ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΧΕΔΙΟ ΠΑΡΟΥΣΙΑΣΗΣ
ΑΤΟΜΙΚΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

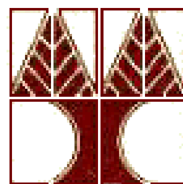
Μάιος 2015

Ατομική Διπλωματική Εργασία

HIGH PERFORMANCE COMPUTING APPLICATION

Κωνσταντίνος Σολομωνίδης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2015

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

HIGH PERFORMANCE COMPUTING APPLICATION

Κωνσταντίνος Σολομωνίδης

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

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή μου κύριο Παρασκευά Ευριπίδου που μου έδωσε την ευκαιρία να ασχοληθώ με το θέμα αυτό. Είναι ένα θέμα που με ενδιαφέρει πολύ και μέσα από την έρευνά μου έχω μάθει αρκετά πράγματα στο τομέα αυτό, τα οποία θα με βοηθήσουν ακόμη περισσότερο στην μετέπειτα πορεία μου. Επίσης θα ήθελα να τον ευχαριστήσω για την βοήθεια και τις συμβουλές του κατά την διάρκεια της έρευνάς μου οι οποίες ήταν καθοδηγητικές έτσι ώστε να φέρω εις πέρας την διπλωματική μου εργασία.

Επίσης θα ήθελα να ευχαριστήσω τον Γιώργο Ματθαίου διδακτορικό του κύριου Ευριπίδου ο οποίος με βοήθησε πάρα πολύ καθ' όλη τη διάρκεια της έρευνάς μου. Οι γνώσεις του στο θέμα αυτό ήταν καθοριστικές ώστε να αντιμετωπίζω κάθε δυσκολία και εμπόδιο που εμφανιζόταν κυρίως στο κομμάτι της υλοποίησης.

Επιπρόσθετα θα ήθελα να ευχαριστήσω τους φίλους Πάτροκλο Πατρόκλου και Χρυστάλλα Τσουτσούκη οι οποίοι μέσα από τις γνώσεις τους και την εμπειρία τους στο θέμα αυτό με βοήθησαν πάρα πολύ καθ' όλη τη διάρκεια της υλοποίησης, καθώς και όλους τους φίλους που μου συμπαραστάθηκαν σε κάθε δυσκολία.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου που μου στάθηκε σε όλη τη διάρκεια της φοίτησής μου ψυχολογικά, και που μου συμπαραστάθηκε σε κάθε δυσκολία και εμπόδιο και ήταν πάντα ο λόγος και η αιτία κάθε επιτυχίας μου.

Περίληψη

Το θέμα της διπλωματικής μου εργασίας είναι High Performance Computing Application. Με τον όρο αυτό εννοούμε την ανάλυση της απόδοσης των υπερ-υπολογιστών και την εξαγωγή συμπερασμάτων που θα μπορούσαν να μας βοηθήσουν στο να κάνουμε τις απαραίτητες αλλαγές που θα βελτίωναν στο μέγιστο την απόδοση τους. Οι παράμετροι που επηρεάζουν στο τομέα αυτό είναι κυρίως η αρχιτεκτονική της μηχανής στην οποία τρέχει κάποιο πρόγραμμα καθώς και η δομή του προγράμματος. Τα προγράμματα που έτρεξα για να μπορέσω να βγάλω συμπεράσματα ήταν σε συγκεκριμένες μηχανές του τμήματος επομένως οι αλλαγές έγιναν στην δομή των προγραμμάτων με βάση την τεχνική παράλληλου προγραμματισμού που χρησιμοποιούσαν (πχ. OpenMP, MPI, DDM, TBB).

Σε αρχικό στάδιο μου δόθηκαν κάποια προβλήματα μαθηματικού τύπου τα οποία τείνουν μεγάλης πολυπλοκότητα και για μεγάλο αριθμό δεδομένων ο χρόνος εκτέλεσης τους σε ένα σειριακό μοντέλο προγραμματισμού είναι αρκετά μεγάλος. Για το σκοπό αυτό μου ζητήθηκε να μελετήσω τα προβλήματα αυτά να βρω τις εξαρτήσεις που υπάρχουν μεταξύ των πράξεων που γίνονται και να δω κατά πόσο το πρόβλημα μπορεί να γίνει παράλληλο και σε πιο βαθμό.

Σε περαιτέρω στάδιο μου ζητήθηκε να μελετήσω κάποιες γλώσσες παράλληλου προγραμματισμού για το πώς λειτουργούν και να τρέξω κάποια έτοιμα προγράμματα υλοποιημένα σε αυτές τις γλώσσες ώστε να βγάλω κάποιες μετρικές και να συγκρίνω τις αποδόσεις τους με άλλες γλώσσες που χρησιμοποιούνται για παραλληλισμό όπως είναι η OpenMP και το MPI που χρησιμοποιείτε κυρίως για κατανεμημένα συστήματα. Η γλώσσες αυτές είναι το SWARM (SWift Adaptive Runtime Machine) και το TBB (Thread Building Block) της εταιρείας Intel. Λόγο όμως των ελάχιστων πηγών ενημέρωσης και εκμάθησης της τεχνολογίας SWARM δεν ήταν εφικτό να αναλυθεί και να μελετηθεί σε ικανοποιητικό βαθμό έτσι επικεντρώθηκα κυρίως στη γλώσσα TBB.

Αφού μελέτησα την γλώσσα TBB το πώς δουλεύει και τι πλεονεκτήματα έχει έτρεξα ορισμένα προγράμματα για να δω την επιτάχυνση τους σε σχέση με το σειριακό αλγόριθμο και σύγκρινα τις μετρικές αυτές με την γλώσσα προγραμματισμού DDM γλώσσα η οποία δημιουργήθηκε από την ομάδα του εργαστηρίου του επιβλέπων καθηγητή μου. Πιο συγκεκριμένα σε πρακτικό επίπεδο υλοποίησα τον αλγόριθμο block-matrix-multiplication ένας από τους πιο γνωστούς και πλήρως παραλληλοποιήσιμους αλγόριθμους μέσα από την οποίο μπορείς να βγάλεις αρκετά συμπεράσματα. Σε τελικό στάδιο χρησιμοποίησα το benchmark HPCG το οποίο χρησιμοποιείτε για να μετρηθεί η απόδοση των υπερ-υπολογιστών και το οποίο είναι υλοποιημένο σε γλώσσα OpenMP και MPI. Υλοποίησα το benchmark σε TBB με MPI και πήρα τις τελικές μετρήσεις και τα τελικά συμπεράσματα για την γλώσσα TBB που είχα να μελετήσω.

Περιεχόμενα

Κεφάλαιο **1** **Εισαγωγή**…………………………………………………………… **1**

1.1 High Performance Computing 1

1.2 High Performance Computing Rise 4

1.3 High Performance Computing Use 5

Κεφάλαιο **2** **Multicore Programming**……………………………………………**15**

2.1 Εισαγωγή στην παράλληλη επεξεργασία 15

2.2 Υλοποίηση ενός παράλληλου αλγόριθμου 18

2.3 Προβλήματα Παράλληλου Προγραμματισμού 20

Κεφάλαιο **3** **HPCG**……………………………………………………….. **25**

3.1 Εισαγωγή 25

3.2 HPCG Δομή και Ρουτίνες 30

3.3 Μοντέλο Ρουτινών HPCG 34

3.4 Μοντέλο Επικοινωνίας HPCG 31

3.5 Μοντελοποίηση ολόκληρου του benchmark 34

3.6 Αποτελέσματα και Επεξήγηση

Κεφάλαιο **4** **Thread Building Block (TBB)** ……………………………….**44**

4.1 Εισαγωγή 44

4.2 Γιατί Εργασίες και όχι νήματα 47

4.3 Λειτουργίες βιβλιοθηκών TBB 52

4.4 Block Matrix Multiplication TBB vs DDM 53

4.5 Αποτελέσματα HPCG με TBB και Επεξήγηση 54

ΚΕΦΑΛΑΙΟ **5** **Συμπεράσματα** …………………………………………………….**55**

5.1 OpenMP vs TBB 57

5.2 Παράλληλος Προγραμματισμός 58

Β ι β λ ι ο γ ρ α φ ί α……………………………………………………………**59**

Κεφάλαιο 1

Εισαγωγή

1.1 High Performance Computing 1

1.2 High Performance Computing Rise 4

1.3 High Performance Computing Use 5

1.1 High Performance Computing

Με τον όρο High Performance Computing εννοούμε την χρήση της παράλληλης επεξεργασίας με τον βέλτιστο δυνατό τρόπο για την εκτέλεση προχωρημένων προγραμμάτων εφαρμογής πιο γρήγορα, αποτελεσματικά και αξιόπιστα. Ο όρος αυτός ισχύει ιδιαίτερα για συστήματα που λειτουργούν πάνω από ένα TeraFlop ή 1012 floating point operations ανά δευτερόλεπτο. Ο όρος αυτός χρησιμοποιείται συχνά και ως συνώνυμο των supercomputers.

Supercomputer ονομάζεται ένας υπολογιστής που διαφέρει αισθητά από τους υπολογιστές που χρησιμοποιούνται από απλούς χρήστες όσον αφορά στον αριθμό των floating point operations που μπορεί να εκτελούν ανά δευτερόλεπτο. Αποτελούνται συνήθως από εκατοντάδες ή και χιλιάδες επεξεργαστές και χρησιμοποιούνται σε μεγάλα εργαστήρια για πολύ απαιτητικές προσομοιώσεις. Η δυνατότητα με την οποία μπορούν να εκτελούν υπολογισμούς μετριέται συνήθως σε Flops (Floating-point Operations Per Second), η οποία σήμερα έχει ξεπεράσει το 1 PetaFlop.

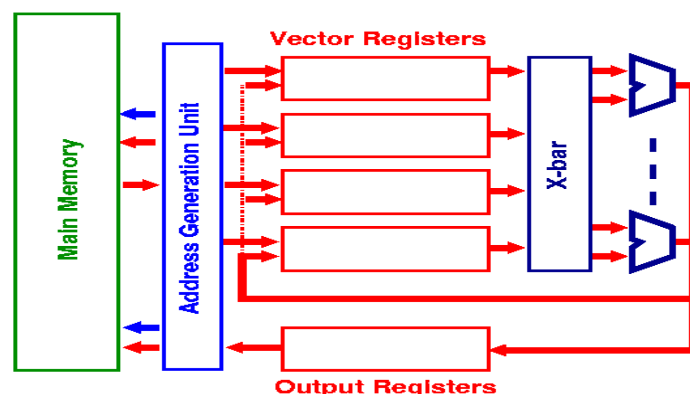
Οι σύγχρονες κατασκευές των supercomputers, χρησιμοποιούν κατά κανόνα σχεδόν χιλιάδες κοινούς επεξεργαστές σε παράλληλη διάταξη για επιτύχουν υψηλές ταχύτητες επεξεργασίας. Για του λόγου το αληθές θα αναφερθώ σε μερικούς από αυτούς τους υπολογιστές και στην απόδοσή τους. Ένας από αυτούς είναι ο Blue Gene/L οποίος περιέχει 65536 μικροεπεξεργαστές και έχει πετύχει μέγιστη ταχύτητα 596 TFlops (596 τρισεκατομμύρια operations το δευτερόλεπτο). Ένας από τους

ισχυρότερους είναι και ο Roadrunner της IBM ο οποίος χρησιμοποιεί αντίστοιχους επεξεργαστές με τον Blue Gene/L με 16000 επιπλέον οκταπήρηνους επεξεργαστές Cell της IBM και η συνολική του επεξεργαστική ισχύ φτάνει το ένα PetaFlop (δεκάκις τρισεκατομμύρια Flop), περίπου τέσσερις φορές μεγαλύτερη από το Blue Gene/L.

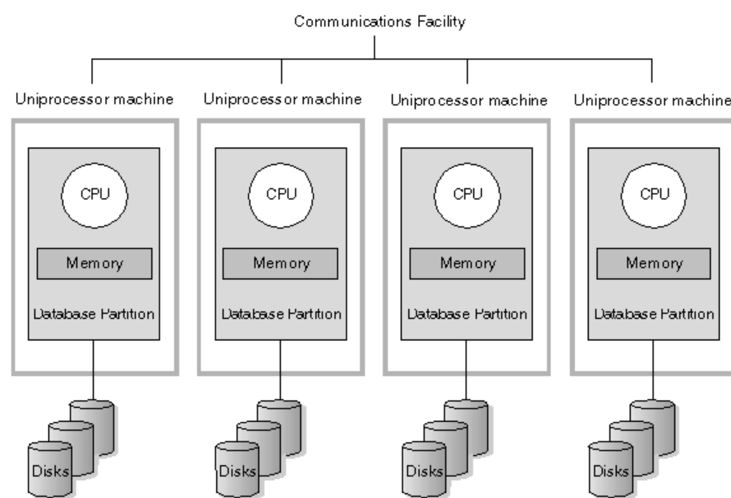
1.2 High Performance Architectures

Αρχικά είναι σημαντικό να αναφερθούμε στις διάφορες αρχιτεκτονικές που κτίστηκαν προκειμένου να υλοποιηθούν τέτοιου είδους μηχανές από τα αρχικά στάδια μέχρι και σήμερα. Οι πρώτες αρχιτεκτονικές για high performance computing ήταν τα vector processors τα οποία μέσα από τα χρόνια εξελίχτηκαν στους σημερινούς υπερυπολογιστές που έχουμε σήμερα. Πιο κάτω παρουσιάζω αναλυτικά όλα τα στάδια της εξέλιξης αυτών των αρχιτεκτονικών.

Vector Processor είναι μια κεντρική μονάδα επεξεργασίας που υλοποιεί ένα σετ εντολών που περιέχει οδηγίες οι οποίες λειτουργούν σε μονοδιάστατες συστοιχίες (one-dimensional arrays) των δεδομένων που ονομάζονται vectors. Τα vectors λειτουργούν αντίθετα με ένα scalar processor του οποίου οι εντολές εκτελούνται πάνω σε μεμονωμένα data items. Αυτού του είδους επεξεργαστές μπορούν να βελτιώσουν σημαντικά τις επιδόσεις σε συγκεκριμένες εργασίες με αριθμητικές προσομοιώσεις και άλλα συναφή.

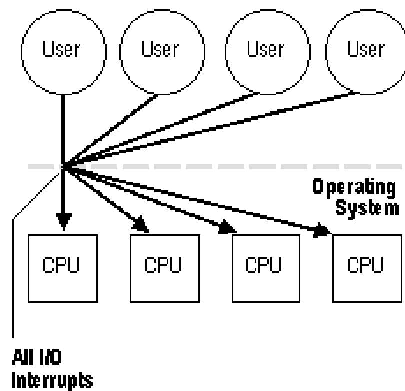


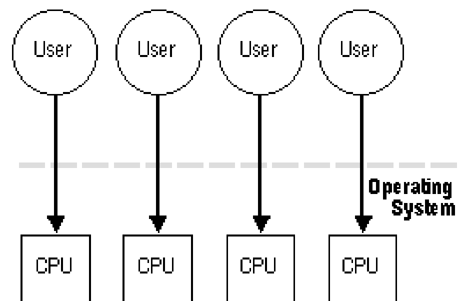
Massively Parallel Processing είναι η συντονισμένη επεξεργασία ενός προγράμματος από πολλαπλούς επεξεργαστές που λειτουργούν σε διαφορετικά μέρη του προγράμματος, με κάθε επεξεργαστή να χρησιμοποιεί το δικό του λειτουργικό σύστημα και μνήμη. Συνήθως οι επεξεργαστές MPP επικοινωνούν χρησιμοποιώντας μια διασύνδεση μηνυμάτων. Υπάρχουν εφαρμογές που 200 ή περισσότεροι επεξεργαστές χρειάζεται να συνεργαστούν για την διεκπεραίωση τους. Η υλοποίηση μιας εφαρμογής σε μια τέτοια αρχιτεκτονική είναι σχετικά περίπλοκη αφού απαιτεί σκέψη για το πώς να ενσωματώσει μια κοινή βάση δεδομένων μεταξύ των processors και για το πώς θα ανατεθούν οι εργασίες μεταξύ των επεξεργαστών.



Symmetric Multiprocessors είναι μια αρχιτεκτονική υπολογιστή όπου δύο ή περισσότεροι όμοιοι επεξεργαστές στο ίδιο μικροτσίπ, συνδέονται με μια κοινή διαμοιραζόμενη κύρια μνήμη μέσω ενός διαύλου και έχουν πρόσβαση στις ίδιες συσκευές εισόδου/εξόδου. Ο όρος αυτός αναφέρεται στο ότι όλοι οι επεξεργαστές είναι όμοιοι και ισότιμοι, ικανοί να εκτελέσουν τις ίδιες λειτουργίες. Τα συστήματα με την αρχιτεκτονική αυτή επιτρέπουν σε οποιαδήποτε διεργασία, ανεξαρτήτως του που βρίσκεται στη μνήμη, να εκτελεστεί από οποιονδήποτε επεξεργαστή. Με την υποστήριξη του λειτουργικού συστήματος μπορεί εύκολα να μετακινεί διεργασίες μεταξύ επεξεργαστών προκειμένου να εξισορροπήσει τον φόρτο εργασίας στον καθένα, αυξάνοντας τη συνολική απόδοση του συστήματος για το λόγο αυτό η αρχιτεκτονική αυτή χρησιμοποιείται στα περισσότερα multiprocessors systems. Ένα από τα ελαττώματα τους όμως είναι ότι δεν έχουν επεκτασιμότητα.

Asymmetric Multiprocessing: Symmetric Multiprocessing:





Distributed Systems ονομάζονται οι υπολογιστές οι οποίοι επιτρέπουν την ταυτόχρονη εκτέλεση πολλαπλών συνεργαζόμενων προγραμμάτων σε μία ή περισσότερες επεξεργαστικές μονάδες. Σε περαιτέρω στάδιο αυτή η αρχιτεκτονική θα αναλυθεί και επεξηγηθεί σε μεγαλύτερο βαθμό και σε περισσότερο βάθος.

Ένας υπολογιστής cluster αποτελείται από ένα σύνολο συνδεδεμένων υπολογιστών που λειτουργούν μαζί με τρόπο ώστε να μπορούν να θεωρηθούν ως ένα ενιαίο σύστημα. Τα συστατικά ενός cluster είναι συνήθως συνδεδεμένα μεταξύ τους με ένα local area network (LAN), με τον κάθε κόμβο να τρέχει το δικό του λειτουργικό σύστημα. Στις περισσότερες περιπτώσεις όλοι οι κόμβοι χρησιμοποιούν το ίδιο hardware και software αν και μπορεί να υπάρξουν περιπτώσεις που να χρησιμοποιούνται εντελώς διαφορετικά συστήματα. Τοποθετούνται συνήθως για να βελτιώσουν τις επιδόσεις και τη διαθεσιμότητα περισσότερο από ότι ενός single computer, καθώς είναι πολύ πιο αποδοτικοί και στην ταχύτητα.

