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

Αξιολόγηση της γλώσσας Chapel
Επιδόσεις και παραγωγικότητα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Α. Χρηστίδης

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2012

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

Αξιολόγηση της γλώσσας Chapel
Επιδόσεις και παραγωγικότητα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Α. Χρηστίδης

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την Πέμπτη 5 Ιουλίου 2012.

.....	.....	.....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Παναγιώτης Τσανάκας
Αναπληρωτής Καθηγητής ΕΜΠ	Επίκουρος Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2012

.....

Χρήστος Α. Χρηστίδης

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

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η αξιολόγηση μιας νέας παράλληλης γλώσσας προγραμματισμού με το όνομα Chapel. Η γλώσσα αυτή προέκυψε μέσα από το πρόγραμμα HPCS της DARPA το οποίο έχει ως διακυρηγμένο στόχο την επίλυση του προβλημάτος της χαμηλής παραγωγικότητας που χαρακτηρίζει τον χώρο ανάπτυξης παράλληλου λογισμικού. Ως βάση για σύγκριση, δημιουργήσαμε μια μικρή «βιβλιοθήκη» μετρο-προγραμμάτων που περιλαμβάνει τους εξής γνωστούς αλγορίθμους:

- Γινόμενο πινάκων
- Αλγόριθμος Floyd-Warshall
- Επαναληπτικός αλγόριθμος Jacobi
- Παράλληλος (Black-Red) αλγόριθμος Gauss-Seidel
- Πολλαπλασιασμός αραιού πίνακα με διάνυσμα

Οι ανωτέρω αλγόριθμοι γράφτηκαν στην Chapel 1.4.0 και σε δύο άλλες καθιερωμένες παράλληλες γλώσσες, τις OpenMP και MPI, και έτρεξαν πάνω σε μια συστοιχία εμπορικών πολυπύρηνων επεξεργαστών συνδεδεμένων μέσω Gigabit Ethernet. Η σύγκριση των τριών υλοποιήσεων έδειξε ότι η Chapel είναι πολύ κοντά στην OpenMP τόσο σε απόλυτους χρόνους όσο και σε κλιμακωσιμότητα. Όσον αφορά την εκτέλεση σε πολλούς κόμβους μπορούμε να διακρίνουμε δύο περιπτώσεις: όταν οι αλγόριθμοι δεν απαιτούν επικοινωνία, η Chapel εμφανίζει άριστη κλιμακωσιμότητα και είναι μόλις 4 φορές πιο αργή από την MPI. Στους υπόλοιπους αλγορίθμους οι χρόνοι της είναι ως και χιλιάδες φορές μεγαλύτεροι της MPI που σημαίνει ότι η χρήση της Chapel δεν είναι καθόλου πρακτική, τουλάχιστον με μη εξειδικευμένο hardware. Διάφοροι λόγοι για την υστέρηση αυτή δίνονται στο κεφάλαιο 6.5. Επίσης, επιχειρήσαμε μία όσο το δυνατόν πιο αντικειμενική σύγκριση για την ευκολία προγραμματισμού (*programmability*), απ'την οποία η Chapel αναδείχθηκε ως ο ξεκάθαρος νικητής.

Ο δεύτερεύων στόχος της διπλωματικής ήταν η διερεύνηση ενός πολύ σημαντικού χαρακτηριστικού της Chapel, που είναι τα πεδία ορισμού (*domains*) που δημιουργούνται από τους χρήστες. Με τον όρο *domain* εννοούμε το σύνολο των δεικτών για τους οποίους ένας πίνακας έχει στοιχεία. Στην Chapel, ο τρόπος που ορίζεται το *domain* ενός πίνακα καθορίζει την κατανομή του τελευταίου στους διάφορους επεξεργαστικούς κόμβους και τον αριθμό των νημάτων που τον επεξεργάζονται παράλληλα. Εμείς υλοποιήσαμε ένα πεδίο ορισμού για την σχετικά δύσκολη κατανομή δεικτών του αλγόριθμου Black-Red και μετρήσαμε την απόδοσή του έναντι της «απλής» μας υλοποίησης του αλγορίθμου.

Λέξεις-κλειδιά

Chapel, HPCS, Υπολογιστικά Συστήματα Υψηλής Παραγωγικότητας, user-defined domains, σουίτα μετροπρογραμμάτων

Abstract

The main objective of this thesis is the evaluation of a new parallel programming language called Chapel. This language came out of the HPCS project initiated by DARPA, with the professed purpose of solving the problem of low productivity that plagues development of parallel software. As a basis for comparison, we created a small library of benchmark programs based on the following well-known algorithms:

- Matrix multiplication
- Floyd-Warshall
- Iterative Jacobi
- Parallel (Black-Red) Gauss-Seidel
- Multiplication of sparse matrix with vector

The above algorithms were written in Chapel 1.4.0 and in two other established parallel "languages", OpenMP and MPI, and were run on a cluster of commodity multi-core processors connected via Gigabit Ethernet. The comparison of the three implementations showed that Chapel's performance is very close to that of OpenMP both in absolute times and scalability. As far as distributed execution is concerned, we can discern two separate cases: on algorithms that lack any communication, Chapel is only 4 times slower than MPI and displays excellent scalability. Whereas on algorithms with remote accesses, Chapel can be up to a thousand times slower than MPI, thus rendering its use impractical, at least when run on non-special hardware. Various reasons for this lag are given in chapter 6.5. Finally, we also compared the three languages in the programmability domain, where Chapel emerged as the clear winner.

The secondary objective of the thesis was the investigation of a very important feature of Chapel called user-defined domains. In Chapel, *domain* is the name given to the index set that represents the indexes for which an array has elements. The way the domain is defined also determines how the array is to be distributed to the various processing nodes as well as the number of threads that will run when processing it in parallel. We implemented such a domain for the relatively difficult index distribution of the Black-Red algorithm and we measured its performance compared to our vanilla Black-Red algorithm.

Keywords

Chapel, HPCS, High Productivity Computing Systems, user-defined domains, benchmark suite

Περιεχόμενα

Πίνακας Σχημάτων και Εικόνων

1 Εισαγωγή

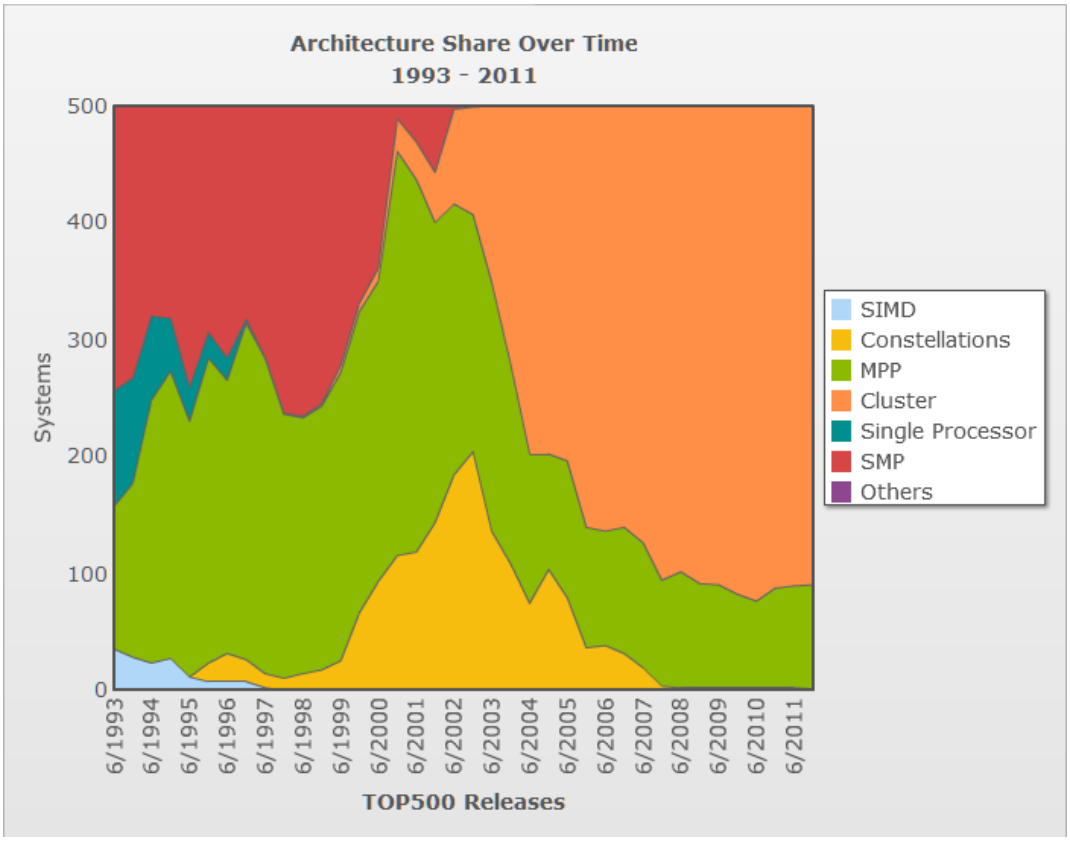
Οι παράλληλοι υπολογιστές δεν είναι πρόσφατη εξέλιξη· ο πρώτος παράλληλος υπολογιστής, ονόματι Burroughs D825, εμφανίστηκε μόλις το 1962 και διέθετε 4 επεξεργαστές οι οποίοι συνδέονταν με την μνήμη μέσω ενός διακόπτη εγκάρσιων ράβδων. Αυτό το πρώιμο ενδιαφέρον για τις παράλληλες αρχιτεκτονικές ήταν απόρροια της ευρείας τότε πεποίθησης μεταξύ των επιστημόνων πληροφορικής ότι η βελτίωση της απόδοσης των επεξεργαστών επρόκειτο πολύ σύντομα να πέσει πάνω στον τοίχο που ακούει στο όνομα ταχύτητα του φωτός. Στην πορεία αυτή η πεποίθηση διαψεύστηκε χάρη σε μια σειρά από επαναστάσεις στον τρόπο κατασκευής και στην αρχιτεκτονική των μονο-επεξεργαστών οι οποίες θα περιγραφούν στη συνέχεια. Πάντως, τα ερευνητικά κέντρα στράφηκαν σε υπερ-υπολογιστές οι οποίοι ήταν άκριβοί εξαιτίας της πολύπλοκης αρχιτεκτονικής και επειδή κατασκευάζονταν σε λίγα αντίτυπα κατόπιν παραγγελίας, ενώ στο εμπορικό κομμάτι κυριάρχησαν οι «φθηνοί» μονο-επεξεργαστές.

Η αύξηση του αριθμού των τρανζίστορ στα ολοκληρωμένα κυκλώματα που ακολούθησε τον νόμο του Moore (διπλασιασμός κάθε 2 χρόνια και αργότερα κάθε 18 μήνες) και η επακόλουθη αύξηση της απόδοσης κράτησε περίπου 4 δεκαετίες. Στο διάστημα αυτό, οι αρχιτέκτονες των μονο-επεξεργαστών εκμεταλλεύτηκαν τα επιπλέον τρανζίστορ για να υλοποιήσουν σημαντικές καινοτομίες με πρώτη και κυρίαρχη την σωλήνωση εντολών (*pipeline instruction*). Με αυτή έγινε εφικτή μια πρώτη περιορισμένη μορφή παραλληλισμού, ο λεγόμενος παραλληλισμός σε επίπεδο εντολών (ILP) χάρη στον οποίο ο επεξεργαστής μπορεί να εκτελεί πολλές εντολές ταυτόχρονα με την κάθεμια να καταλαμβάνει διαφορετικό στάδιο της σωλήνωσης. Η δυσκολία του εγχειρήματος έγκειτο στην ανάγκη δυναμικού ελέγχου των εξαρτήσεων δεδομένων κατά την ώρα εκτέλεσης, αφού πλέον δεν ίσχυε η υπόθεση της ολοκλήρωσης μιας εντολής πριν την έναρξη της επόμενης.

Το πρώτο αυτό άλμα διαδέχτηκαν και άλλες καινοτομίες όπως η εκτέλεση των εντολών εκτός σειράς (*out-of-order execution*) , η πρόβλεψη διακλάδωσης (*branch prediction*) κατά την οποία ο επεξεργαστής διαλέγει ένα μονοπάτι του βρόχου βάσει της πρόσφατης συμπεριφοράς του, και ακόμα η έκδοση πολλαπλών εντολών σε κάθε κύκλο ρολογιού (υπερβαθμωτοί επεξεργαστές) η οποία αύξησε κατακόρυφα την δίψα των τελευταίων για εντολές προς αποφυγή του *pipeline stalling*. Η ανάγκη για πιο γρήγορη τροφοδοσία αντιμετωπίστηκε με διάφορους τρόπους όπως μεγαλύτερες caches, αναδιάταξη των εντολών απ'τον μεταγλωττιστή και την υποθετική εκτέλεση (*speculative execution*) κατά την οποία ο επεξεργαστής εκτελεί κώδικα ο οποίος μπορεί και να μην χρειαστεί. Τέλος, να αναφέρουμε συνοπτικά τον παραλληλισμό μέσω διανυσματικών εντολών, για παράδειγμα το σετ εντολών SSE (*Streaming SIMD Extensions*) που επιτρέπει την ταυτόχρονη επεξεργασία πολλών δεδομένων, αλλά και την τεχνική SMT (*Simultaneous Multi-Threading*) η οποία εκτελεί μια δεύτερη ροή *ελέγχου* σε περίπτωση που ο επεξεργαστής περιμένει λόγω αστοχίας cache της πρώτης (ψευδο-παραλληλισμός σε επίπεδο νήματος).

Η αύξηση της απόδοσης των μονο-επεξεργαστών πήρε τέλος την 1η δεκαετία του 21ου αιώνα όταν και έγινε αντιληπτό ότι η παραγόμενη θερμότητα όρθωνε ανυπέρβλητα εμπόδια στην περαιτέρω αύξηση του ρολογιού. Οι αρχιτέκτονες Η/Υ στράφηκαν στην μοναδική εναπομείνασα επιλογή· χρησιμοποίησαν τα τρανζίστορ που είχαν στη διάθεσή τους για να χωρέσουν στο ίδιο chip δύο πυρήνες (και αργότερα ολόκληρους ελεγκτές μνήμης και μονοπάτια διασύνδεσης). Με αυτό τον τρόπο, οι παράλληλοι υπολογιστές έκαναν την είσοδό τους στον εμπορικό κόσμο με την μορφή των πολυπύρηνων επεξεργαστών.

Από την άλλη πλευρά, η σύγκλιση έγινε την ίδια περίπου εποχή, όταν οι επιστήμονες αντιλήφθηκαν τα ωφέλη της κατασκευής υπερ-υπολογιστών με τη σύνδεση πολλών και φτηνών εξαρτημάτων έναντι λίγων και ισχυρών. Οι πανάκριβοι υπερ-υπολογιστές έδωσαν σιγά σιγά τη θέση τους στα λεγόμενα συστήματα MPP (Massively Parallel Processing) τα οποία αποτελούνται από φθηνούς επεξεργαστές του εμπορίου που διασυνδέονται με ένα ταχύτατο και ακριβό δίκτυο διασύνδεσης. Αργότερα και το εξειδικευμένο δίκτυο υποχώρησε προς χάριν εμπορικών λύσεων οδηγώντας στις λεγόμενες συστοιχίες υπολογιστών (clusters) οι οποίες αποτελούν σήμερα την πλειοψηφία των μεγάλων παράλληλων υπολογιστών. Αυτή η σχετικά πρόσφατη εξέλιξη απεικονίζεται στην εικόνα 1-1.



Εικόνα 1-1: Η αρχιτεκτονική εξέλιξη των top 500 υπολογιστών

Το μείζον θέμα που ανέκυψε με την εμφάνιση των σύγχρονων ανομοιογενών και πολυεπίπεδων συστοιχιών έχει να κάνει με την δυσκολία προγραμματισμού τους. Πράγματι, αν κάποιος εξετάσει την κατάσταση στον τομέα του παράλληλου λογισμικού όπως αυτή διαμορφώνεται τα τελευταία χρόνια θα διαπιστώσει ότι πλέον δεν υφίσταται μια ενιαία λύση. Αντιθέτως, χρησιμοποιείται ένας συνδυασμός από διαφορετικά προγραμματιστικά «παραδειγμάτα», καθένα από τα οποία αντιστοιχεί σε διαφορετικό επίπεδο της αρχιτεκτονικής. Δεν είναι σπάνιο να έχουμε στο ίδιο πρόγραμμα χρήση της MPI για τον παραλληλισμό σε επίπεδο διεργασίας, OpenMP για την πολυνηματική επεξεργασία εντός κόμβου (SMP ή NUMA) και CUDA για τον προγραμματισμό των γραφικών επεξεργαστών που τυχόν επιταχύνουν τον παράλληλο υπολογιστη. Προφανώς, η ταυτόχρονη γνώση όλων αυτών των ιδιωμάτων πέραν της γνώσης των ίδιων των αλγορίθμων, δυσχεραίνουν αφάνταστα τη δουλειά του παράλληλου προγραμματιστή.

Ένα παρεμφερές πρόβλημα είναι το μεγάλο χάσμα που έχει πλέον αναπτυχθεί ανάμεσα στις λεγόμενες σειριακές και τις παράλληλες γλώσσες. Για παράδειγμα η MPI που είναι η *lingua franca* του κόσμου του παράλληλου προγραμματισμού χαρακτηρίζεται από πολύ χαμηλό επίπεδο, αντίστοιχο της C. Την ίδια στιγμή, δεν υπάρχει κάποια άλλη παράλληλη γλώσσα που να αντιστοιχεί στις νέες υψηλού επιπέδου σειριακές γλώσσες οι οποίες εμφανίστηκαν τις τελευταίες δεκαετίες (βλέπε Java, C#, Python) και οι οποίες βρίθουν καινοτομιών και αυτονόητων πλέον χαρακτηριστικών. Η συνέπεια είναι ότι απ'τον γενικότερο πληθυσμό των προγραμματιστών, μόνο ένα μικρό ποσοστό επιλέγει ή τολμά να ασχοληθεί με το επάγγελμα του προγραμματισμού παράλληλου λογισμικού.

Σκοπός μας σε αυτή την διπλωματική είναι να εξετάσουμε ίσως την πιο ελπιδοφόρα από τις προτεινόμενες λύσεις η οποία ακούει στο όνομα Chapel. Η Chapel είναι μια νέα παράλληλη γλώσσα προγραμματισμού που προέκυψε μέσα από το πρόγραμμα HPCS και η οποία φιλοδοξεί να είναι *αυτή* η ενιαία λύση που αναφέραμε παραπάνω. Η ειδοποιός διαφορά της σε σχέση με τις παραδοσιακές παράλληλες γλώσσες είναι η έμφαση που δίνει στην ευκολία προγραμματισμού ή αλλιώς προγραμματισιμότητα με σκοπό την αύξηση της παραγωγικότητας των προγραμματιστών.

Η δομή της διπλωματικής έχει ως εξής: αρχικά, στα κεφάλαια 2 & 3 κάνουμε μία αναδρομή στις βασικές αρχιτεκτονικές και στα πιο διαδεδομένα μοντέλα παράλληλου προγραμματισμού έτσι ώστε να αναδείξουμε τα κύρια πλεονεκτήματα/μειονεκτήματά τους και να τα αντιπαραβάλλουμε με το μοντέλο PGAS στο οποίο βασίζεται η Chapel. Στο κεφάλαιο 4 επιχειρούμε μια παρουσίαση της Chapel συνοδεία κώδικα, επικεντρώνοντας στα στοιχεία που χρησιμοποιήσαμε γι'αυτή την διπλωματική. Ακολουθεί η θεωρητική περιγραφή των αλγορίθμων που υλοποιήσαμε (κεφάλαιο 5) μαζί με τον κώδικα σε Chapel ειδικά για την κατανεμημένη περίπτωση. Το κύριο μέρος της διπλωματικής περιέχεται στο κεφάλαιο 6 όπου παραθέτουμε τα αποτελέσματα των μετρήσεων μας όσον αφορά τις επιδόσεις και την παραγωγικότητα. Τέλος, στο κεφάλαιο 7 περιγράφουμε τον τρόπο συγγραφής κατανομών από τον χρήστη και συγκεκριμένα αξιολογούμε μία κατανομή που γράψαμε για τον αλγόριθμο Black-Red έναντι της απλής υλοποίησης του κεφαλαίου 5.

2 Είδη παράλληλων αρχιτεκτονικών

Στο κεφάλαιο αυτό θα περιγράψουμε συνοπτικά όλα τα είδη των Η/Υ με έμφαση στις παράλληλες αρχιτεκτονικές. Σύμφωνα με την ταξινόμηση του Flynn (1966), όλοι οι υπολογιστές που δύνανται να κατασκευαστούν ανήκουν στις εξής τέσσερεις κατηγορίες:

2.1 SISD (Single Instruction, Single Data)

Μη παράλληλοι μονο-επεξεργαστές (χωρίς εντολές SIMD)

2.2 SIMD (Single Instruction, Multiple Data)

Οι υπολογιστές SIMD αποτελούνται από μεγάλο αριθμό επεξεργαστών οι οποίοι προχωρούν συγχρονισμένα και εκτελούν σε κάθε βήμα την ίδια εντολή πάνω σε διαφορετικά δεδομένα. Συνεπώς, είναι πολύ γρήγοροι σε ειδικές εφαρμογές που απαιτούν απλή επεξεργασία μεγάλου αριθμού δεδομένων, αλλά δυσκολεύονται όταν η δουλειά που πρέπει να εκτελέσει ο κάθε επεξεργαστής είναι διαφορετική. Αν ο κώδικας περιέχει πολλές διακλαδώσεις η ταχύτητα μπορεί να περιοριστεί ως και το $\sim 1/n$ της αρχικής, όπου n είναι ο αριθμός των επεξεργαστών. Τέτοιου είδους «επεξεργαστές» είναι οι σύγχρονες κάρτες γραφικών (GPUs) και σε μικρότερη κλίμακα οι επεξεργαστές που διαθέτουν εντολές τύπου SSE/3DNow!

2.3 MISD (Multiple Instruction, Single Data)

Αρχιτεκτονική που χρησιμοποιείται σε συστήματα στα οποία η ανοχή στα σφάλματα είναι κρίσιμη. Συνήθως, το ίδιο πρόγραμμα ανατίθεται σε δύο ή περισσότερους επεξεργαστές και η σύγκριση των εξόδων τους αναδεικνύει τα πιθανά σφάλματα.

2.4 MIMD (Multiple Instruction, Multiple Data)

Εδώ ανήκει η πλειοψηφία των σύγχρονων παράλληλων υπολογιστών και είναι αυτοί που θα μας απασχολήσουν στην παρούσα διπλωματική. Ανάλογα με τον τρόπο που οργανώνονται οι επεξεργαστές σε σχέση με την μνήμη του συστήματος, οι υπολογιστές MIMD χωρίζονται σε κοινής μνήμης και κατανεμημένης μνήμης. Αυτό έχει άμεση επίδραση στον τρόπο με τον οποίο γράφονται τα προγράμματα για τους υπολογιστές αυτούς.

2.4.1 Παράλληλοι Η/Υ Κοινής Μνήμης

Το χαρακτηριστικό των Η/Υ κοινής μνήμης είναι ότι όλοι οι επεξεργαστές έχουν πρόσβαση σε όλη ανεξαιρέτως την μνήμη του συστήματος μέσω ενός δικτύου διασύνδεσης. Το δίκτυο μπορεί να είναι απλός δίαυλος δεδομένων (χαμηλή κλιμακωσιμότητα) ή δίκτυο τύπου υπερκύβου ή πλέγματος (καλή κλιμακωσιμότητα, υψηλό κόστος). Επίσης, μπορεί και να ενσωματώνεται στο ολοκληρωμένο κύκλωμα όπως γίνεται στους πολυπύρηνους επεξεργαστές. Αν οι επεξεργαστές έχουν ισότιμη πρόσβαση στην μνήμη, ο υπολογιστής ονομάζεται συμμετρικός (SMP). Εάν η καθυστέρηση (*memory latency*) είναι διαφορετική ανάλογα με την διεύθυνση στην οποία γίνεται η πρόσβαση, πρόκειται περί αρχιτεκτονικής NUMA (*Non-Uniform Memory Access*).

Το γεγονός ότι οι επεξεργαστές μπορούν να τροποποιούν δεδομένα μέσα στην cache τους χωρίς να επικοινωνούν με την κύρια μνήμη μπορεί να δημιουργήσει ασυνέπεια στις εικόνες που έχουν για τα κοινά δεδομένα. Η λύση είναι η χρήση ενός πρωτοκόλλου συνέπειας μνήμης, όπως είναι τα MOESI και MESIF, το οποίο υλοποιούν οι ελεγκτές των caches. Συνοπτικά, όταν γράφεται μια cache line ο ελεγκτής το «διαφημίζει» στο κοινό δίαυλο ώστε οι υπόλοιποι να ακυρώσουν την δικιά τους (αν την μοιράζονται). Αντίστοιχα, όταν ένας επεξεργαστής κάνει αίτηση για δεδομένα στην κύρια μνήμη, οι ελεγκτές cache των υπολοίπων «κρυφακούνε» τον δίαυλο και σε περίπτωση που ένας εξ αυτών έχει τα δεδομένα, τα στέλνει απευθείας στον αιτώντα παρακάμπτοντας την κύρια μνήμη. Τέλος να αναφέρουμε ότι υπάρχουν και πρωτόκολλα συνέπειας τύπου καταλόγου τα οποία επιτυγχάνουν καλύτερη κλιμάκωση σε σχέση με τα προαναφερθέντα.

2.4.2 Παράλληλοι Η/Υ Κατανεμημένης Μνήμης

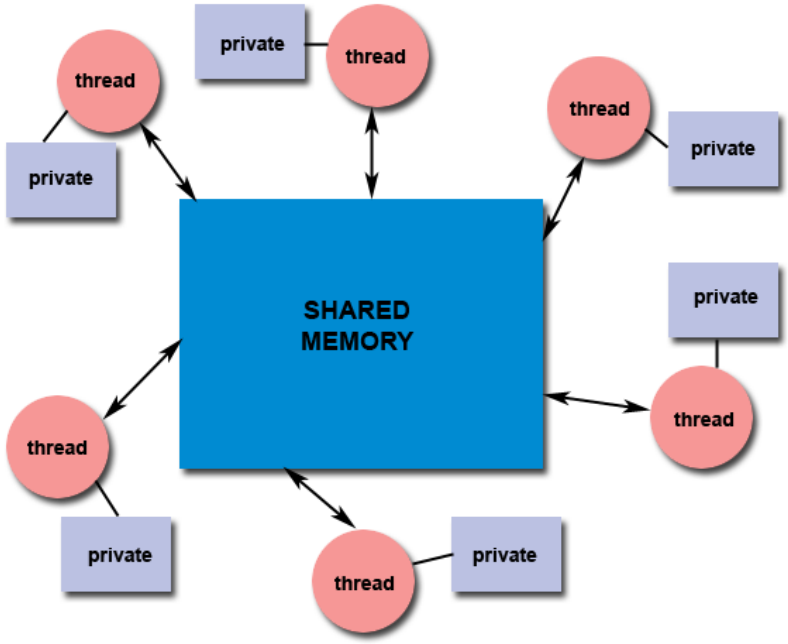
Στους υπολογιστές κατανεμημένης μνήμης κάθε επεξεργαστής έχει πρόσβαση μόνο στην τοπική μνήμη. Για να διαβάσει/γράψει δεδομένα που «κατοικούν» στις μνήμες άλλων υπολογιστών, χρειάζεται να στείλει κάποιου είδους μήνυμα σ'αυτούς πάνω από το δίκτυο που συνδέει τις δύο πλευρές. Εξυπακούεται πως η πρόσβαση στα μακρινά δεδομένα είναι τάξεις μεγέθους πιο δαπανηρή σε σχέση με αυτή στα τοπικά. Το δίκτυο μπορεί να είναι σχετικά συγκεντρωμένο όπως στις συστοιχίες και τους MPPs που αναφέραμε προηγουμένως ή αργό και γεωγραφικά διασπαρμένο όπως το Διαδίκτυο (βλέπε εφαρμογές *SETI@home* και *Folding@home*).

