναι συμμετρικά, με την έννοια ότι εκτελούν σχεδόν πανομοιότυπο φόρτο εργασίας (που περιλαμβάνει επικοινωνία και υπολογισμό) μοιράζοντας το φόρτο εργασίας μεταξύ τους.

Για τον σκοπό αυτό φτιάχνουμε δύο νήματα POSIX μέσα στα πλαίσια της βιβλιοθήκης NPTL του Linux (βλέπε παράγραφο 5.8), καθένα από τα οποία εκτελεί τα ακόλουθα: αποστολή ή λήψη μηνυμάτων και πακετάρισμα ή ξεπακετάρισμα των δεδομένων, και εκτέλεση υπολογισμών στο μισό χώρο επαναλήψεων. Για τον συγχρονισμό των νημάτων ώστε να εξασφαλιστεί η ατομικότητα των εγγραφών στα κοινά δεδομένα και ο μεταξύ τους συντονισμός χρησιμοποιούμε τους μηχανισμούς συγχρονισμού που αναφέρονται στην παράγραφο 5.8. Τέλος για την εξασφάλιση ότι κάθε νήμα θα εκτελείται στον ένα από τους δύο λογικούς επεξεργαστές του ίδιου φυσικού πακέτου χρησιμοποιούμε τη δυνατότητα του CPU Affinity, που και πάλι περιγράφεται αναλυτικά στην παράγραφο 5.8. Παρουσιάζουμε στη συνέχεια μόνο τον ψευδοκώδικα για τις συναρτήσεις που εκτελούν τα δύο συμμετρικά νήματα:

Ψευδοκώδικας 5.5: Παράλληλο Πρόγραμμα με Δύο Συμμετρικά Pthreads

```
1  EvenThread1()
2  {
3     …
4     sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask);
5     while (loops < MAXLOOPS - 1)
6     {
7        sema_wait(s1);
8        Send(τ-1);
9        Compute(1st half, τ);
10       MPI_Waitall(2, requests, status);
11       sema_signal(s2);
12       sema_wait(s1);
13       SetDataToSend(τ);
14       sema_signal(s2);
15    }
16    …
17 }
18
19 EvenThread2()
```

```
20  {
21      …
22      sched_setaffinity(gettid(), sizeof(cpu_set_t), &process1_mask);
23      while (loops < MAXLOOPS - 1)
24      {
25          sema_wait(s2);
26          Receive(τ+1);
27          Compute(2nd half, τ);
28          MPI_Waitall();
29          sema_signal(s1);
30          sema_wait(s2);
31          UnpackData(τ+1);
32          sema_signal(s1);
33      }
34      …
35  }
```

Να σημειώσουμε πως η δημιουργία των νημάτων εκτελείται από τη συνάρτηση main() της αρχικής διεργασίας και ότι ο ένας εκ των δύο σηματοφορέων αρχικοποιείται στο 0 και ο άλλος στο 1 για να μπορεί να ξεκινήσει η εκτέλεση. Όπως παρατηρούμε στον παραπάνω ψευδοκώδικα έχουμε αρκετά σημεία συγχρονισμού που μπορεί να επιφέρουν σημαντική μείωση στην απόδοση. Όμως, επειδή οι φόρτοι των δύο νημάτων είναι σχεδόν πανομοιότυπου βάρους, αυτή η μείωση στην επίδοση ελαχιστοποιείται γιατί κανένα εκ των δύο νημάτων δε θα μπλοκάρει για μεγάλο χρονικό διάστημα.

5.7 Έκδοση με Επικαλυπτόμενη Δρομολόγηση με Χρήση Δύο Ασύμμετρων Νημάτων (Pthreads)

Σε αυτή την παράγραφο παρουσιάζουμε την υλοποίηση του παράλληλου προγράμματος που διαχωρίζει τη ροή του προγράμματος σε δύο ασύμμετρα νήματα. Συγκεκριμένα, στο ένα νήμα ανατίθενται οι λειτουργίες επικοινωνίας και στο άλλο νήμα οι λειτουργίες επεξεργασίας. Με χρήση της επικαλυπτόμενης δρομολόγησης τα δύο αυτά νήματα εκτελούν λειτουργίες σε ανεξάρτητα δεδομένα σε κάθε στιγμιότυπο του βρόχου (ή χρονικό βήμα) και συνεπώς δεν επηρεάζουν το ένα το άλλο. Ο μόνος συγχρονισμός που χρειάζεται είναι για να εξασφαλιστεί ότι βρίσκονται στο ίδιο χρονικό βήμα, έστω τ.

Η υλοποίηση χρησιμοποιεί τους ίδιους μηχανισμούς συγχρονισμού, βιβλιοθήκη νημάτων και το CPU Affinity που χρησιμοποιούνται και στην περίπτωση τω συμμετρικών νημάτων. Επιλέξαμε να δώσουμε τον φόρτο εργασίας που αφορά το γέμισμα των δύο buffer του προγράμματος για αποστολή και λήψη των μηνυμάτων στο νήμα επικοινωνίας γιατί θεωρήσαμε ότι για μεγάλα χωρία (στα οποία επικεντρώνουμε την προσοχή μας) η εκτέλεση

των υπολογισμών που βαραίνει το νήμα επεξεργασίας είναι χρειάζεται περισσότερο χρόνο (κάτι το οποίο επιβεβαιώνεται από τα πειραματικά δεδομένα, κεφάλαιο 6). Η λειτουργία των δύο ασύμμετρων νημάτων φαίνεται παρουσιάζεται παρακάτω:

Ψευδοκώδικας 5.6: Παράλληλο Πρόγραμμα με Σύο Ασύμμετρα Pthreads

```
1  CompThread()
2  {
3     …
4     sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask);
5     while (loops < MAXLOOPS - 1)
6     {
7        sema_wait(s2);
8        Compute(τ);
9        sema_signal(s1);
10    }
11    …
12 }
13
14 CommThread()
15 {
16    …
17    sched_setaffinity(gettid(), sizeof(cpu_set_t), &process1_mask);
18    while (loops < MAXLOOPS - 1)
19    {
20       Receive(τ+1);
21       Send(τ-1);
22       MPI_Waitall();
23       sema_wait(s1);
24       SetDataToSend(τ);
25       UnpackData(τ+1);
26       sema_signal(s2);
27    }
28    …
29 }
```

Σημειώνουμε ότι ο σηματοφορέας s2 αρχικοποιείται στο 1 ενώ ο s1 στο 0, για να ξεκινήσουν και τα δύο νήματα ταυτόχρονα την εκτέλεσή τους. Το νήμα επικοινωνίας πρέπει αναγκαστικά να περιμένει την εκτέλεση των υπολογισμών από το νήμα επεξεργασίας ενός χρονικού βήματος πριν τοποθετήσει τα δεδομένα που υπολογίστηκαν στο buffer λήψης του προγράμματος, ενώ το νήμα επεξεργασίας πρέπει να περιμένει, αντίστοιχα, το ξεπακετάρισμα των δεδομένων που λήφθηκαν την προηγούμενη χρονική στιγμή.

Αυτή η έκδοση του προγράμματος περιμένουμε να έχει την καλύτερη επίδοση σε σχέση με τις προηγούμενες για δύο, κυρίως, λόγους. Πρώτο, γίνεται ουσιαστική επικάλυψη

της επικοινωνίας και επεξεργασίας και τοποθέτησή τους σε δύο νήματα που εκτελούνται από τους δύο ξεχωριστούς λογικούς επεξεργαστές ενός φυσικού τσιπ που υποστηρίζει την τεχνολογία Hyper-Threading. Δεύτερο, και σε σχέση με τη συμμετρική περίπτωση παρουσιάζει λιγότερα σημεία συγχρονισμού και επωφελείται από την ετερογένεια των νημάτων του, η οποία σύμφωνα με αρκετά ερευνητικά paper (όπως στα [20] και [21]) επιφέρει καλύτερη επίδοση σε ένα επεξεργαστή Simultaneous Multithreading (SMT).

5.8 Θέματα Υλοποίησης

5.8.1 Νήματα POSIX

Ένα νήμα (thread) είναι μια ανεξάρτητη ροή ελέγχου μέσα σε μια διεργασία. Όλα τα νήματα που ανήκουν στην ίδια διεργασία μοιράζονται:

- τον εικονικό χώρο διευθύνσεων της διεργασίας
- ανοικτά αρχεία
- χειριστές διακοπών

Όμως σε κάθε νήμα αντιστοιχεί κατ' αποκλειστικότητα:

- το δικό του context (καταχωρητές, μετρητής προγράμματος)
- η δική του στοίβα για τοπικές μεταβλητές

Για την υλοποίηση νημάτων μέσα στα πλαίσια της εφαρμογής μας χρησιμοποιήσαμε την βιβλιοθήκη Native POSIX Threads Library (NPTL) που παρέχει τη δυνατότητα πολυνηματικού προγραμματισμού στο λειτουργικό σύστημα Linux. Αποτελεί την πιο ολοκληρωμένη και συμβατή με το πρότυπο POSIX υλοποίηση, παρέχοντας καλύτερες επιδόσεις από παλαιότερες βιβλιοθήκες πολυνηματικού προγραμματισμού στο Linux. Επίσης σημαντικό ρόλο στην επιλογή της NPTL βιβλιοθήκης είναι το γεγονός ότι στο κάθε νήμα που δημιουργείται στα πλαίσια αυτής της βιβλιοθήκης αποκτά δική του ταυτότητα (t-id) νήματος, ξεχωριστή από την ταυτότητα (p-id) της διεργασίας, κάτι που μας είναι απαραίτητο για να μπορούμε να δρομολογήσουμε τα νήματα σε διαφορετικό hardware νήμα που παρέχεται από το υπολογιστικό cluster, στα πλαίσια του CPU Affinity που αναφέρουμε παρακάτω.

5.8.2 CPU Affinity

Η ικανότητα του χρονοδρομολογητή ενός λειτουργικού συστήματος να αντιστοιχεί (bind) μία διεργασία (ή νήμα) σε ένα συγκεκριμένο επεξεργαστή ονομάζεται CPU affinity. Σε μια τέτοια περίπτωση ο προγραμματιστής έχει τη δυνατότητα να περιορίζει το σύνολο των επεξεργαστών στους οποίους ο δρομολογητής τοποθετεί την προς εκτέλεση διεργασία ή

νήμα. Το σύνολο αυτό μπορεί να δίνεται σαν ένας μόνο επεξεργαστής ή σαν ένα υποσύνολο επεξεργαστών, οπότε και η διεργασία θα εκτελεστεί τυχαία σε έναν από τους επεξεργαστές του δοθέντος υποσυνόλου.

Στην περίπτωση του λειτουργικού συστήματος Linux, η δυνατότητα αυτή παρέχεται στον προγραμματιστή μέσω της κλήσης συστήματος *sched_affintiy()*. Η κλήση αυτή δέχεται ως ορίσματα την ταυτότητα id της διεργασίας ή του νήματος και μία μάσκα *affinity mask* η οποία ορίζει το σύνολο των επεξεργαστών στους οποίους η διεργασία ή νήμα μπορεί να εκτελεστεί. Η μάσκα αυτή είναι ένας μη προσημασμένος ακέραιος αριθμός, όπου το δυαδικό ψηφίο στην k θέση της δυαδική αναπαράσταση της μάσκας αντιστοιχεί στον επεξεργαστή k του συστήματος. Όταν το δυαδικό ψηφίο αυτό είναι 1 τότε επιτρέπεται στη διεργασία να εκτελεστεί στον συγκεκριμένο υπολογιστή, ενώ δεν επιτρέπεται εάν είναι 0. Για παράδειγμα εάν δώσουμε τη μάσκα 1101 (δεκαδικό 13) σε ένα σύστημα με 4 επεξεργαστές τότε επιτρέπεται η εκτέλεση σε όλους τους επεξεργαστές πλην του επεξεργαστή 1 (ξεκινάμε να μετράμε από το 0).

Στην υλοποίησή μας, η δυνατότητα του CPU affinity είναι μεγάλης σημασίας, διότι μέσω αυτού ορίζουμε σε ποιο λογικό επεξεργαστή θα εκτελεστεί το κάθε νήμα στην περίπτωση της παράλληλης υλοποίησης με χρονοδρομολόγηση με επικάλυψη. Έτσι εξασφαλίζουμε ότι το μεν πρώτο από τα δύο νήματα της κάθε MPI διεργασίας τοποθετείται στον πρώτο λογικό επεξεργαστή ενός φυσικού επεξεργαστή και η δεύτερη διεργασία στον δεύτερο λογικό επεξεργαστή του ίδιου φυσικού πακέτου. Μόνο με αυτή τη δυνατότητα μπορούμε πράγματι να ξέρουμε ότι χρησιμοποιούμε τους λογικούς επεξεργαστές του Hyper-Threading για να δρομολογήσουμε σε αυτούς τα κατάλληλα νήματα και να μην οδηγηθούμε σε λανθασμένα συμπεράσματα από τυχόν διαφορετική αντιστοίχηση των νημάτων στους λογικούς επεξεργαστές από το λειτουργικό.

5.8.3 Μηχανισμοί Συγχρονισμού

Στις υλοποιήσεις της εξίσωσης διάχυσης σε παράλληλα προγράμματα που χρησιμοποιούν δρομολόγηση με επικάλυψη και έχουμε δύο νήματα για κάθε MPI διεργασία (και στις δύο περιπτώσεις ασύμμετρων και συμμετρικών νημάτων) κάνουμε χρήση μηχανισμών συγχρονισμού για να εξασφαλιστεί η ατομικότητα των τροποποιήσεων στα κοινά δεδομένα, καθώς και για να επιτευχθεί ο μεταξύ τους συντονισμός για τις εργασίες που έχουν να εκτελέσουν. Οι μηχανισμοί αυτοί πρέπει να είναι όσο το δυνατό πιο ελαφροί στη χρήση υπολογιστικών πόρων, έτσι ώστε η πολύ συχνή χρήση τους να μην επιβαρύνει το σύστημα.

Ένας διαδεδομένος μηχανισμός συγχρονισμού δίνεται μέσω της βιβλιοθήκης NPTL (βλέπε παράγραφο) και συγκεκριμένα τα Pthread Mutexes, σε συνδυασμό με Pthread Conds. Τα μεν

πρώτα εξασφαλίζουν αποκλειστική πρόσβαση σε κάθε νήμα σε μια κρίσιμη περιοχή κώδικα(critical section). Τα δε δεύτερα είναι μεταβλητές κατάστασης (conditional variables) που χρησιμεύουν σε περίπτωση που χρειάζεται ένα νήμα να σημάνει ένα γεγονός σε κάποιο άλλο σήμα, κάτι που σαφώς χρειάζεται στην υλοποίησή μας. Η χρήση αυτών των μηχανισμών γίνεται μέσω των συναρτήσεων *pthread_mutex_lock()* και *pthread_mutex_unlock()* για τα πρώτα και των συναρτήσεων *pthread_cond_wait()* και *pthread_cond_signal()* για τα δεύτερα. Η κλήση των συναρτήσεων *pthread_mutex_lock()* και *pthread_mutex_unlock()* έχει ως αποτέλεσμα την απόκτηση αποκλειστικής πρόσβασης σε ένα κοινό αντικείμενο και την αποδέσμευση της αποκλειστικής πρόσβασης, αντίστοιχα. Η κλήση της συνάρτησης *pthread_cond_wait()* σε μία μεταβλητή κατάστασης έχει ως αποτέλεσμα το μπλοκάρισμα του νήματος που κάλεσε την εντολή αυτή μέχρι να γίνει αληθής μια κατάσταση (εάν είναι αληθής δεν μπλοκάρει). Η κλήση της συνάρτησης *pthread_cond_signal()* από ένα νήμα κάνει την κατάσταση αληθή και ξεμπλοκάρει το πρώτο νήμα στην ουρά αναμονής αυτής της μεταβλητής κατάστασης (conditional variable). Εμείς στα πλαίσια της υλοποίησης του παράλληλου προγράμματος με δύο νήματα στην κάθε MPI διεργασία κάναμε χρήση και των δύο αυτών μηχανισμών για το συγχρονισμό των νημάτων αυτών. Για πιο συμπαγή υλοποίηση δημιουργήσαμε μία βιβλιοθήκη σηματοφορέων (δες Παράρτημα 1.2) όπου η υλοποίηση των συναρτήσεων P() και V() των σηματοφορέων γίνεται αποκλειστικά με χρήση μηχανισμών Pthread Mutexes και Pthread Conds.

Κεφάλαιο 6

Πειραματικά Αποτελέσματα

6.1 Πλαίσιο Πειραματικών Μετρήσεων

6.1.1 Υπολογιστικό Σύστημα

Εκτελέσαμε τις πειραματικές μας μετρήσεις σε ένα cluster 8 υπολογιστών, καθένας από τους οποίους έχει 2 επεξεργαστές Intel Xeon, χρονισμένους στα 2.8GHz, και 2GB μνήμης μοιραζόμενης στους δύο επεξεργαστές. Ο επεξεργαστής αυτός είναι βασισμένος στη μικροαρχιτεκτονική Netburst™, και αποτελεί ένα από τα πρώτα τσιπ mainstream που περικλείουν στοιχειώδεις δυνατότητες simultaneous multithreading (SMT).

Ο επεξεργαστής που εξετάζουμε έχει ένα out-of-order, υπερβαθμωτό, speculative πυρήνα, που χαρακτηρίζεται για τη βαθιά του διασωλήνωση. Αυτός ο πυρήνας έχει εύρος ζώνης φόρτωσης εντολών μέχρι και τρεις μικρολειτουργίες (uops) ανά κύκλο, εύρος ζώνης έκδοσης και εκτέλεσης εντολών μέχρι έξι εντολές σε κάθε κύκλο και εύρος ζώνης απόσυρσης εντολών μέχρι τρεις σε κάθε κύκλο. Το κόστος σε αυτόν τον επεξεργαστή για λανθασμένη πρόβλεψη διακλάδωσης ανέρχεται στους 20 κύκλους. Περιέχει ένα αρχείο καταχωρητών μετονομασίας 128 εγγραφών και μπορούν να βρίσκονται σε ενεργή κατάσταση (in-flight) σε μία δεδομένη χρονική στιγμή μέχρι και 126 εντολές. Από αυτές, το πολύ 48 μπορούν να είναι εντολές load και 24 store. Έχει μία κρυφή μνήμη L1 8-way set associative (συνόλου συσχέτισης) των 16KB, με γραμμή των 64 bytes και latency για φόρτωμα-χρήση 2 κύκλους. Η ενοποιημένη κρυφή μνήμη επιπέδου 2 L2 είναι μία κρυφή μνήμη 1MB 8-way set associative, με επίσης γραμμές των 64 bytes. Το ισοδύναμο μίας τυπικής L1 κρυφής μνήμης εντολών στο σύστημά μας είναι το execution trace cache. Οι κρυφές μνήμες trace δεν αποθηκεύουν εντολές αλλά "ίχνη" (traces) από αποκωδικοποιημένες uops. Η κρυφή μνήμη trace του Xeon έχει χωρητικότητα για 12K uops, και είναι επίσης 8-way set associative.

Ο επεξεργαστή Xeon του συστήματός μας, παρέχει τη δυνατότητα για ρητό prefetch σε προγράμματα μέσω ειδικών εντολών. Ο προγραμματιστής μπορεί να χρησιμοποιήσει αυτές τις εντολές για να υποθέσει τις θέσεις μνήμης που θα προσπελαστούν στη συνέχεια. Ο επεξεργαστής επίσης υλοποιεί ένα μηχανισμό prefetch στο hardware με τον οποίο δεδομένα μνήμης προ-φορτώνονται στην ενιαία L2 κρυφή μνήμη με βάση προηγούμενα μοτίβα

™ Netburst είναι σήμα κατατεθέν της Intel Corporation ή θυγατρικών της στις Ηνωμένες Πολιτείες και σε άλλες χώρες

αναφοράς. Αυτός ο hardware prefetcher υποστηρίζει πολλαπλές ροές και μπορεί να αναγνωρίζει συχνά παρατηρούμενα μοτίβα όπως είναι οι γραμμικές προσπελάσεις forward ή backward.

Η τεχνολογία Hyper-Threading [5] κάνει ένα φυσικό επεξεργαστή να εμφανίζεται σαν δύο λογικοί επεξεργαστές με την εφαρμογή μίας προσέγγισης της τεχνολογίας SMT με δύο νήματα. Το λειτουργικό σύστημα αναγνωρίζει δύο διαφορετικούς λογικούς επεξεργαστές, ο καθένας από τους οποίους διατηρεί μία ξεχωριστή ουρά εκτέλεσης. Σε ένα επεξεργαστή με ενεργοποιημένη την τεχνολογία HT, σχεδόν όλοι οι πόροι εκτέλεσης μοιράζονται: κρυφές μνήμες όλων των επιπέδων, μονάδες εκτέλεσης, τα κυκλώματα φόρτωσης, αποκωδικοποίησης, χρονοδρομολόγησης και απόσυρσης εντολών, ο πίνακας ολικού ιστορικού. Όταν υπάρχει ταυτόχρονη αίτηση για μοιραζόμενους πόρους, η πρόσβαση εναλλάσσεται μεταξύ των νημάτων, συνήθως με λεπτότητα ενός κύκλου. Συνεπώς, όταν και τα δύο νήματα είναι ενεργά, το μέγιστο εύρος ζώνης φόρτωση, αποκωδικοποίησης, έκδοσης, εκτέλεσης και απόσυρσης εντολών είναι ουσιαστικός το μισό για κάθε ένα από αυτά.

Το αρχιτεκτονικό state σε ένα επεξεργαστή με την τεχνολογία Hyper-Threading αντιγράφεται για κάθε νήμα. Αντιγράφονται επίσης οι instruction pointers, τα ITLBs, το κύκλωμα μετονομασίας, η στοίβα επιστροφής και τα buffers ιστορικού διακλαδώσεων. Οι ουρές προσωρινής αποθήκευσης μεταξύ των διαφόρων σταδίων διασωλήνωσης, καθώς επίσης και οι ουρές load/store, διαμερίζονται στατικά, έτσι ώστε κάθε νήμα να μπορεί να χρησιμοποιήσει το πολύ τις μισές εγγραφές τους. Με αυτό τον τρόπο, ένα νήμα μπορεί να προχωρήσει μπροστά στο τι κάνει χωρίς να εξαρτάται και να επηρεάζεται από την πρόοδο του άλλου νήματος. Το buffer αναδιάταξης είναι διαμερίζεται επίσης μεταξύ των δύο νημάτων.