1.3 Why High Performance Computing and Where

Ένα από τα βασικότερα ερωτήματα είναι γιατί να χρησιμοποιήσουμε μια τέτοια τεχνολογία και για ποιες εφαρμογές. Το HPC επιτρέπει όπως αναφέραμε και πιο πάνω να κάνουμε υπολογισμούς πολλές φορές πιο γρήγορα από ότι σε ένα συμβατικό επεξεργαστή. Αυτό σημαίνει ότι μπορεί να επιταχύνει πολλές χρονοβόρες επιχειρηματικές και άλλες διαδικασίες. Πιο κάτω αναφέρω ορισμένες από τις πιο σημαντικές εφαρμογές αυτής της τεχνολογίας και το πώς μπορεί να προσφέρει στην κοινωνία.

- Θα μπορούσε να φέρει επανάσταση σε ιατρικές διαδικασίες μέσω της δημιουργίας εικονικών ανθρώπων από όλα τα σχήματα και τις ηλικίες.
- Θα μπορούσε να αυξήσει την ανάκτηση του πετρελαίου κατά 50-70% με την πιο ακριβή σεισμική μοντελοποίηση των δεξαμενών μοντελοποίηση των δεξαμενών πετρελαίου. Επί του παρόντος, οι αβεβαιότητες στα σεισμικά μοντέλα μπορούν να οδηγήσουν σε σφάλματα κατά τη διάνοιξη το οποίο μπορεί οδηγήσει στην μείωση της εξαγωγής αλλά και στην αύξηση των περιβαλλοντικών επιπτώσεων.
- Επίσης χρησιμοποιείται για το σχεδιασμό αποδοτικών γεννητριών ανέμου και κύματος, βοηθώντας στην αξιοποίηση των ανανεώσιμων πηγών ενέργειας.
- Τέλος θα μπορούσε να χρησιμοποιηθεί για την μοντελοποίηση της εξάπλωσης των επιδημιών, όπου αυτό θα επιτρέπει στους αξιωματούχους της δημόσιας υγείας να παρέμβει κατάλληλα για να σταματήσει την επέκταση των απειλητικών νόσων για την ζωή των ανθρώπων.

Για τις μελλοντικές προβλέψεις πολλοί κατασκευαστές μιλούν ήδη για υπολογιστές κλίμακας ExaFlop (δηλαδή 1018 λειτουργίες ανά δευτερόλεπτο ή 1000 φορές πιο γρήγορα από ότι τους ταχύτερους υπολογιστές σήμερα) πριν από το 2020. Αυτό όμως θα αποφέρει και αρκετές προκλήσεις αφού οι μηχανές αυτής της εμβέλειας προβλέπουν διαφορετικές αρχιτεκτονικές από αυτές που κυριάρχησαν τα τελευταία 10 χρόνια, αρχιτεκτονικές που στο παρόν στάδιο είναι δύσκολο να φανταστούν. Πέρα του σχεδιασμού της αρχιτεκτονικής των μηχανών προβλέπεται και μεγάλη αλλαγή στην

δημιουργία των λογισμικών που θα τις υποστηρίξουν, λογισμικά το οποία θα διαθέτουν εντελώς διαφορετικά μοντέλα προγραμματισμού από αυτά που επικρατούσαν μέχρι σήμερα. Αυτό θα είναι ίσως και η μεγαλύτερη πρόκληση σε ότι αφορά την εξέλιξη αυτής της τεχνολογίας για τα επόμενα χρόνια.

Κεφάλαιο 2

Παράλληλη Επεξεργασία

2.1 Εισαγωγή στην παράλληλη επεξεργασία 15

2.2 Υλοποίηση ενός παράλληλου αλγόριθμου 18

2.3 Προβλήματα Παράλληλου Προγραμματισμού 20

2.1 Εισαγωγή στην παράλληλη επεξεργασία

Ο όρος High Performance Computing είναι συνώνυμος και με τον όρο παράλληλη επεξεργασία. Στο Κεφάλαιο 1 αναφέραμε ότι οι αρχιτεκτονικές αυτών των μηχανών διαθέτουν τεράστιων αριθμό επεξεργαστών και εκμεταλλεύονται στο μέγιστο την παράλληλη επεξεργασία των δεδομένων για επίλυση προβλημάτων μεγάλου μεγέθους σε ελάχιστο χρόνο σε σχέση με το σειριακό μοντέλο υλοποίησης. Είναι σημαντικό λοιπόν να αναφερθούμε στην παράλληλη επεξεργασία και στον παράλληλο προγραμματισμό αφού είναι η πηγή της απόδοσης αυτών των αρχιτεκτονικών [8].

Με τον όρο παράλληλος προγραμματισμός εννοούμε την ανάπτυξη εφαρμογών οι οποίες εκμεταλλεύονται την ύπαρξη πολλαπλών επεξεργαστικών μονάδων σε ένα πολυεπεξεργαστή για να επιτύχουν αύξηση των υπολογιστικών επιδόσεων και μείωση του απαιτούμενου χρόνου εκτέλεσης της εφαρμογής. Επομένως ο παράλληλος προγραμματισμός μπορεί να ιδωθεί ως ειδική περίπτωση ταυτόχρονου προγραμματισμού, όπου η εκτέλεση γίνεται πραγματικά παράλληλα και όχι ψευδοπαράλληλα.

Τα παράλληλα συστήματα διακρίνονται σε πολυεπεξεργαστές κοινής μνήμης, όπου πολλαπλοί επεξεργαστές επικοινωνούν με μια κοινή μνήμη ενιαίου χώρου διευθύνσεων, και σε υπερυπολογιστές κατανεμημένης μνήμης, όπου πολλαπλά πακέτα

επεξεργαστή-ιδιωτικής μνήμης, με τον δικό του χώρο διευθύνσεων το καθένα, διασυνδέονται και επικοινωνούν μεταξύ τους μέσω ενός δικτύου διασύνδεσης. Το μοντέλο κοινού χώρου χρησιμοποιείται στον παράλληλο προγραμματισμό για πολυεπεξεργαστές, ενώ το μοντέλο μεταβίβασης μηνυμάτων χρησιμοποιείται στον παράλληλο προγραμματισμό για υπερυπολογιστές. Στο πρώτο οι επεξεργαστικές μονάδες ανταλλάσουν πληροφορίες προσπελάζοντας κοινόχρηστες μεταβλητές στην κοινή μνήμη , ενώ στο δεύτερο ανταλλάσσοντας μηνύματα. Και τα δύο μοντέλα μπορούν να χρησιμοποιηθούν σε διαφορετικές αρχιτεκτονικές που δεν είναι κατάλληλες για αυτές αλλά με πολύ χαμηλότερες επιδόσεις κάτι που θα δούμε και στην πράξη σε επόμενο κεφάλαιο.

2.2 Υλοποίηση ενός παράλληλου αλγόριθμου

Η παραλληλοποίηση ενός αλγόριθμου γίνεται αρχικά με την διάσπαση του σε πολλαπλά τμήματα τα οποία ανατίθενται σε ξεχωριστά νήματα ή διεργασίες και έτσι εκτελούνται παράλληλα σε διαφορετικές επεξεργαστικές μονάδες. Ωστόσο δεν είναι βέβαιο ότι ένας αλγόριθμος υλοποιημένος σε κάποιο πρόγραμμα μπορεί να παραλληλοποιηθεί πάντα. Υπάρχουν αλγόριθμοι οι οποίοι έχουν εξαρτήσεις μεταξύ των εντολών εκτέλεσης τους. Δηλαδή ή κάθε εκτέλεση εντολής εξαρτάται άμεσα από την προηγούμενη κάτι που είναι εφικτά αδύνατο να υλοποιηθεί παράλληλα. Τέτοιου είδους εξαρτήσεις είναι και το σημαντικότερο εμπόδιο για την παραλληλοποίηση. Ωστόσο ακόμα και σε αυτές τις περιπτώσεις μπορεί να υπάρξουν τρόποι με τους οποίους μπορούμε να βελτιστοποιήσουμε το χρόνο εκτέλεσης κάποιου προγράμματος. Για παράδειγμα αν ένα κομμάτι του προγράμματος εκτελεί ένα μεγάλο αριθμό πράξεων και κάποιες από αυτές τις πράξεις είναι εφικτό να γίνουν με τα μέχρι στιγμής δεδομένα τότε μπορούμε να εκτελέσουμε αυτές τις πράξεις ενόσω περιμένουμε την τιμή η οποία χρειάζεται για να ολοκληρωθεί αυτό το κομμάτι του προγράμματος. Δεν χρειάζεται να περιμένουμε μέχρι η συγκεκριμένη τιμή να είναι έτοιμη για να αρχίσουμε να εκτελούμε όλες τις πράξεις. Αυτή η τεχνική ονομάζεται pipeline μια τεχνική αρκετά χρήσιμη που βελτιώνει σημαντικά την επίδοση του προγράμματος μας όμως δεν παραλληλοποιεί την εκτέλεση του.

Η παραλληλοποίηση ενός σειριακού αλγόριθμου, προκειμένου να εκμεταλλευτούμε στο μέγιστο δυνατό την αύξηση των υπολογιστικών επιδόσεων που προσφέρει ο παραλληλισμός, διακρίνεται σε βήματα που εκτελούνται διαδοχικά. Αρχικά γίνεται η διάσπαση του ολικού υπολογισμού σε επιμέρους εργασίες. Αυτό γίνεται με βάση το πρόβλημα και εναπόκειται στον προγραμματιστή για το πώς θα γίνει αυτό. Στην συνέχεια γίνεται ανάθεση των εργασιών σε οντότητες εκτέλεσης (διεργασίες, νήματα) η οποία ανάθεση μπορεί να γίνει στατική δηλώνοντας για την κάθε μια ένα προκαθορισμένο σύνολο εργασιών που θα εκτελέσει ή δυναμικά, όπου οι εργασίες ανατίθενται μια-μια κατά τον χρόνο εκτέλεσης του προγράμματος ανάλογα με την φόρτο εργασίας της κάθε διεργασίας. Η δυναμική λύση είναι και η προτιμότερη γιατί λύνει ένα από τα προβλήματα του παράλληλου προγραμματισμού που θα αναφέρουμε στο επόμενο υποκεφάλαιο. Στην συνέχεια πρέπει να δηλωθεί ο τρόπος με τον οποίο θα συντονίζονται και θα επικοινωνούν οι οντότητες που εκτελούν τις εντολές του προγράμματος έτσι ώστε να μην υπάρξουν λάθη στο τελικό αποτέλεσμα. Σκοπός πέρα της βελτιστοποίησης του χρόνου είναι και το επιθυμητό αποτέλεσμα. Τελικό στάδιο και προαιρετικό με βάση την αρχιτεκτονική της μηχανής που χρησιμοποιούμε είναι η αντιστοίχηση των οντοτήτων σε επεξεργαστές η οποία πάλι μπορεί να είναι είτε στατική είτε δυναμική.

2.3 Προβλήματα Παράλληλου Προγραμματισμού

Ο παράλληλος προγραμματισμός είναι κάτι που δημιούργησε πολλά προβλήματα σε πολλούς προγραμματιστές. Είναι σημαντικό λοιπόν να κατανοήσουμε τα προβλήματα αυτά πριν από κάθε σχεδιασμό ενός παράλληλου προγράμματος, επειδή πολλά από αυτά γίνονται αντιληπτά μετά από την συνολική αποσύνθεση του προγράμματος και δεν μπορούν εύκολα να επιδιορθωθούν. Πιο κάτω θα αναφερθώ σε μερικά από αυτά τα προβλήματα που πρέπει να προσέχουμε που κατά την άποψη μου είναι και από τα πιο σημαντικά.

Ο αριθμός των νημάτων:

Μπορεί να φαίνεται ότι ένας αριθμός νημάτων έχει μια καλή απόδοση στο χρόνο εκτέλεσης, άρα ένας πολύ μεγαλύτερος αριθμός ίσως να είναι και πολύ καλύτερος. Στην πραγματικότητα το να υπάρχει πολύ μεγαλύτερος αριθμός νημάτων από ότι χρειάζεται μπορεί να μειώσει αισθητά το χρόνο εκτέλεσης περισσότερο και από ένα σειριακό μοντέλο. Και αυτό γιατί όσο πιο μεγάλος είναι ο αριθμός των νημάτων τόσο λιγότερη δουλειά θα διαμοιρασθεί στο κάθε ένα από αυτά, τόσο μικρή που το κόστος δημιουργίας και καταστροφής του νήματος θα κοστίζει πολύ περισσότερο από την δουλειά που θα εκτελέσει.

Επίσης το να δημιουργήσεις πολλά ταυτόχρονα νήματα λογισμικού συνεπάγεται και μεγάλο κόστος στην διαμοίραση των πόρων του hardware. Όταν υπάρχουν περισσότερα software νήματα από hardware, το λειτουργικό σύστημα τυπικά εκτελεί ένα round robin προγραμματισμό. Ο προγραμματισμός αυτός δίνει σε κάθε software νήμα ένα μικρό χρόνο να εκτελεστεί πάνω σε ένα νήμα hardware. Όταν ο χρόνος αυτός τελειώσει, αναστέλλεται προληπτικά το νήμα για να τρέξει ένα άλλο software νήμα στο ίδιο hardware νήμα. Το νήμα αυτό παγώνει στο χρόνο μέχρι να του δοθεί ένα άλλο μικρό κομμάτι χρόνου. Όταν λοιπόν ο προγραμματιστής δημιουργεί πολλά software νήματα, θεωρώντας ότι αυτό θα βελτιστοποιήσει το χρόνο εκτέλεσης του προγράμματος του, αυξάνει ουσιαστικά αυτές τις περιπτώσεις που παγώνει ο χρόνος κάτι που μειώνει την και την απόδοση του προγράμματος.

Race Condition and Convoys:

Το race condition προκύπτει όταν δύο οι περισσότερα νήματα προσπαθούν να ενημερώσουν τον ίδιο χώρο μνήμης. Για παράδειγμα όταν δύο νήματα θέλουν να αυξήσουν μια συγκεκριμένη μεταβλητή κατά ένα τότε και τα δύο νήματα δημιουργούν ένα αντίγραφο του block στο οποίο είναι φυλαγμένη η τιμή αυτής της μεταβλητής στην δική τους μνήμη και το αυξάνουν κατά ένα. Συνολικά η τιμή της μεταβλητής πρέπει να αυξηθεί κατά δύο όμως λόγω του ότι ταυτόχρονα και τα δύο νήματα δημιούργησαν τα αντίγραφαν και δεν πρόλαβαν να δουν το ένα την ενημέρωση του άλλου στο τελικό αποτέλεσμα όταν δηλαδή και τα δύο νήματα ενημερώσουν πίσω την κύρια μνήμη με

την καινούργια τιμή θα είναι αυξημένη κατά ένα μόνο. Σε περιπτώσεις όπου πολλά νήματα χρησιμοποιούν πολύ συχνά κοινές μεταβλητές το φαινόμενο αυτό συμβαίνει πολλές φορές με αποτέλεσμα να έχουμε λάθος αποτελέσματα.

Ένας τρόπος για να αποφευχθεί αυτό είναι πρόβλημα είναι τα οι κλειδαριές (locks). Όταν ένα νήμα θέλει να ενημερώσει μια κοινή μεταβλητή στο σημείο του προγράμματος όπου γίνεται η ενημέρωση βάζουμε κλειδαριές έτσι ώστε κανένα άλλο νήμα να μπορεί ταυτόχρονα να ενημερώσει την ίδια μεταβλητή μέχρις ότου το προηγούμενο νήμα ολοκληρώσει την διαδικασία της ενημέρωσης και ενημερωθεί η κύρια μνήμη για την νέα τιμή ώστε να την βλέπουν όλα τα νήματα.

Αυτή η λύση όμως απαιτεί και σωστό προγραμματισμό γιατί μπορεί να υπάρξουν προβλήματα όταν η συγκεκριμένη εντολή ενημέρωσης που κλειδώνουμε εκτελείται πολλές φορές ή όταν το νήμα το οποίο βρίσκεται μέσα στο κλειδωμένο κομμάτι κώδικα εκτελεί πολύ πιο αργά τις εντολές απ' ότι ένα νήμα που βρίσκεται έξω από αυτό το κομμάτι. Σε αυτές τις περιπτώσεις δημιουργείται ένας συνωστισμός στο σημείο του κλειδώματος με αποτέλεσμα πολλά νήματα να περιμένουν αδρανεί στο σημείο αυτό μέχρι να έρθει η σειρά τους να μπουν και να εκτελέσουν τις εντολές που κλειδώσαμε. Γι' αυτό πρέπει να γνωρίζουμε σε ποιες περιπτώσεις θα πρέπει να βάζουμε κλειδαριές ώστε να μην παρατηρούνται τέτοιου είδους προβλήματα γιατί έχουν σαν αποτέλεσμα την μείωση του χρόνου εκτέλεσης του προγράμματος πιο κάτω και από τη σειριακή εκτέλεση. Μεταβλητές που πρέπει να ενημερώνονται συχνά καλύτερα να δημιουργούμε προσωπικές για το κάθε νήμα ξεχωριστά και στο τέλος να γίνεται μια άθροιση όλων των προσωπικών μεταβλητών, έτσι με αυτό τον τρόπο μειώνουμε την ανάγκη των κλειδαριών στο ελάχιστο που μπορούμε.

Non-Blocking Algorithms:

Ένας τρόπος για την επίλυση των προβλημάτων των κλειδαριών που παρουσιάσαμε πιο πάνω είναι η δημιουργία αλγορίθμων χωρίς τη χρήση των κλειδαριών. Τέτοιοι αλγόριθμοί υπάρχουν και ονομάζονται non-blocking algorithms.

Το χαρακτηριστικό των non-blocking αλγορίθμων είναι ότι η διακοπή ενός νήματος δεν εμποδίζει το υπόλοιπο σύστημα να βελτιστοποίηση την απόδοση του. Δεν προσβάλλονται από προβλήματα όπως το convoying που αναφέραμε πιο πάνω ή τον συνωστισμό των νημάτων σε συγκεκριμένο σημείο του κώδικα όμως παρόλα τα πλεονεκτήματά τους έρχονται με μια νέα σειρά προβλημάτων που πρέπει να γίνουν κατανοητά για να είναι επιτυχημένη και η χρήση τους.