Τα προβλήματα αυτής της κλάσης υπολογιστών είναι πολύ διαφορετικά. Λόγω μεγάλης κλίμακας υπάρχουν πολλά σφάλματα στα οποία η εφαρμογή πρέπει να έχει ανοχή. Ένας επεξεργαστής που δυσλειτουργεί παράγοντας λάθος δεδομένα για παράδειγμα μπορεί να ανιχνευθεί μόνο με αναπαραγωγή της δουλειάς από δεύτερο επεξεργαστή. Επίσης, στην περίπτωση του Διαδικτύου η τοπολογία αλλάζει από στιγμή σε στιγμή και ο κάθε κόμβος δεν έχει παρά μερική εικόνα αυτής. Οπότε οι κατανεμημένες εφαρμογές χρειάζονται σε γενικές γραμμές μεγαλύτερη ευελιξία σε σχέση με αυτές για την κοινή μνήμη.

3 Παράλληλα προγραμματιστικά μοντέλα

3.1 Μοντέλο κοινού χώρου διευθύνσεων

Το μοντέλο προγραμματισμού κοινού χώρου διευθύνσεων δεν είναι παρά μια μεταφορά για τον τρόπο λειτουργίας των υπολογιστών κοινής μνήμης. Η κύρια μονάδα εκτέλεσης είναι το νήμα και κάθε διεργασία αποτελείται από ένα ή περισσότερα νήματα. Τα νήματα μοιράζονται τα καθολικά (σε ορολογία C) δεδομένα του προγράμματος, έχουν όμως τις δικές τους στοίβες εκτέλεσης (*runtime stacks*) και ιδιωτικές τιμές για τους καταχωρητές του επεξεργαστή. Επίσης, κάθε νήμα μπορεί να έχει κάποια προσωπικά δεδομένα τα οποία ονομάζονται threadprivate. Στο σχήμα 3-1 (Barney, 2012) αναπαρίσταται η ταυτόχρονη πρόσβαση των νημάτων στο κοινό χώρο διευθύνσεων.



Σχήμα 3-1: Η πρόσβαση των νημάτων στον κοινό χώρο διευθύνσεων

Ο όρος “κοινός χώρος διευθύνσεων” αναφέρεται στον ενιαίο καθολικό χώρο διευθύνσεων στον οποίο τα νήματα έχουν άμεση πρόσβαση. Η κοινή μνήμη μπορεί να χρησιμοποιηθεί για ανταλλαγή πληροφορίας μεταξύ των νημάτων, συγχρονισμό και τα λοιπά. Χρειάζεται όμως κάποια προσοχή λόγω της παράλληλης πρόσβασης στα ίδια κοινά δεδομένα. Θα δείξουμε δύο χαρακτηριστικά παραδείγματα. Έστω το παρακάτω τμήμα εκτέλεσης:

Thread A	**Thread B**
`count++;`	`count++;`

Λόγω του ότι η εντολή *count++* χρειάζεται τρεις εντολές σε επίπεδο assembly (AT&T syntax),

```
movl  a, %eax
incl  %eax
movl  %eax, a
```

τα δύο νήματα ενδέχεται να διαβάσουν την ίδια αρχική τιμή. Η «προφανής» λύση να συγχρονίσουμε την πρόσβαση στο *count* μέσω τρίτης μεταβλητής,

Thread A	**Thread B**
`count++;`	`while (!flag)`
`flag = 1;`	`;`
	`read(count);`

αποτυγχανει καθότι δεν υπάρχει καμία εγγύηση πως το νήμα Β θα δει την αύξηση. Ο λόγος είναι ότι ο μεταγλωττιστής είναι ελεύθερος να αντιμεταθέσει τις δύο εντολές του νήματος Α για λόγους βελτιστοποίησης. Η εξασφάλιση της ορθότητας του παράλληλου προγράμματος απαιτεί άλλες τεχνικές.

Οι τεχνικές που χρησιμοποιούνται έχουν ως βάση τις ατομικές λειτουργίες, λειτουργίες που ο επεξεργαστής ολοκληρώνει σε ένα βήμα. Μια εντολή τέτοιου τύπου είναι η compare-and-swap η οποία ελέγχει αν η μεταβλητή ισούται με μια τιμή, και αν ναι, της αναθέτει μια νέα τιμή. Οι ατομικές λειτουργίες αποτελούν δομικά στοιχεία για την δημιουργία πολυπλοκότερων δομών οι οποίες εξασφαλίζουν τον σωστό συγχρονισμό ή και τον αμοιβαίο αποκλεισμό, δηλαδή την εκτέλεση ενός τμήματος κώδικα από ένα νήμα τη φορά.

Συνοπτικά, οι δομές υψηλότερου επιπέδου είναι α) τα locks, μεταβλητές που αποκτώνται από ένα νήμα πριν το κρίσιμο τμήμα και απελευθερώνονται μετά, β) οι σημαφόροι (*semaphores*) οι οποίοι αυξάνονται/μειώνονται ατομικά και χρησιμεύουν στον συντονισμό της πρόσβασης σε κάποιο κοινό πόρο και γ) οι mutexes που υλοποιούν τον αμοιβαίο αποκλεισμό και είναι στην ουσία δυαδικοί σημαφόροι. Πάντως, η λάθος χρήση τους μπορεί να οδηγήσει σε αδιέξοδο (*deadlock*) πχ:

Thread A	**Thread B**
`lock(X);`	`lock(Y);`
`lock(Y);`	`lock(X);`

Εδώ τα νήματα εξασφαλίζουν τα locks για τους πόρους Χ και Υ με ανάποδη σειρά. Αν το Α κλείσει το Χ και το Β το Υ ταυτόχρονα, και τα δύο νήματα θα κολλήσουν στο επόμενο lock. Γι'αυτό έχει σημασία η καθιέρωση ορθών πρακτικών όπως η ενιαία σειρά κλειδώματος. Άλλα συνήθη σφάλματα είναι τα εξής:

- Μη απελευθέρωση του πόρου μετά την χρήση
- Απελευθέρωση πόρου που δεν έχει κλειδωθεί
- Χρήση του πόρου χωρίς κλείδωμα

Εν κατακλείδι, το μοντέλο κοινού χώρου διευθύνσεων έχει τα εξής πλεονεκτήματα:

- Ευκολία προγραμματισμού
- Μικρό μέγεθος & καθαρότητα κώδικα
- Δεν χρειάζεται ρητή επικοινωνία μεταξύ των νημάτων

…και τα εξής μειονεκτήματα:

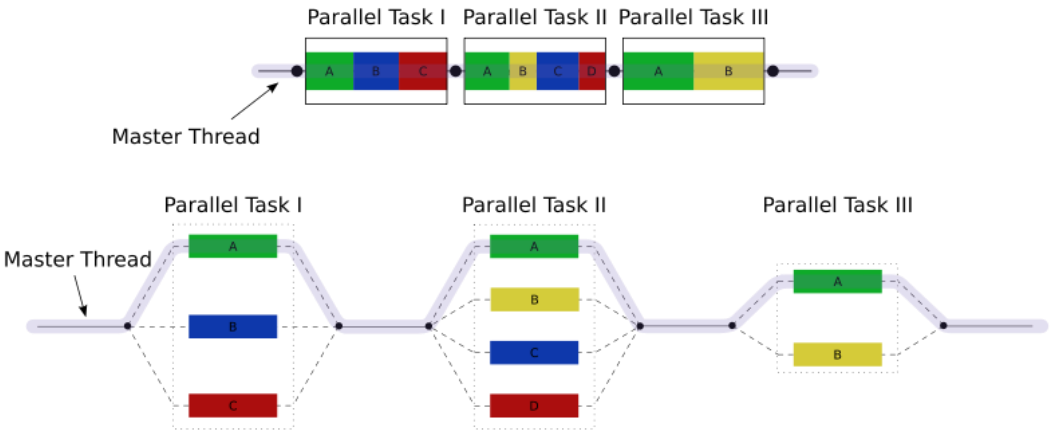
- Δύσκολα ανιχνεύσιμα bugs οπως *race conditions* και *deadlocks*
- Δεν υπάρχει έλεγχος της συνάφειας επεξεργαστή (processor affinity) και της κατανομής δεδομένων (βλέπε NUMA)
- Περιορίζεται στα μηχανήματα κοινής μνήμης, άρα μη κλιμακώσιμο

Παράδειγμα: Open Multi-Processing (OpenMP)

Η OpenMP είναι ένα API για την μετατροπή σειριακών προγραμμάτων σε πολυνηματικά και είναι ο δημοφιλέστερος τρόπος προγραμματισμού υπολογιστών κοινής μνήμης. Ο λόγος είναι η εξαιρετική απλότητα του και η ευρεία υποστήρξη από πληθώρα σημαντικών κατασκευαστών όπως οι AMD, Intel, IBM, HP, Cray κτλ.

Ένα πρόγραμμα OpenMP, εκτός απ'τον σειριακό κώδικα, περιέχει *#pragma directives* και κλήσεις προς έναν μικρό αριθμό συναρτήσεων οι οποίες επιστρέφουν πληροφορίες για την εκτέλεση του προγράμματος, όπως τον αριθμό των νημάτων. Ο προγραμματιστής χρησιμοποιεί τα *directives* για να υποδείξει στον μεταγλωττιστή τα τμήματα του κώδικα που μπορούν να παραλληλοποιηθούν και με ποιο τρόπο. Βάσει αυτών των οδηγιών, ο μεταγλωττιστής μετασχηματίζει τον σειριακό κώδικα σε πολυνηματικό και προσθέτει αυτόματα τον απαραίτητο συγχρονισμό, φράγματα κτλ. Αυτός ο τρόπος προγραμματισμού ευνοεί σε μια μορφή παραλληλισμού ονόματι *data-parallel*, όπου οι επαναλήψεις ενός βρόχου χωρίζονται στα νήματα με στατικό ή δυναμικό τρόπο. Είναι όμως εφικτός ο προγραμματισμός και σε στυλ SPMD με την χρήση οδηγιών κατανομής εργασίας όπως η *#pragma omp sections* και η *#pragma omp task*.

Το μοντέλο εκτέλεσης είναι πολύ απλό: η εκτέλεση του προγράμματος ξεκινά σειριακά με ένα νήμα που αποκαλείται νήμα-αρχηγός (*master thread*). Κάθε φορά που αυτό συναντά μια οδηγία *#pragma omp parallel*, δημιουργεί με *fork* μια νέα ομάδα νημάτων για την παράλληλη εκτέλεση του αντίστοιχου κώδικα και περιμένει στο τέλος μέχρι όλα τα νήματα της ομάδας να κάνουν *join*. Έπειτα η εκτέλεση συνεχίζεται σειριακά και ούτω καθεξής. Η όλη διαδικασία φαίνεται στο σχήμα 3-2. Επιπρόσθετα, κάθε παράλληλο τμήμα μπορεί να περιέχει και άλλες οδηγίες όπως η *#pragma omp sections* για διαμερισμό της εργασίας ή η *#pragma omp for* για μοίρασμα των επαναλήψεων του βρόχου. Κανονικά όλες οι μεταβλητές θεωρούνται κοινές εκτός αν ο προγραμματιστής τις δηλώσει ως private οπότε κάθε νήμα θα αποκτήσει το δικό του thread-private αντίγραφο. Ο συγχρονισμός για την δρομολόγηση εργασιών και την πρόσβαση στα κοινά δεδομένα επιτυγχάνεται με άλλες οδηγίες όπως οι *#pragma omp barrier, #pragma omp atomic* και *#pragma omp wait*.



Σχήμα 3-2: Το μοντέλο εκτέλεσης της OpenMP

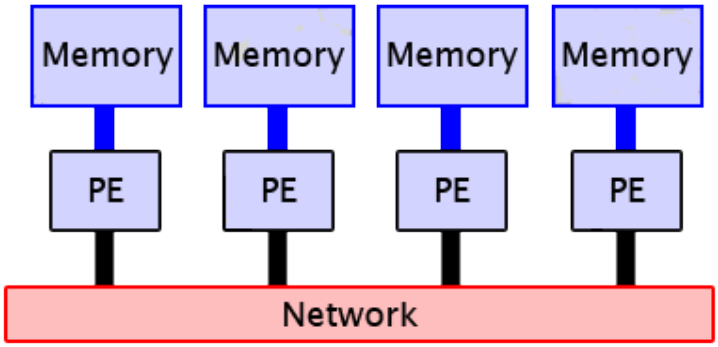
Στον τομέα της ταχύτητας, η θεωρητική επιτάχυνση για την OpenMP ισούται με *N*, όπου *N* ο αριθμός των νημάτων (εξ ορισμού ίσος με τον αριθμό των πυρήνων του επεξεργαστή). Στην πράξη, αυτή είναι κατά πολύ μικροτερη κυρίως λόγω περιορισμών της ίδιας της αρχιτεκτονικής κοινής μνήμης. Ο τρόπος που οι πυρήνες μοιράζονται την cache μπορεί να οδηγήσει σε φαινόμενα *cache-interference* και η τήρηση της συνάφειας είναι μία επιπλέον επιβάρυνση. Επιπλέον, αν το πρόβλημα δεν είναι *embarrassingly parallel*, προστίθεται το κόστος συγχρονισμού. Τέλος, αν η εκτέλεση λαμβάνει μέρος σε μηχάνημα SPM με πολλούς πυρήνες, υπάρχει το ενδεχόμενο κορεσμού του διαθέσιμου *memory bandwidth* με την επακόλουθη καθυστέρηση.

Εκεί που η OpenMP ξεχωρίζει είναι στο πολύ χαμηλό κόστος ανάπτυξης τόσο σε χρόνο όσο και σε χρήμα. Το παράλληλο πρόγραμμα διαφέρει ελάχιστα απ'το σειριακό και η μετατροπή μπορεί να γίνει σταδιακά επιταχύνοντας μόνο τα αργότερα σημεία κάθε φορά. Ακόμα και προγραμματιστές νέοι στον χώρο μπορούν εύκολα να γράψουν το πρώτο τους προγράμμα σε OpenMP αρκεί να γνωρίζουν 4 έως 5 *#pragma directives* και βασικές αρχές λειτουργίας ενός υπολογιστή κοινής μνήμης. Σχετικά με το τελευταίο, ιδιαίτερη προσοχή χρειάζεται στο γεγονός ότι η OpenMP δεν υποθέτει τίποτα για την οργάνωση της μνήμης οπότε ακόμα κι αν ο υπολογιστής εφαρμόζει κάποιο πρωτόκολλο συνέπειας, είναι αναγκαία η χρήση του *#pragma omp flush* τόσο πριν το διάβασμα όσο και μετά το γράψιμο μιας κοινής μεταβλητής. Ο λόγος είναι ότι ο μεταγλωττιστής μπορεί να τοποθετήσει μια κοινή μεταβλητή σε καταχωρητή για λόγους βελτιστοποίησης κάνοντας τις τροποποιήσεις αόρατες στην cache.

Όσον αφορά την φορητότητα, η OpenMP παίρνει άριστα με τον προφανή περιορισμό ότι μιλάμε για αρχιτεκτονικές κοινής μνήμης. Δεδομένου ότι η πλειοψηφία των αρχιτεκτονικών είναι κατανεμημένες, η OpenMP δεν δύναται να διεκδικήσει τη θέση της παράλληλης γλώσσας του μέλλοντος.

3.2 Μοντέλο ανταλλαγής μηνυμάτων

Το μοντέλο αυτό αντιστοιχεί κυρίως στους Η/Υ κατανεμημένης μνήμης. Σύμφωνα με αυτό, δύο διεργασίες μπορούν να επικοινωνήσουν μόνο μέσω της ανταλλαγής μηνυμάτων – στην πράξη αν και οι δύο ανήκουν στον ίδιο κόμβο, μπορούν να ανταλλάξουν μηνύματα μέσω της κοινής μνήμης για μεγαλύτερη αποδοτικότητα. Γενικώς, μια διεργασία που τρέχει σε ένα κόμβο του συστήματος έχει άμεση πρόσβαση μόνο στα δεδομένα της τοπικής μνήμης. Για την πρόσβαση στα δεδομένα των άλλων κόμβων απαιτείται η αποστολή μηνυμάτων πάνω από ένα δίκτυο διασύνδεσης όπως αυτό του σχήματος 3-3 (Parallel Programming Paradigms, 2012)



Εικόνα 3-3: Οργάνωση κατανεμημένης μνήμης

Όλα τα πλεονεκτήματα & μειονεκτήματα του μοντέλου πηγάζουν από τον πλήρη έλεγχο που ασκεί ο προγραμματιστής στην κατανομή των δεδομένων και στον τρόπο και χρόνο στον οποίο επικοινωνούν οι διεργασίες:

- Εύκολη μοντελοποίηση & πρόβλεψη της απόδοσης του προγράμματος
- Πολύ καλή απόδοση σε όλες τις σύγχρονες αρχικτεκτονικές (λόγω ταιριάσματος του αλγορίθμου πάνω στο hardware)
- Κλιμακωσιμότητα
- Απουσία bugs τύπου race condition

Όσον αφορά τα μειονεκτήματα, αυτά είναι:

- Δυσκολία προγραμματισμού: η διαχείριση της επικοινωνίας και ο καταμερισμός των δεδομένων στους κόμβους είναι κοπιώδης δουλειά
- Τα προηγούμενα μπορεί να καταλάβουν έως και 90% του προγράμματος. Αποτέλεσμα: συσκότιση, μεγάλο μήκος κώδικα
- Κίνδυνος deadlock εάν χρησιμοποιείται blocking επικοινωνία

Παράδειγμα: Message Passing Interface (MPI)

Η MPI, όπως φαίνεται και απ'το όνομα της, ακολουθεί το μοντέλο ανταλλαγής μηνυμάτων και είναι το αντίστοιχο της OpenMP για τους υπολογιστές κατανεμημένης μνήμης. Η επιτυχία της οφείλεται τόσο στην απλότητα του μοντέλου προγραμματισμού που ομοιάζει στο σειριακό μοντέλο όσο και στην μεγάλη πρόσοχη που δόθηκε απ'τους σχεδιαστές στη φορητότητα του προτύπου. Εξ ου και οι πολλές και ολοκληρωμένες υλοποιήσεις, όπως η MPICH και η OpenMPI (αποτέλεσμα συνένωσης της LAM/MPI με άλλες υλοποιήσεις).

Το πρότυπο περιγράφει το interface και τις λειτουργίες μιας βιβλιοθήκης συναρτήσεων οι οποίες υλοποιούν την κατανεμημένη επικοινωνία – ο αριθμός αυτών δεν ξεπερνά τις 275, μαζί με αυτές που ορίζονται στην τελευταία έκδοση (*MPI*-2) και προσθέτουν παράλληλο I/O, δυναμική διαχείριση διαδικασιών και μονόπλευρη επικοινωνία. Στην πραγματικότητα, για την πλειοψηφία των προγραμμάτων MPI δεν χρειάζεται παρά ένας μικρός πυρήνας από 6 συναρτήσεις εκ των οποίων οι 4 αφορούν την αρχικοποίηση του προγράμματος και οι υπόλοιπες είναι οι *MPI_Send()* και *MPI_Recv()*. Το πρότυπο περιέχει επίσημα bindings για τις γλώσσες C/C++ και Fortran, ενώ υπάρχουν και ανεπίσημα για τις γλώσσες Python, OCaml και Java.

Ένα πρόγραμμα MPI συνήθως γράφεται στο στυλ SPMD (single program, multiple data) αν και υπάρχει η επιλογή για τελείως διαφορετικά προγράμματα που συνεργάζονται μεταξύ τους. Στην περίπτωση SPMD, ο κώδικας που εκτελεί η κάθε διεργασία διαφοροποιείται βάσει της τιμής μεταβλητών όπως είναι ο αριθμός της διεργασίας που επιστρέφεται απ' την *MPI_Get_rank()*. Η επικονωνία μεταξύ διεργασιών είναι ως επί το πλείστον αμφίπλευρη, με εξαίρεση τις επεκτάσεις της MPI-2, και στηρίζεται στο ταίριασμα του *MPI_Send()* του αποστολέα με κάποιο *MPI_Recv()* απ'την πλευρά του παραλήπτη. Η αποστολή ολοκληρώνεται μόνο εάν οι δύο πλευρές προσδιορίζουν τα ίδια δεδομένα (πχ 2 MPI_INT), το ίδιο tag και το ίδιο *communicator*. Ο τελευταίος όρος είναι η αφαίρεση που χρησιμοποιεί η MPI για μια ομάδα διεργασιών που επικοινωνούν ξεχωριστά και βοηθάει στον λογικό διαχωρισμό σε παράλληλα «σύμπαντα επικοινωνίας». Να συμπληρώσουμε ότι η MPI παρέχει μεγάλη ποικιλία συναρτήσεων για την αποστολή των δεδομένων όπως blocking send, σύγχρονη αποστολή, αποστολή- αν-ο-παραλήπτης-είναι-έτοιμος, αποστολή-μέσω-buffer και ούτω καθεξής – η αποδοτικότητα της κάθεμίας εξαρτάται απ'το μηχάνημα.

Γενικώς, η MPI έχει το πλεονέκτημα έναντι άλλων γλωσσών στον τομέα των επιδόσεων και αυτό γιατί επιτρέπει τον πλήρη έλεγχο της κατανομής δεδομένων και του τρόπου επικοινωνίας. Έτσι ο προγραμματιστής μπορεί να σχεδιάσει τον αλγόριθμο με τρόπο που να εκμεταλλεύεται πλήρως το εκάστοτε hardware για την επίτευξη σχεδόν βέλτιστης απόδοσης. Η άλλη πλευρά του νομίσματος είναι ότι ο αλγόριθμος ταυτίζεται με την υλοποίησή του και είναι δύσκολο να αλλάξει στο μέλλον. Όπως θα δούμε παρακάτω, η γλώσσα Chapel διαπρέπει ακριβώς σε αυτό τον τομέα.

Η MPI είναι μάλλον δύσχρηστη ως γλώσσα παρ'όλη την απλότητά της. Αυτό οφείλεται στην επιβάρυνση του προγραμματιστή με ένα σωρό λεπτομέρειες πέραν της σχεδίασης του κυρίως αλγορίθμου. Όλα, απ'τον καταμερισμό των δεδομένων στις διεργασίες ως τον σχεδιασμό του μοτίβου επικοινωνίας που μπορεί να είναι πολύ πολύπλοκος για ορισμένες εφαρμογές, περνάνε απ'το χέρι του. Μάλιστα το μεγάλο μέγεθος του κώδικα, όπως θα δούμε και αναλυτικά, αυξάνει τις πιθανότητες για δυσεύρετα λάθη και άρα τον χρόνο ανάπτυξης. Μια δεύτερη αρνητική παράμετρος είναι το χαμηλό επίπεδο των host-γλωσσών. Καμία εξ αυτών δεν σχεδιάστηκε εξαρχής με τον παραλληλισμό μεγάλης κλίμακας υπόψη γι'αυτό και απουσιάζουν οι δομές που θα διευκόλυναν κατά πολύ τον επίδοξο προγραμματιστή παράλληλων εφαρμογών. Τέλος, μπορεί η *λογική* του παράλληλου προγράμματος να είναι παρόμοια με αυτή του σειριακού, η μετατροπή όμως από σειριακό σε παράλληλο δεν είναι καθόλου ασήμαντη υπόθεση (εν αντιθέσει με OpenMP και Chapel).

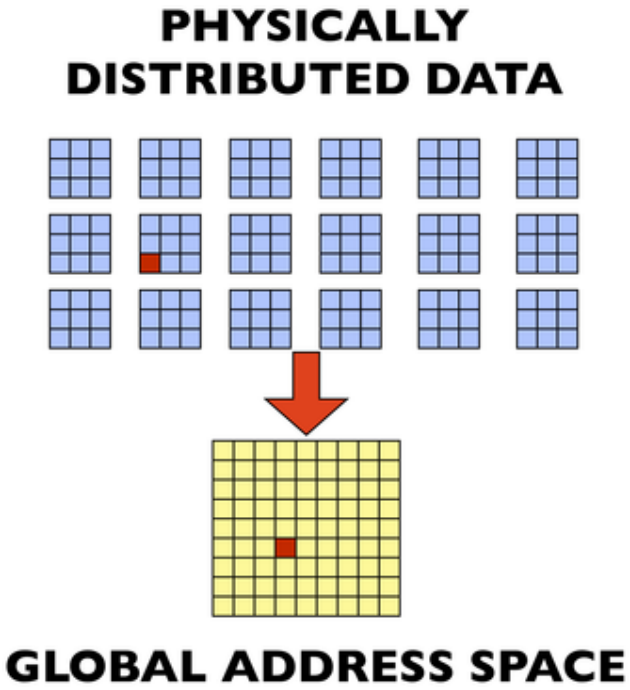
Η φορητότητα της MPI είναι κακή λόγω της εξάρτησης του αλγορίθμου απ'την κατανομή των δεδομένων και την τοπολογία των κόμβων και η μεταφορά αυτού σε άλλο μηχάνημα ενέχει πάντα τον κίνδυνο αλλαγής μεγάλων κομματιών του κώδικα έναντι μόνο κάποιων γραμμών στην Chapel. Η ευρωστία της MPI είναι επίσης χαμηλή λόγω της δυσκολίας προγραμματισμού που αναφέραμε και όχι εξαιτίας κάποιας σχεδιαστικής αδυναμίας. Η διαχείριση σε τόσο χαμηλό επίπεδο συσκοτίζει παρά βοηθάει στην κατανόηση του αλγορίθμου και βεβαίως οδηγεί σε ουκ ολίγα bugs. Εδώ να σημειώσουμε και ένα μεγάλο πλεονέκτημα της MPI σε σχέση με τους ανταγωνιστές που είναι η ωριμότητα της. Πράγματι υπάρχουν άπειρα εργαλεία τα οποία έχουν γραφτεί για C, Fortran κτλ και μπορούν να επαναχρησιμοποιηθούν απλά με κάποιες παράλληλες επεκτάσεις πχ debuggers, βελτιστοποιημένοι μεταγλωττιστές, profilers και άλλα.

3.3 Υβριδικό μοντέλο

Το υβριδικό μοντέλο είναι μια μείξη των δύο προηγούμενων μοντέλων και κληρονομεί τα μειονεκτήματα και τα πλεονεκτήματα των δύο προσεγγίσεων. Παράδειγμα αυτού είναι ο προγραμματισμός σε MPI+OpenMP που είναι και ο πλέον διαδεδομένος τρόπος παράλληλου προγραμματισμού σήμερα. Προφανές μειονέκτημα είναι η ανάγκη εκμάθησης δύο τελείως διαφορετικών «ιδιωμάτων».