Όπως αναφέρεται στο [22], αυτή η στατική διαμέριση υπολογιστικών πόρων και όχι ο δυναμικός τους διαμοιρασμός είναι η βασική διαφορά μεταξύ του Hyper-Threading και της θεωρητικά βέλτιστης αρχιτεκτονικής όπως αυτή προτείνεται στο [23]. Έχει συζητηθεί ερευνητικά στον τομέα του SMT ότι ο δυναμικός διαμοιρασμός των δομών είναι ποιο αποτελεσματικός από τη στατική τους διαμέριση. Ωστόσο, όπως έχουμε σημειώσει, η στατική διαμέριση των πόρων αποτρέπουν από το μη βέλτιστα νήματα να επηρεάζουν την επίδοση παράλληλα εκτελούμενων με αυτά νήματα. Σε κάθε περίπτωση, τα μειονεκτήματα της στατικής διαμέρισης πόρων ελαχιστοποιείται όταν υπάρχουν μόνο δύο hardware contexts, έτσι ώστε η παρατηρούμενη συμπεριφορά του συστήματος δεν απέχει σημαντικά από εκείνη του συστήματος με πλήρως δυναμικό διαμερισμό κοινών δομών.

6.1.2 Λειτουργικό Σύστημα

Το λειτουργικό σύστημα που χρησιμοποιήθηκε στα πειράματά μας ήταν ένα λειτουργικό Linux με έκδοση πυρήνα 2.6.24.2. Αποφασιστικό παράγοντα για την επίδοση πολυνηματικών εφαρμογών σε ένα επεξεργαστή με ενεργοποιημένη την τεχνολογία HT παίζει ο αλγόριθμος δρομολόγησης που χρησιμοποιεί το λειτουργικό σύστημα. Προφανώς, ακόμη και εάν και οι λογικοί επεξεργαστές σε ένα φυσικό πακέτο αναγνωριστούν ως διαφορετικές μονάδες επεξεργασίας, πρέπει να διαχωρίζονται από λογικούς επεξεργαστές που βρίσκονται σε διαφορετικό φυσικό πακέτο.

Ο πυρήνας του Linux από την έκδοση 2.6 και έπειτα έχει υλοποιήσει ένα ενιαίο αλγόριθμο για το χειρισμό δρομολόγησης πολυεπεξεργαστικών συστημάτων στα οποία οι μονάδες επεξεργασίας έχουν μη ομοιόμορφες σχέσεις μεταξύ τους. Τέτοια συστήματα είναι για παράδειγμα μηχανήματα SMP (symmetrical multiprocessing) με επεξεργαστές με ενεργοποιημένη την τεχνολογία HT ή μηχανήματα NUMA (Non-Uniform Memory Access). Αυτός ο αλγόριθμος είναι βασισμένος σε στοιχεί που καλούνται scheduling domain. Τα scheduling domains είναι σύνολα από επεξεργαστικές οντότητες που έχουν κοινές ιδιότητες και πολιτικές δρομολόγησης. Με τα scheduling domains, οι διαφορετικές ιδιότητες των λογικών και φυσικών επεξεργαστικών μονάδων μπορούν να αναγνωριστούν και ο δρομολογητής μπορεί να πράξει ανάλογα.

Ο δρομολογητής του Linux χρησιμοποιεί την εντολή halt για την ενίσχυση της επίδοσης των τρεχουσών διεργασιών. Όταν μόνο ένα νήμα εκτέλεσης είναι διαθέσιμο για ένα φυσικό επεξεργαστή ο δρομολογητής χρησιμοποιεί την εντολή halt στον αδρανή λογικό επεξεργαστή με σκοπό να μεταβεί ο φυσικός επεξεργαστής στην κατάσταση single thread (ST). Με αυτό τον τρόπο, το τρέχων νήμα μπορεί να εκμεταλλευτεί όλους τους πόρους τους επεξεργαστή. Η εντολή halt χρησιμοποιείται επίσης από τον δρομολογητή όταν υπάρχουν δύο διαφορετικές διεργασίες με διαφορετικές προτεραιότητες, μία σε καθένα από τους λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Η διεργασία με τη μεγαλύτερη προτεραιότητα θα εκτελεστεί σε κατάσταση ST για να αξιοποιήσει πλήρως τους πόρους του φυσικού επεξεργαστικού πακέτου.

6.1.3 Διαδικασία Λήψης Μετρήσεων

Λόγω της ύπαρξης πολλών διαφορετικών προγραμμάτων για την προσομοίωση της εξίσωσης διάχυσης έπρεπε να ληφθούν πολλές πειραματικές μετρήσεις. Παρουσιάζουμε σε αυτή την παράγραφο το πλάνο μετρήσεων που ακολουθήσαμε με λεπτομέρεια.

Αρχίζουμε με τις μετρήσεις που πήραμε στο πρόγραμμα προσομοίωσης της εξίσωσης διάχυσης μίας διάστασης (1D advection equation). Πήραμε μετρήσεις για χωρία μεγέθους 10^3, 10^4 σε 2,3,4,6,8,10,12,16,24 και 32 επεξεργαστές (η τοπολογία στην περίπτωση της μίας διάστασης εκφυλίζεται σε ακολουθία επεξεργαστών). Επαναλάβαμε για όλες τις υλοποιήσεις των προγραμμάτων μας, σειριακό, βασικό παράλληλο, παράλληλο με επικαλυπτόμενη δρομολόγηση, παράλληλο με δύο συμμετρικά νήματα και παράλληλο με δύο ασύμμετρα νήματα. Για τις εκδόσεις των υλοποιήσεων που υποστηρίζουν tiling πήραμε δύο μετρήσεις για κάθε χωρίο, μία με ύψος tiling 10 και μία με ύψος tiling 100. Στη χρονική διάσταση είχαμε μέγεθος 1000, δηλαδή 1000 επαναλήψεις, για όλες τις μετρήσεις που πήραμε, και για αυτές στις 2 και 3 χωρικές διαστάσεις που παρουσιάζονται στη συνέχεια.

Για τις υλοποιήσεις που προσομοιώνουν την εξίσωση διάχυσης στις δύο χωρικές διαστάσεις πήραμε μετρήσεις για τα ακόλουθα χωρία: $1K \times 1K, 2K \times 2K, 4K \times 4K$ και $6K \times 6K$. Οι τοπολογίες επεξεργαστών για τις οποία πήραμε μετρήσεις για τα παραπάνω χωρία ήταν τα εξής: $2 \times 1, 2 \times 2, 3 \times 2, 4 \times 2, 3 \times 3, 4 \times 3, 4 \times 4, 5 \times 5$ και 8×4. Επαναλάβαμε τις μετρήσεις για όλες τις υλοποιήσεις και για τις υλοποιήσεις που υποστηρίζουν tiling χρησιμοποιήσαμε μέγεθος tile ίσο με 10 εκτός από την περίπτωση του χωρίου $6K \times 6K$ όπου χρησιμοποιήσαμε μέγεθος 5 λόγο περιορισμού για να μην ξεφύγουμε από το μέγεθος της κύριας μνήμης. Κατά την επεξεργασία των αποτελεσμάτων κρίθηκε σκόπιμο να επαναληφθούν μετρήσεις και για χωρία που δεν είναι τετραγωνικής μορφής, και πήραμε συμπληρωματικές μετρήσεις για τα εξής χωρία: $4K \times 16, 64K \times 16, 256K \times 16, 512K \times 32$.

Στην περίπτωση της τρισδιάστατης εξίσωσης διάχυσης τα χωρία στα οποία επικεντρωθήκαμε ήταν τα εξής: $100 \times 100 \times 100, 200 \times 200 \times 200, 300 \times 300 \times 300$ και $400 \times 400 \times 400$. Για τις υλοποιήσεις με tiling χρησιμοποιήσαμε μέγεθος tile 10, 10, 8 και 4 αντίστοιχα, και πάλι για να μην ξεφύγουμε από το μέγεθος της κύριας μνήμης. Οι τοπολογίες επεξεργαστών στις οποίες επικεντρωθήκαμε ήταν τα εξής: $2 \times 1 \times 1, 2 \times 2 \times 1, 2 \times 2 \times 2, 4 \times 2 \times 1, 3 \times 3 \times 1, 3 \times 2 \times 2, 4 \times 2 \times 2, 4 \times 4 \times 1, 4 \times 3 \times 2$ και $4 \times 4 \times 2$. Όπως και στην περίπτωση των μετρήσεων στις δύο χωρικές διαστάσεις, κρίθηκε και εδώ απαραίτητο να επαναληφθούν μετρήσεις και για χωρία που δεν είναι τετραγωνικής μορφής, και πήραμε συμπληρωματικές μετρήσεις για τα εξής χωρία: $4K \times 16 \times 16, 64K \times 16 \times 16, 64K \times 16 \times 32, 256K \times 16 \times 16$ και $1024K \times 16 \times 16$.

6.2 Αποτίμηση Αποτελεσμάτων

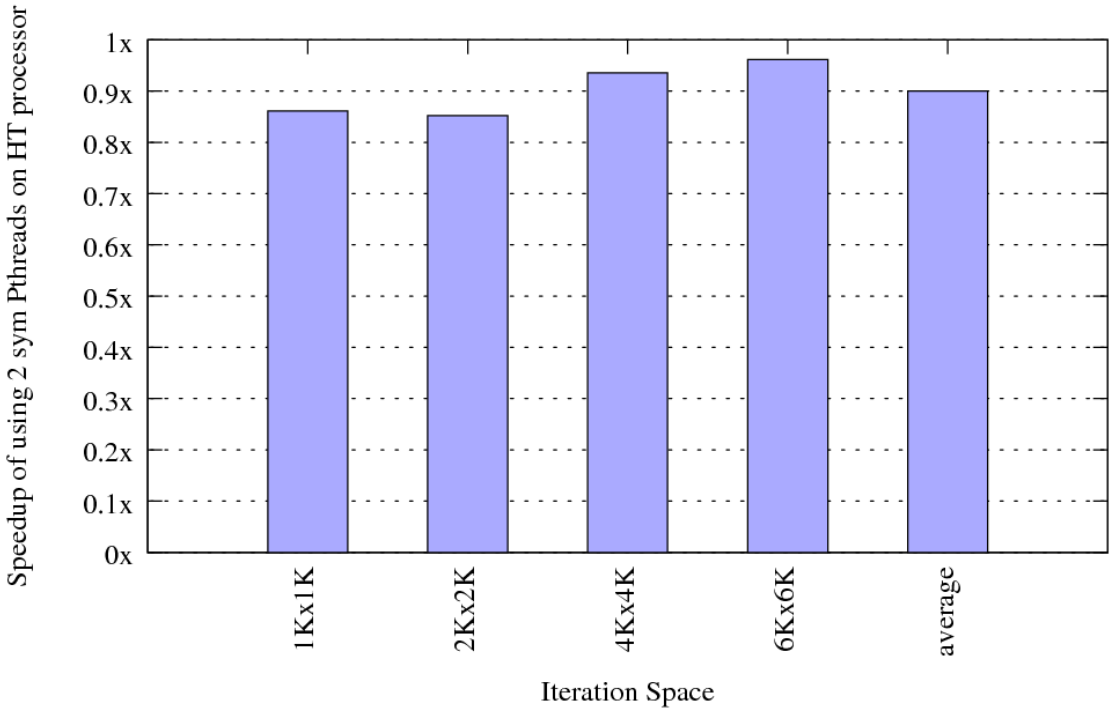
Σε αυτή την παράγραφο προσπαθούμε να κάνουμε μία αποτίμηση των αποτελεσμάτων[3], παρουσιάζοντας γραφικές παραστάσεις και αναλύοντας τα αποτελέσματα αυτά με βάση το θεωρητικό υπόβαθρο που αναπτύχθηκε στα πρώτα κεφάλαια της διπλωματική και από άλλες σχετικές ερευνητικές μελέτες.

6.2.1 Αποτελέσματα Παράλληλης Έκδοσης με Δύο Συμμετρικά Νήματα

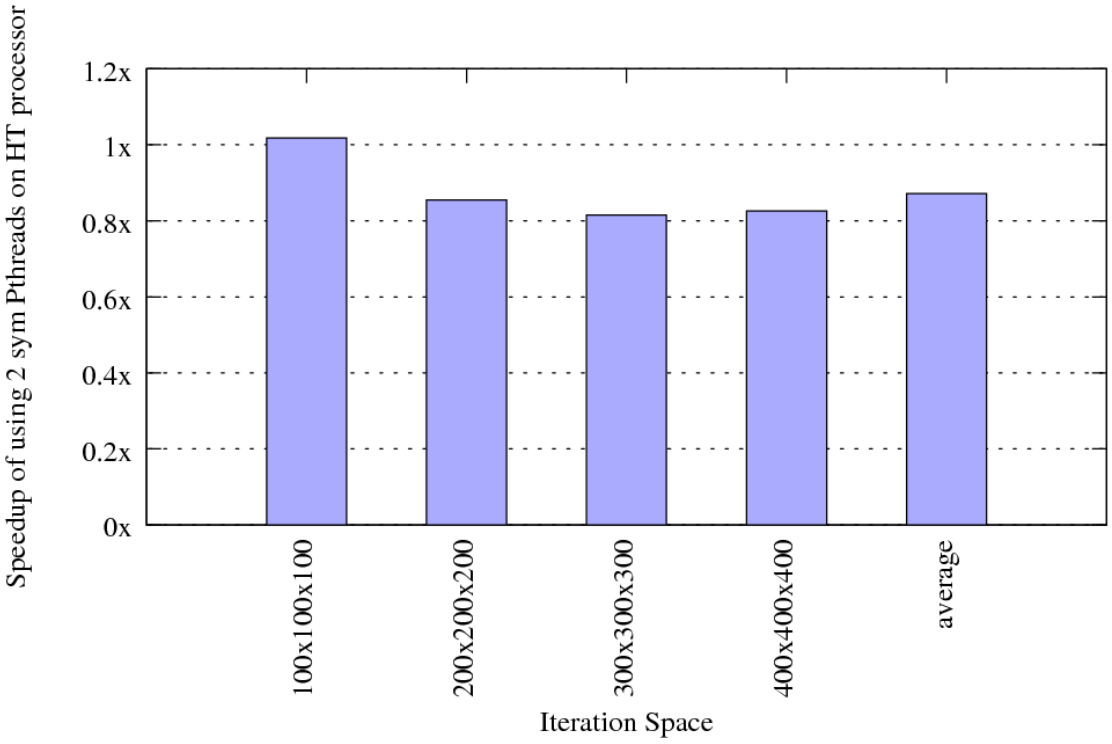
Κρίνουμε σκόπιμο να αρχίσουμε την παρουσίαση των αποτελεσμάτων με εκείνα της έκδοσης του προγράμματός μας που παρουσιάζουν δύο συμμετρικά νήματα. Συγκεκριμένα συγκρίνουμε την έκδοση αυτή με το βασικό παράλληλο πρόγραμμα όταν έχουμε τον ίδιο αριθμό hardware νημάτων. Αυτή η περίπτωση ισχύει όταν αναθέτουμε n MPI διεργασίες στο παράλληλο πρόγραμμα με συμμετρικά νήματα και $2n$ διεργασίες στο βασικό παράλληλο πρόγραμμα, όπου βέβαια $2n \leq$ των διαθέσιμων hardware νημάτων, που στην περίπτωσή μας είναι 32 (16 φυσικοί επεξεργαστές με 2 λογικούς επεξεργαστές ο καθένας). Για πιο ρεαλιστικά αποτελέσματα καλό είναι και στις δύο περιπτώσεις να τρέχουν οι διεργασίες σε ίσο αριθμό λογικών επεξεργαστών που ανήκουν στα ίδια φυσικά πακέτα για να υπεισέρχεται ο διαμερισμός (στατικός κυρίως αλλά και δυναμικός) των υπολογιστικών πόρων που λαμβάνει χώρα στους SMT επεξεργαστές και συγκεκριμένα στους Intel Xeon με τεχνολογία Hyper-Threading, στα πλαίσια των όσων αναφέρονται στο κεφάλαιο 2.

Λαμβάνοντας τα παραπάνω υπόψη παρουσιάζουμε στη συνέχεια αποτελέσματα μετρήσεων για χωρία δύο και τριών διαστάσεων στην περίπτωση που χρησιμοποιούμε 32 MPI διεργασίες στη βασική παράλληλη έκδοση και 16 MPI διεργασίες, με δύο συμμετρικά Pthread νήματα η καθεμιά από αυτές, στην έκδοση με τα συμμετρικά νήματα. Τα αποτελέσματα συνοψίζονται στις γραφικές παραστάσεις του Σχήματος 6.1.

[3] Σημειώνουμε ότι από τις μετρήσεις που πήραμε για τα προγράμματα που αφορούν προσομοιώσεις της εξίσωσης διάχυσης μίας διάστασης δεν παρατηρήσαμε αξιοσημείωτα αποτελέσματα για αυτό και κρίνουμε σκόπιμο να μην αναφερθούμε σε αυτά κατά την παρουσίαση των αποτελεσμάτων. Αυτό οφείλεται στο ότι η ανταλλαγή μηνυμάτων στην περίπτωση της μίας διάστασης εκφυλίζεται στην αποστολή και λήψη ενός μόνο στοιχείου με αποτέλεσμα να μην υπάρχει ουσιαστική διαφορά μεταξύ των υλοποιήσεων. Μοναδική περίπτωση που αυτό διαφέρει είναι στην έκδοση της μίας διάστασης με tiling.



(a) Χώρος επαναλήψεων 2-D



(b) Χώρος επαναλήψεων 3-D

Σχήμα 6.1: Speedup της έκδοσης με συμμετρικά νήματα σε σχέση με τη βασική παράλληλη για 32 hardware νήματα (τοπολογίες 4x4,8x4 και 4x2x2, 4x4x2 για τις δύο περιπτώσεις)

Από τις παραπάνω γραφικές παραστάσεις παρατηρούμε ότι η έκδοση με τα δύο συμμετρικά νήματα δεν παρουσιάζει καθόλου επιτάχυνση, το αντίθετο μάλιστα, βλέπουμε πτώση της επίδοσης κατά μέσο όρο 10%. Αυτό είναι κάτι που περιμέναμε να δούμε, γιατί

ουσιαστικά με τα συμμετρικά νήματα θέλαμε να προσομοιώσουμε τη λειτουργία του συστήματος εάν διπλασιάζαμε τις MPI διεργασίες του, αφού μοιράζουμε στα δύο τη δουλειά της κάθε MPI διεργασίας μεταξύ των δύο νημάτων. Το ότι έχουμε μείωση της απόδοσης αντί να έχουμε ταυτόσημη συμπεριφορά κρίνουμε ότι οφείλεται στους μηχανισμούς συγχρονισμού που επιβαρύνουν σημαντικά την επίδοση του προγράμματος, καθώς και στο ότι η υλοποίηση του MPI είναι βέλτιστη. Επίσης αρκετή ερευνητική δουλειά έχει δείξει ότι σε μηχανήματα με Hyper-Threading τεχνολογία έχουμε περιορισμένες αυξήσεις στην επίδοση όταν τα νήματα που εκτελούνται παρουσιάζουν ομοιογένεια (μοιράζονται δεδομένα), όπως τα [22] και [24].

Σημειώνουμε ότι σε παρόμοια αποτελέσματα κατέληξαν και οι άλλες μετρήσεις μας στην έκδοση με συμμετρικά νήματα με tiling, για αυτό και δεν τις παρουσιάζουμε. Εφόσον είδαμε ότι τα δύο συμμετρικά νήματα δεν επιφέρουν κάποια σημαντική διαφορά στην επίδοση δε θα ασχοληθούμε μαζί τους στη συνέχεια.

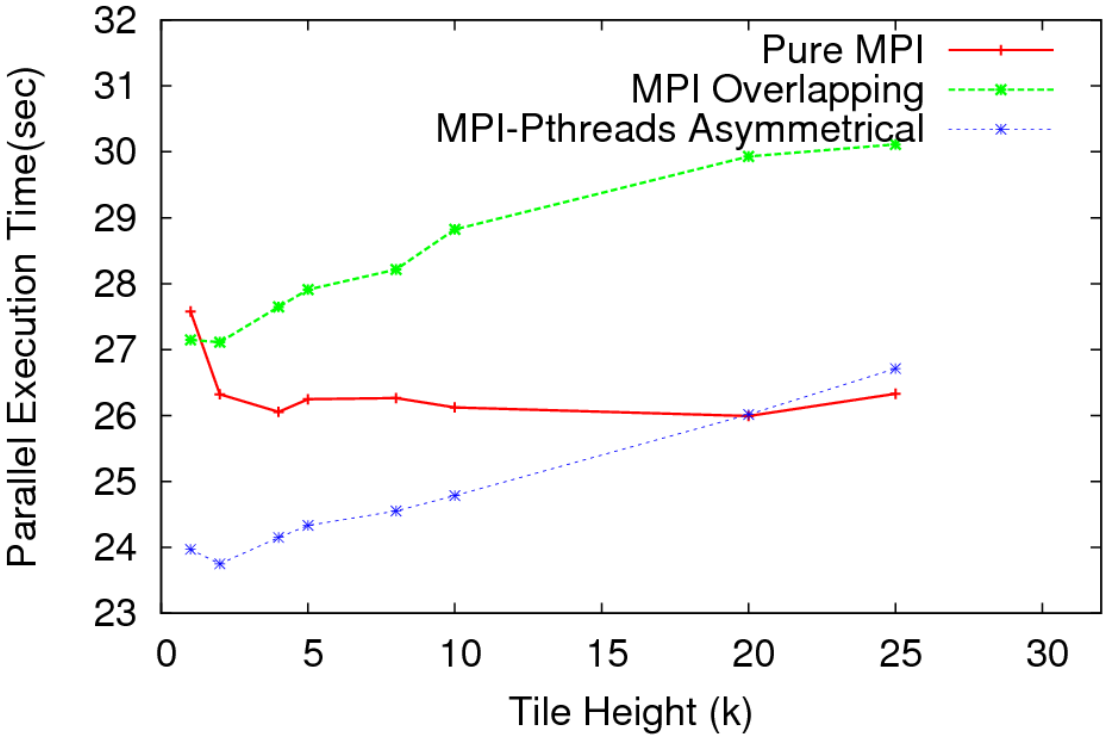
6.2.2 Αποτίμηση του Παράγοντα Tiling των Υλοποιήσεων

Σαν δεύτερο παράγοντα που εξετάζουμε είναι αυτός του tiling και οι διαφορές στην απόδοση που αυτό επιφέρει. Η υλοποίησή μας του μετασχηματισμού υπερκόμβων στα πλαίσια της διπλωματικής εργασίας δεν αγγίζει σε βάθος την ερευνητική μελέτη που έχει γίνει στο θέμα αυτό αλλά αποτελεί μία πρώτη προσέγγιση στο δύσκολο αυτό ζήτημα. Άλλωστε δεν αποτελεί τον κύριο στόχο της δουλειάς μας αλλά ένα ακόμα σκέλος του ζητήματος που εξετάζουμε.