Οι αλγόριθμοι αυτοί βασίζονται σε ατομικές πράξεις γι' αυτό και λίγοι από αυτούς είναι απλοί. Οι περισσότεροι είναι πολύπλοκοι, διότι πρέπει να χειριστούν όλες τις πιθανότητες διαπλοκής των εντολών που υποστηρίζουν οι επεξεργαστές. Χρησιμοποιώντας ατομικές λειτουργίες δεν είναι αρκετό για να αποφύγουμε προβλήματα όπως το race condition, για το λόγο ότι συνθέτοντας ασφαλείς εντολές για κάθε νήμα ατομικά δεν συνεπάγεται αναγκαστικά ότι η όλη διαδικασία είναι ασφαλείς. Στο πιο κάτω σχήμα παρουσιάζεται μια περίπτωση λάθους υλοποίησης ενός non-blocking αλγόριθμου με μια πιο σωστή προγραμματιστικά λύση.

WRONG

```
InterlockedDecrement(&p->ref_count);
if (p->ref_count==0) {
    delete p;
```

another thread might decrement ref_count right before the if executes

CORRECT

```
if (InterlockedDecrement(&p->ref_count)==0)
    delete p;
```

Στο σχήμα φαίνεται ξεκάθαρα μια περίπτωση λάθους προγραμματισμού και πως μπορεί να διορθωθεί. Στο λάθος κώδικα εάν το count ήταν 2 κανονικά, δύο νήματα που εκτελούν αυτό το κώδικα μπορεί και τα δύο να μειώσουν το count, και τότε και τα δύο να βλέπουν την τιμή της μεταβλητής ίση με μηδέν την ίδια στιγμή. Ο σωστός αλγόριθμος εκτελεί την μείωση της μεταβλητής και το έλεγχο σε μια ατομική εντολή αποφεύγοντας αυτή την περίπτωση λάθους. Κάθε περίπτωση παράλληλου προγραμματισμού απαιτεί μεγάλη προσοχή, πρέπει σε εντολή να προνοούμε τον τρόπο με τον οποίο μπορεί να την αντιμετωπίσουν τα νήματα.

Memory, cache issues and consistency:

Τις τελευταίες δεκαετίες, η ταχύτητα των μικροεπεξεργαστών έχει αναπτυχθεί πολύ πιο γρήγορα από ό, τι η ταχύτητα της μνήμης. Ένας μικροεπεξεργαστής μπορεί να εκτελέσει εκατοντάδες πράξεις στο χρόνο που χρειάζεται για να διαβάσει ή να γράψει στην κύρια μνήμη. Γι' αυτό το λόγο ο χρόνος εκτέλεσης των προγραμμάτων περιορίζεται από τη συμφόρηση που γίνεται στην μνήμη και όχι από την ταχύτητα του επεξεργαστή.

Για να μπορέσουμε να διατηρήσουμε το εύρος ζώνης (bandwidth) πρέπει να κινούμε τα δεδομένα στην μνήμη με λιγότερη συχνότητα μεταξύ των επεξεργαστών, ή να "δένουμε" τα δεδομένα πιο σφιχτά. Η καλή συσκευασία των δεδομένων είναι μια σχετικά απλή διαδικασία και βελτιώνει την απόδοση και στα σειριακά μοντέλα. Για παράδειγμα μπορούμε συσκευάσουμε Boolean πίνακες μέσα σε μία Boolean τιμή ανά bit, αντί για κάθε Boolean τιμή μέσα σε ένα byte. Μπορούμε επίσης να χρησιμοποιούμε το μικρότερο τύπο που μπορεί να φυλάχτειι σε μια τιμή.

Working in the Cache

Η λιγότερη μετακίνηση των δεδομένων μεταξύ των επεξεργαστών είναι ακόμη πιο αποδοτικό από το να προγραμματίζουμε χρησιμοποιώντας το βέλτιστο τύπο μεταβλητών, καθώς οι γλώσσες προγραμματισμού δεν διαθέτουν ρητές εντολές που να μεταφέρουν δεδομένα μεταξύ πυρήνα και μνήμης. Η κυκλοφορία των δεδομένων προκύπτει από τον τρόπο που οι πυρήνες διαβάζουν και γράφουν στην μνήμη. Υπάρχουν δύο κατηγορίες αλληλεπιδράσεων: αυτές μεταξύ πυρήνα και μνήμης και αυτές μεταξύ των πυρήνων. Υπάρχει μια τεχνική που ονομάζεται cache-oblivious blocking η ο οποία αναδρομικά χωρίζει ένα πρόβλημα σε μικρότερα υποπροβλήματα, τόσο μικρά ώστε το κάθε ένα να χωράει στην κρυφή μνήμη. Με αυτό τον τρόπο μειώνουμε αισθητά την διακίνηση των δεδομένων και αυξάνεται η απόδοση του προγράμματος αφού δεν κάνουμε πολλές προσβάσεις στην κύρια μνήμη.

Γενικά η υλοποίηση ενός παράλληλου κώδικα απαιτεί μεγάλη προσοχή και έρευνα προκειμένου να πετύχουμε το βέλτιστο αποτέλεσμα.

Κεφάλαιο 3

HPCG (High Performance Conjugate Gradient)

3.1 Εισαγωγή

Το HPCG είναι ένα μια προσπάθεια να δημιουργηθεί μια πιο σχετική μέτρηση για την κατάταξη των συστημάτων HPC από το High Performance Linpack (HPL) το οποίο χρησιμοποιείται σήμερα από το TOP500 benchmark. Η λίστα των TOP500 είναι η πιο διαδεδομένη ευρέως σε ότι αφορά την κατάταξη των υπερυπολογιστών, με βάση τα Gflop/s που υπολογίζονται από το HPL. Η κατάταξη των ισχυρότερων υπολογιστών είναι πολύ σημαντική γιατί με αυτό τον τρόπο οι δημιουργοί αυτών των αρχιτεκτονικών θέλουν να βελτιστοποιήσουν όσο το δυνατό μπορούν την απόδοση των μηχανών τους και αυτός ο ανταγωνισμός επιφέρει και μεγάλα επιτεύγματα και γι' αυτό το λόγο οι κυβερνήσεις χρηματοδοτούν τεράστια πόσα προκειμένου να πάρουν μια θέση στο βάθρο.

Οι υπολογιστές υψηλών επιδόσεων έχουν αναδειχθεί ως ένα ισχυρό εργαλείο για την έρευνα και τη βιομηχανία. Γι' αυτό το λόγο η σύγκριση των υπερυπολογιστών έχει γίνει κεντρικό θέμα. Για το σκοπό αυτό έχει επιλεγεί αρχικά το HPL benchmark ως το μετρικό για να μπορεί να υπολογίζει τις αποδόσεις των μηχανών αυτών σε Gflop/s και να τα κατατάξει στην λίστα των TOP500. Το HPL ήταν ένα παράδειγμα ενός εξαιρετικά επεκτάσιμου προγράμματος MPI, βελτιστοποιημένο σε μεγάλο βαθμό για να μπορεί να αποσπά τη μέγιστη απόδοση από τις παράλληλες μηχανές. Σε συνδυασμό με την υπολογιστική ισχύ ενός δυνατού πυρήνα το HPL μπορούσε να σπρώξει μια μηχανή σχεδόν στο μέγιστο της απόδοσης της σε Gflop/s. Παρ' όλα αυτά, η πληθώρα των real-world εφαρμογών χαρακτηρίζουν πυρήνες οι οποίοι δεσμεύονται σε ένα πιο σύγχρονο hardware και με το HPL φτάνουν μόνο στο 20% της θεωρητικής τους απόδοσης. Αυτό είχε σαν αποτέλεσμα ασύμμετρες μετρήσεις και η κατάταξη βασιζόταν καθαρά σε μεγάλο βαθμό στις βελτιστοποιήσεις των υπολογισμών των εφαρμογών και όχι των μηχανών και αυτό έκανε την κατάταξη ασυνήθιστη με βάση τις real-word εφαρμογές.

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,503.17	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	27.78
2	3,631.86	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66
5	3,130.95	ROMEO HPC Center - Champagne-Ardenne	romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x	81.41
6	3,068.71	GSIC Center, Tokyo Institute of Technology	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x	922.54
7	2,702.16	University of Arizona	iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR14, NVIDIA K20x	53.62
8	2,629.10	Max-Planck-Gesellschaft MPI/IPP	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	269.94
9	2,629.10	Financial Institution	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	55.62
10	2,358.69	CSIRO	CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Infiniband FDR, Nvidia K20m	71.01

TOP500 Ranking November 2014

Η διαφορά στην απόδοση μεταξύ HPL και στις πραγματικές εφαρμογές άρχισε να μεγαλώνει τα τελευταία χρόνια και ο κυρίως λόγος ήταν η έντονη τάση ανάπτυξης των αρχιτεκτονικών· η διαθέσιμη υπολογιστική ισχύς αυξάνεται πολύ πιο γρήγορα από την ταχύτητα της μνήμης. Ως εκ τούτου, η μνήμη περιορίζει την διακίνηση των δεδομένων σε πυρήνες με μεγάλο αριθμό υπολογισμών. Παρ' όλα αυτά το HPL δεν επηρεάστηκε από αυτό το γεγονός λόγω του ότι η ποσότητα των δεδομένων που απαιτούνταν ανά υπολογισμό (Byte/Flop) ήταν σχετικά πολύ μικρή. Αυτό όμως δεν άλλαζε το γεγονός ότι οι real-world εφαρμογές απαιτούσαν επεξεργασία μεγάλων όγκων δεδομένων. Οι συζητήσεις για την ανάγκη ενός νέου μετρικού έγιναν όλο και πιο έντονες τα τελευταία χρόνια και αυτό είχε σαν αποτέλεσμα την δημιουργία HPCG

3.2 HPCG structure and routines

Ο κώδικας του HPCG είναι γραμμένος σε C++ και για την παραλληλοποίηση ο χρήστης μπορεί να διαλέξει MPI και/ή OpenMP κατά την διάρκεια της μεταγλώττισης. Το μέγεθος του προβλήματος και ο ελάχιστος χρόνος εκτέλεσης είναι καθορισμένο στο αρχείο εισόδου. Για να παράγουμε αποδεκτά αποτελέσματα, ο χρόνος εκτέλεσης πρέπει να είναι τουλάχιστο μία ώρα. Οι δημιουργοί του HPCG συστήνουν να αυξάνουμε το ελάχιστο μέγεθος του προβλήματος σε βαθμό που να επιτυγχάνουμε προσβάσεις σε χώρους μνήμης πέρα από την L3 cache για να παίρνουμε όσο πιο σωστά αποτελέσματα γίνεται και να έχουμε μια πιο σωστή εκτίμηση της απόδοσης της μηχανής [1].

Παρόλο που το benchmark είναι υλοποιημένο και σε MPI και σε OpenMP οι αποδόσεις του MPI είναι πολύ καλύτερες από το OpenMP και από το MPI μαζί με OpenMP. Αυτό συμβαίνει για τους εξής λόγους: Πρώτον ο HPCG αλγόριθμος είναι καλά προσαρτημένος σε MPI επίπεδο, και το χαρακτηριστικό της δυναμικής ισορρόπησης του OpenMP δεν μπορεί να προσφέρει κάποια βελτίωση στην απόδοση, και δεύτερον, τα νήματα του OpenMP παραμένουν αδρανείς κατά την διάρκεια της επικοινωνίας λόγο του fork/join μοντέλου της OpenMP το οποίο μειώνει κατά πολύ απόδοση.

Το HPCG είναι βασισμένο σε μια λύση συζυγιών, όπου ο προϋπολογισμός γίνεται σε μια ιεραρχική multi-grid (MG) μέθοδο. Ο αριθμός των επαναλήψεων ορίζεται σε 50 ανά σετ. Ο αλγόριθμος ξεκινά με την MG η οποία περιέχει μια συμμετρική Gauss-Seidel (SYMGS) και ένα αραιό matrix-vector multiplication (SpMV) για κάθε επίπεδο. Τα δεδομένα είναι κατανεμημένα μεταξύ των κόμβων, γι' αυτό και η SYMGS και SpMV ζητούν δεδομένα από τους γείτονες τους. Ο προκάτοχος τους, μια halos(ExchangeHalos) ρουτίνα, παρέχει αυτά τα δεδομένα στις δύο συναρτήσεις εκτελώντας επικοινωνία με τους γειτονικούς κόμβους. Η DotProduct (DDOT) τοπικά υπολογίζει το υπόλοιπο, καθώς η MPI_Allreduce ακολουθεί την DDOT και ολοκληρώνει μια γενική dot product επανάληψη. Η συνάρτηση WAXPBY ενημερώνει ένα vector με το άθροισμα δύο κλιμακούμενων διανυσμάτων.

Οι ρουτίνες SYMGS, SpMV, WAXPBY, DDOT είναι τα βασικά υπολογιστικά block του HPCG, καθώς οι ρουτίνες MPI_Allreduce και ExchangeHalos είναι τα βασικά block επικοινωνίας. Περισσότερες λεπτομέρειες για τις λειτουργίες του benchmark θα τις δούμε στο επόμενο υποκεφάλαιο.

```
for ( i = 0; i<50 && normr>err; i++ ){
  MG(A,r,z);
  DDOT( r ,t ,rtz );
  Allreduce ( rtz );

  if( i > 1 )
    beta = rtz/rtzold;
   WAXPBY( z, beta, p );

  ExchangeHalos( A, p);
  SpMV( A, p, Ap );
  DDOT ( p, Ap, pAp );
  Allreduce ( pAp);
  alpha =rtz/pAp;
  WAXPBY( x, alpha, p);
  WAXPBY( r, -alpha, Ap);
  DDOT( r, r, normr );
  Allreduce (normr);

  normr = sqrt( normr);
}
```

```
/*MG routine*/

if( depth <3){
      ExchangeHalos( );
      SYMGS( );
      ExchangeHalos( );
      SpMV( );
      MG( depth++ )
      ExchangeHalos( );
      SYMGS( );
}else{
      ExchangeHalos( );
      SYMGS( );
}
```

Ψευδοκώδικας του κυρίως βρόγχου

3.3 Μοντέλο ρουτινών HPCG

Ο χρόνος εκτέλεσης όλων των υπολογιστικών πυρήνων εξαρτάται από το μέγεθος του τρισδιάστατου αραιού πίνακα και τον αριθμό των μη-μηδενικών στοιχείων ανά τοπική γραμμή. Ο αριθμός των τοπικών γραμμών είναι ίσος με nx * ny *nz, καθώς ο αριθμός των μη μηδενικών στοιχείων σε μια γραμμή είναι 27 ή λιγότερα. Για μεγάλο μέγεθος προβλήματος θεωρούμε τον αριθμό των μη μηδενικών γραμμών ίσο 27. Δείγματα από υπολογιστικές ρουτίνες καλούνται άμεσα από τον κυρίως βρόγχο, καθώς η MG ρουτίνα καλεί αναδρομικά ένα σετ από υπολογιστικές ρουτίνες, μειώνοντας την ανάλυση ανά επίπεδο [2].

SYMGS Η Symmetric Gauss-Seidel Method είναι η ρουτίνα που παίρνει τον περισσότερο χρόνο στο benchmark (εκτός από την MG που είναι ένας συνδυασμός από ρουτίνες). Εκτελεί δύο βήματα: προς τα μπροστά και προς τα πίσω σαρώσεις. Με βάση το αποτύπωμα της μνήμης και τον αριθμό και το είδος των εργασιών, το δύο αυτά βήματα είναι πανομοιότυπα. Κάθε βήμα εκτελεί ένα δισδιάστατο βρόγχο, ο εξωτερικός αριθμός επαναλήψεων είναι LNI και ο εσωτερικός 27. Ο πυρήνας είναι βασισμένος πάνω σε ένα Flop με διπλή ακρίβεια, όπου ένας παράγοντας έχει έμμεση αντιμετώπιση. Ο διπλός βρόγχος φέρνει δύο double και ένα integer σε κάθε επανάληψη που κάνει συνολικά 20 Bytes. Η μονάδα επεξεργασίας φέρνει επίσης τους πίνακες c και d στην εξωτερική επανάληψη και ένα αριθμό μη μηδενικών στοιχείων, κάτι που αυξάνει το συνολικό μέγεθος των δεδομένων στην μνήμη. Ο χρόνος εκτέλεσης μοντελοποιείται με την διαίρεση των απαιτούμενων δεδομένων με αποτελεσματικό bandwidth που από την κύρια μνήμη.

WAXPB Η ρουτίνα WAXPB συμπεριφέρεται σαν ένας τριαδικός vector πυρήνας, που είναι το πιο σύνθετο σενάριο όλων των vector πυρήνων. Οι vectors a, b και c περιέχουν double στοιχεία, άρα χρειάζονται 24 Bytes από την μνήμη για κάθε επανάληψη. Ο αριθμός των επαναλήψεων είναι πάντα ο ίδιος για συγκεκριμένο μέγεθος εισόδου καθώς μόνο στο κυρίως loop καλείται απευθείας η μέθοδος αυτή. Ο χρόνος εκτέλεσης αυτής της ρουτίνας μοντελοποιείται με την πιο κάτω φόρμουλα:

$$executionWAXPB(sec) = \frac{LNI * 24(Bytes)}{BWeff(Bytes/sec)}$$

DDOT Αρχικά η DDOT ρουτίνα υπολογίζει τοπικά ένα γινόμενο πριν να υπολογίσει ένα γενικό άθροισμα μέσα στο σύστημα. Ενώ υπολογιστικά η WAXPB και η DDOT είναι εντελώς διαφορετικές, η πρόσβαση που κάνουν στην μνήμη είναι πολύ παρόμοια. Ωστόσο, η ρουτίνα DDOT απαιτεί μόνο 16 Bytes ανά επανάληψη σε σχέση με την WAXPB που απαιτούσε 24. Ο χρόνος εκτέλεσης της DDOT χωρίς επικοινωνία διαμορφώνεται ως εξής:

$$executionWAXPB(sec) = \frac{LNI * 24(Bytes)}{BWeff(Bytes/sec)}$$

3.4 Μοντέλο Επικοινωνίας HPCG

Ο αλγόριθμος του HPCG εκτελείται παράλληλα με τη χρήση MPI, το οποίο απαιτεί στατική κατανομή των δεδομένων μεταξύ των διαδικασιών με ξεχωριστούς χώρους διευθύνσεων. Η φυσική αποσύνθεση των δεδομένων είναι τριών διαστάσεων, λόγω του τρισδιάστατου αραιού πίνακα. Κάθε διαδικασία λαμβάνει το ίδιο μέγεθος εισόδου και ο αλγόριθμός δουλεύει σχεδόν τέλεια και ισορροπημένα με βάση το φόρτο εργασίας για κάθε διαδικασία (load balanced). Η επικοινωνία μεταξύ των διεργασιών χρησιμοποιεί τη διασύνδεση MPI και υπάρχουν δύο διαδικασίες επικοινωνίας: με την MPI Allreduce που ολοκληρώνει τη ρουτίνα DDOT και την halo που ανταλλάζει δεδομένα μεταξύ γειτονικών MPI διεργασιών [7].

