Πτυχιακή Εργασία
Παπαδογιαννάκης Αλέξανδρος
Οκτώβριος 2012

Τοπολογία επεξεργαστών και επιδόσεις μνήμης σε πολυπύρηνα συστήματα

Επιβλέπων:
Βασίλειος Δημακόπουλος

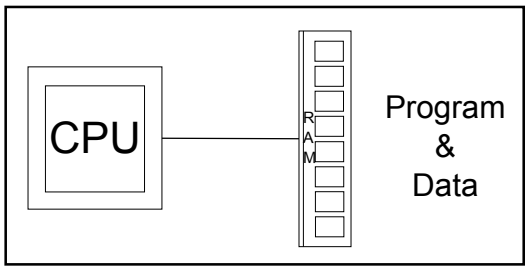
Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

Στην αρχή της ιστορίας τους, οι υπολογιστές απαρτίζονταν από μία κεντρική μονάδα επεξεργασίας συνδεδεμένη με μία μνήμη. Στην απλή αυτή αρχιτεκτονική (μοντέλο von Neumann, σχήμα 1.1) οι εντολές και τα δεδομένα βρίσκονται αποθηκευμένα στη μνήμη. Ο επεξεργαστής συνεχώς διαβάζει μία εντολή από την μνήμη, την εκτελεί και αποθηκεύει το αποτέλεσμα της πίσω στην μνήμη, πριν να διαβάσει την επόμενη εντολή.



Σχήμα 1.1: Μοντέλο von Neumann

Στη διαρκή προσπάθεια βελτίωσης των επιδόσεων των επεξεργαστών ακολουθήθηκαν δύο κατευθύνσεις. Η πρώτη ήταν η βελτίωση της τεχνολογίας, αυξάνοντας τον αριθμό των πυλών (gates) και μικραίνοντας το μέγεθος τους, γεγονός που επιτρέπει την αύξηση της συχνότητας (Hz) στην οποία λειτουργούν, συνεπώς αυξάνοντας τον αριθμό των εντολών που εκτελούνται σε μια μονάδα χρόνου. Η δεύτερη ήταν μέσω της βελτίωσης της αρχιτεκτονικής του, δημιουργώντας νέες οργανώσεις, και υλοποιώντας νέες τεχνικές υπολογισμών, εκμεταλλευόμενοι τις δυνατότητες της διαθέσιμης τεχνολογίας.

Η τεχνολογία πλέον φτάνει σε κάποιο όριο με το μέγεθος των πυλών να είναι πολύ μικρό και η αύξηση της συχνότητας να έχει καταστεί δύσκολη. Επιπλέον η μεγάλη αύξηση στην κατανάλωση καταντάει απαγορευτική την περαιτέρω αύξηση της συχνότητας. Η αύξηση των επιδόσεων με αυτόν τον τρόπο έχει κορεστεί. Επομένως οι προσπάθειες για βελτίωση επικεντρώνονται κυρίως στη βελτίωση της αρχιτεκτονικής. Η πλέον διαδεδομένη τεχνική είναι η παραλ-

ληλοποίηση στο επίπεδο των επεξεργαστών. Χρησιμοποιώντας περισσότερους από έναν επεξεργαστές μπορούμε να αυξήσουμε την επεξεργαστική μας ισχύ μέχρι και Ν φορές (όπου Ν ο αριθμός των επεξεργαστών). Τα συστήματα αυτά ονομάζονται παράλληλα μιας και επιτρέπουν την ταυτόχρονη εκτέλεση κώδικα.

Στις μέρες μας έχουμε πρόσβαση σε πολυπύρηνα συστήματα χωρίς να χρειάζεται να έχουμε πρόσβαση σε έναν υπέρ-υπολογιστή μιας και όλοι οι καινούργιοι επεξεργαστές που κατασκευάζονται αποτελούνται από περισσότερους από έναν πυρήνες ή ακόμα και από περισσότερα από ένα dies στο ίδιο τσιπ. Ο αριθμός των πυρήνων που συναντάμε στους επεξεργαστές όλο και αυξάνεται. Πλέον έχουν αρχίσει να χρησιμοποιούνται πολυπύρηνοι επεξεργαστές και σε μικρότερα συστήματα όπως τα κινητά και τα tablets.

Η αύξηση όμως των πυρήνων δεν σημαίνει απαραίτητα και αύξηση στις επιδόσεις. Τα προγράμματα πρέπει να είναι κατάλληλα γραμμένα ώστε να να εκμεταλλεύονται τους επιπλέον πυρήνες. Γι' αυτό πλέον ο παράλληλος προγραμματισμός είναι απαραίτητος.

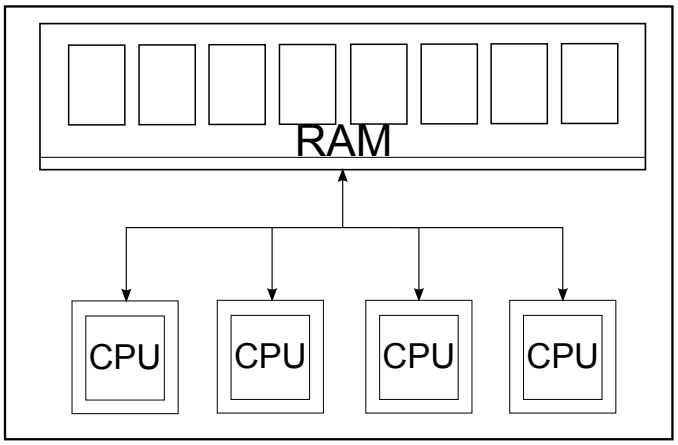
1.1 Αρχιτεκτονικές παράλληλων συστημάτων και προγραμματισμός

Ο προγραμματισμός των πρώτων παράλληλων συστημάτων ήταν πολύ δύσκολος. Από τότε όμως έχουν αλλάξει πολλά στον τομέα του προγραμματισμού, με την δημιουργία γλωσσών προγραμματισμού που είναι ανεξάρτητες από το υλικό. Όσον αφορά τον παράλληλο προγραμματισμό έχουν κυριαρχήσει δύο κυρίως μοντέλα, το μοντέλο κοινού χώρου διευθύνσεων και το μοντέλο μεταβίβασης μηνυμάτων, που αντιστοιχούν στις δυο μεγάλες κατηγορίες παράλληλων συστημάτων, αυτά της κοινόχρηστης μνήμης και αυτά της κατανεμημένης μνήμης.

1.1.1 Συστήματα κοινής μνήμης

Τα συστήματα κοινόχρηστης μνήμης αποτελούνται από μία κοινή μνήμη στην οποία έχουν πρόσβαση όλοι οι επεξεργαστές μέσω κάποιου κοινού δίαυλου επικοινωνίας (σχήμα 1.2). Κάθε επεξεργαστής εκτελεί μια διαφορετική εργασία (είτε μια διεργασία είτε κάποιο νήμα της αν αποτελείται από περισσότερα από ένα) από τους άλλους αλλά όλοι έχουν πρόσβαση στα ίδια δεδομένα και "βλέπουν" τις αλλαγές που κάνουν οι άλλοι επεξεργαστές πάνω σε αυτά.

Συνήθως σε αυτά τα συστήματα χρησιμοποιείται το προγραμματιστικό μοντέλο κοινού χώρου διευθύνσεων, όπου όλες οι εργασίες που εκτελούνται παράλληλα έχουν πρόσβαση σε έναν κοινό χώρο διευθύνσεων. Το γεγονός αυτό κάνει τον προγραμματισμό του σχετικά εύκολο μιας και είναι αρκετά κοντά στον σειριακό, αλλά ο προγραμματιστής πρέπει να λάβει υπόψιν του τις παράλληλες προσπελάσεις στην μνήμη, και να χρησιμοποιήσει μηχανισμούς προ-



Σχήμα 1.2: Σύστημα κοινόχρηστης μνήμης

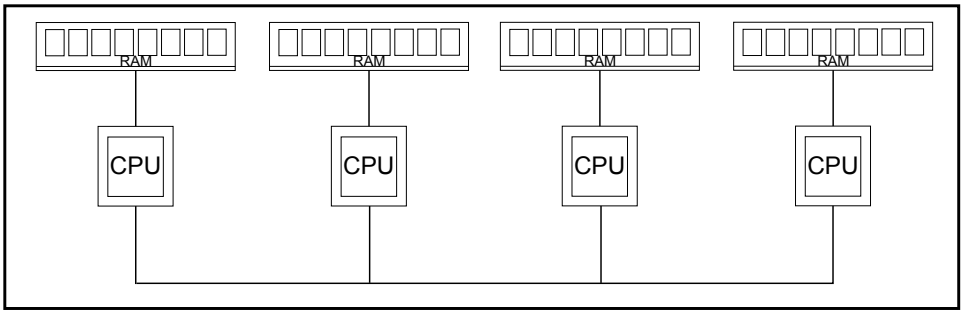
στασίας, όπως πχ. κλειδαριές για να διασφαλίσει την ορθότητα τους. Συνήθως προτιμάται η χρήση νημάτων λόγω του ότι όλα τα νήματα έχουν πρόσβαση στον χώρο διευθύνσεων της διεργασίας που τα δημιούργησε. Το μοντέλο κοινής μνήμης είναι ιδιαίτερα διαδεδομένο σε εφαρμογές που τρέχουν σε προσωπικούς υπολογιστές, που συνήθως αποτελούνται από έναν πολυπύρηνο επεξεργαστή.

Η πιο διαδεδομένη πλατφόρμα για τον προγραμματισμό κοινού χώρου διευθύνσεων είναι τα posix νήματα (Pthreads), τα οποία παρέχουν κλήσεις για τη δημιουργία και το συντονισμό νημάτων. Μία άλλη πλατφόρμα που διευκολύνει πολύ την δημιουργία παράλληλων προγραμμάτων είναι το OpenMP. Χρησιμοποιώντας το OpenMP ο προγραμματιστής μπορεί να μετατρέψει το ένα σειριακό πρόγραμμα σε παράλληλο με τη χρήση μερικών απλών οδηγιών (directives) χωρίς να χρειάζεται να ασχοληθεί με το συντονισμό των νημάτων και την προστασία των κοινόχρηστων μεταβλητών μιας και τα αναλαμβάνει το OpenMP.