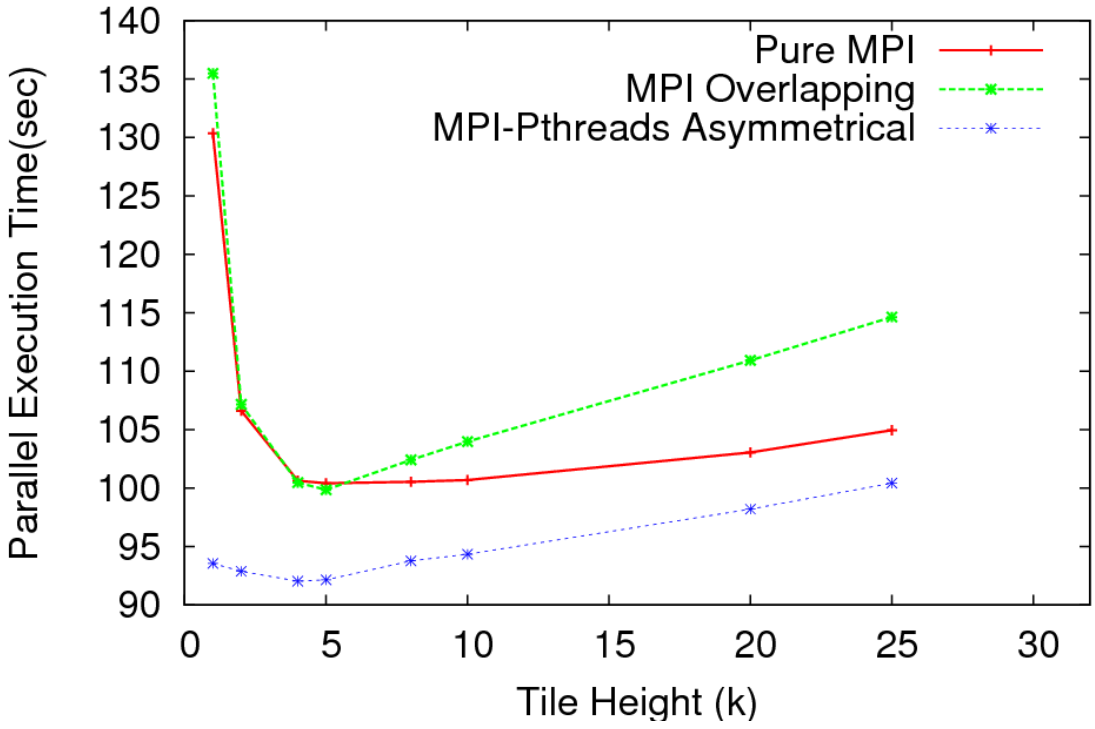
Αξίζει να αναφέρουμε αρχικά ότι η μόνη ενδιαφέρουσα μέτρηση από εκείνες της μίας διάστασης αποτελεί εκείνη για την επίδραση του tiling. Και στα δύο χωρία που εξετάζονται στη μία διάσταση το tiling επιφέρει σημαντική βελτίωση στο σύνολο των περιπτώσεων, αγγίζοντας μερικές φορές σε επιταχύνσεις 2x και 3x. Αυτό βέβαια μπορεί να εξηγηθεί εύκολα αν αναλογιστεί κανείς ότι χωρίς το tiling στην περίπτωση της μίας διάστασης αποστέλλεται και λαμβάνεται ένα στοιχείο τη φορά. Το κόστος έναρξης και λήξης της επικοινωνίας υφίσταται παρόλο που αποστέλλεται ένα μόνο στοιχείο. Μαζεύοντας με το tiling πολλά στοιχεία μαζί έχουμε καλύτερη χρήση του μηχανισμού επικοινωνίας, με σημαντική αύξηση στην επίδοση.

Ασχολούμαστε τώρα με τις πιο ουσιαστικές περιπτώσεις των χωρίων δύο και τριών διαστάσεων και για πλέγματα με πολλούς επεξεργαστές. Περιμένουμε να δούμε καλύτερη συμπεριφορά του tiling εκεί που έχουμε σημαντική μείωση της επίδοσης από πολλή επικοινωνία. Αυτό συμβαίνει περισσότερο στην περίπτωση των τριών χωρικών διαστάσεων γιατί κάθε υπολογιστικός κόμβος έχει περισσότερους γειτονικούς κόμβους. Βέβαια πολύ μεγάλο μέγεθος tile σε ήδη μεγάλα χωρία μπορεί να χειροτερεύει τη επίδοση αντί να την αυξάνει γιατί κάνει πολύ μεγάλο το κόστος επικοινωνίας. Μία μέση λύση πιστεύουμε πως

είναι η πιο αποδοτική, κάτι που υποστηρίζεται και από τις μετρήσεις μας, χαρακτηριστικό παράδειγμα των οποίων παρουσιάζεται στο παρακάτω σχήμα.



(a) Χώρος επαναλήψεων 2-D, 6000x6000



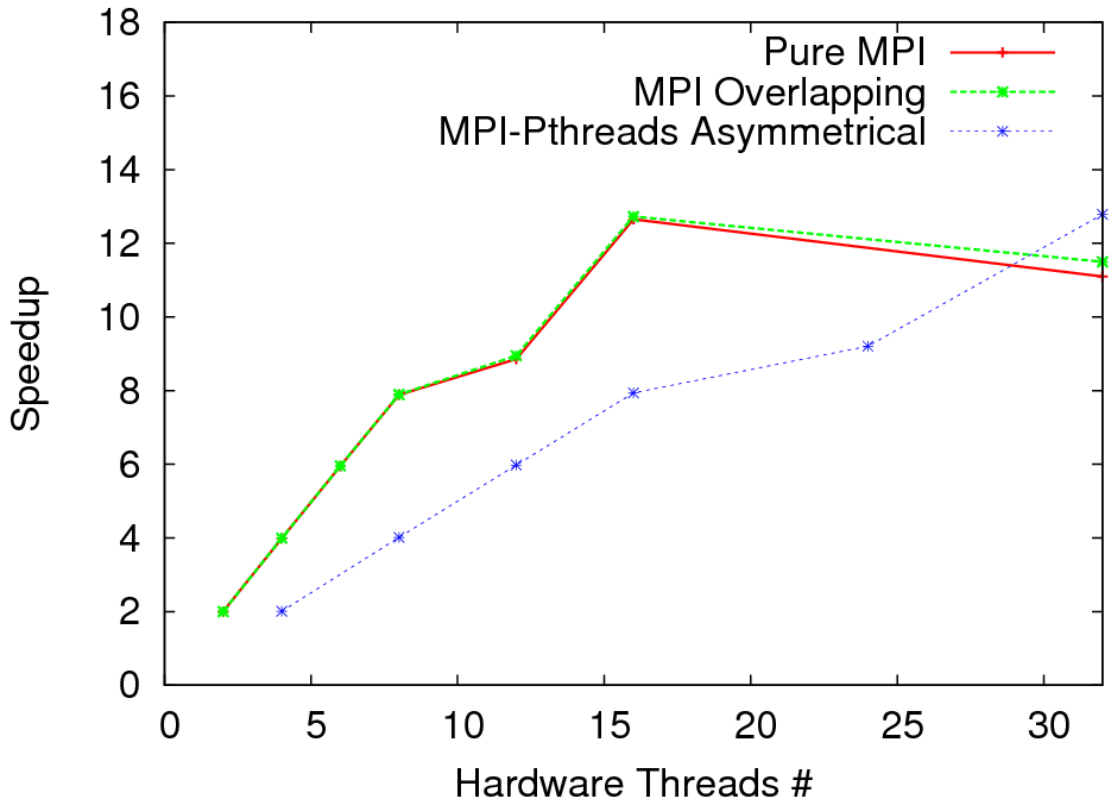
(b) Χώρος επαναλήψεων 3-D, 400x400x400

Σχήμα 6.2: Επίδραση του παράγοντα tiling στην επίδοση των παράλληλων προγραμμάτων της υλοποίησής μας σε δύο και τρεις διαστάσεις, για 32 hardware νήματα

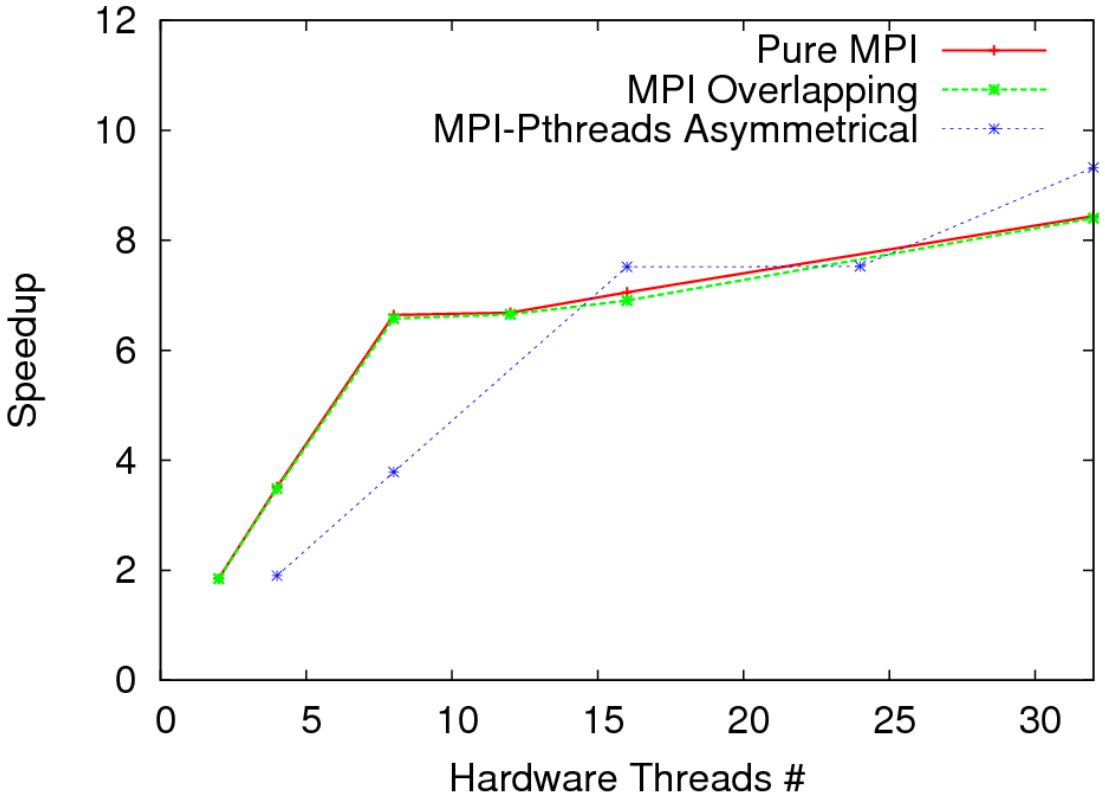
Από τις παραπάνω γραφικές βλέπουμε ότι μπορούμε να πετύχουμε κάποια αύξηση στην επίδοση, κάποιες φορές πολύ σημαντική, με την επιλογή του κατάλληλου μεγέθους tile. Η εύρεση του κατάλληλου μεγέθους tile έχει συζητηθεί αρκετά ερευνητικά, όπως για παράδειγμα στο [25]. Εμείς θα αρκεστούμε στη χρήση του βέλτιστου αποτελέσματος που πήραμε πειραματικά για το υπόλοιπο των αποτελεσμάτων.

6.2.3 Αποτίμηση της Συμπεριφοράς (Scalability) των Υλοποιήσεων

Σε αυτή την παράγραφο μελετούμε το πώς συμπεριφέρονται οι παράλληλες υλοποιήσεις μας στον αριθμό των επεξεργαστών στους οποίους ανατίθενται για εκτέλεση. Το scalability αφορά δύο παράγοντες: (α) την υλοποίηση του περιβάλλοντος MPI την οποία χρησιμοποιούμε (την οποία θεωρούμε βέλτιστη) και (β) το πόσο αποδοτικά παραλληλοποιούμε τον σειριακό αλγόριθμο μέσα στην υλοποίησή μας, σύμφωνα με όσα αναφέρονται στην παράγραφο 1.2.1. Επικεντρωνόμαστε και πάλι στις τρεις εκδώσεις που χρησιμοποιήσαμε σιωπηλά και στην προηγούμενη παράγραφο, εκείνες της βασικής έκδοσης, της έκδοσης με επικαλυπτόμενη δρομολόγηση και της έκδοσης με δύο ασύμμετρα νήματα, που ενδιαφέρει περισσότερο. Τα αποτελέσματα των μετρήσεών μας παρουσιάζονται στο ακόλουθο σχήμα.



(a) Χώρος επαναλήψεων 2-D, 6000x6000



(b) Χώρος επαναλήψεων 3-D, 400x400x400

Σχήμα 6.3: Συμπεριφορά των διαφόρων εκδόσεων των προγραμμάτων σε σχέση με τον αριθμό των hardware νημάτων στα οποία ανατίθενται για εκτέλεση

Από τις γραφικές του παραπάνω σχήματος παρατηρούμε ότι τα προγράμματά μας παρουσιάζουν αρκετά καλή συμπεριφορά, με την υλοποίηση με δύο ασύμμετρα νήματα να έχει καλύτερη συμπεριφορά από τις άλλες δύο, των οποίων η συμπεριφορά σχεδόν ταυτίζεται. Η κάπως μειωμένη επίδοση της έκδοσης με δύο ασύμμετρα νήματα σε σχέση με τις άλλες δύο για όλες τις περιπτώσεις αριθμού hardware νημάτων, εκτός της τελευταίας (32), οφείλεται στο ότι στις διεργασίες των δύο άλλων εκδόσεων ανατίθενται λογικοί επεξεργαστές από διαφορετικούς φυσικούς επεξεργαστές με αποτέλεσμα να έχουν όλους τους υπολογιστικούς πόρους στη διάθεσή του (ο φυσικός επεξεργαστής τρέχει σε κατάσταση ενός νήματος). Αντίθετα, στην έκδοση με τα ασύμμετρα νήματα γίνεται χρήση και των δύο λογικών επεξεργαστών ενός φυσικού πακέτου σε κάθε περίπτωση, με αποτέλεσμα να έχουμε ανταγωνισμό για τους μοιραζόμενους πόρους, για τους δυναμικά μοιραζόμενους πόρους, και διαθέσιμους τους μισούς πόρους, όπου αυτοί μοιράζονται στατικά. Όταν όμως φτάσουμε να χρησιμοποιούν όλοι και τους δύο λογικούς επεξεργαστές του ίδιου φυσικού επεξεργαστή (32 hardware νήματα) βλέπουμε την έκδοση με τα δύο ασύμμετρα νήματα να ανεβαίνει υψηλότερα των άλλων σε επίδοση. Σε αυτή ακριβώς την περίπτωση θα επικεντρωθούμε στη συνέχεια.

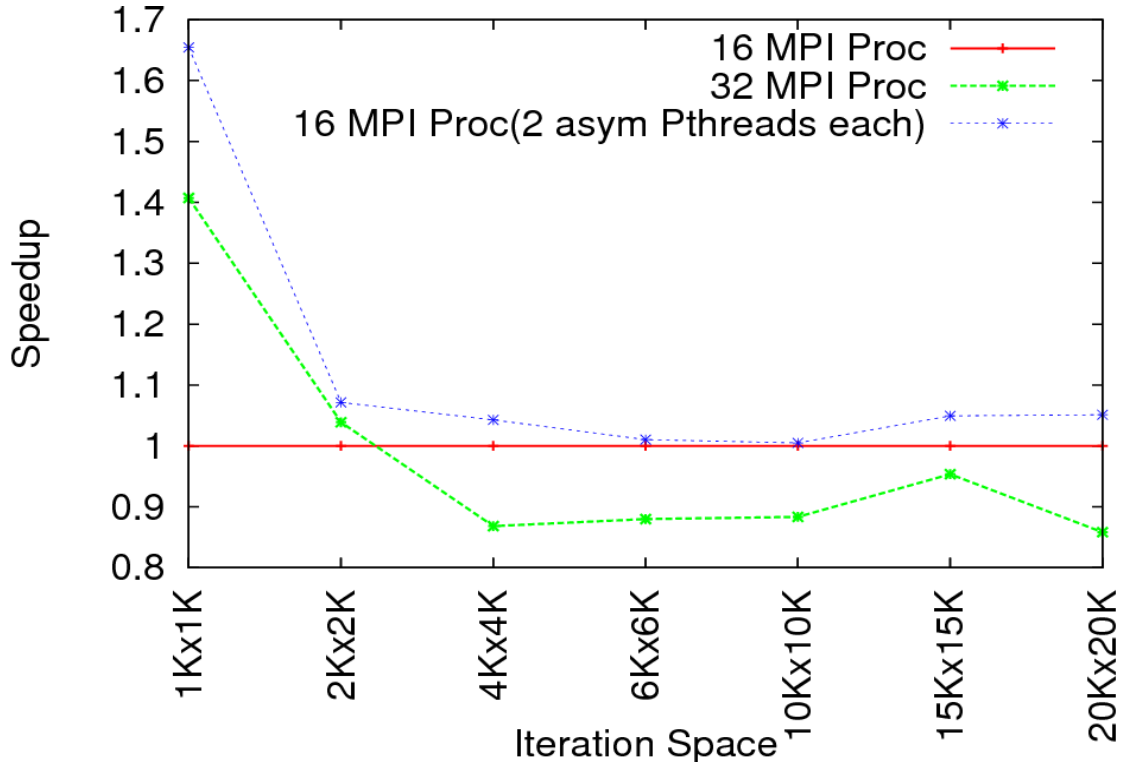
Αξίζει να παρατηρήσουμε και την επίδοση της παράλληλης έκδοσης με επικαλυπτόμενη δρομολόγηση χωρίς τη χρήση νημάτων. Βλέπουμε ότι η επίδοσή της ταυτίζεται σχεδόν με εκείνη της βασικής έκδοσης με δρομολόγηση χωρίς επικάλυψη. Εάν θυμηθούμε τη συζήτηση που κάναμε στην παράγραφο 5.5, θα δούμε ότι κάτι τέτοιο είναι αναμενόμενο. Από τη στιγμή που όλες οι λειτουργίες (υπολογισμοί, κλήσεις επικοινωνίας κλπ) εκτελούνται από την ίδια μονάδα επεξεργασίας είναι φυσικό να μην έχουμε αύξηση στην επίδοση. Μπορεί θεωρητικά να έχουμε δυνατότητες επικάλυψης, δεν τις εκμεταλλευόμαστε όμως αφού στην ουσία ο ίδιος επεξεργαστής εκτελεί όλο το φόρτο εργασίας. Υπάρχει μόνο ένα περιθώριο βελτίωσης που πιθανώς να οφείλεται στο ότι το γέμισμα των kernel buffer αποστολής και λήψης γίνεται από άλλη διεργασία (OS kernel), αλλά και πάλι μόνο στην περίπτωση που αυτή η λειτουργία εκτελείται από άλλο επεξεργαστή. Τις πραγματικές δυνατότητες της επικάλυψης βλέπουμε εάν τοποθετήσουμε τις δύο λειτουργίες, επεξεργασία και επικοινωνία σε δύο διαφορετικούς επεξεργαστές όπως συμβαίνει στην περίπτωση των ασύμμετρων νημάτων.