3.4 Μοντέλο PGAS

Το μοντέλο PGAS (Partitioned Global Address Space) κατά μία έννοια επεκτείνει το μοντέλο κοινού χώρου διευθύνσεων στους υπολογιστές κατανεμημένης μνήμης. Συγκεκριμένα, ένα τμήμα της μνήμης του συστήματος σχηματίζει ένα (λογικό) καθολικό χώρο διευθύνσεων στον οποίο έχουν πρόσβαση όλα τα νήματα ανεξαρτήτως του κόμβου στον οποίο τρέχουνε. Παρόλο που κάθε διεύθυνση του κοινού τμήματος σχετίζεται (φυσικά) με ένα και μόνο κόμβο, το μοντέλο συμπεριφέρεται σε όλες τις διευθύνσεις ως ισοδύναμες, αποκρύπτει δηλαδή την επικοινωνία (όχι όμως και το κόστος της) από τον προγραμματιστή. Η εικόνα που έχει ο προγραμματιστής για την μνήμη φαίνεται στην εικόνα 3-4 (Markidis, 2012)

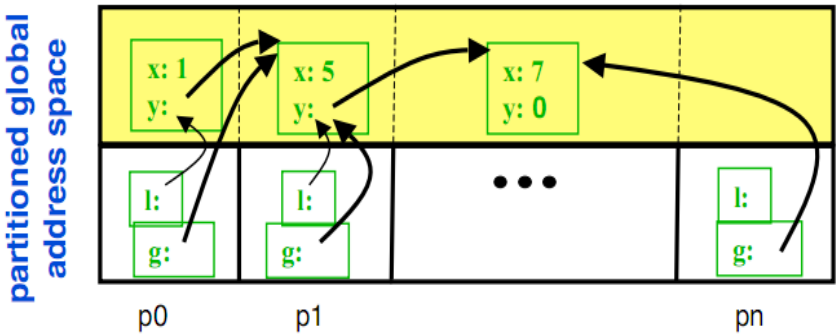


Εικόνα 3-4: Η μνήμη στο μοντέλο PGAS

Παράδειγμα: Unified Parallel C (UPC)

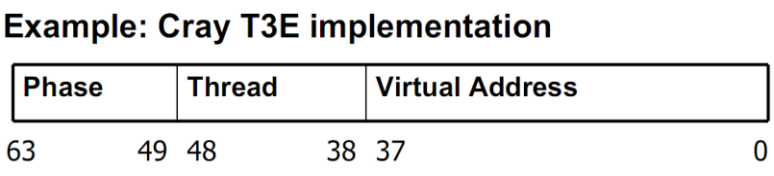
Οι κύριες γλώσσες που υλοποιούν το κλασικό μοντέλο PGAS είναι η UPC, η CAF (Co-array Fortran) και η Titanium που επεκτείνει την Java. Το μοντέλο αυτό είναι το νεότερο ιστορικά παράλληλο μοντέλο και είναι πολύ σημαντικό πολύ σημαντικό, για την ακρίβεια αποτελεί την βάση για το μοντέλο APGAS (Asynchronous PGAS) των HPCS γλωσσών, θα παρουσιάσουμε τα κοινά χαρακτηριστικά των αρχικών PGAS γλωσσών με συγκεκριμένα παραδείγματα από την γλώσσα UPC βάσει του (El-Ghazawi, 2012).

Καταρχήν, και οι τρεις γλώσσες χρησιμοποιούν το σύστημα τύπων για να διαφοροποιήσουν ρητά κατά την ώρα μεταγλώττισης τις μακρινές από τις κοντινές αναφορές στη μνήμη, κάτι που βοηθάει στην επίτευξη υψηλής απόδοσης. Στην UPC, αυτό επιτυγχάνεται με τη προσθήκη του qualifier *shared* μπροστά από μια μεταβλητή, δείκτη ή πίνακα. Ένας shared πίνακας μοιράζεται σε όλα τα νήματα με round-robin ή block-cyclic κατανομή και ανήκει εξ ολοκλήρου στον καθολικό χώρο διευθύνσεων. Οι ιδιωτικοί δείκτες (default) μπορούν να δείχνουν μόνο σε τοπικά δεδομένα ή στα *τοπικά* τμήματα του καθολικού χώρου. Αντιθέτως, ένας δείκτης-σε-shared μπορεί να δείχνει οπουδήποτε μέσα στον καθολικό χώρο. Η εικόνα 3-5 (Yelick, Parallel Languages: Past, Present and Future, 2007) δείχνει τρόπους πρόσβασης στην «κοινή» μνήμη μέσω *local* και *global* δεικτών.



Εικόνα 3-5: Είδη δεικτών και αναφορές στην γλώσσα UPC

Αν το μηχάνημα δεν μπορεί να υποστηρίξει την καθολική μνήμη σε επίπεδο hardware, ο κόμβος κωδικοποιείται *μέσα* στον καθολικό δείκτη και σε κάθε έμμεση αναφορά αποκωδικοποιείται και συγκρίνεται με τον τωρινό προσθέτωντας σημαντική επιβάρυνση. Η στατική πληροφορία που αναφέραμε προηγουμένως βοηθάει τον μεταγλωττιστή να αλλάζει τους καθολικούς δείκτες με απλούς όπου είναι δυνατόν. Στην εικόνα 3-6 (El-Ghazawi, 2012) φαίνεται η δομή του καθολικού δείκτη της UPC στον Cray T3E (*Phase* είναι η διεύθυνση μέσα στο μπλοκ που δείχνει η *Virtual Address*).



Εικόνα 3-6: Καθολικός δείκτης στον Cray T3E

Η πιο σημαντική δομή της UPC είναι η *upc_forall()* η οποία μοιράζει τον χώρο επαναλήψεων στους διάφορους κόμβους βάσει μιας επιπλέον, σε σχέση με το απλό for, έκφρασης η οποία καθορίζει τη συνάφεια. Δηλαδή για δείκτη βρόχου i, η έκφραση i % THREADS θα μοιράσει τους δείκτες του βρόχου κυκλικά σε όλους τους κόμβους (στη UPC το THREADS δίνει τον αριθμό των νημάτων που εκτελούν το πρόγραμμα). Πέρα απ'αυτό η UPC ακολουθεί τον δρόμο της MPI και παρέχει συλλογικές συναρτήσεις για συγχρονισμό και ανταλλαγή ιδιωτικών δεδομένων, όπως οι *upc_barrier(), upc_broadcast(), upc_all_exchange()* και *upc_reduce()*.

Οι άλλες PGAS γλώσσες είναι παρόμοιες με την UPC όσον αφορά το μοντέλο εκτέλεσης αλλά διαφέρουν σε αυτά που παρέχουν. Για παράδειγμα, η Titanium που βασίζεται στην Java κληρονομεί το υψηλό επίπεδο και παρέχει πολυδιάστατους πίνακες που υποστηρίζουν iterators, υποπίνακες και αντιγραφή. Επίσης, *single* μεταβλητές που χρησιμοποιούνται στον συγχρονισμό των νημάτων, overloading τελεστών και regions στις οποίες μπορούμε να αντικαταστήσουμε τον garbage collector με έναν πιο αποδοτικό αλγόριθμο.

Ο μεγάλος περιορισμός και των τριών κλασσικών PGAS γλωσσών είναι ότι ο τόπος και χρόνος δημιουργίας των νημάτων καθορίζεται στην έναρξη της εκτέλεσης και δεν μπορεί να αλλάξει (σαν την MPI) και γι'αυτό είναι μάλλον ακατάλληλες όταν τα προβλήματα έχουν δυναμικά σύνολα δεδομένων. Από την άλλη, το σύστημα εκτέλεσης (*runtime system*) δεν χρειάζεται να είναι πολύπλοκο όπως συμβαίνει στις HPCS γλώσσες. Τέλος, οι κατανομές που υποστηρίζονται είναι στοιχειώδεις, όπως *block*, *cyclic* και *block-cyclic*. Όπως θα δούμε στο δεύτερο σκέλος της διπλωματικής, ο περιορισμός αυτός δεν ισχύει για την Chapel, απεναντίας ο χρήστης μπορεί να ορίσει ακόμα και τις δικές του κατανομές!

4 Η νέα γενιά παράλληλων γλωσσών

Στο κεφάλαιο αυτό περνάμε στις νέες προγραμματιστικές γλώσσες οι οποίες προέκυψαν μέσα από το πρόγραμμα HPCS της DARPA και οι οποίες βασίζονται – και επεκτείνουν – το μοντέλο PGAS που περιγράψαμε στο προηγούμενο κεφάλαιο. Το μεγαλύτερο μέρος του κεφαλαίου αφιερώνεται στην περιγραφή της γλώσσας Chapel η οποία βρίσκεται στο επίκεντρο της εργασίας αυτής.

4.1 Το πρόγραμμα HPCS

Το 2002, η DARPA ανακοίνωσε ένα νέο φιλόδοξο πρόγραμμα με το όνομα HPCS (High Productivity Computing Systems) με σκοπό την επανάκτηση της πρωτοκαθεδρίας των ΗΠΑ στον τομέα του HPC (High Performance Computing) και την χρήση της νέας τεχνολογίας για την ανάπτυξη στιβαρών παράλληλων εφαρμογών στους τομείς της άμυνας και της βιομηχανίας. Το επαναστατικό στοιχείο του προγράμματος ήταν η αναγνώριση – για πρώτη φορά – του κόστους που συνεπάγεται η ίδια η ανάπτυξη του παράλληλου λογισμικού, τόσο σε χρόνο όσο και σε χρήμα, και η συμπερίληψη ρητώς της υψηλής παραγωγικότητας στις απαιτήσεις. Τα τέσσερα κριτήρια που έθεσε η DARPA για την αξιολόγηση του νέου υπολογιστικού συστήματος (software και hardware) είναι τα εξής:

- Καλές επιδόσεις σε μια ευρεία γκάμα κοινών αλγορίθμων
- Μείωση του κόστους και του χρόνου ανάπτυξης του λογισμικού
- Καλή φορητότητα των αλγορίθμων
- Ευρωστία και ασφάλεια ενάντια σε έξωθεν επιθέσεις

Το πρόγραμμα HPCS είχε ως αποτέλεσμα την δημιουργία 3 νέων γλωσσών: της X10 (IBM), της Fortress (Sun) και της Chapel (Cray). Από αυτές μόνο η X10 βασίστηκε σε προϋπάρχουσα γλώσσα (Java), ενώ οι δημιουργοί της Fortress πειραματίστηκαν με την μαθηματική διατύπωση του ίδιου του κώδικα μέσω της χρήσης Unicode (σημ. η Fortress έχει αποκλειστεί από την 3η φάση του προγράμματος). Όλες οι εταιρίες βάσισαν τις γλώσσες τους στις αντίστοιχες αρχιτεκτονικές hardware που ανέπτυξαν στα πλαίσια του ίδιου προγράμματος. Για παράδειγμα, η Cray σχεδίασε την Chapel έτσι ώστε να μπορεί να προγραμματίσει τον μελλοντικό της υπερυπολογιστή με ονομασία Cascade. Ταυτόχρονα όμως και οι τρεις γλώσσες καλούνται να έχουν, αν όχι βέλτιστες, τουλάχιστον αξιοπρεπείς επιδόσεις στις αρχιτεκτονικές που αποτελούν την πλειοψηφία των σημερινών παράλληλων υπολογιστών (Yelick & Lusk, 2007).

Παρ'όλες τις διαφορές τους, οι τρεις γλώσσες ανήκουν στην ίδια οικογένεια και εισάγουν παρόμοια προχωρημένα χαρακτηριστικά τα οποία μέχρι τώρα αποτελούσαν προνόμιο των σειριακών γλωσσών. Εμείς στην παρούσα διπλωματική θα ασχοληθούμε με την Chapel ως την πλέον ελπιδοφόρα της ομάδας και θα προσπαθήσουμε να αξιολογήσουμε τις πιθανότητες που έχει να καταστεί μια αξιόλογη εναλλακτική στον χώρο της ανάπτυξης παράλληλων εφαρμογών γενικού σκοπού.

4.2 Η γλώσσα Chapel

Αν έπρεπε να περιγράψουμε την Chapel με μία πρόταση θα λέγαμε ότι είναι μία αντικειμενοστρεφής παράλληλη γλώσσα προγραμματισμού με δυνατότητες γενικού προγραμματισμού (*generic programming*). Το μοντέλο μνήμης στο οποίο βασίζεται είναι το Ασύγχρονο PGAS (APGAS), το οποίο σημαίνει PGAS με δυνατότητα δημιουργίας νημάτων κατά την ώρα εκτέλεσης (*dynamic tasks*). Όπως και στις κλασικές PGAS γλώσσες, έτσι κι εδώ υπάρχει μία *ενιαία καθολική όψη* τόσο για τις δομές δεδομένων όσο και για τη ροή του ελέγχου (*global-view programming*). Όμως, ο τρόπος αυτός προγραμματισμού δεν είναι δεσμευτικός και ο προγραμματιστής μπορεί αν επιθυμεί να προγραμματίσει σε στυλ SMPD ακριβώς όπως στην MPI. Εκμεταλλευτήκαμε αυτή την ελευθερία για να υλοποιήσουμε ορισμένους απ'τους αλγορίθμους που δεν ήταν εφικτό να εκφραστούν με μορφή καθολικής όψης. Ο πίνακας 4-1 (Chamberlain B. , Programming Models and Chapel: Landscaping for Exascale Computing, 2011) συνοψίζει τις κύριες διαφορές μεταξύ των παράλληλων γλωσσών που έχουμε αναφέρει ως τώρα, με το μοντέλο μνήμης να αντιστοιχεί στο μοντέλο προγραμματισμού που περιγράψαμε στο κεφάλαιο 2.

	memory model	*programming model*	*execution model*	*data structures*	*communication*
MPI	distributed memory	cooperating executables (often SPMD in practice)		manually fragmented	APIs
OpenMP	shared memory	global-view parallelism	shared memory multithreaded	shared memory arrays	N/A
PGAS Languages: CAF	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
PGAS Languages: UPC				1D block-cyc arrays/ distributed pointers	implicit
PGAS Languages: Titanium				class-based arrays/ distributed pointers	method-based
Chapel	PGAS	global-view parallelism	distributed memory multithreaded	global-view distributed arrays	implicit

Πίνακας 4-1: Χαρακτηριστικά των κύριων παράλληλων γλωσσών

Στις επόμενες σελίδες θα περιγράψουμε συνοπτικά τα κύρια χαρακτηριστικά της Chapel που την διαφοροποιούν από τις περισσότερες άλλες γλώσσες έτσι ώστε ο αναγνώστης να εξοικιωθεί με αυτή και να μπορεί να ακολουθήσει τον πηγαίο κώδικα που περιλαμβάνεται σε αυτή την εργασία. Οι περισσότερες πληροφορίες στο κεφάλαιο αυτό βασίζονται στο official specification της Chapel (Chapel Language Specification Version 0.91, 2012) καθώς και σε διάφορες παρουσιάσεις της Cray.

Δηλώσεις και τύποι μεταβλητών

Στην Chapel οι μεταβλητές και οι σταθερές δηλώνονται με τις λέξεις-κλειδιά **var** και **const**, ακολουθούμενες απ'το όνομα της μεταβλητής/σταθεράς και τον τύπο. Αν ο προγραμματιστής παραλείψει τον τύπο, η Chapel τον συνάγει αυτόματα βάσει του τύπου της έκφρασης αρχικοποίησης (που μπορεί να είναι συνάρτηση κτλ) και αρχικοποιεί αυτόματα την μεταβλητή σε 0, 0.0, "" και ούτω καθεξής (η αυτόματη αρχικοποίηση ισχύει και για τους σύνθετους τύπους όπως πίνακες και κλάσεις). Αν τέλος προηγείται η λέξη-κλειδί **config**, η αντίστοιχη μεταβλητή μπορεί να αρχικοποιηθεί μέσω της γραμμής εντολών. Στον κώδικα που ακολουθεί δείχνουμε μερικούς από τους τύπους της Chapel.

```
var i, j = 5, pi = 3.14;     // i, j είναι ints με τιμή 5, pi είναι real
config const greet = "hi";   // ./a.out --greet "bye"
var t = (7, "dwarves");      // t είναι tuple, t(1) = 7, t(2) = "dwarves"

var r = 1..j;                // το r έχει τύπο range (είδος iterator)
for i in r do
  write(i);                  // "12345"
for i in r by 2 do
  write(i);                  // "135"
```

Domains ή αλλιώς πεδία ορισμού

Ο τύπος **domain** είναι ο ακρογωνιαίος λίθος της Chapel γιατί βάσει αυτού γίνεται ο ορισμός των πινάκων και η κατανομή τους στους διάφορους κόμβους. Ορίζεται ως το σύνολο δεικτών στους οποίους αντιστοιχούν τα στοιχεία ενός ή περισσότερων πινάκων. Αντίθετα με άλλες γλώσσες, ο τύπος των δεικτών δεν περιορίζεται σε ακέραιους αλλά μπορεί να είναι οτιδήποτε, ακόμα και κλάσεις· σ'αυτή την περίπτωση λέμε ότι το *domain* είναι συσχετιστικό (*associative*) και μοιάζει με τα *dictionaries* της γλώσσας Python. Τέλος, η αλλαγή ενός *domain* κατά την ώρα εκτέλεσης γίνεται αμέσως ορατή σε όλους τους πίνακες που έχουν οριστεί βάσει αυτού: αν προσθέσουμε κάποιους δείκτες, οι πίνακες αποκτούν τα αντίστοιχα στοιχεία με την default τους τιμή, ενώ αν αφαιρέσουμε δείκτες, οι πίνακες χάνουν τα στοιχεία που αντιστοιχούν σε αυτούς. Το παρακάτω παράδειγμα αποσαφηνίζει τον τρόπο λειτουργίας των *domains*.

```
const D = [1..10, 1..10];     // τύπος domain
var A, B, C: [D] int;         // 3 ακέραιοι πίνακες με δείκτες ∈ D

for (i,j) in D do
  A[i,j] = 10*i + j;

for idx in D do               // Ο B παίρνει τις ίδιες τιμές (idx: tuple)
  B[idx] = 10*idx(1) + idx(2);

const innerD = [2..9, 2..9];
C[innerD] = 42;               // ανάθεση στο εσωτερικό του C
```

Έλεγχος της τοποθεσίας

Στις σύγχρονες πολυεπίπεδες αρχιτεκτονικές, ο έλεγχος της τοποθεσίας όπου εκτελούνται τα νήματα και κατανέμονται τα δεδομένα είναι κρίσιμος για την επίτευξη καλής απόδοσης. Η σημασία του αντικατοπτρίζεται στην Chapel με τον ξεχωριστό τύπο μεταβλητής για την αναπαράσταση της τοποθεσίας ονόματι **locale**. Ο τύπος αυτός αντιστοιχεί σε ένα «κόμβο» του οποίου οι επεξεργαστές έχουν ισότιμη ή *σχεδόν* ισότιμη πρόσβαση στη μνήμη (πχ SMP ή NUMA επεξεργαστής). Ο προγραμματιστής μπορεί να πάρει την πληροφορία αυτή από τον προκαθορισμένο πίνακα **Locales** ο οποίος περιέχει όλα τα *locales* στα οποία τρέχει το πρόγραμμα. Επίσης, χρησιμοποιώντας την έκφραση **on**, μπορεί να εκτελέσει οποιαδήποτε εντολή σε κάποιο *locale* άλλο από το 0 όπου ξεκινάει η εκτέλεση του προγράμματος. Στο παρακάτω, το **here** επιστρέφει το *locale* στο οποίο εκτελείται η εντολή.

```
// Ορισμός του πίνακα Locales (το LocaleSpace ισούται με 1..numLocales)
// const Locales: [LocaleSpace] locale;

writeln("I start on " , here.id);   // "I start on 0"
on Locales[1] do
  writeln("now on " , here.id);     // "now on 1"
writeln("back on " , here.id);      // "back on 0"
```

Παραλληλισμός Δεδομένων

Ο *παραλληλισμός καθολικής όψης* ή αλλιώς παραλληλισμός δεδομένων εμφανίζει πολλές ομοιότητες με τον αντίστοιχο στην γλώσσα UPC. Μπορεί να είναι φανερός όπως όταν χρησιμοποιούμε την έκφραση **forall** (είδος παράλληλου *for*) ή κρυφός όπως όταν έχουμε πράξεις μεταξύ πινάκων και **reductions** πάνω σε *iterators* ή πίνακες. Πάντα η παράλληλη εκτέλεση εντός *locale* χρησιμοποιεί τόσα νήματα όσα και οι πυρήνες του *locale* (εκτός αν έχουμε ορίσει διαφορετικά). Το επόμενο παράδειγμα δείχνει κάποιες περιπτώσεις όπου έχουμε παραλληλισμό δεδομένων.

```
const D = [1..100];
const A: [D] int;       // ο πίνακας A αρχικοποιείται αυτόματα με 0

// Όλες οι εκφράσεις που ακολουθούν εκτελούνται από πολλά νήματα,
// ο αριθμός των οποίων ισούται με τον αριθμό των πυρήνων του locale.

forall idx in D do      // κάθε νήμα τρέχει ένα τμήμα των δεικτών.
  A(idx) = 1;

// Το forall γράφεται ισοδύναμα και ως: [idx in D] A(idx) = 1;
// Επίσης, άλλος τρόπος επίτευξης του παραπάνω είναι ο εξής:
// [a in A] a = 1;   ή ακόμα απλούστερα A = 1;

// Τα reductions είναι επίσης παράλληλα και υποστηρίζουν τις εξή πράξεις:
// *,&&,||,&,|,^,min,max,minloc,maxloc. Η έξοδος του writeln είναι 100.
writeln(+ reduce A);
```

Αν τώρα αλλάξουμε γνώμη και θέλουμε να εκτελέσουμε τον παραπάνω αλγόριθμο σε περισσότερα του ενός μηχανήματα (κατανεμημένα) αρκεί η αλλαγή μίας και μόνο γραμμής, αυτής που ορίζει το *domain*, ως εξής:

```
const D = [1..10] dmapped Block(boundingBox=[1..10]);
```

Αν έχουμε 2 *locales*, οι δείκτες 1 ως 5 θα ανήκουν στο πρώτο και οι υπόλοιποι στο δεύτερο ενώ το ίδιο θα ισχύει για τα αντίστοιχα στοιχεία των A, B. Στον βρόχο *forall*, το κάθε *locale* θα εκτελέσει τις επαναλήψεις που αντιστοιχούν στους δικούς του δείκτες και θα τροποποιήσει μόνο τα δικά του στοιχεία, χρησιμοποιώντας τόσα νήματα όσα και οι επεξεργαστές του. Άρα θα έχουμε ταυτόχρονα παραλληλισμό σε δύο επίπεδα, μεταξύ-κόμβων και εντός-κόμβου. Μια δεύτερη αλλαγή μπορεί να είναι η αντικατάσταση της κατανομής Block με Cyclic η οποία γίνεται πάλι χωρίς να αγγίξουμε τον αλγόριθμο. Το ίδιο αν αλλάξουμε τους πίνακες σε αραιούς (βλ. κώδικα 5-11 στο κεφάλαιο 5.5). Παρατηρούμε λοιπον ότι η καρδιά του προγράμματος, ο αλγόριθμος, γράφεται μια φορά και δεν αλλάζει ποτέ με αποτέλεσμα να έχουμε μέγιστη φορητότητα αλγορίθμου. Η σημασία του τελευταίου *δεν μπορεί να υπερτονιστεί* – πρόκειται για πλήρη διαχωρισμο μεταξύ του αλγόριθμου και της υλοποίησης, τα γνωστά μας *policy* και *implementation* (μια αντιπαραβολή με την MPI θα σας πείσει).

Παραλληλισμός Εργασιών (tasks)

Ένας δεύτερος τρόπος για την επίτευξη παραλληλισμού είναι μέσω της δημιουργίας tasks. Αυτό γίνεται με τις εκφράσεις **begin**, **cobegin** και **coforall**. Η *begin* δημιουργεί ένα νέο ασύγχρονο task για την εκτέλεση της εντολής που ακολουθεί. Το νήμα που εκτελεί την *begin* δεν περιμένει το τέλος task αλλά συνεχίζει ευθύς αμέσως. Το *cobegin* είναι μία πιο δομημένη έκφραση για την δυναμική εκκίνηση πολλών tasks μαζί. Συνοδεύεται από ένα μπλοκ εντολών και ισοδυναμεί με την δημιουργία ενός νέου ασύγχρονου task για την εκτέλεση κάθε εντολής. Τέλος, το *coforall* είναι παρόμοιο σε λειτουργία με το *cobegin* αλλά αυτό δημιουργεί ένα καινούριο task για κάθε επανάληψη του βρόχου. Ακολουθούν παραδείγματα για κάθε έκφραση.

```
// το κύριο νήμα δεν περιμένει, η σειρά είναι undefined!
begin writeln("chicken");
writeln("egg");

// και εδώ το κύριο νήμα συνεχίζει παράλληλα με τα 2 tasks
cobegin {
  writeln("some task");
  writeln("some other task");
}

// αντιθέτως, η coforall έχει ένα implicit φράγμα στο τέλος
coforall i in 1..3 do
  writeln("hi from task ", i);
```

Συγχρονισμός και ατομικότητα

Στον τομέα του συγχρονισμού, λείπουν από την Chapel δομές τύπου *barrier* λόγω της δυσκολίας σωστής υλοποίησης σε ένα περιβάλλον με δυναμική δημιουργία νημάτων. Αντ'αυτού χρησιμοποιούνται δύο τύποι *μεταβλητών συγχρονισμού*, οι **sync** και **single**. Μια *sync* μεταβλητή μπορεί να είναι *άδεια* ή *γεμάτη*: κάθε ανάθεση τιμής την γεμίζει ενώ κάθε ανάγνωση την αδειάζει. Αν ένα νήμα προσπαθήσει να διαβάσει μια άδεια *sync* μεταβλητή, μπλοκάρεται μέχρι κάποιο άλλο να της αναθέσει κάποια τιμή. Και αντίστροφα, αν ένα νήμα προσπαθήσει να αναθέσει τιμή σε γεμάτη *sync* μεταβλητή, μπλοκάρεται μέχρι κάποιο άλλο να διαβάσει την προηγούμενη. Το αποτέλεσμα είναι ότι η τροποποίηση της τιμής της μεταβλητής γίνεται μόνο **ατομικά**. Η διαφορά της *single* με την *sync* μεταβλητή είναι ότι μπορεί να γραφτεί μόνο μια φορά και έπειτα παραμένει γεμάτη ακόμα και αν διαβάζεται από τα νήματα. Συνηθίζεται οι μεταβλητές αυτές να γράφονται με ένα $ στο τέλος για να ξεχωρίζουν. Ο τρόπος χρήσης τους φαίνεται στα παρακάτω παραδείγματα:

```
var sum$: sync int = 0;        // το sum$ ξεκινά "γεμάτο"

[i in 1..N] sum$ += i;         // τα νήματα κάνουν update ατομικά


var num1$: single int;         // το num1$ ξεκινά "άδειο"

begin on Locales(1) do         // λόγω begin το locale 0 δεν περιμένει
  num1$ = func1();             // πότε θα τελειώσει το task στο locale 1

var num2: int = func2();
writeln(num1$ + num2);         // Το locale 0 ενδέχεται να μπλοκάρει!
```

Που ζουν οι απλές μεταβλητές στην Chapel;

Απλές μεταβλητές ή πίνακες με μη κατανεμημένο domain ανήκουν πάντα στο locale στο οποίο εκτελείται το πρόγραμμα στο σημείο του ορισμού τους. Επομένως, οι μεταβλητές στο υψηλότερο επίπεδο ανήκουν στο locale 0, ενώ αυτές που ορίζονται σε κάποιο *on-block* ανήκουν στο αντίστοιχο locale και έχουν εμβέλεια μόνο ως το τέλος του μπλοκ. Τέλος, μεταβλητές εντός ενός *forall* ή *coforall* μπλοκ είναι *thread-private*, δηλαδή κάθε νήμα έχει την δικιά του μεταβλητή. Τα παραπάνω δείχνονται αναλυτικά στα επόμενα παραδείγματα.

```
var c: myClass;          // Η αναφορά c "ζει» στο locale 0...

on Locales(1) {
  c = new myClass();     // το αντικείμενο της στο locale 1!
  var b = 3;             // το b έχει εμβέλεια μόνο εντός του μπλοκ
}

// Αν θέλω το b να έχει εμβέλεια όπως το c πρέπει να το ορίσω ως εξής:
on Locales(1) x: int;    // Δυστυχώς αυτό ΔΕΝ έχει υλοποιηθεί ως τώρα.
```

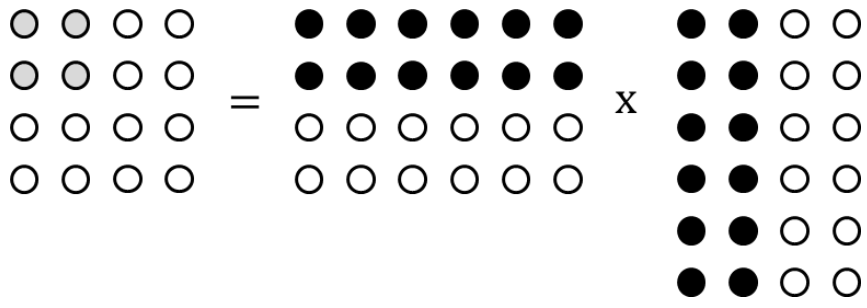
5 Περιγραφή των αλγορίθμων

Οι πέντε αλγόριθμοι που θα παρουσιάσουμε σ'αυτό το κεφάλαιο, από τον απλούστερο στον πιο περίπλοκο, έχουν επιλεγεί έτσι ώστε να καλύπτουν ένα ευρύ φάσμα όσον αφορά την συμπεριφορά τους ως προς τον τρόπο πρόσβασης στην μνήμη, τις ανάγκες για επικοινωνία μεταξύ των νημάτων και την ποιότητα της επικοινωνίας (*coarse vs fine-grain parallelism*). Επίσης, κάθε αλγόριθμος συνοδεύεται από τον κώδικα για την κατανεμημένη περίπτωση σε Chapel έτσι ώστε να καταδείξουμε τις κύριες διαφορές με την υλοποίηση σε MPI.