Και οι δύο ρουτίνες χρησιμοποιούν MPI_COMM_WORLD για επικοινωνία. Δεν υπάρχει καμιά παρεμβολή της επικοινωνίας μεταξύ των διαφορετικών πληροφοριοδοτών, γεγονός που καθιστά τη δρομολόγηση της πληροφορίας εύκολη. Και οι δύο ρουτίνες χρησιμοποιούν το κλείδωμα MPI και η φύση του αλγόριθμου δεν

φέρει καμιά δυνατότητα για επικάλυψη της επικοινωνίας και του υπολογισμού. Η επικοινωνία συμπεριφέρεται σαν σημείο συγχρονισμού για όλες τις διεργασίες.

Η MPI_Allreduce είναι η μόνη συλλογική επικοινωνία που χρησιμοποιείται στον αλγόριθμο. Η λειτουργία της μειώνει μια μεταβλητή μεγέθους double μέσα σε όλες τις διεργασίες. Όπως συμβαίνει με όλες τις συλλογικές λειτουργίες, η υλοποίηση του MPI_Allreduce βασίζεται σε point-to-point επικοινωνία και η βέλτιστη εφαρμογή εξαρτάται από την τοπολογία του ίδιου του δικτύου. Ο αλγόριθμος hypercube επιτυγχάνει μείωση των βημάτων μεταξύ των διεργασιών σε log (N). Επίσης μειώνει την πληροφορία στον ελάχιστο αριθμό βημάτων που χρειάζονται και δείχνει κατά αυτό τον τρόπο την υψηλότερη απόδοση για συνηθισμένες τοπολογίες. Η ποσότητα των δεδομένων ανά βήμα επικοινωνίας είναι 8 byte, έτσι θεωρούμε την καθυστέρηση μεταξύ των διεργασιών ως η μόνη σημαντική παράμετρος του δικτύου και όχι το εύρος ζώνης (bandwidth) του μοντέλου. Για N MPI διεργασίες και μια λ δεδομένη καθυστέρηση μεταξύ της επικοινωνίας τους υπολογίζουμε το χρόνο εκτέλεσης όλων των λειτουργιών ως:

$$executionAllreduce(sec) = \sum_{i=1}^{M} l_i(log(M_i) - log(M_{i-1}))$$

Στην halo exchange γίνεται ανταλλαγή των δεδομένων με τον πλησιέστερο γείτονα. Είναι ένα συνηθισμένο πρότυπο επικοινωνίας για εφαρμογές MPI-HPC. Ο αριθμός των γειτόνων μιας δεδομένης διαδικασίας MPI εξαρτάται από τη θέση της μέσα στο πλέγμα. Αν το benchmark τρέχει σε 27 ή περισσότερες MPI διεργασίες τουλάχιστο μια από αυτές θα έχει 26 γείτονες στη φάση halo exchange. Το μέγιστο μέγεθος των δεδομένων που δέχεται η που στέλνει μια διεργασία κατά τη διάρκεια μιας halo exchange προκύπτει από την εξής φόρμουλα:

$$maxHaloSize(Bytes) = (2(nx * ny + nx * nz + ny * nz) + 4(nx + ny + nz) + 8) * 8Bytes$$

Η ρουτίνα MG καλεί την halo exchange από διαφορετικά επίπεδα, μειώνοντας το μέγεθος της halo. Η πιο κάτω εξίσωση δείχνει το χρόνο εκτέλεσης της ρουτίνας ExchangeHalos για διαφορετικά μεγέθη. Υποθέτουμε το ελάχιστο εύρος ζώνης (bandwidth) για την κίνηση δεδομένων σε όλο το δίκτυο μέσα από την halo exchange. Πρέπει να σημειώσουμε ότι η ανταλλαγή των δεδομένων της halo επιτυγχάνεται μέσω μιας αλληλουχίας MPI_Irec, MPI_Isend και MPI_wait. Με βάση αυτές τις παραμέτρους ο χρόνος εκτέλεσης της halo υπολογίζεται από την εξίσωση:

$$executionHaloEx(sec) = \frac{maxHaloSize}{IC_BW_{eff}} + 26(overhead(Irecv, Send, Wait)) + overhead(Rendezvousprotocol)$$

Τα overheads που υπάρχουν από την Irec, Isend και την Wait προσδιορίζονται άμεσα από την λανθάνουσα κατάσταση (latency), ενώ το overhead της Randezvousprotocol καθορίζεται από το pingpong MPI benchmark. Το τελευταίο μπορεί να αποφευχθεί εύκολα με την προσαρμογή της αντίστοιχης παραμέτρου παρόλα αυτά δεν εφαρμόζεται η συγκεκριμένη παράμετρος λόγω του ότι επηρεάζει αρκετά το μοντέλο.

3.5 Μοντελοποίηση ολόκληρου του benchmark

MG – Η MG ρουτίνα συνδυάζει πολλαπλές ρουτίνες τις οποίες καλεί από διάφορα επίπεδα. Το multi-grid επίπεδο μειώνει το μέγεθος του προβλήματος σε $2^{3*\text{επίπεδο}}$. Έτσι, όσο μεγαλύτερο είναι το επίπεδο τόσο πιο μικρό είναι το μέγεθος του προβλήματος άρα και λιγότερος ο χρόνος εκτέλεσης. Στην σάρωση της αναδρομής από μπροστά, η MG καλεί την ακολουθία HaloEx-SYMGS-HaloEx-SpMV μέχρι και το επίπεδο 2, ενώ στο επίπεδο 3 καλεί μόνο την HaloEx-SYMGS. Στην σάρωση προς τα πίσω η αναδρομή καλεί πάλι την HaloEx-SYMGS. Έτσι με βάση τα πιο πάνω ο συνολικός χρόνος εκτέλεσης της MG υπολογίζεται με την εξής φόρμουλα:

$$executionMG = HaloEx(depth = 3) + SYMGS(depth = 3) + \sum_{depth=0}^{2} (2 * SYMGS(depth) + SpMV(depth) + 3 * HaloEx(depth))$$

Συνολικός Χρόνος Εκτέλεσης: Ο κυρίως βρόγχος εκτελεί 50 επαναλήψεις καλώντας την ακολουθία MG-DDOT-WAXPB-SpMV-DDOT-WAXPB-WAXPB-DDOT. Η πρώτη επανάληψη καλεί ένα λιγότερο στιγμιότυπο της WAXPB ρουτίνας, άρα ο συνολικός χρόνος εκτέλεσης μιας επανάληψης μοντελοποιείται ως:

$$totalTime = MG + SpMV(depth = 0) + 3(DDOT + WAXPB)$$

Υπολογισμός των Gflop/s: Για τον υπολογισμό των Gflop/s, το HPCG προβλέπει τον αριθμό των πράξεων κινητής υποδιαστολής (floating point operations) ανά αναγκαία ρουτίνα και μετρά το χρόνο εκτέλεσης. Τα δεδομένα εισόδου και ο συνολικός αριθμός των μη μηδενικών στοιχείων καθορίζουν το συνολικό αριθμό των πράξεων κινητής υποδιαστολής (flop). Αν υποθέσουμε 27 μη μηδενικά στοιχεία ανά γραμμή για ένα μεγάλο μέγεθος εισόδου τότε ο συνολικός αριθμός των μη μηδενικών στοιχείων είναι: nmz = N * nx * ny * nz * 27. Ο συνολικός αριθμός των flops που προκύπτει είναι:

$$MG_{flop} = 10 * (nnz + nnz/8 + nnz/64) + 4 * (nnz/128)$$
$$SMPV_{flop} = 2 * nnz$$
$$DDOT_{flop} = 6 * N * nx * ny * nz$$

$$WAXPB_{flop} = 6 * N * nx * ny * nz$$

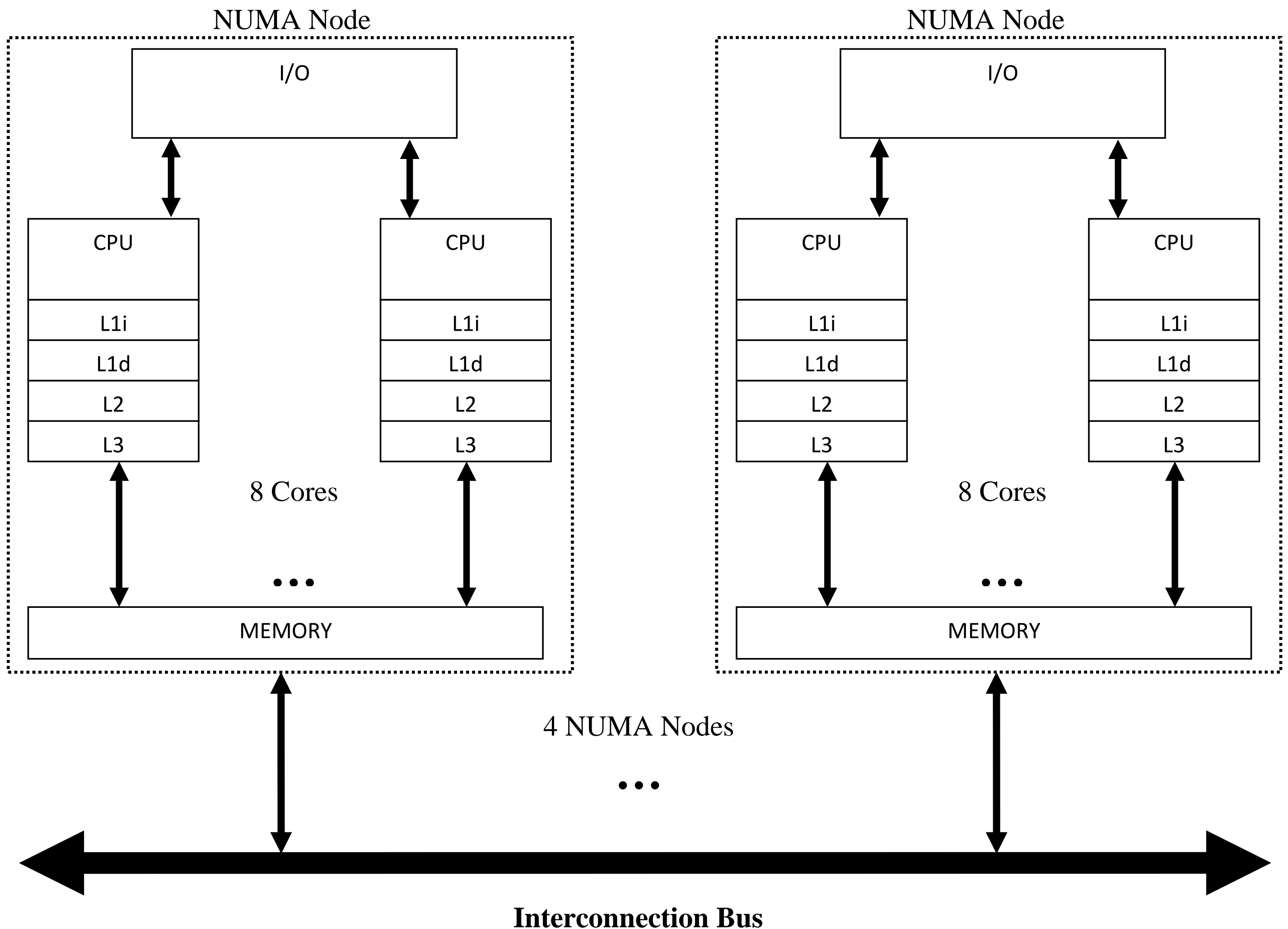
Σε συνδυασμό με την πρόβλεψη του χρόνου εκτέλεσης, και με αυτή τη φόρμουλα που μας επιτρέπει να προβλέψουμε την απόδοση σε Gflop/s, έχουμε μια ολοκληρωμένη εικόνα σε ότι αφορά την επίδοση του μοντέλου HPCG. Το μοντέλο είναι κατάλληλο για μεγάλα μεγέθη προβλημάτων (ανά διεργασία MPI), και είναι κατάλληλο και για πολύ μεγάλα συστήματα, κάτι που κάνει το HPCG το νέο αξιόπιστο benchmark για την λίστα των TOP500.

3.6 Αποτελέσματα και Επεξήγηση

Σε αυτό το υποκεφάλαιο θα παρουσιάσω τα αποτελέσματα που πήρα από τις δοκιμές του benchmark που έκανα στις μηχανές του εργαστηρίου cs8471 και cs8472. Πριν παρουσιάσω τα αποτελέσματα πρέπει πρώτα να δείξω την αρχιτεκτονική της μηχανής στην οποία έτρεχα το benchmark κάτι που θα βοηθήσει και στην επεξήγηση των αποτελεσμάτων.

Architecture:	x86_64	Stepping:	2
CPU op-mode(s):	32-bit, 64-bit	CPU MHz:	2300.000
Byte Order:	Little Endian	BogoMIPS:	4599.95
CPU(s):	32	Virtualization:	AMD-V
On-line CPU(s) list:	0-31	L1d cache:	16K
Thread(s) per core:	2	L1i cache:	64K
Core(s) per socket:	8	L2 cache:	2048K
Socket(s):	2	L3 cache:	6144K
NUMA node(s):	4	NUMA node0 CPU(s):	0-7
Vendor ID:	AuthenticAMD	NUMA node1 CPU(s):	8-15
CPU family:	21	NUMA node2 CPU(s):	16-23
Model:	1	NUMA node3 CPU(s):	24-31

Πιο πάνω παρουσιάζονται τα χαρακτηριστικά της μηχανής η οποία όπως φαίνεται διαθέτει 4 NUMA nodes με 8 64-bit CPUs ο κάθε ένας. Υπάρχουν 4 επίπεδα κρυφής μνήμης: η L1d με 16K, η L1i με 64K, η L2 με 2048 K και η L3 με 6144K. Πιο κάτω παρουσιάζω γραφικά την αρχιτεκτονική της μηχανής.



Οι Non-Uniform Memory Access αρχιτεκτονικές είναι ένα σχέδιο μνήμης του υπολογιστή που χρησιμοποιείται στην πολυεπεξεργασία, όπου ο χρόνος πρόσβασης στην μνήμη εξαρτάται από τη θέση της μνήμης σε σχέση με τον επεξεργαστή. Ένας επεξεργαστής μπορεί να έχει πρόσβαση στη δική του τοπική μνήμη ταχύτερα από ό, τι στη τοπική μνήμη ενός άλλου επεξεργαστή ή σε μνήμη που μοιράζονται όλοι επεξεργαστές. Τα πλεονεκτήματα αυτής της αρχιτεκτονικής περιορίζονται για συγκεκριμένο φόρτο εργασίας ιδικά σε servers όπου οι λειτουργίες και τα καθήκοντα προσδιορίζονται αυστηρά σε συγκεκριμένους επεξεργαστές.

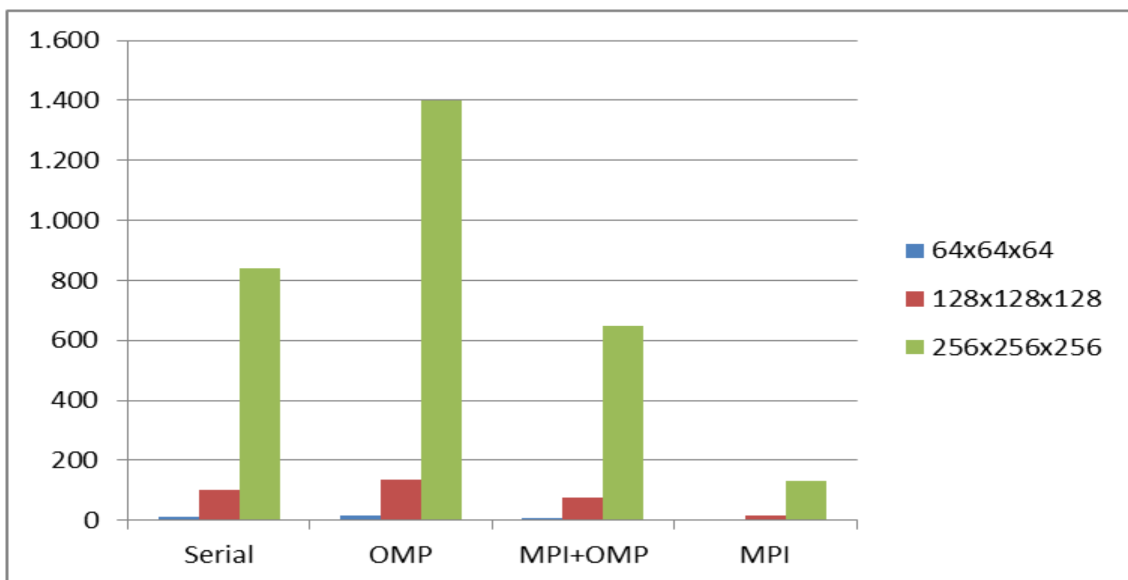
Αποτελέσματα της απόδοσης της μηχανής με βάση το HPCG

Υπάρχουν τέσσερις τρόποι να τρέξουμε το benchmark:

- Serial
- OpenMP
- MPI + OpenMP
- MPI

Για τους τρείς παράλληλους τρόπους θα παρουσιάσω τα αποτελέσματα για sizes 64, 128, 256 μόνο για το λόγο ότι για πιο μικρά sizes το speedup τους σε σχέση με το σειριακό μοντέλο είναι ελάχιστο λόγο των overheads που αντιμετωπίζουν αυτές οι τεχνικές παράλληλου προγραμματισμού. Για μεγαλύτερα sizes υπάρχει πρόβλημα υπερχείλισης της κύριας μνήμης και το πρόγραμμα δεν τελειώνει.