6.2.4 Αρχικά Αποτελέσματα Παράλληλης Έκδοσης με Δύο Ασύμμετρα Νήματα και Επικαλυπτόμενη Δρομολόγηση

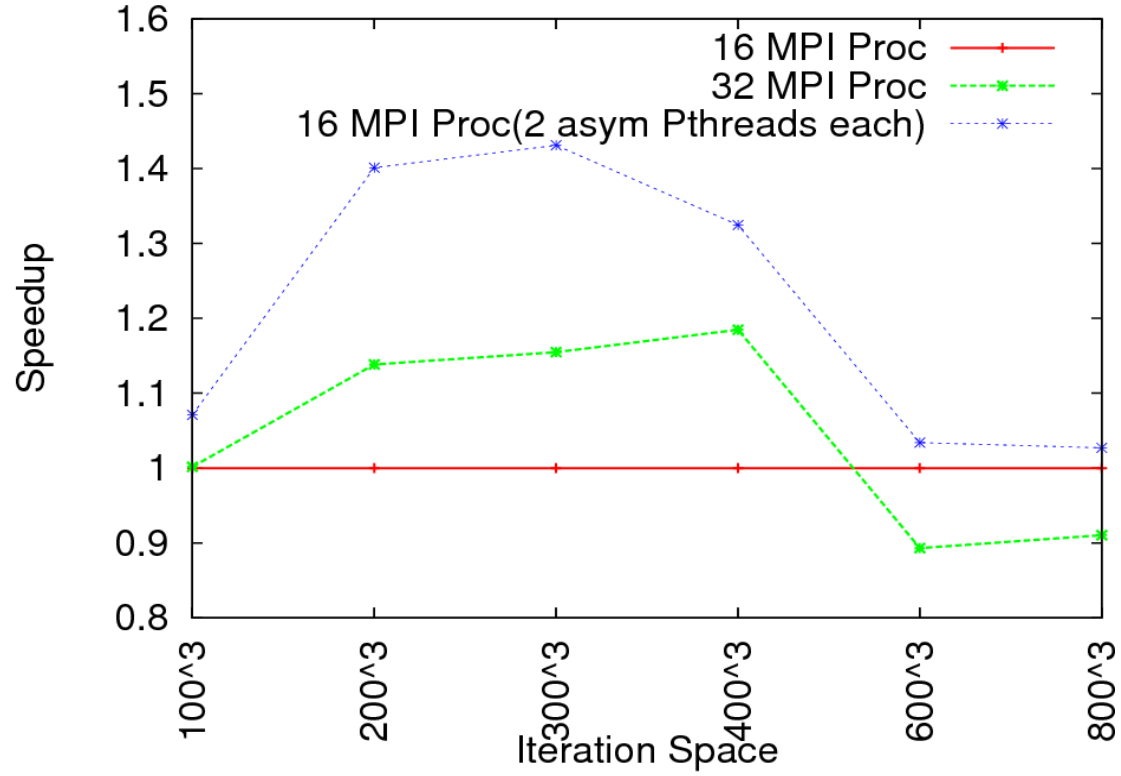
Επικεντρώνουμε την προσοχή μας στην περίπτωση της έκδοσης του παράλληλου προγράμματος με δύο ασύμμετρα νήματα, τα οποία τοποθετούνται στους δύο λογικούς επεξεργαστές του ίδιου φυσικού πακέτου. Να σημειώσουμε ότι η περίπτωση αυτή είναι εκείνη που ενδιαφέρει περισσότερο και αποτελεί τον καταληκτικό σκοπό της διπλωματικής μας εργασίας. Με βάση προηγούμενη δουλειά που έχει γίνει για την επικαλυπτόμενη δρομολόγηση [17], όπου σαν δεύτερη υπολογιστική μονάδα χρησιμοποιείται μία DMA engine) περιμένουμε να έχουμε κάποια βελτίωση όταν τοποθετήσουμε τις δύο λειτουργίες, εκτέλεσης υπολογισμών και επικοινωνίας, που εκτελούνται στον ίδιο βρόχο του χώρου επαναλήψεων, σε δύο διαφορετικές μονάδες επεξεργασίας. Εμείς επιλέξαμε να τοποθετήσουμε τις δύο αυτές λειτουργίες στους δύο διαφορετικούς λογικούς επεξεργαστές του ίδιου φυσικού επεξεργαστή με τεχνολογία Hyper-Threading. Από τη σχετική ερευνητική δουλειά που έχει γίνει στα πλαίσια του SMT αλλά και για την υλοποίηση της Intel στο Hyper-Threading, όπως στα [22] και [24], περιμένουμε να έχουμε αύξηση της επίδοσης λόγο ύπαρξης νημάτων ετερογενούς φορτίου εργασίας (που εκτελούν εντολές σε διαφορετικά δεδομένα) στους δύο λογικούς επεξεργαστές.

Στο Σχήμα 6.4 βλέπουμε μια πρώτη εκτίμηση των αποτελεσμάτων όταν συγκρίνουμε την έκδοση με δύο ασύμμετρα νήματα (16 MPI διεργασίες, 32 hardware νήματα) με εκείνη της βασικής παράλληλης έκδοσης με 16 και 32 MPI διεργασίες για διάφορα χωρία δύο και τριών χωρικών διαστάσεων. Η σύγκριση γίνεται με σημείο αναφοράς την παράλληλη έκδοση

σε 16 MPI διεργασίες, όπου και οι 16 διεργασίες τρέχουν σε διαφορετικούς φυσικούς επεξεργαστές σε κατάσταση ενός νήματος.



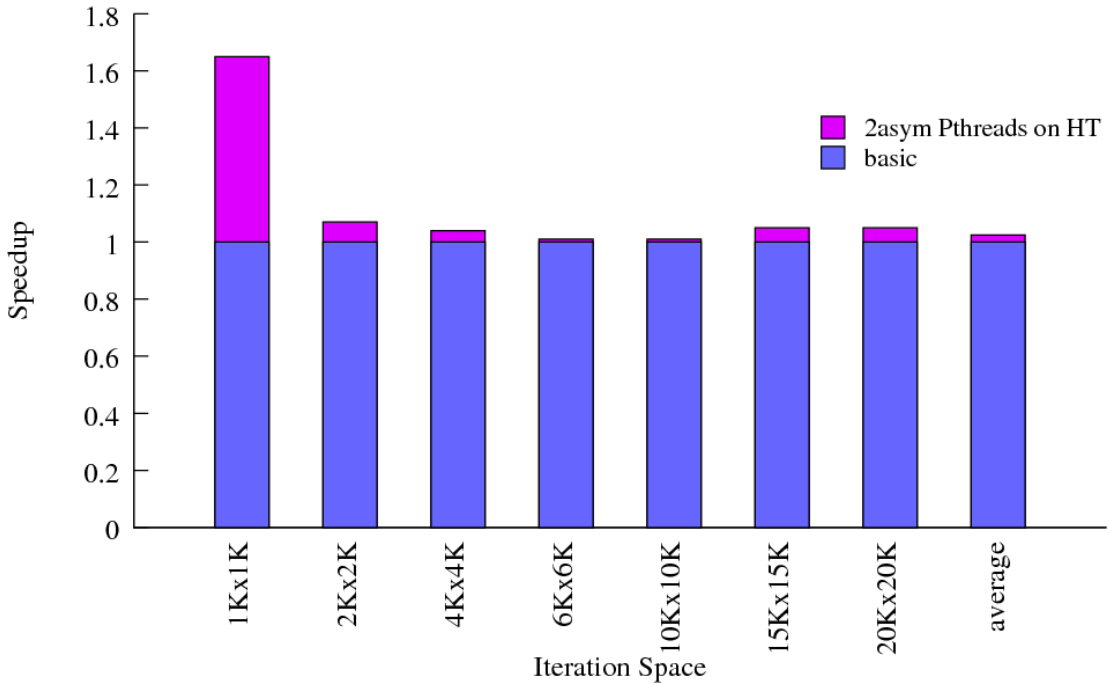
(a) Χώροι επαναλήψεων 2-D



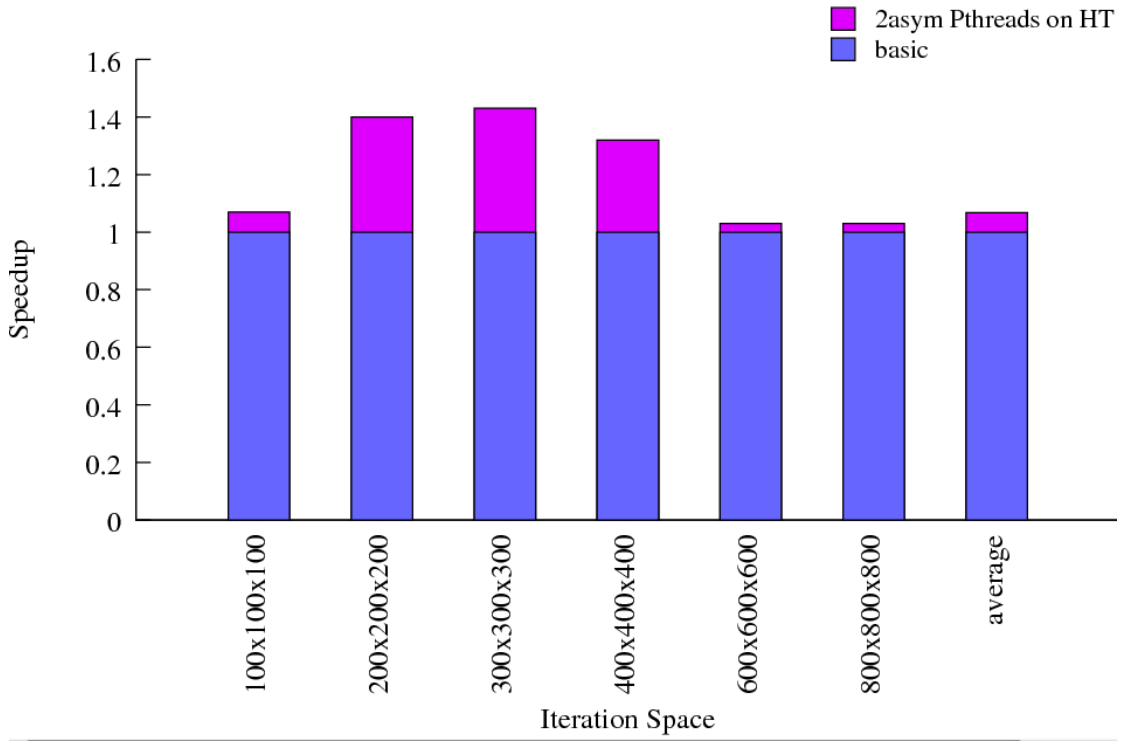
(b) Χώροι επαναλήψεων 3-D

Σχήμα 6.4: Αύξηση στη επίδοση από χρήση δύο ασύμμετρων νημάτων με επικαλυπτόμενη δρομολόγηση

Παρατηρούμε ότι έχουμε κάποια βελτίωση στην επίδοση από τη χρήση δύο ασύμμετρων νημάτων, που ποτέ δεν πέφτει κάτω από την επίδοση του βασικού παράλληλου με 16 MPI διεργασίες. Από την άλλη όταν χρησιμοποιούνται 32 MPI διεργασίες, δηλαδή έχουμε και πάλι χρήση και των δύο λογικών επεξεργαστών της τεχνολογίας Hyper-Threading με συμμετρικά όμως νήματα (που φτιάχνει το MPI) έχουμε μικρότερες επιπτώσεις στην επίδοση, που παίρνουν όμως και αρνητικές τιμές (κυρίως όσο αυξάνεται το μέγεθος του χώρου επαναλήψεων) και κατά τη διάρκεια των μετρήσεών μας δεν παρουσίασαν κάποια σταθερή τάση, αλλά αυξομειώσεις στην επιτάχυνση. Βέβαια στην περίπτωση των 32 MPI διεργασιών έχουμε και μεγαλύτερη ανταλλαγή πληροφορίας, λόγω τοπολογίας με περισσότερους επεξεργαστές. Για τους λόγους αυτούς θα επικεντρωθούμε στην αύξηση της επίδοσης που προσδίδει η έκδοση με τα δύο ασύμμετρα νήματα με χρήση του Hyper-Threading σε σχέση με την απλή παράλληλη έκδοση των 16 MPI διεργασιών. Τη σύγκριση αυτών των δύο περιπτώσεων βλέπουμε στο Σχήμα 6.5.



(a) Χώροι επαναλήψεων 2-D



(b) Χώροι επαναλήψεων 3-D

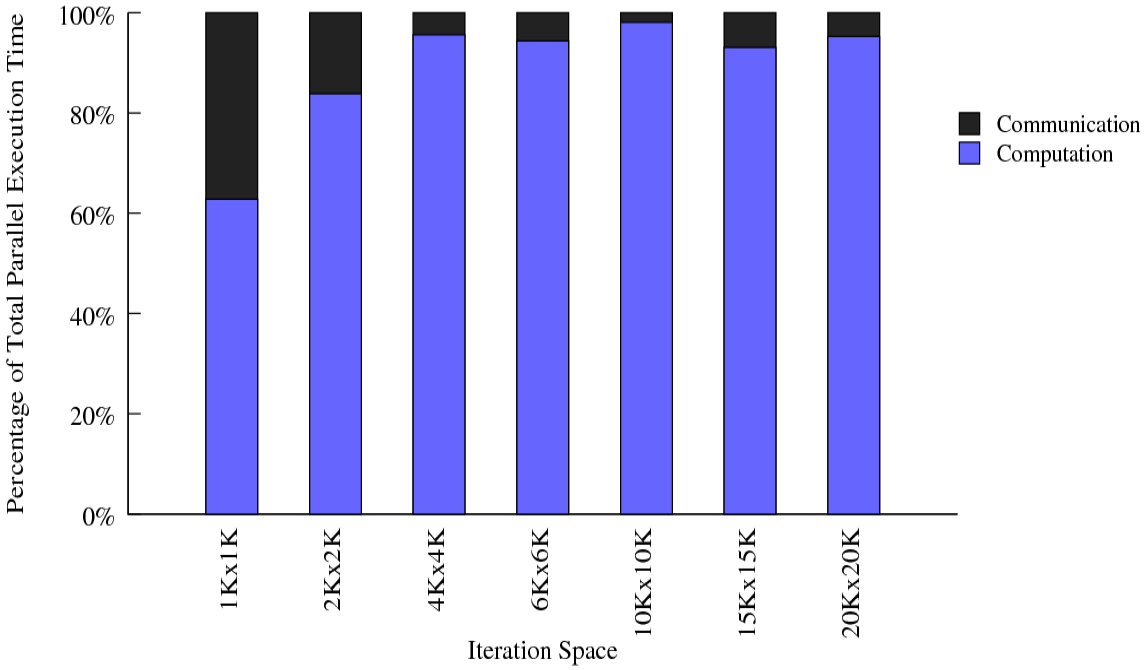
Σχήμα 6.5: Αύξηση στη επίδοση από χρήση δύο ασύμμετρων νημάτων με επικαλυπτόμενη δρομολόγηση

Βλέπουμε μία αύξηση στην επίδοση που κυμαίνεται από 1% (πρακτικά καθόλου βελτίωση) ,στη χειρότερη περίπτωση, μέχρι 65% στην καλύτερη, με μέση τιμή 5% στις δύο χωρικές διαστάσεις και 14% στις τρεις χωρικές διαστάσεις. Για την καλύτερη εξήγηση των αποτελεσμάτων, και με βάση τα όσα συζητούμε στην παράγραφο 4.2 και στα πλαίσια αυτών που προτείνονται στο [17], αποφασίσαμε να κάνουμε μία ανάλυση του χρονικού βήματος σε φάσεις επικοινωνίας και εκτέλεσης υπολογισμών, παίρνοντας τις κατάλληλες μετρήσεις.

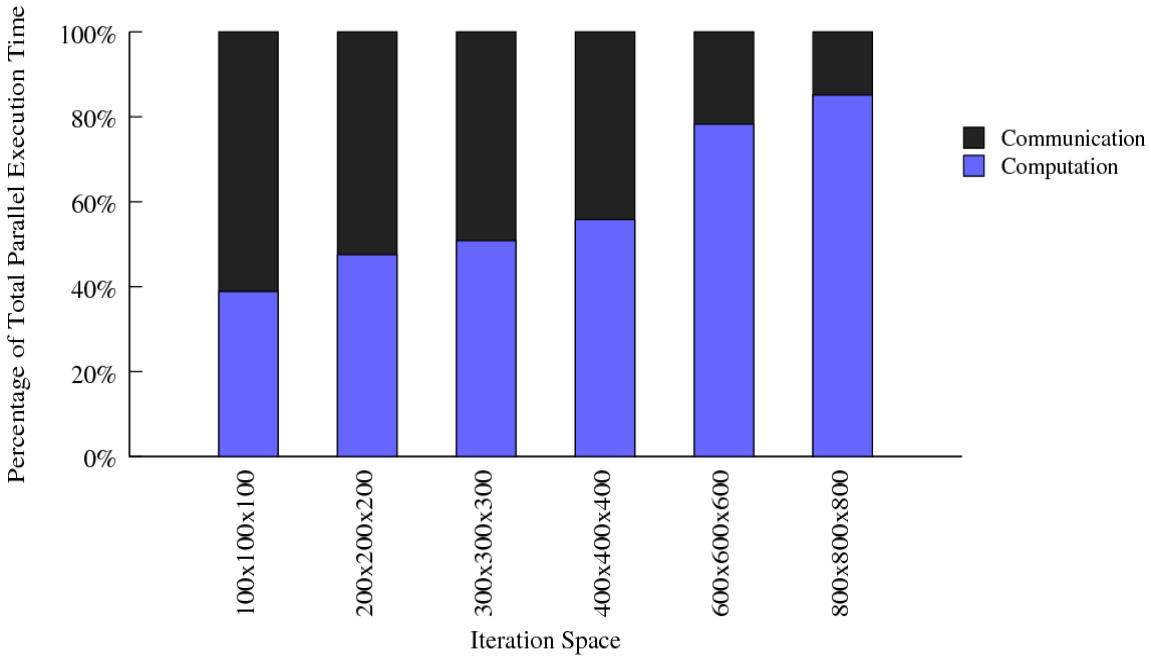
6.2.5 Ανάλυση Χρόνου Εκτέλεσης σε Επικοινωνία και Επεξεργασία

Προσπαθούμε σε αυτή την παράγραφο να κάνουμε μία ανάλυση του ολικού παράλληλου χρόνου εκτέλεσης σε χρόνο επικοινωνίας και χρόνο εκτέλεσης, σε μια προσπάθεια εκτίμησης των αποτελεσμάτων της προηγούμενης παραγράφου. Συγκεκριμένα, με βάση την ανάλυση του χρονικού βήματος του Σχήματος 4.4 και της συζήτησης στην παράγραφο 4.2 (που πηγάζουν από το [17]), περιμένουμε να έχουμε μεγαλύτερη αύξηση στην επίδοση όταν οι χρόνοι επικοινωνίας και επεξεργασίας είναι πολύ κοντά μεταξύ τους. Όταν δηλαδή το κόστος της επικοινωνίας και το κόστος της εκτέλεσης των υπολογισμών βρίσκονται σε κοντινά επίπεδα. Αυτό γιατί σε μια τέτοια περίπτωση έχουμε τη μεγαλύτερη δυνατότητα επικάλυψης των δύο φάσεων και συνεπώς την μεγαλύτερη προοπτική σε αύξηση της επίδοσης.

Ας μην ξεχνάμε ότι στο τέλος κάθε χρονικού βήματος (επανάληψης του βρόχου) έχουμε συγχρονισμό των δύο νημάτων επικοινωνίας και επεξεργασίας για να εξασφαλίσουμε ότι εκτελούν το ίδιο στιγμιότυπο. Έτσι φυσικό είναι να περιμένουμε καλύτερη επίδοση όταν τα δύο νήματα έχουν να εκτελέσουν ισοζυγισμένο φόρτο εργασίας. Αν για παράδειγμα το κόστος επικοινωνίας είναι πολύ μικρότερο περιμένουμε το νήμα επικοινωνίας να τελειώνει γρήγορα τη δουλειά του και να μπλοκάρει για αρκετό χρόνο περιμένοντας το νήμα επεξεργασίας να τελειώσει τους υπολογισμούς κάθε βρόχου. Οι μετρήσεις που πήραμε για τη χρονική ανάλυση συνοψίζονται στις γραφικές του παρακάτω σχήματος, όπου για λόγους παρουσίασης έχουμε κανονικοποιήσει τους χρόνους επικοινωνίας και επεξεργασίας σε σχέση με τον ολικό χρόνο εκτέλεσης.



(a) Χώροι επαναλήψεων 2-D

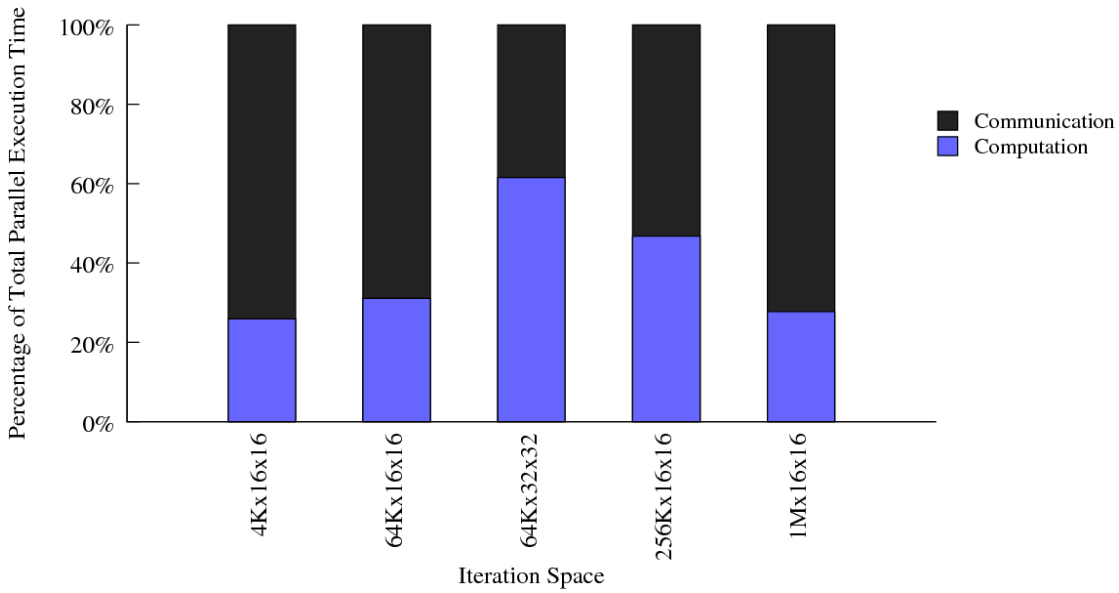


(b) Χώροι επαναλήψεων 3-D

Σχήμα 6.6: Ανάλυση του ολικού χρόνου εκτέλεσης σε επικοινωνία και εκτέλεση υπολογισμών, για το βασικό παράλληλο πρόγραμμα σε 16 MPI διεργασίες

Συγκρίνοντας τα αποτελέσματα του Σχήματος 6.6 με εκείνα του Σχήματος 6.5 που αντιστοιχούν στους ίδιους χώρους επαναλήψεων παρατηρούμε τα εξής: Έχουμε σημαντική αύξηση της επίδοσης όταν το κόστος επικοινωνίας βρίσκεται κοντά ή λίγο πάνω από το κόστος εκτέλεσης (χωρία $1K \times 1K$,, $200 \times 200 \times 200$ και $300 \times 300 \times 300$) ενώ όταν οι δύο χρόνοι απέχουν σημαντικά μεταξύ τους έχουμε μικρά κέρδη στην επίδοση (υπόλοιπα χωρία). Δηλαδή τα αποτελέσματα αυτά συνάδουν με αυτά που περιμέναμε να δούμε, με κάποια μικρή απόκλιση στο ότι παρατηρήσαμε σημαντικά αυξανόμενη απόδοση και σε περιπτώσεις όπου είχαμε πολύ μεγάλο κόστος επικοινωνίας σε σχέση με το κόστος υπολογισμού.