5.1 Γινόμενο πινάκων

Ο πρώτος αλγόριθμος που είναι ο πολλαπλασιασμός πίνακα με πίνακα ή αλλιώς το γινόμενο πινάκων είναι μια πολύ απλή πράξη η οποία ωστόσο παίζει σημαντικό ρόλο σε πολλές σπουδαίες εφαρμογές πχ στον χώρο της κβαντομηχανικής και αλλού. Ένα συγκεκριμένο παράδειγμα είναι η χρήση ενός πίνακα $m\ x\ n$ για την αναπαράσταση ενός γραμμικού μετασχηματισμού. Αυτός ο μετασχηματισμός πολλαπλασιαζόμενος με ένα διάνυσμα $x \in R^n$, το μεταφέρει ή αλλιώς το απεικονίζει από τον διανυσματικό χώρο R^n στον χώρο R^m. Στην περίπτωση αυτή, το γινόμενο δύο πινάκων A, B αντιστοιχεί στον συνδυασμό δύο γραμμικών μετασχηματισμών έτσι ώστε να ικανοποιείται η σχέση $(AB)x = A(Bx)$. (Linear map, 2012)

Έχουν προταθεί πολλοί αλγόριθμοι οι οποίοι βελτιώνουν την αλγοριθμική πολυπλοκότητα του γινομένου όπως ο αλγόριθμος Strassen ο οποίος την μειώνει από $O(n^3)$ σε $O(n^{2.807})$. Πάντως εμείς θα χρησιμοποιήσουμε τον απλό ή αλλιώς αφελή αλγόριθμο καθότι δεν ενδιαφερόμαστε τόσο για την απόδοση όσο για την σύγκριση μεταξύ των 3 γλωσσών. Επίσης, θα απλοποιήσουμε την κατανομή των στοιχείων έτσι ώστε κάθε νήμα να έχει όλα όσα χρειάζεται για τον υπολογισμό του δικού του χωρίου, κοινώς θα υπάρξει κάποια επικάλυψη στην κατανομή. Με αυτόν τον τρόπο, η επικοινωνία εξαλοίφεται και ο αλγόριθμος γίνεται *embarrassingly parallel*. Στο σχήμα 5-1 δείχνουμε τα στοιχεία που ανήκουν στο 1ο νήμα (τα γκρι υπολογίζονται βάσει των μαύρων).



Σχήμα 5-1: Πολλαπλασιασμός 2 πινάκων (νήμα 1)

Στην επόμενη σελίδα φαίνεται ο κώδικας Chapel για την κατανεμημένη περίπτωση (κώδικας 5-2). Για να διευκολύνουμε την κατανόηση, θα είμαστε κάπως πιο αναλυτικοί στην περιγραφή ορισμένων στοιχείων της γλώσσας, τα οποία όμως εμφανίζονται και στους επόμενους αλγορίθμους.

Η πρώτη παρατήρηση αφορά το στυλ προγραμματισμού που είναι το γνωστό SPMD. Ο λόγος που καταφεύγουμε σε αυτό είναι η δυσκολία έκφρασης της κατανομής που φαίνεται στο σχήμα 5-1 μέσω *global-view programming*, λόγω του ότι κάποια τμήματα του πίνακα κατανέμονται σε περισσότερα του ενός locales. Βλέπουμε δηλαδή ότι αν η κατανομή είναι κάπως πιο περίπλοκη από μπλοκ κτλ, δεν μπορούμε να βασιζόμαστε στις έτοιμες λύσεις της Chapel. Μια εναλλακτική ήταν η χρησιμοποίηση της κατανομής **Replicated** η οποία κατανέμει τον πίνακα σε όλα τα locales, αυτό όμως θα είχε δυσμενείς επιπτώσεις στο μέγεθος της μνήμης και κατά συνέπεια στην απόδοση της cache.

Στην αρχή του προγράμματος ερχόμαστε αντιμέτωποι με ένα σημαντικό περιορισμό στην εκφραστικότητα της Chapel και αυτός είναι η απουσία οποιουδήποτε μηχανισμού που να επιστρέφει τους δείκτες που ανήκουν σε κάθε locale για ένα κατανεμημένο domain. Ο μόνος τρόπος να πάρουμε αυτή την πληροφορία είναι έμμεσα μέσω του ίδιου του iteration πάνω στο domain. Επομένως, η λύση που διαλέξαμε ήταν ένα προκαταρκτικό iteration στο CDomain το οποίο εκτελείται από όλα τα locales με όλα τα νήματά τους, εξ ου και η χρήση μεταβλητών συγχρονισμού για την σωστή ενημέρωση των κοινών μεταβλητών mlow κτλ. Να σημειώσουμε ότι η προσθήκη ενός τρόπου για την απόκτηση αυτής της πληροφορίας είναι στα άμεσα σχέδια των δημιουργών.

Η ιδιωματική έκφραση `coforall loc in Locales do on loc` την οποία θα συναντήσουμε και στα επόμενα, είναι ο τρόπος με τον οποίο ξεκινάει το SPMD τμήμα κάθε προγράμματος Chapel. Αναλύεται ως εξής: το locale 0 στο οποίο τρέχουμε αρχικά, δημιουργεί ένα παράλληλο νήμα για κάθε *locale* μέσα στον πίνακα *Locales*. Τα νήματα αυτά με τη σειρά τους μεταφέρουν την εκτέλεση στα αντίστοιχα *locales* μέσω της έκφρασης **on**. Οπότε το σώμα του κυρίως βρόχου εκτελείται από όλα τα *locales* ακριβώς όπως και στην περίπτωση της MPI.

Η τελευταία παρατήρηση αφορά τον τρόπο επιλογής των τυχαίων τιμών για ανάθεση στους πίνακες A και B. Η χρησιμοποίηση τιμών στο διάστημα [1.0, 2.0] στοχεύει στην αποφυγή εμφάνισης αριθμών κινητής υποδιαστολής σε denormalized μορφή (βλέπε IEEE 754). Ο λόγος είναι ότι σε όλους τους σύγχρονους επεξεργαστές, αυτοί οι αριθμοί οδηγούν έως και σε 100 φορές αργότερη λειτουργία σε κάποιες ακραίες περιπτώσεις. (Fog, 2012)

```
// C = AB, with A = MxQ and B = QxN (obv. C = MxN)
use BlockDist, Random;

config const M = 10, Q = 10, N = 10;

// C is distributed to the locales
const CSpace  = [1..M, 1..N],
      CDomain = CSpace dmapped Block(boundingBox=CSpace);
var   C: [CDomain] real;  // auto-initialized to 0.0

// No standard way yet to get a locale's indexes in a distributed domain!
// Hack: I must iterate to find each mlow, nlow for each locale
const LocaleDomain = LocaleSpace dmapped Block(boundingBox=LocaleSpace);
var   mlow, nlow, mhigh, nhigh: [LocaleDomain] sync int;

mlow = M; mhigh = 0;
nlow = N; nhigh = 0;

// Each locale iterates on its indexes (sync vars ensure atomicity)
forall (m,n) in CDomain {
  const myid = here.id;

  mlow[myid]  = min(mlow[myid],  m);
  nlow[myid]  = min(nlow[myid],  n);
  mhigh[myid] = max(mhigh[myid], m);
  nhigh[myid] = max(nhigh[myid], n);
}

coforall loc in Locales do on loc {

  // readXX() does not "empty" the sync vars
  const myid = here.id,
        myRows = mlow[myid].readXX()..mhigh[myid].readXX(),
        myCols = nlow[myid].readXX()..nhigh[myid].readXX();

  // From A and B, I use only the rows and columns I need
  const myDomA = [myRows, 1..Q],
        myDomB = [1..Q, myCols];
  var   A: [myDomA] real,
        B: [myDomB] real;

  // Fill A, B with random values in [1.0, 2.0]
  const rs = new RandomStream();
  [a in A] a = rs.getNext() + 1.0;
  [b in B] b = rs.getNext() + 1.0;

  // Main loop
  forall (i,j) in [myRows, myCols] do
    for k in 1..Q do
      C(i,j) += A(i,k) * B(k,j);
}

/* testing/timing code ommitted */
```

Κώδικας 5-2: Γινόμενο πινάκων σε Chapel για πολλά locales

5.2 Αλγόριθμος Floyd-Warshall

Ο αλγόριθμος Floyd-Warshall είναι ένας αλγόριθμος που παίρνει ως είσοδο ένα γράφο χωρίς αρνητικούς κύκλους και βγάζει ως έξοδο την ελάχιστη απόσταση μεταξύ όλων των κορυφών του τελευταίου. Επίσης με μια απλή τροποποίηση μπορεί να βρει και ποια είναι αυτά τα μονοπάτια. Η βασική ιδέα έχει ως εξής.

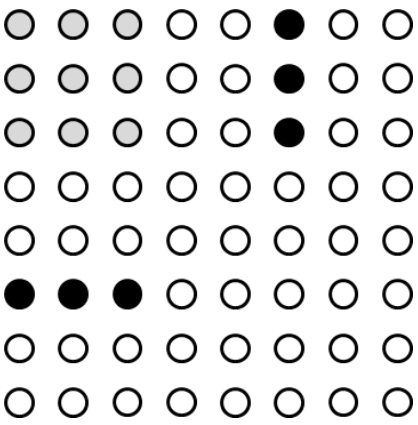
Υπάρχει ένας γράφος G με σύνολο κορυφών V και σύνολο ακμών E. Σε κάθε ακμή αντιστοιχεί ένα βάρος – όσο πιο μεγάλο, τόσο πιο μακρινή η απόσταση μεταξύ των δύο κορυφών. Βάσει των παραπάνω, φτιάχνουμε έναν αρχικό αρχικό πίνακα $D^{(0)}$ ο οποίος περιέχει όλες τις αποστάσεις μεταξύ των κόμβων. Οι τιμές των στοιχείων του δίνονται από την εξίσωση 5.1.

$$d_{i,j}^{(0)} = \begin{cases} weight_{i,j} & \text{εάν οι κόμβοι συνδέονται} \\ inf & \text{εάν δεν συνδέονται} \\ 0 & \text{εάν } i = j \end{cases} \quad [5.1]$$

Αν $d_{i,j}^{(k)}$ είναι η μικρότερη απόσταση απ'τον κόμβο i στον j, αν χρησιμοποιήσουμε μόνο τους κόμβους που περιέχονται στο σύνολο $\{1, 2, \dots, k\}$, τότε η επαναληπτική διαδικασία ορίζεται απ'την αναδρομική εξίσωση 5.2:

$$d_{i,j}^{(k)} = \begin{cases} weight_{i,j} & \text{εάν } k = 0. \\ min\left(d_{i,j}^{(k-1)},\ d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right) & \text{εάν } k \geq 1. \end{cases} \quad [5.2]$$

Έπειτα από n βήματα, όπου n ο αριθμός του συνόλου των κόμβων, η λύση βρίσκεται στον $D^{(n)}$. Στο σχήμα 5-3, επισημαίνουμε με γκρι τα στοιχεία που ανήκουν στο 1ο νήμα και με μαύρο τα στοιχεία που αυτό χρειάζεται κατά την επανάληψη $k = 6$.



Σχήμα 5-3: Το νήμα 1 κατά την επανάληψη k = 6

Ο κώδικας 5-4 είναι ένα καλό παράδειγμα για την εκφραστική δύναμη της Chapel όταν είναι εφικτή η διατύπωση ενός αλγορίθμου σε *global view*. Ο κύριος βρόχος του προγράμματος εκτελείται σειριακά όσον αφορά το k στο locale 0 και γίνεται παράλληλος όταν συναντά το forall, οπότε και κάθε locale χρησιμοποιεί όλα του τα νήματα για να ενημερώσει τα στοιχεία του πίνακα που του ανήκουν. Να επισημάνουμε ότι δεν υπάρχει κίνδυνος συγχρονισμού γιατί στην επανάληψη k, το στοιχείο στη θέση (i, j) συγκρίνεται με τα (i, k)+(k, j). Και τα δύο στοιχεία στα οποία βασίζεται μένουν αναλλοίωτα κατά την διάρκεια της επανάληψης πχ το (k, j) συγκρίνεται με το (k,k) + (k,j) = (k,j).

```
use BlockDist, Random;

config const N = 10;
const Space  = [1..N, 1..N],
      Domain = Space dmapped Block(boundingBox=Space);
var path: [Domain] real;

// Fill path with random values in [1.0, 2.0]
const rs = new RandomStream();
[path_elmt in path] path_elmt = rs.getNext() + 1.0;

// Main loop
for k in 1..N do
  forall (i,j) in Domain do
    path(i,j) = min(path(i,j), path(i,k) + path(k,j));

/* testing/timing code ommitted */
```

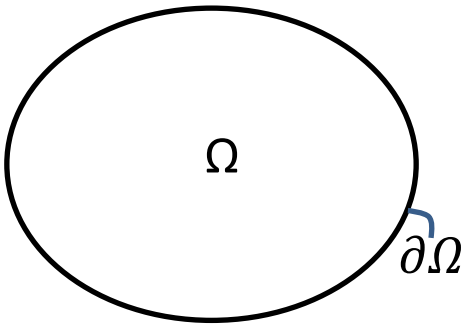
Κώδικας 5-4: Floyd Warshall σε Chapel για πολλά locales

5.3 Επαναληπτικός αλγόριθμος Jacobi

Ο αλγόριθμος Jacobi είναι ένας επαναληπτικός αλγόριθμος που αποδίδεται στον Carl Jacobi και βασίζεται στην μέθοδο της χαλάρωσης (*relaxation iterative solver*). Λόγω της μεγάλης καθυστέρησης που εμφανίζουν γενικώς οι επαναληπτικοί αλγόριθμοι στην επίτευξη ικανής ακρίβειας, αυτοί περιορίζονται συνήθως σε συστήματα εξισώσεων όπου η πλειοψηφία των στοιχείων του πίνακα είναι μηδενικά. Οι πρακτικές εφαρμογές είναι διάφορες όπως ανάλυση κυκλωμάτων, προβλήματα οριακής τιμής και μερικές διαφορικές εξισώσεις. Το μεγάλο πλεονέκτημα του αλγορίθμου είναι η εύκολη παραλληλοποίηση.

Ο αλγόριθμος στην μορφή που θα τον χρησιμοποιήσουμε εδώ προκύπτει έπειτα από εφαρμογή της διακριτοποίησης σε προβλήματα τύπου αρχικής/οριακής τιμής (Karniadakis & Kirby II, 2003). Αρχικά θεωρούμε την εξίσωση διάχυσης 5.3 για τον τρισδιάστατο χώρο που απεικονίζεται στο σχήμα 5-5.

$$\nabla^2\Theta + q = 0 \ στον \ \Omega, με \ \Theta(x,y,z) = 0 \ επάνω \ στην \ \partial\Omega \qquad [5.3]$$



Σχήμα 5-5: Χώρος Ω με όριο την επιφάνεια ∂Ω

Αν στην εξίσωση 5.3 προσθέσουμε μια ψευδοπαράγωγο ως προς τον χρόνο, παίρνουμε την εξίσωση 5.4.

$$\frac{\partial\Theta}{\partial t} = \nabla^2\Theta + q \ στον \ \Omega, με \ \Theta(x,y,z) = 0 \ επάνω \ στην \ \partial\Omega \qquad [5.4]$$

Τέλος, εφαρμόζοντας την εμπρόσθια μέθοδο διακριτοποίησης Euler (τα βήματα της οποίας είναι εκτός των πλαισίων της παρούσας διπλωματικής), καταλήγουμε στον γνωστό αλγόριθμο Jacobi που είναι ο πυρήνας που θα χρησιμοποιήσουμε:

$$\Theta_{i,j,k}^{n+1} = \frac{1}{6}(\Theta_{i+1,j,k}^{n} + \Theta_{i,j+1,k}^{n} + \Theta_{i,j,k+1}^{n} + \Theta_{i-1,j,k}^{n} + \Theta_{i,j-1,k}^{n} + \Theta_{i,j,k-1}^{n})$$

Ο κώδικας 5-6 δεν χρειάζεται κάποιο ιδιαίτερο σχόλιο πέραν του ότι χρησιμοποιούμε μια μεταβλητή myConverged για κάθε locale η οποία ενημερώνεται από όλα τα νήματα μέσα στο *forall*. Καθότι η ανάθεση γίνεται μόνο όταν πρέπει να αλλάξει η τιμή σε false, δεν υπάρχει κίνδυνος συγχρονισμού και γι'αυτό η μεταβλητή είναι απλή και όχι sync.

```
use BlockDist;

config const X = 10, Y = 10, Z = 10, boundaryValue = 10.0,
             initialValue = 0.0, maxError = 1e-5, maxLoops = 0;
const ONE_SIXTH = 1.0/6.0;

// innerDom slices Domain and inherits its domain map
const Space    = [1..X, 1..Y, 1..Z],
      Domain   = Space dmapped Block(boundingBox=Space),
      innerDom = Domain[2..X-1, 2..Y-1, 2..Z-1];
var A, B: [Domain] real;

/* Initialization of A, B ommitted */

var converged: bool, numLoops = 0;

// Each locale gets its own local myConverged
const LocaleDomain = LocaleSpace dmapped Block(boundingBox=LocaleSpace);
var   myConverged: [LocaleDomain] bool;

// these tuples used for clarity
const up    = (-1,0,0), down  = (1,0,0),  left = (0,-1,0),
      right = (0,1,0),  front = (0,0,-1), back = (0,0,1);

// Auxiliary procedure used in main loop
inline proc compute(x, y) {
  numLoops += 1;

  coforall loc in Locales do on loc do
    myConverged[here.id] = true;

  forall idx in innerDom {
    y(idx) = ONE_SIXTH * (x(idx+up)    + x(idx+down)  + x(idx+left) +
                          x(idx+right) + x(idx+front) + x(idx+back));
    if myConverged[here.id] && abs(y(idx) - x(idx)) > maxError then
      myConverged[here.id] = false;
  }

  converged = && reduce myConverged;
}

// Main loop unrolled by 2.
do {
  compute(A, B);
  if converged || numLoops == maxLoops then
    break;
  compute(B, A);
} while !converged && numLoops != maxLoops;

/* testing/timing code ommitted */
```

Κώδικας 5-6: Jacobi σε Chapel για πολλά locales

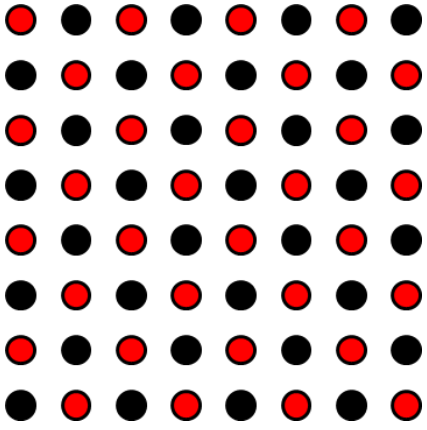
5.4 Παράλληλος (Black-Red) Gauss-Seidel

Ο (σειριακός) αλγόριθμος Gauss-Seidel είναι το προφανές επόμενο βήμα στην αναζήτηση ενός αλγορίθμου χαλάρωσης με ταχύτερη σύγκλιση. Η βασική ιδέα είναι ότι μπορούμε να χρησιμοποιούμε τιμές στοιχείων που υπολογίστηκαν εντός της τρέχουσας επανάληψης για να επιταχύνουμε την όλη διαδικασία (ο Jacobi «επιβάλλει» την χρησιμοποίηση τιμών μόνο απ'το προηγούμενο βήμα). Η εξίσωση που δίνει την νέα τιμή για κάθε στοιχείο καθώς διατρέχουμε τον πίνακα με *row-major* φορά είναι η εξής:

$$\Theta_{i,j,k}^{n+1} = \frac{1}{6}(\Theta_{i+1,j,k}^{n} + \Theta_{i,j+1,k}^{n} + \Theta_{i,j,k+1}^{n} + \Theta_{i-1,j,k}^{n+1} + \Theta_{i,j-1,k}^{n+1} + \Theta_{i,j,k-1}^{n+1})$$

Τα πλεονεκτήματα του Gauss-Seidel είναι πολλαπλά. Καταρχήν, τώρα οι νέες τιμές υπολογίζονται επιτόπου και δεν απαιτείται η εναλλάξ χρήση δύο πινάκων οπότε έχουμε μεγάλη εξοικονόμηση μνήμης. Μάλιστα είναι πιθανή η επιτάχυνση του προγράμματος στην περίπτωση που ο πίνακας χωράει εξ ολοκλήρου στην cache. Το δεύτερο σημαντικό πλεονέκτημα του Gauss-Seidel είναι η μεγάλη ταχύτητα σύγκλισης η οποία, όπως φαίνεται και διαισθητικά απ'την ανωτέρω εξίσωση, είναι περίπου διπλάσια αυτής του Jacobi.

Βέβαια, ο Gauss Seidel σε αυτή την μορφή μας είναι άχρηστος γιατί δεν μπορεί να παραλληλοποιηθεί έτσι ώστε τα αποτελέσματά του να είναι ντετερμινιστικά. Απαιτείται ένα επιπλέον βήμα που είναι ο χρωματισμός των στοιχείων του πίνακα με περισσότερα του ενός χρώματα και ο υπολογισμός κάθε χρωματικής ομάδας σε κάθε επανάληψη ξεχωριστά. Η απλούστερη περίπτωση με τα δύο χρώματα (Black-Red) είναι αυτή που θα εξετάσουμε και στην παρούσα εργασία. Το σχήμα 5-7 δείχνει τον χρωματισμό για την την δισδιάστατη περίπτωση.



Σχήμα 5-7: Χρωματισμός για το 2D Black-Red

Οι εξισώσεις που δίνουν τα 2 βήματα της κάθε επανάληψης φαίνονται ακριβώς από κάτω. Όπως βλέπουμε, το 2ο βήμα χρησιμοποιεί τα κόκκινα στοιχεία τα οποία έχουν υπολογιστεί στο προηγούμενο βήμα.

$$R_{i,j,k}^{n+1} = \frac{1}{6}(B_{i+1,j,k}^{n} + B_{i,j+1,k}^{n} + B_{i,j,k+1}^{n} + B_{i-1,j,k}^{n} + B_{i,j-1,k}^{n} + B_{i,j,k-1}^{n})$$

$$B_{i,j,k}^{n+1} = \frac{1}{6}(R_{i+1,j,k}^{n+1} + R_{i,j+1,k}^{n+1} + R_{i,j,k+1}^{n+1} + R_{i-1,j,k}^{n+1} + R_{i,j-1,k}^{n+1} + R_{i,j,k-1}^{n+1})$$

Ο αλγόριθμος Black-Red «κληρονομεί» την ταχύτητα σύγκλισης του απλού Gauss-Seidel που είναι διπλάσια του Jacobi, απαιτεί όμως δύο βήματα σε κάθε επανάληψη. Αυτό μπορεί να έχει επιπτώσεις αν ο πίνακας δεν χωράει στην κρυφή μνήμη γιατί μπορεί να οδηγήσει σε διπλάσια cache misses απ'ότι στον αλγόριθμο Jacobi. Στις δικές μας μετρήσεις αυτό δεν παίζει ρόλο διότι το μέγεθος του πίνακα είναι μικρό σε σχέση με την cache.

Ο κώδικας 5-8 στην επόμενη σελίδα δεν διαφέρει πολύ απ'αυτόν του αλγορίθμου Jacobi αν εξαιρέσουμε τον υπολογισμό των στοιχείων σε δύο βήματα. Ένα αναγκαίο κακό είναι η χρήση του if εντός του βρόχου για τον έλεγχο του χρώματος των στοιχείων. Δοκιμάστηκαν πολλές εναλλακτικές λύσεις όπως το σπάσιμο σε δύο πίνακες Red και Black ή διαφορετικοί τρόποι iteration, όλες όμως ήταν σημαντικά πιο περίπλοκες και με παρόμοια απόδοση.

Στο κεφάλαιο 7 δείχνουμε και μια άλλη εκδοχή του αλγορίθμου Black-Red για την περίπτωση που το domain για τις δύο χρωματικές ομάδες σχεδιάζεται απ'τον ίδιο τον προγραμματιστή. Είναι ενδιαφέρουσα η σύγκριση των επιδόσεων μεταξύ των δύο εκδοχών.

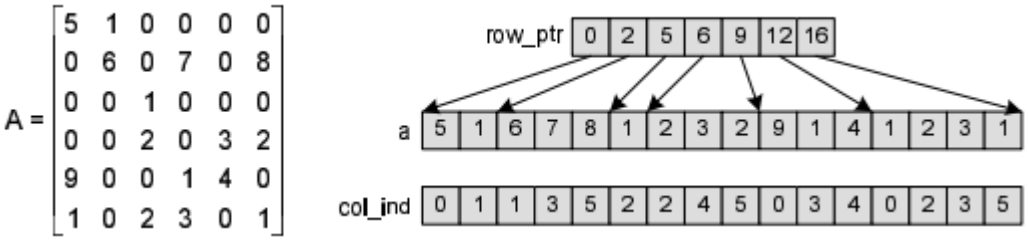
```
// Red set starts with first element of innerDom (2,2,2) = even
use BlockDist;

config const X = 10, Y = 10, Z = 10, boundaryValue = 10.0,
             initialValue = 0.0, maxError = 1e-5, maxLoops = 0;
const ONE_SIXTH = 1.0/6.0;
enum color {black = 0, red};

// innerDom slices Domain and inherits its domain map
const Space    = [1..X, 1..Y, 1..Z],
      Domain   = Space dmapped Block(boundingBox=Space),
      innerDom = Domain[2..X-1, 2..Y-1, 2..Z-1];
var A: [Domain] real;

/*** Initialization of A ommitted ***/

var converged: bool, numLoops = 0;

// Each locale gets its own local myConverged
const LocaleDomain = LocaleSpace dmapped Block(boundingBox=LocaleSpace);
var   myConverged: [LocaleDomain] bool;

// Auxiliary procedure used in main loop
inline proc compute(c: color) {
  forall (i,j,k) in innerDom {
    var oldValue: real;  // thread-private

    if (i+j+k + c) % 2 {
      oldValue = A(i,j,k);
      A(i,j,k) = ONE_SIXTH * (A(i-1,j,k) + A(i+1,j,k) + A(i,j-1,k) +
                              A(i,j+1,k) + A(i,j,k-1) + A(i,j,k+1));
      if myConverged[here.id] && abs(A(i,j,k) - oldValue) > maxError then
        myConverged[here.id] = false;
    }
  }
}

// Main loop
do {
  numLoops += 1;

  coforall loc in Locales do on loc do
    myConverged[here.id] = true;

  compute(color.red);
  compute(color.black);

  converged = && reduce myConverged;
} while !converged && numLoops != maxLoops;

/* testing/timing code ommitted */
```

Κώδικας 5-8: Black-Red σε Chapel για πολλά locales

5.5 Γινόμενο αραιού πίνακα με διάνυσμα

Αραιός πίνακας ονομάζεται ο πίνακας του οποίου η συντριπτική πλειοψηφία των στοιχείων είναι μηδενικά. Τέτοιοι πίνακες συνήθως εμφανίζονται όταν εξετάζουμε φυσικά συστήματα των οποίων τα μέρη είναι χαλαρά συζευγμένα. Λόγω του ότι η αποθήκευση αυτών των πινάκων με παραδοσιακό τρόπο καταλαμβάνει περισσότερη μνήμη απ'όσο χρειάζεται και άρα οδηγεί σε μη αναγκαία cache misses, έχουν προταθεί προταθεί πολλές εναλλακτικές δομές οι οποίες εκμεταλλεύονται την πλεονάζουσα πληροφορία. Τέτοιες είναι οι δομές *COO (coordinate list)*, *Dictionary of Keys (DOK)* και άλλες και κάθεμια προσανατολίζεται είτε σε ταχύτερες αριθμητικές πράξεις είτε σε ταχύτερη τροποποίηση του πίνακα. (Sparse matrix, 2012)

Εμείς εδώ θα χρησιμοποιήσουμε την κλασσική δομή Compressed Sparse Row (CSR) η οποία φαίνεται στο σχήμα 5-9. (Goumas, Kourtis, Anastopoulos, Karakasis, & Koziris, 2008) Στον πίνακα a αποθηκεύονται οι τιμές των μη-μηδενικών στοιχείων του A με σειρά row-major, ενώ ο πίνακας col_ind περιέχει την στήλη για το κάθε στοιχείο. Τέλος ο πίνακας row_ptr δείχνει που ξεκινάει και που σταματά η κάθε σειρά στον πίνακα a.



