Ατομική Διπλωματική Εργασία

MULTICORE PROGRAMMING

Πάτροκλος Πατρόκλου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2015

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Multicore Programming

Πάτροκλος Πατρόκλου

Επιβλέπων Καθηγητής
Παρασκευάς Ευριπίδου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2015

Ευχαριστίες

Φθάνοντας στο τέλος των σπουδών μου στο Τμήμα Πληροφορικής τα άτομα που πρέπει να ευχαριστήσω είναι πάρα πολλά. Σε αυτά τα τέσσερα όμορφα χρόνια που πέρασα είχα πάντα δίπλα μου τους συμφοιτητές μου. Μαζί περάσαμε αμέτρητες ώρες στα εργαστήρια δουλεύοντας σε διάφορα projects και ασκήσεις. Μαζί περάσαμε δυσκολίες, αποτυχίες και επιτυχίες. Σας ευχαριστώ αγαπημένοι μου συμφοιτητές και πάνω απ' όλα φίλοι και σας εύχομαι κάθε επιτυχίας στις ζωές σας..

Θα ήθελα να ευχαριστήσω επίσης την Ειρήνη Σμολέσκη που πέρασε μαζί μου αυτά τα τέσσερα χρόνια και ήταν πάντα δίπλα μου σε κάθε στιγμή χαράς και λύπης.

Επίσης θα ήθελα να ευχαριστήσω την οικογένεια μου που με στήριζε αυτά τα τέσσερα χρόνια. Η στήριξη που μου παρείχε η οικογένεια μου ήταν το πιο σημαντικό συστατικό στην προσπάθεια απόκτησης του πτυχίου.

Τέλος, θα ήθελα να ευχαριστήσω τον Καθηγητή Παρασκευά Ευριπίδου που μου έδωσε την ευκαιρία να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα και τον Διδακτορικό ερευνητή Γιώργο Ματθαίου που με τις συμβουλές τους με καθοδηγούσαν από την αρχή της εργασίας μέχρι το τέλος της.

Σας Ευχαριστώ Όλους

Περίληψη

Στις μέρες μας η τεχνολογία της πληροφορικής έχει γίνει τρόπος ζωής για τους περισσότερους ανθρώπους. Ολοένα και περισσότερες ανάγκες γεννιούνται καθημερινά που αφορούν στην ποιότητα των υπηρεσιών που παρέχουμε σαν επιστήμονες πληροφορικής στους χρήστες. Τεράστιοι όγκοι δεδομένων ανταλλάσσονται μέσω του διαδικτύου σε όλο τον κόσμο αποτέλεσμα που επιβάλει να χρησιμοποιούνται αποδοτικοί τρόποι για να επεξεργάζονται αυτά τα δεδομένα. Με τον παράλληλο προγραμματισμό μπορούμε όπου είναι δυνατό να έχουμε επιταχύνσεις στους χρόνους εκτέλεσης διάφορων προβλημάτων.
Η εργασία αυτή έχει ως σκοπό την μελέτη του Προγραμματισμού σε Πολλαπλούς πυρήνες. Συγκεκριμένα θα αναλυθεί ο λόγος που οδηγηθήκαμε από τους επεξεργαστές με ένα πυρήνα στους επεξεργαστές με πολλούς πυρήνες και θέματα που αφορούν τον προγραμματισμό σε πολλαπλούς πυρήνες. Συν τοις άλλοις, θα γίνει αναφορά σε προγραμματιστικές βιβλιοθήκες που χρησιμοποιούνται για την συγγραφή παράλληλων προγραμμάτων.
Επίσης στην εργασία αυτή θα παρουσιαστεί μελέτη σχετικά με το HPCG Benchmark ενός προγράμματος που είναι υποψήφιο για να χρησιμοποιείται στο μέλλον στην κατάταξη των TOP500 συστημάτων του κόσμου.

Τέλος, χρησιμοποιώντας το HPCG Benchmark εισαγάγαμε το FREDDO Framework το οποίο είναι ένα framework προγραμματισμού για γλώσσα προγραμματισμού C++ που υλοποιεί λειτουργείες του Data Driven Multithreading (DDM). Στα πλαίσια της εργασίας αυτής μελετήσαμε το FREDDO και κάναμε κάποιες εισηγήσεις που αφορούσαν στην υλοποίηση και στις λειτουργείες του.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλη Επεξεργασία

Η παράλληλη επεξεργασία αρχίζει σταδιακά να γίνεται απαραίτητη στις γνώσεις ενός επιστήμονα πληροφορικής. Στις μέρες μας όσο καλό και να είναι ένα προϊόν που κυκλοφορεί στον τομέα της πληροφορικής αν δεν έχει υψηλές επιδόσεις δεν μπορεί να ανταπεξέλθει στις ανάγκες των πελατών. Μέσω της παράλληλης επεξεργασίας έχουμε την δυνατότητα συγγραφής ταχύτερων υλοποιήσεων. Με τον όρο παράλληλη επεξεργασία εννοούμε την ταυτόχρονη αξιοποίηση υπολογιστικών πόρων για την επίλυση ενός υπολογιστικού προβλήματος. Στις μέρες μας οι πόροι ενός συστήματος αυξάνονται συνεχώς φτάνοντας σε δεκάδες από υπολογιστικούς πυρήνες σε συνηθισμένους υπολογιστές και μέχρι εκατομμύρια από πυρήνες σε υπολογιστικά συστήματα υπερ.-υπολογιστών. Με την παράλληλη επεξεργασία υπάρχει η δυνατότητα αξιοποίησης όλων αυτών των πόρων για να επιτυγχάνεται μέχρι και η μέγιστη επιτάχυνση.

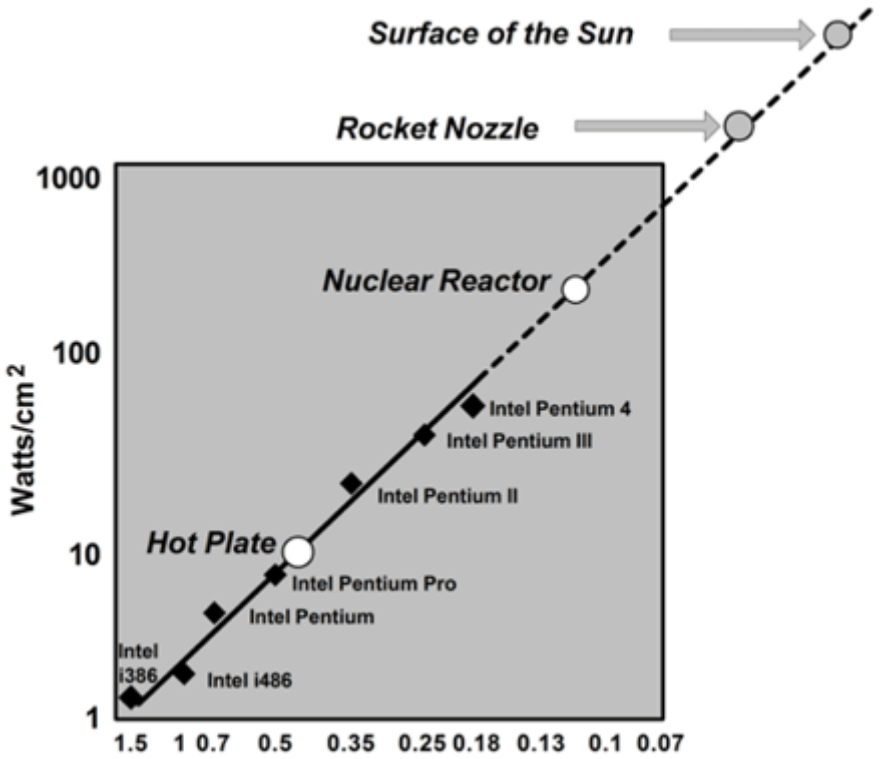
Η παράλληλη επεξεργασία άρχισε να εμφανίζεται κυριολεκτικά σε συστήματα αφού οι σειριακοί επεξεργαστές έφθασαν στο τέλος της ζωής τους. Την περίοδο όπου κυριαρχούσαν οι σειριακοί επεξεργαστές ο μόνος τρόπος βελτίωσης της απόδοσης ενός προγράμματος ήταν με την αύξησή της απόδοσης της συχνότητας ενός επεξεργαστή. Η μέθοδος αυτή βίωσε κορεσμό όταν παράγοντες όπως η ενέργεια και η θερμότητα ήταν αποτρεπτικοί στην περεταίρω αύξηση της συχνότητας των επεξεργαστών. Τότε οι κατασκευαστές στράφηκαν σε μια εντελώς διαφορετική κατεύθυνση από αυτή που κινούνταν ξεκινώντας την κατασκευή επεξεργαστών με περισσότερους από ένα πυρήνα με αποτέλεσμα να οδηγηθούμε έτσι σε μια νέα εποχή στην παράλληλη επεξεργασία.

Σήμερα η συντριπτική πλειοψηφία των επεξεργαστών που κατασκευάζονται είναι πολυπύρηνοι επεξεργαστές και αυτό δεν αναμένεται να αλλάξει σύντομα. Αντιθέτως οι μελλοντικές τάσεις στον κόσμο της αρχιτεκτονικής υπολογιστών τείνουν προς επεξεργαστές με ακόμη περισσότερους πυρήνες και αναμένεται ότι θα ξεπερνούν τους 100 στο άμεσο μέλλον. Είναι αναντίλεκτο λοιπόν το γεγονός ότι η παράλληλη επεξεργασία είναι αδήριτη ανάγκη να ενταχθεί στη ζωή των σημερινών και μελλοντικών προγραμματιστών γιατί θα ήταν αδιανόητο να έχουμε στην διάθεση μας εκατοντάδες πυρήνες και εμείς να συνεχίζουμε να γράφουμε προγράμματα με τον παραδοσιακό σειριακό τρόπο αφήνοντας έτσι ανεκμετάλλευτους τους υπολογιστικούς πόρους των συστημάτων.

1.2 Εξέλιξη των πολυπύρηνων επεξεργαστών

Μέχρι το 2005 οι σειριακοί επεξεργαστές υπερτερούσαν των πολυπύρηνων επεξεργαστών. Οι μεγάλες εταιρείες κατασκευής επεξεργαστών Intel και Amd εστιάζονταν μόνο στην εξέλιξη των σειριακών επεξεργαστών. Συγκεκριμένα ασχολούνταν μόνο με το να αυξάνουν την συχνότητα των σειριακών επεξεργαστών. Οι εταιρείες αντιμετώπισαν μεγάλες δυσκολίες όταν έφθασαν κοντά στα 4GHz στην συχνότητα των επεξεργαστών. Στο σημείο αυτό παρουσιάστηκαν προβλήματα με το σημαντικότερο το πρόβλημα του Power Wall (σχήμα 1.2). Το πρόβλημα του Power Wall εμφανίστηκε όταν η συχνότητα των σειριακών επεξεργαστών ήταν τόσο μεγάλη που ξεπερνούσε τα όρια της ψύξης που μπορούσε να παραχθεί από τον αέρα. Επιπρόσθετα, χρειαζόταν τεράστια αύξηση της ενέργειας για να επιτευχθεί ελάχιστα μεγαλύτερη ταχύτητα. Λόγω της ενέργειας που χρειαζόταν για να μπορούν οι επεξεργαστές να τρέχουν σε τέτοιες ψηλές συχνότητες η θερμότητα αυξανόταν φθάνοντας σε επίπεδα χωρίς δυνατότητα διαχείρισης από τις γνωστές τεχνικές ψύξης. Επομένως, δεν ήταν ωφέλιμο να συνεχιστεί η αύξηση της συχνότητας γιατί θα χρειαζόταν να χρησιμοποιηθούν άλλοι τρόποι ψύξης(π.χ. ψύξη με νερό) . Από τη μια θα υπήρχε αύξηση στο κόστος της κατασκευής που θα χρειάζονταν και μεγαλύτερο χώρο για να μπορούν να εγκατασταθούν μετατρέποντας την κατασκευή και την ευχρηστία των υπολογιστών ασύμφορη.

Επίσης λόγω του Power Wall εκτός από τα προβλήματα ψύξης παρουσιάστηκαν και προβλήματα ενέργειας. Οι επεξεργαστές όσο αυξανόταν η συχνότητα τους αυξανόταν και η κατανάλωση ενέργειας. Επομένως, με την αύξηση της συχνότητας και συνεπώς της ενέργειας θα ήταν αδύνατη η χρησιμοποίηση των επεξεργαστών σε φορητές συσκευές γιατί η μπαταρία θα εξαντλούνταν πολύ γρήγορα.



Σχήμα 1.2: Το πρόβλημα του Power Wall

Από το 2005 μέχρι και σήμερα οι πολυπύρηνοι επεξεργαστές συνεχίζουν να εξελίσσονται ασταμάτητα. Οι πρώτοι πολυπύρηνοι επεξεργαστές που κυκλοφόρησαν στην αγορά ήταν οι λεγόμενοι Dual-core επεξεργαστές που είχαν 2 πυρήνες. Σήμερα έχουν κάνει την εμφάνιση τους συμβατικοί επεξεργαστές από την AMD και την Intel που φθάνουν μέχρι και τους οκτώ πυρήνες.

1.3 Μελλοντικές τάσεις στην παράλληλη επεξεργασία

Η μελλοντική τάση στην παράλληλη επεξεργασία είναι η μετάβαση από τα συστήματα πολλαπλών πυρήνων(multicore) σε συστήματα πολύ-πυρήνων(many core). Με το όρο πολυπύρηνες εννοούμε συστήματα που έχουν από δεκάδες μέχρι εκατοντάδες πυρήνες. Σήμερα αρχίζουν σιγά σιγά να εμφανίζονται επεξεργαστές με περισσότερους από δέκα πυρήνες ενώ προβλέπετε ότι στο άμεσο μέλλον οι πυρήνες που θα έχει ένας συμβατός υπολογιστής θα ξεπερνούν τους 100.

Φυσικά με την αύξηση των πυρήνων αυξάνονται και τα θέματα που πρέπει να αντιμετωπιστούν προτού γίνου πραγματικότητα. Συγκεκριμένα είναι δεδομένο ότι η αύξηση των πυρήνων θα φέρει πολλά προβλήματα σε θέματα της επικοινωνίας και της σύνδεσης των πυρήνων. Θα πρέπει να δημιουργηθούν νέα πρωτόκολλα συνέπειας μνήμης γιατί τα γνωστά σημερινά πρωτόκολλα (π.χ snooping) θα πάσχουν από την ανάγκη συνεχούς απασχόλησης του διαύλου μνήμης (memory bus) για να εκτελούν τις λειτουργείες τους (π.χ invalidates).

Επίσης θέματα ενέργειας και ψύξης θα πρέπει να αντιμετωπιστούν αποδοτικά για να μπορούν να είναι αξιόπιστοι οι νέοι επεξεργαστές και να αντιμετωπίζονται αποδοτικά πιθανά προβλήματα υπερθέρμανσης. Επιπρόσθετα, είναι προφανές ότι θα πρέπει να δημιουργηθούν αποδοτικοί τρόποι διαχείρισης ενέργειας για να αποτρέπονται καταστάσεις όπου όλοι οι επεξεργαστές θα δουλεύουν σε ψηλά ποσοστά απόδοσης ταυτόχρονα με αποτέλεσμα να σπαταλιέται μεγάλη ενέργεια και να υπάρχει κίνδυνος υπερθέρμανσης.

1.4 Μεθοδολογία

Στα πλαίσια της εργασίας αυτής η δουλειά οργανώθηκε σε τρία μέρη. Στο πρώτο μέρος έγινε μελέτη σχετικά με την παράλληλη επεξεργασία και τον προγραμματισμό σε πολυπύρηνους επεξεργαστές. Μελετήθηκαν θέματα σχετικά με τις αρχιτεκτονικές παράλληλης επεξεργασίας και οι παράγοντες που επηρεάζουν την απόδοση ενός παράλληλου προγράμματος. Επιπρόσθετα, μελετήθηκαν διάφορα προγραμματιστικά πλαίσια που χρησιμοποιούνται σήμερα για την συγγραφή παράλληλων προγραμμάτων.

Στο δεύτερο μέρος αναλύσαμε το HPCG Benchmark μέσα από δημοσιεύσεις των δημιουργών του[3]. Για περαιτέρω ανάλυση του benchmark τρέξαμε δικά μας πειράματα σε δικές μας μηχανές και μελετήσαμε θέματα που αφορούσαν την επιτάχυνση του benchmark.