1.1.2 Συστήματα κατανεμημένης μνήμης

Τα συστήματα κατανεμημένης μνήμης αποτελούνται από συστάδες επεξεργαστών κάθε ένας εκ των οποίων διαθέτει την δικιά του μνήμη (σχήμα 1.3). Οι επεξεργαστές είναι συνδεδεμένοι μέσω κάποιου δικτύου. Ένας επεξεργαστής έχει πρόσβαση μόνο στην ιδιωτική του μνήμη οπότε για να προσπελάσει δεδομένα που βρίσκονται σε κάποια απομακρυσμένη μνήμη, πρέπει να ζητήσει από τον επεξεργαστή στου οποίου τη μνήμη βρίσκονται να τα στείλει μέσω του δικτύου. Τα συστήματα αυτά, τα οποία συνήθως αποτελούνται από πολλούς επεξεργαστές ανεξάρτητους μεταξύ τους, που επικοινωνούν μέσω κάποιου δικτύου έχουν επικρατήσει στον προγραμματισμό συστημάτων υψηλών επιδόσεων (high performance computing). Σε αυτά η από κοινού χρήση κάποιας μνήμης είναι συνήθως αδύνατη, λόγω του μεγάλου αριθμού επεξεργαστών και της συμφόρησης που θα προκαλούνταν αν υπήρχε μονάχα μία κοινόχρηστη μνήμη,

αλλά και επειδή πολλές φορές αποτελούνται από πολλούς ανεξάρτητους υπολογιστές.



Σχήμα 1.3: Σύστημα κατανεμημένης μνήμης

Εδώ χρησιμοποιείται το μοντέλο μεταβίβασης μηνυμάτων όπου η επικοινωνία μεταξύ των παράλληλων διεργασιών γίνεται μέσω μηνυμάτων. Σε αυτό το μοντέλο ο προγραμματιστής πρέπει να μοιράσει κατάλληλα τα δεδομένα χρησιμοποιώντας μηνύματα και να φροντίσει για την διαχείριση των κοινόχρηστων δεδομένων ενημερώνοντας όλα τα αντίγραφα που υπάρχουν στις διάφορες μνήμες κάθε φορά που μια διεργασία τα αλλάζει. Δημοφιλέστερη πλατφόρμα για αυτό το μοντέλο είναι το MPI, το οποίο καθορίζει μια σειρά από συναρτήσεις, ανεξάρτητες από κάποια γλώσσα προγραμματισμού για τη διαχείριση των μηνυμάτων μεταξύ των επεξεργαστών. Σήμερα υπάρχουν πολλές υλοποιήσεις του MPI σε διάφορες γλώσσες

Στις μέρες μας τα συστήματα που συναντάμε είναι συνήθως συνδυασμός των δύο αυτών κατηγοριών. Με την αύξηση της διαθεσιμότητας και άρα και την μείωση τις τιμής των πολυπύρηνων επεξεργαστών, τα κατανεμημένα συστήματα τους χρησιμοποιούν πλέον κατά κόρον. Ως αποτέλεσμα στα συστήματα αυτά οι πυρήνες που βρίσκονται στον ίδιο επεξεργαστή έχουν πρόσβαση σε μια κοινόχρηστη μνήμη. Σε άλλα συστήματα χρησιμοποιούμε ιδιωτικές μνήμες ανά επεξεργαστή ή ομάδα επεξεργαστών, στις οποίες όμως έχουν πρόσβαση και οι άλλοι επεξεργαστές με κάποια καθυστέρηση (αρχιτεκτονική NUMA κεφάλαιο 4).

1.1.3 Μοντέλα προγραμματισμού παράλληλων συστημάτων

Πλέον υπάρχουν στη διαθεση μας διάφορες πλατφόρμες για προγραμματισμό παράλληλων συστημάτων όπως τα Pthreads, το OpenMP, το MPI, το OpenCL κ.λ.π. που βασίζονται σε αυτά τα μοντέλα συστημάτων. Η καταλληλότητα της κάθε πλατφόρμας στην επίλυση ενός προβλήματος κρίνεται κάθε φορά σύμφωνα με το πρόβλημα και το σύστημα που έχουμε στην διάθεση μας. Πολλές από αυτές, όπως για παράδειγμα το OpenMP που βασίζεται κυρίως στο μοντέλο κοινής μνήμης και χρησιμοποιεί νήματα για την παραλληλοποίηση του κώδικα μας, μας επιτρέπουν να μετατρέψουμε το αρχικά σειριακό πρόγραμμα μας χρη-

σιμοποιώντας μόνο μερικές επιπλέον εντολές στα σημεία του σειριακού κώδικα που επιθυμούμε να εκτελεστεί παράλληλα. Άλλες "κρύβουν" το σύστημα από τον προγραμματιστή επιτρέποντας του έτσι να χρησιμοποιήσει προγραμματισμό κοινού χώρου διευθύνσεων ακόμα και όταν το πρόγραμμα εκτελείται σε διαφορετικούς υπολογιστές όπου δεν υπάρχει κάποια κεντρική μνήμη.

Δεν αρκεί όμως απλά να μετατρέψουμε το πρόγραμμα μας από σειριακό σε παράλληλο. Οι αποδόσεις ενός προγράμματος είναι άμεσα συνδεδεμένες και με το σύστημα στο οποίο τρέχει, φαινόμενο το οποίο είναι ιδιαίτερα έντονο στον παράλληλο προγραμματισμό . Μπορεί μία υλοποίηση να είναι πολύ γρήγορη σε ένα σύστημα αλλά όχι και τόσο καλή σε ένα άλλο. Επιπλέον η μέγιστη αύξηση ταχύτητας που μπορούμε να επιτύχουμε σε ένα παράλληλο πρόγραμμα σε σχέση με το αντίστοιχο σειριακό εξαρτάται από το ποσοστό του προγράμματος που είναι παραλληλοποιήσιμο. Η μέγιστη αυτή αύξηση είναι διατυπωμένη στον νόμο του Amdahl [1].

1.2 Αντικείμενο πτυχιακής εργασίας

Η εργασία αυτή ασχολείται με τη μελέτη των παράλληλων συστημάτων καθώς και των εργαλείων που έχουμε στη διάθεση μας προκειμένου να μπορεί ένα παράλληλο πρόγραμμα να αντλήσει τις μέγιστες δυνατές επιδόσεις από το διαθέσιμο υλικό. Πιο συγκεκριμένα θα μιλήσουμε για την τοπολογία σύγχρονων συστημάτων δίνοντας έμφαση στις επιδόσεις της μνήμης, και θα δούμε πώς η χρήση συστημάτων NUMA μπορεί να λύσει το πρόβλημα της περιορισμένης ταχύτητας επικοινωνίας (bandwidth) με τη μνήμη όταν αυτή προσπελαύνεται από περισσότερους από έναν επεξεργαστές.

Όπως είπαμε οι επιδόσεις ενός προγράμματος είναι άρρηκτα συνδεδεμένες με την τοπολογία του συστήματος στο οποίο εκτελείται. Η ταυτόχρονη προσπέλαση της μνήμης από πολλούς επεξεργαστές μπορεί να αποβεί μοιραία για τις επιδόσεις του προγράμματος μας. Αφού αρχικά δούμε τις κύριες διαφορές μεταξύ των διαδεδομένων αρχιτεκτονικών θα χρησιμοποιήσουμε τη βιβλιοθήκη hwloc (Κεφάλαιο 2.3) σε συνδυασμό με το ευρέως διαδεδομένο μετροπρόγραμμα Stream (Κεφάλαιο 3) για να διεξάγουμε πειράματα σε αντιπροσωπευτικά δείγματα αρχιτεκτονικών. Σκοπός των πειραμάτων αυτών είναι η μελέτη της ταχύτητα μεταφοράς της μνήμης (bandwidth) στις διάφορες αρχιτεκτονικές όταν αυτή προσπελαύνεται από περισσότερους του ενός επεξεργαστές.

Η βιβλιοθήκη hwloc είναι μια μεταφέρσιμη (portable) βιβλιοθήκη γραμμένη σε γλώσσα προγραμματισμού C που μας παρέχει κλήσεις για την αναγνώριση της τοπολογίας του συστήματος καθώς και επιλογής συγκεκριμένων μονάδων επεξεργασίας, στις οποίες θα εκτελεστεί το πρόγραμμα μας. Μας παρέχει μια διεπαφή (API) μέσω της οποίας μπορούμε, ανεξάρτητα του λειτουργικού στο οποίο εκτελούμε το πρόγραμμα μας, να αναγνωρίσουμε τον αριθμό των sockets/επεξεργαστών/νημάτων, τον αριθμό, το μέγεθος καθώς και τη συνδεσμολογία των cache με τους επεξεργαστές όπως επίσης και το μέγεθος και τους

κόμβους - στην περίπτωση που το σύστημα μας είναι ανομοιόμορφης προσπέλασης μνήμης NUMA (Κεφάλαιο 4) - της μνήμης.

Το Stream είναι ένα σειριακό πρόγραμμα που μετράει το bandwidth της μνήμης με τέσσερα απλά πειράματα που κάνουν συνεχώς αιτήσεις στη μνήμη για προσπέλαση δεδομένων. Εκτός από τη σειριακή έκδοση, μπορεί να γίνει compile με την χρήση OpenMP για τη μέτρηση σε πολυπύρηνα συστήματα. Λόγω του ότι θέλαμε να ελέγχουμε ποιοι και πόσοι επεξεργαστές θα χρησιμοποιούνταν σε κάθε πείραμα, δε χρησιμοποιήσαμε την έκδοση του Stream που χρησιμοποιεί OpenMP, αλλά παραλληλοποιήσαμε τη σειριακή έκδοση με τη χρήση νημάτων.

1.3 Οργάνωση πτυχιακής εργασίας

Στη συνέχεια ακολουθεί μια περιληπτική περιγραφή των κεφαλαίων που ακολουθούν:

- Κεφάλαιο 2: Παρουσίαση των διαφόρων διαδεδομένων αρχιτεκτονικών. Αναζήτηση εργαλείων για την αναγνώριση της αρχιτεκτονικής του κάθε συστήματος. Τέλος, παρουσίαση της βιβλιοθήκης hwloc και της διεπαφής (API) που μας παρέχει για χρήση στα προγράμματα μας.

- Κεφάλαιο 3: Ανάλυση του προγράμματος Stream καθώς και των αλλαγών που κάναμε στον κώδικα του, για την παραλληλοποίηση του και τη χρήση του στα πειράματα μας.