Σχήμα 5-9: Δομή αποθήκευσης CSR

Ο αλγόριθμος αυτός έχει πολλά ενδιαφέροντα σημεία σε σχέση με τους προηγούμενους. Καταρχήν, είναι *memory-bound* και ασκεί μεγάλη πίεση στο υποσύστημα της μνήμης λόγω της μικρής αναλογίας του αριθμού πράξεων προς τον αριθμό αναφορών στην μνήμη – ο πολλαπλασιασμός πίνακα επί διάνυσμα εκτελεί $O(n^2)$ πράξεις σε $O(n^2)$ δεδομένα. Αντίθετα, το γινόμενο πινάκων εκτελεί $O(n^3)$ πράξεις σε $O(n^2)$ δεδομένα και άρα απαιτεί μικρότερο *memory bandwidth*. Επιπλέον, υπάρχουν κι άλλα ζητήματα όπως η σχεδόν τυχαία πρόσβαση στο διάνυσμα x ή το μεγάλο overhead των βρόχων με τα οποία όμως δεν θα ασχοληθούμε στην παρούσα εργασία.

Ο κώδικας για τον αλγόριθμο που ακολουθεί έχει γραφτεί σε στυλ SPMD εξαιτίας περιορισμών που οφείλονται στο πρώιμο στάδιο της γλώσσας. Συγκεκριμένα, ενώ υπάρχει υποστήριξη για αραιούς πίνακες μέσω του βασικού τύπου **sparse subdomain**, δεν έχει ακόμα υλοποιηθεί το αντίστοιχο *domain map* που θα τους κατανέμει στα διάφορα *locales*. Αυτό σημαίνει ότι επιβαρυνόμαστε με τον προγραμματισμό της δομής CSR και ακόμα συνδέουμε στενά τον αλγόριθμο με την συγκεκριμένη υλοποίηση. Για να φανεί η θεμελιώδης διαφορά στην ευκολία κατανόησης και στην ανεξαρτησία αλγορίθμου-υλοποίησης, παραθέτουμε και τον κώδικα όπως θα ήταν εάν η γλώσσα ήταν έτοιμη.

```
// y = Ax, with sparse A = MxN and x = Nx1 (obv. Y = Mx1)

use BlockDist, Random;

config const sparsefile = "", numLoops = 1;

if sparsefile == "" then
  halt("You must provide a CSR file via --sparsefile");

// All clones have the same disk image and
// can read the "same" file simultaneously.
coforall loc in Locales do on loc {
  var f = new file(sparsefile, FileAccessMode.read);
  f.open();

  var M, N, nnz: int;
  f.read(M); f.read(N); f.read(nnz);

  // Split rows between locales
  const myid = here.id,
        M_chunk = M / numLocales, M_rest  = M % numLocales,
        myM = M_chunk + if myid < M_rest then 1 else 0,
        myFirstRow = 1 + myid*M_chunk + if myid < M_rest
                                          then myid else M_rest;

  var y: [1..myM] real,  // auto-initialized to 0.0
      row_ptr: [1..myM+1] int;

  // Estimate size of A and col_ind, will expand D if necessary
  var numEntries = nnz/numLocales + 1,  // at least 1!
      D = [1..numEntries],
      A: [D] real, col_ind: [D] int;

  // Read sparse matrix from file
  var i = 1, my_nnz = 0, previousRow = 0,
      currentRow, column: int, value: real;

  while (nnz) {
    f.read(currentRow); f.read(column); f.read(value);
    currentRow -= myFirstRow - 1;  // my rows will be 1..myM
    if (currentRow > 0 && currentRow <= myM) {
      my_nnz += 1;
      col_ind(my_nnz) = column; A(my_nnz) = value;
      if (currentRow > previousRow) {
        while (i <= currentRow) {
          row_ptr(i) = my_nnz;
          i += 1;
        }
        previousRow = currentRow;
      }
    }


                    /* Continued in next page */
```

```
    if my_nnz == numEntries then {
      D = D.expand(512).translate(512);  // adds 1024 entries
      numEntries += 1024;
    }

    nnz -= 1;
  }

  // Fill empty entries at the end of row_ptr with my_nnz+1
  while i <= myM+1 {
    row_ptr(i) = my_nnz+1;
    i += 1;
  }

  // Shorten D down to exact size
  D = [1..my_nnz];

  f.close();

  // Fill x with random values in [1.0, 2.0]
  var x: [1..N] real;
  const rs = new RandomStream();
  [x_elmt in x] x_elmt = rs.getNext() + 1.0;

  // Main loop
  for 1..numLoops {
    forall i in 1..myM {
      y(i) = 0.0;
      for j in row_ptr(i)..row_ptr(i+1)-1 do
        y(i) += A(j) * x(col_ind(j));
    }
  }
}

/* testing/timing code ommitted */
```

Κώδικας 5-10: Γινόμενο αραιού πίνακα με διάνυσμα σε Chapel για πολλά locales

Ο κώδικας 5-10 δεν χρειάζεται ιδιαίτερο σχολιασμό: είναι ακριβώς η υλοποίηση που ακολουθήσαμε και στην MPI με την διαφορά ότι είναι ελαφρώς πιο μαζεμένος χάρη στην μεγαλύτερη εκφραστική ικανότητα της Chapel. Το πρόγραμμα είναι γραμμένο σε *fragmented view* και ο κώδικας είναι μακροσκελής ενώ η καθαρότητα του αλγορίθμου χάνεται μέσα στις λεπτομέρειες της υλοποίησης. Η κεντρική ιδέα είναι ότι ο πίνακας κατανέμεται στα locales ανά μπλοκ σειρών (μια κατανομή βάσει του αριθμού των στοιχείων θα ήταν σαφώς προτιμητέα αν και θα περιέπλεκε σημαντικά τα πράγματα).

Ο κώδικας 5-11 που ακολουθεί στην επόμενη σελίδα είναι η υλοποίηση που θα κάναμε στην περίπτωση που η Chapel ήταν έτοιμη. Βλέπουμε ότι ο κώδικας είναι πολύ πιο σαφής και περιεκτικός με σχεδόν τις μισές γραμμές. Όμως το σπουδαιότερο είναι ότι η καρδιά του αλγορίθμου είναι ανεξάρτητη του τρόπου με τον οποίο ορίζεται ο πίνακας, δηλαδή αν αυτός ήταν πυκνός ο κύριος βρόχος θα ήταν ο ίδιος!

```
// y = Ax, with sparse A = MxN and x = Nx1 (obv. Y = Mx1)

use BlockDist, Random;

config const sparsefile = "", numLoops = 1;

if sparsefile == "" then
  halt("You must provide a CSR file via --sparsefile");

var f = new file(sparsefile, FileAccessMode.read);
f.open();

var M, N, nnz: int;
f.read(M); f.read(N); f.read(nnz);

// Define a distributed sparse array
const D = [1..M, 1..N] dmapped Block(boundingBox=[1..M,1..N]);
var   S: sparse subdomain(D),
      A: [S] real;      // initially no elements

var i, j: int;
for 1..nnz {
  i = f.read(int);
  i = f.read(int);
  S += (i,j);           // add index to sparse domain
  f.read(A(i,j));
}

f.close();

var y: [1..M] real, x: [1..N] real;

// Fill x with random values in [1.0, 2.0]
const rs = new RandomStream();
[x_elmt in x] x_elmt = rs.getNext() + 1.0;

// Main loop
for 1..numLoops {
  y = 0.0;
  forall (i,j) in S do
    y(i) += A(i,j) * x(j);
}

/*** testing/timing code ommitted ***/
```

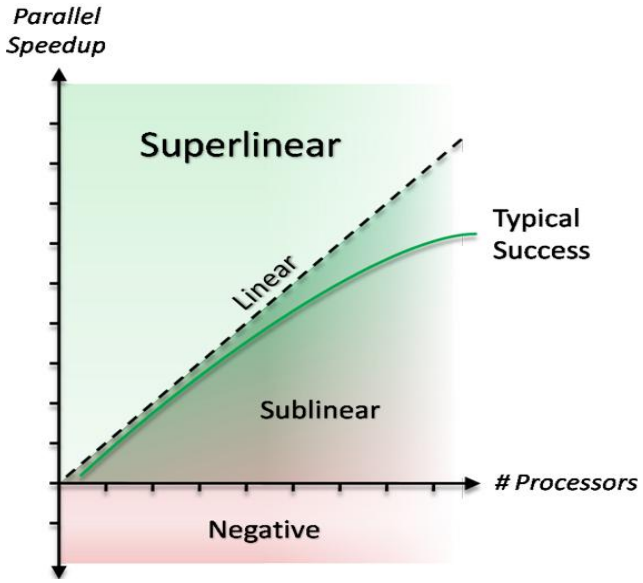
Κώδικας 5-11: Εναλλακτικός κώδικας για γινόμενο αραιού πίνακα με διάνυσμα

6 Πειραματική αξιολόγηση

Η αξιολόγηση που θα επιχειρήσουμε έχει δύο σκέλη. Το πρώτο είναι η μέτρηση της απόδοσης της Chapel σε ένα κόμβο έναντι της OpenMP και σε πολλούς έναντι της MPI. Με τον όρο απόδοση εννοούμε τόσο τον απόλυτο χρόνο όσο και την κλιμακωσιμότητα. Για την δεύτερη, θα μετρήσουμε και θα σχεδιάσουμε την λεγόμενη επιτάχυνση ή αλλιώς *speedup* η οποία δίνεται από την εξίσωση 6.1.

$$S_p = \frac{T_1}{T_p} \qquad [6.1]$$

όπου T_1 ο χρόνος εκτέλεσης για έναν πυρήνα και T_p ο χρόνος εκτέλεσης για p πυρήνες. Η ιδανική περίπτωση ($S_p = p$) απαντάται όταν ο διπλασιασμός των πυρήνων οδηγεί σε υπο-διπλασιασμό του χρόνου εκτέλεσης. Σε ορισμένες περιπτώσεις, κυρίως για μικρότερα χωρία, μπορεί να εμφανιστεί και υπεργραμμική επιτάχυνση η οποία βεβαίως οφείλεται σε παρενέργειες της ιεραρχίας μνήμης: με την χρήση επιπλέον πυρήνων αυξάνεται και η συνολική διαθέσιμη cache του συστήματος με αποτέλεσμα ένα μεγαλύτερο μέρος ή και ολόκληρο το σύνολο εργασίας *(working set)* του προβλήματος να χωράει σε αυτή και άρα να έχουμε δραματική βελτίωση της ταχύτητας. Η εικόνα 6-1 (Sutter, 2012) δείχνει αναλυτικά τις διάφορες περιπτώσεις για την επιτάχυνση.



Εικόνα 6-1: Πιθανές τιμές της επιτάχυνσης

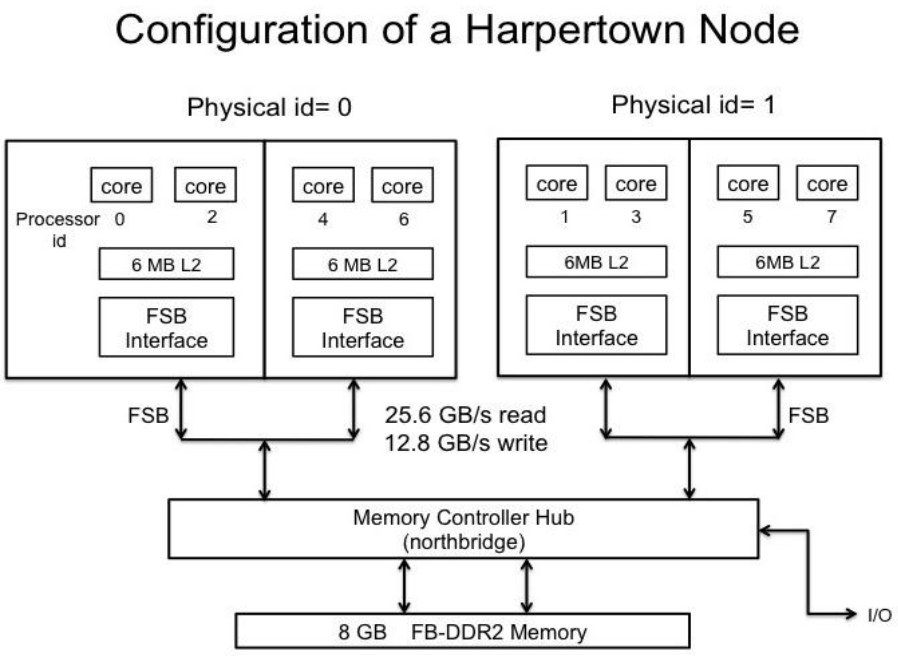
Το δεύτερο σκέλος της σύγκρισης είναι φυσικά η παραγωγικότητα. Δεδομένου ότι δεν κρατήσαμε τους χρόνους ανάπτυξης, η μόνη εύλογη μονάδα μέτρησης είναι τα SLOCs (*source lines of code*) αλλά από την ανάποδη. Δηλαδή λιγότερα SLOCs σημαίνει καλύτερη προγραμματισιμότητα, για ένα σωρό λόγους όπως μικρότερο χρόνο αποσφαλμάτωσης, καλύτερη κατανόηση του κώδικα και ούτω καθεξής.

6.1 Οι πλατφόρμες δοκιμών (test beds)

Στη διάθεσή μας για τα πειράματα είχαμε δύο συστοιχίες που χρησιμοποιούν Gigabit Ethernet ως δίκτυο διασύνδεσης. Η πρώτη είναι ομοιογενής και αποτελείται από 8-πύρηνους *Intel Xeon E5405 @ 2.00GHz* (αρχικτεκτονική *Harpertown*), ενώ η δεύτερη είναι ανομοιογενής και περιέχει 8-πύρηνους νεότερης γενιάς όπως οι *Intel Xeon X5560 @ 2.80GHz* και *Intel Xeon W5580 @ 3.20GHz* (αρχιτεκτονικής *Nehalem*). Οι διαφορές μεταξύ των δύο αρχιτεκτονικών είναι σημαντικές και αυτό φαίνεται και στα αποτελέσματα των μετρήσεων. Στο κεφάλαιο αυτό θα περιγράψουμε συνοπτικά τα κύρια χαρακτηριστικά.

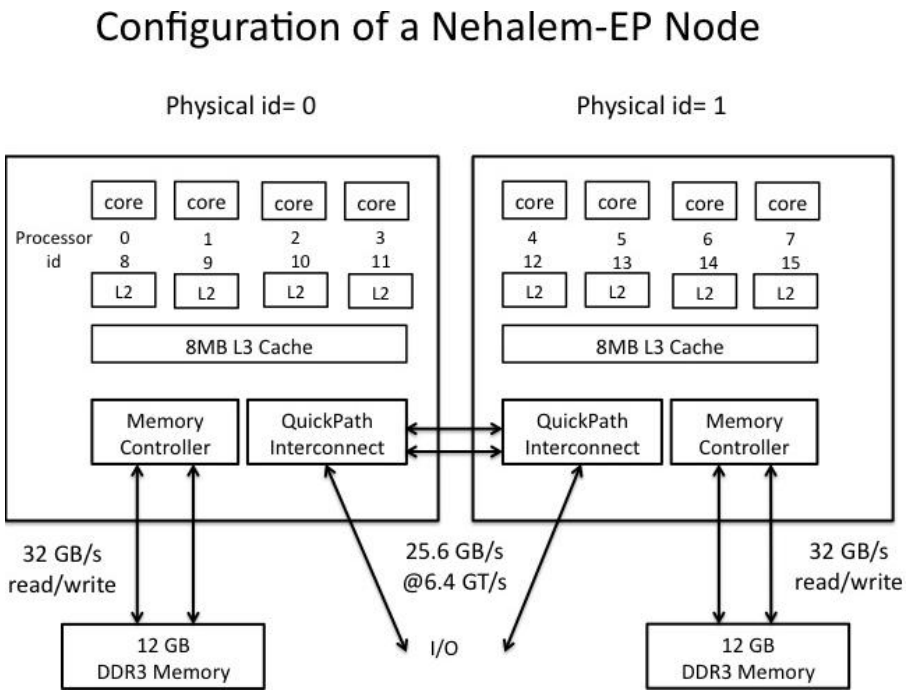
Η αρχιτεκτονική *Harpertown* είναι η παραδοσιακή SMP αρχιτεκτονική της Intel, στην οποία όλοι οι πυρήνες προσπελαύνουν την κοινή μνήμη μέσω ενός κοινού διαδρόμου με ένα και μοναδικό ελεγκτή μνήμης. Όπως προαναφέρθηκε στο κεφάλαιο 2.4.1, αυτό έχει ως συνέπεια την απουσία κλιμακωσιμότητας γιατί η προσθήκη νέων επεξεργαστών οδηγεί σε αύξηση του απαιτούμενου εύρους ζώνης μνήμης. Αν οι προσπελάσεις γίνονται με ρυθμό μεγαλύτερο απ'αυτόν που μπορεί να εξυπηρετήσει ο ελεγκτής, αναγκαστικά αυτές θα σειριοποιούνται με το αντίστοιχο πλήγμα στις επιδόσεις.

Άλλα μειονεκτήματα της σχεδίασης είναι τα εξής. Κάθε 8-πύρηνος κόμβος είναι στην ουσία ένα ζευγάρι τετραπύρηνων επεξεργαστών που και αυτοί δεν είναι παρά δύο διπύρηνοι «κολλημένοι» στο ίδιο die – βλέπε σχήμα 6-2 (Harpertown Processors, 2012). Αυτή η χαλαρή σύζευξη σημαίνει ότι ένα σημαντικό μέρος της κίνησης για την εξασφάλιση της συνέπειας μνήμης περνάει μέσα απ'τον κοινό διάδρομο μειώνοντας περαιτέρω το διαθέσιμο εύρος ζώνης για τα δεδομένα. Τέλος, η θέση του ελεγκτή πάνω στην Northbridge δηλαδή εκτός chip, σημαίνει ότι ο τελευταίος λειτουργεί με πολύ χαμηλότερη ταχύτητα σε σχέση με την αρχιτεκτονική *Nehalem*.



Σχήμα 6-2: Οργάνωση κόμβου Harpertown

Απεναντίας, η αρχιτεκτονική *Nehalem* που είναι η βάση της νέας γενιάς επεξεργαστών της Intel, εγκαταλείπει την παραδοσιακή οργάνωση κοινού διαδρόμου προς όφελος μιας NUMA αρχιτεκτονικής για λόγους κλιμακωσιμότητας. Ένας κόμβος *Nehalem-EP* (βλέπε σχήμα 6-3 (Nehalem-EP Processors, 2012)) αποτελείται από έναν ή περισσότερους πολυπύρηνους που ο καθένας έχει δικιά του μνήμη και δικό του *on-die* ελεγκτή. Ο ελεγκτής εξυπηρετεί όλες τις αιτήσεις προς την τοπική μνήμη είτε από τον τοπικό είτε από τους μακρινούς επεξεργαστές. Οι αιτήσεις και τα δεδομένα που έρχονται από μακριά, μεταφέρονται μέσω ενός δικτύου υψηλού εύρους ζώνης που διασυνδέει τους επεξεργαστές και φέρει το όνομα *QuickPath Interconnect* (QPI) – το αντίστοιχο του *HyperTransport* στην αρχιτεκτονική *Barcelona* (AMD). Η διαφορά του QPI με το δεύτερο είναι ότι οι επεξεργαστές χρειάζονται το πολύ δύο άλματα (έναντι τριών) για να προσπελάσουν την μακρινή μνήμη, με αποτέλεσμα το πρωτόκολλο συνάφειας να είναι απλούστερο και το *memory latency* μόλις κατά 30% υψηλότερο. (Kanter, 2012)



Σχήμα 6-3: Οργάνωση ενός κόμβου Nehalem-EP

Το μεγάλο πλεονέκτημα της παραπάνω αρχιτεκτονικής έγκειται στην κλιμακωσιμότητα της οργάνωσης μνήμης η οποία επιτρέπει την ταυτόχρονη χρήση των DIMMs που ανήκουν σε διαφορετικούς ελεγκτές. Μια εφαρμογή που είναι *memory-bound* μπορεί να ωφεληθεί τα μάλα από την προσεκτική κατανομή των δεδομένων σε όλες τις μνήμες του συστήματος έτσι ώστε να έχουμε αυξημένο συνολικό εύρος ζώνης μνήμης (ίσο με το άθροισμα των ξεχωριστών *bandwidths*). Τέλος, να σημειώσουμε ότι οι συγκεκριμένοι επεξεργαστές υποστηρίζουν και την τεχνολογία *Hyper-Threading* η οποία εμφανίζει κάθε φυσικό πυρήνα ως δύο λογικούς στο λειτουργικό σύστημα.

6.2 Μετρήσεις σε ένα κόμβο

Για τις μετρήσεις εντός κόμβου χρησιμοποιούμε τους πολυπύρηνους *Intel Xeon E5405 @ 2.00GHz (Harpertown)* και *Intel Xeon X5560 @ 2.80GHz (Nehalem)*. Η διαδικασία έχει ως εξής: αρχικά τρέχουμε τα προγράμματα μόνο με ένα νήμα και σε κάθε βήμα διπλασιάζουμε τον αριθμό των νημάτων μέχρι να πληρωθούν όλοι οι πυρήνες του κόμβου (8 και για τις δύο αρχιτεκτονικές). Όσον αφορά τον κόμβο *Nehalem*, οι μετρήσεις επεκτείνονται ως και τα 16 νήματα με ενεργοποίηση της τεχνολογίας *Hyper-Threading*. Τα προγράμματα OpenMP μεταγλωττίζονται με `-O3`, ενώ τα προγράμματα Chapel με `--fast` (αντίστοιχο του `-O3`) και `--local` (υπόδειξη προς τον μεταγλωττιστή ότι δεν θα γίνουν αναφορές σε άλλα locales). Η έκδοση της Chapel είναι η 1.4.0.

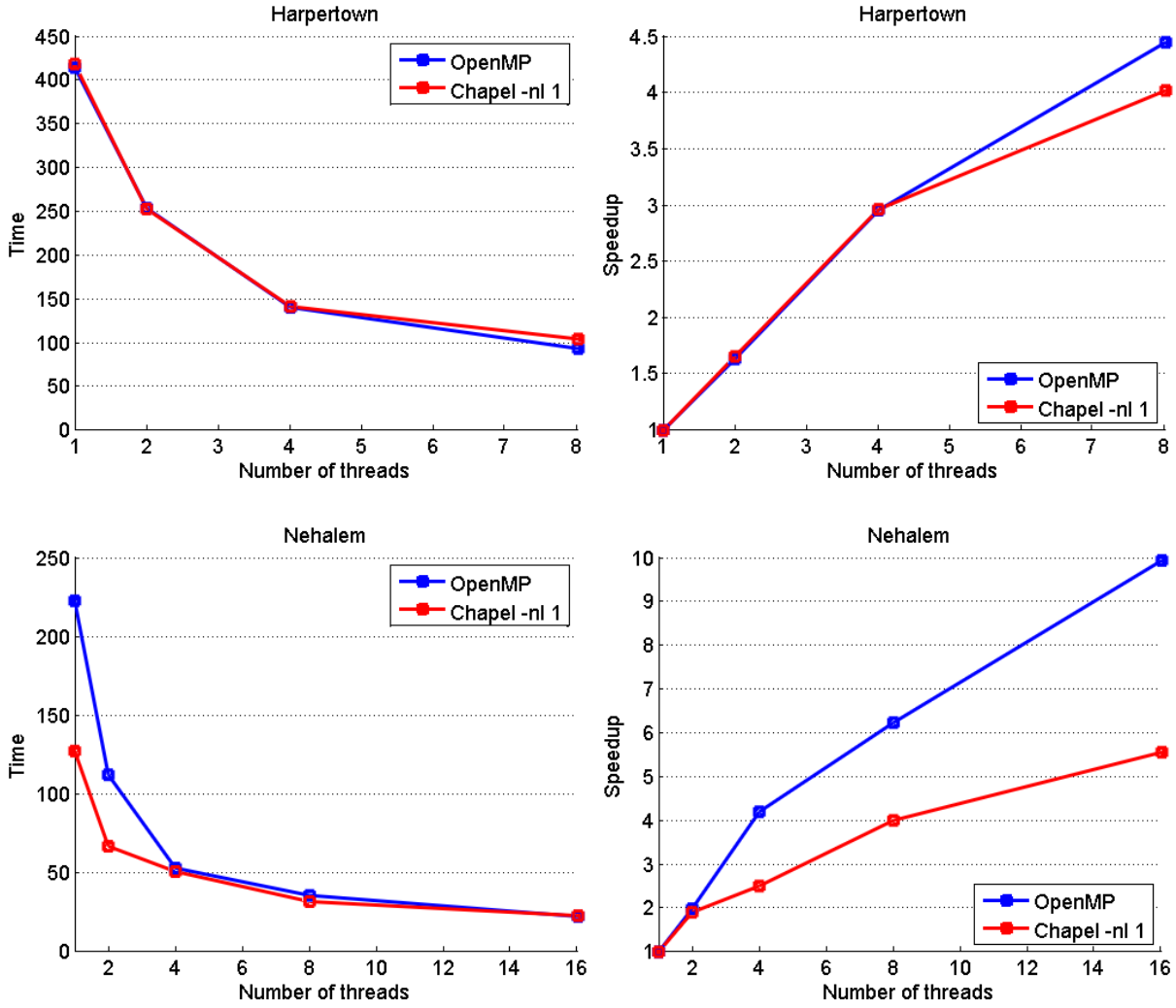
Μια σημαντική παράμετρος που επηρεάζει τις επιδόσεις και στις δύο αρχιτεκτονικές είναι ο τρόπος ανάθεσης νημάτων στους πυρήνες. Δύο νήματα που χειρίζονται τα ίδια δεδομένα μπορεί να επωφεληθούν εάν τρέχουν σε πυρήνες που έχουν την ίδια cache ή και το ανάποδο αν έχουμε καταστροφική παρεμβολή όπως στην περίπτωση που η αλλαγή μιας μη κοινής μεταβλητής ακυρώνει ολόκληρη την cache line. Η διατήρηση της συνάφειας νήματος-επεξεργαστή είναι επίσης πολύ σημαντική γιατί αποφεύγεται η άσκοπη μεταφορά δεδομένων από την μία cache στην άλλη. Στην OpenMP η συνάφεια ελέγχεται μέσω της μεταβλητής περιβάλλοντος *GOMP_CPU_AFFINITY* και μπορούμε να εξασφαλίσουμε ότι ένα νήμα θα συνεχίσει να χειρίζεται τα ίδια δεδομένα χρησιμοποιώντας στατική χρονοδρομολόγηση (*static scheduling*). Πάντως, οι μετρήσεις μας δεν έδειξαν ουσιαστικές διαφορές με την ανάθεση σε διαφορετικούς πυρήνες.