Στο τρίτο και τελευταίο μέρος υλοποιήσαμε μια δική μας εκδοχή του benchmark χρησιμοποιώντας το πλαίσιο προγραμματισμού FREDDO. Το FREDDO είναι ένα πλαίσιο προγραμματισμού που υλοποιείται αυτή την στιγμή στο DDM Group του πανεπιστημίου Κύπρου. Το FREDDO ακολουθεί το μοντέλο προγραμματισμού Data Driven. Δημιουργός του FREDDO είναι ο διδακτορικός ερευνητής του τμήματος Πληροφορικής Γιώργος Ματθαίου οποίος μας παρέχει και το πλαίσιο. Μέσω αυτής της υλοποίησης είμασταν σε θέση να προτείνουμε αλλαγές στην έκδοση του FREDDO αλλά και να εξάγουμε χρήσιμα συμπεράσματα συγκρίνοντας την δική μας έκδοση με την έκδοση της OpenMP εκτέλεσης.

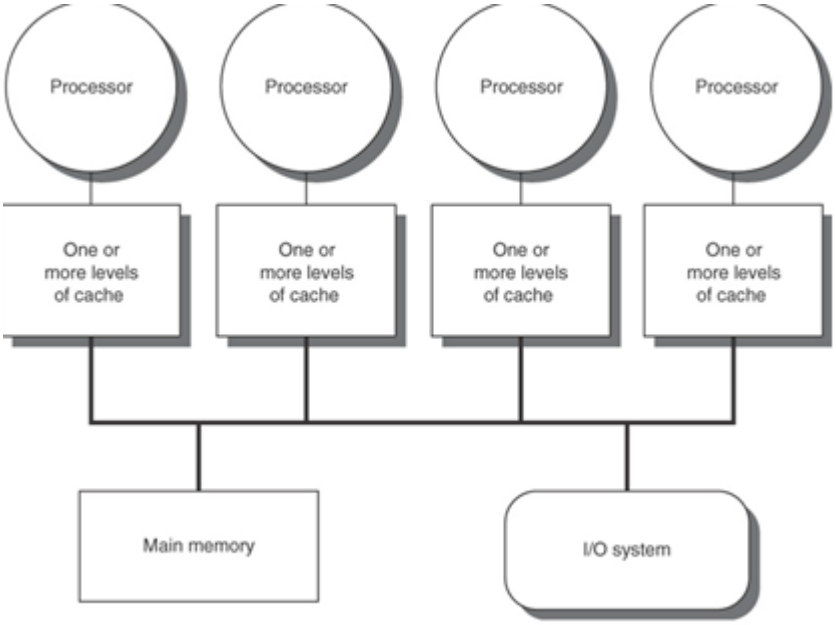
Κεφάλαιο 2

Θέματα Παράλληλου Προγραμματισμού

2.1 Αρχιτεκτονικές παράλληλης επεξεργασίας

Η παράλληλη επεξεργασία μπορεί να εφαρμοστεί σε διαφόρου τύπου αρχιτεκτονικές υπολογιστών με τις πιο διαδεδομένες αυτές της κοινής μνήμης και της κατανεμημένης μνήμης.

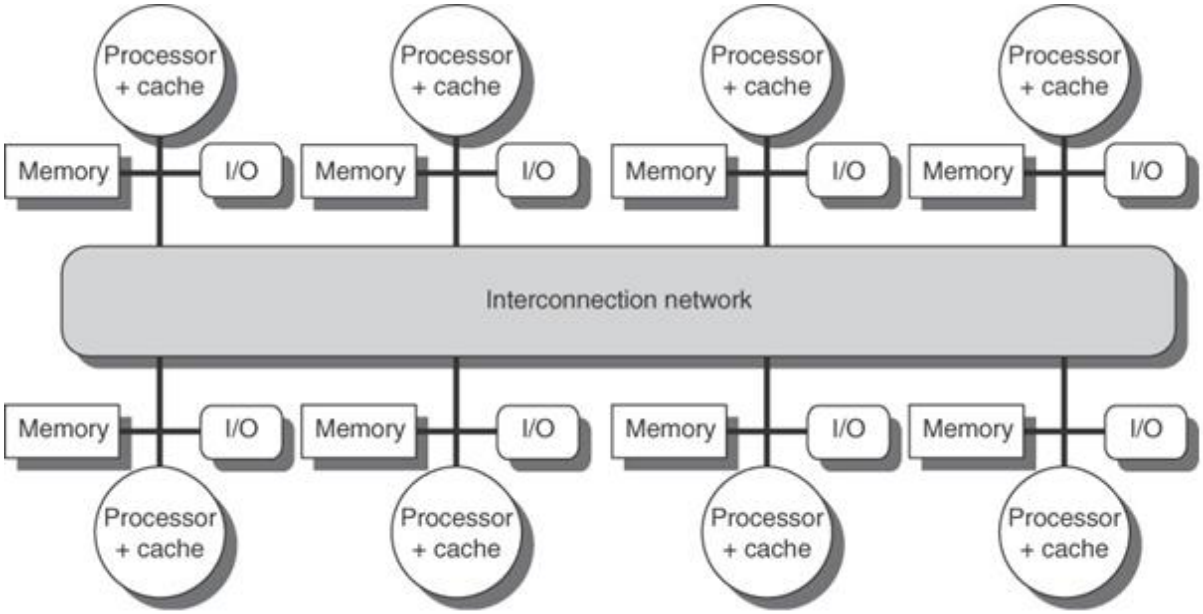
Το μοντέλο της κοινής μνήμης (shared memory) είναι το μοντέλο όπου όλοι οι επεξεργαστές μπορούν να έχουν πρόσβαση σε οποιοδήποτε μέρος της κύριας μνήμης. Τέτοιου είδος μοντέλα βρίσκονται κατά πλειοψηφία σήμερα σε συμβατούς υπολογιστές.



Σχήμα 2.1.1 Παράδειγμα αρχιτεκτονικής κοινής μνήμης

Στην αρχιτεκτονική κοινής μνήμης κάθε επεξεργαστής έχει την δική του ανεξάρτητη κρυφή μνήμη και όλοι οι επεξεργαστές χρησιμοποιούν ένα κοινό δίαυλο επικοινωνίας για να μπορούν να γράφουν και να διαβάζουν από την κοινή κύρια μνήμη.

Το μοντέλο της κατανεμημένης μνήμης(distributed memory) είναι το μοντέλο που κάθε επεξεργαστής έχει την δική του αποκλειστική μνήμη και κανένας άλλος επεξεργαστής δεν μπορεί να έχει πρόσβαση σε αυτή. Τέτοιου είδους μοντέλα βρίσκονται κυρίως σε υπερ-υπολογιστές και σε συστήματα cluster.



Σχήμα 2.1.2 Παράδειγμα αρχιτεκτονικής κατανεμημένης μνήμης

Όπως παρουσιάζεται και στο σχήμα 2.1.2 κάθε επεξεργαστής έχει δική του ανεξάρτητη κύρια μνήμη και συνεπώς κρυφή μνήμη. Οι κόμβοι μπορούν να επικοινωνήσουν μεταξύ τους χρησιμοποιώντας το δίκτυο διασύνδεσης.

Κάθε αρχιτεκτονική έχει και τα θετικά και τα αρνητικά σημεία της. Για παράδειγμα στο μοντέλο της κοινής μνήμης δεν χρειάζεται να ορίζονται ανταλλαγές δεδομένων μεταξύ των νημάτων γιατί κάθε νήμα μπορεί να έχει πρόσβαση σε όλα τα δεδομένα. Ως επακόλουθο όμως της δυνατότητας αυτής υπάρχουν άλλα προβλήματα συνέπειας μνήμης. Το πιο γνωστό πρόβλημα είναι το race condition που θα εξηγηθεί σε μεταγενέστερο στάδιο στην εργασία αυτή.

Στο μοντέλο της κατανεμημένης μνήμης δεν υπάρχουν προβλήματα συνέπειας μνήμης αφού ο κάθε επεξεργαστής έχει την δική του μνήμη την οποία επεξεργάζεται μόνο αυτός. Υπάρχουν άλλα προβλήματα όμως που είναι αποτέλεσμα της μη-κοινής μνήμης. Το κύριο πρόβλημα είναι η επικοινωνία μεταξύ επεξεργαστών που επηρεάζει κατά πολύ την συνολική απόδοση ενός παράλληλου προγράμματος.

2.2 Amdahl's Law

Ο νόμος του Amdahl είναι μια φόρμουλα υπολογισμού της μέγιστης αναμενόμενης επιτάχυνσης σε παράλληλα προγράμματα με την προϋπόθεση ότι τα μέγεθος του προβλήματος είναι το ίδιο και στην παράλληλη εκτέλεση και στην σειριακή. Συγκεκριμένα, ο νόμος του Amdahl λαμβάνει υπόψη το ποσοστό του κώδικα που παραμένει σειριακός, τον αριθμό των παράλληλων κομματιών και τον χρόνο εκτέλεσης του σειριακού προγράμματος.

Ο νόμος του Amdahl λέει ότι αν το P είναι το ποσοστό του κώδικα που μπορεί να είναι παράλληλο και το (1-P) είναι το ποσοστό του κώδικα που δεν μπορεί να γίνει παράλληλο(το σειριακό κομμάτι) τότε η αναμενόμενη επιτάχυνση είναι:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Όπου N ο αριθμός των παράλληλων κομματιών. Για παράδειγμα αν χρησιμοποιούμε 10 νήματα στα παράλληλα κομμάτια το N θα ισούται με 10.

2.3 Παράγοντες καθορισμού απόδοσης παράλληλων προγραμμάτων

Η απόδοση ενός παράλληλου προγράμματος εξαρτάται από πολλούς παράγοντες. Πολλοί υποθέτουν ότι με τον παράλληλο προγραμματισμό επιτυγχάνεται πάντα επιτάχυνση γεγονός που ως γνωστό απέχει κατά πολύ από την πραγματικότητα. Στα πλαίσια της εργασίας αυτής μελετήσαμε τις επιπτώσεις που έχουν σε ένα παράλληλο πρόγραμμα διάφοροι παράγοντες όπως : κόστος δημιουργίας νημάτων και Context Switching , θέματα συνέπειας μνήμης(false sharing) και το κόστος των mutexes/barriers.

Κόστος Δημιουργίας νημάτων και Context Switching:

Είναι ευρέως γνωστό ότι η δημιουργία νημάτων έχει σημαντικό κόστος στην απόδοση ενός παράλληλου προβλήματος. Πολλές φορές αν τα νήματα που δημιουργούνται είναι υπεραρκετά από τις ανάγκες του προβλήματος τότε η επιτάχυνση δεν θα είναι η αναμενόμενη γιατί το κόστος δημιουργίας θα επικαλύπτει την οποιαδήποτε επιτάχυνση.

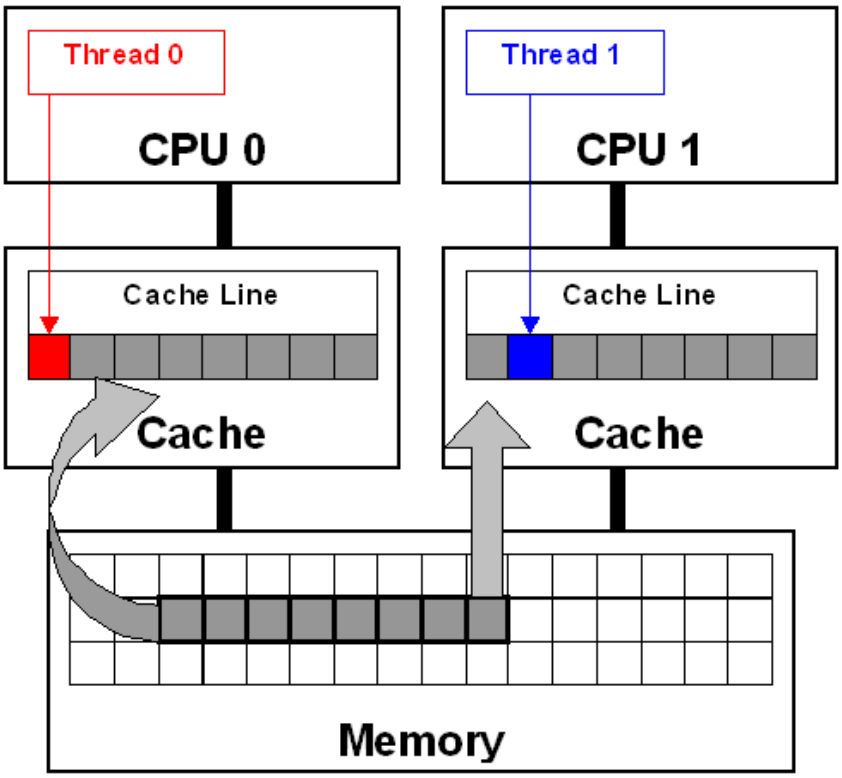
Επίσης, αν τα νήματα που δημιουργούνται για να τρέξουν ένα κομμάτι παράλληλου κώδικα είναι περισσότερα από τα νήματα που υποστηρίζονται από τους φυσικούς πόρους του συστήματος και μπορούν πραγματικά να τρέχουν παράλληλα στην μηχανή τότε παρουσιάζεται το θέμα του Context Switching.

Όταν παρουσιάζεται το Context Switching τα νήματα μπορούν να γίνονται interrupt, δηλαδή να σταματούν την εκτέλεση τους και την θέση τους στον επεξεργαστή παίρνει ένα άλλο νήμα. Η κατάσταση στην οποία ήταν τα νήματα πριν από το interrupt αποθηκεύεται και όταν μετέπειτα επανέλθουν στον επεξεργαστή, συνεχίζουν την εκτέλεση από το σημείο που ήταν όταν έγιναν interrupt.

Ο χειρισμός του Context Switching είναι αποκλειστικά θέμα του λειτουργικού συστήματος και δεν μπορεί ο προγραμματιστής να κατευθύνει την εκτέλεση των νημάτων σε περίπτωση που υπάρχει θέμα Context Switching.

Προβλήματα συνέπειας μνήμης:

Αν το πρόβλημα είναι δομημένο με τρόπο που οι μεταβλητές που χρησιμοποιούνται από τα νήματα είναι κοντά στην φυσική μνήμη τότε το πιο πιθανό είναι ότι η εκτέλεση θα έχει το λεγόμενο πρόβλημα false-sharing. Το πρόβλημα του false-sharing εμφανίζεται όταν ενώ τα νήματα έχουν σωστές τιμές στην κρυφή μνήμη τους εξαναγκάζονται να αποταθούν στην κύρια μνήμη για να πάρουν την "νέα" τιμή γιατί η δική τους τιμή έγινε invalidate από το πρωτόκολλο συνέπειας μνήμης. Στο σχήμα 2.3 απεικονίζεται το πρόβλημα του false sharing:

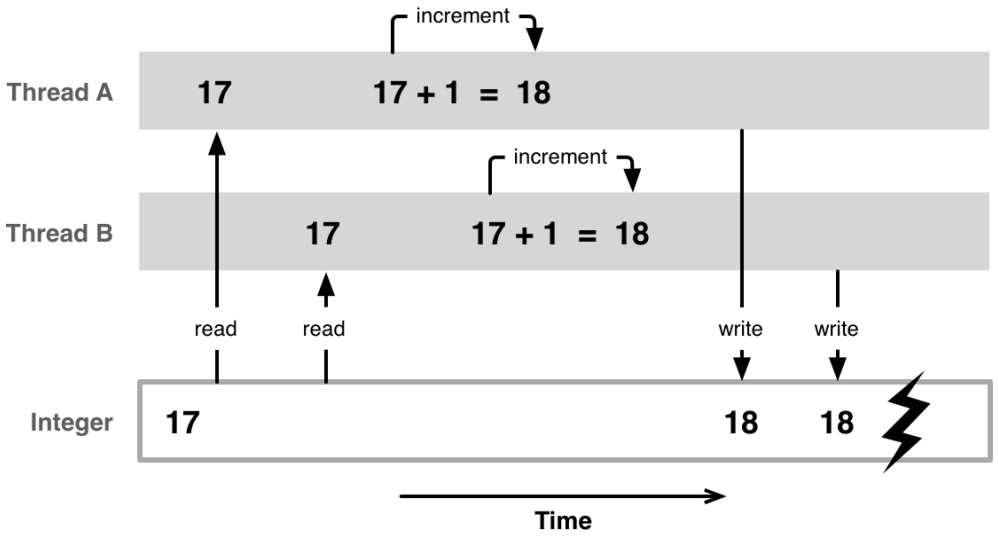


Σχήμα 2.3.1: Αναπαράσταση false sharing

Η αντιμετώπιση του false sharing είναι αποκλειστική ευθύνη του προγραμματιστή. Συγκεκριμένα ο προγραμματιστής πρέπει να δομήσει τον παράλληλο κώδικα του με τέτοιο τρόπο που να αποφεύγει το false sharing. Η συνήθης τεχνική που ακολουθείται είναι τα νήματα να έχουν τοπικές μεταβλητές και στο τέλος με την χρήση mutex να γίνεται ανανέωση της κοινής μεταβλητής αν είναι αναγκαίο.