- Κεφάλαιο 4: Παρουσίαση της αρχιτεκτονικής ανομοιόμορφης προσπέλασης μνήμης (NUMA), καθώς και των εργαλείων που έχουμε στη διάθεση μας για τη διαχείριση τους στα Linux μέσω του NUMA API.

- Κεφάλαιο 5: Περιγραφή των συστημάτων στα οποία διεξήχθησαν πειράματα και στη συνέχεια παρουσίαση και ανάλυση των αποτελεσμάτων των πειραμάτων σε κάθε ένα από αυτά.

- Κεφάλαιο 6: Επισκόπηση της πτυχιακής.

Κεφάλαιο 2

Τοπολογία Συστημάτων

Στο κεφάλαιο αυτό θα ασχοληθούμε με την τοπολογία των επεξεργαστών σε ένα παράλληλο σύστημα και με τα εργαλεία που μας παρέχει ένα σύστημα τύπου Unix προκειμένου να την ανακαλύψουμε και να αντλήσουμε πληροφορίες για αυτήν.

2.1 Βασικά στοιχεία ενός υπολογιστικού συστήματος

Τα βασικότερα σημεία της αρχιτεκτονικής ενός υπολογιστικού συστήματος αφορούν στο υποσύστημα επεξεργαστή – μνήμης.

Σε ένα κλασικό σειριακό σύστημα η σημαντικότερη ίσως διαφοροποίηση που συναντάμε σε σχέση με άλλα σειριακά συστήματα είναι η οργάνωση της κρυφής μνήμης (cache), δηλαδή ο αριθμός των cache, το μέγεθος τους, καθώς και ο αριθμός των επιπέδων τους.

Στα παράλληλα συστήματα όπου υπάρχουν τουλάχιστον 2 επεξεργαστικές μονάδες συναντάμε πολύ περισσότερες διαφοροποιήσεις ανάμεσα στα συστήματα. Ένα σύστημα μπορεί να αποτελείται από πολλές υποδοχές για επεξεργαστές (sockets). Ο τρόπος που συνδέονται τα sockets μεταξύ τους καθώς και με την κύρια μνήμη (SMP ή NUMA), επιδρά άμεσα στις επιδόσεις της εφαρμογής μας και θα πρέπει να λαμβάνεται υπόψιν αν δε θέλουμε να καταλήξει να εκτελείται σειριακά, ειδικά εάν αποτελείται από απλές εντολές που χειρίζονται μεγάλο όγκο δεδομένων. Επιπλέον το σύστημά μας μπορεί να αποτελείται από διαφορετικούς επεξεργαστές (heterogeneous computing systems), γεγονός που αυξάνει την πολυπλοκότητα τους.

Σε συστήματα που αποτελούνται από πολυπύρηνους επεξεργαστές, ένας επεξεργαστής παύει να συγκροτεί μια αυτοτελή οντότητα και πλέον αποτελείται από περισσότερες από μια επεξεργαστικές μονάδες (πυρήνες/cores) που λειτουργούν παράλληλα, και οι οποίες μπορεί στη συνέχεια να αποτελούνται από περισσότερα των ενός νήματων εκτέλεσης (threads). Οι πυρήνες μπορεί είτε να έχουν ιδιωτικές κρυφές μνήμες (caches), είτε να μοιράζονται κάποια επίπεδα, είτε να έχουν κοινά όλα τα επίπεδα cache.

Όσον αφορά την μνήμη συναντάμε δύο κυρίως διαφοροποιήσεις:

- Συστήματα κοινόχρηστης μνήμης
- Συστήματα κατανεμημένης μνήμης

Στα συστήματα κοινόχρηστης μνήμης όλοι οι επεξεργαστές του συστήματος "βλέπουν" και έχουν πρόσβαση σε ένα κοινό χώρο διευθύνσεων με όλα τα δεδομένα να βρίσκονται στην ίδια μνήμη και κάθε πρόσβαση σε αυτά περνάει από έναν κοινόχρηστο δίαυλο. Σε αυτά ο δίαυλος αποτελεί κέντρο συμφόρησης και μπορεί να καθυστερεί όλο το σύστημα, πράγμα που καθιστά τη χρήση μεγάλου αριθμού επεξεργαστών απαγορευτική σε τέτοια συστήματα. Περιμένουμε η συνολική ταχύτητα προσπέλασης της μνήμης (bandwidth) όχι μόνο να μην αυξάνεται όταν περισσότεροι από ένας επεξεργαστές προσπαθούν να την προσπελάσουν, αλλά αντιθέτως να μειώνεται λόγω της συμφόρησης (congestion).

Σε αντίθεση με τα συστήματα κοινόχρηστης μνήμης, στα συστήματα κατανεμημένης μνήμης κάθε επεξεργαστής διαθέτει τη δική του ατομική μνήμη. Σε αυτά οποιαδήποτε κοινόχρηστη μεταβλητή υπάρχει στο πρόγραμμα μας πρέπει να διανεμηθεί σε όλους τους επεξεργαστές μέσω κάποιου πρωτοκόλλου επικοινωνίας (π.χ. μέσω sockets) και κάθε τροποποίηση τους από έναν επεξεργαστή πρέπει να κοινοποιείται σε όσους έχουν αντίγραφο της. Η αναδιανομή αυτή της μνήμης στους επεξεργαστές συνήθως είναι πολύ δύσκολο να γίνει. Σε αντίθεση με τα συστήματα κοινόχρηστης μνήμης, οι προσπελάσεις προς τη μνήμη δεν παρουσιάζουν κανένα πρόβλημα έναντι των σειριακών συστημάτων, γιατί η μνήμη και η σύνδεση του επεξεργαστή με αυτήν είναι μοναδική και άρα αναμένεται το συνολικό bandwidth του συστήματος να αποτελεί το άθροισμα των επιμέρους, δηλαδή να είναι ανάλογο του αριθμού των επεξεργαστών που χρησιμοποιούμε.

Πολλές φορές οι δυο αυτές αρχιτεκτονικές συνδυάζονται, είτε με τη χρήση πολυπύρηνων επεξεργαστών σε συστήματα κατανεμημένης μνήμης, οπότε οι πυρήνες του επεξεργαστή βλέπουν μία κοινόχρηστη μνήμη στο μηχάνημα στο οποίο είναι τοποθετημένοι, είτε με την χρήση "τοπικών" μνημών ανά επεξεργαστή στις οποίες όμως έχουν πρόσβαση και οι υπόλοιποι επεξεργαστές με κάποιο επιπλέον overhead (τα συστήματα αυτά λέγονται NUMA και θα μιλήσουμε περισσότερο γι' αυτά στο κεφάλαιο 4) .

2.2 Αναγνώριση τοπολογίας

Όπως αναφέραμε και πιο πάνω, χρειαζόμασταν ένα τρόπο να ανακαλύπτουμε την τοπολογία κάθε μηχανήματος και να μπορούμε να επιλέξουμε σε ποιον επεξεργαστή του θα εκτελεστεί το κάθε νήμα. Στα Linux μπορούμε να πάρουμε κάποιες πληροφορίες για τους επεξεργαστές διαβάζοντας τις πληροφορίες που βρίσκονται στο /proc/cpuinfo (cat /proc/cpuinfo). Εκεί υπάρχουν πληροφορίες, όπως το μοντέλο, ο αριθμός των cores, η ταχύτητα του κ.α. Όσο για την ανάθεση νημάτων σε επεξεργαστές στα Linux υπάρχει η sched_setaffinity(pid_t pid,

unsigned int cpusetsize, cpu_set_t *mask) και σε κάποιες διανομές υπάρχει και η pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset). Αυτές παίρνουν σαν τρίτο όρισμα μια δυαδική μεταβλητή η οποία σε κάθε bit που έχει άσσο σημαίνει ότι ο αντίστοιχος επεξεργαστής μπορεί να χρησιμοποιηθεί από το νήμα (bitset), την οποία μπορούμε να την ορίσουμε με κάποιες συναρτήσεις που υπάρχουν (CPU_ZERO(cpu_set_t *set), CPU_SET(int cpu, cpu_set_t *set) κλπ) [4] και μπορεί να χειριστεί το πολύ CPU_SETSIZE επεξεργαστές.

Το πρόβλημα με το /proc/cpuinfo είναι ότι δε μας παρέχει πληροφορίες για το μέγεθος της κάθε cache παρά μόνο για το συνολικό μέγεθος τους. Επιπλέον δεν είχαμε τρόπο να αντιστοιχίσουμε τον αριθμό του κάθε επεξεργαστή που χρησιμοποιεί η sched_setaffinity με τους επεξεργαστές από το /proc/cpuinfo και επιπλέον δεν υπάρχει και μεταφερσιμότητα (portability) σε άλλα λειτουργικά συστήματα. Τα παραπάνω προβλήματα επιλύει η hwloc την οποία και χρησιμοποιήσαμε για την πραγματοποίηση των πειραμάτων.

2.3 Hwloc

Η hwloc [5, 6] είναι μια βιβλιοθήκη που διαθέτει συναρτήσεις με πληροφορίες για την τοπολογία του συστήματος. Δουλεύει σε διάφορα λειτουργικά συστήματα όπως π.χ. Linux, Windows, Solaris κλπ. Επιπλέον μας παρέχει την δυνατότητα να επιλέξουμε σε ποιο επεξεργαστή θα τρέξει το κάθε νήμα μας. Προήλθε από τη συγχώνευση δυο άλλων βιβλιοθηκών, της libtopology [7] του INRIA και της PLPA [8] του OpenMPI.