Με γνώμονα τα αποτελέσματα αυτά αποφασίσαμε να πάρουμε συμπληρωματικές μετρήσεις για μη τετραγωνικά χωρία με μεγαλύτερη τη μία διάσταση (έστω την διάσταση i_1) που παρουσιάζουν μεγαλύτερη ομοιογένεια ως προς το ισοζύγιο κόστους επικοινωνίας και υπολογισμών. Τις μετρήσεις που αφορούν τη χρονική ανάλυση για μη τετραγωνικά χωρία τριών διαστάσεων αυτά βλέπουμε στο Σχήμα 6.7.

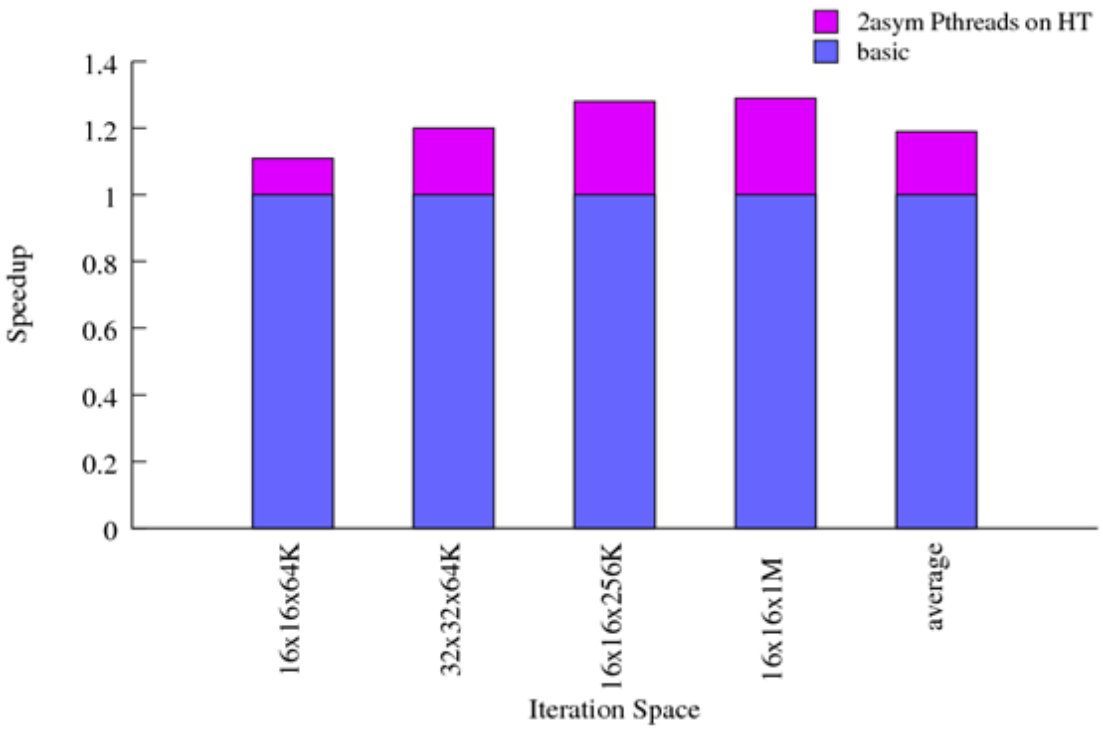


Σχήμα 6.7: Ανάλυση του ολικού χρόνου εκτέλεσης σε επικοινωνία και εκτέλεση υπολογισμών σε μη τετραγωνικά χωρία, για το βασικό παράλληλο πρόγραμμα σε 16 MPI διεργασίες

Βλέπουμε ότι έχουμε μια πιο ισοζυγισμένη εικόνα μεταξύ χρόνου επεξεργασίας και υπολογισμού σε αυτή την περίπτωση, με κάποιο μεγαλύτερο κόστος επικοινωνίας στο μικρότερο και μεγαλύτερο χωρίο.

6.2.6 Συμπληρωματικά Αποτελέσματα Παράλληλης Έκδοσης με Δύο Ασύμμετρα Νήματα και Επικαλυπτόμενη Δρομολόγηση

Τέλος παρουσιάζουμε τα αποτελέσματα από τις συμπληρωματικές μετρήσεις, όπως προέκυψαν από τη συζήτηση της προηγούμενης παραγράφου. Τα αποτελέσματα αυτά φαίνονται στο Σχήμα 6.8 και αποτελούν σαφή βελτίωση των αρχικών, με περισσότερη ομοιογένεια στην αύξηση της επίδοσης κατά μήκος των διαφόρων χωρίων επανάληψης. Συγκεκριμένα παρατηρείται αύξηση της επίδοσης που κυμαίνεται από 11% μέχρι 29% σε σχέση με τη βασική έκδοση και με μέση βελτίωση της τάξης του 24%. Η εκτέλεση σε αυτά τα χωρία αυξάνει τη μέση αύξηση επίδοσης σε σχέση με τα τετραγωνικά κατά 10% (από 14% σε 24%).



Σχήμα 6.8: Αύξηση στη επίδοση από χρήση δύο ασύμμετρων νημάτων με επικαλυπτόμενη δρομολόγηση σε μη τετραγωνικά χωρία τριών διαστάσεων

6.3 Συμπεράσματα και Μελλοντική Εργασία

Από τα αποτελέσματα που παρουσιάστηκαν και συζητήθηκαν σε αυτό το κεφάλαιο έχουμε καταλήξει σε αρκετά συμπεράσματα. Πρώτα παρουσιάσαμε ότι η υλοποίηση με δύο συμμετρικά νήματα δεν εμφανίζει αξιοσημείωτες μεταβολές στην επίδοση και δεν ασχοληθήκαμε μαζί της στη συνέχεια. Δεύτερον είδαμε ότι η προσθήκη απλής μορφής tiling στα παράλληλη μας προγράμματα επιφέρει μεταβολές στην επίδοση από θετικές μέχρι αρνητικές ανάλογα με το μέγεθος του tile που χρησιμοποιούμε. Με την κατάλληλη επιλογή μεγέθους tile μπορούμε να επιτύχουμε σημαντική βελτίωση στην επίδοση.

Στη συνέχεια είδαμε το πώς συμπεριφέρεται η έκδοση με επικαλυπτόμενη δρομολόγηση σε σχέση με την επίδοση. Από τις μετρήσεις μας καταλήξαμε στο συμπέρασμα ότι εφόσον οι δύο φάσεις επικοινωνίας και υπολογισμού δεν ανατεθούν σε διαφορετικές υπολογιστικές μονάδες δεν επωφελούμαστε της ανεξαρτησίας των δύο φάσεων και της μεταξύ τους επικάλυψης. Όταν δηλαδή εκτελούνται οι δύο φάσεις από τον ίδιο επεξεργαστή έχουμε ελάχιστα κέρδη στην απόδοση. Ακολούθως παρατηρήσαμε τη συμπεριφορά των προγραμμάτων μας σε σχέση με τον αριθμό των hardware νημάτων στα οποία εκτελούνται. Η συμπεριφορά τους ήταν αρκετά καλή, πλησιάζοντας την ιδανική γραμμική περίπτωση, με την υλοποίηση με τα δύο ασύμμετρα νήματα να παρουσιάζει την καλύτερη συμπεριφορά στην κλιμάκωση της επίδοσης.

Τέλος επικεντρώσαμε την προσοχή μας στην περίπτωση που αποτέλεσε και τον πυρήνα της διπλωματικής μας εργασίας, εκείνη της υλοποίησης του παράλληλου προγράμματος με επικαλυπτόμενη δρομολόγηση, με ανάθεση των δύο φάσεων επικοινωνίας και επεξεργασίας σε δύο ασύμμετρα νήματα που τοποθετούνται στους δύο ξεχωριστούς λογικούς επεξεργαστές που παρέχει ένας φυσικός επεξεργαστής με Hyper-Threading. Από τα αποτελέσματα των μετρήσεων είδαμε ότι μπορούμε να κερδίσουμε σημαντικά σε επίδοση χρησιμοποιώντας τους δύο λογικούς επεξεργαστές για να εκτελέσουν τα δύο ανομοιογενή νήματα επικοινωνίας και εκτέλεσης υπολογισμών στα πλαίσια δρομολόγησης με επικάλυψη. Καταλήξαμε επίσης στο ότι η μέγιστη δυνατή επίδοση παρατηρείται σε χώρους επανάληψης που παρουσιάζουν ισοζυγισμένο κόστος επικοινωνίας σε σχέση με το κόστος υπολογισμών ή σε μερικές περιπτώσεις πολύ μεγάλο κόστος επικοινωνίας. Η υλοποίηση αυτή παρουσιάζει μεγαλύτερη ανεκτικότητα σε μεγάλα κόστη επικοινωνίας λόγω της επικάλυψης και της ξεχωριστής εκτέλεσης των δύο φάσεων από δύο νήματα.

Ως μελλοντική δουλειά για βελτίωση και συνέχιση των όσων παρουσιάσαμε στα πλαίσια της διπλωματικής εργασίας θα μπορούσαμε να επικεντρωθούμε στα εξής:

- Επέκταση της υλοποίησης της προσομοίωσης στις n-διαστάσεις, με ενιαίο τρόπο ώστε ανάλογα με το χώρο επαναλήψεων και της τοπολογίας επεξεργαστών που δίνεται ως είσοδος να έχουμε και ανάλογη εκτέλεση.
- Δημιουργία εργαλείου που να δέχεται ως είσοδο ένα παράλληλο πρόγραμμα MPI εμφωλευμένων βρόχων και να του προσθέτει δυνατότητες επικαλυπτόμενης δρομολόγησης και καταμερισμό των φάσεων επικοινωνίας και υπολογισμού σε δύο ασύμμετρα νήματα. Για το σκοπό αυτό μπορεί να συνεργαστεί με διάφορα εργαλεία παραγωγής παράλληλου κώδικα MPI που έχουν ήδη παραχθεί.
- Έλεγχο του κόστους των μηχανισμών συγχρονισμού και πιθανή τροποποίησή τους για μείωση του κόστους αυτού.
- Ανάλυση της επίδοσης της έκδοσης με ασύμμετρα νήματα για πολύ μεγαλύτερο αριθμό χώρων επαναλήψεων με λεπτομερέστερη ανάλυση του χρονικού βήματος σε κόστος επικοινωνίας και υπολογισμού για καλύτερη κατανόηση των τάσεων που παρουσιάζει στην αύξηση της επίδοσης.
- Επέκταση των υλοποιήσεων που υποστηρίζουν tiling για πιο αποδοτική και ευέλικτη υποστήριξη του tiling ώστε να εκμεταλλεύονται καλύτερα αυτό τον παράγοντα.

Βιβλιογραφία

1. **Almassi G.S., Gottlieb A.** *Highly Parallel Computing.* Redwood City, CA : Benjamin/Cummings, 1989.

2. **Hennessy J.L., Jouppi N.** *Computer Technology and Architecture: An Evolving Interaction.* 9, s.l. : IEEE Computer, 1991, Τόμ. 24, σσ. 18-29.

3. **Amdahl, G.M.** *Validity of the Single-Processor Approach to Achieving Large Scale Computing Abilities.* Atlantic City, N.J. : AFIPS Press, 1967. AFIPS Conference Proceedings. Τόμ. 30, σσ. 483-485.

4. **Eggers S.J., Emer J, Levy H.M., Lo J.L., Stamm R., Tullsen D.M.** Simultaneous Multithreading: A Platform for Next-generation Processors. *IEEE Micro.* 1997, Τόμ. 17, 5.

5. **Marr D, Binns F, Hill D, Hinton G, Koutafy D, Miller JA, Upton M.** *Hyper-Threading Technology Architecture and Microarchitecture.* s.l. : Intel Technology Journal, 2002, Τόμ. 6, σσ. 4-15.

6. **Intel Corporation.** *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.* 2001.

7. **Intel Corporation.** *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide.* 2001.

8. **Shang W., Fortes J.A.B. .** Indepentent Partitioning of Algorithms with Uniform Dependencies. *IEEE Trans. Comput.* 1992, Τόμ. 41, 2, σσ. 190-206.

9. **E.H., Hollander.** Partitioning and Labeling Loops by Unimodular Transformations. *IEEE Trans. on Parallel and Distributed Systems.* 1992, Τόμ. 9, 5, σσ. 465-476.

10. **Tsanakas P., Koziris N., Papakonstantinou G.** Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays. *IEEE Trans. on Parallel and Distributed Systems.* 2004, Τόμ. 57, 2, σσ. 941-955.

11. **Drossitis I., Goumas G., Koziris N., Papakonstantinou G., Tsanakas P.** *Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces.* Toronto, Canada : s.n., 2000. International Conference on Parallel Processing. σσ. 469-476.

12. **D'Hollander E.** *Partitioning and Labeling of Loops by Unimodular Transformations.* 4, s.l. : IEEE trans., 1992, Τόμ. 3, σσ. 465-476.

13. **S.Lotfi, S. Parsa.** *A New Genetic Algorithm for Loop Tiling.* 3, s.l. : Journal of Supercomputing, 2006, Τόμ. 37, σσ. 249-269.

14. **M. Kandemir, R. Bordwekar, AChoudhary, J. Ramanujam.** *A Unified Tiling Approach for Out-of-Core Computations.* Aachen, Germany:Forschungzentrum Julich GmbH : s.n., 1996. Workshop on Compilers for Parallel Computers. σσ. 323-334.

15. **G.E. Karniadakis, R.M. Kirby.** *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implemantation.* s.l. : Campridge University Press, 2002.

16. **Andronikos T., Koziris N. , Papakonstantinou G., Tsanakas P.** *Optimal Scheduling for UEI/UET-UCT Generilized N-Dimensional Grid Task Graphs.* 1999, Journal of Parallel and Distributed Computing, Τόμ. 57, σσ. 140-165. no.2.

17. **Goumas G., Sotiropoulos A., Koziris N.** *Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping.* San Francisco, California : IEEE Press, 2001. International Parallel and Distributed Processing Symposium (IPDPS2001).

18. **E.Hodzic, W. Shang.** *On Supernode Transformation with Minimized Total Running Time.* 5, 1998, IEEE Trans. on Parallel and Distributed Systems, Τόμ. 9, σσ. 417-428.

19. **Andronikos, N. Koziris, G. Papakonstantinou, P. Tsanakas.** *Optimal Scheduling for UEI/UET-UCT Generilized N-Dimensional Grid Task Graphs.* **T.** 1999, Journal of Parallel and Distributed Computing, Τόμ. 57, σσ. 140-165. no.2.

20. **Eggers S.J., Emer J, Levy H.M., Lo J.L., Stamm R., Tullsen D.M.** Converting Thread-Level Parallelism into Instruction Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems.* 1997, σσ. 322-354.

21. **Snavely A., Tullsen D.** *Symbiotic Job-Scheduling for a Simultaneous Multithreading Architecture.* 2002. Eighth Internationa Conference on Architectural Support for Programming Languages and Operating Systems.

22. **Tuck N, Tullsen D.** *Initial observations of the simultaneous multithreading Pentium 4 processor.* New Orleans, LA : s.n., 2003. 12th international conference on parallel architectures and compilation techniques (PACT '03).

23. **Tullsen D, Eggers S, Emer J, Levy H, Lo J, Stamn R.** *Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor.* Philadelphia,

PA : s.n., 1996. 23rd annual international symposium on computer architecture (ISCA). σσ. 191-202.

24. **Bulpin J., Bratt I.** *Multiprogramming Performance on the Pentium 4 with Hyper Threading.* Munich : s.n., 2004. The third annual workshop on duplicating, deconstructing and debunking. σσ. 53-62.

25. **Goumas Γ., Drosinos N., Koziris N.** Communication-aware Supernode Shape. *IEEE Transactions on Parallel and Distributed Systems.* 2008.

26. **Hennessy J.L., Patterson D.A.** *Computer Organization and Design.* 3rd Edition. San Diego : Elsevier, 2005.

27. **Hennessy J.L., Patterson D.A.**. *Computer Architecture: A Quantative Approach.* 2nd Edition. San Fransisco : Morgan Kaufmann, 1996.

28. *April 18,1994(PowerPc 604), May 30, 1994 (UltraSPARC 1), October 3, 1994 (HP PA-8000), October 24, 1994 (R10000).* s.l. : Microprocessor Report.

29. **Marr D.T., Binns F., Hill D.L., Hinton G., Koutafy D.A., Miller J.A., Upton M.** *Hyper-Threading Technology Architecture and Microarchitecture.* s.l. : Intel Corporation, 2002.

Παράρτημα Α

Το Περιβάλλον Ανταλλαγής Μηνυμάτων MPI

Α.1 Γενικά

Σε αυτό το παράρτημα κάνουμε μια εισαγωγή στις βασικές έννοιες του περιβάλλοντος ανταλλαγής μηνυμάτων MPI (Message Passing Interface), δίνοντας έμφαση στις λειτουργίες που χρησιμοποιήσαμε στα πλαίσια των παράλληλων προγραμμάτων που υλοποιήσαμε.

Α.2 Εισαγωγή στο MPI

Εξορισμού, ένας ακολουθιακός υπολογιστικός αλγόριθμος είναι φορητός σε οποιαδήποτε αρχιτεκτονική υποστηρίζει το ακολουθιακό μοντέλο. Ωστόσο, οι προγραμματιστές ζητούν πολύ περισσότερα από αυτό: θέλουν η δική τους υλοποίηση αυτού του αλγορίθμου στη μορφή ενός συγκεκριμένου προγράμματος να είναι μεταφέρσιμος – δηλαδή θέλουν φορητότητα πηγαίου κώδικα.

Το ίδιο ισχύει και για τα προγράμματα ανταλλαγής μηνυμάτων και αυτό αποτελεί το κίνητρο πίσω από το MPI. Το MPI παρέχει τη δυνατότητα της φορητότητας πηγαίου κώδικα προγραμμάτων που γράφτηκαν σε γλώσσα προγραμματισμού C ή Fortran για ένα σύνολο από αρχιτεκτονικές.

Παρόλο που η βασική ιδέα της επικοινωνίας μεταξύ διεργασιών με ανταλλαγή μηνυμάτων μεταξύ τους έχει κατανοηθεί εδώ και αρκετά χρόνια, είναι σχετικά πρόσφατη η υλοποίηση συστημάτων ανταλλαγής μηνυμάτων που επιτρέπουν τη φορητότητα πηγαίου κώδικα.

Το MPI αποτέλεσε την πρώτη προσπάθεια παραγωγής ενός περιβάλλοντος ανταλλαγής μηνυμάτων που αγγίζει όλη την κοινότητα παράλληλης επεξεργασίας. Δύο χρόνια μετά από τη δημιουργία του “MPI Forum” προέκυψε ένα έγγραφο που περιγράφει το πρωτόκολλο του MPI.

Α.3 Προγράμματα MPI

Α.3.1 Προκαταρτικά

Το MPI αποτελεί ουσιαστικά μια βιβλιοθήκη. Μία διεργασία MPI αποτελείται από ένα πρόγραμμα C ή Fortran που επικοινωνεί με άλλες MPI διεργασίες καλώντας ρουτίνες του MPI. Οι ρουτίνες του MPI παρέχουν στον προγραμματιστή ένα συνεπές περιβάλλον που ισχύει για ένα πλήθος διαφορετικών πλατφορμών.

Α.3.2 MPI Χειριστές (Handles)

Το MPI διατηρεί εσωτερικές δομές δεδομένων που σχετίζονται με την επικοινωνία και άλλους παράγοντες, και οι χρήστες μπορούν να αναφερθούν σε αυτές μέσω *handles*. Οι handles επιστρέφουν στον χρήστη από κάποιες κλήσεις MPI και μπορούν να χρησιμοποιηθούν σε άλλες κλήσεις του MPI.

Οι handles μπορούν να αντιγραφούν από τις συνήθεις λειτουργίες ανάθεσης της C ή της Fortran.

Α.3.3 Σφάλματα MPI (Errors)

Γενικά, οι C ρουτίνες MPI επιστρέφουν ένα int που περιέχει τον κωδικό σφάλματος. Η προκαθορισμένη ενέργεια από το MPI όταν αναγνωρίσει ένα σφάλμα είναι ο τερματισμός της παράλληλης εκτέλεσης, αντί να επιστρέψει με ένα κωδικό σφάλματος.

Λόγω της δυσκολίας υλοποίησης του MPI κατά μήκος μιας ποικιλίας από αρχιτεκτονικές, ένα πλήρες σύνολο από αναγνωρισμένα σφάλματα και τους αντίστοιχούς τους κωδικούς δεν υπάρχει. Ένα πρόγραμμα μπορεί να είναι λανθασμένο με την έννοια ότι δεν καλεί τις ρουτίνες του MPI ορθά, αλλά το πρωτόκολλο δεν εγγυάται ότι θα ανιχνεύσει όλα αυτά τα λάθη.

Α.3.4 Αρχικοποίηση του MPI

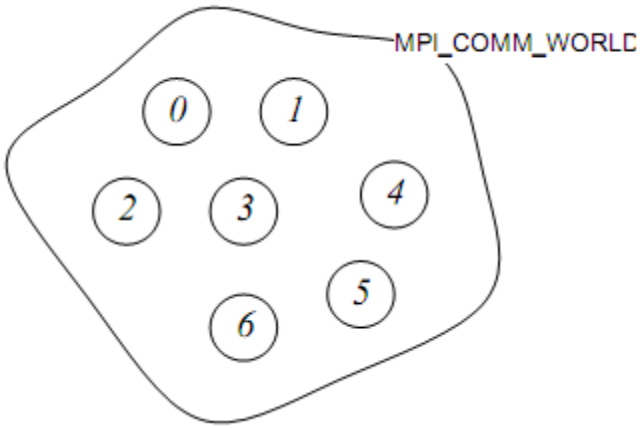
Η πρώτη ρουτίνα του MPI που καλείται από κάθε πρόγραμμα MPI πρέπει να είναι η ρουτίνα αρχικοποίησης MPI_Init[4]. Η έκδοση της ρουτίνας αυτής για τη C δέχεται ως παραμέτρους τις παραμέτρους της συνάρτησης main, τα argc και argv.

```
int MPI_Init(int *argc, char **argv)
```

[4] Υπάρχει μια εξαίρεση σε αυτό, η ρουτίνα MPI_Initialized που επιτρέπει στον προγραμματιστή να ελέγξει εάν η MPI_Init έχει ήδη κληθεί.