Race condition:

Το πρόβλημα του race condition είναι ίσως το πιο κοινό πρόβλημα που έχουν να αντιμετωπίσουν οι προγραμματιστές όταν γράφουν παράλληλα προγράμματα. Το πρόβλημα αυτό παρουσιάζεται όταν ανεξάρτητα νήματα που εκτελούνται παράλληλα γράφουν σε κοινή μεταβλητή. Στο σχήμα 2.3.1 παρουσιάζεται γραφικά το πρόβλημα του Race Condition.



Σχήμα 2.3.1: Το πρόβλημα του Race condition

Στο παράδειγμα του σχήματος 2.3.1 δύο νήματα διαβάζουν την τιμή 17 από την μνήμη. Και τα δύο νήματα προσθέτουν ένα στην τιμή και γράφουν και τα δύο την τιμή 18 στην μνήμη. Το λάθος παρουσιάζεται επειδή όταν τα νήματα έχουν πλέον την τιμή στην τοπική τους μνήμη δεν μπορούν να ξέρουν αν η τιμή αυτή άλλαξε από κάποιο άλλο νήμα.

Προβλήματα από mutexes/barriers:

Η συνεχής χρήση mutexes/barriers μπορεί να αποκομίσει στο παράλληλο πρόγραμμα την γνωστή κατάσταση serialization. Η κατάσταση serialization παρουσιάζεται όταν όλα τα νήματα ή διεργασίες είναι αδρανείς γιατί δεν μπορούν να εκτελούν παράλληλα κομμάτια σε μια χρονική στιγμή. Ένα παράδειγμα που πάσχει από serialization είναι το πιο κάτω σενάριο: Όλα τα threads εκτελούν κάποιες λειτουργίες (π.χ άθροισμα ενός πίνακα τιμών) και πρέπει να προσθέτουν σε μια κοινή μεταβλητή το αποτέλεσμα τους. Για να αποφευχθεί το πρόβλημα του Race Condition έστω ότι ο προγραμματιστής όρισε ένα mutex για να εγγυάται ότι ένα νήμα θα έχει πρόσβαση στην μεταβλητή κάθε φορά. Όμως, αυτή η τεχνική έχει αποτέλεσμα όλα τα νήματα να περιμένουν στην σειρά για να μπορέσουν να συνεχίσουν. Δηλαδή το πρόγραμμα παύει να εκτελείται παράλληλα σε αυτό το χρονικό σημείο.

Κεφάλαιο 3

Μοντέλα παράλληλου Προγραμματισμού

3.1 OpenMP

Το OpenMP [6] είναι μια προγραμματιστική διεπαφή (API) η οποία υποστηρίζει παράλληλο shared memory προγραμματισμό σε γλώσσες προγραμματισμού όπως C,C++ και Fortran. Μπορεί να τρέξει στις περισσότερες αρχιτεκτονικές επεξεργαστών και σε πολλά λειτουργικά συστήματα. Συγκεκριμένα τα λειτουργικά συστήματα που υποστηρίζει το OpenMP σήμερα είναι: Linux,Windows,Mac OS X, Solaris,AIX και HP-UX.

Το OpenMP αποτελείται από σετ οδηγιών του μεταγλωττιστή (compiler directives) ,library routines και μεταβλητές περιβάλλοντος τα οποία καθοδηγούν την run time συμπεριφορά ενός προγράμματος.

Μέσω ενός κινητού και scalable μοντέλου που χρησιμοποιεί το OpenMP προσφέρει στους προγραμματιστές μια απλή διεπαφή που τους δίνει την δυνατότητα να γράφουν παράλληλες εφαρμογές από προσωπικούς υπολογιστές μέχρι υπέρ-υπολογιστές.

Βασισμένο σε ένα μοντέλο fork-join το OpenMP δημιουργεί παράλληλα κομμάτια κώδικα, όπου ζητάτε από τον προγραμματιστή. Για κάθε παράλληλο κομμάτι το OpenMP δημιουργεί νήματα(threads) τα οποία υπάρχουν μόνο κατά την διάρκεια εκτέλεσης του παράλληλου κομματιού. Με την ολοκλήρωση του παράλληλου κώδικα τα νήματα αυτά καταστρέφονται και συνεχίζεται η σειριακή εκτέλεση.

Το OpenMP εξ ορισμού δημιουργεί όσα νήματα είναι φυσικά διαθέσιμα στην μηχανή που τρέχει το πρόγραμμα. Για παράδειγμα σε μια μηχανή με τέσσερα νήματα το OpenMP θα δημιουργήσει τέσσερα νήματα ενώ σε μηχανή με 32 νήματα για το ίδιο πρόβλημα το OpenMP θα δημιουργήσει 32 νήματα. Αυτό προσφέρει ένα εύκολο τρόπο στους προγραμματιστές να δημιουργούν scalable κώδικα που προσαρμόζεται ανάλογα με την μηχανή που τρέχει χωρίς να

χρειάζεται να ελέγχουν τους διαθέσιμους πόρους κάθε μηχανής πριν να τρέξουν τον κώδικα τους.

Η εισαγωγή του OpenMP στο κώδικα είναι αρκετά απλή. Στο σχήμα 3.1 παρουσιάζεται το πώς με μια γραμμή μπορεί να γίνει παράλληλη μια επανάληψη for. Φυσικά, ο προγραμματιστής πρέπει να χειρίζεται τα διάφορα προβλήματα συνέπειας μνήμης και εξαρτάται από αυτό αν τα αποτελέσματα θα είναι ορθά.

```
#pragma omp parallel for

for (int i = 0; i< sz; i++)
   prod[i] = x[i] * y[i];
```

Σχήμα 3.1: Παράδειγμα παράλληλου for με OpenMP

Μέσω του OpenMP παρέχονται επίσης πολλές λειτουργίες που ευκολύνουν την χρήση του από τους προγραμματιστές. Υπάρχουν για παράδειγμα built-in λειτουργίες όπως το reduction.

3.2 MPI

Η Διεπαφή Μεταβίβασης Μηνυμάτων (Message Parsing Interface) [7] είναι μια βιβλιοθήκη μεταβίβασης μηνυμάτων στατικής σύνδεσης βασισμένη στην προδιαγραφή του MPI Forum. Στο MPI Forum συμμετέχουν από 40 οργανισμοί στους οποίους συμπεριλαμβάνονται προμηθευτές , ερευνητές, σχεδιαστές βιβλιοθηκών λογισμικού και χρήστες.

Η Διεπαφή Μεταβίβασης Μηνυμάτων χρησιμοποιείται κυρίως σε συστήματα κατανεμημένης μνήμης. Οι λειτουργίες ανταλλαγής μνημάτων που παρέχει η MPI είναι ιδανικές σε συστήματα που δεν έχουν κοινή μνήμη. Μέσω της MPI μπορούν οι διάφορες διεργασίες ενός προγράμματος να επικοινωνούν μεταξύ τους ανταλλάσσοντας δεδομένα. Παρέχονται εντολές send και receive και blocking και not blocking.

Η λογική του MPI είναι απλή. Μέσω ενός Communicator το πρόγραμμα είναι κοινό σε όλες τις διεργασίες και συνεπώς όλες οι διεργασίες εκτελούν τον ίδιο κώδικα. Περαιτέρω διαμόρφωση του κώδικα επιτυγχάνεται συνήθως ανάλογα με το ID της κάθε διεργασίας . Δηλαδή εξαρτάται απόλυτα από τον προγραμματιστή που θα εκτελεί η κάθε διεργασία.

Το μοναδικό αυτό ID ονομάζεται Rank στην MPI και χρησιμοποιείται συνήθως για να κατευθύνεται από τον προγραμματιστή η εκτέλεση κάθε διεργασίας (π.χ if rank=0 do this / if rank=1 do that).

Blocking εντολές MPI:

Με το όρο blocking εννοούμε ότι η διεργασία που αποστέλλει το μήνυμα θα παραμείνει σε αδράνεια μέχρι το μήνυμα να παραληφθεί πλήρως από τον παραλήπτη.
Υπάρχουν εντολές blocking αποστολής και παράληψης μηνυμάτων και οι εντολές αυτές είναι η MPI_send και MPI_receive.

Non-Blocking εντολές MPI:

Με το όρο non-blocking εννοούμε ότι η διεργασία συνεχίζει την εκτέλεση της ακόμη και αν δεν λάβει τα δεδομένα ο παραλήπτης. Το ίδιο ισχύει και στην περίπτωση του receive. Η διεργασία συνεχίζει την εκτέλεση της ακόμη και αν δεν έχουν ληφθεί οποιαδήποτε δεδομένα από τον αποστολέα.

MPI_Barrier:

Μία από τις σημαντικότερες λειτουργίες που παρέχει το MPI είναι η λειτουργία του barrier. Η λειτουργία αυτή όταν ορισθεί κατευθύνει τις διεργασίες να περιμένουν σε ένα σημείο μέχρι να φθάσουν στο σημείο αυτό όλες οι διεργασίες. Η χρήση του barrier είναι απαραίτητη σε εφαρμογές που απαιτούν συγχρονισμό μεταξύ των διεργασιών.

3.3 Eti Swarm και Intel TBB

Eti Swarm:

Στα πλαίσια της εργασίας αυτής είχαμε μια πρώτη επαφή με το Eti Swarm (SWift Adaptive Runtime Machine) [8] πλαίσιο προγραμματισμού (framework). Το Eti Swarm προσφέρεται για γλώσσα προγραμματισμού C και αναπτύσσεται από την εταιρεία Eti International.

Το Swarm είναι ένα πλήρες πλαίσιο προγραμματισμού που έχει ως σκοπό να παρέχει την μέγιστη απόδοση παρέχοντας ταυτόχρονα και επεκτασιμότητα (scalability). Χρησιμοποιώντας καινοτόμες μεθόδους για δυναμική διαχείριση των πόρων και δρομολόγησης διεργασιών πετυχαίνει την μέγιστη χρησιμοποίηση όλων των συστατικών του συστήματος. Το Swarm λειτουργεί σε πολλές πλατφόρμες όπως x86,servers και clusters, GPUS και στους Xeon Phi της Intel.

Δυστυχώς η έκδοση του Swarm που είχαμε δεν ήταν ολοκληρωμένη και αντιμετωπίσαμε πάρα πολλά προβλήματα ακόμη και στην εγκατάσταση του Swarm στις μηχανές που δουλεύαμε. Το γεγονός αυτό μαζί με το ότι δεν υπήρχε υποστήριξη από την εταιρεία μας ώθησε στην εγκατάλειψη του Swarm. Επίσης, φαίνεται ότι το project αυτό έχει εγκαταλειφθεί από τους δημιουργούς του λόγω του ότι η τελευταία έκδοση που έχουν δημοσιεύσει πάει πίσω στο 2013.

Intel TBB:

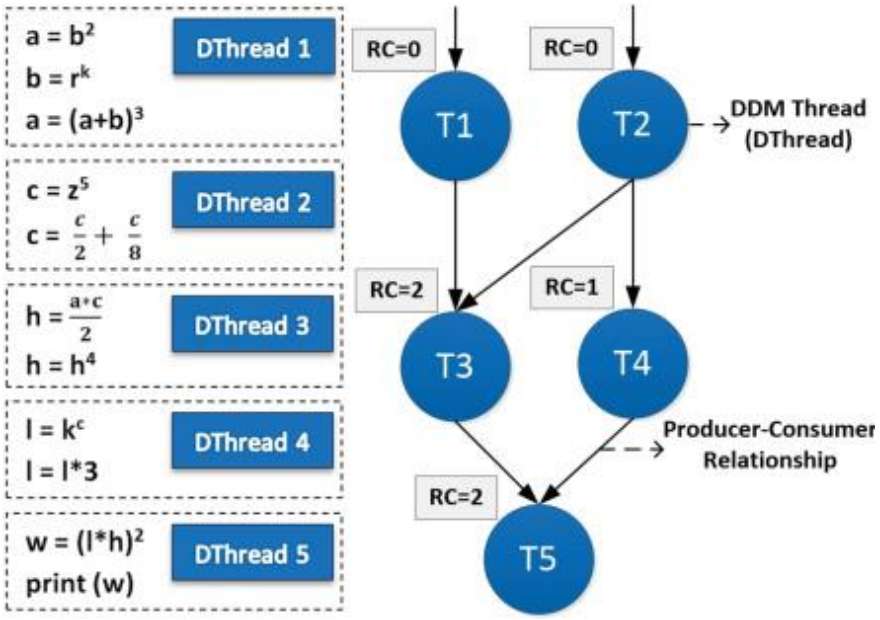
Η Intel TBB [9] είναι μια βιβλιοθήκη της γλώσσας προγραμματισμού C++ που δημιουργήθηκε από την Intel για εγγραφή προγραμμάτων που εκμεταλλεύονται πολυπύρηνους επεξεργαστές. Η βιβλιοθήκη παρέχει στον προγραμματιστή δομές δεδομένων και αλγορίθμους που του επιτρέπουν να αποφεύγει διάφορες πολυπλοκότητες που εμφανίζονται όταν χρησιμοποιούνται άλλα τρόποι multithreading όπως τα Posix Threads. Στις περιπτώσεις που χρησιμοποιούνται Posix νήματα τα νήματα πρέπει να δημιουργούνται, να συγχρονίζονται και να τερματίζονται ρητά από τον προγραμματιστή. Η Intel μέσω της βιβλιοθήκης TBB εκτελεί αυτές τις λειτουργείες δυναμικά χωρίς να χρειάζεται ο προγραμματιστής να ασχολείται με αυτές. Επίσης, το TBB μπορεί να διαχωρίσει τις διάφορες εργασίες σε tasks δημιουργώντας γράφους συγχρονισμού και εξάρτησης μεταξύ τους και οι οποίες εκτελούνται στους επεξεργαστές του συστήματος δυναμικά.

Ένα από τα σημαντικότερα χαρακτηριστικά που έχει το TBB είναι το ονομαζόμενο work stealing. Το work stealing εφαρμόζεται σε περιπτώσεις που ένας επεξεργαστής τελειώνει την εργασία του ενώ οι υπόλοιποι επεξεργαστές έχουν ακόμη αρκετή δουλειά. Τότε το TBB "κλέβει" εργασία από τους άλλους πυρήνες και την δίνει στον πυρήνα που ήταν αδρανές.

3.4 DDM/FREDDO

Το Data Driven Multithreading είναι ένα non-blocking multithreading μοντέλο το οποίο δρομολογεί την εκτέλεση των νημάτων ανάλογα με την διαθεσιμότητα των δεδομένων που χρειάζεται το κάθε νήμα για την εκτέλεση του. Ένα πρόγραμμα DDM είναι βασισμένο σε νήματα από εντολές τα οποία ονομάζονται Dthreads και τα οποία έχουν μεταξύ τους σχέσεις παραγωγού -καταναλωτή. Η καρδιά του μοντέλου DDM είναι το TSU-Thread Scheduling Unit το οποίο είναι υπεύθυνο για την διαχείριση των Dthreads. Συγκεκριμένα ο TSU για κάθε DThread συλλέγει μετά-δεδομένα ή αλλιώς Thread Templates τα οποία χρησιμοποιεί για να διαχειρίζεται και να ορίζει πότε ένα Dthread είναι έτοιμο για εκτέλεση. Για να δρομολογηθεί ένα Dthread για εκτέλεση από τον TSU πρέπει όλοι οι παραγωγοί αυτού του Dthread να έχουν τελειώσει την εκτέλεση τους. Αυτό διασφαλίζει ότι όταν ένα Dthread ξεκινήσει την εκτέλεση του έχει όλα τα δεδομένα που χρειάζεται.

Στο σχήμα 3.4.1 [10] μπορείτε να δείτε ένα παράδειγμα εκτέλεσης ενός DDM προγράμματος. Το RC που αναφέρεται στο σχήμα είναι το Ready Count ενός νήματος που δείχνει πόσους παραγωγούς έχει το νήμα αυτό. Για να δρομολογηθεί για εκτέλεση ένα Dthread πρέπει το Ready Count του να είναι ίσο με μηδέν, γεγονός που θα υποδεικνύει ότι αυτό το Dthread δεν περιμένει κάποιο άλλο Dthread και είναι έτοιμο για εκτέλεση.