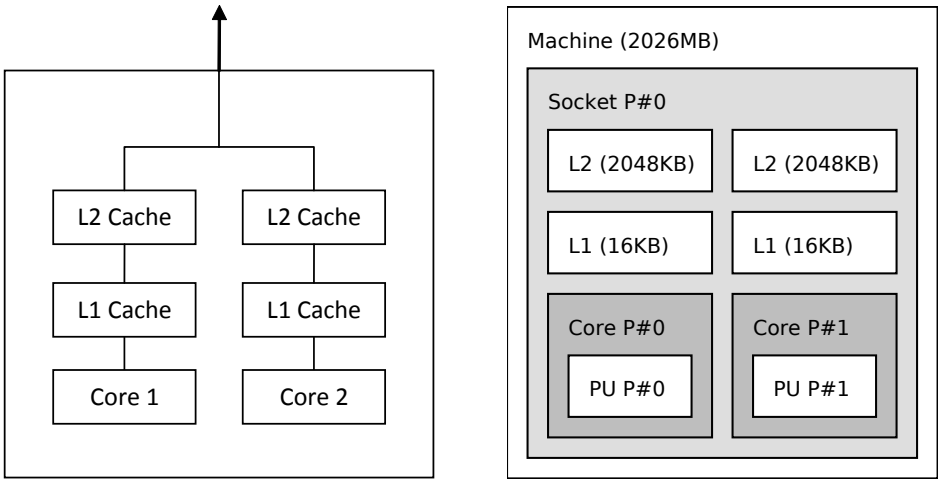
2.3.1 lstopo

Αρχικά αφού πρώτα εγκαταστήσαμε τη βιβλιοθήκη χρησιμοποιήσαμε το πρόγραμμα lstopo που συμπεριλαμβάνεται στην hwloc για να γνωρίσουμε την τοπολογία κάθε προγράμματος. Στο σχήμα 2.1 παρουσιάζουμε ένα παράδειγμα της εξόδου του lstopo. Όπως φαίνεται και από τα σχήματα το σύστημα αυτό είναι κοινόχρηστης μνήμης. Αριστερά (2.1α΄) είναι ένα διάγραμμα που φτιάξαμε για την τοπολογία του υπολογιστή και δεξιά (2.1β΄) είναι μια εικόνα που παράγει το lstopo. Ακολουθεί ένα παράδειγμα εξόδου που παράγει το lstopo στο ίδιο μηχάνημα όταν δεν έχει τη δυνατότητα να παράγει εικόνα ή όταν επιλέξουμε η έξοδος να είναι σε κείμενο και όχι σε εικόνα.

```
Machine (2026MB) + Socket L#0
  L2 L#0 (2048KB) + L1 L#0 (16KB) + Core L#0 + PU L#0 (P#0)
  L2 L#1 (2048KB) + L1 L#1 (16KB) + Core L#1 + PU L#1 (P#1)
```

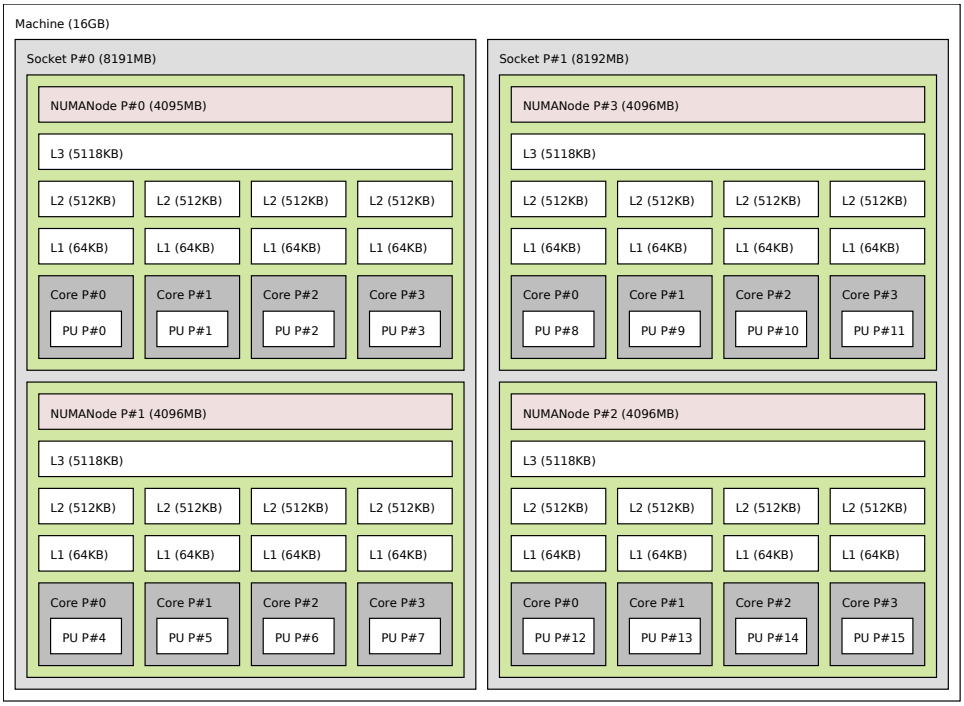
Και στο σχήμα 2.2 βλέπουμε ένα σύστημα NUMA. Ακολουθεί η έξοδος σε κείμενο του lstopo για αυτό το σύστημα.

```
Machine (16GB)
```



(α′) CPU παραδείγματος (β′) Έξοδος lstopo

Σχήμα 2.1: Παράδειγμα γραφικής εξόδου lstopo: σύστημα με κοινόχρηστη μνήμη



Σχήμα 2.2: Παράδειγμα γραφικής εξόδου lstopo: NUMA

```
Socket L#0 (8191MB)
  NUMANode L#0 (P#0 4095MB) + L3 L#0 (5118KB)
    L2 L#0 (512KB) + L1 L#0 (64KB) + Core L#0 + PU L#0 (P#0)
    L2 L#1 (512KB) + L1 L#1 (64KB) + Core L#1 + PU L#1 (P#1)
```

```
      L2 L#2 (512KB) + L1 L#2 (64KB) + Core L#2 + PU L#2 (P#2)
      L2 L#3 (512KB) + L1 L#3 (64KB) + Core L#3 + PU L#3 (P#3)
    NUMANode L#1 (P#1 4096MB) + L3 L#1 (5118KB)
      L2 L#4 (512KB) + L1 L#4 (64KB) + Core L#4 + PU L#4 (P#4)
      L2 L#5 (512KB) + L1 L#5 (64KB) + Core L#5 + PU L#5 (P#5)
      L2 L#6 (512KB) + L1 L#6 (64KB) + Core L#6 + PU L#6 (P#6)
      L2 L#7 (512KB) + L1 L#7 (64KB) + Core L#7 + PU L#7 (P#7)
  Socket L#1 (8192MB)
    NUMANode L#2 (P#3 4096MB) + L3 L#2 (5118KB)
      L2 L#8 (512KB) + L1 L#8 (64KB) + Core L#8 + PU L#8 (P#8)
      L2 L#9 (512KB) + L1 L#9 (64KB) + Core L#9 + PU L#9 (P#9)
      L2 L#10 (512KB) + L1 L#10 (64KB) + Core L#10 + PU L#10 (P#10)
      L2 L#11 (512KB) + L1 L#11 (64KB) + Core L#11 + PU L#11 (P#11)
    NUMANode L#3 (P#2 4096MB) + L3 L#3 (5118KB)
      L2 L#12 (512KB) + L1 L#12 (64KB) + Core L#12 + PU L#12 (P#12)
      L2 L#13 (512KB) + L1 L#13 (64KB) + Core L#13 + PU L#13 (P#13)
      L2 L#14 (512KB) + L1 L#14 (64KB) + Core L#14 + PU L#14 (P#14)
      L2 L#15 (512KB) + L1 L#15 (64KB) + Core L#15 + PU L#15 (P#15)
```

2.3.2 Αρχικοποίηση

Στη συνέχεια χρησιμοποιήσαμε τη βιβλιοθήκη στο stream, ώστε να μετρήσουμε το bandwidth κάθε συστήματος σε διάφορες αναθέσεις νημάτων ανά επεξεργαστές ανάλογως με την τοπολογία. Για να χρησιμοποιήσουμε τη βιβλιοθήκη πρέπει να αρχικοποιήσουμε μια μεταβλητή τύπου hwloc_topology_t. Η αρχικοποίηση γίνεται με την κλήση της hwloc_topology_init().

Αφού αρχικοποιήσουμε τη μεταβλητή μας έχουμε τη δυνατότητα να επιλέξουμε τι στοιχεία θέλουμε να ανιχνεύσουμε στην τοπολογία. Κάποια από τα στοιχεία που μπορούμε να ανιχνεύουμε είναι NUMA nodes, sockets, caches, cores, PUs. Το default είναι να ανιχνεύει όλα τα στοιχεία του συστήματος. Μπορούμε να επιλέξουμε να μην ανιχνεύει κάποια χρησιμοποιώντας hwloc_topology_ignore_type(). Τέλος για να μπορούμε να χρησιμοποιήσουμε τη μεταβλητή μας πρέπει να πούμε στην βιβλιοθήκη να συλλέξει πληροφορίες από το σύστημα καλώντας την hwloc_topology_load().

2.3.3 Πληροφορίες τοπολογίας

Εφόσον έχουμε ανιχνεύσει την τοπολογία του συστήματος μπορούμε να καλέσουμε διάφορες εντολές για να πάρουμε πληροφορίες γι' αυτήν όπως πχ να δούμε πόσα επίπεδα (βάθος depth) έχει η τοπολογία με την συνάρτηση hwloc_topology_get_depth(), να βρούμε τι είδους στοιχεία υπάρχουν στο x depth (hwloc_get_type_depth()), να βρούμε σε ποιο depth βρίσκονται τα στοιχεία που ψάχνουμε (hwloc_get_depth_type()), να βρούμε πόσα στοιχεία υπάρχουν στο x depth (hwloc_get_nbobjs_by_depth()) ή να βρούμε πόσα στοιχεία από το συγκεκριμένο είδος υπάρχουν (hwloc_get_nbobjs_by_type()) [9].

2.3.4 Objects

Οι βασικές μεταβλητές που χρησιμοποιεί η hwloc είναι τύπου hwloc_obj [10]. Είναι κόμβοι ενός δέντρου με ρίζα το σύστημα και φύλλα τους επεξεργαστές. Κάθε struct τύπου hwloc_obj περιέχει πεδία με διάφορες πληροφορίες όπως το είδος του κόμβου στον οποίο ανήκει (type), ένα δείκτη στον πατέρα του (parent), έναν πίνακα με δείκτες στα παιδιά του (children) καθώς και τον αριθμό των παιδιών (arity), ένα δείκτη στον επόμενο αδερφό του (next_sibling) και ένα στον προηγούμενο (prev_sibling), δείκτες στον επόμενο και στον προηγούμενο ξάδερφο (next_cousin, prev_cousin), ένα κείμενο που περιγράφει τον κόμβο (name) και ένα πεδίο τύπου hwloc_cpuset_t (cpuset) το οποίο περιέχει τους επεξεργαστές που βρίσκονται κάτω από αυτό τον κόμβο. Υπάρχουν δυο συναρτήσεις που επιστρέφουν το hwloc_obj ενός κόμβου, η hwloc_get_obj_by_depth() που μας επιστρέφει το ν-οστό κόμβο σε ένα δεδομένο depth και η hwloc_get_obj_by_type() που μας επιστρέφει το ν-οστό κόμβο δεδομένου κάποιου είδος κόμβου.