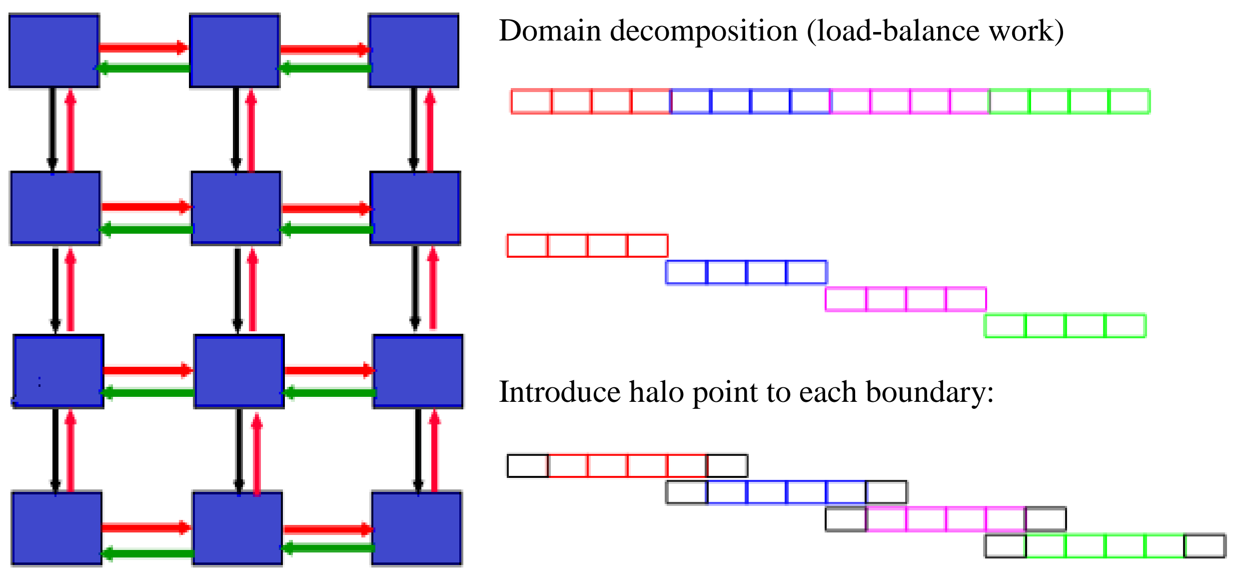
Γραφική Χρόνου εκτέλεσης του benchmark σε δευτερόλεπτα



Αυτό που παρατηρούμε από την γραφική είναι ότι ο χρόνος εκτέλεσης με MPI είναι πολύ πιο μικρός σε σχέση με το σειριακό και το OpenMP. Από την άλλη ο χρόνος εκτέλεσης του OpenMP είναι πολύ πιο μεγάλος από το MPI ακόμη και από το σειριακό. Αυτό συμβαίνει για το λόγο ότι υπάρχουν πολλές αναφορές στην μνήμη. Η μνήμη του κάθε επεξεργαστή είναι share memory και όταν τα νήματα πηγαίνουν να διαβάζουν από

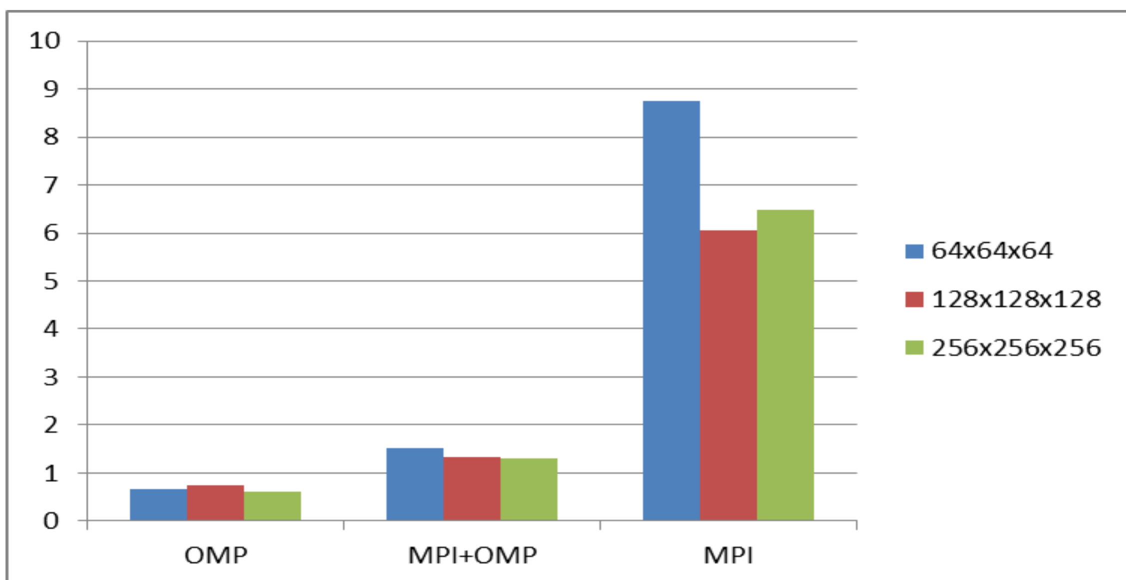
την μνήμη δημιουργείται υπάρχει συνωστισμός στο σημείο πρόσβασης (bottleneck). Επίσης μελετώντας το συμβαίνει στην μνήμη κατά την εκτέλεση του benchmark παρατήρησα ότι υπάρχουν αρκετές αναφορές στην μνήμη. Αυτό λόγω του ότι είναι share memory το μοντέλο της μηχανής υπάρχει το φαινόμενο του bottleneck καθώς επίσης με βάση τον αριθμό προσβάσεων στην μνήμη ο αριθμός των cache misses είναι πολύ μεγάλος.

Ο λόγος για τον οποίο ο χρόνος εκτέλεσης του MPI είναι πολύ πιο γρήγορο είναι ότι το benchmark σχεδιάστηκε για distributed συστήματα και το γεγονός ότι το MPI χρησιμοποιεί την συνάρτηση exchangehalo. Πολλές εφαρμογές σε High Performance Computing χρησιμοποιούν την αποσύνθεση τομέα (domain decomposition) για να κάνουν κατανομή της εργασίας μεταξύ διαφόρων στοιχείων επεξεργασίας (processing elements). Για την διαχείριση του συγχρονισμού των overheads, οι αποσυνθεμένοι υπο-τομείς (sub-domains) υπερκαλύπτονται στα όρια και ενημερώνονται από τις τιμές των γειτόνων τους πριν από τον υπολογισμό των εσωτερικών πράξεων.



Ουσιαστικά κάθε διεργασία ανταλλάσει δεδομένα με το πιο κοντινό της γείτονα. Γίνεται η διάσπαση των δεδομένων στις διεργασίες και μεταξύ τους οι διεργασίες ανταλλάσουν δεδομένα κατά την διάρκεια υπολογισμού των πράξεων πριν ακόμη γίνει όλη η επεξεργασία των δεδομένων και υπολογιστεί το τελικό αποτέλεσμα για κάθε μια από αυτές.

Γραφική Επιτάχυνσης κάθε περίπτωσης σε σχέση με το σειριακό χρόνο

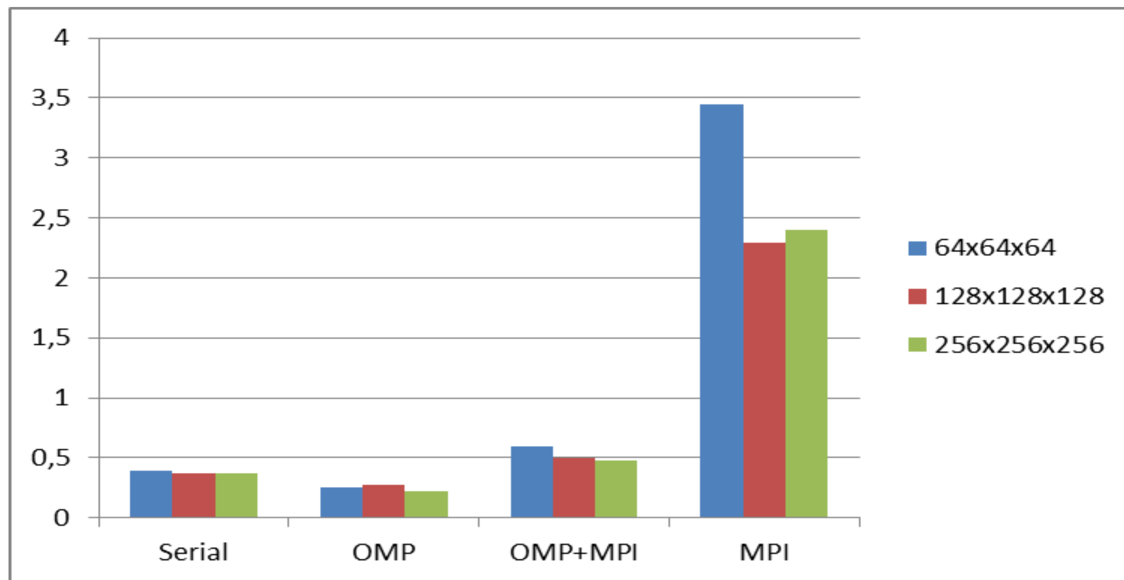


Η πιο πάνω γραφική παρουσιάζει την επιτάχυνση του OMP, του MPI+OMP και του MPI σε σχέση με το σειριακό χρόνο εκτέλεσης για κάθε μέγεθος. Ο υπολογισμός της επιτάχυνσης γίνεται με την διαίρεση του σειριακού χρόνου εκτέλεσης με το παράλληλο χρόνο εκτέλεσης.

$$\text{SpeedUp} = \frac{\text{Tsequential}}{\text{Tparallel}}$$

Η επιτάχυνση του OMP είναι κάτω από το ένα (1) και στις τρεις περιπτώσεις για το λόγο ότι ο χρόνος εκτέλεσης του είναι μεγαλύτερος από του σειριακού. Με MPI και OMP υπάρχει μια ελάχιστη επιτάχυνση λίγο πιο πάνω από ένα, ενώ αισθητά πιο μεγάλη είναι η επιτάχυνση στο MPI μόνο και είναι γύρω στο οκτώ (8) και για τις τρείς περιπτώσεις. Αυτό συμβαίνει γιατί σε κάθε node (2 nodes μόνο) υπάρχουν 8 processors στους οποίους κατανέμονται τα δεδομένα προς επεξεργασία [3].

Γραφική παράσταση των GFLOP/s για κάθε περίπτωση

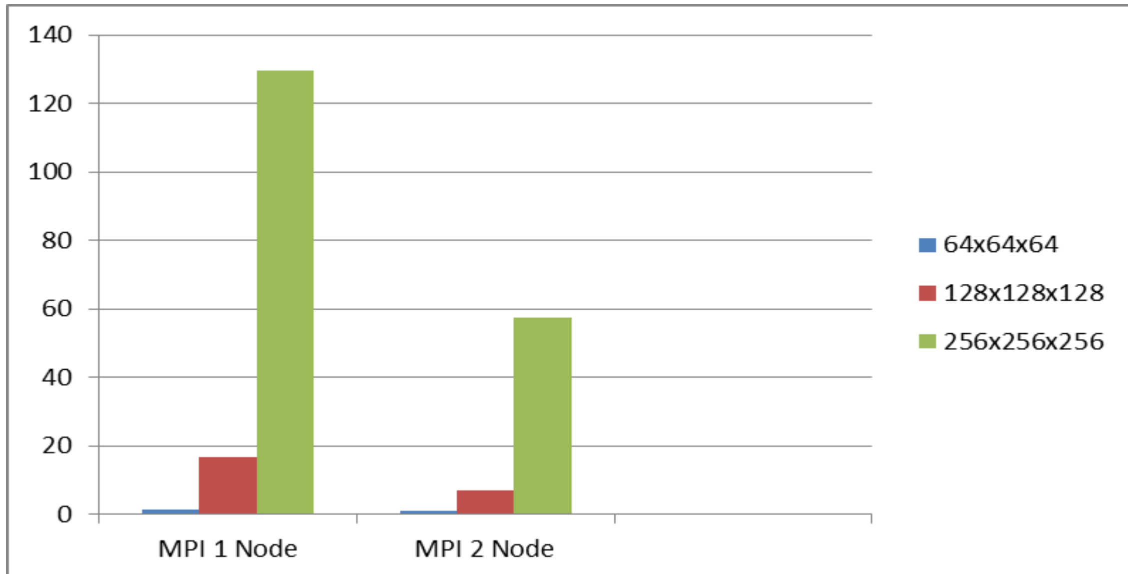


Η απόδοση των cluster μηχανών όπως αναφέραμε και στην αρχή του κεφαλαίου μετριέται ανάλογα με το πόσα GFLOPs εκτελούν ανά δευτερόλεπτο. Αυτός είναι και ο τρόπος με τον οποίο κατατάσσονται οι κορυφαίες μηχανές στην λίστα των TOP500. Είναι σημαντικό λοιπόν να αναφερθούμε και στην απόδοση της μηχανής στην οποία τρέξαμε το benchmark με βάση αυτό το μετρικό.

Το OMP εκτελεί περίπου 0.25 GFLOPs ($0.25 * 10^9$ Floating Point Operations) το δευτερόλεπτο λιγότερο από τη σειριακή εκτέλεση που είναι γύρω στα 0.3, ενώ στην καλύτερη περίπτωση πετύχαμε σχεδόν 3.5 GFLOPs το δευτερόλεπτο με μόνο MPI για μέγεθος 64. Με την χρήση OMP και MPI είχαμε μέγιστη απόδοση γύρω στα 0.6 GFLOPs το δευτερόλεπτο στην περίπτωση όπου το μέγεθος είναι 64.

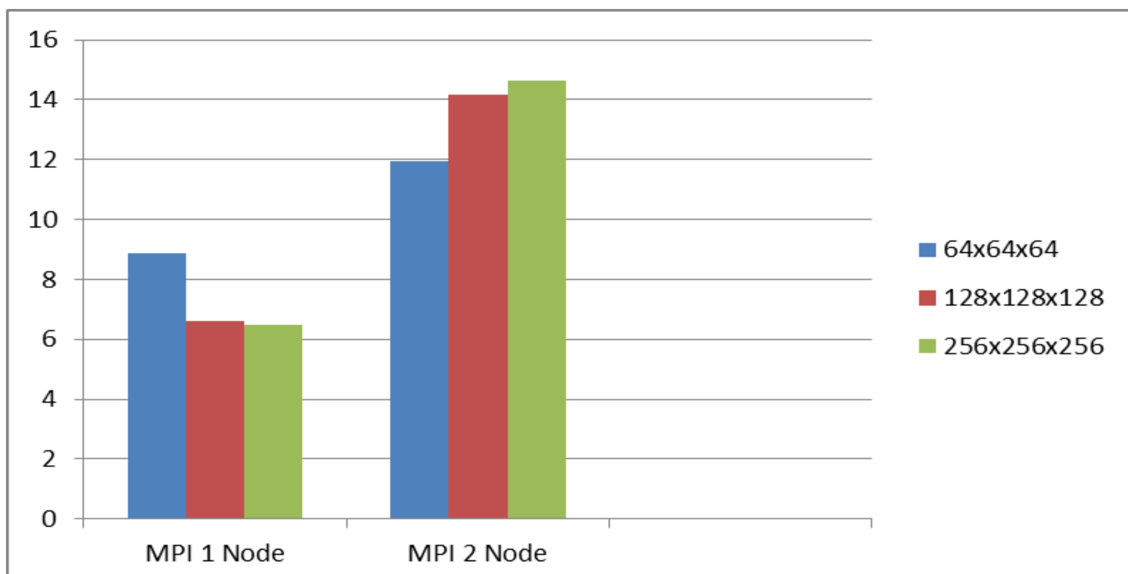
Οι πιο κάτω γραφικές παρουσιάζουν τα αποτελέσματα που πήρα όταν έτρεξα το benchmark σε δύο μηχανές (cs8472 και cs8471) με την ίδια αρχιτεκτονική.

Χρόνος Εκτέλεσης σε δευτερόλεπτα

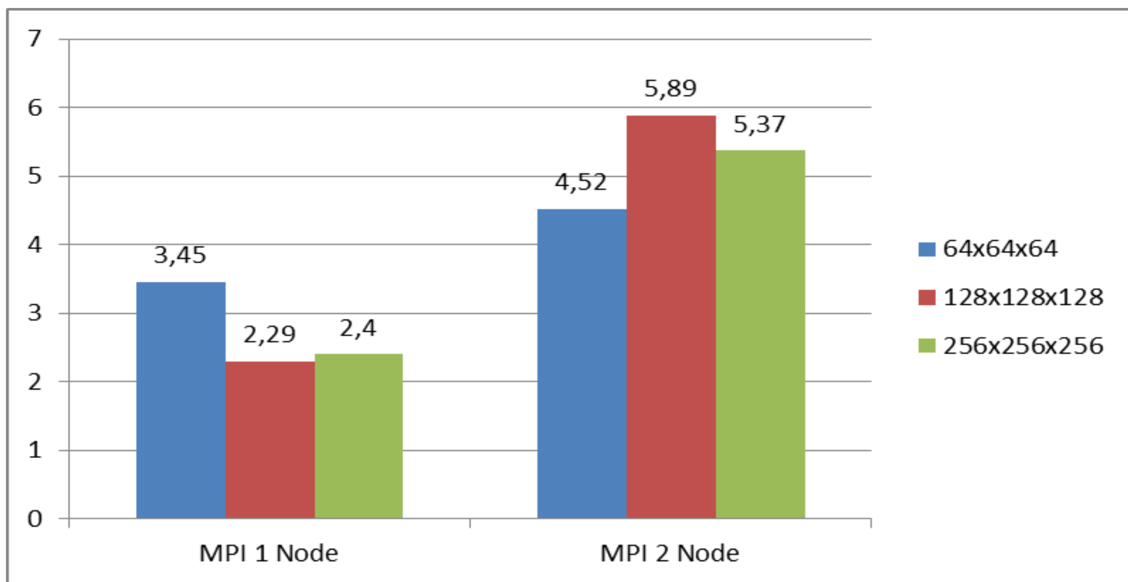


Όπως ήταν αναμενόμενο ο χρόνος εκτέλεσης του benchmark όταν το τρέχουμε σε δύο μηχανές είναι περίπου στο μισό. Τα δεδομένα διασπούνται στις δύο μηχανές οι οποίες εκτελούν με τον ίδιο τρόπο όπως και πριν τις λειτουργίες του benchmark.

Επιτάχυνσης (SpeedUp) σε σχέση με το σειριακό



GFLOPs το δευτερόλεπτο



Παρόμοια αποτελέσματα βλέπουμε και στις γραφικές της επιτάχυνσης και των GFLOPs. Παρατηρούμε επίσης ότι για μέγεθος 64 δεν υπάρχει διπλάσια απόδοση και αυτό λόγω του ότι το μέγεθος του προβλήματος είναι σχετικά μικρό και το κόστος επικοινωνίας αλλά και διάσπασης των δεδομένων στους δύο nodes είναι αρκετό ώστε να υπάρχει λιγότερη απόδοση από την αναμενόμενη.