Σχήμα 3.4.1 [10]: Παράδειγμα DDM προγράμματος

Στο σχήμα 3.4.1 παρουσιάζεται ένα μοντέλο εκτέλεσης ενός DDM προγράμματος. Το Dthread 3 έχει Ready Count ίσο με δύο που μεταφράζεται ότι το Dthread αυτό περιμένει δύο παραγωγούς. Συγκεκριμένα στο παράδειγμα αυτό το Dthread 3 πρέπει να περιμένει την εκτέλεση του Dthread 1 και του Dthread 2 επειδή τα δύο αυτά Dthreads υπολογίζουν τις μεταβλητές a και b που χρειάζεται το Dthread 3 για τους υπολογισμούς του.

Εμείς στα πλαίσια της διπλωματικής αυτής ασχοληθήκαμε με το FREDDO(an efficient Framework for Runtime Execution of Data-Driven Objects) που είναι μια υλοποίηση DDM μοντέλου για γλώσσα προγραμματισμού C++.

Στο μοντέλο FREDDO ένα Dthread αποτελείται από τα πιο κάτω χαρακτηριστικά:

- **Instruction Frame Pointer (IFP):** Ουσιαστικά είναι ένας pointer που καθορίζει την συνάρτηση που θα εκτελέσει το dthread.
- **Thread ID (TID):** Το μοναδικό ID κάθε dthread
- **Nesting:** Υποδηλώνει το φώλιασμα του Dthread
- **Scheduling Method:** Υποδηλώνει την μέθοδο δρομολόγησης του Dthread στους πυρήνες του συστήματος.
- **Ready Count:** Υποδηλώνει το πόσους παραγωγούς έχει το Dthread. Ελάχιστη τιμή είναι το 1.

<u>Το χαρακτηριστικό Nesting:</u>

Κάθε Dthread έχει το χαρακτηριστικό Nesting που υποδηλώνει τον φώλιασμα αυτού του dthread. Συγκεκριμένα το nesting μπορεί να χρησιμοποιείτε και για την παράλληλη εκτέλεση επαναλήψεων for και υποστηρίζει μέχρι επαναλήψεις τριπλού φωλιάσματος.

Το nesting χαρακτηριστικό χωρίζεται σε τέσσερις κατηγορίες:

- Nesting-Zero: Δηλώνει ότι το dthread αυτό δεν εκτελεί επανάληψη for
- Nesting-One: Δηλώνει ότι το dthread αυτό εκτελεί επανάληψη μονού φωλιάσματος
- Nesting-Two: Δηλώνει ότι το dthread αυτό εκτελεί επανάληψη διπλού φωλιάσματος
- Nesting-Three: Δηλώνει ότι το dthread αυτό εκτελεί επανάληψη τριπλού φωλιάσματος.

Μετά την δημιουργία του Dthread για να ξεκινήσει η πραγματική εκτέλεση του Dthread πρέπει το Ready Count να είναι 0. Για να μειωθεί το ready count ενός dthread πρέπει να καλεστεί η εντολή update. Μέσω της εντολής update μειώνεται η τιμή Ready Count και δημιουργούνται τα instances του dthread.

Ανάλογα με το Nesting χαρακτηριστικό δηλώνονται και από τον προγραμματιστή τα χαρακτηριστικά Context. Το context δηλώνει το αριθμό των δυναμικών εκτελέσεων του Dthread. Για παράδειγμα αν ένα Dthread έχει context ίσο με 10, τότε θα δημιουργηθούν 10 διαφορετικά αντίγραφα αυτού του dthread που θα εκτελέσουν την συνάρτηση του Dthread με μόνη διαφορά το Context χαρακτηριστικό τους.

Το χαρακτηριστικό Context:

Το context χαρακτηριστικό όπως και το nesting χωρίζεται σε κατηγορίες. Οι κατηγορίες αυτές είναι τρεις και είναι οι πιο κάτω:

- Context->Outer: Δηλώνει την επανάληψη του εξωτερικού for
- Context->Middle: Δηλώνει το αριθμό της επανάληψης του μεσαίου for
- Context->Inner: Δηλώνει το αριθμό της επανάληψης του εσωτερικού for

Scheduling Method:

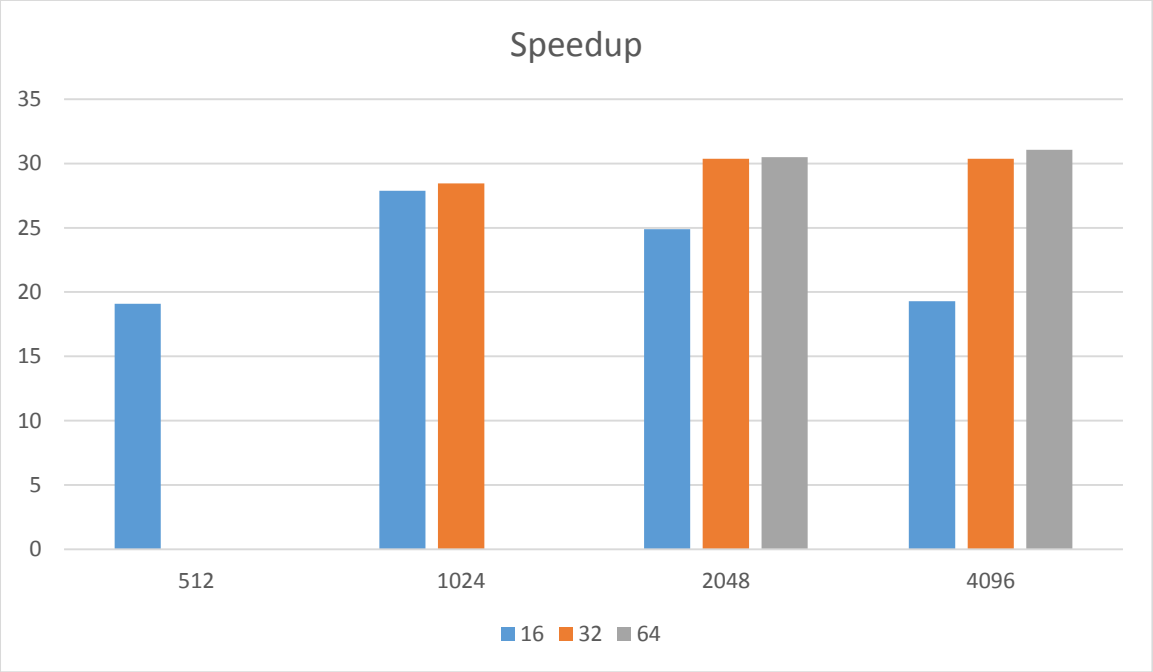
Το Scheduling Method υποδηλώνει πως θα δρομολογούνται τα threads στους πυρήνες. Το FREDDO υποστηρίζει τρεις διαφορετικούς τρόπους:

- Static: Τα Dthread θα τρέχουν στατικά μόνο σε ένα πυρήνα
- Dynamic: Τα Dthread θα τρέχουν δυναμικά σε όλους τους πυρήνες
- Round Robin: Τα Dthread θα τρέξουν με σειρά σε όλους τους πυρήνες

Σήμερα, το DDM μοντέλο βρίσκεται σε διαρκές εξέλιξη και αξιολόγηση στο εργαστήριο DDM Group στο τμήμα πληροφορικής του Πανεπιστημίου Κύπρου. Έχει υλοποιηθεί μέχρι στιγμής σε τρία projects. Το πρώτο, D2Now αφορούσε την υλοποίηση του DDM σε ένα δίκτυο από workstations. Η δεύτερη υλοποίηση, TFlux [4] έδωσε έμφασή στην φορητότητα (portability). Το TFlux είναι μια φορητή (portable) πλατφόρμα λογισμικού που έχει ως στόχο να επιτρέψει στους χρήστες της να αναπτύξουν και να εκτελέσουν DDM εφαρμογές με εύκολο τρόπο σε συστήματα με οποιαδήποτε αρχιτεκτονική. Έχει την ικανότητα να εικονοποιήσει (virtualize) τα χαρακτηριστικά του συστήματος στο οποίο εκτελείτε. Με την εικονοποίηση αυτή το TFlux μπορεί να παρέχει ένα κοινό προγραμματιστικό μοντέλο ανεξαρτήτως αρχιτεκτονικής. Η τρίτη υλοποίηση, DDM-VM [4] είναι μια εικονική μηχανή που υποστηρίζει την εκτέλεση DDM σε ομογενή (homogeneous) και ετερογενή (heterogeneous) συστήματα πολλαπλών πυρήνων. Η DDM-VM εικονοποιεί τους πόρους της μηχανής και χρησιμοποιεί μια γενική, ενιαία αναπαράσταση για τα DDM προγράμματα.

Σήμερα αναπτύσσεται το προγραμματιστικό πλαίσιο FREDDO από τον Διδακτορικό ερευνητή Γιώργο Ματθαίου.

Πιο κάτω παρουσιάζονται αποτελέσματα του FREDDO πλαισίου προγραμματισμού για C++ στο πρόβλημα του Blocked Matrix Multiplication(σχήμα 3.4.3)



Σχήμα 3.4.3: Επιτάχυνση στο BMATMULT ανάλογα μέγεθος πίνακα X μέγεθος block

Η εφαρμογή Blocked Matrix Multiplication αφορά τον υπολογισμό του αποτελέσματος μεταξύ δύο πινάκων και αποθήκευση του αποτελέσματος σε ένα τρίτο πίνακα : A x B=C. Στο πρόβλημα αυτό το FREDDO εκμεταλλεύεται πλήρως τα χαρακτηριστικά nesting που παρέχει επιτυγχάνει μέχρι και επιτάχυνση 31 φορές σε σύγκριση με την σειριακή εκτέλεση. Η επιτάχυνση αυτή είναι η θεωρητική μέγιστη επιτάχυνση αφού τα πειράματα έγιναν σε μηχανή με 32 πυρήνες και λαμβάνοντας υπόψη ότι ο ένας πυρήνας είναι δεσμευμένος από το TSU τότε η μέγιστη επιτάχυνση είναι 31.

Μερικά από τα πλεονεκτήματα του FREDDO σε σχέση με άλλες υλοποιήσεις DDM είναι ότι είναι η πρώτη υλοποίηση που υποστηρίζει πολλαπλά αρχεία(multifile support) κώδικα. Επίσης, το δεύτερο σημαντικό πλεονέκτημα είναι ότι υποστηρίζει run time δέσμευση των πυρήνων. Σε παλαιότερες εκδόσεις για να τρέξει ένα DDM πρόγραμμα με διαφορετικό αριθμό πυρήνων έπρεπε να ξαναγίνεται compile.

Κεφάλαιο 4

Το HPCG Benchmark

4.1 Περιγραφή του HPCG Benchmark

Το HPCG (High Performance Conjugate Gradients) Benchmark είναι ένα project που έχει στόχο να αντικαταστήσει το High Performance LINPACK (HPL) benchmark που χρησιμοποιείτε σήμερα στην κατάταξη των TOP500 υπολογιστών στον κόσμο.
Το project TOP500[1] κατατάσσει τα 500 πιο αποδοτικά συστήματα στον κόσμο. Ξεκίνησε το 1993 και δημοσιεύει τη λίστα με τα καλύτερα συστήματα δύο φορές το χρόνο. Η απόδοση των συστημάτων μετράτε σε floating point operations ανά δευτερόλεπτο. Στο σχήμα 4.4.1 [10] παρουσιάζεται η πιο πρόσφατη κατάταξη του TOP500 όπως παρουσιάστηκε το Νοέμβριο του 2014.

Top 10 positions of the 44th TOP500 on November, 2014

Rank	Rmax Rpeak (PFLOPS)	Name	Computer design Processor type, interconnect	Vendor	Site Country, year	Operating system
1	33.863 54.902	*Tianhe-2*	**NUDT** Xeon E5–2692 + Xeon Phi 31S1P, TH Express-2	NUDT	National Supercomputing Center in Guangzhou China, 2013	Linux (Kylin)
2	17.590 27.113	*Titan*	**Cray XK7** Opteron 6274 + Tesla K20X, Cray Gemini Interconnect	Cray Inc.	Oak Ridge National Laboratory United States, 2012	Linux (CLE, SLES based)
3	17.173 20.133	*Sequoia*	**Blue Gene/Q** PowerPC A2, Custom	IBM	Lawrence Livermore National Laboratory United States, 2013	Linux (RHEL and CNK)
4	10.510 11.280	*K computer*	**RIKEN** SPARC64 VIIIfx, Tofu	Fujitsu	RIKEN Japan, 2011	Linux
5	8.586 10.066	*Mira*	**Blue Gene/Q** PowerPC A2, Custom	IBM	Argonne National Laboratory United States, 2013	Linux (RHEL and CNK)
6	6.271 7.779	*Piz Daint*	**Cray XC30** Xeon E5–2670 + Tesla K20X, Aries	Cray Inc.	Swiss National Supercomputing Centre Switzerland, 2013	Linux (CLE)
7	5.168 8.520	*Stampede*	**PowerEdge C8220** Xeon E5–2680 + Xeon Phi, Infiniband	Dell	Texas Advanced Computing Center United States, 2013	Linux (CentOS)[11]
8	5.008 5.872	*JUQUEEN*	**Blue Gene/Q** PowerPC A2, Custom	IBM	Forschungszentrum Jülich Germany, 2013	Linux (RHEL and CNK)
9	4.293 5.033	*Vulcan*	**Blue Gene/Q** PowerPC A2, Custom	IBM	Lawrence Livermore National Laboratory United States, 2013	Linux (RHEL and CNK)
10	3.577 6.132		**Cray CS** Xeon E5-2660v2 10C and Nvidia K40, Infiniband	Cray Inc.	United States, 2014	Linux

Σχήμα 4.1.1: Κατάταξη TOP500 το Νοέμβριο του 2014

Δημιουργοί του HPCG benchmark είναι ο Jack Dongarra του Πανεπιστημίου της Τενεσί και ο Micahel A.Heroyx των εργαστηρίων Sandia. Το κίνητρο αυτού του project είναι να δημιουργηθεί ένα πιο σχετικό και αξιόπιστο benchmark από το HPL και φιλοδοξεί ότι θα αντικαταστήσει στο προσεχές μέλλον το HPL.

Γιατί το HPL έχασε την σχετικότητα του:

Το HPL που χρησιμοποιείται μέχρι και σήμερα φαίνεται ότι έχει χάσει την σχετικότητα του με τις πραγματικές σημερινές ανάγκες στο κόσμο του High Performance Computing και σύντομα θα αντικατασταθεί από ένα νέο benchmark που θα είναι πιο σχετικό στις σημερινές ανάγκες.
Το HPL παρουσιάστηκε για να χρησιμοποιείται από τον κόσμο του High Performance Computing το 1993 και είχε ως σκοπό να ανιχνεύει τα γρηγορότερα συστήματα. Σήμερα 2015, 12 χρόνια μετά οι υπολογιστικές ανάγκες δεν μοιάζουν σχεδόν καθόλου με εκείνες του 1993. Λόγω τις αύξησης των δεδομένων το HPL έχει χάσει την σχετικότητα του σε σχέση με τις σημερινές εφαρμογές. Αυτό έχει ως αποτέλεσμα πολλά υπολογιστικά συστήματα να επιτυγχάνουν εξαιρετικές αποδόσεις στο HPL αλλά να αποτυγχάνουν σε άλλες εφαρμογές.