2.3.5 Binding

Για την ανάθεση νημάτων στους επεξεργαστές (binding) η hwloc παρέχει τρεις συναρτήσεις. Η πρώτη hwloc_set_cpubind() κάνει bind το τρέχων νήμα ή διεργασία, η hwloc_set_proc_cpubind() κάνει bind τη διεργασία που προσδιορίζουμε μέσω της hwloc_pid_t pid, και τέλος η hwloc_set_thread_cpubind κάνει bind το νήμα hwloc_thread_t tid.

Επειδή η hwloc χρησιμοποιεί κλήσεις του συστήματος για την υλοποίηση των συναρτήσεων της κάποιες εντολές όπως αυτές για το binding ενδέχεται να μη λειτουργούν σε κάποια λειτουργικά, ή αν έχουν πρόβλημα στο λειτουργικό να έχουν πρόβλημα και στη βιβλιοθήκη (ανατρέξτε στην ενότητα Paraguay 5.1.3).

2.3.6 Άλλες δυνατότητες της hwloc

Πέραν από την εύρεση των συνδέσεων μεταξύ επεξεργαστών, κρυφών μνημών και μνήμης, πρόσφατες εκδόσεις της hwloc έχουν την δυνατότητα εύρεσης και απεικόνισης θυρών PCI που υπάρχουν στο σύστημα μας καθώς και των συσκευών που βρίσκονται σε αυτές. Άλλη μια λειτουργία που προστέθηκε σε νεώτερες εκδόσεις της βιβλιοθήκης είναι η διαχείριση της μνήμης σε συστήματα NUMA (περισσότερα γι' αυτά στο κεφάλαιο 4) με συναρτήσεις αντίστοιχες αυτών που μας παρέχει το NUMA API. Δυστυχώς όταν προστέθηκε αυτή η δυνατότητα είχαμε ήδη τελειώσει τα πειράματα μας χρησιμοποιώντας τη libnuma.

2.4 Πώς επηρεάζει η τοπολογία τα προγράμματα μας

Όπως αναφέραμε στην εισαγωγή, η τοπολογία και η αρχιτεκτονική ενός συστήματος επηρεάζει τις επιδόσεις των εφαρμογών που εκτελούνται σε αυτό. Αν θέλουμε ένα πρόγραμμα να εκμεταλλεύεται στο έπακρον τις δυνατότητες του συστήματος και να έχει βέλτιστες επιδόσεις, θα πρέπει να λαμβάνει υπόψιν του την τοπολογία του συστήματος και να προσαρμόζεται κατάλληλα.

Στα σειριακά συστήματα όπως είπαμε και στο κεφάλαιο 2.1 για να βελτιώσουμε τις επιδόσεις του προγράμματος μας συνήθως αρκεί να λάβουμε υπόψιν μας την κρυφή μνήμη. Πρέπει να φροντίζουμε ώστε τα δεδομένα που επεξεργάζονται να είναι τοποθετημένα κοντά μεταξύ τους στη μνήμη, ώστε να εξαντλούνται όσο είναι δυνατόν κάθε φορά οι αναγκαίες προσπελάσεις μιας θέσης μνήμης, να είναι δηλαδή ομαδοποιημένες οι εντολές που προσπελαύνουν γειτονικές θέσεις μνήμης, ώστε να εκμεταλλεύεται η κρυφή μνήμη (cache) στο έπακρον.

Στα παράλληλα συστήματα λόγω των μεγαλύτερου εύρους διαφοροποιήσεων που συναντάμε η συγγραφή αποδοτικών προγραμμάτων δυσκολεύει. Επιπλέον το γεγονός ότι ένα πρόγραμμα δουλεύει αποδοτικά σε ένα σύστημα δε σημαίνει ότι θα είναι εξίσου καλό και σε ένα άλλο γεγονός που αυξάνει την πολυπλοκότητα προγραμματισμού τους και μειώνει τη φορητότητα του κώδικα μας (δηλαδή τη δυνατότητα του να εκτελεστεί σε ένα άλλο σύστημα με τον ελάχιστο δυνατό αριθμό τροποποιήσεων).

Για παράδειγμα ένα πρόγραμμα γραμμένο σε μοντέλο κοινού χώρου διευθύνσεων δεν θα δουλεύει σε ένα κατανεμημένο σύστημα. Ακόμα και αν μπορούμε να εκτελέσουμε το πρόγραμμα μας σε κάποιο σύστημα μπορεί να μην έχουμε ικανοποιητικές επιδόσεις, όπως π.χ. σε συστήματα NUMA όπου η τοποθέτηση των νημάτων και η αρχικοποίηση των δεδομένων επηρεάζουν την ταχύτητα επικοινωνίας της μνήμης (περισσότερα στα κεφάλαια 4 και 5.2.4).

Κεφάλαιο 3

Stream

Το stream [2] είναι ένα απλό πρόγραμμα που μετράει την ταχύτητα επικοινωνίας του επεξεργαστή με την κύρια μνήμη (bandwidth). Προκειμένου να μετρήσει το bandwidth εκτελεί τέσσερα πειράματα (Copy, Scale, Add, Triad), NTIMES φορές το καθένα, ώστε να είναι σίγουρο ότι έχει τα βέλτιστα αποτελέσματα, και επιστρέφει την ταχύτητα σε MB/s, το μέσο χρόνο, τον ελάχιστο χρόνο και το μέγιστο χρόνο για κάθε πείραμα.

3.1 Τι μετράει

Κάθε πείραμα αποτελείται από μια απλή εντολή η οποία αντιγράφει δεδομένα από μια θέση της μνήμης σε μια άλλη αφού κάνει κάποια απλή πράξη. Εδώ να σημειώσουμε ότι ο όγκος των δεδομένων είναι αρκετά μεγάλος ώστε να μην χωράει σε μια κρυφή μνήμη. Ο κύριος βρόγχος του παρουσιάζεται παρακάτω. Το Copy (γραμμές 4-7) αντιγράφει σε έναν πίνακα μεγέθους N τα στοιχεία ενός άλλου (c[j] = a[j]). Το Scale (γραμμές 9-12) αποθηκεύει σε έναν πίνακα το γινόμενο ενός σταθερού αριθμού με των στοιχείων ενός πίνακα (b[j] = scalar*c[j]). Το Add (γραμμές 14-17) αποθηκεύει σε έναν πίνακα το άθροισμα 2 άλλων (c[j] = a[j]+b[j]) και τέλος το Triad (γραμμές 19-22) αποθηκεύει σε έναν πίνακα το άθροισμα ενός πίνακα με το πολλαπλάσιο ενός άλλου (a[j] = b[j]+scalar*c[j]).

```
1  scalar = 3.0;
2  for (k=0; k<NTIMES; k++)
3  {
4    // Copy
5    times[0][k] = mysecond();
6    for (j=0; j<N; j++)
7      c[j] = a[j];
8    times[0][k] = mysecond() - times[0][k];
9
10   // Scale
11   times[1][k] = mysecond();
12   for (j=0; j<N; j++)
```

```
13      b[j] = scalar*c[j];
14    times[1][k] = mysecond() - times[1][k];
15
16    // Add
17    times[2][k] = mysecond();
18    for (j=0; j<N; j++)
19      c[j] = a[j]+b[j];
20    times[2][k] = mysecond() - times[2][k];
21
22    // Triad
23    times[3][k] = mysecond();
24    for (j=0; j<N; j++)
25      a[j] = b[j]+scalar*c[j];
26    times[3][k] = mysecond() - times[3][k];
27  }
```

3.2 Πώς δουλεύει

Για να μετρήσει το bandwidth, αρχικά μετράει το χρόνο που κάνει κάθε πείραμα, κρατώντας σε μια μεταβλητή τον χρόνο πριν να ξεκινήσει το πείραμα και στη συνέχεια τον αφαιρεί από το χρόνο στο τέλος του πειράματος. Στο τέλος του προγράμματος, όταν πλέον έχουν τρέξει όλα τα πειράματα από NTIMES το καθένα, βρίσκει τον ελάχιστο και το μέγιστο χρόνο που έκανε κάθε επανάληψη ενός πειράματος και υπολογίζει το μέσο χρόνο, αγνοώντας την πρώτη επανάληψη κάθε πειράματος λόγω του ότι μπορεί να είναι ασυνεπής εξαιτίας των αρχικοποιήσεων. Για να υπολογίσει το bandwidth διαιρεί το συνολικό μέγεθος των πινάκων που χρησιμοποιήθηκαν στο κάθε πείραμα (MB) με τον ελάχιστο χρόνο που έκανε για το συγκεκριμένο πείραμα (s), βρίσκοντας έτσι το μέγιστο ρυθμό επικοινωνίας του επεξεργαστή με τη μνήμη.

3.3 Γιατί το χρησιμοποιήσαμε

Αυτό που μας ενδιέφερε ήταν να μελετήσουμε το bandwidth που έχει ένα πρόγραμμα όταν χρησιμοποιεί περισσότερους από έναν επεξεργαστή, και κατά πόσο αυτό διαφέρει από το bandwidth που θα είχε αν έτρεχε σε μόνο έναν επεξεργαστή. Για τον σκοπό αυτό μετατρέψαμε το stream, ώστε να χρησιμοποιεί 2,4,8 νήματα (threads). Επιπλέον χρειαζόμασταν έναν τρόπο να γνωρίζουμε την αρχιτεκτονική του εκάστοτε μηχανήματος καθώς και να μπορούμε να επιλέξουμε σε ποιον επεξεργαστή θα τρέχει το κάθε νήμα οπότε για τον σκοπό αυτό χρησιμοποιήσαμε την βιβλιοθήκη Hwloc (2.3). Στη συνέχεια κάναμε μετρήσεις σε διάφορα συστήματα, κάνοντας πειράματα ανάλογα με την αρχιτεκτονική καθενός.

3.4 Αλλαγές

Όπως είπαμε και στην εισαγωγή το Stream διαθέτει και οδηγίες OpenMP στον κώδικα του για την εκτέλεση σε πολυπύρηνα συστήματα. Επειδή για τα πειράματα μας χρειαζόμασταν τον έλεγχο του πόσα νήματα θα εκτελούνται και που, δεν χρησιμοποιήσαμε OpenMP. Το παραλληλοποιήσαμε με την χρήση νημάτων τα οποία με την βοήθεια της βιβλιοθήκης hwloc (2.3) κάνουμε bind σε συγκεκριμένους επεξεργαστές ανάλογα με τι θέλουμε να μετρήσουμε κάθε φορά.