Από την πλευρά της Chapel δεν υπάρχει ούτε ο μηχανισμός για την πρόσδεση νημάτων σε συγκεκριμένους πυρήνες αλλά, και αυτό είναι το σημαντικότερο, ούτε η σταθερή αντιστοιχία νημάτων και επαναλήψεων βρόχου – σύμφωνα με τους δημιουργούς (Chamberlain B. , chapel-users archives, 17/4/2012), η αντιστοίχηση των tasks σε νήματα POSIX και των δεύτερων στους πυρήνες είναι τυχαία και άρα τα tasks/νήματα/πυρήνες που εκτελούν ένα υποσύνολο των επαναλήψεων ενός *forall* βρόχου μπορεί να είναι τελείως διαφορετικά από αυτά που εκτελούν το ίδιο υποσύνολο στο επόμενο *forall*! Προφανές είναι ότι αυτό αυξάνει την μεταπήδηση των νημάτων απ'τον ένα πυρήνα στον άλλο και το ίδιο ισχύει για τα δεδομένα που αυτά χρησιμοποιούν, έχουμε δηλαδή αύξηση των cache misses.

Η έτερη παράμετρος που είναι σημαντική μόνο για τον επεξεργαστή Nehalem είναι η τοποθέτηση της μνήμης στα memory modules των κόμβων. Όπως ξέρουμε, το Linux ακολουθεί την πολιτική του *first-touch allocation* κατά την οποία μια σελίδα της μνήμης μεταφέρεται στην cache όταν αγγίζεται για 1η φορά απ'τον αντίστοιχο πυρήνα. Πρακτικά αυτό σημαίνει ότι στην OpenMP πρέπει να αρχικοποιούμε τους πίνακες με το *scheduling* που θα χρησιμοποιήσουμε αργότερα, ενώ για την Chapel η αυτόματη αρχικοποίηση οδηγεί σε τυχαία ανάθεση για τους λόγους που αναφέραμε στην προηγούμενη παράγραφο.

Γινόμενο πινάκων

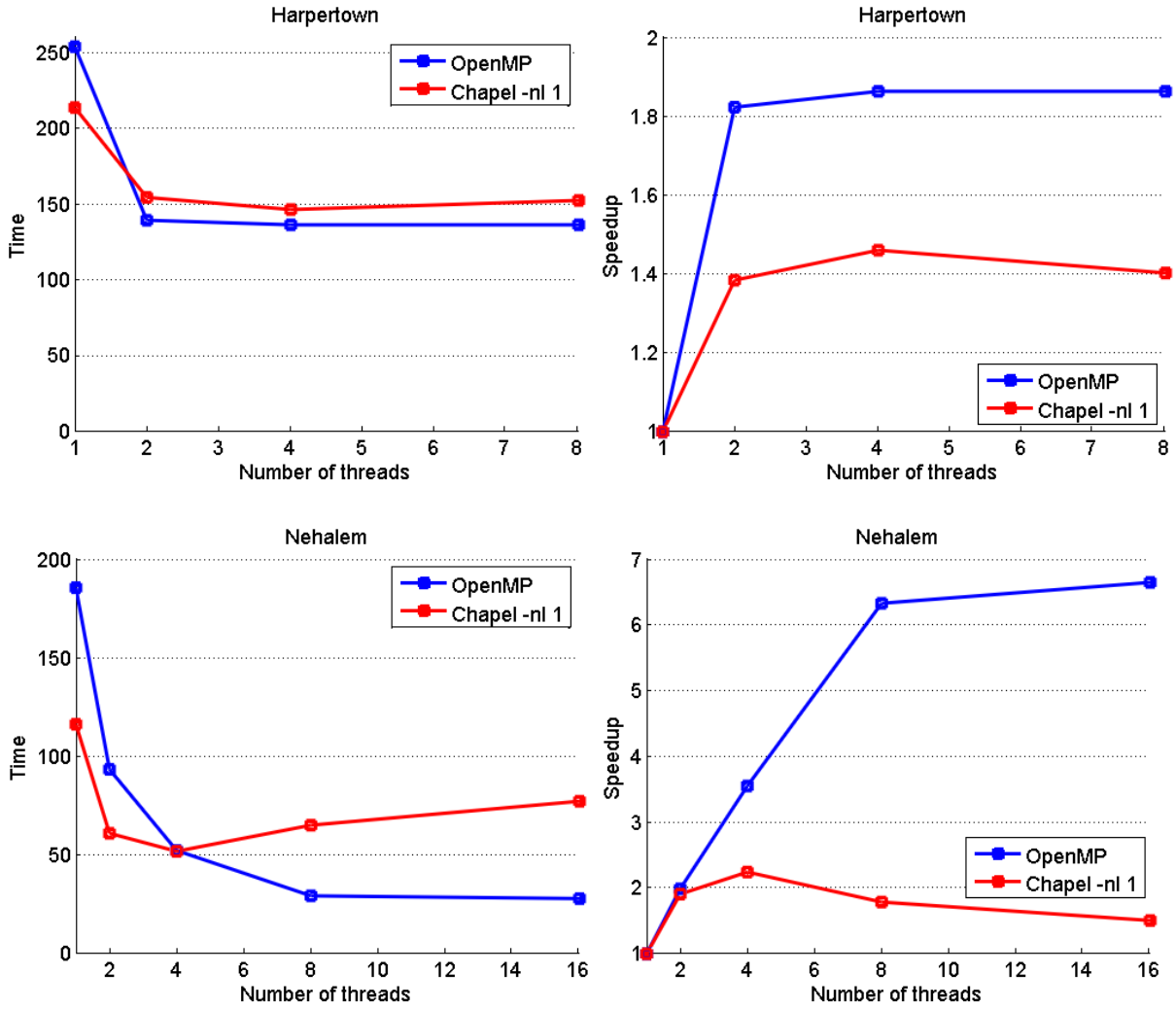
Στο σχήμα 6-4 βλέπουμε τα αποτελέσματα για το μεγαλύτερο χωρίο που τρέξαμε (πίνακες μεγέθους 2048x2048). Παρατηρούμε ότι οι επιδόσεις και η επιτάχυνση της Chapel στον Harpertown είναι εφάμιλλες αυτών της OpenMP, κάτι που είναι πολύ ενθαρρυντικό. Επίσης, η Chapel δείχνει να «ταιριάζει» καλύτερα στην αρχιτεκτονική Nehalem, όπου έχει ως και δύο φορές καλύτερους χρόνους για $n = 1, 2$ νήματα. Έτσι και με δεδομένο ότι το πρόβλημα είναι *embarrassingly parallel*, μπορούμε εύκολα να συνάγουμε ότι η Chapel είναι επαρκώς βελτιστοποιημένη για την περίπτωση του ενός κόμβου και για αλγορίθμους που δεν απαιτούν επικοινωνία μεταξύ των νημάτων.



Σχήμα 6-4: Γινόμενο πινάκων, μέγεθος 2048x2048

Floyd-Warshall

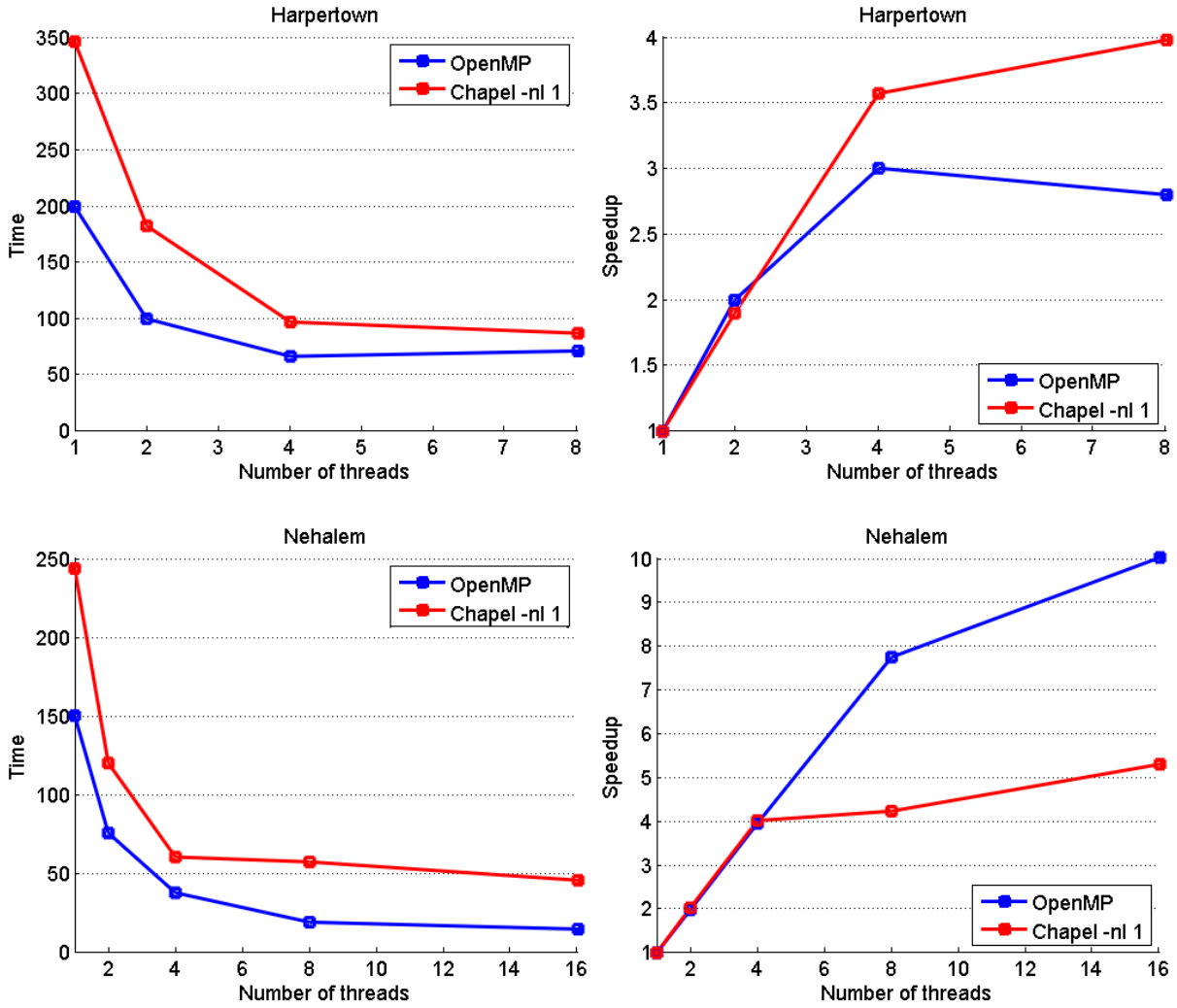
Το γεγονός ότι αυτή είναι η αφελής εκδοχή του αλγορίθμου δεν αφήνει μεγάλα περιθώρια για την επίτευξη καλής κλιμακωσιμότητας. Αυτό όμως που μας ενδιαφέρει πρωτίστως είναι η *σχετική* επίδοση της Chapel έναντι της OpenMP και όχι ο βέλτιστος αλγόριθμος. Επίσης, χρειάζεται προσοχή στο ότι χρησιμοποιούμε *double* και όχι *int* που είναι το πιο σύνηθες. Στο σχήμα 6-5 φαίνονται τα αποτελέσματα για τον μεγαλύτερο πίνακα που τρέξαμε (μεγέθους 4096x4096). Όπως βλέπουμε, στον *Harpertown* η Chapel εμφανίζει παρόμοια συμπεριφορά με την OpenMP, δηλαδή παράλληλη επιβράδυνση για περισσότερα από 2 νήματα. Στον *Nehalem* και οι δύο βελτιώνουν την επιτάχυνσή τους αλλά η Chapel έχει σημαντική επιβράδυνση για περισσότερα από 4 νήματα. Παρά την *fine-grain* επικοινωνία η Chapel εμφανίζει ικανοποιητικές επιδόσεις.



Σχήμα 6-5: Floyd Warshall, μέγεθος 4096x4096

Jacobi

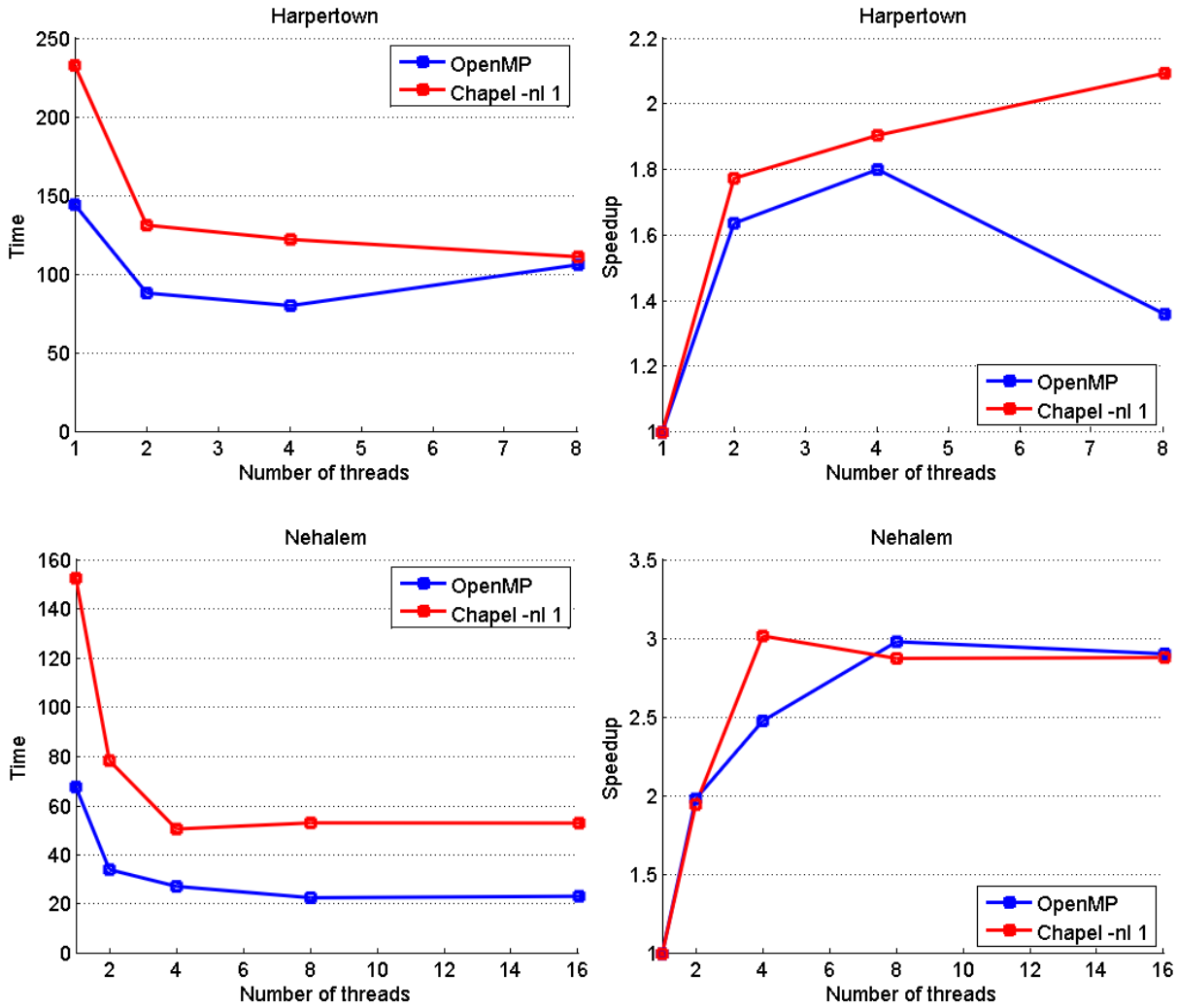
Αυτός ο αλγόριθμος έχει ακόμη μεγαλύτερες απαιτήσεις σε επικοινωνία απ'ότι ο Floyd-Warshall, γιατί σε κάθε επανάληψη τα νήματα ανταλλάσσουν δεδομένα με όλους τους γείτονές τους (έξι στην τρισδιάστατη περίπτωση). Αναμένουμε επομένως η απόδοση της Chapel να είναι χειρότερη της OpenMP. Στο σχήμα 6-6 φαίνονται τα αποτελέσματα για την περίπτωση του μεγαλύτερου πίνακα που χωράει στην μνήμη (512x512x512) και για 100 επαναλήψεις του κυρίως βρόχου. Όντως, η Chapel είναι αισθητά πιο αργή και για τις δύο αρχιτεκτονικές αλλά ειδικά στον *Harpertown* εμφανίζει καλύτερη επιτάχυνση με αποτέλεσμα να υπάρχει κάποια σύγκλιση. Βλέπουμε πάντως ότι όταν υπάρχει επικοινωνία, η OpenMP εκμεταλλεύεται καλύτερα την αρχιτεκτονική *Nehalem* απ'ότι η Chapel.



Σχήμα 6-6: Jacobi, μέγεθος 512x512x512, 100 επαναλήψεις

Black-Red

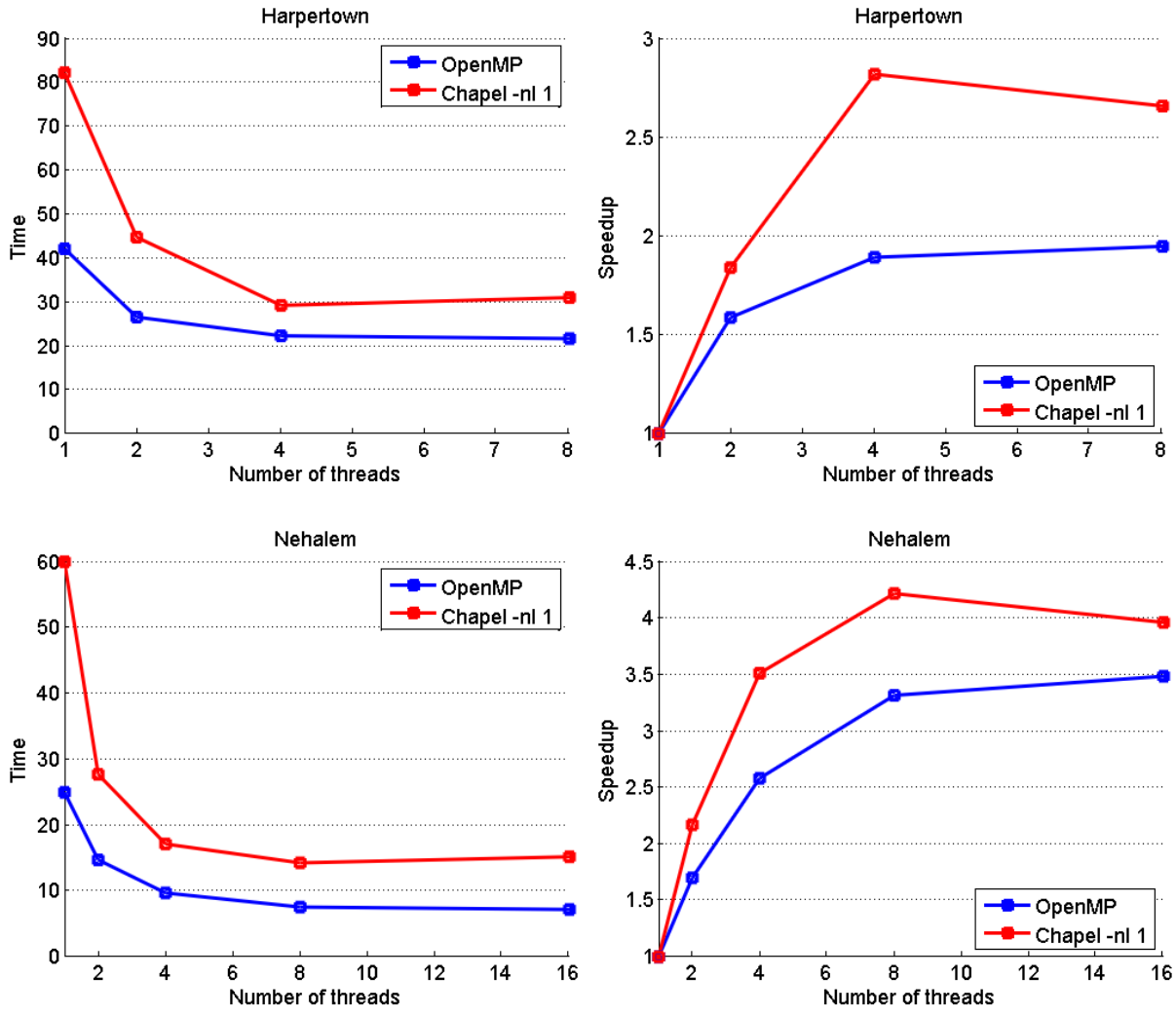
Ο Black-Red κάνει ακριβώς τις ίδιες μακρινές αναφορές με τον Jacobi και τον ίδιο αριθμό cache misses. Η συμπεριφορά όμως των δύο αλγορίθμων και στους δύο τομείς είναι ελαφρώς διαφορετική πχ ο Black-Red επικοινωνεί έπειτα απ'τους μισούς υπολογισμούς κάθε φορά και επίσης διατρέχει τον ίδιο πίνακα δύο φορές ενώ ο Jacobi διατρέχει δύο πίνακες ταυτόχρονα. Στα διαγράμματα του σχήματος 6-7 βλέπουμε τα αποτελέσματα για το ίδιο χωρίο που χρησιμοποιήσαμε στον Jacobi (512x512x512, 100 επαναλήψεις). Ως προς τον Jacobi, τόσο η OpenMP όσο και η Chapel βελτιώνουν τον χρόνο τους σχεδόν στο μισό αλλά η επιτάχυνσή τους περιορίζεται στο ½ της προηγουμένης με αποτέλεσμα οι χρόνοι για τον μέγιστο αριθμό νημάτων να είναι κατώτεροι αυτών του Jacobi. Το αποτέλεσμα αυτό δεν πρέπει να συγχέεται με την αποτελεσματικότητα των αλγορίθμων, αφού ο Black-Red χρειάζεται μόνο τις μισές επαναλήψεις για τα ίδια αποτελέσματα. Κατά τα άλλα η συμπεριφορά των OpenMP και Chapel δεν διαφέρει πολύ απ'αυτή στον Jacobi.



Σχήμα 6-7: Black-Red, μέγεθος 512x512x512, 100 επαναλήψεις

Γινόμενο αραιού πίνακα με διάνυσμα

Στην περίπτωση αυτή δεν μετρήσαμε ένα πίνακα άλλα τον μέσο χρόνο και επιτάχυνση για μια σειρά από πινάκες παρμένους από την γνωστή συλλογή αραιών πινάκων (Davis, 2012). Οι έξι πίνακες που διαλέξαμε παρουσιάζουν μεγάλη ποικιλία στην κατανομή των στοιχείων τους και είναι οι xenon2, Chebyshev4, atmosmodj, Freescale1, parabolic_fem και offshore. Τα αποτελέσματα που φαίνονται στο σχήμα 6-8 είναι για 1000 επαναλήψεις του βασικού βρόχου. Βλέπουμε ότι η Chapel υστερεί σε απόλυτους χρόνους αλλά εμφανίζει καλύτερη επιτάχυνση από την OpenMP με αποτέλεσμα να έχουμε κάποια σύγκλιση καθώς αυξάνονται τα νήματα. Πάντως, αυτό οφείλεται και στο ότι η OpenMP δεν μπορεί εύκολα να βελτιώσει τους χαμηλούς χρόνους της, γιατί ο αλγόριθμος περιορίζεται από το εύρος ζώνης μνήμης και όχι από τους υπολογισμούς.



Σχήμα 6-8: Αραιός πίνακας επί διάνυσμα, 1000 επαναλήψεις

6.3 Μετρήσεις σε πολλούς κόμβους

Για τις κατανεμημένες μετρήσεις χρησιμοποιούμε την συστοιχία *Harpertown* η οποία είναι ομοιογενής και αποτελείται από επεξεργαστές *Intel Xeon E5405 @ 2.00GHz* συνδεόμενους μέσω Gigabit Ethernet. Οι περιπτώσεις που εξετάζουμε είναι δύο: εκτέλεση σε 2 και 4 κόμβους. Σε αντιστοιχία με τα προηγούμενα, ξεκινάμε με ένα νήμα σε κάθε κόμβο και διπλασιάζουμε μέχρι να πληρωθούν όλοι οι διαθέσιμοι πυρήνες (8 ανά κόμβο). Το ίδιο κάνουμε και με την MPI με την διαφορά ότι αντί για νήματα αυξάνουμε τον συνολικό αριθμό των (μονονηματικών) διεργασιών. Για να είναι πιο δίκαιη η σύγκριση, τα προγράμματα MPI μεταγλωττίζονται με τα flags `-O3 --mca btl tcp,sm,self` που σημαίνει ότι η ανταλλαγή μηνυμάτων μέσα στον κόμβο θα χρησιμοποιεί την κοινή μνήμη. Τέλος, τα προγράμματα Chapel χρησιμοποιούν την επιλογή `--fast` που αντιστοιχεί στο `-O3`. Η έκδοση της Chapel είναι η 1.4.0.

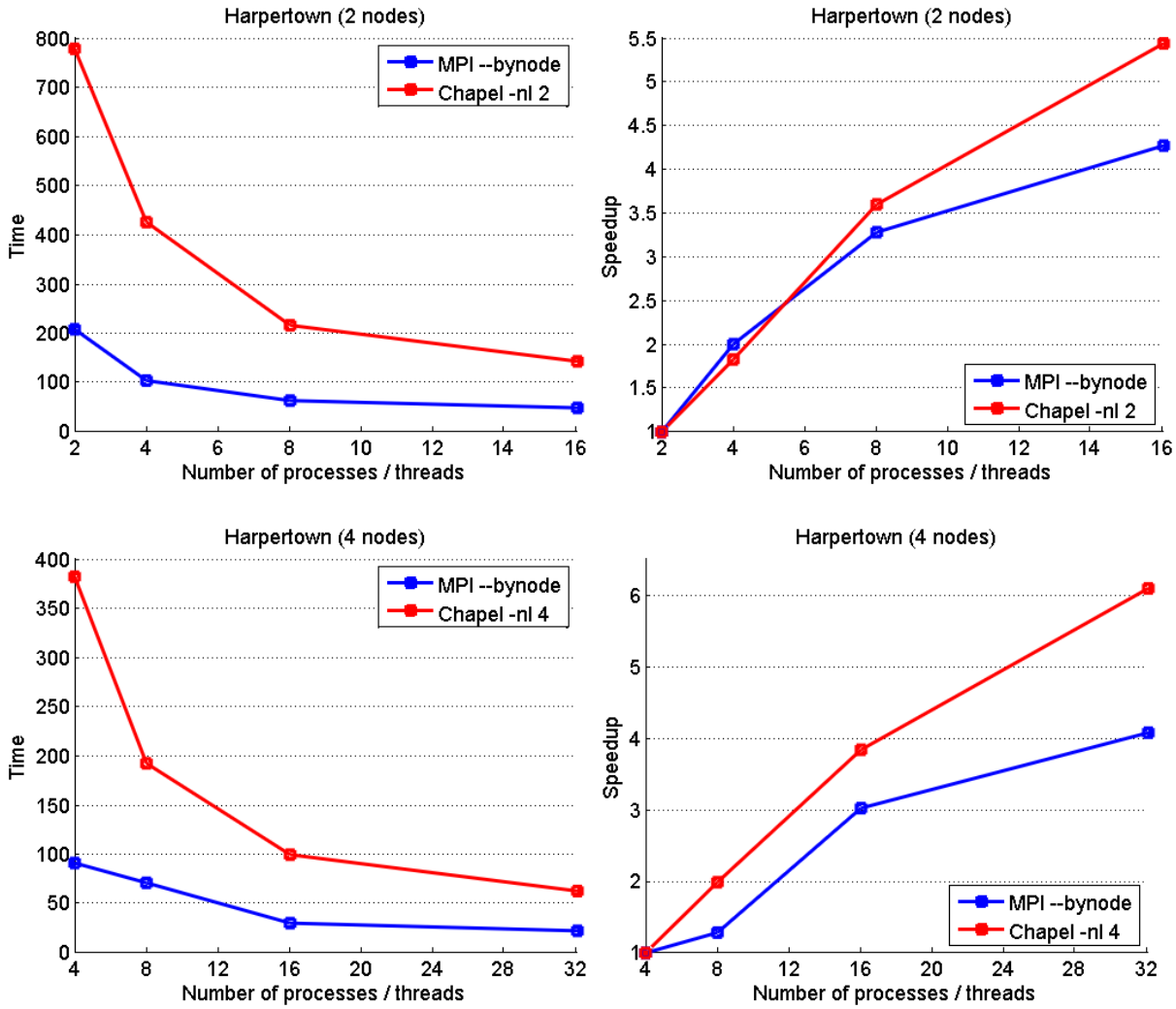
Από την φύση της η Chapel, όπως και όλες σχεδόν οι PGAS γλώσσες, απαιτεί μάλλον εξειδικευμένα δίκτυα για την υποστήριξη της υπερβολικά *fine-grain* επικοινωνίας· κάθε μακρινή αναφορά δημιουργεί ένα νέο μήνυμα εν αντιθέσει με την MPI όπου έχουμε ομαδοποίηση των δεδομένων πριν την αποστολή. Επίσης, η υλοποίηση της Chapel είναι πολύ επιφυλακτική όσον αφορά τις βελτιστοποιήσεις· ο κατανεμημένος πίνακας και το domain αυτού καταχωρούνται στο locale 0 και δεν επιτρέπεται το data caching μεταξύ των locales για να εξασφαλιστεί ότι κάθε αλλαγή στο domain θα είναι άμεσα ορατή σε όλα. Το τελευταίο ισχύει ακόμα και αν το domain ορίζεται ως σταθερό! Το αποτέλεσμα είναι ότι κάθε προσπέλαση σε στοιχείο κατανεμημένου πίνακα οδηγεί σε τρεις μακρινές αναφορές, δύο προς το locale 0 για τον πίνακα και το domain και μία για το ίδιο το στοιχείο προς το locale που είναι ο ιδιοκτήτης. Τα προηγούμενα επιβεβαιώνονται και μέσω της συνάρτησης *getCommDiagnostics()* που παρέχεται απ'την Chapel για το profiling της επικοινωνίας.