Κεφάλαιο 4

Thread Building Block (TBB Intel)

4.1 Εισαγωγή 70

4.2 Γιατί εργασίες και όχι νήματα 73

4.3 Benchmarks of TBB 70

4.4 HPCG with TBB 75

4.1 Εισαγωγή

Το TBB είναι μια βιβλιοθήκη στο πρότυπο της C++ γλώσσας προγραμματισμού που αναπτύχθηκε από την εταιρεία Intel για σύνταξη προγραμμάτων λογισμικού που εκμεταλλεύονται στο μέγιστο τα συστήματα των multicore-processors. Η βιβλιοθήκη αποτελείται από δομές δεδομένων και αλγόριθμους που επιτρέπουν σε έναν προγραμματιστή να αποφύγει κάποιες επιπλοκές και δυσκολίες που μπορεί να προκύπτουν από την χρήση των πακέτων για νήματα (threads), όπου η δημιουργία τους ο συγχρονισμός τους και ο τερματισμός σε πολλές περιπτώσεις πρέπει να γίνεται από τον προγραμματιστή. Αντί αυτού οι βιβλιοθήκη αποσπά πρόσβαση στους πολλαπλούς επεξεργαστές επιτρέποντας στις λειτουργίες να αντιμετωπίζονται από "tasks" τα οποία κατανέμονται σε επιμέρους πυρήνες δυναμικά από τη μηχανή χρόνου της βιβλιοθήκης, και αυτοματοποιώντας την αποδοτική χρήση της κρυφής μνήμης στον επεξεργαστή. Ένα πρόγραμμα TBB δημιουργεί, συγχρονίζει και καταστρέφει γραφήματα εξαρτημένων εργασιών με βάση κάποιους αλγόριθμους. Οι εργασίες (tasks) στη συνέχεια εκτελούν με βάση τις εξαρτήσεις που παρουσιάζονται στα γραφήματα αυτά. Αυτή η προσέγγιση μαζί με μια πληθώρα επιπρόσθετων λύσεων των δυσκολιών του παράλληλου προγραμματισμού, η βιβλιοθήκη της TBB αποσυνδέει τον προγραμματιστή από τα στοιχεία του σχεδιασμού της μηχανής και κάνει τον παράλληλο προγραμματισμό πιο εύκολο [5].

Η TBB υλοποιεί την τεχνική work-stealing για να εξισορροπήσει τον παράλληλο φόρτο εργασίας μεταξύ των διαθέσιμων πυρήνων με σκοπό την μέγιστη χρήση του πυρήνα, ως εκ τούτου και τις συνολικής απόδοσης. Αρχικά ο φόρτος εργασίας είναι χωρισμένος στους διαθέσιμους πυρήνες. Αν ένας πυρήνας ολοκληρώσει τις εργασίες του ενώ οι άλλοι εξακολουθούν να έχουν ένα σημαντικό πόσο εργασίας στην ουρά τους, με την τεχνική αυτή εκχωρεί εκ νέου ορισμένες από τις εργασίες του απασχολημένου πυρήνα στην ουρά του πυρήνα που βρίσκεται σε αδράνεια. Αυτή η δυναμική ικανότητα αποσυνδέει τον προγραμματιστή από τον έλεγχο της μηχανής επιτρέποντας εφαρμογές που έχουν γραφτεί χρησιμοποιώντας τις βιβλιοθήκες για να βελτιστοποιήσει στο μέγιστο την χρήση των διαθέσιμων πυρήνων, χωρίς να χρειάζεται να κάνει αλλαγές στον πηγαίο κώδικα του προγράμματος ή στο εκτελέσιμο αρχείο του.

4.2 Γιατί εργασίες και όχι νήματα

Οι εργασίες είναι μια πιο "ελαφριά" εναλλακτική λύση για τα νήματα, προσφέροντας ταχύτερη εκκίνηση και τερματισμό, καλύτερη εξισορρόπηση φόρτου εργασίας και χρήσης των διαθέσιμων πόρων. Πιο κάτω θα παρουσιάσω μια γενική επισκόπηση προγραμματισμού με εργασίες και ορισμένες κατευθυντήριες γραμμές για να μπορεί κάποιος να αποφασίσει κατά ποσό πρέπει να χρησιμοποιήσει νήματα ή εργασίες και σε ποιες περιπτώσεις. Είναι σημαντικό για να μπορέσουμε να έχουν μια γενική εικόνα για το πότε οι TBB βιβλιοθήκες είναι αποδοτικές και πότε όχι.

Ο προγραμματισμός με νήματα είναι συχνά μια κακή επιλογή σε περιπτώσεις πολυνηματικού προγραμματισμού. Τα νήματα που δημιουργούνται από τον προγραμματιστή είναι λογικά νήματα τα οποία συνδέονται από το λογισμικό σύστημα πάνω σε φυσικά νήματα του hardware. Όπως αναφέραμε και στο Κεφάλαιο του Multicore Programming η δημιουργία πολύ λίγων λογικών νημάτων προκαλεί μείωση της απόδοσης του συστήματος σπαταλώντας αρκετούς από τους διαθέσιμους πόρους του hardware, ενώ σε αντίθετη περίπτωση που δημιουργούμε πολλά λογικά νήματα υπερφορτώνουν το σύστημα καθώς χρειάζεται αρκετός χρόνος για να έχουν πρόσβαση στους πόρους του. Με την χρήση νημάτων απευθείας, ο προγραμματιστής γίνεται υπεύθυνος στο να ταιριάξει τον παραλληλισμό του προγράμματος του με της πηγές του συστήματος κάτι που κάνει τον προγραμματισμό πολύ πιο περίπλοκο.

Ένας τρόπος για να αντιμετωπίσεις αυτή τη δυσκολία είναι να δημιουργήσεις έτοιμα νήματα (thread pool) τα οποία χρησιμοποιούνται κατά διάρκεια εκτέλεσης ολόκληρου του προγράμματος. Τυπικά ένα λογικό (software) νήμα δημιουργείται για κάθε φυσικό (hardware) νήμα. Το πρόγραμμα στη συνέχεια προγραμματίζει δυναμικά τους υπολογισμούς στα νήματα του thread pool. Χρησιμοποιώντας αυτή την τεχνική βοηθά στο να γίνεται αποδοτική η παράλληλη σύνδεση με τους πόρους του hardware και ταυτόχρονα βοηθά στο να αποφεύγονται τα overheads από την συνεχή δημιουργία και καταστροφή των νημάτων.

Το προγραμματιστικό μοντέλο του TBB παρέχει δυναμισμούς με τα πλεονεκτήματα της τεχνικής του thread pool χωρίς όμως την ανάγκης πλήρους διαχείρισης του. Με αυτό το μοντέλο, οι προγραμματιστές εκφράζουν το λογικό προγραμματισμό στο κώδικά τους με την χρήση των εργασιών (tasks) και με την βοήθεια της βιβλιοθήκης runtime η οποία δρομολογεί τις εργασίες μέσα στα δικά του αίτημα νήματα (worker thread pool). Η χρήση των εργασιών διευκολύνει τους προγραμματιστές να συγκεντρώνονται στο λογικό παραλληλισμό του προγράμματος τους χωρίς να ανησυχούν για την διαχείριση του παραλληλισμού. Επίσης, όντας ελαφρύτερες οι εργασίες από τα νήματα είναι δυνατό να εκφραστεί ο παραλληλισμός σε πιο διακριτικό επίπεδο.

Ένα άλλο πλεονέκτημα του TBB όπως αναφέραμε και στην εισαγωγή του κεφαλαίου είναι και διαχείριση των εργασιών με τρόπο ώστε να δουλεύουν με βάση την διαμοίραση φόρτου εργασίας (work-stealing). Σε αυτό το μηχανισμό, κάθε νήμα στο thread pool διατηρεί ένα τοπικό αριθμό από εργασίες (task pool) που είναι οργανωμένες σαν μια double-ended ουρά. Το νήμα χρησιμοποιεί το δικό του task pool σαν μια στοίβα, βάζοντας νέες εργασίες που περιμένουν έτοιμες στην κορυφή της στοίβας. Η εργασία στην κορυφή της στοίβας είναι η πιο καινούργια και γι' αυτό το λόγο είναι και αυτή με την μεγαλύτερη πιθανότητα να πιάσει δεδομένα που βρίσκονται στην κρυφή του μνήμη. Εάν δεν υπάρχουν εργασίες τοπικά στο task pool του νήματος, προσπαθεί να "κλέψει" δουλειά από ένα άλλο νήμα (victim). Κατά την διαδικασία αυτή το νήμα χρησιμοποιεί την ουρά του νήματος από το οποίο θέλει να πάρει δουλειά και "κλέβει" την πιο παλιά του εργασία. Για αναδρομικούς αλγόριθμους, αυτές οι παλιές εργασίες είναι κόμβοι που βρίσκονται ψηλά στο task tree και γι' αυτό υπάρχουν πολλά κομμάτια

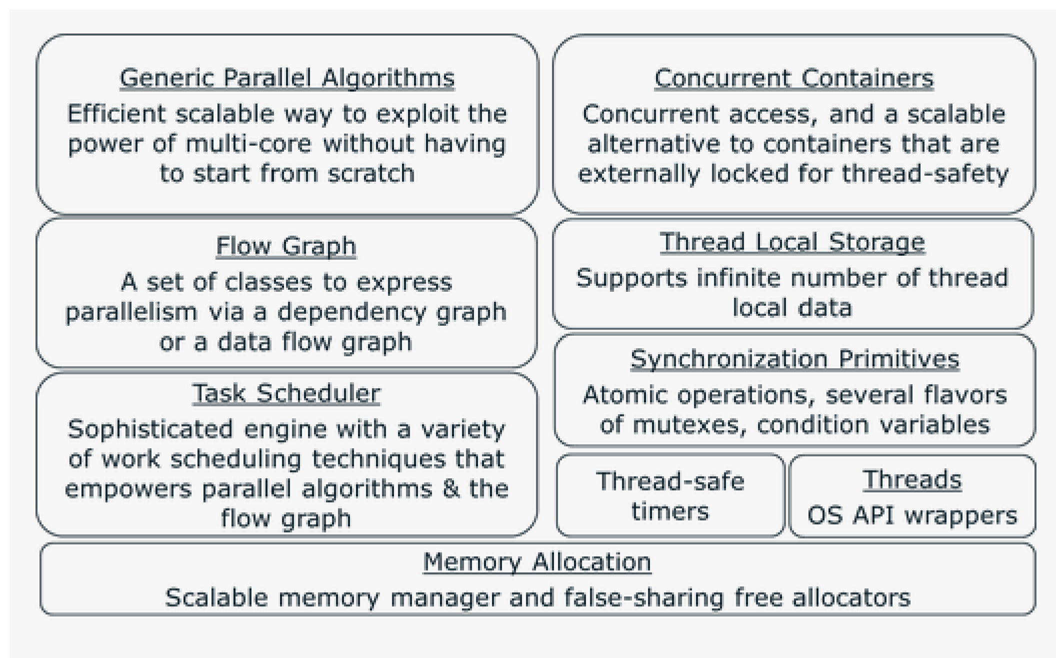
υπολογισμού σε αυτές, δουλεία η οποία δεν βρίσκεται στην κρυφή μνήμη του νήματος από το οποίο θέλει κλέψει. Αυτό κάνει τον μηχανισμό αυτό να είναι αποδοτικός για εξισορρόπηση του φόρτου εργασίας διατηρώντας παράλληλα την τοπικότητα της κρυφή μνήμης.

Η χρήση των εργασιών είναι συνήθως μια καλή προσέγγιση για να προσθέσεις προγραμματισμό με νήματα για να βελτιώσεις την απόδοση. Ο προγραμματισμός των εργασιών του TBB δεν είναι προληπτικός. Οι εργασίες προορίζονται στο να βελτιώνουν την απόδοση ενός προγράμματος σε αλγόριθμους που είναι non-blocking και υπάρχει μείωση της απόδοσης γιατί όταν ένα task μπλοκάρεται συχνά το νήμα στο οποίο ανήκει το task δεν μπορεί να δουλέψει σε οποιαδήποτε άλλα tasks. Μπλοκαρίσματα εμφανίζονται συχνά όταν περιμένει το πρόγραμμα κάποια είσοδο ή σε περιπτώσεις όπου υπάρχουν mutexes για μεγάλη διάρκεια. Όταν ένα νήμα περιμένει σε mutex για αρκετό χρόνο, ο κώδικας δεν λειτουργεί αποδοτικά βάση των αριθμών των νημάτων που περιμένουν. Αντί για blocking tasks, είναι καλύτερο να χρησιμοποιούμε νήματα αντί αυτών.

Σε περιπτώσεις όπου η χρήση των tasks είναι καλύτερη, δεν είναι αναγκαίο για τον προγραμματιστή να υλοποιήσει τα πρότυπα αυτά από την αρχή. Η βιβλιοθήκη της Intel παρέχει μια διεπαφή και ψηλού επιπέδου αλγόριθμους που υλοποιούν κάποιους από τους πιο σημαντικούς μηχανισμούς όπως το parallel_invoke, το parallel_for και το parallel_reduce οι οποίοι μπορούν αν χρησιμοποιηθούν από τον προγραμματιστή με σιγουριά όποτε είναι δυνατόν για να αποφεύγεται οποιαδήποτε δυσκολίας υλοποιήσεις τους από τον ίδιο.

4.3 Λειτουργίες βιβλιοθηκών TBB

Όπως αναφέραμε στην αρχή του κεφαλαίου η βιβλιοθήκη TBB παρέχει στους προγραμματιστές την δυνατότητα και την διευκόλυνση να γράφουν παράλληλα προγράμματα σε γλώσσα C++. Το γνωστό πλεονέκτημα της βιβλιοθήκης είναι ότι καθιστά την παράλληλη απόδοση και επεκτασιμότητα εύκολα προσβάσιμη από τους προγραμματιστές γράφοντας εφαρμογές με την χρήση των tasks. Η βιβλιοθήκη περιλαμβάνει μια σειρά γενικών παράλληλων αλγορίθμων που θα δούμε και θα αναλύσουμε πιο κάτω.



Πίνακας λειτουργιών της βιβλιοθήκης TBB

Generic Parallel Algorithms

Η βιβλιοθήκη παρέχει τη συνάρτηση parallel_for μέσω της οποίας μπορούμε να παραλληλοποιήσουμε ένα βρόγχο ο οποίος έχει την δυνατότητα να παραλληλοποιηθεί.

```
void SerialApplyFoo (float a[], size_t n){
        for (size_t i=0; i!=n ++i)
                Foo (a[i]);
}
```

Το πιο πάνω πρόγραμμα μπορεί να παραλληλοποιηθεί με την χρήση της συνάρτησης parallel_for η οποία μοιράζει τις επαναλήψεις σε tasks τα οποία προωθεί στην βιβλιοθήκη task scheduler για παράλληλη εκτέλεση. Επομένως η πιο πάνω συνάρτηση μπορεί να παραλληλοποιηθεί με τον εξής τρόπο:

```
#inlude “tbb/tbb.h”
using namespace tbb;
void ParallelApplyFoo (float a[], size_t n){
        parallel_for (0, n, [&] (int i) {
                Foo (a[i]);
        });
}
```

Αυτός είναι ο πιο απλός τρόπος της χρήσης του parallel_for. Υπάρχουν πολλές πιο περίπλοκες χρήσεις του σε πιο πολύπλοκα προβλήματα.

Concurrent Containers

Τα Concurrent Containers επιτρέπουν σε πολλαπλά νήματα την ταυτόχρονη πρόσβαση και ενημέρωση των στοιχείων σε δομές που διαθέτουν περισσότερα από ένα στοιχεία. Συνήθως αυτές οι δομές δεν επιτρέπουν ταυτόχρονη αλλαγή στα στοιχεία τους καθώς αυτό μπορεί να διαφθείρει το περιεχόμενο τους λόγο των προβλημάτων που προκύπτουν από τον παράλληλο προγραμματισμό. Για να επιτευχθεί αυτό με ασφάλεια γίνεται ανάγκη της χρήσης των mutex που μπλοκάρουν το σημείο στο οποίο γίνεται η ενημέρωση επιτρέποντας μόνο ένα νήμα να εισέλθει στο κομμάτι αυτό του κώδικα, όμως αυτό έχει σαν αποτέλεσμα την δημιουργία μιας σειριακής εκτέλεσης κάτι που περιορίζει κατά πολύ την παράλληλη επιτάχυνση.

Οι containers που παρέχονται από το TBB προσφέρουν πολύ πιο ψηλότερο επίπεδο συγχρονισμού, με τη χρήση το δύο πιο κάτω μεθόδων:

- **Fine-grained locking**: Πολλά νήματα εκτελούν αυτή την μέθοδο μπλοκάροντας μόνο στις περιπτώσεις στις οποίες όντως υπάρχει ανάγκη να μπλοκαριστούν. Υπάρχουν περιπτώσεις όπου διαφορετικά νήματα έχουν πρόσβαση σε διαφορετικές δομές και ο συγχρονισμός είναι επιτρεπτός άρα δεν χρειάζονται όλα τα νήματα να γίνονται μπλοκ.
- **Lock-free techniques**: Είναι η μέθοδος κατά την οποία επιπρόσθετα νήματα μετρούν και διορθώνουν αρνητικές επιδράσεις από τειχών μη επιτρεπτούς συγχρονισμούς.

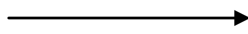
Πιο κάτω φαίνεται ένα παράδειγμα όπου χωρίς τη χρήση αυτών των μεθόδων για να ενημερωθεί η ουρά χρειάζεται να εκτελεστούν τρείς εντολές. Λόγω του ότι τα νήματα μπαινοβγαίνουν ακανόνιστα στο στον επεξεργαστή για να εκτελέσουν την λειτουργία τους, αυτός ο τρόπος είναι επικίνδυνος και μπορεί να διαφθείρει το περιεχόμενο της ουράς και έτσι καλείται η ανάγκη των mutex. Με το TBB όμως μπορούμε να εκτελέσουμε αυτές τις εντολές με τη χρήση μιας, η οποία ταυτόχρονα ελέγχει και εγγυάται ότι δεν θα γίνει κάποιο λάθος αποφεύγοντας με αυτό τον τρόπο τα mutex που μειώνουν την απόδοση του προγράμματος.

Χωρίς τη χρήση TB:

```
extern std::queue<T> MySerialQueue;
T item;
If (!MySerialQueue.empty()){            }
        item=MySerialQueue.front();     }   Τρείς εντολές για την έξοδο
        MySerialQueue.pop_front();      }   στοιχείου από την ουρά
        ….process item….
}
```

Με τη χρήση TBB:

```
extern concurrent_queue<T> MyQueue;
T item;
If (MyQueue.try_pop(item)){
        ….process item….
}
```

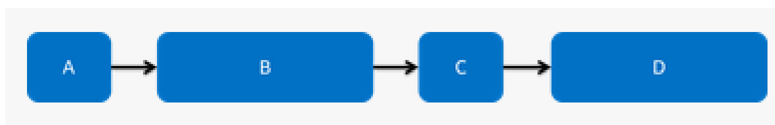


Έλεγχος και έξοδος με μια εντολή

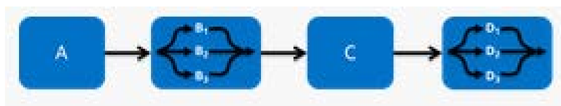
Flow Graphs

Τα Flow Graphs που παρέχει το TBB επεκτείνουν τις δυνατότητες του προγράμματος επιτρέποντας μια πιο γρήγορη και αποτελεσματική υλοποίηση με την χρήση γράφων που δηλώνουν τις εξαρτήσεις μεταξύ των λειτουργιών ενός κώδικα. Ο προγραμματιστής μπορεί να δηλώσει ρητά τις εξαρτήσεις μεταξύ των λειτουργιών με την δημιουργία ενός γράφου εξαρτήσεων. Όπου οι λειτουργίες είναι ανεξάρτητες μεταξύ τους δίνει την δυνατότητα σε αυτές να εκτελούνται ταυτόχρονα. Πιο κάτω φαίνεται ένα παράδειγμα γράφου.