Αφού πήραμε τον πηγαίο κώδικα της τελευταίας έκδοσης του stream [3] (v5.9) και αφαιρέσαμε ότι αναφορά είχε σε OpenMP καθώς και όποια συνάρτηση δε μας χρησίμευε κάπου, μετατρέψαμε τη mysecond ώστε να δουλεύει και σε Linux και σε Windows. Στη συνέχεια μοιράσαμε την δουλειά που κάνει κάθε πείραμα στα νήματα, δίνοντας σε κάθε νήμα N/THREADN κελιά κάθε πίνακα, όπου THREADN είναι ο αριθμός των νημάτων, τα οποία βάλαμε να περιμένουν σε ένα σημείο συγχρονισμού (barrier) πριν και μετά από κάθε πείραμα. Συγχρόνως στη main κοιτάμε την ώρα, καλούμε το barrier που έχουμε πριν από το κάθε πείραμα, καλούμε το barrier που βρίσκεται μετά το πείραμα, και όταν επιστρέφει από το barrier, δηλαδή όταν είχαν τελειώσει όλα τα νήματα με την εκτέλεση του πειράματος, υπολογίζουμε το χρόνο που διήρκεσε. Με αυτό τον τρόπο δυστυχώς στο χρόνο που έχουμε μετρήσει, υπάρχει και ο χρόνος εκτέλεσης των barriers, ο οποίος είναι αρκετά μικρός ώστε να μην επηρεάζει πολύ τα αποτελέσματα μας.

Παρακάτω δίνεται ο κύριος κορμός του κώδικα που τρέχουν τα νήματα, καθώς και του κώδικα που τρέχει η main:

```
1  /*************************************************************\
2   *                    Βασικός κώδικας των νημάτων             *
3  \*************************************************************/
4  int loop_start = threadID * (N/THREADN);
5  int loop_end = (threadID + 1 == THREADN) ? N : loop_start +
6                                                      N/THREADN;
7
8  /* ... */
9
10 // Wait till main has finished setting the affinity and
11 // then initialize the arrays
12 pthread_barrier_wait(&init_start);
13 for (j=loop_start; j<loop_end; j++) {
14   a[j] = 1.0;
15   b[j] = 2.0;
16   c[j] = 0.0;
17 }
18 pthread_barrier_wait(&init_end);
19 /* ... */
20 scalar = 3.0;
21 for (k=0; k<NTIMES; k++) {
22
23   // Copy
```

```
24    pthread_barrier_wait(&start_copy);
25    for (j=loop_start; j<loop_end; j++)
26      c[j] = a[j];
27    pthread_barrier_wait(&end_copy);
28
29    /* ... */
30  }
31
32
33  /*****************************************************************\
34   *                      Βασικός κώδικας της main                 *
35  \*****************************************************************/
36  for (i = 0; i < THREADN; i++) {
37    pthread_create(&pid[i], NULL, threadCode, (void *)i);
38  }
39
40  // Set threads' affinity
41  set_affinity(pid, argc, argv);
42
43  // Tell threads that binding is over and they can start
44  // initializations
45  pthread_barrier_wait(&init_start);
46
47  /* ... */
48
49  for (k=0; k<NTIMES; k++) {
50    // Copy
51    times[0][k] = mysecond();
52    pthread_barrier_wait(&start_copy);
53    pthread_barrier_wait(&end_copy);
54    times[0][k] = mysecond() - times[0][k];
55
56    /* ... */
57  }
58
59  /* ... */
60
61  for (k=1; k<NTIMES; k++) /* note -- skip first iteration */
62  {
63  for (j=0; j<4; j++)
64    {
65    avgtime[j] = avgtime[j] + times[j][k];
66    mintime[j] = MIN(mintime[j], times[j][k]);
67    maxtime[j] = MAX(maxtime[j], times[j][k]);
68    }
69  }
70
71  printf("Function      Rate (MB/s)   Avg time     Min time     Max time\n");
72  for (j=0; j<4; j++) {
73    avgtime[j] = avgtime[j]/(double)(NTIMES-1);
74
75    printf("%s%11.4f  %11.4f  %11.4f  %11.4f\n", label[j],
76    1.0E-06 * bytes[j]/mintime[j],
77    avgtime[j],
```

```
78    mintime[j],
79    maxtime[j]);
80  }
```

Κεφάλαιο 4

NUMA & NUMA API

Καθώς συνέχεια αυξάνει η ταχύτητα των επεξεργαστών, σπαταλούν όλο και περισσότερο χρόνο περιμένοντας δεδομένα από τη μνήμη που είναι αρκετά πιο αργή. Στις μέρες μας που αυξάνεται και ο αριθμός των επεξεργαστών, το πρόβλημα γίνεται ακόμα χειρότερο καθώς πολλοί επεξεργαστές προσπαθούν να προσπελάσουν συγχρόνως τη μνήμη, και άρα πρέπει να περιμένουν να τελειώσουν οι προηγούμενες προσπελάσεις που τυχόν έκαναν οι υπόλοιποι επεξεργαστές, πριν να έρθει η σειρά τους και να λάβουν τα δεδομένα που χρειάζονται.

Για να αποφύγουν αυτό ακριβώς το πρόβλημα, και καθώς μόνο ένας επεξεργαστής μπορεί να έχει πρόσβαση στη μνήμη κάθε φορά, κάποια συστήματα έχουν ξεχωριστές ιδιωτικές μνήμες ανά επεξεργαστή ή ομάδα επεξεργαστών. Επειδή όμως πολλές φορές τα δεδομένα που επεξεργάζεται ένας επεξεργαστής μπορεί να χρησιμοποιούνται και από άλλους, υπάρχει επιπλέον υλικό (hardware) που επιτρέπει την προσπέλαση μιας απομακρυσμένης μνήμης.

Τα συστήματα που ακολουθούν αυτή τη σχεδίαση ονομάζονται Non-Uniform Memory Access [20] (NUMA ανομοιόμορφη προσπέλαση μνήμης). Σε αυτά η μνήμη και οι επεξεργαστές μοιράζονται σε κόμβους (nodes). Οι επεξεργαστές έχουν πρόσβαση στη μνήμη που υπάρχει στους άλλους κόμβους, αλλά η πρόσβαση της τοπικής μνήμης (δηλαδή αυτής που βρίσκεται στον ίδιο κόμβο με τον επεξεργαστή) είναι πιο γρήγορη απ' ότι η πρόσβαση σε μνήμη που βρίσκεται σε κάποιον άλλο κόμβο, σε αντίθεση με τα κλασσικά πολυπύρηνα συστήματα (SMP symmetric multiprocessing), όπου οι επεξεργαστές και η μνήμη βρίσκονται στον ίδιο δίαυλο και η προσπέλαση της μνήμης ήταν ομοιόμορφη (UMA).

Το κύριο πλεονέκτημα αυτής της αρχιτεκτονικής σε σχέση με τα μηχανήματα UMA, όπου όλοι οι επεξεργαστές είναι συνδεδεμένοι με μια κύρια μνήμη είναι η επεκτασιμότητα της (scalability), η ικανότητα της δηλαδή να υποστηρίζει όλο και περισσότερους επεξεργαστές με μικρή απώλεια αποδόσεων. Το μειονέκτημα της είναι ότι επιβαρύνεται το έργο του προγραμματιστή καθώς τα προγράμματα που τρέχουν σε αυτήν θα πρέπει να χειρίζονται κατάλληλα τα νήματα τους και τη μνήμη τους ώστε κάθε νήμα να προσπελαύνει την τοπική μνήμη όσο αυτό είναι εφικτό, για να αποφεύγεται η συμφόρηση σε ένα κόμβο.

Στο παρελθόν συναντούσαμε NUMA μόνο σε προχωρημένα συστήματα όπως για παράδειγμα το SGI Origin 2000 (έτος κυκλοφορίας 1996), το οποίο δεχόταν μέχρι και 128 επεξεργαστές με ιδιωτικές μνήμες, που ήταν συνδεδεμένοι μέσω ενός δικτύου υπερκύβου και διέθετε κοινό χώρο διευθύνσεων με συνεκτικές κρυφές μνήμες (cache-coherent NUMA). Στις μέρες μας συναντάται όλο και πιο συχνά και είναι συνηθισμένη αρχιτεκτονική σε πολυπύρηνους επεξεργαστές με πάνω από 8 πυρήνες, όπως για παράδειγμα οι επεξεργαστές Opteron της AMD καθώς και πρόσφατα μοντέλα Itanium και Xeon της Intel. Στα συστήματα αυτά ο συντελεστής NUMA (NUMA factor) προσδιορίζει την καθυστέρηση που υφίσταται ένας επεξεργαστής κατά την προσπέλαση απομακρυσμένης μνήμης.

Το NUMA API [21, 22] είναι ένα σύνολο εργαλείων για τα linux που μας επιτρέπει να διαχειριστούμε την εκχώρηση της μνήμης σε NUMA συστήματα. Αποτελείται από μια βιβλιοθήκη (libnuma) μέρος της οποίας αποτελεί πλέον κομμάτι του πυρήνα των linux, κάποιες κλήσεις συστήματος καθώς και κάποια εργαλεία (numactl, numastat και numademo).

4.1 Libnuma

Η libnuma είναι μια βιβλιοθήκη επιπέδου χρήστη των linux, που επιτρέπει στον χρήστη να διαχειριστεί τη μνήμη του συστήματος σε μηχανήματα NUMA και αποτελεί μέρος του NUMA API. Χωρίς τη χρήση της βιβλιοθήκης τα linux κατανέμουν τη μνήμη ενός προγράμματος στον κόμβο που βρίσκεται ο επεξεργαστής που την προσπελαύνει για πρώτη φορά. Σε περιπτώσεις που θέλουμε να έχουμε τη μικρότερη καθυστέρηση (latency) στην επικοινωνία επεξεργαστή-μνήμης, η πολιτική αυτή αποτελεί τη βέλτιστη λύση. Υπάρχουν όμως περιπτώσεις που αυτή η πολιτική μπορεί να μην έχει καλές επιδόσεις. Για παράδειγμα, σε ένα πρόγραμμα που οι αρχικοποιήσεις της μνήμης γίνονται όλες από ένα κεντρικό νήμα, όλη η μνήμη του προγράμματος θα εκχωρηθεί σε έναν κόμβο, με αποτέλεσμα νήματα που εκτελούνται σε άλλους κόμβους να έχουν μεγαλύτερο latency. Ακόμα χειρότερα, επειδή όλα τα νήματα θα προσπελαύνουν την ίδια μνήμη, αυτή θα γίνεται σημείο συμφόρησης, με αποτέλεσμα είτε βρίσκονται στον ίδιο κόμβο με το κεντρικό νήμα είτε όχι, να καθυστερούν κατά την επικοινωνία τους με την μνήμη. Σε αυτή την περίπτωση θα μπορούσαμε να χρησιμοποιήσουμε τη libnuma ώστε να κατανέμουμε τη μνήμη στους κόμβους, λαμβάνοντας υπόψιν μας σε ποιο κόμβο θα βρίσκεται το νήμα που θα προσπελαύνει το εκάστοτε κομμάτι μνήμης.