Απ'τα προηγούμενα είναι εμφανές ότι ο καθοριστικός παράγοντας δεν είναι τόσο το εύρος του δικτύου όσο η καθυστέρηση (*network latency*) η οποία είναι περίπου 0.2 ms για το δίκτυο Ethernet και προστίθεται σε κάθε αποστολή. Τα πλεονεκτήματα της MPI σε σχέση με την Chapel είναι πολλαπλά. Καταρχήν, η ομαδοποίηση των δεδομένων προς αποστολή μειώνει δραματικά τον χρόνο επικοινωνίας. Επιπλέον, υπάρχει η δυνατότητα της ασύγχρονης επικοινωνίας που σημαίνει ότι ο επεξεργαστής θέτει τις παραμέτρους της μεταφοράς και συνεχίζει με τον υπολογισμό άλλων στοιχείων. Ακόμα και αν η τελευταία δεν υποστηρίζεται από το δίκτυο, η σειρά των υπολογισμών στην MPI, με πρώτα αυτά που δεν έχουν μακρινές εξαρτήσεις, μπορεί και να απαλείψει την ανάγκη για αναμονή.

Η επιτάχυνση (*speedup*) που απεικονίζεται στα επόμενα κεφάλαια έχει ελαφρώς διαφορετικό ορισμό από αυτόν που δώσαμε στην αρχή του κεφαλαίου 6. Συγκεκριμένα, την (ξανα)ορίζουμε ως τον απόλυτο χρόνο του προγράμματος όταν αυτό εκτελείται με 1 *νήμα/διεργασία ανά κόμβο* προς τον χρόνο εκτέλεσης με *p νήματα/διεργασίες ανά κόμβο*. Άρα με άλλα λόγια μετράμε την επιτάχυνση του κατανεμημένου προγράμματος κρατώντας σταθερή την επικοινωνία και αυξάνοντας τα νήματα. Επιπλέον, ισχύουν αυτά που είπαμε στο κεφάλαιο 6.2 περί ανάθεσης νημάτων και μνήμης στους πυρήνες.

Γινόμενο πινάκων

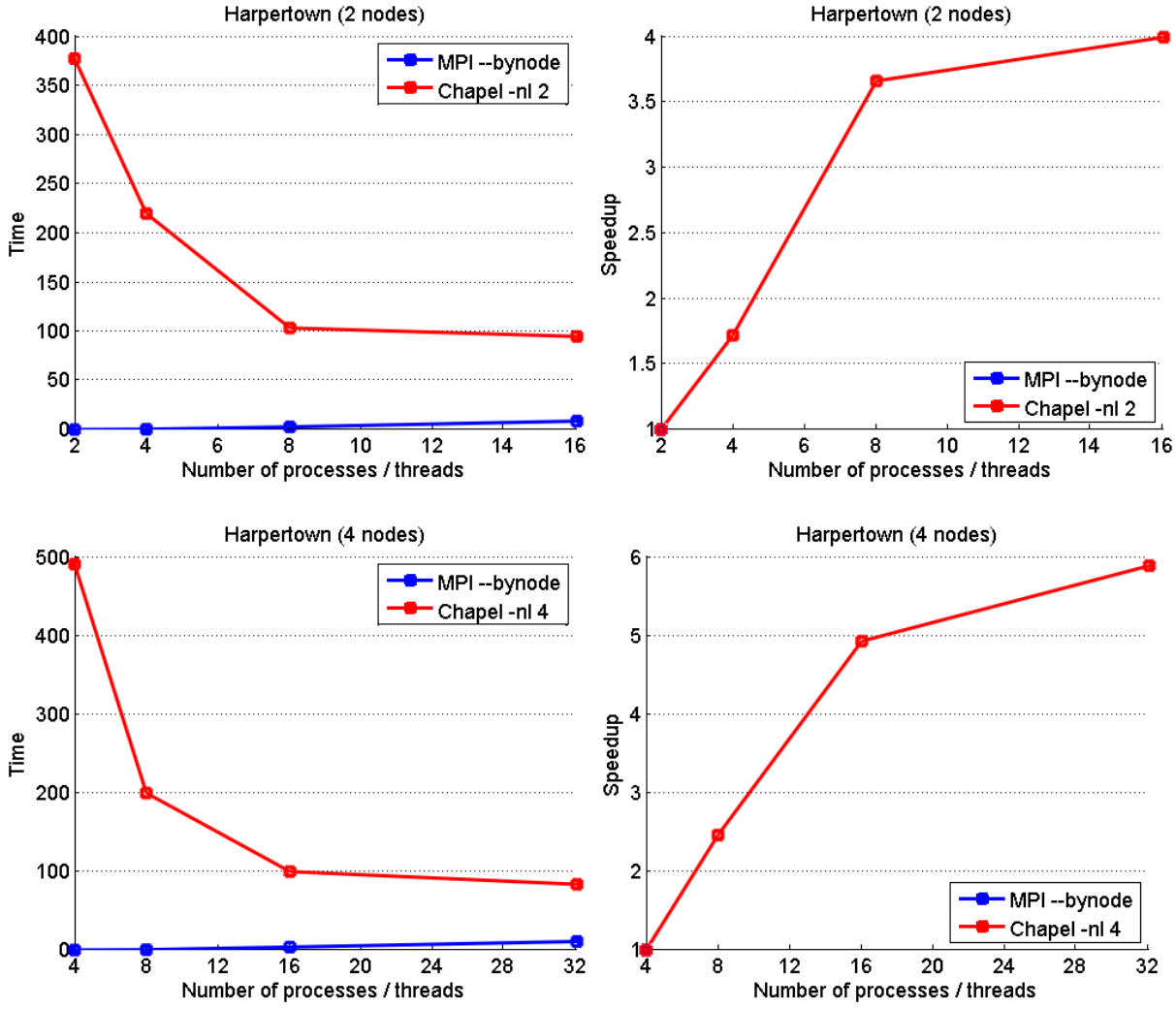
Στο γινόμενο πινάκων όπου δεν έχουμε επικοινωνία η Chapel έχει συμπεριφορά ανάλογη αυτής του σχήματος 6-4 για ένα κόμβο και για το ίδιο χωρίο. Επίσης, εμφανίζει επιτάχυνση καλύτερη της MPI αν και δεν πρέπει να ξεχνάμε ότι η Chapel δημιουργεί επιπλέον νήματα ενώ η MPI δημιουργεί διεργασίες. Δεδομένης της έλλειψης επικοινωνίας, το γεγονός ότι η Chapel για 2 και 4 κόμβους είναι σχεδόν δύο φορές πιο αργή από τους χρόνους της σε ένα κόμβο δείχνει ότι έχουμε να κάνουμε με μη βελτιστοποιημένη επικοινωνία, πχ χρήση καθολικών δεικτών οι οποίοι πρέπει να αποκωδικοποιούνται σε κάθε αναφορά. Στο σχήμα 6-9 φαίνεται η περίπτωση του πίνακα διαστάσεων 2048x2048 για 2 και για 4 κόμβους.



Σχήμα 6-9: Γινόμενο πινάκων για 2 & 4 κομβους, μέγεθος 2048x2048

Floyd-Warshall

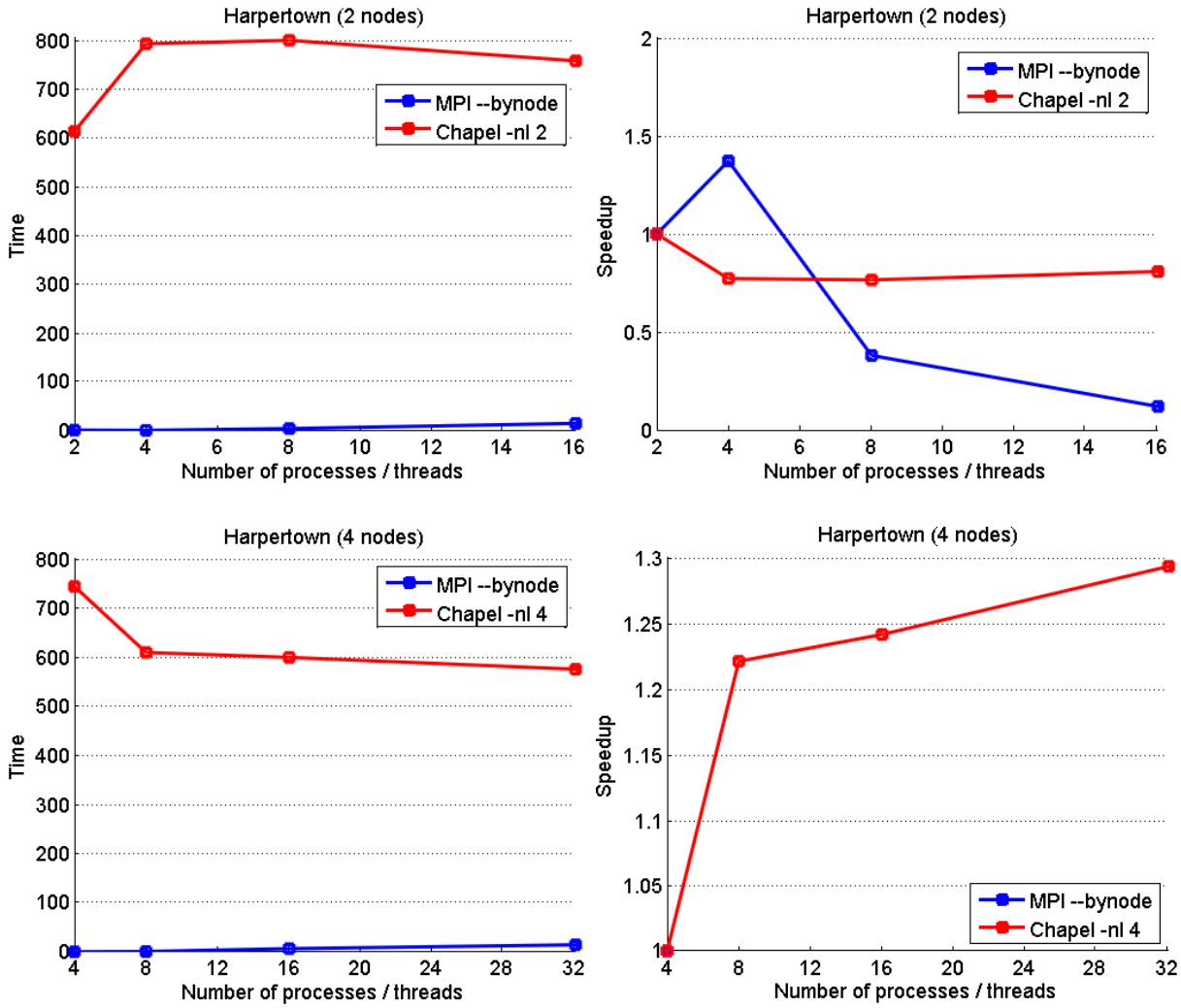
Οι πολύ κακοί χρόνοι της Chapel μας αναγκάζουν να επιλέξουμε ένα υπερβολικά μικρό χωρίο μεγέθους 128x128 που είναι μη πρακτικό για την MPI. Οι χρόνοι της που είναι ήδη κοντά στο μηδέν, με την αύξηση των διεργασιών απλά επιβαρύνονται με την επιπλέον επικοινωνία. Αυτό δεν ισχύει όμως και για την Chapel για την οποία παρατηρούμε ότι η επιτάχυνση για 2 και για 4 κόμβους είναι αντίστοιχη αυτής του σχήματος 6-5 για ένα κόμβο, έχοντας πάντα υπόψη ότι μιλάμε για ελαφρώς διαφορετικούς ορισμούς (βλέπε αρχή του κεφαλαίου). Το συμπέρασμα είναι ότι η αύξηση των νημάτων όντως μειώνει τον χρόνο επεξεργασίας, όμως ο τελευταίος δεν είναι άσχετος με την επικοινωνία όπως στην MPI αλλά εξαρτάται άμεσα από αυτή λόγω του ότι σε κάθε βήμα έχουμε και μια αίτηση για μακρινό δεδομένο οπότε οι πυρήνες περνάνε τον περισσότερο χρόνο τους στην αναμονή. Επομένως, οι περιοριστικοί παράγοντες είναι η καθυστέρηση του δικτύου, τα τρία μηνύματα ανά μακρινή προσπέλαση και, ειδικά για την υλοποίησή μας, η μεταφορά των ίδιων δεδομένων παραπάνω φορές απ'ότι είναι αναγκαίο (απόρροια της έκφρασης του προβλήματος σε καθολική μορφή, βλέπε κώδικα 5-4). Σε σχέση με την MPI, η Chapel είναι ως και 100 φορές πιο αργή και άρα μη πρακτική.



Σχήμα 6-10: Floyd Warshall για 2 & 4 κόμβους, μέγεθος 128x128

Jacobi

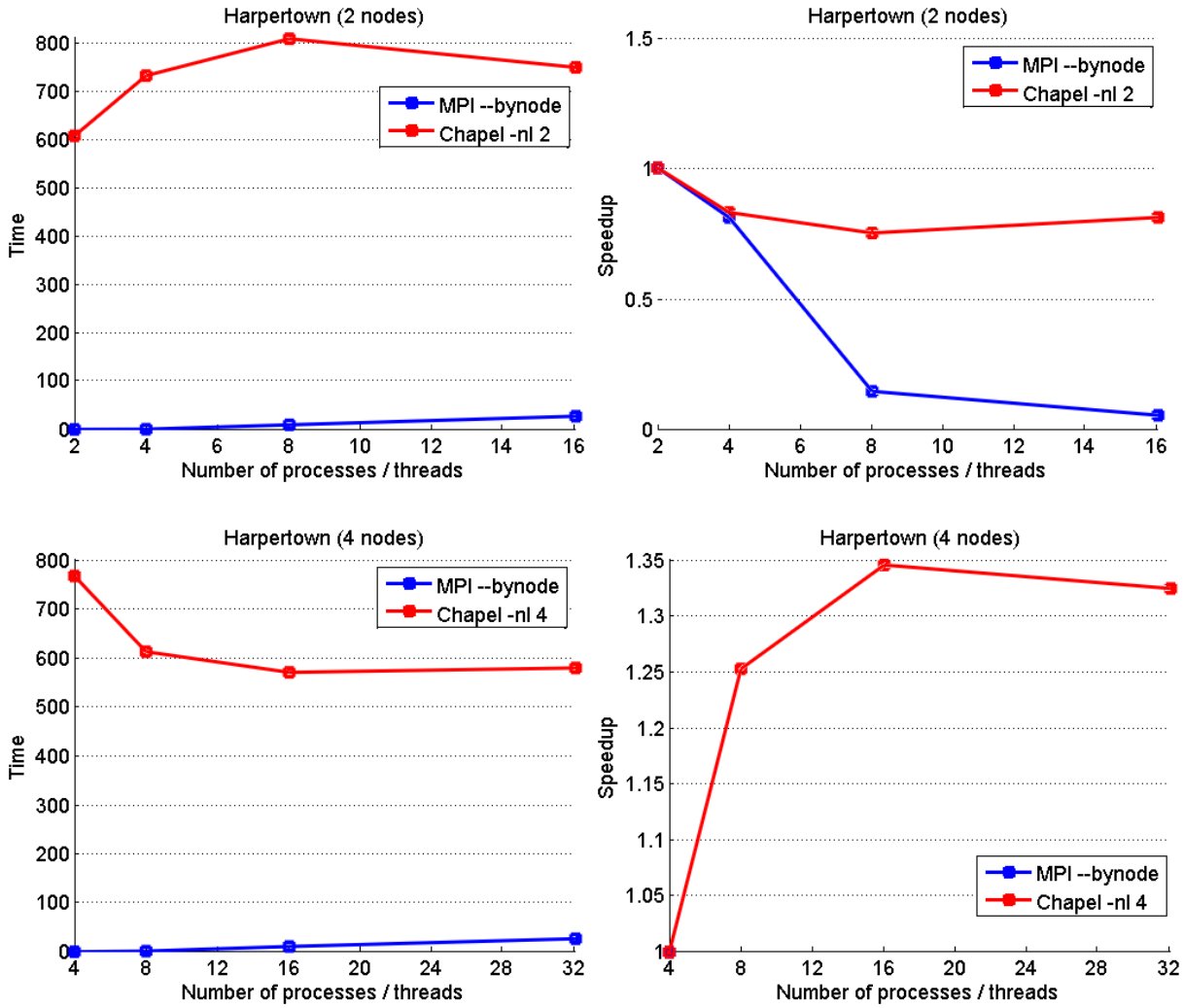
Η περίπτωση του αλγορίθμου Jacobi είναι ενδεικτική των προβλημάτων που αντιμετωπίζει η Chapel σε προβλήματα με σχετικά μεγάλες απαιτήσεις σε επικοινωνία. Στο σχήμα 6-11 βλέπουμε τους χρόνους και τις επιταχύνσεις για έναν μικρό πίνακα μεγέθους 128x128x128. Οι απόλυτοι χρόνοι της Chapel είναι συνεπείς με αυτούς στον Floyd-Warshall (είναι εκατοντάδες φορές πιο αργοί από της MPI) αλλά πλέον δεν έχουμε επιτάχυνση, για την ακρίβεια η προσθήκη νημάτων οδηγεί αρχικά σε παράλληλη επιβράδυνση και μόνο τότε έχουμε μια κάποια βελτίωση. Πάλι όπως και πριν η επιτάχυνση της MPI δεν έχει νόημα ενώ οι χρόνοι της δίνονται αυστηρά για σύγκριση με την Chapel.



Σχήμα 6-11: Jacobi για 2 & 4 κόμβους, μέγεθος 128x128x128, 100 επαναλήψεις

Black-Red

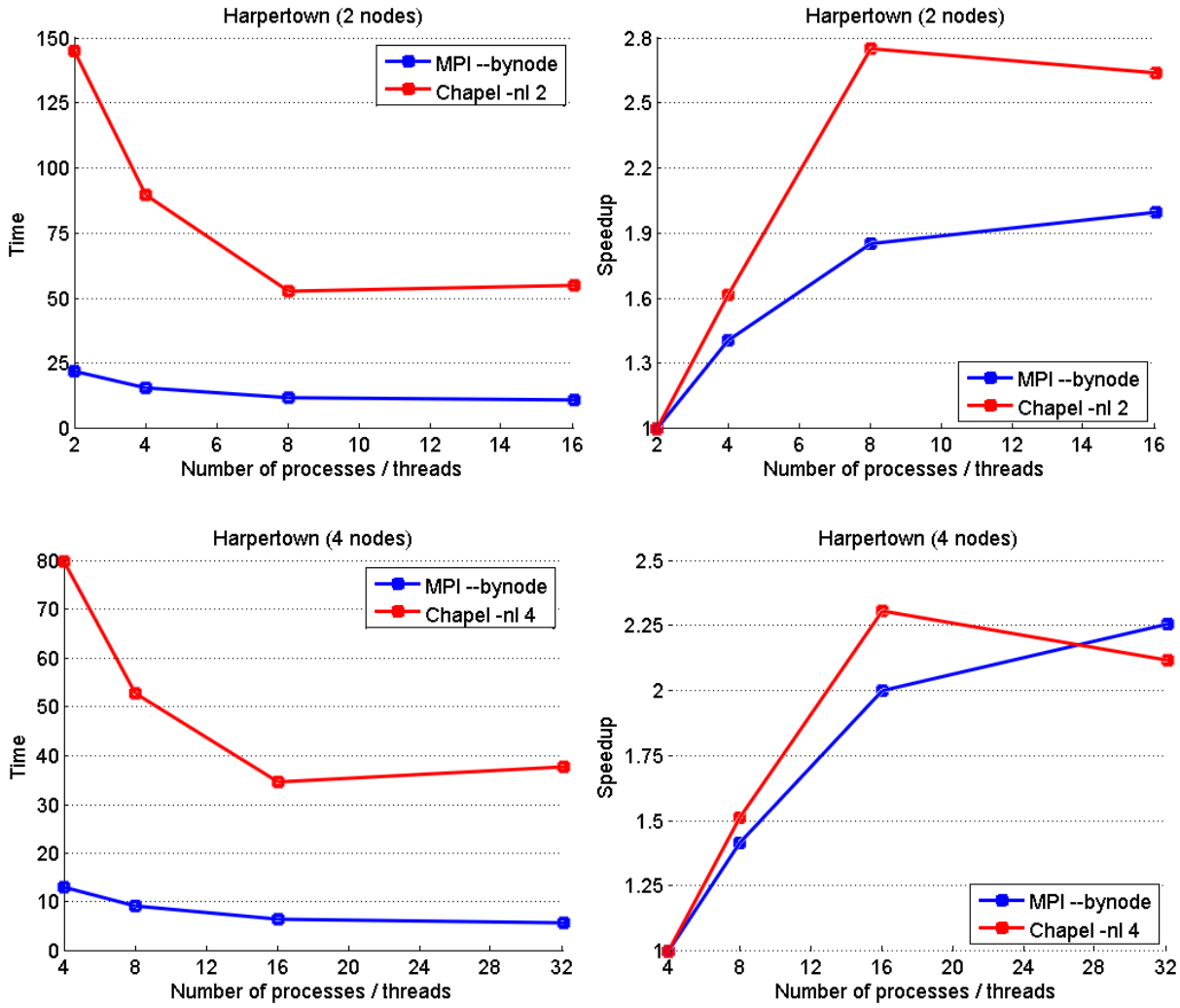
Τρέξαμε τον αλγόριθμο Black-Red για το ίδιο χωρίο με πριν (Jacobi) και όπως βλέπουμε στο σχήμα 6-12, η συμπεριφορά της Chapel είναι σχεδόν πανομοιότυπη. Αν μάλιστα συγκρίνουμε με τα αποτελέσματα των αντίστοιχων μετρήσεων για ένα κόμβο θα δούμε ότι εκεί ο Black-Red αλγόριθμος ήταν περίπου 50% πιο γρήγορος, κάτι που δεν ισχύει εδώ. Αυτό επιβεβαιώνει τα συμπεράσματα που βγάλαμε προηγουμένως, δηλαδή ότι η μεγάλη καθυστέρηση οφείλεται στο μεγάλο latency της επικοινωνίας η οποία καθυστερεί τους υπολογισμούς αλλά και στα πολλά μικρά μηνύματα που στέλνει η Chapel. Να σημειώσουμε ότι και πάλι η συμπεριφορά της MPI δεν έχει νόημα καθότι το χωρίο είναι πολύ μικρό και οι χρόνοι της δίνονται μόνο για σύγκριση.



Σχήμα 6-12: Black-Red για 2 & 4 κόμβους, μέγεθος 128x128x128, 100 επαναλήψεις

Γινόμενο αραιού πίνακα με διάνυσμα

Η περίπτωση του αραιού πίνακα επί διάνυσμα είναι ενδιαφέρουσα καθώς απ'ότι φαίνεται ο καθοριστικός παράγοντας δεν είναι τόσο το latency της επικοινωνίας όσο το εύρος ζώνης της μνήμης. Βλέπουμε, στην πρώτη σειρά του σχήματος 6-13, ότι για 2 κόμβους η επίδοση της Chapel είναι σχεδόν δύο φορές χειρότερη αυτής της μη κατανεμημένης Chapel (1ο διάγραμμα στο σχήμα 6-8) ενώ η επιτάχυνση είναι ελαφρώς χειρότερη. Καθώς όμως προσθέτουμε κόμβους και πάμε στους 4, η επίδοση βελτιώνεται και φτάνει αυτή της μη κατανεμημένης Chapel, κάτι που δεν επιτεύχθηκε σε κανέναν απ'τους προηγούμενους αλγορίθμους. Αυτό δείχνει την μεγάλη σημασία που έχει η αύξηση του εύρους ζώνης γι'αυτόν το αλγόριθμο. Να σημειώσουμε ότι η MPI εμφανίζει ξανά χειρότερη επιτάχυνση (όπως και στο γινόμενο πίνακα) κάτι που καταδεικνύει ότι το μερίδιο των υπολογισμών στο συνολικό χρόνο είναι μάλλον μικρό.



Σχήμα 6-13: Αραιός πίνακας επί διάνυσμα για 2 & 4 κόμβους, 1000 επαναλήψεις

6.4 Μέτρηση της παραγωγικότητας

Δυστυχώς, λόγω του ότι δεν κρατήσαμε τους χρόνους ανάπτυξης και αποσφαλμάτωσης, δεν μπορούμε παρά να κάνουμε μια υποκειμενική εκτίμηση στον τομέα του χρόνου. Σύμφωνα με αυτή την εκτίμηση, οι χρόνοι ανάπτυξης OpenMP και Chapel ήταν παραπλήσιοι. Αντιθέτως, ο *καθαρός* χρόνος ανάπτυξης σε MPI ήταν ίσως και δεκαπλάσιος αυτού της Chapel, κάτι που οφείλεται στο γράψιμο περισσότερου κώδικα, στην μεγαλύτερη πολυπλοκότητα του τελευταίου και στον μεγάλο χρόνο αποσφαλμάτωσης. Ο λόγος που χρησιμοποιούμε την λέξη «καθαρός» είναι ότι από την πλευρά της Chapel, ο περισσότερος χρόνος δαπανήθηκε στην εκμάθησή της (MPI και OpenMP ήταν ήδη γνωστές).

Ο έτερος τρόπος μέτρησης της παραγωγικότητας είναι οι λεγόμενες γραμμές πηγαίου κώδικα ή αλλιώς *Source Lines Of Code* (SLOCs), μια μονάδα μέτρησης η οποία είναι πολύ διαδεδομένη στον κόσμο της ανάπτυξης λογισμικού (Sommerville, 2006). Θεωρητικά, ο μικρότερος αριθμός γραμμών κώδικα συνδέεται ευθέως με τον χρόνο ανάπτυξης αλλά και με τον αριθμό των bugs που εμφανίζονται στον κώδικα όταν αυτός ξεπερνά κάποιο ελεγχόμενο μέγεθος. Η άλλη διάσταση είναι ότι ο κώδικας που έχει γραφεί σε μία απλή και περιεκτική γλώσσα είναι πολύ ευκολότερος στην κατανόηση και στην τροποποίηση, κάτι που μειώνει σημαντικά τον χρόνο ανάπτυξης. Στον πίνακα που ακολουθεί φαίνεται ο αριθμός των γραμμών κώδικα (SLOCs) για όλους τους αλγορίθμους του κεφαλαίου 5.