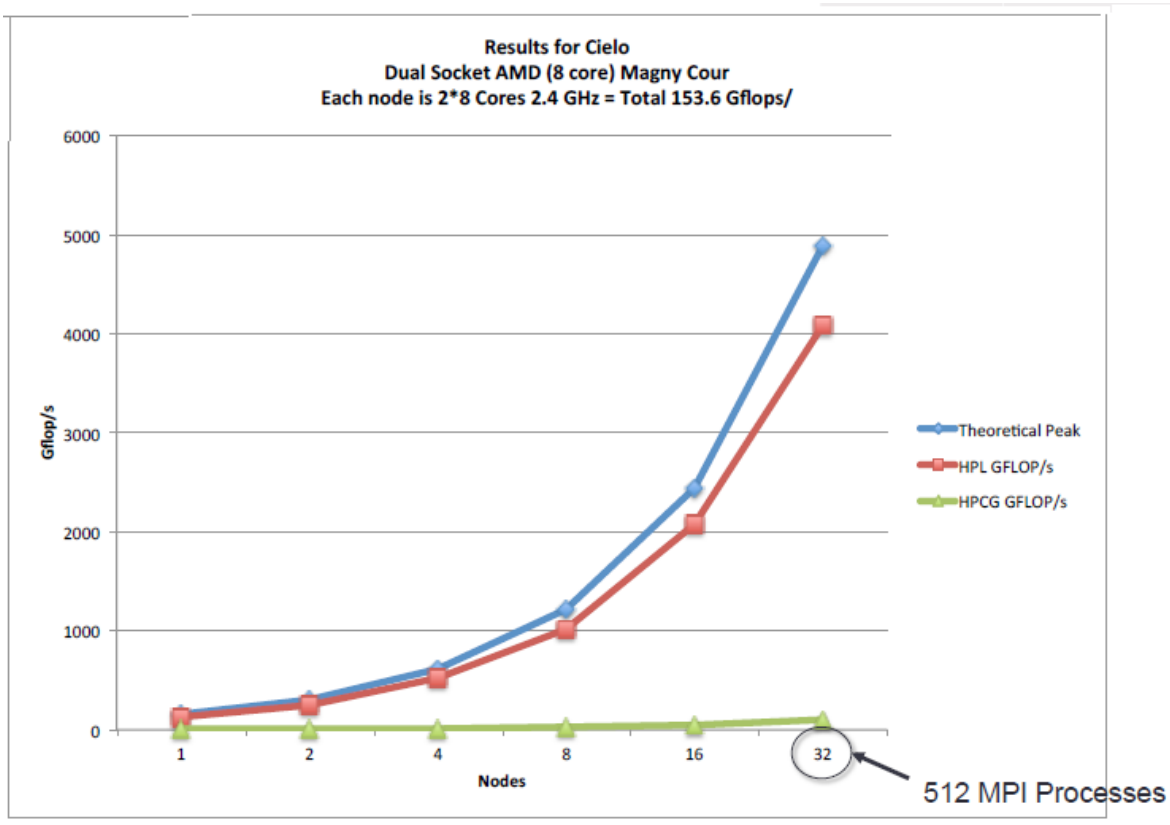
Η πρώτη έκδοση του Linpack που παρουσιάστηκε έλυνε τον πολλαπλασιασμό ενός πυκνού (dense) πίνακα που αποτελούνταν από 100 X 100 γραμμικές εξισώσεις. Στην συνέχεια όταν οι μηχανές άρχισαν να έχουν μεγαλύτερες δυνατότητες προχώρησαν σε πίνακα 1000 X 1000 εξισώσεων ενώ στην τελευταία έκδοση αφαίρεσαν το προκαθορισμένο μέγεθος πίνακα και έδωσαν την ευκαιρία να μπορεί το benchmark να τρέχει σε μοντέρνα συστήματα τα οποία επωφελούνταν το πλεονέκτημα των σύγχρονων δικτύων επικοινωνίας (modern interconnection networks).

Οι τύποι των υπολογισμών και τα μοτίβα πρόσβασης δεδομένων στο HPL είναι αυτά που ο Dongarra και ο Heroux αναφέρονται σαν Type 1. Στο Type 1 τα συστήματα φορτώνεται από πολλούς floating point υπολογισμούς και τα δεδομένα μπορούν να οργανωθούν με τέτοιο τρόπο που τα σωστά δεδομένων θα είναι στον σωστό Floating Point unit την σωστή χρονική στιγμή. Στα μοτίβα Type 2 τα μοτίβα πρόσβασης δεδομένων είναι λιγότερο κοινότυπα και συνεπώς δεν μπορούν να οργανώνονται από πριν με τρόπο όπως γίνεται στα Type 1.

Ουσιαστικά όπως λένε οι Heroux και Dongara αν σχεδιαστεί ένα σύστημα που θα έχει εξαιρετική απόδοση στο Linpack τότε το πιο πιθανό είναι αυτό το σύστημα να μην είναι καθόλου αποδοτικό σε εφαρμογές πραγματικού κόσμου καταρρίπτοντας έτσι τις βασικές αρχές του οργανισμού Top500 που είναι να καθορίζει ποιο πραγματικά είναι το πιο αποδοτικό σύστημα.

Οι Heroux και Dongara παρέχουν και το πιο κάτω παράδειγμα στις δημοσιεύσεις τους [3]:

“ Το σύστημα Titan του Oak Ridge National Laboratory έχει 18,688 υπολογιστικούς κόμβους, στους οποίους κάθε κόμβος έχει 16 πύρινους AMD Opteron επεξεργαστές, 32Gb κύρια μνήμη και μια 6GB Nvidia K20 GPU. Το Titan ήταν το πρώτο στην κατάταξη τον Νοέμβρη του 2012 χρησιμοποιώντας το HPL benchmark. Ωστόσο, για να επιτευχθεί αυτό η επιτυχία οι Opteron επεξεργαστές είχαν μόνο υποστηρικτικό ρόλο στο αποτέλεσμα. Όλοι οι υπολογισμοί και όλα τα δεδομένα κατευθύνονταν και εκτελούνταν από το Nvida GPU. Ο τρόπος αυτός είναι τελείως αντίθετος από τις πραγματικές εφαρμογές γιατί οι περισσότεροι υπολογισμοί γίνονται στα CPU και όχι στα GPU. “



Σχήμα 4.1.2: Απόδοση του υπολογιστή Cielo στο HPL και στο HPCG

Στο σχήμα 4.1.2[11] παρουσιάζεται ένα παράδειγμα υπολογιστή που επιτύγχανε εξαιρετικές αποδόσεις με το HPL αλλά είχε εντελώς διαφορετικά αποτελέσματα με το HPCG.

4.2 Ανάλυση του HPCG Benchmark

Το HPCG είναι ένας ολοκληρωμένος stand-alone κώδικας που υπολογίζει την απόδοση ενός υπολογιστικού συστήματος χρησιμοποιώντας μεταξύ άλλων τις πιο κάτω λειτουργίες:

- Πολλαπλασιασμός αραιού πίνακα-vector (Sparse matrix-vector multiplication) (Συνάρτηση ComputeSPMV)
- Sparse triangular Solve με την μέθοδο Gauss-Seidel (Συνάρτηση ComputeSYMGS)
- Ενημερώσεις διανυσμάτων (Vector updates) (Συνάρτηση ComputeWAXBPY)
- Global Dot products (Συνάρτηση ComputeDotProduct)

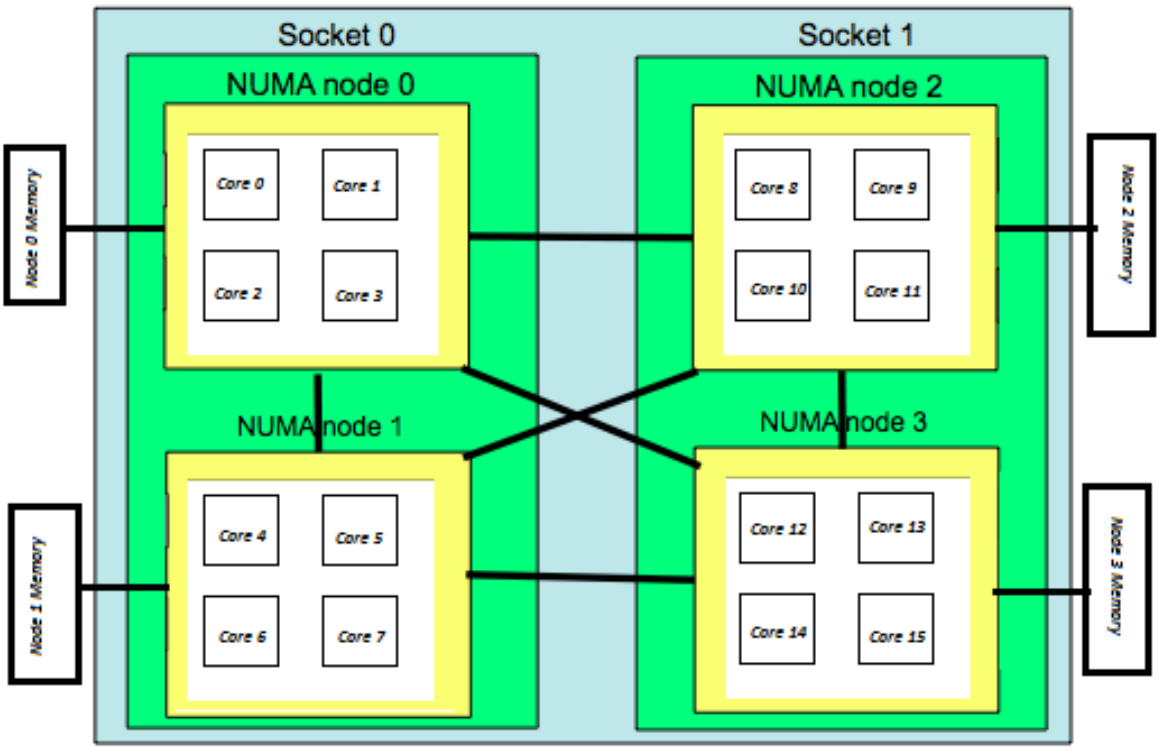
Το HPCG υπολογίζει εξισώσεις του τύπου Ax=b όπου το A είναι ο πίνακας με τις τιμές ο οποίος είναι αραιός, το b είναι γνωστό και το x υπολογίζεται.

Το benchmark είναι γραμμένο σε γλώσσα προγραμματισμού C++ και μπορεί ο καθένας να το προμηθευθεί δωρεάν από την επίσημη ιστοσελίδα. Επιτρέπονται οι οποιεσδήποτε αλλαγές στον κώδικα του benchmark αλλά στο τέλος κάθε εκτέλεσης εκτελείται ένα στάδιο επιβεβαίωσης(validation) των αποτελεσμάτων και επιστρέφεται ένα αρχείο στο οποίο αναφέρεται αν τα αποτελέσματα είναι έγκυρα.

Το HPCG μπορεί να εκτελεστεί με τέσσερις διαφορετικούς τρόπους. Ο πρώτος τρόπος είναι η σειριακή εκτέλεση στην οποία ο αλγόριθμος Conjugate Gradient εκτελείται απολύτως σειριακά. Ο δεύτερος τρόπος είναι εκτέλεση με OpenMP η οποία χρησιμοποιείται σε κομμάτια κώδικα που μπορούν να εκτελεστούν παράλληλα(όπως επαναλήψεις for,reductions). Ο τρίτος και πιο αποδοτικός τρόπος όπως φάνηκε από την μελέτη του benchmark είναι η εκτέλεση του αλγορίθμου με MPI.Στην εκτέλεση MPI το benchmark επωφελείται από τις λειτουργίες επικοινωνίας που παρέχεται από την MPI και εκτελείτε ο αλγόριθμος Preconditioned Conjugate Gradient ο οποίος είναι κατά πολύ ταχύτερος από τον απλό Conjugate Gradient. Τέλος, το HPCG μπορεί να εκτελεστεί και Υβριδικά χρησιμοποιώντας συνδυασμό MPI με OpenMP.

4.3 Υλικό που χρησιμοποιήθηκε

Στα πλαίσια της εργασίας αυτής μελετήσαμε το HPCG Benchmark χρησιμοποιώντας τέσσερις μηχανές που δόθηκαν. Οι μηχανές αυτές διαθέτουν 32 πυρήνες AMD οι κάθε μια και τρέχουν λειτουργικό σύστημα Ubuntu 14,04 ή Ubuntu 12.04. Κάθε μηχανή διαθέτει δύο sockets με το κάθε socket να έχει οκτώ φυσικούς πυρήνες. Κάθε φυσικός πυρήνας χωρίζεται σε δύο νήματα και κάθε πυρήνας είναι χρονισμένος στα 1,4 GHz. Επίσης η κύρια μνήμη των υπολογιστών είναι 48GB. Η μηχανή είναι χωρισμένη σε τέσσερα Non Uniform Memory Access κόμβους. Στο σχήμα 4.3.1 παρουσιάζεται γραφική αναπαράσταση της μηχανής.



Σχήμα 4.3.1: Αναπαράσταση μηχανής CS8472

4.4 Αποτελέσματα HPCG Benchmark

Στα πλαίσια αξιολόγησης του HPCG Benchmark χρησιμοποιήσαμε και τους τέσσερις τρόπους που είναι διαθέσιμοι για εκτέλεση. Συγκεκριμένα, τρέξαμε το benchmark με το σειριακό τρόπο, με OpenMP, MPI και με υβριδική εκτέλεση MPI με συνδυασμό OpenMP.

Επιπρόσθετα, χρησιμοποιώντας το εργαλείο gprof κάναμε profiling στην εφαρμογή του HPCG για να διαπιστώσουμε ποιες συναρτήσεις έχουν τον μεγαλύτερο χρόνο εκτέλεσης. Τα αποτελέσματα του profiling παρουσιάζονται στον πίνακα 4.4.1.

Percentage of Execution Time	Function
61.91%	ComputeSYMGS
31.53%	ComputeSPMV
6.54%	Other

Πίνακας 4.4.1: Αποτελέσματα του profiling με gprof

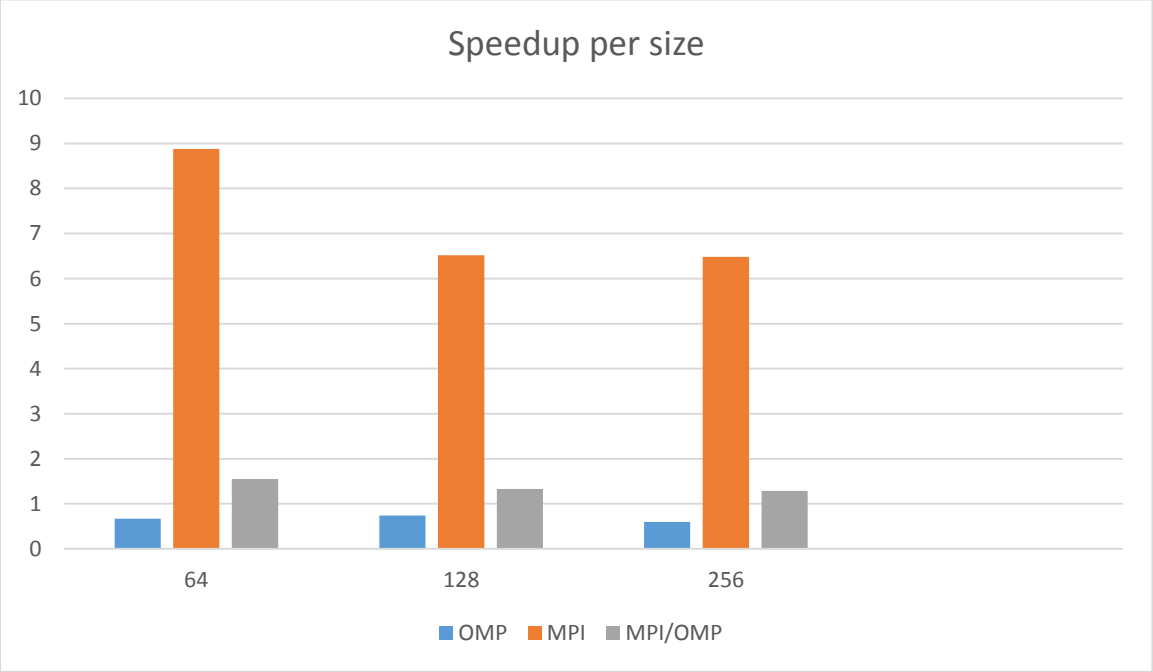
Η συνάρτηση ComputeSYMGS που ο χρόνος εκτέλεσης της είναι μεταξύ 60% και 65% είναι εντελώς σειριακή και δυστυχώς δεν μπορεί να εκτελεστεί παράλληλα. Συγκεκριμένα η συνάρτηση αποτελείται από επαναλήψεις for στις οποίες οι επαναλήψεις τους έχουν εξαρτήσεις μεταξύ τους. Το γεγονός ότι η ComputeSYMGS δεν μπορούσε να εκτελεστεί παράλληλα επηρέασε αρκετά το speedup στις εκτελέσεις OpenMP που τρέξαμε και στις εκτελέσεις DDM-FREDDO που θα παρουσιαστούν αργότερα.

Αρχικά χρησιμοποιήσαμε την 32 πύρινη μηχανή CS8472 και οι εκτελέσεις χωρίζονται όπως πιο κάτω:

- Σειριακή εκτέλεση
- Εκτέλεση OpenMP με 32 νήματα
- Εκτέλεση MPI με 32 διεργασίες
- Εκτέλεση MPI/OpenMP με 2 διεργασίες και 16 νήματα στην κάθε διεργασία

Τα μεγέθη του προβλήματος που τρέξαμε ήταν 64x64x64,128x128x128 και 256x256x256. Δεν κατέστη δυνατό να τρέξουμε το πρόβλημα με μεγαλύτερα μεγέθη γιατί το μέγεθος του προβλήματος που παραγόταν ήταν μεγαλύτερο από την κύρια μνήμη της μηχανής και δεν ήταν δυνατή η δημιουργία των δεδομένων του προβλήματος.