Ένας άλλος λόγος για τον οποίο μπορούμε να χρησιμοποιήσουμε τη βιβλιοθήκη είναι στην περίπτωση που μας ενδιαφέρει περισσότερο η ταχύτητα μεταφοράς (bandwitdth) απ' ότι η ταχύτητα προσπέλασης. Σε αυτή την περίπτωση μπορούμε, χρησιμοποιώντας τη libnuma, να μοιράσουμε τη μνήμη στους κόμβους, έτσι ώστε όταν ένα νήμα προσπελαύνει ένα αρκετά μεγάλο κομμάτι μνήμης να λαμβάνει κομμάτια από περισσότερες από μία μνήμες και το διαθέσιμο

bandwidth να είναι όσο το σύνολο των bandwidth προς κάθε μια από τις μνήμες.

4.1.1 Χρήση της βιβλιοθήκης

Για να χρησιμοποιήσουμε τη libnuma σε ένα πρόγραμμα μας, πρέπει καταρχάς να κάνουμε include τη βιβλιοθήκη numa.h και όταν κάνουμε compile το πρόγραμμα μας να χρησιμοποιήσουμε το όρισμα -lnuma.

Αρχικά στο πρόγραμμα μας πρέπει να καλέσουμε τη συνάρτηση numa_available(). Αν μας επιστρέψει αρνητικό αριθμό τότε δεν μπορούμε να χρησιμοποιήσουμε τη libnuma στο σύστημα μας. Στη συνέχεια μπορούμε να μάθουμε τον αριθμό των κόμβων στο σύστημα μας καλώντας τη numa_max_node(), να αλλάξουμε την πολιτική (policy) που θα ακολουθείται κατά την εκχώρηση της μνήμης κάθε νήματος (4.1.2), να εκχωρήσουμε μνήμη χρησιμοποιώντας κάποια συγκεκριμένη πολιτική (4.1.3), να αλλάξουμε την πολιτική με την οποία έχει εκχωρηθεί ένα κομμάτι μνήμης (4.1.4) (χρησιμοποιείται κυρίως σε περιπτώσεις που έχουμε εκχωρήσει κάποια κοινόχρηστη μνήμη μέσω mmap() ή shmat()) ή τέλος να επιλέξουμε σε ποιους κόμβους θα εκτελείται κάποιο νήμα (4.1.5). Οι πολιτικές που είναι διαθέσιμες είναι:

- local allocation, όπου η μνήμη εκχωρείται στον κόμβο στον οποίο εκτελείται το νήμα
- preferred node allocation, όπου η εκχώρηση θα γίνεται στον επιλεγμένο κόμβο εφόσον υπάρχει αρκετή μνήμη ελεύθερη, αλλιώς γίνεται σε κάποιον από τους άλλους κόμβους
- page interleaving, όπου η εκχώρηση γίνεται με Round-Robin στους επιλεγμένους κόμβους για κάθε σελίδα.
- εκχώρηση μόνο σε συγκεκριμένους κόμβους

4.1.2 Αλλαγή πολιτικής

Η αλλαγή της πολιτικής που ακολουθεί κάποιο νήμα δεν επηρεάζει τα άλλα νήματα που τυχόν υπάρχουν στην διεργασία και κληρονομείται στα νήματα που δημιουργούνται από αυτό.

- numa_set_preferred(): Χρήση preferred node allocation. Παίρνει σαν ορίσματα τον αύξοντα αριθμό του κόμβου που θέλουμε να επιλέξουμε.
- numa_set_interleave_mask(): Χρήση page interleaving στους κόμβους που βρίσκονται σε μια μεταβλητή τύπου bitmask (η bitmask είναι μια μεταβλητή τύπου bitmap, δηλαδή μια μεταβλητή που αποτελείται από άσσους στη θέση των επιλεγμένων κόμβων).
- numa_set_membind(): Επιλογή συγκεκριμένων κόμβων στους οποίους επιτρέπεται να γίνεται εκχώρηση μνήμης από το νήμα μας.

- numa_set_localalloc(): Επαναφορά της προεπιλεγμένης πολιτικής, όπου κάθε εκχώρηση μνήμης γίνεται στην τοπική μνήμη (δηλαδή στη μνήμη που βρίσκεται στον κόμβο που εκτελείται το νήμα κατά την εκχώρηση).

4.1.3 Εκχώρηση μνήμης

Μπορούμε να εκχωρήσουμε μνήμη ακολουθώντας κάποια συγκεκριμένη πολιτική χωρίς να αλλάξουμε την πολιτική που θα ακολουθεί το νήμα για άλλες εκχωρήσεις. Οι διαθέσιμες συναρτήσεις είναι:

- numa_alloc_onnode(): Εκχώρηση μνήμης στον δοθέν κόμβο.
- numa_alloc_local(): Εκχώρηση μνήμης στον τοπικό κόμβο.
- numa_alloc_interleaved(): Εκχώρηση μνήμης χρησιμοποιώντας page interleaving.
- numa_alloc_interleaved_subset(): Όπως η numa_alloc_interleaved() επιτρέποντας μας να επιλέξουμε σε ποιους κόμβους θα γίνει το page interleaving.
- numa_alloc(): Εκχώρηση μνήμης σύμφωνα με την πολιτική που ακολουθεί το νήμα μας. Σημείωση: οι συναρτήσεις αυτές είναι αργές σε σχέση με τη malloc. Επιπλέον ότι μνήμη καταχωρείται με τη χρήση αυτών των συναρτήσεων, πρέπει να "ελευθερώνεται" με τη χρήση της numa_free().

4.1.4 Αλλαγή πολιτικής ήδη εκχωρημένης μνήμης

Υπάρχουν συναρτήσεις με τις οποίες μπορούμε να αλλάξουμε την πολιτική με την οποία έχει εκχωρηθεί μνήμη. Αυτό είναι χρήσιμο κυρίως όταν έχουμε εκχωρήσει κοινόχρηστη μνήμη με χρήση είτε της mmap() είτε της shmap() (σε διαφορετική περίπτωση είναι προτιμότερο είτε να αλλάζουμε την πολιτική εκχώρησης και να χρησιμοποιούμε malloc είτε να χρησιμοποιούμε τις numa_alloc*() που περιγράψαμε στην προηγούμενη παράγραφο). Προϋπόθεση είναι να μην έχουμε προσπελάσει τη μνήμη που έχουμε εκχωρήσει ακόμα. Οι συναρτήσεις είναι:

- numa_interleave_memory(): Κατανομή της μνήμης στους κόμβους που ορίζει μια μεταβλητή τύπου bitmask.
- numa_tonode_memory(): Τοποθέτηση της μνήμης στον επιλεγμένο κόμβο.
- numa_tonodemask_memory(): Τοποθέτηση της μνήμης στους κόμβους που υποδεικνύει μια μεταβλητή τύπου bitmask.
- numa_setlocal_memory() : Τοποθέτηση της μνήμης στον τοπικό κόμβο.
- numa_police_memory(): Τοποθέτηση της μνήμης σύμφωνα με την πολιτική που εφαρμόζεται στο νήμα μας.

4.1.5 Επιλογή κόμβου εκτέλεσης

Εκτός από τις συναρτήσεις που ασχολούνται με τη μνήμη, η libnuma διαθέτει επιπλέον και κάποιες συναρτήσεις για τη διαχείριση και τοποθέτηση νημάτων σε κόμβους και επεξεργαστές.

- numa_node_of_cpu(): Επιστρέφει τον κόμβο στον οποίο ανήκει ο επιλεγμένος επεξεργαστής (CPU).
- numa_node_to_cpus(): Επιστρέφει ένα bitmask με τους επεξεργαστές που ανήκουν στον επιλεγμένο κόμβο.
- numa_run_on_node() και numa_run_on_node_mask(): Μας επιτρέπουν να επιλέξουμε έναν ή περισσότερους αντίστοιχα κόμβους στους οποίους θα επιτρέπεται να τρέχει το νήμα μας.
- numa_bind(): Συνδυάζει τη numa_run_on_node_mask() με τη numa_set_membind(), τοποθετώντας το τρέχον νήμα στους επιλεγμένους κόμβους, και θέτοντας σαν πολιτική του να χρησιμοποιεί μνήμη από αυτούς τους κόμβους.

Επιπλέον είναι διαθέσιμες και οι numa_sched_getaffinity και numa_sched_setaffinity που λειτουργούν όπως και οι sched_getaffinnity και sched_setaffinity αντίστοιχα, αλλά χρησιμοποιούν τα bitmasks της libnuma αντί των CPU sets.

4.1.6 Άλλες συναρτήσεις

Τέλος αξίζει να αναφέρουμε τη numa_distance() που μας επιστρέφει την "απόσταση" μεταξύ δύο κόμβων καθώς και το γεγονός ότι η βιβλιοθήκη διαθέτει και συναρτήσεις για τη διαχείριση των bitmask που χρησιμοποιεί (numa_bitmask_*()).