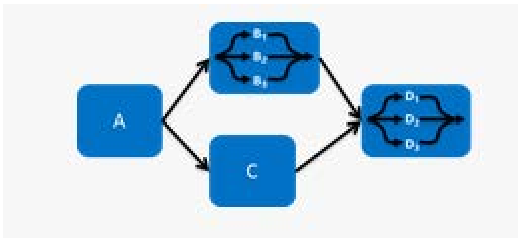
Έστω το πιο κάτω σειριακό πρόγραμμα με τις λειτουργίες A,B,C,D.



Οι λειτουργίες B, D μπορούν να εκτελέσουν των κώδικα τους παράλληλα με τη χρήση του parallel_for.



Αυτό όμως δεν είναι η βέλτιστη σχεδίαση που μπορούμε να κάνουμε αν γνωρίζουμε ότι οι λειτουργίες B και C μπορούν να εκτελεστούν παράλληλα καθώς δεν έχουν καμιά εξάρτηση μεταξύ τους. Με την δημιουργία γράφου στο πρόγραμμα μας μπορούμε να δηλώσουμε τις λειτουργίες αυτές να εκτελούνται την ίδια ώρα και το αποτέλεσμα τους να δοθεί στην D λειτουργία που το χρειάζεται για να μπορεί να εκτελεστεί.



Scalable Memory Allocator

Το TBB επιπρόσθετα παρέχει δύο memory allocators, τον scalable_allocator<T> και τον cache_aligned_allocator<T>, οι οποίοι αντιμετωπίζουν θέματα σημαντικά θέματα που απασχολούν τον παράλληλο προγραμματισμό όπως είναι το scalability και το false sharing.

Τα προβλήματα scalability προκύπτουν όταν χρησιμοποιούμε memory allocators σχεδιασμένους για σειριακά προγράμματα σε πολλαπλά νήματα. Σε κάποιους από αυτούς τους allocators, τα νήματα αυτά μπορεί να βρεθούν σε κατάσταση αμοιβαίου ανταγωνισμού προκειμένου να έχουν πρόσβαση στην κοινή μνήμη στην οποία μόνο ένα νήμα μπορεί να έχει πρόσβαση. Με την χρήση του allocator που παρέχεται από το TBB αποφεύγει τέτοιου είδους bottlenecks βελτιώνοντας την απόδοση των προγραμμάτων που συχνά δεσμεύουν και αποδεσμεύουν μνήμη.

Τα προβλήματα false sharing προκύπτουν όταν δύο νήματα κάνουν πρόσβαση σε δύο διαφορετικά words τα οποία μοιράζονται την ίδια γραμμή στην κρυφή μνήμη cache. Το πρόβλημα είναι ότι η γραμμή της κρυφής μνήμης είναι μονάδα ανταλλαγής πληροφοριών μεταξύ των caches ενός επεξεργαστή. Αν κάποιος επεξεργαστής λοιπόν

τροποποιεί μια γραμμή της cache ενώ ένας άλλος την ίδια ώρα θέλει να διαβάσει την ίδια γραμμή, η γραμμή αυτή πρέπει να μεταφερθεί από τον ένα επεξεργαστή στον άλλο ακόμη και όταν οι δύο επεξεργαστές ασχολούνται με διαφορετικά words στην ίδια γραμμή. Λάθος χρήση της κοινής μνήμης μπορεί να βλάψει αρκετά τις επιδόσεις του προγράμματος, επειδή οι γραμμές της cache παίρνουν εκατοντάδες κύκλους για να μετακινηθούν. Μπορούμε να χρησιμοποιούμε το cache_aligned_allocator<T> του TBB το οποίο δεσμεύει μια ολόκληρη γραμμή της cache, για παράδειγμα αν δύο αντικείμενα είναι δηλωμένα με αυτό τον τρόπο είναι εγγυημένο πως δεν θα έχουν προβλήματα false sharing μεταξύ τους.

Thread Local Storage:

Thread local storage είναι μια προγραμματιστική μέθοδος κατά την οποία χρησιμοποιείται static ή global μνήμη σε ένα νήμα τοπικά. Το TBB παρέχει δύο ειδών κλάσεις γι' αυτό το σκοπό. Και οι δύο παρέχουν σε κάθε νήμα πρόσβαση σε ένα τοπικό στοιχείο το οποίο δημιουργούν κατά με την εκτέλεση μιας εντολής.

- Η κλάση **combinable** παρέχει στο νήμα μια τοπική αποθήκευση για να μπορεί να κρατά υπο-υπολογισμούς που αργότερα μπορεί αν χρησιμοποιηθούν και να παράξουν ένα αποτέλεσμα.
- Η κλάση **enumerable_thread_specific** παρέχει και αυτή μια τοπική αποθήκευση σε κάθε νήμα που λειτουργεί σαν container STL (Standard Template Library). Το container αυτό επιτρέπει την επανάληψη πάνω από στοιχεία που χρησιμοποιούν την συνηθισμένη επανάληψη STL.

Task Based Programming

Η TBB βιβλιοθήκη παρέχει επίσης την δυνατότητα προγραμματισμού των tasks σε ψηλό αλλά και σε χαμηλό επίπεδο. Αυτή η μέθοδος μπορεί να χρησιμοποιηθεί σε αλγόριθμους που δεν μπορούν να χρησιμοποιήσουν τεχνικές που προσφέρονται από το TBB. Η κλάση task_group για παράδειγμα επιτρέπει στον προγραμματιστή να δημιουργήσει εύκολα σε ψηλό επίπεδο ομάδες (groups) από tasks που δουλεύουν

παράλληλα. Σε χαμηλό επίπεδο η κλάση αυτή επιτρέπει τον πιο λεπτομερή έλεγχο των tasks, όπως για παράδειγμα τον έλεγχο σε περιπτώσεις εξαιρέσεων (exceptions), όμως είναι δύσκολο στον προγραμματισμό και δεν είναι φιλικό προς το χρήστη.

Πιο κάτω φαίνεται ένα παράδειγμα του αλγόριθμου Fibonacci με την χρήση των task_groups. Συγκεκριμένα για τον υπολογισμό της σειράς Fibonacci ενός αριθμού n μπορούμε με την χρήση του task_group να δημιουργήσουμε δύο tasks x, y που υπολογίζουν αναδρομικά το n-1 και το n-2 αναδρομικά. Στο τέλος με το g.wait() περιμένουμε και τα δύο tasks να τελειώσουν για να τα αθροίσουμε και να βγάλουμε το τελικό αποτέλεσμα [6].

```
#include “tbb/task_group.h”
using namespace tbb;
int Fib (int n) {
        if (n<2){
                return n;
        }else{
                int x,y;
                task_group g;
                g.run ([&] {x=Fib (n-1);});  //spawn a task
                g.run ([&] {y=Fib (n-1);});  //spawn another task
                g.wait();                    // wait for both tasks to complete
                return x+y;
        }
}
```

Synchronization Primitives

Τέλος η TBB βιβλιοθήκη παρέχει κάποιες αρχές συγχρονισμού για αμοιβαίο αποκλεισμό και την εκτέλεση ατομικών λειτουργιών. Ο αμοιβαίος αποκλεισμός ελέγχει πόσα νήματα μπορούν ταυτόχρονα να τρέχουν ένα κομμάτι κώδικα. Με το TBB υλοποιείται με τη χρήση των mutexes και των locks. Mutex είναι ένα αντικείμενο από το οποίο ένα νήμα μπορεί να αποκτήσει ένα lock. Οποιαδήποτε χρονική στιγμή

μόνο ένα νήμα μπορεί να έχει lock σε ένα αντικείμενο mutex, τα υπόλοιπα πρέπει να περιμένουν τη σειρά τους.

Όπως αναφέραμε στην αρχή το TBB μας δίνει την δυνατότητα να χρησιμοποιήσουμε ατομικές λειτουργίες αποφεύγοντας τον αμοιβαίο αποκλεισμό. Όταν ένα νήμα εκτελεί μια ατομική λειτουργία, τα άλλα νήματα το βλέπουν σαν να εκτελείται η λειτουργία ακαριαία, δηλαδή είναι σχετικά πολύ πιο γρήγορη η εκτέλεση της ατομικής πράξης από τις κλειδαριές. Το μειονέκτημα όμως είναι ότι κάνουν μόνο ένα περιορισμένο αριθμό λειτουργιών και συχνά αυτός ο αριθμός δεν είναι αρκετός για να συνθέσουν πιο πολύπλοκες λειτουργίες αποτελεσματικά.

4.4 Block Matrix Multiplication TBB vs DDM

Αρχικά για την μελέτη και τη συμπεριφορά του TBB υλοποίησα τον αλγόριθμο block matrix multiplication ο οποίος είναι ένας από τους πιο απλούς και πλήρως παραλληλοποιήσιμους αλγόριθμους μέσα από τον οποίο μπορεί κάποιος να δει πόσο πιο βέλτιστος είναι ο παράλληλος αλγόριθμος που υλοποίησε σε σχέση με τον σειριακό αλγόριθμο.

Block matrix multiplication:

Έστω οι πίνακες A και B:

$$A = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]$$

δ

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$B = \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

$$B_{11} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

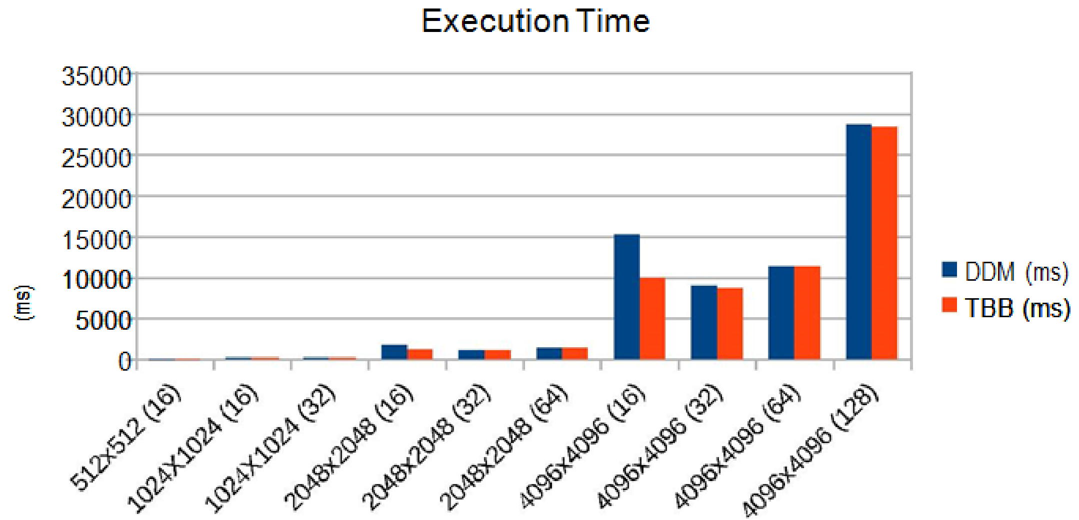
Παράλληλη Εκτέλεση:

$A_{11}B_{11} + A_{12}B_{21}$
$A_{21}B_{11} + A_{22}B_{21}$
$A_{11}B_{12} + A_{12}B_{22}$
$A_{22}B_{12} + A_{22}B_{22}$

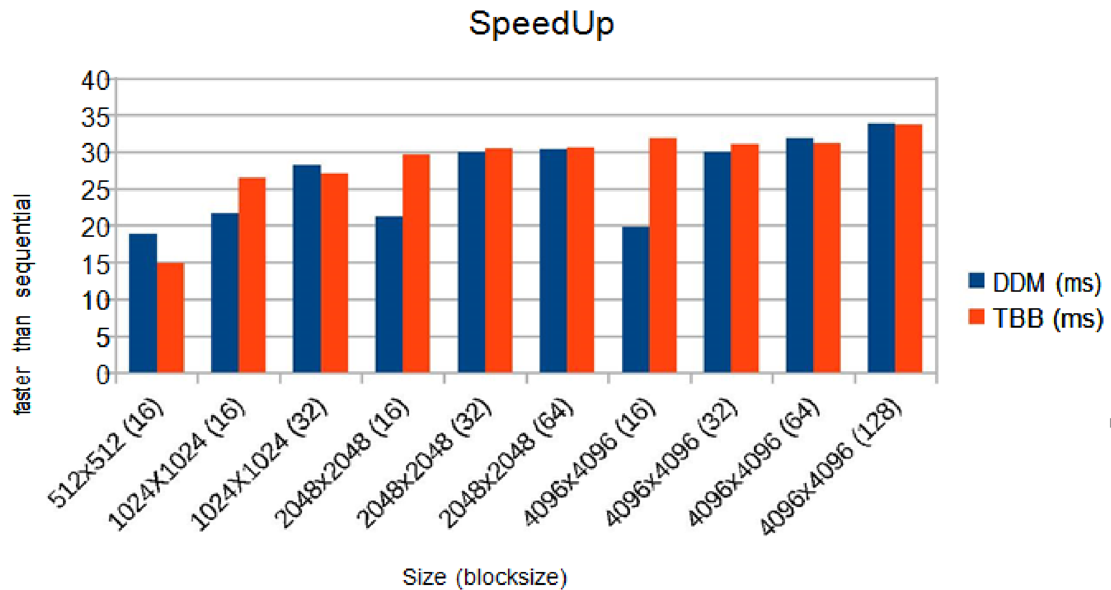
$$AB = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{22}B_{12} + A_{22}B_{22} \end{array}\right]$$

Η σύγκριση πέρα από το σειριακό χρόνο εκτέλεσης έγινε και με βάση την επιτάχυνση του παράλληλου αλγόριθμου υλοποιημένου σε DDM που είναι η γλώσσα του εργαστηρίου. Επιγραμματικά το DDM χρησιμοποιεί ένα νήμα μέσα στο οποίο δημιουργούνται κάποια contexes. Το κάθε contex είναι προγραμματισμένο να εκτελέσει μια συγκεκριμένη λειτουργία και αρχικά είναι ορισμένα με μια μεταβλητή που δείχνει τον αριθμό των εξαρτήσεων που περιμένει μέχρι να μπορέσει να ξεκινήσει την λειτουργία του (όλα τα contex αρχικά έχουν την μεταβλητή αυτή ορισμένη με 1). Όταν η μεταβλητή αυτή γίνει ίση με μηδέν το contex ξεκινά την λειτουργία του. Σημαντικό είναι το TSU το οποίο ενημερώνει αυτή τη μεταβλητή σε όλα τα contexes κάθε φορά που κάποια λειτουργία τελειώνει.

Γραφική παράσταση χρόνου εκτέλεσης του TBB και του DDM σε σχέση με το σειριακό χρόνο.



Γραφική παράσταση επιτάχυνσης του TBB και του DDM σε σχέση με το σειριακό χρόνο.

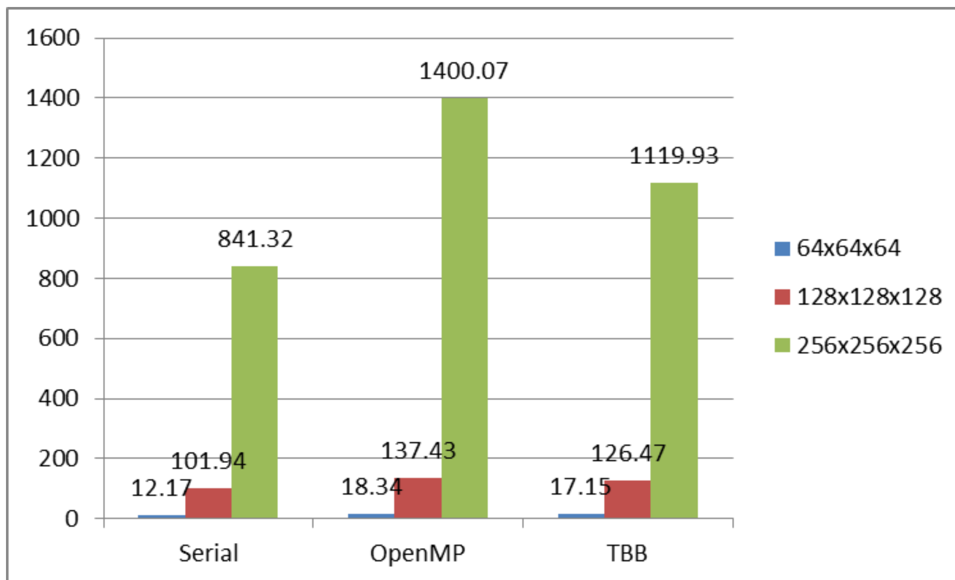


Παρατηρούμε από τις γραφικές ότι το TBB έχει ελάχιστα καλύτερη επίδοση από το DDM. Αυτό συμβαίνει λόγω των tasks που χρησιμοποιεί το TBB όπου σε κάθε task κατανέμεται μια εργασία (ένα block). Επίσης το DDM έχει το μειονέκτημα ότι σπαταλά ένα νήμα για τον έλεγχο του TSU χωρίς να κάνει οποιαδήποτε άλλη λειτουργία. Το speedup στο TBB κυμαίνεται στα ίδια επίπεδα 27 με 33 φορές πιο γρήγορο από το σειριακό μοντέλο. Το DDM έχει ελαφρώς μικρότερη επίδοση και κυρίως σε περιπτώσεις όπου το block size είναι πιο μικρό. Η επίδοση αυτή του TBB οφείλεται στην δημιουργία των tasks. Για κάθε block το TBB δημιουργεί ένα task το οποίο εκτελεί matrix multiplication για να υπολογίσει τις νέες τιμές του block. Για τέτοιες εφαρμογές το TBB έχει αρκετά καλές επιδόσεις.