Αποτελέσματα:



Γραφική Παράσταση 4.4.2: Επιτάχυνση HPCG ανά μέγεθος και ανά προγραμματιστικό μοντέλο

Όπως είναι εμφανές από την γραφική παράσταση η καλύτερη λύση για το Benchmark HPCG είναι η εκτέλεση με MPI. Όπως αναφέρεται και σε διάφορες δημοσιεύσεις που αναλύουν το μοντέλο εκτέλεσης του HPCG Benchmark η μείωση της απόδοσης που παρατηρείται στην εκτέλεση με OpenMP οφείλεται στους πιο κάτω λόγους:

α) Το HPCG είναι well balanced για εκτελέσεις MPI γιατί εκτελείται ο αλγόριθμος preconditioned Conjugate and Gradient. Το benchmark εκμεταλλεύεται τις λειτουργίες επικοινωνίας που έχει το MPI μεταξύ των διεργασιών(send,receive) και μέσω αυτών μπορεί να εκτελείται ο preconditioned αλγόριθμος που είναι κατά πολύ ταχύτερος από τον κανονικό αλγόριθμο. Απλά για να γίνει κατανοητή η μεγάλη διαφορά μεταξύ του preconditioned αλγορίθμου και του κανονικού στο πίνακα 4.4.1 μπορείτε να δείτε των αριθμό των εντολών που χρειάζονται στην MPI εκτέλεση σε σύγκριση με τις εντολές που χρειάζονται στην OpenMP εκτέλεση.

	Instructions	**Cache References**	**Cache Misses**	**Cache Miss Rate**
MPI	2,105,590,570	983,907,784	9,964,653	1.013 %
OPENMP	128,164,601,249	39,888,821,445	47,043,363	0.118 %

Πίνακας 4.4.1: Σύγκριση εντολών εκτέλεσης MPI και OpenMP

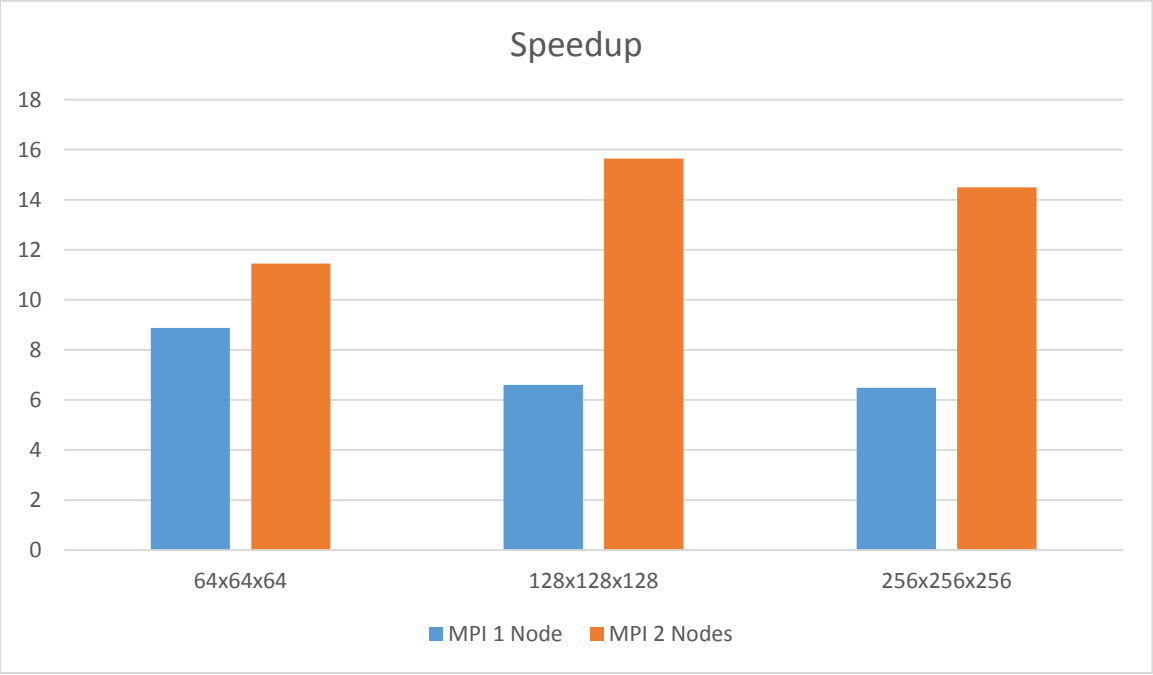
β) Η συνάρτηση ComputeSYMGS που παίρνει περίπου του 60% με 65% του χρόνου δεν μπορεί να εκτελεστεί παράλληλα λόγω των εξαρτήσεων που έχει η κάθε επανάληψη στο for loop με προηγούμενες επαναλήψεις. Συνεπώς οποιαδήποτε επιτάχυνση επιτυγχάνεται από άλλες παράλληλες συναρτήσεις επικαλύπτεται από τον χρόνο εκτέλεσης της ComputeSYMGS που παραμένει σειριακή.

Για την περαιτέρω αξιολόγηση της απόδοσης του HPCG Benchmark χρησιμοποιήσαμε μόνο την εκτέλεση MPI. Συγκεκριμένα εγκαταστήσαμε σε δύο μηχανές, την CS8471 και την CS8472 την βιβλιοθήκη MPI. Μέσω της βιβλιοθήκης MPI παρέχεται η δυνατότητα εκτέλεσης οποιουδήποτε προγράμματος σε περισσότερες από μια μηχανές. Απαραίτητες προϋποθέσεις για να είναι εφικτή αυτού του είδους εκτέλεση είναι το πρόγραμμα να είναι ακριβώς το ίδιο και στις δύο μηχανές και στο ίδιο directory και οι μηχανές να έχουν κάποιο δίκτυο επικοινωνίας (interconnection network). Επίσης, η έκδοση MPI που τρέχει στις μηχανές πρέπει να είναι η ίδια.

Τα τεχνικά χαρακτηριστικά της MPI εκτέλεσης ήταν τα πιο κάτω:

- Δύο nodes τα οποία διαθέτουν 32 cores το καθένα
- 64 process που διαμοιράζονται ίσα στους δύο κόμβους
- Τρία διαφορετικά μεγέθη προβλήματος(64x64x64,128x128x128,256x256x256).

Στο σχήμα (4.4.3) που ακολουθεί παρουσιάζονται τα αποτελέσματα της εκτέλεσης του HPCG Benchmark με MPI σε δύο υπολογιστικούς κόμβους.



Σχήμα 4.4.3: Σύγκριση επιτάχυνσης εκτέλεσης MP1 σε ένα και δύο κόμβους

Όπως ήταν αναμενόμενο η εκτέλεση σε δύο κόμβους είναι αρκετά ταχύτερη από την εκτέλεση σε ένα κόμβο. Η επιτάχυνση που επιτυγχάνεται εκτός από το μέγεθος 64x64x64 είναι μεγαλύτερη από την διπλάσια από την εκτέλεση σε ένα κόμβο. Αυτό οφείλεται γιατί όπως αναφέρθηκε και πιο πριν το HPCG Benchmark είναι καλά δομημένο για MPI και λόγω του preconditioned αλγόριθμου που χρησιμοποιείται. Όσο περισσότερες είναι οι διεργασίες τόσο πιο γρήγορος είναι ο preconditioned αλγόριθμος.

Κεφάλαιο 5

HPCG και DDM-FREDDO

5.1 Εισαγωγή του FREDDO στο HPCG

Στα πλαίσια της εργασίας θέλαμε να αξιολογήσουμε το FREDDO μοντέλο που βρίσκεται σε εξέλιξη μέσω του HPCG Benchmark. Συγκεκριμένα θέλαμε να δούμε αν το μοντέλο αυτό μπορεί να ανταπεξέλθει σε ένα πρόβλημα τόσο γνωστό και απαιτητικό σε θέματα επιτάχυνσης και σε θέματα ορθότητας. Η υλοποίηση του FREDDO μοντέλου χωρίστηκε σε δύο κατηγορίες:

- Υλοποίηση με μοντέλο παρόμοιο με αυτό που χρησιμοποιεί το OpenMP.Δηλαδή η δημιουργία νημάτων γινόταν κάθε φορά που ήταν απαραίτητα και με την ολοκλήρωση του παράλληλου κώδικα αποδεσμεύονταν.
- Υλοποίηση με Data Driven εξωτερικό μοντέλο. Δηλαδή κάθε thread είχε παραγωγούς και καταναλωτές και η εκτέλεση του ξεκινούσε μόνο αν ολοκλήρωναν προηγουμένως την εκτέλεση τους οι παραγωγοί του. Η ιδέα ήταν κάθε thread να εκτελούσε και μια συνάρτηση και να ξεκινούσε μόλις τα δεδομένα τις συνάρτησης ήταν έτοιμα.

Η υλοποίηση εστιάστηκε κυρίως στις συναρτήσεις ComputeSPMV, ComputeWAXPBY και ComputeDotProduct του benchmark που είναι οι κύριες συναρτήσεις υπολογισμών και είναι οι συναρτήσεις που λαμβάνονται υπόψη στο τελικό αποτέλεσμα. Δεν κατέστη δυνατό να γίνει παράλληλη η συνάρτηση ComputeSYMGS που έχει το μεγαλύτερο κόστος στο χρόνο εκτέλεσης λόγω των εξαρτήσεων που υπήρχαν στην συνάρτηση αυτή.

Δυστυχώς το Data Driven μοντέλο όπως αποδείχθηκε δεν μπορούσε να επιτύχει οποιαδήποτε επιτάχυνση στο benchmark γιατί η δομή του benchmark ήταν τέτοια που δεν μπορούσαν περισσότερες από μια συναρτήσεις να εκτελούνται παράλληλα λόγω των εξαρτήσεων στα δεδομένα που είχαν όλες οι εξαρτήσεις μεταξύ τους. Δηλαδή, δεν υπήρχαν συναρτήσεις που

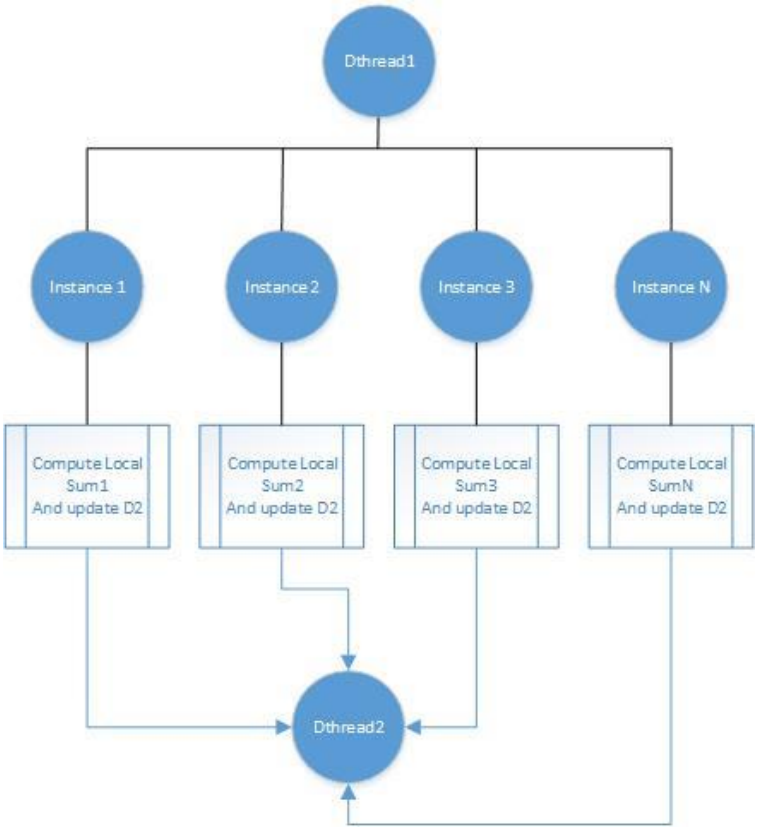
εκτελούσαν υπολογισμούς σε δεδομένα που δεν υπολογίζονταν προηγουμένως από κάποια άλλη συνάρτηση. Επιπρόσθετα, πολλές λειτουργίες που εκτελούνταν στο benchmark δεν μπορούσαν να υλοποιηθούν λόγο της φύσης του FREDDO. Οι λειτουργίες αυτές αφορούσαν κυρίως αναδρομικές συναρτήσεις.

Πρώτο Μοντέλο:

Στο πρώτο μοντέλο εισήγαμε το FREDDO σε κάθε συνάρτηση που υπήρχε εκτέλεση OpenMP. Οι συναρτήσεις αυτές αποτελούνταν κυρίως από επαναλήψεις for. Επίσης, στην συνάρτηση ComputeDotProduct υλοποιήθηκε και ένα μοντέλο reduction για να υπολογίζεται το άθροισμα τιμών μεταξύ πινάκων.

α)Reduction με Data Driven multithreading:

Στο σχήμα 5.1.1 παρουσιάζεται πως υλοποιήθηκε η λειτουργεία reduction με DDM μοντέλο. Για την υλοποίηση χρειάζονται δύο Dthreads τα οποία έχουν εξαρτήσεις μεταξύ τους. Συγκεκριμένα το πρώτο Dthread είναι αυτό που θα υπολογίζει το άθροισμα των τιμών και το δεύτερο είναι το Dthread περιμένει να ολοκληρωθεί το πρώτο Dthread για να αθροίσει τα αποτελέσματα.



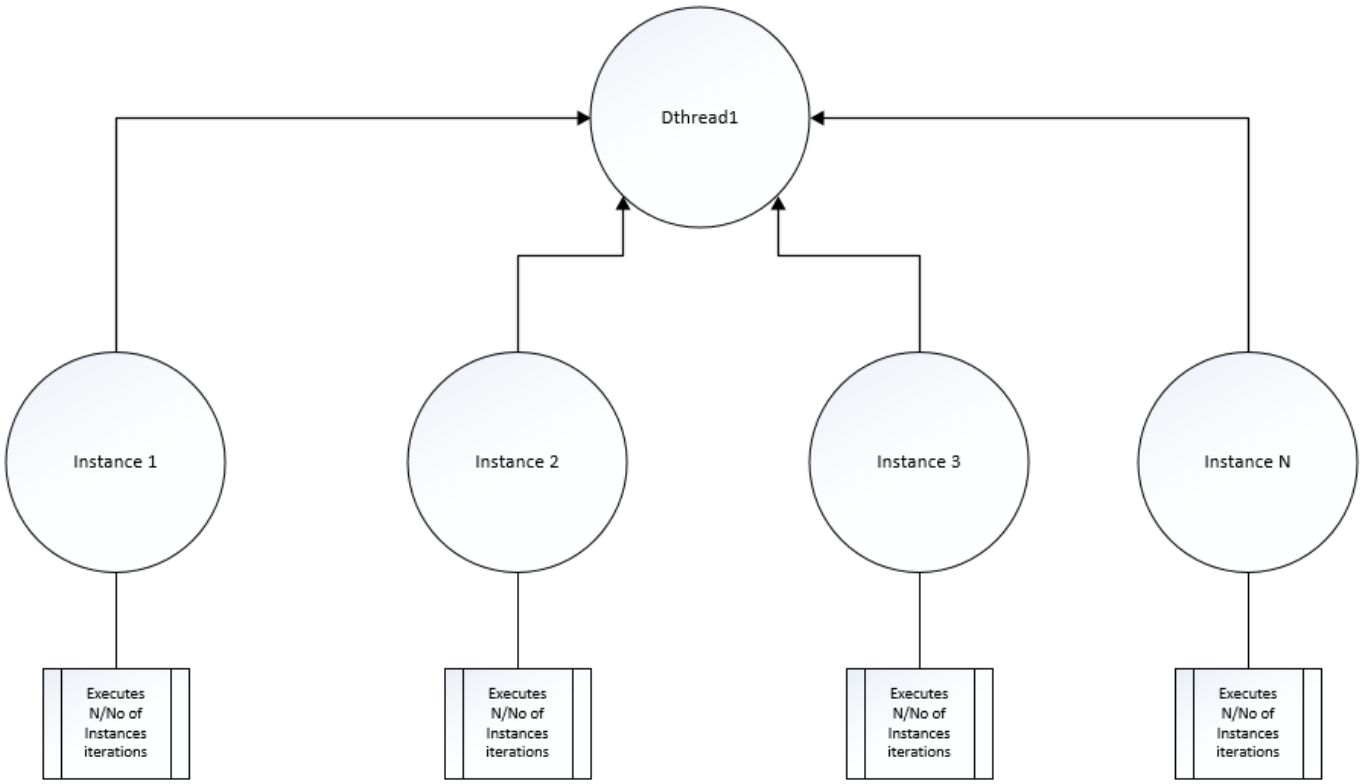
Σχήμα 5.1.1: Υλοποίηση Reduction με μοντέλο DDM