	OpenMP	Chapel (1 locale)	Ποσοστό (%)	MPI	Chapel (πολλά locales)	Ποσοστό (%)
Matrix Mult.	111	37	33.3	271	82	30.3
Floyd-Warshall	95	30	31.6	296	32	10.8
Jacobi	153	63	41.2	647	69	10.7
Black-Red	167	55	32.9	1018	63	6.2
Sparse x Vector	141	72	51.1	240	113	47.1
Μέσος Όρος	**133.4**	**51.4**	**38.0**	**494.4**	**71.8**	**21.0**

Πίνακας 6-14: Αριθμός γραμμών κώδικα για όλα τα προγράμματα

Στους παραπάνω αριθμούς συμπεριλαμβάνονται οι συναρτήσεις για την εκτύπωση των αποτελεσμάτων καθότι μέρος των «πραγματικών» αλγορίθμων, ενώ παραλείπεται ο κώδικας για την χρονομέτρηση. Όπως βλέπουμε, η οικονομία που επιτυγχάνεται σε κώδικα με την Chapel είναι πολύ σημαντική. Καταρχήν, στην κατανεμημένη περίπτωση το μέγεθος του μέσου προγράμματος είναι μόλις το 1/5 αυτού της MPI! Επίσης εντυπωσιακή είναι η βελτίωση απέναντι στην ήδη «λακωνική» OpenMP η οποία αγγίζει τα 2/5. Αν μάλιστα συγκρίνουμε τον αριθμό των χαρακτήρων, η διαφορά γίνεται ακόμα μεγαλύτερη. Τέλος, να επισημάνουμε ότι καθώς θα προστίθενται νέα χαρακτηριστικά στη γλώσσα, η διαφορά της Chapel με τις άλλες γλώσσες αναμένεται να μεγαλώσει (βλέπε εναλλακτικό κώδικα στο κεφάλαιο 5.5 για παράδειγμα).

6.5 Συμπεράσματα

Οι μετρήσεις μας σε ένα κόμβο έδειξαν ότι η Chapel μπορεί να ανταγωνιστεί επάξια την OpenMP όντας στην χειρότερη περίπτωση μόλις 2 φορές πιο αργή, ενώ σε κάποιους αλγόριθμους και ειδικά για μικρό αριθμό νημάτων είναι ταχύτερη! Αυτό είναι πολύ ελπιδοφόρο αν λάβουμε υπόψη ότι η γλώσσα ακόμα αλλάζει και δεν έχει δοθεί έμφαση στις βελτιστοποιήσεις. Δυστυχώς δεν μπορούμε να πούμε το ίδιο για τις κατανεμημένες μετρήσεις τουλάχιστον όσον αφορά απλά δίκτυα όπως το Gigabit Ethernet στο οποίο τρέξαμε τις δοκιμές μας. Εκεί η Chapel είναι ανταγωνιστική *μόνο* όταν δεν υπάρχει επικοινωνία οπότε και οι χρόνοι της είναι 5x αυτών της MPI – σε αντίθετη περίπτωση, είναι χιλιάδες φορές πιο αργή και η χρήση της έναντι της MPI είναι απλά μη πρακτική. Οι λόγοι γι'αυτή την υστέρηση, απ'τον σημαντικότερο στον λιγότερο σημαντικό, είναι οι εξής.

- **Απουσία εξειδικευμένου hardware**. Η επικοινωνία στην Chapel γίνεται μέσω πολλών και μικρών μηνυμάτων, ως εκ τούτου η χαμηλή καθυστέρηση του δικτύου είναι πολύ σημαντική για την επίτευξη καλής επίδοσης. Εξάλλου, ο επεξεργαστής έχει πολλά διαστήματα κατά τα οποία παραμένει αδρανής περιμένοντας τα μακρινά δεδομένα και αυτό ρίχνει τις επιδόσεις στο επίπεδο ταχύτητας του δικτύου. Πράγματι οι χρόνοι της Chapel στο δίκτυο Gigabit Ethernet του εργαστηρίου είναι ανάλογοι των χρόνων του δικτύου (3 τάξεις μεγαλύτεροι από αυτούς της MPI).
- **Μη βελτιστοποιημένη επικοινωνία**. Λόγω του πρώιμου σταδίου, δεν έχουν υλοποιηθεί πολλές βελτιστοποιήσεις· μία απ'αυτές είναι το data caching/replication του κατανεμημένου πίνακα ή τουλάχιστον του domain του (όταν αυτό είναι σταθερό) έτσι ώστε να μην χρειάζονται 3 μηνύματα ανά μακρινή αναφορά. Μια πιθανή κατεύθυνση για μελλοντική έρευνα είναι και η ενοποίηση κάποιων μηνυμάτων σε ένα μεγάλο bulk μήνυμα, ίσως καθ'υπόδειξη του προγραμματιστή.
- **Τυχαία ανάθεση των tasks**. Όπως προαναφέραμε στο κεφάλαιο 6.2, η τωρινή υλοποίηση αναθέτει τα tasks στα νήματα και τα νήματα στους πυρήνες εντελώς τυχαία. Αυτό σημαίνει ότι δεν υπάρχει καμιά εγγύηση ότι δύο διαδοχικά *forall* θα αναθέσουν τους ίδιους δείκτες στους ίδιους πυρήνες (η ανάθεση των *δυνάμικων* tasks με τρόπο που να σέβονται την συνάφεια επεξεργαστή είναι ένα δύσκολο πρόβλημα). Πιθανή απώλεια της συνάφειας έχει τα αναμενόμενα αποτελέσματα· μεταφορά δεδομένων απ'την μία cache στην άλλη (*cache misses*) ή περιττές μακρινές αναφορές στην περίπτωση ενός NUMA επεξεργαστή και γενικώς οι επιδόσεις υποφέρουν.

Όσον αφορά την παραγωγικότητα, όπως είδαμε στο κεφάλαιο 6.4 η Chapel επιτυγχάνει και με το παραπάνω τον στόχο της. Πράγματι, αν υποθέσουμε ότι υπάρχει κάποια αλήθεια στο σχήμα *λιγότερες γραμμές κώδικα → λιγότερα bugs → μικρότερο κόστος ανάπτυξης*, τότε η αύξηση της παραγωγικότητας είναι της τάξης του 75%. Ούτως ή άλλως, πρόκειται περί μιας γλώσσας με τεράστια εκφραστική δύναμη και μεγάλη σαφήνεια η οποία υστερεί μόνο στην απόδοσή της στην κατανεμημένη επικοινωνία. Πάντως, είναι πολύ νωρίς για να απαντήσουμε εάν μπορεί να αποτελέσει την λύση του μέλλοντος για την ανάπτυξη παράλληλου λογισμικού.

7 Domains οριζόμενα από τον χρήστη

Αυτό το κεφάλαιο είναι το αποτέλεσμα παράπλευρης εργασίας πάνω στο API που παρέχει η Chapel για τους επίδοξους χρήστες που θέλουν να δημιουργήσουν τα δικά τους *domains*. Μια πλήρης υλοποίηση ενός κατανεμημένου πολυνηματικού domain, όπως είναι το *Block* το οποίο παρέχεται εξαρχής με την γλώσσα, θα μπορούσε κάλλιστα να αποτελέσει μια ολόκληρη διπλωματική από μόνο του, οπότε αναγκαστικά περιοριστήκαμε στην δημιουργία ενός πολυνηματικού *domain* το οποίο όμως εκτελείται μόνο σε ένα *locale*. Οι πληροφορίες που περιέχονται σε αυτό το κεφάλαιο βασίζονται κυριώς στα papers της Cray (User-Defined Parallel Zippered Iterators in Chapel, 2011) και (Authoring User-Defined Domain Maps in Chapel, 2011) αν και πολλές λεπτομέρειες της υλοποίησης έχουν έκτοτε αλλάξει καθώς το API δεν έχει ακόμα σταθεροποιηθεί.

Η βασική ιδέα πίσω από τη λειτουργία ενός πολυνηματικού iterator, όπως είναι ένα domain, είναι η διαφορετική συμπεριφορά του ανάλογα με τα συμφραζόμενα στο σημείο της κλήσης. Για παράδειγμα, ένας iterator εκτελείται σειριακά όταν καλείται μέσω *for* και παράλληλα όταν καλείται μέσω *forall*. Στην Chapel αυτό επιτυγχάνεται με τον ορισμό τριών διαφορετικών συναρτήσεων, μία για την σειριακή περίπτωση και δύο για την παράλληλη. Ο κώδικας 7-1 δείχνει τις συναρτήσεις αυτές για έναν πολύ απλό iterator ονόματι split. Το **param** είναι παράμετρος η οποία πρέπει να είναι γνωστή την ώρα της μεταγλώττισης, ενώ το **where** είναι τρόπος καθορισμού-περιορισμού των μεταβλητών εισόδου μιας συνάρτησης Chapel.

```
// Serial iterator
iter split(r: range) {
  for num in r do
    yield num;
}

// Parallel leader
iter split(param tag: iterator, r: range) where tag == iterator.leader {
  const chunk = r.length / 2;
  coforall i in 1..here.numCores do
    yield r.low+i*chunk..r.low+(i+1)*chunk;
}

// "Parallel" Follower
iter split(param tag: iterator, follower, r: range) where tag ==
 iterator.follower {
  for num in follower do
    yield num;
}
```

Κώδικας 7-1: Ορισμός ενός απλού iterator

Η λειτουργία του split είναι απλή. Όταν καλείται ως split(1..10) μέσω *for* χρησιμοποιείται η σειριακή εκδοχή του η οποία καλείται στην αρχή κάθε επανάληψης και επιστρέφει ένα ένα

τα στοιχεία της range. Αν όμως κληθεί μέσω *forall*, η range χωρίζεται στα δύο και εκτελείται παράλληλα από δύο νήματα. Ο λόγος που χρειάζονται δύο «παράλληλες» συναρτήσεις αντί για μία έχει να κάνει με την σύζευξη δύο και πλέον iterators. Το επόμενο παράδειγμα δείχνει τον μετασχηματισμό του κώδικα από τον μεταγλωττιστή στην περίπτωση αυτή (ο μετασχηματισμένος κώδικας δεν είναι έγκυρη σημασιολογικά Chapel αλλά ψευδοκώδικας).

```
// Έστω δύο iterators A, B
forall (a_item, b_item) in (A,B) do ...

// Ακολουθεί ο μετασχηματισμός του παραπάνω βρόχου σε ψευδοκώδικα
A_lead();
for (a_item, b_item) in (A_follow(), B_follow()) do ...

// Ο βρόχος for εκτελείται από όλα τα νήματα με διαφορετικά yields
```

Αυτό που βλέπουμε είναι ότι ο πρώτος τη σειρά iterator είναι αυτός ο οποίος θα καθορίσει και τον τρόπο με τον οποίο θα κατανεμηθεί η δουλειά. Η inline κλήση του *parallel leader* του A δημιουργεί τα παράλληλα tasks, όπως είδαμε και στον parallel leader του κώδικα 7-1 με το *coforall*, και επίσης αναθέτει σ'αυτά ένα τμήμα της δουλειάς. Ο βρόχος *for* που ακολουθεί εκτελείται από κάθε νήμα με κλήση στις συναρτήσεις-ακολούθους οι οποίοι απλά εργάζονται πάνω στο τμήμα που τους έχει ανατεθεί.

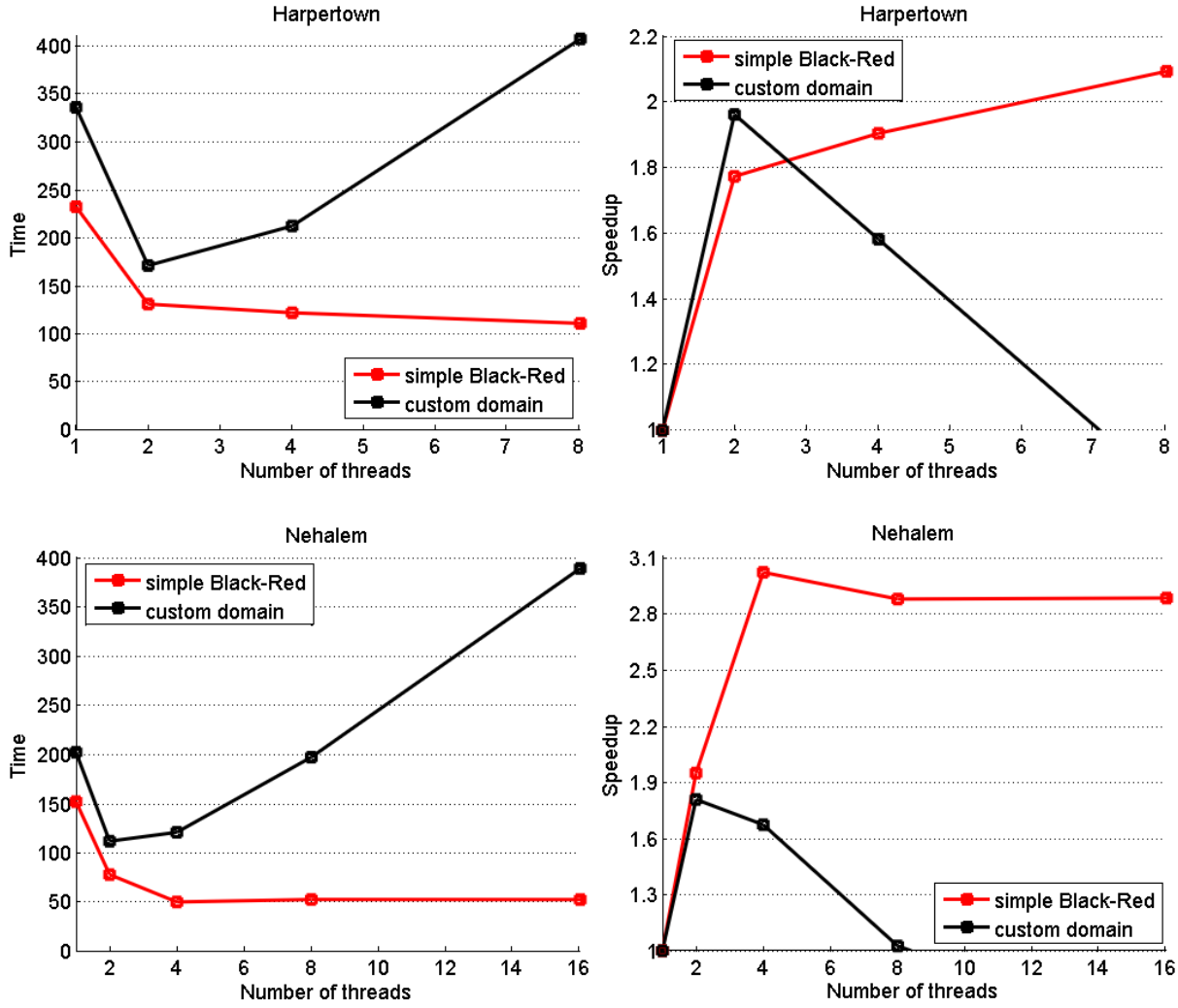
Εννοείται ότι το θέμα αυτό δεν εξαντλείται σε τόσες λίγες γραμμές καθώς ανακύπτουν διάφορα ζητήματα όπως η συμβατή αναπαράσταση της εργασίας έτσι ώστε να μπορούμε να συζεύξουμε iterators που είναι τελειώς άσχετοι μεταξύ τους και άλλα, όλα αυτά όμως ξεφεύγουν από τα πλαίσια της παρούσας διπλωματικής. Αυτό στο οποίο εμείς εστιάζουμε την προσοχή μας είναι η ευκολία ή δυσκολία συγγραφής ενός τέτοιου iterator που είναι απαραίτητος για την δημιουργία μιας νέας κατανομής καθώς και οι επιδόσεις του σε σχέση με την απλή έκδοση του αλγορίθμου του κεφαλαίου 6.2.

7.1 Αξιολόγηση του custom Black-Red domain

Παραθέτουμε την υλοποίηση του πολυνηματικού Black-Red domain στο παράρτημα Α. Όπως είναι εμφανές, η ποσότητα κώδικα που απαιτείται δεν είναι διόλου ευκαταφρόνητη (γύρω στις 160 γραμμές αν και κάποια τμήματα επαναλαμβάνονται). Επίσης, χρειάζεται κάποια εντρύφηση στους εσωτερικούς μηχανισμούς της Chapel, η έκδοση δε για πολλά locales μπορεί να φτάσει ως και τις 5000 γραμμές αν πάρουμε ως παράδειγμα τον κώδικα για την κατανομή Block. Συμπερασματικά, είναι προφανές ότι η συγγραφή μιας τέτοιας κατανομής δεν έχει νόημα αν δεν έχουμε κάποιο σοβαρό κίνητρο όπως η επίτευξη της μέγιστης απόδοσης ή η αποφυγή επανάληψης σε περίπτωση που χρησιμοποιούμε την κατανομή πολύ συχνά πχ μια κατανομή τριγωνικού πίνακα.

Στο *coforall* στην σελίδα 73 βλέπουμε ότι κάθε νήμα διατρέχει τις «σειρές» του domain διαδοχικά και εκτελεί 2 yields/σειρά με το καθένα να επιστρέφει ένα καρτεσιανό γινόμενο από *strided ranges*. Στην αρχική υλοποίηση τα νήματα έκαναν μόνο 4 yields (δηλαδή ήταν και η 1η διάσταση strided) αλλά οι επιδόσεις ήταν χειρότερες λόγω περισσότερων cache misses αφού διατρέχαμε τον πίνακα 4 φορές. Στο άλλο άκρο, δοκιμάσαμε να κάνουμε ένα yield για κάθε ζεύγος (i, j) έτσι ώστε να «σεβαστούμε» και την σειρά της 2ης διάστασης για τα ελάχιστα δυνατά cache misses, όμως αυτή η εναλλακτική ήταν επίσης χειρότερη λόγω των υπερβολικών yields. Οπότε η τωρινή υλοποίηση είναι ένας συμβιβασμός ανάμεσα στα πολλά cache misses και στα πολλά yields που και τα δύο καταβαραθρώνουν την απόδοση.

Στο σχήμα 7-1 βλέπουμε τα αποτελέσματα για χωρίο μεγέθους 512x512x512 και 100 επαναλήψεις, όπου με κόκκινο φαίνονται τα αποτελέσματα για την απλή υλοποίηση του κεφαλαίου 6.2. Όπως βλέπουμε, οι χρόνοι για περισσότερα από δύο νήματα χειροτερεύουν καθώς αυξάνονται τα yields και μένουν ίδια τα cache misses. Αυτή η συμπεριφορά είναι και θεωρητικά αναμενομένη γιατί λόγω της υπερβολικά fine-grain υφής της κατανομής Black-Red, δεν είναι δυνατό να αποφύγουμε τα περιττά cache misses χωρίς να αυξήσουμε υπέρμετρα τα yields και άρα το overhead.



Σχήμα 7-1: custom vs simple Black-Red, μέγεθος 512x512x512, 100 επαναλήψεις

Βιβλιογραφία

Chapel Language Specification Version 0.91. (2012, April 25). Ανάκτηση από http://chapel.cray.com/spec/spec-0.91.pdf

Harpertown Processors. (2012, April 25). Ανάκτηση από http://www.nas.nasa.gov/hecc/support/kb/Harpertown-Processors_78.html

Linear map. (2012, April 25). Ανάκτηση από http://en.wikipedia.org/wiki/Linear_map

MPI: A Message-Passing Interface Standard Version 2.2. (2012, April 25). Ανάκτηση από http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

Nehalem-EP Processors. (2012, April 25). Ανάκτηση από http://www.nas.nasa.gov/hecc/support/kb/Nehalem-EP-Processors_79.html

OpenMP Application Program Interface Version 3.1. (2012, April 25). Ανάκτηση από http://www.openmp.org/mp-documents/OpenMP3.1.pdf

Parallel Programming Paradigms. (2012, April 25). Ανάκτηση από http://daugerresearch.com/vault/parallelparadigm.shtml

Sparse matrix. (2012, April 25). Ανάκτηση από http://en.wikipedia.org/wiki/Sparse_matrix

Barney, B. (2012, April 25). *POSIX Threads Programming*. Ανάκτηση από https://computing.llnl.gov/tutorials/pthreads/

Chamberlain, B. (2011). Programming Models and Chapel: Landscaping for Exascale Computing. *INT Exascale Workshop*.

Chamberlain, B. (2012, 4 17). *chapel-users archives*. Ανάκτηση από http://sourceforge.net/mailarchive/forum.php?forum_name=chapel-users

Chamberlain, B. L., Choi, S.-E., Deitz, S. J., & Navarro, A. (2011). User-Defined Parallel Zippered Iterators in Chapel. *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS 2011)*. Galveston Island, Texas.

Chamberlain, B. L., Choi, S.-E., Deitz, S. J., Iten, D., & Litvinov, V. (2011). Authoring User-Defined Domain Maps in Chapel. *Cray Users Group 2011 (CUG 2011)*. Fairbanks, Alaska.

Davis, T. (2012, April 25). *The University of Florida Sparse Matrix Collection*. Ανάκτηση από http://www.cise.ufl.edu/research/sparse/matrices/

El-Ghazawi, T. (2012, April 25). *Unified Parallel C - UPC Tutorial*. Ανάκτηση από http://www2.hpcl.gwu.edu/pgas09/tutorials/upc_tut.pdf

Fog, A. (2012, April 25). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Ανάκτηση από http://www.agner.org/optimize/instruction_tables.pdf

Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., & Koziris, N. (2008). Understanding the Performance of Sparse Matrix-Vector Multiplication. *16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2008)*. Toulouse, France.

Hennessy, J. L., & Patterson, D. A. (2011). Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann.

Kanter, D. (2012, April 25). *Inside Nehalem: Intel's Future Processor and System*. Ανάκτηση από http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719

Karniadakis, G. E., & Kirby II, R. M. (2003). Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation. Cambridge University Press.

Markidis, S. (2012, April 25). *PGAS languages*. Ανάκτηση από http://www.pdc.kth.se/research/newprgmodels/pgas-languages

Mathews, J. H. (2012, April 25). *Jacobi and Gauss-Seidel Iteration*. Ανάκτηση από http://math.fullerton.edu/mathews/n2003/gaussseidelmod.html

Sommerville, I. (2006). *Software Engineering*. Addison Wesley.

Sutter, H. (2012, April 25). *Super Linearity and the Bigger Machine*. Ανάκτηση από http://www.drdobbs.com/article/print?articleId=206903306

Yelick, K. (2007). Parallel Languages: Past, Present and Future. *HOPL-III*. San Diego, California.

Yelick, K., & Lusk, E. (2007). Languages for High-Productivity Computing: The DARPA HPCS Language Project. *Parallel Processing Letters, 17*(1), 89-102.

Παράρτημα Α: Κώδικας για *user-defined Black-Red domain*

```
// red is the color of the 1st element, always
enum color {red, black};

// returns prime factors of num, for num >= 2
proc factor(num: int)
{
  var x = num, cnt = 2, idx = 0;
  var D: domain(int);
  var A: [D] int;

  if num < 1 then
    halt("factor called with num < 1");

  if num == 1 {
    D.add(1);
    A(1) = 1;
    return A;
  }

  while cnt <= x {
    if x % cnt == 0 then {
      idx += 1;
      D.add(idx);
      A(idx) = cnt;
      x = x / cnt;
    } else
      cnt += 1;
  }

  return A;
}


// returns partition for domain eg [2 3 1] for 6 tasks
proc partition(numTasks: int, numDims: int)
{
  var A = factor(numTasks),  // returns int array
      idx: int;

  // equalize # of factors in A to # of dims
  while A.numElements > numDims {
    A(2) *= A(1);
    for i in 1..(A.numElements-1) do
      A(i) = A(i+1);
    A.domain.remove(A.numElements);

    // sort the array
    idx = 1;
    while idx <= A.numElements-1 && A(idx) > A(idx+1) {
      A(idx) <=> A(idx+1);
      idx += 1;
    }
  }
```

```
  idx = A.numElements + 1;
  while A.numElements < numDims {
    A.domain.add(idx);
    A(idx) = 1;
    idx += 1;
  }

  return A;
}


// Parallel leader
iter RedBlack3D(param tag: iterator, D: domain, c: color)
  where tag == iterator.leader && D.rank == 3
{
  const redIsOdd   = if (+ reduce [i in 1..D.low.size] D.low(i)) % 2
                       then true else false,
        blackIsOdd = !redIsOdd;  // convenience

  const numTasks = if dataParTasksPerLocale == 0
                   then here.numCores else dataParTasksPerLocale;

  const myPartition = partition(numTasks, 3);

  const (X, Y, Z) = (1, 2, 3),  // convenience
        (Xblocks, Yblocks, Zblocks) = (myPartition(X), myPartition(Y),
                                       myPartition(Z)),
        XYblocks = Xblocks*Yblocks;

  const blockSizes = [d in X..Z] D.dim(d).length / myPartition(d),
        remaining  = [d in X..Z] D.dim(d).length % myPartition(d);

  coforall n in 0..(numTasks-1) {
    const ijk = ((n % XYblocks) % Xblocks, (n % XYblocks) / Xblocks,
                 n / XYblocks);

    proc getLimits(d) {
      const x = ijk(d),
            s = D.dim(d).low +
                     x*blockSizes(d) + min(x,  remaining(d)),
            e = D.dim(d).low +
                   (x+1)*blockSizes(d) + min(x+1,remaining(d)) - 1;
      return (s,e);
    }

    const (xlow, xhigh) = getLimits(X),
          (ylow, yhigh) = getLimits(Y),
          (zlow, zhigh) = getLimits(Z);

    const y_odd  = ylow + abs(ylow+1) % 2,
          y_even = ylow + abs(ylow)   % 2,
          z_odd  = zlow + abs(zlow+1) % 2,
          z_even = zlow + abs(zlow)   % 2;


                    /* procedure continued in next page */
```

```
    if ((c == color.red && redIsOdd) || (c == color.black && blackIsOdd)) {
      for i in xlow..xhigh {  // yield odd
        if i % 2 {
          yield [i..i, y_odd..yhigh  by 2, z_odd..zhigh  by 2];
          yield [i..i, y_even..yhigh by 2, z_even..zhigh by 2];
        } else {
          yield [i..i, y_even..yhigh by 2, z_odd..zhigh  by 2];
          yield [i..i, y_odd..yhigh  by 2, z_even..zhigh by 2];
        }
      }
    } else {
      for i in xlow..xhigh {  // yield even
        if i % 2 {
          yield [i..i, y_odd..yhigh  by 2, z_even..zhigh by 2];
          yield [i..i, y_even..yhigh by 2, z_odd..zhigh  by 2];
        } else {
          yield [i..i, y_even..yhigh by 2, z_even..zhigh by 2];
          yield [i..i, y_odd..yhigh  by 2, z_odd..zhigh  by 2];
        }
      }
    }
  }  /* end of coforall */
}


// Parallel follower
iter RedBlack3D(param tag: iterator, follower, D: domain, c: color)
  where tag == iterator.follower && D.rank == 3
{
  for idx in follower do
    yield idx;
}


// Serial iterator
iter RedBlack3D(D: domain, c: color) where D.rank == 3
{
  const redIsOdd   = if (+ reduce [i in 1..D.low.size] D.low(i)) % 2
                       then true else false,
        blackIsOdd = !redIsOdd;  // convenience

  const xfactor = if (c == color.red && redIsOdd) ||
                     (c == color.black && blackIsOdd) then 1 else 0;

  // pre-compute the two sub-ranges for k's iteration
  const (xDim, yDim, zDim) = (D.dim(1), D.dim(2), D.dim(3)),
        (zlow, zhigh) = (zDim.low, zDim.high),
        ranges = (zlow..zhigh by 2, (zlow+1)..zhigh by 2);

  for i in xDim {
    for j in yDim {
      const lastRange = ranges[1 + abs(i+j+zlow+xfactor) % 2];
      for k in lastRange do
        yield (i,j,k);
    }
  }
}
```