Α.3.5 Η Δομή MPI_COMM_WORLD και οι Δομές Επικοινωνίας

Η ρουτίνα MPI_Init ορίζει κάτι που ονομάζεται MPI_COMM_WORLD για κάθε διεργασία που την καλεί. Το MPI_COMM_WORLD είναι μία δομή ορισμού επικοινωνίας (communicator). Οι δομές αυτού του τύπου χρειάζονται σε κάθε κλήση επικοινωνίας του MPI σαν παράμετρο και οι MPI διεργασίες μπορούν να επικοινωνούν μεταξύ τους μόνο εφόσον μοιράζονται μία τέτοια δομή.



Σχήμα Α.1: Η Προκαθορισμένη δομή ορισμού επικοινωνίας για εφτά διεργασίες. Οι αριθμοί αντιπροσωπεύουν τον βαθμό της κάθε διεργασίας.

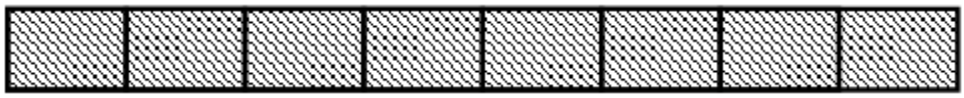
Ο κάθε communicator επικοινωνίας περιέχει ένα γκρουπ, μία λίστα από διεργασίες. Δεύτερον, ένα γκρουπ είναι στην πράξη τοπικό για μία συγκεκριμένη διεργασία. Η φαινομενική αντίθεση αυτής της πρότασης εξηγείται ως εξής: το γκρουπ στα πλαίσια ενός communicator έχει προσυμφωνηθεί από τις διεργασίες τη στιγμή που ορίστηκε. Οι διεργασίες κατατάσσονται αριθμούνται συνεχόμενα αρχίζοντας από το 0, και ο αριθμός της κάθε διεργασίας είναι γνωστός ως ο βαθμός της. Ο βαθμός αυτός ξεχωρίζει την κάθε διεργασία στα πλαίσια του communicator. Αξίζει να σημειώσουμε ότι γενικά μία διεργασία μπορεί να έχει πολλούς communicator και συνεπώς να ανήκει σε διάφορα γκρουπ, τυπικά με διαφορετικό βαθμό σε κάθε γκρουπ.

Α.3.6 Τερματισμός του MPI

Ένα πρόγραμμα MPI πρέπει να καλεί τη ρουτίνα MPI_Finalize όταν όλες οι επικοινωνίες έχουν τερματιστεί. Αυτή η ρουτίνα καθαρίζει όλες τις δομές δεδομένων του MPI και όλα τα σχετικά. Δεν ακυρώνει επικοινωνίες που βρίσκονται σε εξέλιξη, για αυτό είναι ευθύνη του προγραμματιστή να εξασφαλίσει ότι όλες οι επικοινωνίες έχουν τερματιστεί. Από τη στιγμή που κληθεί αυτή η ρουτίνα, καμία άλλη κλήση σε ρουτίνα του MPI δεν μπορεί να εκτελεστεί, ούτε καν εάν ξανακαλεστεί η ρουτίνα MPI_Init.

Α.4 Τι Περιέχεται σε ένα Μήνυμα

Ένα μήνυμα MPI δεν είναι τίποτε άλλο από ένας πίνακας από στοιχεία ενός συγκεκριμένου τύπου δεδομένων MPI (MPI datatype).



Σχήμα Α.2: Ένα μήνυμα MPI

Όλα τα μηνύματα MPI έχουν τύπο υπό την έννοια ότι ο τύπος των περιεχομένων τους πρέπει να είναι συγκεκριμένος κατά την αποστολή και τη λήψη. Οι βασικοί τύποι δεδομένων του MPI που αντιστοιχούν στους βασικούς τύπους δεδομένων της C με τον τρόπο που φαίνεται στον Πίνακα Α.1.

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Πίνακας Α.1: Οι βασικοί τύποι δεδομένων της C στο MPI

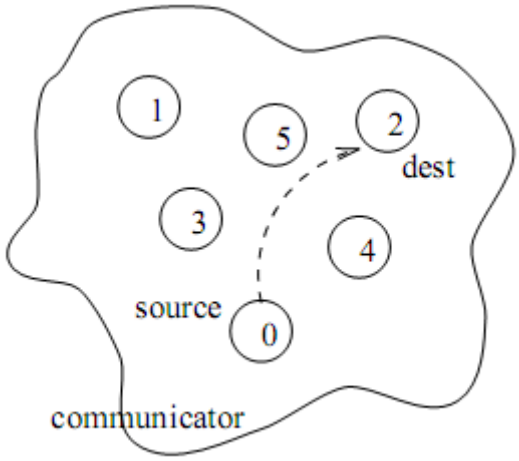
Υπάρχουν κανόνες για το ταίριασμα τύπων δεδομένων και, με μερικές εξαιρέσεις, ο τύπος δεδομένων που ορίζεται σε μία αποστολή πρέπει να ταιριάζει με τον τύπο δεδομένων που ορίζεται στη λήψη. Το μεγάλο πλεονέκτημα αυτού είναι ότι το MPI μπορεί να υποστηρίξει ετερογενείς παράλληλες αρχιτεκτονικές, π.χ. αρχιτεκτονικές βασισμένες σε διαφορετικού επεξεργαστές, επειδή μπορεί να γίνει μετατροπή τύπων όπου αυτό κριθεί αναγκαίο. Συνεπώς δύο επεξεργαστές μπορούν να αντιπροσωπεύουν, για παράδειγμα, έναν

ακέραιο με διαφορετικούς τρόπους, αλλά οι διεργασίες MPI σε αυτούς τους επεξεργαστές μπορούν να χρησιμοποιήσουν το MPI για την αποστολή μηνυμάτων ακεραίων χωρίς να είναι ενήμεροι για την ύπαρξη ετερογένειας.

Πιο σύνθετοι τύπο δεδομένων μπορούν να παραχθούν στο χρόνο εκτέλεσης. Αυτοί ονομάζονται παραγόμενοι τύπο δεδομένων και παράγονται από τους βασικούς. Μπορούν να χρησιμοποιηθούν για την αποστολή διανυσμάτων, structs της C κτλ.

Α.5 Επικοινωνία Σημείου-Προς-Σημείο

Μία επικοινωνία σημείου-προς-σημείο αφορά πάντα ακριβώς δύο επεξεργαστές. Η μία διεργασία αποστέλλει ένα μήνυμα στην άλλη. Αυτό ξεχωρίζει αυτό τον τύπο επικοινωνίας από τον άλλο τύπο επικοινωνίας του πρωτοκόλλου MPI, τη συλλογική επικοινωνία, που εμπλέκει ένα γκρουπ διεργασιών κάθε φορά.



Σχήμα Α.3: Στην επικοινωνία σημείου-προς-σημείο μία διεργασία αποστέλλει ένα μήνυμα σε μία άλλη συγκεκριμένη διεργασία

Για να στείλει ένα μήνυμα, η διεργασία πηγή εκτελεί μία κλήση MPI στην οποία καθορίζει τη διεργασία προορισμό με το βαθμό της στον κατάλληλο communicator (π.χ. MPI_COMM_WORLD). Η διεργασία προορισμός πρέπει επίσης να εκτελέσει μία κλήση MPI για να πάρει το μήνυμα.

Α.6 Τρόποι επικοινωνίας

Το πρωτόκολλο MPI παρέχει τέσσερις τρόπους επικοινωνίας: τον τυπικό, τον σύγχρονο, τον buffered και την επικοινωνία σε κατάσταση ετοιμότητας. Δεν έχει νόημα να μιλήσουμε για τους τρόπους επικοινωνίας αναφερόμενοι στη λειτουργία receive. Η “ολοκλήρωση” ενός send σημαίνει εξορισμού ότι το buffer αποστολής μπορεί με ασφάλεια να επαναχρησιμοποιηθεί. Οι

τρεις πρώτοι τρόποι επικοινωνίας διαφέρουν μόνο στο πώς η ολοκλήρωση ενός send εξαρτάται από τη λήψη του μηνύματος.

	Completion condition
Synchronous send	Only completes when the receive has completed.
Buffered send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Standard send	Either synchronous or buffered.
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Receive	Completes when a message has arrived.

Πίνακας Α.2: Τρόποι επικοινωνίας του MPI

Και οι τέσσερις τρόποι επικοινωνίας διατίθενται σε εκδόσεις blocking και non-blocking. Στην περίπτωση των blocking υλοποιήσεων, η επιστροφή από τη ρουτίνα συνεπάγεται ολοκλήρωση της αποστολής. Στις non-blocking περιπτώσεις, όλοι οι τρόποι μπορούν να ελεγχθούν για το κατά πόσο ολοκληρώθηκαν με τις συνηθισμένες ρουτίνες (MPI_Test, MPI_Wait, κτλ.)

	Blocking form
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

Πίνακας Α.3: Ρουτίνες επικοινωνίας του MPI

Α.6.1 Τυπικό Send

Το τυπικό send ολοκληρώνεται όταν το μήνυμα έχει αποσταλεί, το οποίο μπορεί να συνεπάγεται ότι το μήνυμα έφθασε στον προορισμό, κάτι τέτοιο όμως δεν είναι εξασφαλισμένο. Το μήνυμα μπορεί αντί αυτού να βρίσκεται “μέσα στο δίκτυο επικοινωνίας” για κάποιο χρονικό διάστημα. Ένα πρόγραμμα που χρησιμοποιεί τυπικά send πρέπει να υπακούει σε αρκετούς κανόνες

- Δεν πρέπει να υποθέτει ότι η αποστολή θα ολοκληρωθεί πριν αρχίσει η λήψη. Για παράδειγμα, δύο διεργασίες δεν πρέπει να χρησιμοποιούν τυπικές blocking send για

την ανταλλαγή μηνυμάτων, αφού αυτό μπορεί σε κάποια περίπτωση να οδηγήσει σε αδιέξοδο.

- Δεν ότι η αποστολή θα ολοκληρωθεί μετά που θα αρχίσει η λήψη. Για παράδειγμα, ο αποστολέας δεν πρέπει να στείλει επιπλέον μηνύματα των οποίων η σωστή ερμηνεία εξαρτάται από την υπόθεση ότι ένα προηγούμενο μήνυμα έφθασε κάπου αλλού · μπορεί κανείς να υποθέσει σενάρια όπου η σειρά των μηνυμάτων είναι μη ντετερμινιστική στο πλαίσιο του τυπικού send.
- Οι διεργασίες πρέπει να είναι ανυπόμονοι αναγνώστες, δηλαδή να εξασφαλίζουν ότι τελικά θα λάβουν όλα τα μηνύματα που αποστέλλονται προς αυτές, διαφορετικά το δίκτυο μπορεί να υπερφορτωθεί.

Εάν ένα πρόγραμμα παραβιάσει αυτούς τους κανόνες, μπορεί να έχει σαν αποτέλεσμα απρόβλεπτες συνέπειες. Προγράμματα μπορεί να εκτελεστούν επιτυχώς τη μία φορά αλλά την άλλη μπορεί να έχουμε λανθασμένη εκτέλεση με μη ντετερμινιστικό τρόπο.

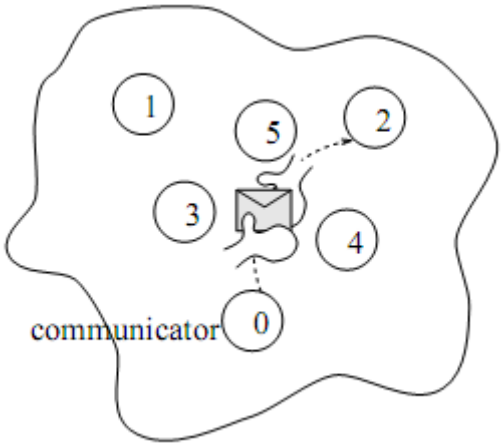
Το τυπικό send έχει την εξής μορφή:

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

όπου buf η διεύθυνση των δεδομένων προς αποστολή, count ο αριθμός των στοιχείων του τύπου δεδομένων MPI που περιέχονται στο buf, datatype είναι ο τύπος δεδομένων του MPI, dest είναι η διεργασία παραλήπτης του μηνύματος, όπως ορίζεται από το βαθμό του μέσα στα πλαίσια του communicator comm, και comm είναι ο κοινός communicator μεταξύ των διεργασιών λήψης και αποστολής.

Α.6.2 Σύγχρονο Send

Εάν η διεργασία αποστολέας χρειάζεται να ξέρει ότι ένα μήνυμα λήφθηκε από τη διεργασία παραλήπτη, τότε και οι δύο διεργασίες πρέπει να χρησιμοποιήσουν σύγχρονη επικοινωνία. Κατά τη σύγχρονη επικοινωνία ο παραλήπτης αποστέλλει πίσω ένα αναγνωριστικό (σε μια διαδικασία γνωστή ως χειραψία) όπως φαίνεται στο Σχήμα



Σχήμα Α.4: Στη σύγχρονη επικοινωνία ο αποστολέας ξέρει εάν ο παραλήπτης έλαβε το μήνυμα

Η ρουτίνα του MPI που αντιστοιχεί το σύγχρονο send έχει παρόμοια μορφή με το τυπικό send, όπως για παράδειγμα της blocking περίπτωσης :

```
MPI_SSend(buf, count, datatype, dest, tag, comm)
```

Εάν μια διεργασία που εκτελεί ένα blocking σύγχρονο send βρίσκεται πιο μπροστά από τη διεργασία που εκτελεί το ταιριαστό receive, τότε θα παραμείνει ανενεργή μέχρι η διεργασία παραλήπτης να την φτάσει, και αντίστροφα. Η σύγχρονη μορφή επικοινωνίας μπορεί συνεπώς να είναι πιο αργή από την τυπική μορφή, όμως αποτελεί μια πιο ασφαλή μορφή επικοινωνίας υπό την έννοια ότι το δίκτυο επικοινωνίας δεν μπορεί ποτέ να υπερφορτωθεί με απαράδοτα μηνύματα. Έχει επίσης το πλεονέκτημα ότι είναι πιο προβλέψιμη γιατί πάντα συγχρονίζει τον αποστολέα με τον παραλήπτη. Αυτό κάνει τη συμπεριφορά του προγράμματος πιο ντετερμινιστική.

Α.6.3 Buffered Send

Το buffered send εγγυάται ότι θα επιστρέψει αμέσως, αντιγράφοντας το μήνυμα σε ένα buffer για αποστολή αργότερα εάν χρειαστεί. Το πλεονέκτημα σε σχέση με το τυπικό send είναι η προβλεψιμότητα – ο αποστολέας και ο παραλήπτης εγγυώνται ότι δε θα είναι συγχρονισμένοι και ότι εάν το δίκτυο υπερφορτωθεί, ένα σφάλμα θα προκύψει. Το μειονέκτημα αυτού του τύπου send είναι ότι ο προγραμματιστής δεν μπορεί να χρησιμοποιήσει κανένα προ-ανατιθέμενο buffered χώρο και πρέπει ρητά να επισυνάψει αρκετό χώρο μνήμης για το πρόγραμμα με κλήση της ρουτίνας MPI_Buffer_Attach. Το non-blocking buffered send δεν παρουσιάζει κανένα πλεονέκτημα σε σχέση με την blocking περίπτωση.

Α.6.3 Το Τυπικό Blocking Receive

Η μορφή του τυπικού blocking receive είναι η εξής:

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

όπου buf η διεύθυνση όπου θα αποθηκευτούν τα δεδομένα όταν φθάσουν και πρέπει να είναι αρκετά μεγάλο για να χωράει όλο το μήνυμα, count ο αριθμός των στοιχείων του τύπου δεδομένων MPI που περιέχονται στο buf και πρέπει να ταιριάζει με τον αριθμό των στοιχείων που δόθηκε κατά την αποστολή, datatype είναι ο τύπος δεδομένων του MPI και πρέπει να ταιριάζει με τον τύπο που δόθηκε κατά την αποστολή, source είναι η διεργασία αποστολής του μηνύματος, όπως ορίζεται από το βαθμό του μέσα στα πλαίσια του communicator comm. ή οποιαδήποτε διεργασία εάν τοποθετηθεί MPI_ANY_SOURCE, το tag χρησιμοποιείται για να δηλώσει ότι πρέπει να ληφθεί ένα μήνυμα με συγκεκριμένη ετικέτα (ή MPI_ANY_TAG) και comm είναι ο κοινός communicator μεταξύ των διεργασιών λήψης και αποστολής.

Α.7 Επικοινωνία Non-Blocking

Στην επικοινωνία non-blocking οι διεργασίες καλούν μία ρουτίνα MPI για την έναρξη της επικοινωνίας (αποστολή ή λήψη), αλλά η ρουτίνα επιστρέφει πριν την ολοκλήρωση της επικοινωνίας. Η επικοινωνία μπορεί να συνεχιστεί στο παρασκήνιο και η διεργασία μπορεί να συνεχίσει να εκτελεί το έργο της, επιστρέφοντας σε μία μελλοντική στιγμή για να ελέγξει ότι η επικοινωνία ολοκληρώθηκε επιτυχώς. Η επικοινωνία μοιράζεται συνεπώς σε δύο λειτουργίες: την έναρξη και τον έλεγχο ολοκλήρωσης. Η επικοινωνία non-blocking είναι ανάλογη με μία μορφή ανάθεσης – ο χρήστης κάνει ένα αίτημα για στο MPI για επικοινωνία και ελέγχει ότι η αίτησή του ολοκληρώθηκε επιτυχώς μόνο όταν χρειάζεται να το γνωρίζει για να συνεχίσει με την εργασία του.

Α.7.1 Έναρξη Non-Blocking Επικοινωνίας στο MPI

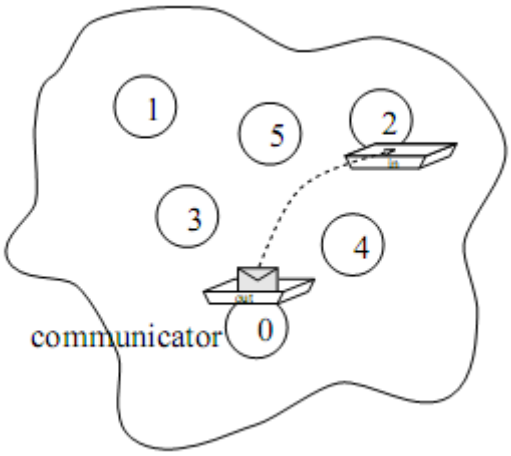
Οι non-blocking ρουτίνες παίρνουν ανάλογες παραμέτρους με τις αντίστοιχες blocking. Οι ρουτίνες non-blocking αποστολής παίρνουν μία επιπλέον παράμετρο. Αυτή η παράμετρος, που ονομάζεται status, είναι πολύ σημαντική καθώς παρέχει ένα χειριστή που χρησιμοποιείται για τον έλεγχο του κατά πόσον η επικοινωνία έχει ολοκληρωθεί. Η non-blocking ρουτίνα λήψης δεν παίρνει την παράμετρο status, αλλά έχει αντί αυτού ένα χειριστή σε ένα αίτημα επικοινωνίας (request).

	Blocking form
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

Πίνακας Α.4: Μοντέλα επικοινωνίας για non-blocking επικοινωνίες

Α.7.2 Non-Blocking Sends

Η αρχή πίσω από το non-blocking send παρουσιάζεται στο Σχήμα Α.5:



Σχήμα Α.5: Μια αποστολή non-blocking

Η διεργασία αποστολής ξεκινά την αποστολή χρησιμοποιώντας την ακόλουθη ρουτίνα (σε σύγχρονη επικοινωνία):

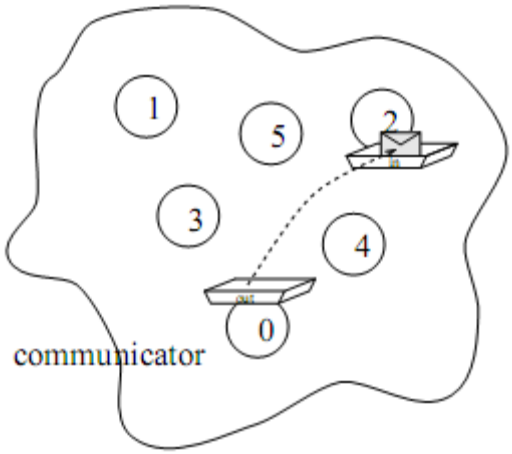
```
MPI_ISSend(buf, count, datatype, dest, tag, comm, request)
```

Στη συνέχεια συνεχίζει με άλλους υπολογισμούς που δεν αλλοιώνουν το buffer αποστολής. Πριν η διεργασία αποστολή μπορέσει να ενημερώσει το buffer αποστολής πρέπει να ελέγξει ότι η αποστολή έχει ολοκληρωθεί χρησιμοποιώντας τις κατάλληλες ρουτίνες..

Α.7.3 Non-Blocking Receives

Μπορούν λήψεις non-blocking να ταιριάσουν με blocking αποστολές και αντίστροφα.

Μία non-blocking receive παρουσιάζεται στο Σχήμα



Σχήμα Α.6: Μια λήψη non-blocking

Η διεργασίας παραλήπτης εκτελεί την ακόλουθη ρουτίνα για την έναρξη της λήψης:

```
MPI_IRecv(buf, count, datatype, source, tag, comm, request)
```

Η διεργασία παραλήπτης μπορεί στη συνέχεια να εκτελεί άλλες εντολές μέχρι να χρειαστεί τα δεδομένα που λήφθηκαν. Ελέγχει στη συνέχεια το buffer λήψης για να ελέγξει εάν η επικοινωνία έχει ολοκληρωθεί.