Ο αλγόριθμος εκτελείται ως ακολούθως:

- Δημιουργία πίνακα δεδομένων που θα διαμοιραστεί στα instances του Dthread1
- Δημιουργία Dthread1 με αριθμό instances=αριθμό των πυρήνων και δεδομένα τον πίνακα δεδομένων που δημιουργήθηκε στο προηγούμενο βήμα
- Δημιουργία Dthread2 με ένα instance και ready count=με αριθμό instances του Dthread1
- Κάθε instance του Dthread1 υπολογίζει το δικό του άθροισμα και κάνει update το Dthread2
- Με την ολοκλήρωση όλων των instances του Dthread1 το Dthread2 έχει ready count=0 και συνεπώς είναι έτοιμο για εκτέλεση
- Εκτελείτε το Dthread2 και αθροίζει τα αθροίσματα όλων των instances του Dthread1 μεταξύ τους
- Το τελικό αποτέλεσμα του Dthread2 είναι και το αποτέλεσμα του reduction

β) Επαναλήψεις For με FREDDO:

Η λογική εκτέλεσης επαναλήψεων For με το FREDDO ήταν αρκετά απλή. Συγκεκριμένα δηλωνόταν ένα Dthread με instances ίσα με των αριθμό των πυρήνων που δεσμεύτηκε το FREDDO. Ο λόγος που τα instances ήταν ίσα με των αριθμό των πυρήνων και όχι με των αριθμό των επαναλήψεων ήταν το ότι οι επαναλήψεις ήταν μεγάλες σε αριθμό και δεν θα ήταν αποδοτικό αν είχαμε ένα instance για κάθε επανάληψη. Όσο γρήγορα και να γίνεται το map από τον TSU στους πυρήνες όταν τα instances είναι για παράδειγμα 200,000 δεν θα είναι αποδοτική αυτή η εκτέλεση. Έτσι, αντί αυτής της εκτέλεσης χρησιμοποιήσαμε instances όσα ήταν οι πυρήνες του TSU και κάθε instance εκτελούσε αριθμό επαναλήψεων ίσο με το συνολικό αριθμό επαναλήψεων/συνολικό αριθμό instances. Στο σχήμα 5.5.2 παρουσιάζεται διαγραμματικά το πώς ένα for γίνεται παράλληλο με την χρήση ενός Dthread



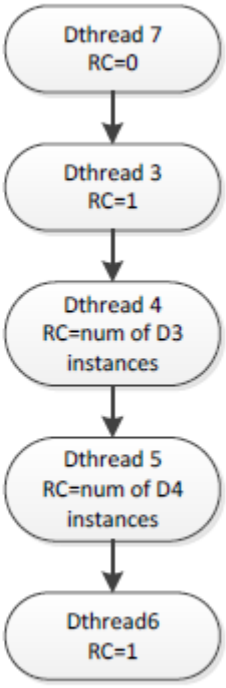
Σχήμα 5.1.2: Οργάνωση των Instances ενός Dthread για επανάληψη for

Δεύτερο Μοντέλο:

Το μοντέλο αυτό υλοποιήθηκε κυρίως για εκπαιδευτικούς σκοπούς και δεν αναμενόταν να επιφέρει την οποιαδήποτε επιτάχυνση. Ο λόγος ήταν αυτός που εξηγήθηκε και προηγουμένως. Όλες οι συναρτήσεις είχαν εξαρτήσεις δεδομένων μεταξύ τους και δεν μπορούσε καμιά να εκτελείται ταυτόχρονα με κάποια άλλη.

Το Data Driven μοντέλο μοιράστηκε σε τέσσερις φάσεις. Η πρώτη φάση ήταν απλή κλήση δύο συναρτήσεων ενώ οι φάσεις δύο, τρία και τέσσερα καλούν την ίδια συνάρτηση. Συγκεκριμένα είναι η συνάρτηση CG που βασικά είναι η κύρια συνάρτηση που τρέχει τον αλγόριθμο Conjugate and Gradient.

Για να δημιουργηθεί μια Data Driven εκτέλεση πρέπει πρώτα να δημιουργηθεί ο γράφος εξαρτήσεων (Dependency Graph). Ο γράφος εξαρτήσεων των φάσεων δυο, τρία και τέσσερα που εφαρμόσαμε στο HPCG παρουσιάζεται στο σχήμα 5.1.3.

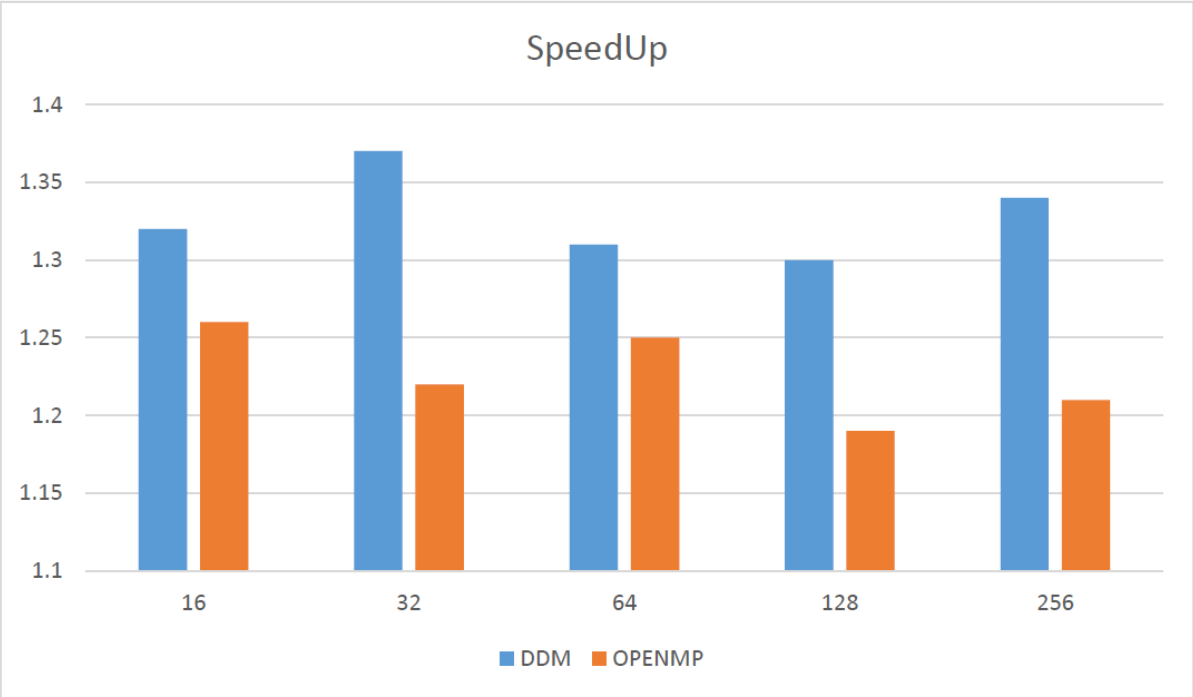


Σχήμα 5.1.3: Γράφος εξαρτήσεων για τις φάσεις δύο, τρία και τέσσερα στο HPCG

Σε κάθε Dthread είχε ανατεθεί μια συνάρτηση η οποία έκανε update την επόμενη συνάρτηση που έπρεπε να εκτελεστεί. Για παράδειγμα στο Dthread 7 είχε ανατεθεί η συνάρτηση CG που εκτελούσε τον αλγόριθμο Conjugate and Gradient. Στο εσωτερικό της συνάρτησης αυτής γινόταν update το Dthread 3 που εκτελούσε την συνάρτηση SPMV που ήταν η επόμενη που έπρεπε να εκτελεστεί.

5.2 Σύγκριση απόδοσης του FREDDO με OpenMP στο HPCG Benchmark

Όπως είπαμε προηγουμένως εισαγάγαμε το πλαίσιο προγραμματισμού FREDDO με δύο τρόπους στο HPCG. Πιο κάτω παρουσιάζονται τα αποτελέσματα του πρώτου μοντέλου που χρησιμοποιήθηκε σε σύγκριση με την εκτέλεση της OpenMP και της σειριακής εκτέλεσης. Αξίζει να σημειωθεί ότι η εκτελέσεις έγιναν στην ίδια μηχανή και χωρίς οποιαδήποτε optimization flags του μεταγλωττιστή για να έχουμε πιο αντικειμενική εικόνα για το δικό μας μοντέλο που στο παρόν στάδιο του δεν επωφελείται ακόμη από optimizations του μεταγλωττιστή.



Σχήμα 5.2.1: Επιτάχυνση FREDDO σε σύγκριση με OpenMP

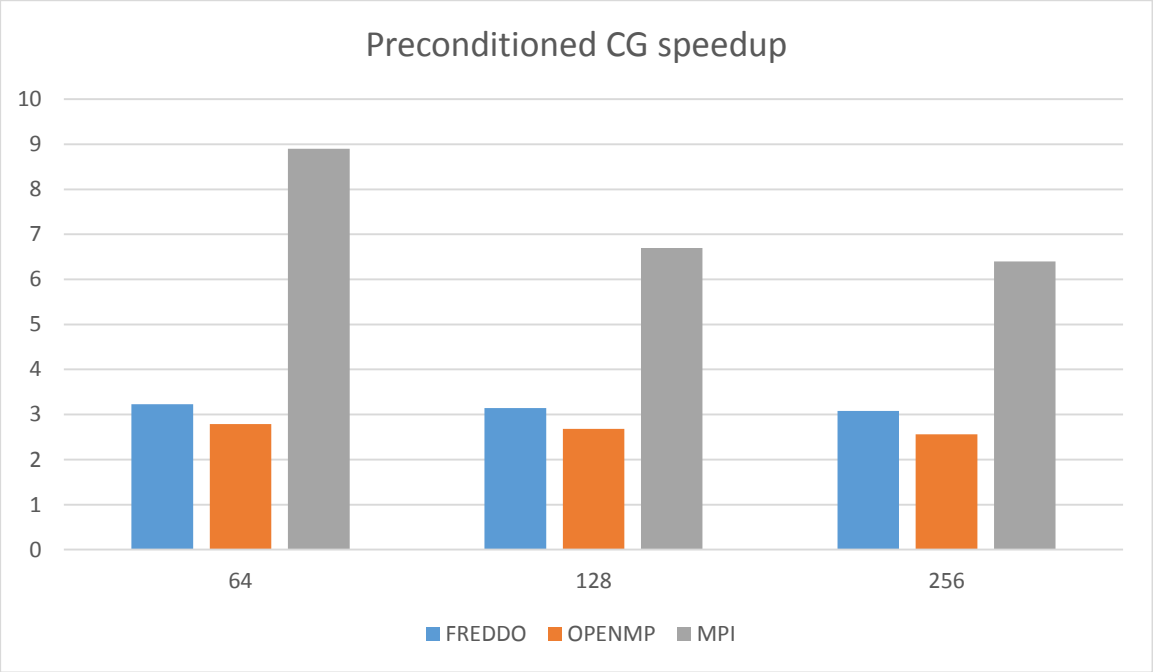
Όπως είναι εμφανές η εκτέλεση του FREDDO ήταν πιο αποδοτική από την εκτέλεση με OpenMP. Ο λόγος που παρουσιάζονται καλύτερα αποτελέσματα είναι απλός. Στο OpenMP fork-join μοντέλο τα threads δημιουργούνται μόλις ξεκινά ο παράλληλος κώδικας και αποδεσμεύονται με το πέρας της εκτέλεσης του παράλληλου κώδικα. Επομένως, κάθε φορά που καλείτε μια παράλληλη συνάρτηση για παράδειγμα, τα threads δημιουργούνται εκτελούν το κώδικα που τους ανατίθεται και στο τέλος καταστρέφονται.

Αντιθέτως, στο FREDDO μοντέλο δημιουργείται από την αρχή του προγράμματος ένα thread pool, δηλαδή τα νήματα είναι είδη δημιουργημένα και απλά όταν δηλώνετε ένα νέο dthread ανατίθεται σε ένα από τα υπάρχοντα threads. Μετά την ολοκλήρωση της δουλειάς του dthread το νήμα δεν καταστρέφεται αλλά αντιθέτως επιστρέφει στο threadpool και αναμένει νέα εργασία. Με τον τρόπο αυτό το FREDDO αποφεύγει τα διάφορα κόστη (overheads) που

παράγονται κατά την δημιουργία threads. Αυτός είναι και ο κυριότερος λόγος που το FREDDO μοντέλο υπερέχει του OpenMP .

Να σημειωθεί ότι με τα optimizations flags του μεταγλωττιστή το OpenMP μοντέλο ήταν ταχύτερο από το FREDDO γιατί η υπάρχον έκδοση ακόμη δεν επωφελείται από οποιαδήποτε optimizations του μεταγλωττιστή.

Στα πλαίσια περαιτέρω μελέτης του μοντέλου FREDDO τρέξαμε επίσης τον αλγόριθμο Preconditioned CG μέσω του HPCG. Συγκρίναμε στην συνέχεια την επιτάχυνση του FREDDO στον Preconditioned αλγόριθμο με αυτή του απλού αλγορίθμου και με τις εκτελέσεις OpenMP και MPI. Τα αποτελέσματα σχετικά με την επιτάχυνση παρουσιάζονται στο σχήμα 5.2.2.



Σχήμα 5.2.2: Η επιτάχυνση των FREDDO,OpenMP και MPI στον Preconditioned CG

Και σε αυτή την περίπτωση το FREDDO ήταν ταχύτερο από την OpenMP για τους λόγους που εξηγήθηκαν πιο πάνω (thread pool αντί fork-join). Η εκτέλεση MPI ήταν αρκετά ταχύτερη από τις υπόλοιπες γιατί η εκτέλεση MPI εκμεταλλεύεται τις δυνατότητες επικοινωνίας που έχει μεταξύ των διεργασιών και στον Preconditioned αλγόριθμο χρησιμοποιούνται όλες οι διεργασίες για τις προεργασίες που εκτελεί ο αλγόριθμος πριν τον τελικό υπολογισμό.

5.3 Προβλήματα και λύσεις που προτάθηκαν για το μοντέλο FREDDO

Κατά την διάρκεια της εισαγωγής του FREDDO στο HPCG παρουσιάστηκαν μερικά προβλήματα που δυσκόλευαν την όλη διαδικασία. Το γεγονός αυτό φυσικά ήταν θετικό γιατί ο κύριος σκοπός της εργασίας αυτής ήταν να δοκιμαστεί και να αξιολογηθεί το FREDDO μοντέλο που βρίσκεται υπό ανάπτυξη. Το HPCG ήταν ένα μεγάλος κώδικας που περιείχε πέραν των 20 αρχείων και έφθανε μέχρι τις 5,500 γραμμές κώδικα και όπως ήταν αναμενόμενο εμφανίστηκαν μερικά προβλήματα. Τα προβλήματα αυτά προτάθηκαν στην ομάδα υλοποίησης του FREDDO και διορθώθηκαν.

Πρόβλημα 1:

Ως γνωστό το πιο κύριο κομμάτι του DDM μοντέλου είναι το Thread Scheduling Unit. Ο TSU γίνεται initialize στην αρχή του προγράμματος και πρέπει να είναι κοινός σε όλα τα μέρη του προγράμματος. Στην αρχική έκδοση του FREDDO που είχαμε ο TSU ήταν ένα αντικείμενο τύπου TSU. Συνεπώς για να μπορούν όλα τα μέρη του προγράμματος να μπορούν να χρησιμοποιούν το TSU έπρεπε να εισάγεται σαν παράμετρος στα arguments των διαφόρων συναρτήσεων. Σε προγράμματα του μεγέθους του HPCG αυτό δημιουργούσε μεγάλα προβλήματα στον προγραμματιστή γιατί θα έπρεπε με κάποιο τρόπο να περάσει το TSU αντικείμενο σε όλο τον κώδικα. Μετά την επισήμανση του προβλήματος αυτού η νέα έκδοση του FREDDO δεν απαιτεί την δημιουργία αντικειμένου TSU αλλά πλέον όλες οι λειτουργίες του FREDDO είναι προσβάσιμες μέσω των C++ namespaces. Το μόνο που απαιτείται για να παρέχονται οι λειτουργίες είναι η συμπερίληψη της βιβλιοθήκης FREDDO σε κάθε αρχείο που την χρησιμοποιεί.