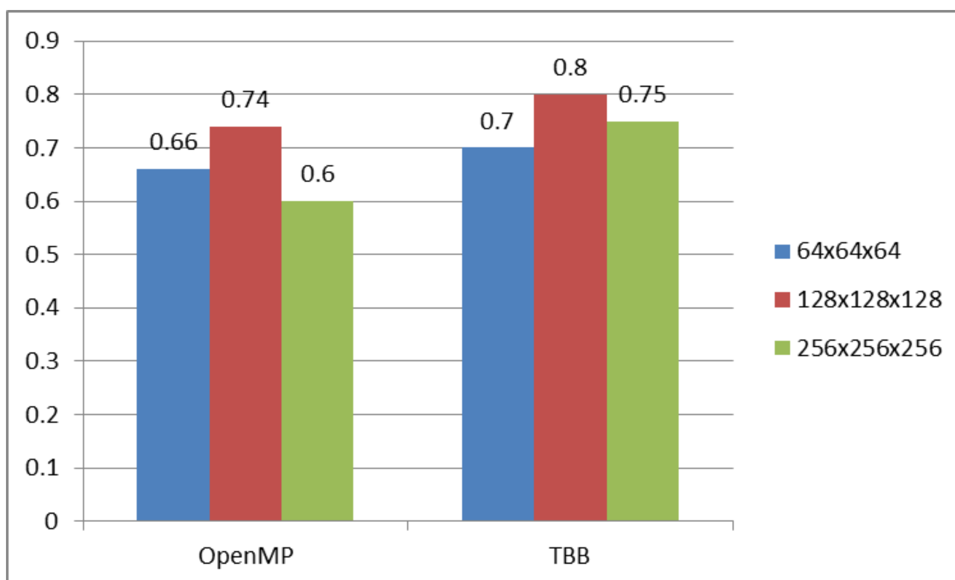
4.4 HPCG with TBB instead of OpenMP

OpenMP vs TBB

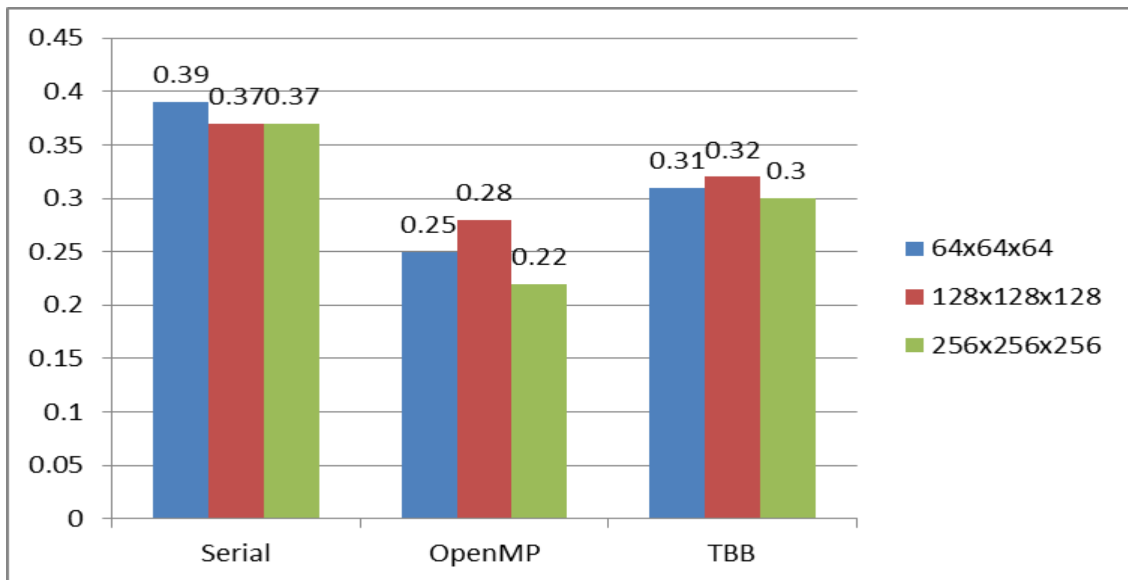
Γραφική παράσταση εκτέλεσης χρόνου του HPCG με OpenMP και HPCG με TBB σε σχέση με το σειριακό χρόνο εκτέλεσης



Γραφική παράσταση επιτάχυνσης HPCG με OpenMp και HPCG με TBB



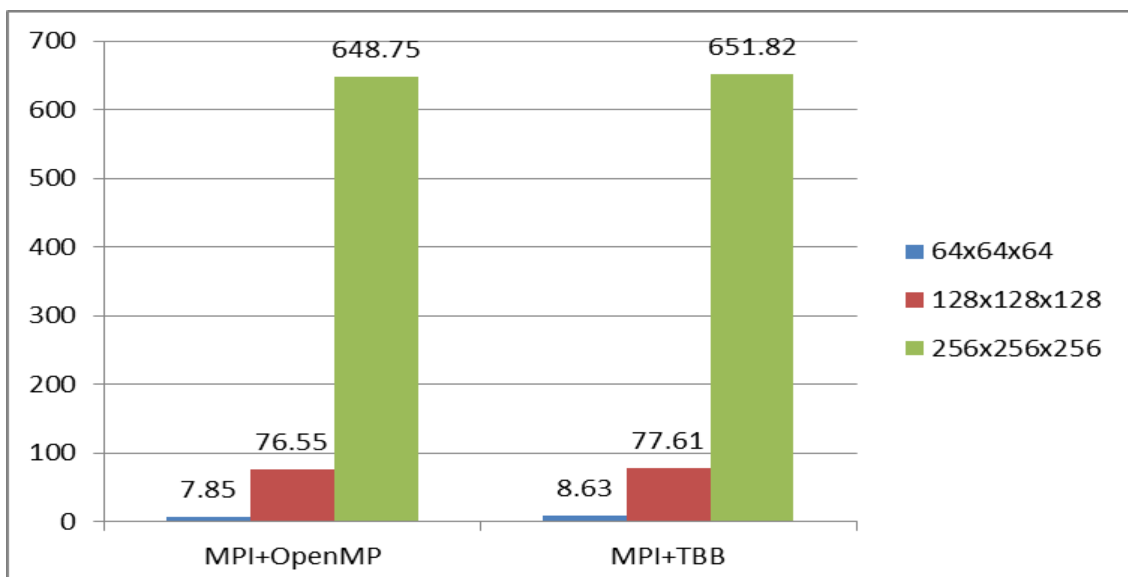
Γραφική παράσταση σύγκρισης των GFLOPs/s στο σειριακό μοντέλο, με OpenMp και TBB



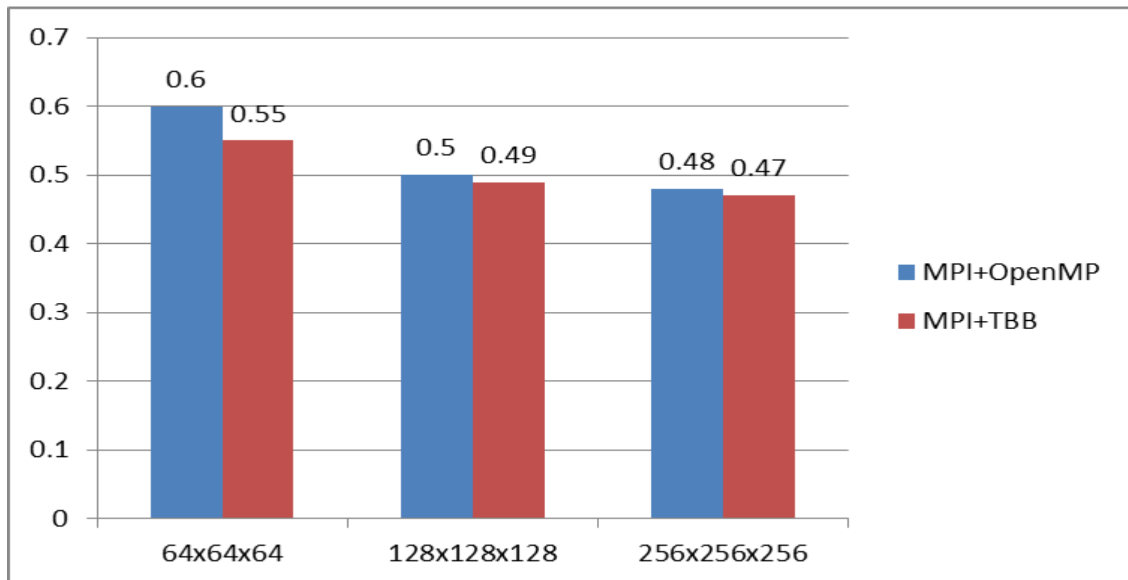
Από τις γραφικές παρατούμε ότι το TBB έχει ελάχιστα καλύτερη απόδοση από την OpenMP. Το μειονέκτημα της OpenMP στην περίπτωση του HPCG είναι η δημιουργία και καταστροφή των threads που γίνεται κάθε φορά το benchmark εκτελεί τη βασική συνάρτηση. Η δημιουργία και η καταστροφή τους προκαλεί αρκετά overheads και αυτό μειώνει την απόδοση στην περίπτωση του OpenMP.

MPI+OpenMP vs MPI+TBB

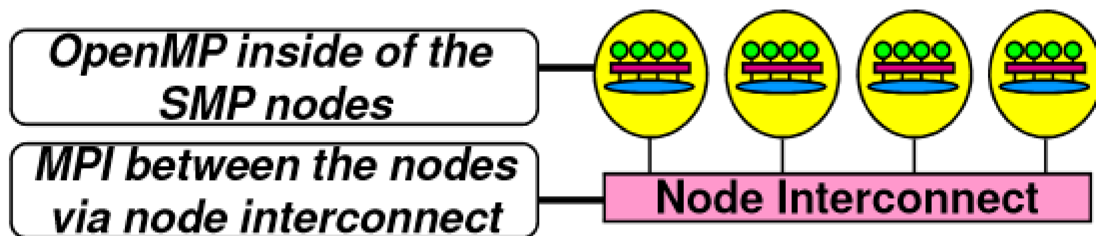
Χρόνος εκτέλεσης του HPCG με MPI+OpenMP και MPI+TBB



GFLOPs το δευτερόλεπτο για MPI+OpenMP και MPI+TBB



Στην περίπτωση χρήσης του MPI το OpenmMP δουλεύει καλύτερα με το MPI παρά το TBB για το λόγο ότι το MPI διασπά τα δεδομένα και το και ελαχιστοποιεί τις λειτουργίες του OpenMP μειώνοντας τα overheads που δημιουργούνταν πριν με μόνο OpenMP.



Κεφάλαιο 5

Συμπεράσματα

Στο κεφάλαιο αυτό θα συγκρίνουμε το TBB με το OpenMP με βάση τις μετρήσεις που πήραμε από το Κεφάλαιο 4. Γενικά ένα από τα πλεονεκτήματα που προσφέρουν και τα δύο προγραμματιστικά μοντέλα είναι η ότι δημιουργούν και διαχειρίζονται τα thread pool για το προγραμματιστή χωρίς χρειάζεται να τον δυσκολεύει με το να χειρίζεται ο ίδιος προγραμματιστικά [4].

Εάν ο κώδικας είναι γραμμένος σε C++, είναι πιθανό το TBB να είναι και η καλύτερη επιλογή, γιατί ταιριάζει πολύ καλύτερα ιδιαίτερα όταν ο κώδικας είναι χρησιμοποιεί αρκετή αντικειμενοστρέφια και κάνει μεγάλη χρήση των προτύπων της C++. Εάν ο κώδικας είναι γραμμένος σε C, η OpenMP θα ήταν καλύτερη λύση γιατί ταιριάζει καλύτερα στην δομή του κώδικα και για σε απλά προβλήματα, χρειάζεται λιγότερο κόστος κωδικοποίησης. Αλλά ακόμη και με την C++ εάν ο κώδικας κυριαρχείται από πολύ επεξεργασία πινάκων, η OpenMP μπορεί πάλι να είναι μια καλύτερη επιλογή αλλά εξαρτάται πάντα και από την πολυπλοκότητα του αλγόριθμου.

Πέρα της γλώσσας προγραμματισμού που θα χρησιμοποιήσουμε για την επιλογή του κατάλληλου προγραμματιστικού μοντέλου πρέπει στη συνέχεια πρέπει να κοιτάξουμε το τι θέλουμε να παραλληλοποιήσουμε. Χρησιμοποιούμε OpenMP αν ο παραλληλισμός γίνεται κατά κύριο λόγο σε φραγμένους βρόγχους, βρόγχοι δηλαδή που γνωρίζουμε πότε θα τελειώσουν και όταν χρησιμοποιούμε μεταβλητές με τύπους της γλώσσας. Το TBB βασίζεται στο γενικό προγραμματισμό, άρα χρησιμοποιούμε τον παραλληλισμό του σε βρόγχους όταν χρειάζεται να εργαστούμε με συνηθισμένες επαναλήψεις ή περίπλοκες reduction λειτουργίες. Επίσης μπορούμε να χρησιμοποιήσουμε TBB αν πρέπει να προχωρήσουμε πέρα από τον παραλληλισμό του βρόγχου, δεδομένου ότι παρέχει γενικά πρότυπα παραλληλισμού για while-loops και data-flow pileline models.

Ο παραλληλισμός σε φωλιασμένα loops υποστηρίζεται από την OpenMP, όμως ίσως να είναι δύσκολο να αποφύγουμε την υπερβολική χρήση των πόρων με αυτό τον τρόπο παραλληλισμού, ενώ αντίθετα το TBB έχει σχεδιαστεί ώστε φυσικά να υποστηρίζει φωλιασμένο αλλά και αναδρομικό παραλληλισμό. Ένας σταθερός αριθμός νημάτων διαχειρίζεται από την τεχνική task stealing του task scheduler του TBB επιτυγχάνοντας δυναμική διαμοίραση εργασιών σε αυτά. Επίσης το TBB μπορεί να κρατήσει όλους τους επεξεργαστές σε συνεχή λειτουργία κάνοντας χρήσιμη δουλειά χωρίς να ξεπερνά τον κατάλληλο αριθμό νημάτων (μεγάλος αριθμός νημάτων σημαίνει αχρείαστο κόστος, και μικρός αριθμός νημάτων δεν εκμεταλλευόμαστε πλήρως τον αριθμό των επεξεργαστών) .

Σύγκριση δυνατοτήτων

	TBB	**OpenMP**
Task level parallelism	+	+
Data decomposition support	+	+
Complex parallel patterns	+	-
Scalable nested parallelism support	+	-
Built-in load balancing	+	+
Scalable memory allocator	+	-
Compiler support is not required	+	-

5.2 Παράλληλος προγραμματισμός

Με βάση και τα όσα είδαμε από όλα τα κεφάλαια μπορούμε να πούμε ότι ο παράλληλος προγραμματισμός έγινε αναπόφευκτο κομμάτι στις μέρες μας σε πολλούς τομείς. Η έννοια του single-thread προγραμματισμού έχει πάψει να υφίσταται και έχουμε πάρει ένα αναστρέψιμο βήμα προς τις παράλληλες αρχιτεκτονικές.

Τέλος ο παράλληλος προγραμματισμός είναι μια δύσκολη διαδικασία. Απαιτεί πολλές γνώσεις και εξαρτάται από πολλές παραμέτρους όπως την αρχιτεκτονική της μηχανής, το προγραμματιστικό μοντέλο που χρησιμοποιούμε, τα δεδομένα, το πρόβλημα και τις εξαρτήσεις του. Γι’ αυτό το λόγο ποτέ δεν μπορείς να υλοποιήσεις τον τέλειο παράλληλο αλγόριθμο.

Bibliography

[1] HPCG About, [https://software.sandia.gov/hpcg/about.php**]**

[2] HPCG: High Performance Conjugate Gradient Benchmark,
[https://software.sandia.gov/hpcg/html/index.html]

[3] Massage Passing Interface (MPI), [https://computing.llnl.gov/tutorials/mpi/**]**

[4] OpenMP, [https://computing.llnl.gov/tutorials/openMP/**]**

[5] Intel Thread Building Blocks, [https://www.threadingbuildingblocks.org/]

[6] Intel® Optimized Technology Preview for High Performance Conjugate Gradient Benchmark, [https://software.intel.com/en-us/articles/intel-optimized-technology-preview-for-high-performance-conjugate-gradient-benchmark]

[7] Looking Beyond Linpack: New Supercomputing Benchmark in the Works,
[http://www.datacenterknowledge.com/archives/2013/07/24/supercomputing-benchmark-set-to-evolve/**]**

[8] **Introduction to Parallel Computing**,
[https://computing.llnl.gov/tutorials/parallel_comp/]

Παράρτημα Α-1

Συνάρτηση ComputeMG_ref με την χρήση του TBB για την δημιουργία γράφου εξαρτήσεων.

```
int count=0;
int ComputeMG_ref(const SparseMatrix & A, const Vector & r, Vector & x) {
count++;
assert(x.localLength==A.localNumberOfColumns);
ZeroVector(x);
tbb::task_group g;
int ierr = 0;
if (A.mgData!=0) {
	int numberOfPresmootherSteps = A.mgData->numberOfPresmootherSteps;
	for (int i=0; i< numberOfPresmootherSteps; ++i)
		ierr += ComputeSYMGS_ref(A, r, x); if (ierr!=0) return(ierr);
	g.run( [=]{ComputeSPMV_ref(A, x, *A.mgData->Axf);} );
	g.run( [=]{ComputeRestriction_ref(A, r);} );
	g.run( [=]{ComputeMG_ref(*A.Ac,*A.mgData->rc, *A.mgData->xc);} );
	g.wait();

	ierr = ComputeProlongation_ref(A, x);
	if (ierr!=0) return(ierr);
	int numberOfPostsmootherSteps = A.mgData->numberOfPostsmootherSteps;
	for (int i=0; i< numberOfPostsmootherSteps; ++i)
		ierr += ComputeSYMGS_ref(A, r, x);
	if (ierr!=0) return(ierr);
} else {
ierr = ComputeSYMGS_ref(A, r, x);
if (ierr!=0)
	return(ierr);
}
return(0);
}
```

Παράρτημα Α-2

Reduction in function DotProduct using TBB

```
using namespace tbb;

struct Sum {
	double value;
	Sum() : value(0) {}
	Sum( Sum& s, split ) {value = 0.0;}
	void operator()( const blocked_range<double*>& r ) {
		double temp = value;
		for(double* a=r.begin(); a!=r.end(); ++a )
			temp += *a * *a;
		value = temp;
		}
	void join( Sum& rhs ) {value += rhs.value;}
  };

double ParallelSum( double array[], int n ) {
	Sum total;
	parallel_reduce( blocked_range<double*>( array, array+n ), total );
	return total.value;
}



local_result=ParallelSum(xv,n);
```

Simple Parallel Loop of the function ComputeRestriction_ref using TBB

```
tbb::parallel_for (0, nc, [&](int i){ rcv[i] = rfv[f2c[i]] - Axfv[f2c[i]]; });
```