Α.7.4 Έλεγχος για Ολοκλήρωση της Επικοινωνίας

Όταν χρησιμοποιούμε επικοινωνία non-blocking είναι απαραίτητο να εξασφαλίσουμε ότι η επικοινωνία έχει ολοκληρωθεί πριν χρησιμοποιήσουμε τα αποτελέσματα της επικοινωνίας ή το buffer επικοινωνίας. Οι τρόποι ελέγχου για ολοκλήρωση της επικοινωνίας είναι δύο:

- Ο τύπος αναμονής WAIT. Αυτές οι ρουτίνες μπλοκάρουν μέχρι η επικοινωνία να ολοκληρωθεί. Είναι χρήσιμες όταν τα δεδομένα από την επικοινωνία χρειάζονται για υπολογισμούς ή εάν πρόκειται να επαναχρησιμοποιηθεί το buffer επικοινωνίας. Συνεπώς μία επικοινωνία non-blocking που ακολουθείται αμέσως από ένα έλεγχο τύπου WAIT ισοδυναμεί με την αντίστοιχη περίπτωση blocking.
- Ο τύπο ελέγχου TEST. Αυτές οι ρουτίνες επιστρέφουν την τιμή ΑΛΗΘΕΣ ή ΨΕΥΔΕΣ ανάλογα με το εάν η επικοινωνία έχει ολοκληρωθεί. Δεν μπλοκάρουν και είναι χρήσιμες σε περιπτώσεις όπου θέλουμε να ξέρουμε εάν η επικοινωνία έχει ολοκληρωθεί αλλά δεν χρειαζόμαστε ακόμα το αποτέλεσμα ή να επαναχρησιμοποιήσουμε το buffer επικοινωνίας, δηλαδή η διεργασία έχει τη δυνατότητα να εκτελέσει χρήσιμους υπολογισμούς στο ενδιάμεσο.

Α.7.4.1 Έλεγχος για Ολοκλήρωση της Επικοινωνίας

Το τεστ για έλεγχο τύπου WAIT είναι:

```
MPI_Wait(request, status)
```

Αυτή η ρουτίνα μπλοκάρει μέχρι η επικοινωνία που ορίζεται από το χειριστή request να ολοκληρωθεί. Ο χειριστής request πρέπει να έχει επιστραφεί προηγουμένων από μια κλήση non-blocking ρουτίνας επικοινωνίας. Το τεστ για έλεγχο τύπου TEST είναι:

```
MPI_Test(request, flag, status)
```

Σε αυτή την περίπτωση η επικοινωνία που ορίζεται από το χειριστή request απλά ελέγχεται για το κατά πόσο έχει ολοκληρωθεί και το αποτέλεσμα του ελέγχου αυτού επιστρέφεται αμέσως στην παράμετρο flag.

Να σημειώσουμε ότι υπάρχουν αντίστοιχες ρουτίνες για έλεγχο πολλαπλών επικοινωνιών.

Παράρτημα Β

Πηγαίος Κώδικας

Παρουσιάζουμε στο παράρτημα αυτό τον πηγαίο κώδικα για μερικές από τις υλοποιήσεις μας. Κρίνουμε απαραίτητο να παρουσιάσουμε μόνο τον πηγαίο κώδικα που αφορά τις δύο χωρικές διαστάσεις γιατί είναι αυτός στον οποίο αναφερόμασταν στα παραδείγματα σε διάφορα κεφάλαια της εργασίας και γιατί είναι πιο ευανάγνωστος από την τρισδιάσταση περίπτωση, χωρίς όμως να είναι τετριμμένος όπως η υλοποίηση σε μία χωρική διάσταση.

Β.1 Βασικό Παράλληλο Πρόγραμμα Προσομοίωσης Εξίσωσης Διάχυσης-2D

```
1   /*
2    * Αρχείο advent2d.c
3    * Υλοποίηση βασικού παράλληλου προγράμματος προσομοίωσης
4    * της εξίσωσης διάχυσης δύο χωρικών διαστάσεων
5    *
6    * Νικόλας Ιωάννου 2008
7    */
8   #include <math.h>
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <string.h>
12  #include <sys/time.h>
13  #include "mpi.h"
14  #include "lib.h"
15
16  #define X_LEFT   0.234
17  #define Y_BACK   0.453
18  #define T 1000
19  #define WEST    0
20  #define EAST    1
21  #define NORTH   2
22  #define SOUTH   3
23
24  #ifdef DEBUG
25  FILE *fp;
26  void CollectAll(int);
27  #endif
28
29  int nproc, my_rank, MAXLOOPS = 2500,
30      X = 10, Y = 10, t,
31      width, length,
32      surfaceSizeX, surfaceSizeY,
33      startX, startY,
34      p1 = 1, p2 = 1, P = 1;
35
```

```
36  double ***U, **sendData, **recvData;
37  char  comm[4];
38
39  void Init2D();
40  void Send();
41  void Receive();
42  inline void SetDataToSend(int);
43  inline void UnpackData(int);
44
45
46
47  int main(int argc, char **argv)
48  {
49     int x, y, loops = 0;
50     double a = 1.001 , dt = 0.00001, dx = 0.00001;
51     struct timeval  start, finish;
52
53     MPI_Init(&argc, &argv);
54     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
55     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
56
57     //Diastaseis xwriou
58     if(argc == 6){
59        X  = atoi(argv[1]);
60        Y  = atoi(argv[2]);;
61        p1 = atoi(argv[3]);
62        p2 = atoi(argv[4]);
63        MAXLOOPS = atoi(argv[5]);
64        P  = p1*p2;
65        if (P > 32 || P <= 0)
             error("Invalid number of processors given, max 16!", my_rank);
66        if (P!=nproc)
             error("Processors allocated != P (P = %d, size = %d)!", P, nproc);
67        if (X<p1 || Y<p2)
             error("Wrong dimensions given, supposed X,Y,Z>=p1,p2,p3");
68     }
69     else if(argc != 1)
          error("Invalid number of arguments, 0 or 5 required!");
70
71     Init2D();
72
73     MPI_Barrier(MPI_COMM_WORLD);
74
75     if(my_rank==0)gettimeofday(&start, (struct timezone *)NULL);
76
77     /*Επαναληπτική διαδικασία που εκτελεί τον υπολογισμό της εξίωσης διάχυσης-2D
78       και τερματίζει όταν τελειώσει ο προκαθορισμένος αριθμός επαναλήψεων.*/
79     while (loops < MAXLOOPS)
80     {
81        t = (++loops)%2;
82
83        //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
84        Receive();
85        UnpackData(t);
86        for(y = startY; y < length+1; y++) {
87           for(x = startX; x < width+1; x++){
```

```
88
                U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] - a*dt/dx*(U[t][y-1][x] +
                             U[t][y][x-1]);
89          SetDataToSend(t);
90          Send();
91       }
92       //Για να ξέρουμε ότι όλες οι διεργασίες τελείωσαν τους υπολογισμούς
93       MPI_Barrier(MPI_COMM_WORLD);
94
95       if(my_rank==0){
96          gettimeofday(&finish, (struct timezone *)NULL);
97          subtimeval(&finish, &start);
98       }
99
100      if (my_rank == 0){
101         printf("\nmpi2d X=%d Y=%d T=%d %dx%d Time taken: %lds and %ldus.\n", X, Y,
                  MAXLOOPS, p1, p2, finish.tv_sec, finish.tv_usec);
102      }
103   #  if DEBUG > 0
104      CollectAll(t);
105   #  endif
106      //FreeAll();
107      MPI_Finalize();
108      return 0;
109   }
110
111   /*Εκτελούνται οι κατάλληλες αρχικοποιήσεις*/
112   void Init2D()
113   {
114      int t, i, j, k, sum,
115          partitionX[p1], partitionY[p2];
116
117      sum = X;
118      for(i = 0; i < p1; i++){
119         partitionX[i] = sum/(p1 - i);
120         sum -= sum/(p1 - i);
121      }
122      width = partitionX[my_rank%p1];
123
124      sum = Y;
125      for(i = 0; i < p2; i++){
126         partitionY[i] = sum/(p2 - i);
127         sum -= sum/(p2 - i);
128      }
129      length = partitionY[(my_rank%(p1*p2))/p1];
130
131      surfaceSizeX = length;
132      surfaceSizeY = width;
133
134      sendData = (double **) malloc(sizeof(double *)*4);
135      recvData = (double **) malloc(sizeof(double *)*4);
136      if(sendData == NULL || recvData == NULL){
137         fprintf(stderr, "\nMemory allocation problem!");
138         exit(1);
139      }
140      //Συνεχόμενη δέσμευση μνήμης
141      U = (double ***) malloc(2*sizeof(double **));
142      if(U == NULL) error("Memory allocation problem1\n!");
```

```
143
144     U[0] = (double **) malloc(2*(length+1)*sizeof(double*));
145     if(U[0] == NULL) error("Memory allocation problem2\n!");
146     U[1] = U[0] + length+1;
147
148     U[0][0] = (double *) malloc(2*(length+1)*(width+1)*sizeof(double));
149     if(U[0][0] == NULL) error("Memory allocation problem3\n!");
150     for(t = 0; t < 2; t++)
151        for(i = 0; i < length+1; i++)
152            U[t][i] = U[0][0] + t*(length+1)*(width+1)+i*(width+1);
153
154     //Αρχικοποίηση πίνακα
155     for(t = 0; t < 2; t++)
156           for(j = 0; j < length+1; j++)
157              for(k = 0; k < width+1; k++) {U[t][j][k] = 0;}
158
159     for(i = 0; i < 4; i++) comm[i] = 0;
160
161     //Αν λαμβάνω από αριστερά, αλλιώς κάνω κατάλληλη αρχικοποίηση
162     if(((my_rank-1)%p1 >= 0) && (my_rank%p1 != 0)){
163        comm[WEST] = 1;
164        recvData[WEST] = (double *) malloc(sizeof(double)*surfaceSizeX);
165     }
166     else for(j = 0; j < length+1; j++)
167                {U[0][j][0] = U[0][j][1] = U[1][j][0] = U[1][j][1] = X_LEFT;}
168
169     //Αν στέλλω στα δεξιά
170     if((my_rank+1)%p1 > 0){
171        comm[EAST] = 1;
172        sendData[EAST] = (double *) malloc(sizeof(double)*surfaceSizeX);
173     }
174
175     //Αν στέλλω προς τα μπρος
176     if(my_rank + p1 < P){
177        comm[NORTH] = 1;
178        sendData[NORTH] = (double *) malloc(sizeof(double)*surfaceSizeY);
179     }
180
181     //Αν λαμβάνω από πίσω, αλλιώς κάνω κατάλληλη αρχικοποίηση
182     if((my_rank-p1) >= 0){
183        comm[SOUTH] = 1;
184        recvData[SOUTH] = (double *) malloc(sizeof(double)*surfaceSizeY);
185     }
186     else for(k = 0; k < width+1; k++)
187                {U[0][0][k] = U[0][1][k] = U[1][0][k] = U[1][1][k] = Y_BACK;}
188
189     /* Θέτουμε τις αρχές και τέλη του πίνακα με βάση τις
190        συνοριακές συνθήκες */
191     if(comm[WEST]   == 1) startX = 1;      else startX = 2;
192     if(comm[SOUTH]  == 1) startY = 1;      else startY = 2;
193
194  #  ifdef DEBUG
195     printf("MyRank=%d, width,length=%d,%d p1,p2,P=%d,%d,%d
196  #  endif
197  }
198
199  /* Συνάρτηση που εκτελεί τη λειτουργία αποστολής */
200  void Send()
```

```
201
    {
202     int i, tag, partner;
203     MPI_Request requests[2];              /* communication identifier      */
204     MPI_Status status[2];             /* status of completed communication */
205
206     /* initialize requests */
207     for (i = 0; i < 2; i++) requests[i] = MPI_REQUEST_NULL;
208
209
210     /* send messages to my partners */
211     if(comm[EAST] == 1){
212        partner = my_rank+1; tag = WEST;
213        MPI_Isend(sendData[EAST], surfaceSizeX, MPI_DOUBLE, partner, tag,
214     }
215     if(comm[NORTH] == 1){
216        partner = my_rank+p1; tag = SOUTH;
217        MPI_Isend(sendData[NORTH], surfaceSizeY, MPI_DOUBLE, partner, tag,
218     }
219     //Αναμονή για ολοκλήρωση της αποστολής
220     MPI_Waitall(2, requests, status);
221 }
222
223 /* Συνάρτηση που εκτελεί τη λειτουργία λήψης */
224 void Receive()
225 {
226     int i, tag, partner;
227     MPI_Request requests[2];              /* communication identifier      */
228     MPI_Status status[2];             /* status of completed communication */
229     /* initialize requests */
230     for (i = 0; i < 2; i++) requests[i] = MPI_REQUEST_NULL;
231
232     /* receive incoming messages */
233     if(comm[WEST] == 1){
234        partner = my_rank-1; tag = WEST;
235        MPI_Irecv(recvData[WEST], surfaceSizeX, MPI_DOUBLE, partner, tag,
236     }
237     if(comm[SOUTH] == 1){
238        partner = my_rank-p1; tag = SOUTH;
239        MPI_Irecv(recvData[SOUTH], surfaceSizeY, MPI_DOUBLE, partner, tag,
240     }
241     //Αναμονή για ολοκλήρωση της λήψης
242     MPI_Waitall(2, requests, status);
243 }
244
245 /*Ετοιμάζουμε τα στοιχεία που θα αποσταλούν*/
246 inline void SetDataToSend(int t)
247 {
248     int y,x;
249     if(comm[EAST] == 1)
250        for(y = 1; y < length+1; y++) sendData[EAST][y-1] = U[t][y][width];
251     if(comm[NORTH] == 1)
252        for(x = 1; x < width+1; x++)  sendData[NORTH][x-1] = U[t][length][x];
253 }
254
255
256 /*Ξεπακετάρουμε τα στοιχεία που πήραμε*/
```

```
257
    inline void UnpackData(int t)
258 {
259    int y, x;
260    if(comm[WEST] == 1){
261       for(y = 1; y < length+1; y++) U[t][y][0] = recvData[WEST][y-1];
262    }
263    if(comm[SOUTH] == 1){
264       for(x = 1; x < width+1; x++) U[t][0][x] = recvData[SOUTH][x-1];
265    }
266 }
267
268 /*  Για σκοπούς debbuging, τυπώνουμε τα αποτελέσματα σε αρχείο*/
269 #ifdef DEBUG
270 void CollectAll(int t)
271 {
272    int y,x,px,py,i,m, offset[P], sum, partitionX[p1], partitionY[p2];
273    double *Ularge, *buffer;
274
275    sum = X;
276    for(i = 0; i < p1; i++){partitionX[i] = sum/(p1 - i);sum -= sum/(p1 - i);}
277
278    sum = Y;
279    for(i = 0; i < p2; i++){partitionY[i] = sum/(p2 - i);sum -= sum/(p2 - i);}
280
281    buffer = (double *) malloc(sizeof(double)*partitionX[p1-1]*partitionY[p2-1]);
282
283    if(my_rank==0) {
284       Ularge = (double *) malloc(sizeof(double)*P*partitionX[p1-1]*
285       for(y = 0; y < Y; y++)
286          for(x = 0; x < X; x++) Ularge[y*width+x] = 10;
287    }
288
289    MPI_Barrier(MPI_COMM_WORLD);
290
291    for(y = 1; y < length+1; y++)
292       for(x = 1; x < width+1; x++) buffer[(y-1)*width+x-1] = U[t][y][x];
293
294    MPI_Gather(buffer, partitionX[p1-1]*partitionY[p2-1],
295               MPI_DOUBLE, Ularge, partitionX[p1-1]*partitionY[p2-1], MPI_DOUBLE,
296
297    if(my_rank==0){
298       for(m = 0; m < P; m++) offset[m] = m*partitionX[p1-1]*partitionY[p2-1];
299
300       for(py = 0; py < p2; py++){
301          for(y = 0; y < partitionY[py]; y++){
302             for(px = 0; px < p1; px++){
303                for(x = 0; x < partitionX[px]; x++){
304                   m = py*p1+px;
305                   fprintf(fp, "%12f ", Ularge[offset[m]+ y*partitionX[px]+x]);
306                }
307             }
308             fprintf(fp, "\n");
309          }
310          }
311       fclose(fp);
312       for(m=0; m < P; m++){
```

```
313
                for(y=0; y<partitionY[p2-1]; y++){
314                 for(x=0; x<partitionX[p1-1]; x++)
315                     printf("%f ", Ularge[m*partitionX[p1-1]*partitionY[p2-1]+
316                 printf("\n");
317               }
318               printf("\n");
319           }
320       }
321       free(buffer); if(my_rank == 0) free(Ularge);
322   }
323   #endif
```

Β.2 Παράλληλο Πρόγραμμα Προσομοίωσης Εξίσωσης Διάχυσης-2D με Tiling[5]

```
1  /*
2   * Αρχείο advent2d_tiling.c
3   * Υλοποίηση παράλληλου προγράμματος προσομοίωσης
4   * της εξίσωσης διάχυσης δύο χωρικών διαστάσεων με
5   * δυνατότητες tiling
6   *
7   * Νικόλας Ιωάννου 2008
8   */
9
10 /*...*/
11
12 int main(int argc, char **argv)
13 {
14
15    /*...*/
16
17    Init2D();
18
19    MPI_Barrier(MPI_COMM_WORLD);
20
21    if(my_rank==0)gettimeofday(&start, (struct timezone *)NULL);
22
23    /*Επαναληπτικός βρόχος όπου γίνεται σε κάθε βήμα επικοινωνία
24      και υπολογισμός μέχρι ένα αριθμό επαναλήψεων.*/
25    while (loops < MAXLOOPS)
26    {
27       //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
28       Receive();
29       //loops+K+2 % K+1 = loops+1 % K+1
30       UnpackData((loops+1)%(K+1));
31       for(k = 0; k < K; k++){
32          t = (++loops)%(K+1);
33          for(y = startY; y < length+1; y++)
34             for(x = startX; x < width+1; x++)
                  U[t][y][x] = (1+2*a*dt/dx)*U[(t+K)%(K+1)][y][x] - a*dt/dx*
35                             (U[t][y-1][x] + U[t][y][x-1]);
36       }
37       //Βάζουμε τα δεδομένα που υπολογίσαμε στο buffer που θα σταλεί
38       SetDataToSend((t+2)%(K+1));
39       Send();
40    }
41    //Για να ξέρουμε ότι όλες οι διεργασίες τελείωσαν τους υπολογισμούς
42    MPI_Barrier(MPI_COMM_WORLD);
43
44    if(my_rank==0){
45       gettimeofday(&finish, (struct timezone *)NULL);
         if(finish.tv_usec >= start.tv_usec){finish.tv_usec -= start.tv_usec;
46                                             finish.tv_sec -= start.tv_sec;}
         else{finish.tv_usec += 1000000 - start.tv_usec;
47            finish.tv_sec  -= start.tv_sec + 1;}
48       }
```

[5] Παρουσιάζουμε μόνο τα σημεία που διαφέρουν σημαντικά από τη βασική υλοποίηση

```
49
50     //PrintFinalArray();
51
52     if (my_rank == 0){
          printf("\nmpi2dtil X=%d Y=%d T=%d K=%d %dx%d Time taken: %lds and %ldus.\n"
53               ,X, Y, MAXLOOPS, K, p1, p2, finish.tv_sec, finish.tv_usec);
54     }
55
56  #  if DEBUG > 0
57     CollectAll(t);
58  #  endif
59
60     //FreeAll();
61     MPI_Finalize();
62     return 0;
63  }
64
65  /*Εκτελούνται οι κατάλληλες αρχικοποιήσεις*/
66  void Init2D()
67  {
68     /*...*/
69
70     surfaceSizeX = length*K;
71     surfaceSizeY = width*K;
72
73     sendData = (double **) malloc(sizeof(double *)*4);
74     recvData = (double **) malloc(sizeof(double *)*4);
75
       if(sendData == NULL || recvData == NULL){fprintf(stderr, "\nMemory allocation
76                                                              problem!");exit(1);}
77     //Συνεχόμενη δέσμευση μνήμης
78     U = (double ***) malloc((K+1)*sizeof(double **));
79     if(U == NULL) error("Memory allocation problem1\n!");
80
81     U[0] = (double **) malloc((K+1)*(length+1)*sizeof(double*));
82     if(U[0] == NULL) error("Memory allocation problem2\n!");
83     for(t = 1; t < (K+1); t++) U[t] = U[0] + t*(length+1);
84
85     U[0][0] = (double *) malloc((K+1)*(length+1)*(width+1)*sizeof(double));
86     if(U[0][0] == NULL) error("Memory allocation problem3\n!");
87     for(t = 0; t < (K+1); t++)
88        for(i = 0; i < length+1; i++)
89            U[t][i] = U[0][0] + t*(length+1)*(width+1)+i*(width+1);
90
91
92     //Αρχικοποίηση πίνακα
93     for(t = 0; t < (K+1); t++)
94        for(j = 0; j < length+1; j++)
95           for(k = 0; k < width+1; k++) {U[t][j][k] = 0;}
96
97     for(i = 0; i < 4; i++) comm[i] = 0;
98
99     //Αν λαμβάνω από αριστερά, αλλιώς κάνω κατάλληλη αρχικοποίηση
100    if(((my_rank-1)%p1 >= 0) && (my_rank%p1 != 0)){
101       comm[WEST] = 1;
102       recvData[WEST] = (double *) malloc(sizeof(double)*surfaceSizeX);
103    }
104    else for(t = 0; t < (K+1); t++)
```

```
105             for(j = 0; j < length+1; j++)
106                {U[t][j][0] = U[t][j][1]  = X_LEFT;}
107     //Αν στέλνω στα δεξιά
108     if((my_rank+1)%p1 > 0){
109        comm[EAST] = 1;
110        sendData[EAST] = (double *) malloc(sizeof(double)*surfaceSizeX);
111     }
112     //Αν στέλνω προς τα μπρος
113     if(my_rank + p1 < P){
114        comm[NORTH] = 1;
115        sendData[NORTH] = (double *) malloc(sizeof(double)*surfaceSizeY);
116     }
117     //Αν λαμβάνω από πίσω, αλλιώς κάνω κατάλληλη αρχικοποίηση
118     if((my_rank-p1) >= 0){
119        comm[SOUTH] = 1;
120        recvData[SOUTH] = (double *) malloc(sizeof(double)*surfaceSizeY);
121     }
122     else for(t = 0; t < (K+1); t++)
123             for(k = 0; k < width+1; k++)
124                {U[t][0][k] = U[t][1][k] = Y_BACK;}
125
126     /* Θέτουμε τις αρχές και τέλη του πίνακα με βάση τις
127        συνοριακές συνθήκες */
128     if(comm[WEST]  == 1) startX = 1; else startX = 2;
129     if(comm[SOUTH] == 1) startY = 1; else startY = 2;
130
131  #  ifdef DEBUG
       printf("MyRank=%d, width,length=%d,%d p1,p2,P=%d,%d,%d
132  startX,Y=%d,%d\n",my_rank,width,length,p1,p2,P,startX,startY);
133  #  endif
134  }
135
136  /*...*/
137
138  /*Ετοιμάζουμε τα στοιχεία που θα αποσταλούν*/
139  inline void SetDataToSend(int t)
140  {
141     int y,x,k;
142     if(comm[EAST] == 1)
143        for(k = 0; k < K; k++)
              for(y = 1; y < length+1; y++) sendData[EAST][k*length + y-1] =
144  U[(t+k)%(K+1)][y][width];
145     if(comm[NORTH] == 1)
146        for(k = 0; k < K; k++)
              for(x = 1; x < width+1; x++)  sendData[NORTH][k*width  + x-1] =
147  U[(t+k)%(K+1)][length][x];
148  }
149
150
151  /*Ξεπακετάρουμε τα στοιχεία που πήραμε*/
152  inline void UnpackData(int t)
153  {
154     int y, x, k;
155     //printf("rank = %d, k=%d\n", my_rank,k);
156     if(comm[WEST] == 1){
157        for(k = 0; k < K; k++)
              for(y = 1; y < length+1; y++) U[(t+k)%(K+1)][y][0] =
158  recvData[WEST][k*length + y-1];
159     }
```

```
160     if(comm[SOUTH] == 1){
161
              for(x = 1; x < width+1; x++) U[(t+k)%(K+1)][0][x] =
162  recvData[SOUTH][k*width + x-1];
163     }
164  }
165  //inline functions unpack, setdata
166
167  /*...*/
```