Πρόβλημα 2:

Το δεύτερο σημαντικό πρόβλημα που παρουσιάστηκε αφορούσε στον τρόπο που τα διάφορα instances ενός dthread είχαν πρόσβαση σε δεδομένα του προγράμματος. Στην αρχική έκδοση του FREDDO ο μόνος τρόπος για να βλέπει ένα instance δεδομένα ήταν αυτά τα δεδομένα να είχαν global εμβέλεια. Λόγο του προβλήματος αυτού, όπου χρησιμοποιείτο το FREDDO έπρεπε να αλλαχθεί η δομή του προγράμματος και να δηλωθούν global μεταβλητές. Η λύση που προτάθηκε στην ομάδα ανάπτυξης του FREDDO ήταν να μπορεί κάθε instance να παίρνει δικά του δεδομένα, για παράδειγμα μέσο ενός struct. Στην νέα έκδοση του FREDDO που μας δόθηκε κάθε instance πλέον μπορεί να παίρνει και σαν παράμετρο στην συνάρτηση του δεδομένα. Με αυτό τον τρόπο δεν χρειάζεται πλέον να αλλάζεται η δομή ολόκληρου του προγράμματος για την εισαγωγή του FREDDO γιατί πλέον με την χρήση του struct ο προγραμματιστής μπορεί να περνά στο instance οτιδήποτε δεδομένα χρειάζεται.

Πρόβλημα 3:

Τέλος, εμφανίστηκε πρόβλημα όταν δεσμεύαμε περισσότερα από 16 νήματα στην 32πύρηνη μηχανή που τρέχαμε τις δοκιμές. Το πρόβλημα αυτό αναφέρθηκε στην ομάδα ανάπτυξης του FREDDO και λύθηκε. Όπως μας αναφέρθηκε το πρόβλημα παρουσιαζόταν λόγω κάποιων περιορισμών που είχε το λειτουργικό σύστημα που αφορούσε τον αριθμό των νημάτων που επιτρεπόταν να δημιουργηθούν από μια διεργασία.

Κεφάλαιο 6

Συμπεράσματα

6.1 Συμπεράσματα

Η πρώτη επαφή με το προγραμματιστικό πλαίσιο FREDDO που αναπτύσσεται στο τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου ήταν αρκετά εποικοδομητική. Μέσω της εργασίας αυτής είχαμε την δυνατότητα να γνωρίσουμε μια νέα τεχνική παράλληλου προγραμματισμού, διαφορετική με εκείνες που γνωρίζαμε μέχρι τότε. Θεωρούμε ότι πέτυχαμε τους στόχους μας που ήταν η επαφή με ένα νέο είδος προγραμματισμού, το Data Driven μοντέλο, η αξιολόγηση του πλαισίου FREDDO και ο εντοπισμός τυχόν προβλημάτων του πλαισίου.

Σε συνδυασμό με το HPCG benchmark ,που είναι ένας ευρέος γνωστός και μεγάλος κώδικας, είχαμε την ευκαιρία να δοκιμάσουμε εκτενώς το FREDDO σε θέματα προγραμματισμού. Το μοντέλο αυτό όπως είναι φυσικό έχει και πλεονεκτήματα αλλά και μειονεκτήματα/περιορισμούς τα οποία φάνηκαν και κατά την διάρκεια της αξιολόγησης του στο HPCG benchmark. Τα σημαντικότερα πλεονεκτήματα του είναι τα πιο κάτω:

Πλεονεκτήματα:

- Πιο αποδοτικό μοντέλο διαχείρισης των νημάτων λόγω του ότι τα νήματα είναι δημιουργημένα σε ένα threadpool και αποφεύγονται τα συνεχές κόστη δημιουργίας νέων νημάτων
- Το χαρακτηριστικό nesting που δίνει την δυνατότητα παράλληλης εκτέλεσης επανάληψης for μέχρι και 3 φωλιάσματος.
- Μέσω του Data Driver μοντέλου μπορεί να επιτευχθεί ο μέγιστος δυνατός προγραμματισμός. Μέσω του μοντέλου αυτού αποφεύγονται οποιαδήποτε barriers και κανένα νήμα δεν μένει στάσιμο χωρίς να εκτελεί εντολές.

Μειονεκτήματα:

- Ο Thread Scheduling Unit(TSU) δεσμεύει ένα πυρήνα του συστήματος και ο πυρήνας αυτός δεν μπορεί να χρησιμοποιηθεί για οποιοδήποτε άλλη εργασία.

- Το μοντέλο αυτό είναι δύσκολο να εφαρμοστεί σε υπάρχων προβλήματα. Δεν μπορεί δηλαδή να ενσωματωθεί εύκολα σε υπάρχων κώδικα και απαιτείται μεγάλος προγραμματιστικός κόπος για να οργανωθεί το πρόβλημα σε ένα μοντέλο Data Driven.

- Στο παρόν στάδιο του FREDDO πλαισίου προγραμματισμού για C++ δεν μπορεί να έχει κέρδος από optimizations του μεταγλωττιστή όπως έχουν αντίστοιχα μοντέλα. Αναμένετε φυσικά μελλοντικές εκδόσεις του FREDDO να έχουν αυτή την δυνατότητα.

Η προσωπική άποψη που αποκόμισα από την εργασία αυτή είναι ότι το μοντέλο FREDDO έχει προοπτικές ανάπτυξης και μπορεί να ανταγωνιστεί ισάξια τα άλλα μοντέλα παραλληλισμού που υπάρχουν σήμερα. Αναμφισβήτητα, χρειάζεται αρκετή προσπάθεια μέχρι την τελική έκδοση του FREDDO για C++ αλλά είμαι σίγουρος ότι θα ‘ρθει η μέρα που το FREDDO θα γίνει ευρέως γνωστό στον προγραμματιστικό κόσμο της πληροφορικής.

Πιστεύω, ότι στο προσεχές μέλλον κατά την μετάβαση της τεχνολογίας στα many-core το DDM σαν μοντέλο προγραμματισμού θα γίνει απαραίτητο. Ο συγχρονισμός που παρέχει το μοντέλο DDM είναι ο ιδανικός για συστήματα many core. Με το DDM μπορεί να ελαχιστοποιείται η χρήση πρωτοκόλλου συνέπειας μνήμης γιατί το DDM ουσιαστικά δουλεύει σαν ένα πρωτόκολλο συνέπειας μνήμης. Συγκεκριμένα, το ότι κάθε νήμα εκτελείται μόνο όταν τα δεδομένα του είναι έτοιμα ουσιαστικά συνεπάγει το ότι όταν εκτελείται το νήμα δεν υπάρχει η οποιαδήποτε ανάγκη υλοποίησης πρωτοκόλλου συνέπειας μνήμης. Επιπρόσθετα, με το DDM στα many cores θα παρέχει ένα απλό τρόπο συντονισμού των νημάτων. Ουσιαστικά, με τον TSU αυτοματοποιείται οι δρομολόγηση των νημάτων με ένα αποδοτικό τρόπο.

6.2 Μελλοντική Εργασία

Λόγω της συνεχής ανάπτυξης του FREDDO framework για C++ η αξιολόγηση του πρέπει να συνεχιστεί δοκιμάζοντας το και σε άλλα προβλήματα. Επειδή το HPCG Benchmark όπως αποδείχθηκε δεν ήταν το ιδανικό για μοντέλο Data Driven θα ήταν εξαιρετικά χρήσιμο να δοκιμαστεί σε άλλα προβλήματα που το FREDDO θα μπορούσε να επιτύχει αξιόλογη επιτάχυνση εκμεταλλεύοντας πλήρως το Data Driven μοντέλο. Πρόσθετα, θα ήταν ωφέλιμο αν το μοντέλο αυτό δοκιμαζόταν και σε περισσότερους από 32 πυρήνες που το δοκιμάσαμε εμείς και σε μηχανές με περισσότερη κύρια μνήμη για να είναι δυνατή η εκτέλεση του HPCG με μεγαλύτερο μέγεθος προβλήματος.

Επίσης, θα ήταν χρήσιμο αν το framework αναπτυχθεί και σε άλλες γλώσσες προγραμματισμού πέραν από την C++ για να μπορεί να χρησιμοποιείται από περισσότερους προγραμματιστές. Θα μπορούσαν επίσης να αναπτυχθούν και να ενσωματωθούν έτοιμοι αλγόριθμοι στην βιβλιοθήκη παρόμοιοι με αυτούς που έχει η OpenMP(π.χ reduction) που θα βοηθούσε στην καλύτερη εμπειρία χρησιμοποίησης της βιβλιοθήκης από τους προγραμματιστές.

Βιβλιογραφία

[1] http://en.wikipedia.org/wiki/TOP500

[2] http://www.top500.org/

[3] Jack Dongarra Michael A. Heroux,“Toward a New Metric for Ranking High Performance Computing Systems”,

[4] Stavrou K., Nikolaides M., Pavlou D., Arandi S., Evripidou P., Trancoso P. "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," Parallel Processing, 2008. ICPP '08. 37th International Conference on, vol., no., pp.25-34, 9-12 Sept. 2008

[5] S. Arandi and P. Evripidou, “DDM-VMc:The Data-Driven Multithreading Virtual Machine on the Cell Processor ,” Univerisity of Cyprus, Tech. Rep. TR-09-1, 2009.

[6] http://openmp.org/wp/

[7] http://www.open-mpi.org/

[8] http://www.etinternational.com/index.php/products/swarmbeta

[9] www.threadingbuildingblocks.org

[10] Matheou, George, and Paraskevas Evripidou. "Architectural support for data-driven execution." *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015): 52.

[11] HPCG Overview Talk from SC'13 TOP 500 BOF.(.pdf), Jack Dongarra & Piotr Luszczek University of Tennessee/ORNL Michael Heroux Sandia National Labs

Παράρτημα Α:

Στο παράρτημα αυτό παρουσιάζονται κομμάτια κώδικα των συναρτήσεων ComputeSPMV, ComputeWAXBY και ComputeDotProduct οι οποίες υλοποιήθηκαν με το πλαίσιο προγραμματισμού FREDDO

ComputeSPMV:

α)Συνάρτηση που εκτελούν τα instances των Dthreads

```
typedef struct {
    const SparseMatrix * A;
    double *x;
    double *y;
} Info;
void multiply(KernelID worker, ContextArg context, void* data) {
    Info* d = (Info*) data;

    local_int_t start=(int)context->Inner*space;
    local_int_t end=start+space;

    if(context->Inner==num_cores-1){
        end=n;

    }

    for(int i=start;i<end;i++){
    double sum = 0.0;
    const double * const cur_vals = d->A->matrixValues[i];
    const local_int_t * const cur_inds = d->A->mtxIndL[i];
    const int cur_nnz = d->A->nonzerosInRow[i];
    for (int j = 0; j < cur_nnz; j++)
        sum += cur_vals[j] * d->x[cur_inds[j]];

    d->y[i] = sum;
    }
}
```

β) Δημιουργία,update και έναρξη Dthread

```
#ifndef HPCG_NODDM
    DDM = true;
#endif
    if (DDM) {
        n = A.localNumberOfRows;
        if(n>=num_cores){
        space=n/num_cores;

        }
        else{
            space=1;
            num_cores=n;
        }

        ddm::addDThread(multiply, 2, Nesting::ONE, SchMethod::Dynamic, 0, 1,
                &info);
        ddm::update(0, 2,CREATE_N1(0), CREATE_N1(num_cores-1));
        ddm::run();

    } else {
```

Compute WAXBY:

α) Συναρτήσεις που εκτελούν τα instances

```
void compute_alpha(KernelID worker, ContextArg context, void* data) {
    Info *d = (Info*) data;
    local_int_t start = (int)context->Inner * space1;
    local_int_t end = start + space1;
    if (context->Inner == num_cores1 - 1) {
        end = n1;
    }
    //printf("%d %d\n",start,end);
    //ww[i] = xv[i] + beta * yy[i];
    for (local_int_t i = start; i < end; i++)
        d->w[i] = d->x[i] + d->beta * d->y[i];

}

void compute_beta(KernelID worker, ContextArg context, void* data) {
    Info *d = (Info*) data;
    //ww[i] = alpha * xv[i] + yy[i];
    local_int_t start = (int)context->Inner * space1;
    local_int_t end = start + space1;

    if (context->Inner == num_cores1 - 1) {
        end = n1;
    }

    for (local_int_t i = start; i < end; i++)
        d->w[i] = d->alpha * d->x[i] + d->y[i];

}

void compute_else(KernelID worker, ContextArg context, void* data) {
    Info *d = (Info*) data;
    //ww[i] = alpha * xv[i] + beta * yy[i];
    local_int_t start = (int)context->Inner * space1;

    local_int_t end = start + space1;
    if (context->Inner == num_cores1 - 1) {
        end = n1;
    }

    for (local_int_t i = start; i < end; i++)
        d->w[i] = d->alpha * d->x[i]+ d->beta * d->y[i];

}
```

β)Δημιουργία,update και έναρξη του Dthread

```
} else {
    n1=n-1;
    if(n>=num_cores1){
    space1 = n1 / num_cores1;
    }
    /*else{
        num_cores1=n;
        space1=1;
    }*/
    if (alpha == 1.0) {
        ddm::addDThread(compute_alpha, 1, Nesting::ONE, SchMethod::Dynamic,0, 1, &info);

    } else if (beta == 1.0) {
        ddm::addDThread(compute_beta, 1, Nesting::ONE, SchMethod::Static,
                0, 1, &info);

    } else {
        ddm::addDThread(compute_else, 1, Nesting::ONE, SchMethod::Static,
                0, 1, &info);

    }
    ddm::update(0, 1, CREATE_N1(0), CREATE_N1(num_cores1-1));
    ddm::run();
    ddm::removeDThread(1);
}
```

γ) ComputeDotProduct:

α) Συναρτήσεις που εκτελούν τα instances

```
void compute_reduction(KernelID worker, ContextArg context, void* data) {
    Info* d = (Info*) data;
    local_int_t start = (int) context->Inner * sp;
    local_int_t end = start + sp;
    if (context->Inner == num_c - 1) {
        end = total;
    }
    for (int i = start; i < end; i++) {
        d[context->Inner].sum += d[context->Inner].x[i]
                * d[context->Inner].x[i];
    }
    ddm::update(worker, 4, CREATE_N1(0), CREATE_N1(0));
}

void compute_reduction2(KernelID worker, ContextArg context, void* data) {
    Info* d = (Info*) data;

    local_int_t start = (int) context->Inner * sp;
    local_int_t end = start + sp;

    if (context->Inner == num_c - 1) {
        end = total;
    }
    for (int i = start; i < end; i++) {
        d[context->Inner].sum += d[context->Inner].x[i]
                * d[context->Inner].y[i];
    }
    ddm::update(worker, 4, CREATE_N1(0), CREATE_N1(0));

}
```

β)Δημιουργία,update και έναρξη των Dthreads

```
        if (on_ddm) {
            ddm::addDThread(compute_reduction, 3, Nesting::ONE,
                    SchMethod::Dynamic, 0, 1, &instances_data, num_c, 1, 1);
            ddm::addDThread(collect_sums, 4, Nesting::ZERO, SchMethod::Dynamic,
                    0, num_c, &instances_data);
            ddm::update(0, 3, CREATE_N1(0), CREATE_N1(num_c-1));
            time_start = gtod_micro1();
            ddm::run();
            time_finish = gtod_micro1();
            /*printf("ddm time %f result %f \n", time_finish - time_start,
                    ddm_sum);*/

            ddm::removeDThread(3);
            ddm::removeDThread(4);
        } else {
#ifndef HPCG_NOOPENMP
            time_start = gtod_micro1();
#pragma omp parallel for reduction (+:local_result)
#endif

            for (local_int_t i = 0; i < n; i++) {
                local_result += xv[i] * xv[i];
            }

            time_finish = gtod_micro1();
            /*printf("omp time %f result %f\n", time_finish - time_start,
                    local_result);*/
        }
    } else {
        if (on_ddm) {
            ddm::addDThread(compute_reduction2, 3, Nesting::ONE,
                    SchMethod::Dynamic, 0, 1, &instances_data, num_c, 1, 1);
            ddm::addDThread(collect_sums, 4, Nesting::ZERO, SchMethod::Dynamic,
                    0, num_c, &instances_data);
            ddm::update(0, 3, CREATE_N1(0), CREATE_N1(num_c-1));
            time_start = gtod_micro1();
            ddm::run();
```