Β.3 Παράλληλο Πρόγραμμα Προσομοίωσης Εξίσωσης Διάχυσης-2D με Επικαλυπτόμενη Δρομολόγηση[6]

```
1  /*
2   * Αρχείο advent2d_overlap.c
3   * Υλοποίηση παράλληλου προγράμματος προσομοίωσης
4   * της εξίσωσης διάχυσης δύο χωρικών διαστάσεων με
5   * επικαλυπτόμενη δρομολόγηση
6   *
7   * Νικόλας Ιωάννου 2008
8   */
9
10 /*...*/
11
12 int main(int argc, char **argv)
13 {
14
15    /*...*/
16
17    MPI_Barrier(MPI_COMM_WORLD);
18
19    if(my_rank==0)gettimeofday(&start, (struct timezone *)NULL);
20
21    /*Πρώτη επικοινωνία ώστε να είναι έτοιμα τα πρώτα δεδομένα για να
22      μπορεί να ξεκινήσει η overlapping επικοινωνία*/
23    Receive(&requests[0]);
24    MPI_Waitall(2, &requests[0], &status[0]);
25    t = (++loops)%2;
26    UnpackData(t);
27    //Πρώτος υπολογισμός
28    for(y = startY; y < length+1; y++)
29       for(x = startX; x < width+1; x++)
             U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
30                        a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
31
32    SetDataToSend(t);
33    /*Δεύτερο receive για να μπορεί να γίνει ο υπολογισμός της
34      χρονικής στιγμής 2*/
35    Receive(&requests[0]);
36    MPI_Waitall(2, &requests[0], &status[0]);
37    UnpackData((t+1)%2);
38
39    /* Κυρίως βρόχος όπου, αν θεωρήσουμε το k-οστό βήμα, γίνεται λήψη
40       του k+1 συνόρου, αποστολή του k-1 συνόρου και υπολογισμός του
41       k-οστού βήματος "ταυτόχρονα"*/
42    while (loops < MAXLOOPS - 1)
43    {
44       t = (++loops)%2;
45
46       //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
47       Receive(&requests[0]);
48       Send(&requests[2]);
```

[6] Παρουσιάζουμε μόνο τα σημεία που διαφέρουν σημαντικά από τη βασική υλοποίηση, και μόνο την έκδοση χωρίς tiling, το tiling εφαρμόζεται όπως στο Β.2

```
49          for(y = startY; y < length+1; y++) {
50             for(x = startX; x < width+1; x++){
                  U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
51                               a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
52
53             }
54
55          }
56          MPI_Waitall(4, requests, status);
57          SetDataToSend(t);
58          UnpackData((t+1)%2);
59       }
60
61       /* Αντίστοιχα με την αρχική πρώτη έναρξη επικοινωνίας χρειάζεται
62          κλείσιμο της επικοινωνίας με αποστολή του (K-1)-οστού,
63          υπολογισμό του K-οστού και αποστολή του K-οστού.*/
64       Send(&requests[2]);
65       MPI_Waitall(2, &requests[2], &status[2]);
66
67       t = (++loops)%2;
68       //K-οστός υπολογισμός
69       for(y = startY; y < length+1; y++)
70          for(x = startX; x < width+1; x++)
               U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
71                            a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
72       //K-οστή αποστολή δεδομένων
73       SetDataToSend(t);
74       Send(&requests[2]);
75       MPI_Waitall(2, &requests[2], &status[2]);
76
77       //Για να ξέρουμε ότι όλες οι διεργασίες τελείωσαν τους υπολογισμούς
78       MPI_Barrier(MPI_COMM_WORLD);
79
80       /*...*/
81    }
82
83    /*...*/
```

Β.4 Παράλληλο Πρόγραμμα Προσομοίωσης Εξίσωσης Διάχυσης-2D με Επικαλυπτόμενη Δρομολόγηση σε Δύο Συμμετρικά Νήματα[7]

```
 1  /*
 2   * Αρχείο advent2d_overlap.c
 3   * Υλοποίηση παράλληλου προγράμματος προσομοίωσης
 4   * της εξίσωσης διάχυσης δύο χωρικών διαστάσεων με
 5   * τη ροή να σπάει σε δύο συμμετρικά νήματα
 6   *
 7   * Νικόλας Ιωάννου 2008
 8   */
 9  #define _GNU_SOURCE
10  #include <sched.h>
11  #include <stdio.h>
12  #include <stdlib.h>
13  #include <sys/time.h>
14  #include <pthread.h>
15  #include <sys/types.h>
16  #include <linux/unistd.h>
17  #include <errno.h>
18  #include "mpi.h"
19  #include "lib.h"
20  #include "semaphore.h"
21
22  /*...*/
23  /*Κώδικας Υλοποίησης συνάρτησης gettid(), προσφορά του
24    Νίκου Αναστόπουλου*/
25  #define __syscall_clobber "r11","rcx","memory"
26  #define __syscall "syscall"
27
28  #define __syscall_return(type, res) \
29  do { \
30        if ((unsigned long)(res) >= (unsigned long)(-127)) { \
31                errno = -(res); \
32                res = -1; \
33        } \
34        return (type) (res); \
35  } while (0)
36
37
38
39  #define _syscall0(type,name) \
40        type name(void) \
41  { \
42        long __res; \
43        __asm__ volatile (__syscall \
44                                : "=a" (__res) \
45                                : "0" (__NR_##name) : __syscall_clobber ); \
46        __syscall_return(type,__res); \
47  }
48
49  static _syscall0(pid_t, gettid)
50
```

[7] Παρουσιάζουμε μόνο τα σημεία που διαφέρουν σημαντικά από τη βασική υλοποίηση, και μόνο την έκδοση χωρίς tiling, το tiling εφαρμόζεται όπως στο Β.2

```
51  void* tf(void *args)
52  {
53        fprintf(stderr, "My id: %d\n", (int)gettid());
54
55        pthread_exit(NULL);
56  }
57
58  void * EvenThread(void *);
59
60  /*...*/
61  /*Σηματοφορείς*/
62  sema_t * s1, * s2;
63
64  int main(int argc, char **argv)
65  {
66     /*...*/
67     pthread_t thread_tids[1];
68     cpu_set_t process0_mask;
69     unsigned long cur_mask;
70     /**/
71
72     MPI_Barrier(MPI_COMM_WORLD);
73
74     if(my_rank==0)gettimeofday(&start, (struct timezone *)NULL);
75
76     /*********************************************************************/
77     /*****************ΕΠΙΚΟΙΝΩΝΙΑ ΚΑΙ ΥΠΟΛΟΓΙΣΜΟΙ**********************/
78     /*********************************************************************/
79
80     /*Πρώτη επικοινωνία ώστε να είναι έτοιμα τα πρώτα δεδομένα για να
81       μπορεί να ξεκινήσει η overlapping επικοινωνία*/
82     Receive(&requests[0]);
83     MPI_Waitall(2, &requests[0], &status[0]);
84     t = (++loops)%2;
85     UnpackData(t);
86     //Πρώτος υπολογισμός
87     for(y = startY; y < length+1; y++)
88        for(x = startX; x < width+1; x++)
           U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] - a*dt/dx*(U[t][y-1][x] +
89  U[t][y][x-1]);
90
91     SetDataToSend(t);
92     /*Δεύτερο receive για να μπορεί να γίνει ο υπολογισμός της
93       χρονικής στιγμής 2*/
94     Receive(&requests[0]);
95     MPI_Waitall(2, &requests[0], &status[0]);
96     UnpackData((t+1)%2);
97
98     /*********************************************************************/
99     /*****************************ΝΗΜΑΤΑ*******************************/
100    /*********************************************************************/
101
102    /* Η ροή σπάει σε δύο συμμετρικά νήματα που αναλαμβάνουν τον
103       υπολογισμό και την επικοινωνία*/
104    pthread_create(&thread_tids[0], NULL, EvenThread, (void *) NULL);
105    /*Ορισμός του CPU Affinity ώστε το 1ο συμμετρικό νήμα να εκτελείται
106      στον πρώτο εκ των δύο λογικών επεξεργαστών του Xeon*/
107    CPU_ZERO(&process0_mask);
108    CPU_SET(t1, &process0_mask);
```

```
109     if (sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask)) {
110        error("error: could not set pid %d's affinity.\n", gettid());
111     }
112     /* Κυρίως βρόχος όπου, αν θεωρήσουμε το k-οστό βήμα, γίνεται λήψη
113        του k+1 συνόρου, αποστολή του k-1 συνόρου και υπολογισμός του k-οστού
114        βήματος "ταυτόχρονα"*/
115     while (loops < MAXLOOPS - 1)
116     {
117        t = (++loops)%2;
118
119        //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
120        sema_wait(s2);
121        Receive(&requests[0]);
122
123        for(y = startY; y < (length+1)/2; y++)
124           for(x = startX; x < width+1; x++)
                 U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] - a*dt/dx*(U[t][y-1][x]
125  + U[t][y][x-1]);
126        MPI_Waitall(2, requests, status);
127        sema_signal(s1);
128        sema_wait(s2);
129
130
131        UnpackData((t+1)%2);
132        sema_signal(s1);
133     }
134
135     pthread_join(thread_tids[0], NULL);
136     /* Εδώ έχει τελειώσει η λειτουργία των νημάτων */
137     /*********************************************************************/
138     /*********************************************************************/
139     /*********************************************************************/
140
141     /* Αντίστοιχα με την αρχική πρώτη έναρξη επικοινωνίας χρειάζεται
142        κλείσιμο της επικοινωνίας με αποστολή του (K-1)-οστού, υπολογισμό
143        του K-οστού και αποστολή του K-οστού.*/
144     Send(&requests[2]);
145     MPI_Waitall(2, &requests[2], &status[2]);
146
147     t = (++loops)%2;
148     //K-οστός υπολογισμός
149     for(y = startY; y < length+1; y++)
150        for(x = startX; x < width+1; x++)
             U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] - a*dt/dx*(U[t][y-1][x] +
151  U[t][y][x-1]);
152     //K-οστή αποστολή δεδομένων
153     SetDataToSend(t);
154     Send(&requests[2]);
155     MPI_Waitall(2, &requests[2], &status[2]);
156
157     //Για να ξέρουμε ότι όλες οι διεργασίες τελείωσαν τους υπολογισμούς
158     MPI_Barrier(MPI_COMM_WORLD);
159     /*********************************************************************/
160     /*********************************************************************/
161
162     /*...*/
163  }
164
165  /*Εκτελούνται οι κατάλληλες αρχικοποιήσεις*/
```

```
166  void Init2D()
167  {
168     /*...*/
169
170     s1 = sema_create(0); s2 = sema_create(1);
171     if(my_rank -8 < 0) { t1 = 0; t2 = 2;}
172     else { t1 = 1; t2 = 3;}
173  }
174  /*...*/
175  /*Συνάρτηση που εκτελείται από το 2ο συμμερικό νήμα*/
176  void * EvenThread(void * arg)
177  {
178     int x, y, t, loops = 1;
179     double a = 1.001 , dt = 0.00001, dx = 0.00001;
180     MPI_Request requests[2];
181     MPI_Status status[2];
182     cpu_set_t process0_mask, cur_mask;
183
184     /*Ορισμός του CPU Affinity ώστε το 2ο συμμετρικό νήμα να εκτελείται
185       στον πρώτο εκ των δύο λογικών επεξεργαστών του Xeon*/
186     CPU_ZERO(&process0_mask);
187     CPU_SET(t2, &process0_mask);
188     if (sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask)) {
189        error("error: could not set pid %d's affinity.\n", gettid());
190     }
191
192
193     /* Κυρίως βρόχος όπου, αν θεωροήσουμε το k-οστό βήμα, γίνεται λήψη
194        του k+1 συνόρου, αποστολή του k-1 συνόρου και υπολογισμός του k-οστού
195        βήματος "ταυτόχρονα"*/
196     while (loops < MAXLOOPS - 1)
197     {
198        t = (++loops)%2;
199
200        //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
201        sema_wait(s1);
202        Send(&requests[0]);
203
204        for(y = (length+1)/2; y < length+1; y++)
205           for(x = startX; x < width+1; x++)
                 U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] - a*dt/dx*(U[t][y-1][x]
206                            + U[t][y][x-1]);
207        MPI_Waitall(2, requests, status);
208        sema_signal(s2);
209        sema_wait(s1);
210
211
212        SetDataToSend(t);
213        sema_signal(s2);
214     }
215     /*Τερματισμός του νήματος*/
216     pthread_exit((void *) 0);
217  }
218
219  /*...*/
```

Β.5 Παράλληλο Πρόγραμμα Προσομοίωσης Εξίσωσης Διάχυσης-2D με Επικαλυπτόμενη Δρομολόγηση σε Δύο Ασύμμετρα Νήματα[8]

```
 1  /*
 2   * Αρχείο advent2d_overlap.c
 3   * Υλοποίηση παράλληλου προγράμματος προσομοίωσης
 4   * της εξίσωσης διάχυσης δύο χωρικών διαστάσεων με
 5   * τη ροή να σπάει σε δύο ασύμμετρα νήματα
 6   *
 7   * Νικόλας Ιωάννου 2008
 8   */
 9  #define _GNU_SOURCE
10  #include <sched.h>
11  #include <stdio.h>
12  #include <stdlib.h>
13  #include <sys/time.h>
14  #include <pthread.h>//#include <sys/syscall.h>
15  #include <sys/types.h>
16  #include <linux/unistd.h>
17  #include <errno.h>
18  #include "mpi.h"
19  #include "lib.h"
20  #include "semaphore.h"
21
22  /*...*/
23  /*Κώδικας Υλοποίησης συνάρτησης gettid(), προσφορά του
24    Νίκου Ανατόπουλου*/
25
26  #define __syscall_clobber "r11","rcx","memory"
27  #define __syscall "syscall"
28
29  #define __syscall_return(type, res) \
30  do { \
31        if ((unsigned long)(res) >= (unsigned long)(-127)) { \
32                errno = -(res); \
33                res = -1; \
34        } \
35        return (type) (res); \
36  } while (0)
37
38
39
40  #define _syscall0(type,name) \
41        type name(void) \
42  { \
43        long __res; \
44        __asm__ volatile (__syscall \
45                                : "=a" (__res) \
                                : "0" (__NR_##name) : __syscall_clobber );
46  \
47        __syscall_return(type,__res); \
48  }
49
```

[8] Παρουσιάζουμε μόνο τα σημεία που διαφέρουν σημαντικά από τη βασική υλοποίηση, και μόνο την έκδοση χωρίς tiling, το tiling εφαρμόζεται όπως στο Β.2

```
50  static _syscall0(pid_t, gettid)
51
52  void* tf(void *args)
53  {
54        fprintf(stderr, "My id: %d\n", (int)gettid());
55
56        pthread_exit(NULL);
57  }
58
59  /*...*/
60  void * CommThread(void *);
61
62  sema_t * s1, * s2;
63
64  int main(int argc, char **argv)
65  {
66     /*...*/
67     pthread_t thread_tids[1];
68
69     cpu_set_t process0_mask;
70     unsigned long cur_mask;
71
72
73     /*...*/
74
75     /*********************************************************************/
76     /*****************ΕΠΙΚΟΙΝΩΝΙΑ ΚΑΙ ΥΠΟΛΟΓΙΣΜΟΙ**********************/
77     /*********************************************************************/
78
79     /*Πρώτη επικοινωνία ώστε να είναι έτοιμα τα πρώτα δεδομένα για να
80       μπορεί να ξεκινήσει η overlapping επικοινωνία*/
81     Receive(&requests[0]);
82     MPI_Waitall(2, &requests[0], &status[0]);
83     t = (++loops)%2;
84     UnpackData(t);
85     //Πρώτος υπολογισμός
86     for(y = startY; y < length+1; y++)
87        for(x = startX; x < width+1; x++)
           U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
88                      a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
89
90     SetDataToSend(t);
91     /*Δεύτερο receive για να μπορεί να γίνει ο υπολογισμός της
92       χρονικής στιγμής 2*/
93     Receive(&requests[0]);
94     MPI_Waitall(2, &requests[0], &status[0]);
95     UnpackData((t+1)%2);
96
97     /*********************************************************************/
98     /*****************************ΝΗΜΑΤΑ*******************************/
99     /*********************************************************************/
100    /* Η ροή σπάει σε δύο ασύμμετρα νήματα που αναλαμβάνουν τον
101       υπολογισμό και την επικοινωνία*/
102    pthread_create(&thread_tids[0], NULL, CommThread, (void *) NULL);
103
104    /*Ορισμός του CPU Affinity ώστε το νήμα υπολογισμού να εκτελείται
105      στον πρώτο εκ των δύο λογικών επεξεργαστών του Xeon*/
106    CPU_ZERO(&process0_mask);
107    CPU_SET(t1, &process0_mask);
```

```
      if (sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask))
108   {
109      error("error: could not set pid %d's affinity.\n", gettid());
110   }
111   /* Το κυρίως νήμα αναλαμβάνει τον υπολογισμό*/
112
113   /* Κυρίως βρόχος όπου, αν θεωρήσουμε το k-οστό βήμα, γίνεται λήψη
114      του k+1 συνόρου, αποστολή του k-1 συνόρου και υπολογισμός του
115      k-οστού  βήματος "ταυτόχρονα"*/
116   while (loops < MAXLOOPS - 1)
117   {
118      t = (++loops)%2;
119
120      sema_wait(s2);
121      for(y = startY; y < length+1; y++)
122         for(x = startX; x < width+1; x++)
               U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
123                          a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
124
125      sema_signal(s1);
126   }
127   pthread_join(thread_tids[0], NULL);
128
129   /* Εδώ έχει τελειώσει η λειτουργία των νημάτων */
130   /*****************************************************************/
131   /*****************************************************************/
132
133   /* Αντίστοιχα με την αρχική πρώτη έναρξη επικοινωνίας χρειάζεται
134      κλείσιμο της επικοινωνίας με αποστολή του (K-1)-οστού, υπολογισμό
135      του K-οστού και αποστολή του K-οστού.*/
136   Send(&requests[2]);
137   MPI_Waitall(2, &requests[2], &status[2]);
138
139   t = (++loops)%2;
140   sema_wait(s2);
141   //K-οστός υπολογισμός
142   for(y = startY; y < length+1; y++)
143      for(x = startX; x < width+1; x++)
            U[t][y][x] = (1+2*a*dt/dx)*U[(t+1)%2][y][x] -
144                       a*dt/dx*(U[t][y-1][x] + U[t][y][x-1]);
145   //K-οστή αποστολή δεδομένων
146   SetDataToSend(t);
147   Send(&requests[2]);
148   MPI_Waitall(2, &requests[2], &status[2]);
149
150   /*...*/
151 }
152
153 /*Εκτελούνται οι κατάλληλες αρχικοποιήσεις*/
154 void Init2D()
155 {
156   /*...*/
157
158   s1 = sema_create(0); s2 = sema_create(1);
159   if(my_rank -8 < 0) { t1 = 0; t2 = 2;}
160   else { t1 = 1; t2 = 3;}
161 }
162
163 /*...*/
```

```
164
165  /*Συνάρτηση που εκτελείται από το νήμα επικοινωνίας*/
166  void * CommThread(void * arg)
167  {
168     int t, loops = 1;
169     MPI_Request requests[4];
170     MPI_Status status[4];
171
172     cpu_set_t process0_mask, cur_mask;
173
174      /*Ορισμός του CPU Affinity ώστε το νήμα επικονωνίας να εκτελείται
175        στον δεύτερο εκ των δύο λογικών επεξεργαστών του Xeon*/
176     CPU_ZERO(&process0_mask);
177     CPU_SET(t2, &process0_mask);
178
179     if (sched_setaffinity(gettid(), sizeof(cpu_set_t), &process0_mask)){
180        error("error: could not set pid %d's affinity.\n", gettid());
181     }
182     if (sched_getaffinity(gettid(), sizeof(cpu_set_t), &cur_mask) < 0) {
183        error("1error: could not get pid %d's affinity.\n", gettid());
184     }
185
186
187     /* Κυρίως βρόχος όπου, αν θεωρήσουμε το k-οστό βήμα, γίνεται λήψη
188        του k+1 συνόρου, αποστολή του k-1 συνόρου και υπολογισμός του k-
189        οστού βήματος "ταυτόχρονα"*/
190     while (loops < MAXLOOPS - 1)
191     {
192
193        t = (++loops)%2;
194        //Επικοινωνία για αποστολή και παραλαβή μηνυμάτων
195        Receive(&requests[0]);
196        Send(&requests[2]);
197
198        MPI_Waitall(4, requests, status);
199        sema_wait(s1);
200        SetDataToSend(t);
201        UnpackData((t+1)%2);
202        sema_signal(s2);
203     }
204
205     pthread_exit((void *) 0);
206  }
207
208  /*...*/
209
```