4.1.7 Παράδειγμα

Στην ενότητα αυτή ακολουθεί ένα παράδειγμα του NUMA API με αναλυτικά σχόλια για την κάθε κλήση.

```
1  #include <numa.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5
6  void *numaExampleThread(void *arg);
7
8  int main() {
9    int i, *pointer1, *pointer2, *pointer3;
10   pthread_t tid;
11
```

```
12    // Αρχικά ελέγχουμε τη διαθεσιμότητα της βιβλιοθήκης
13    if (numa_available() == -1) {
14      printf("Can't use numa functions");
15      exit(1);
16    }
17
18    // Για να δούμε πόσοι κόμβοι είναι διαθέσιμοι στο σύστημα χρησι-
19    // μοποιούμε τη numa_max_node(). Προσοχή: η numa_max_node() μας
20    // επιστρέφει τον αριθμό του μέγιστου κόμβου. Επειδή η αρίθμηση
21    // ξεκινάει από το μηδέν ο αριθμός των κόμβων στο σύστημα θα εί-
22    // ναι numa_max_node() + 1.
23    unsigned int nNodes = numa_max_node() + 1;
24    printf("Number of nodes in the system: %d\n",nNodes);
25
26    // Η κλήση της malloc χρησιμοποιεί το policy του προγράμματος.
27    // Εφόσον δεν το έχουμε αλλάξει θα χρησιμοποιηθεί το προεπιλεγμε-
28    // γμένο policy, το οποίο κάνει την εκχώρηση στη μνήμη που αντι-
29    // στοιχεί στον επεξεργαστή στον οποίο εκτελείται το νήμα.
30    pointer1 = (int *) malloc(1000 * sizeof(int));
31
32
33    // Δημιουργία ενός bitmask
34    // Αρχικά πρέπει να δηλώσουμε μια μεταβλητή
35    struct bitmask *mask;
36    // Στη συνέχεια την αρχικοποιούμε
37    mask = numa_allocate_nodemask();
38    // Και τέλος της λέμε ποια nodes θέλουμε
39    for (i = 0; i <= numa_max_node(); i++) {
40      numa_bitmask_setbit(mask, i);
41    }
42    // Για να χρησιμοποιήσουμε όλα τα nodes, θα μπορούσαμε να χρησι-
43    // μοποιήσουμε τη numa_all_nodes_ptr που υπάρχει ήδη δηλωμένη
44    // στη βιβλιοθήκης ή της numa_bitmask_setall, αντί του παραπάνω
45    // loop.
46
47
48    // Τώρα μπορούμε να χρησιμοποιήσουμε το bitmask μας σε οποιαδήπο-
49    // τε συνάρτηση δέχεται bitmasks ως όρισμα. Στο συγκεκριμένο πα-
50    // ράδειγμα αλλάζουμε την πολιτική.
51    numa_set_interleave_mask(mask);
52
53    // Αυτή η κλήση της malloc θα χρησιμοποιήσει memory interleaving
54    // κατά την εκχώρηση.
55    pointer2 = (int *) malloc(1000 * sizeof(int));
56
57
58    // Το νήμα που θα δημιουργηθεί από την pthread_create θα ακολου-
59    // θεί το ίδιο policy με αυτό το νήμα
60    pthread_create(&tid, NULL, numaExampleThread, NULL);
61
62
63    // Διατίθενται επιπλέον και κλήσεις για την εκχώρηση μνήμης με
64    // κάποιο συγκεκριμένο policy. Στο παράδειγμα που ακολουθεί εκχω-
65    // ρούμε μνήμη στο πρώτο node. Αυτές οι κλήσεις είναι αργές σε
```

```
66    // σχέση με την malloc.
67    pointer3 = (int *) numa_alloc_onnode(1000 * sizeof(int), 0);
68
69
70    // Η βιβλιοθήκη διαθέτει και κλήσεις για να αλλάξουμε ήδη εκχωρη-
71    // μένη μνήμη η οποία δεν έχει προσπελαστεί ακόμα. Αυτές οι κλή-
72    // σεις χρησιμοποιούνται σε συνδυασμό με τις mmap και shmat
73    //numa_interleave_memory(void  *start, size_t size,
74    //struct bitmask *nodemask);
75
76
77    // Απελευθέρωση της μνήμης που χρησιμοποιεί το bitmask
78    numa_free_nodemask(mask);
79    numa_free(pointer3, 1000 * sizeof(int));
80
81    pthread_join(tid, NULL);
82    free(pointer1);
83    free(pointer2);
84
85    return 0;
86  }
87
88  void *numaExampleThread(void *arg) {
89    int *pointer1, *pointer2;
90
91    // Εδώ η κλήση της malloc θα γίνει με memory interleaving, γιατί
92    // ο πατέρας αυτού του νήματος είχε σαν policy το memory inter-
93    // leaving.
94    pointer1 = (int *) malloc(1000 * sizeof(int));
95
96    // Αλλάζουμε το policy σε local allocation. Η αλλαγή ισχύει μόνο
97    // για αυτό το νήμα (και για όσα θα δημιουργήσει) και δε θα επη-
98    // ρεάσει το policy που ακολουθεί το κύριο νήμα ή κάποιο άλλο νή-
99    // μα που ήδη τρέχει.
100   numa_set_localalloc();
101
102   // Στον κώδικα που ακολουθεί τοποθετούμε το τρέχον νήμα στους ε-
103   // πεξεργαστές που βρίσκονται στο δεύτερο node του συστήματος.
104   // Θα μπορούσαμε να χρησιμοποιήσουμε τη numa_run_on_node ή τη
105   // numa_run_on_nodemask. Επίσης μπορούμε να προσθέσουμε και
106   // άλλους επεξεργαστές χρησιμοποιώντας τη numa_bitmask_setbit
107   struct bitmask *mask = numa_allocate_cpumask();
108   numa_node_to_cpus(1, mask);
109   numa_sched_setaffinity(pthread_self(), mask);
110
111   // Τώρα η malloc θα εκχωρήσει μνήμη στο δεύτερο node, λόγω του
112   // policy και του ότι τρέχει πλέον στο συγκεκριμένο node
113   pointer2 = (int *) malloc(1000 * sizeof(int));
114
115   numa_free_cpumask(mask);
116
117   free(pointer1);
118   free(pointer2);
119
```

```
120   pthread_exit(NULL);
121 }
```

4.2 Άλλα εργαλεία

Το numactl μας επιτρέπει να αλλάξουμε την πολιτική σύμφωνα με την οποία θα γίνονται οι εκχωρήσεις σε κάποιο πρόγραμμα. Αν θέλουμε το πρόγραμμα μας να χρησιμοποιεί μία πολιτική, μπορούμε αντί να αλλάξουμε τον κώδικα μας να χρησιμοποιήσουμε το numactl. Επίσης μπορούμε να το χρησιμοποιήσουμε αν έχουμε κάποιο εκτελέσιμο και θέλουμε να χρησιμοποιήσει κάποια συγκεκριμένη πολιτική, γιατί πιστεύουμε ότι έτσι θα έχουμε καλύτερα αποτελέσματα από την προεπιλεγμένη πολιτική. Τέλος αν καλέσουμε το numactl με όρισμα –hardware παίρνουμε πληροφορίες όπως ο αριθμός των κόμβων, ο αριθμός των επεξεργαστών σε κάθε κόμβο, η συνολική και η διαθέσιμη μνήμη κάθε κόμβου και τέλος οι “αποστάσεις” μεταξύ κόμβων.

Το numastat είναι ένα πρόγραμμα που μας παρέχει για κάθε κόμβο πληροφορίες για τον αριθμό των hits/misses κατά την εκχώρηση μνήμης, είτε κανονικής είτε interleaved, καθώς και για το πόσες εκχωρήσεις έγιναν στον τοπικό ή σε κάποιον απομακρυσμένο κόμβο.

Κεφάλαιο 5

Πειράματα

Στο κεφάλαιο που ακολουθεί, αρχικά θα μιλήσουμε για τα συστήματα στα οποία έγιναν τα πειράματα και στη συνέχεια θα δούμε και θα αναλύσουμε τα αποτελέσματα των πειραμάτων.

5.1 Πολυπύρηνα συστήματα στα οποία έγιναν πειράματα

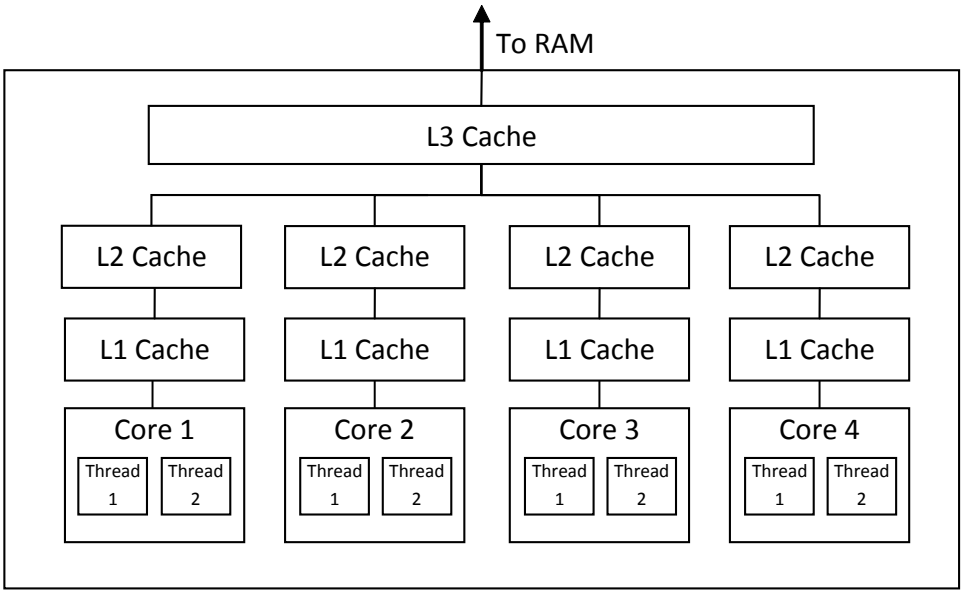
Υπάρχουν πολλά διαφορετικά συστήματα και χαρακτηριστικά όπως η υλοποίηση κάθε επεξεργαστή, ο αριθμός των cache και το μέγεθος τους, το είδος, η ταχύτητα και ο αριθμός των καναλιών της κεντρικής μνήμης (Ram) καθώς και ο αριθμός των διαύλων επικοινωνίας των επεξεργαστών με τη μνήμη επηρεάζουν τα αποτελέσματα. Επομένως κάναμε τις μετρήσεις σε περισσότερα από ένα μηχανήματα τα οποία αντιπροσωπεύουν μια ευρεία γκάμα χαρακτηριστικών. Πριν δούμε τα αποτελέσματα των πειραμάτων θα παρουσιάσουμε τα μηχανήματα που είχαμε στη διάθεση μας.

5.1.1 PC

Ένας προσωπικός υπολογιστής με επεξεργαστή Intel Core i7-860 [17] στα 2.8 GHz με τέσσερις πυρήνες. Διαθέτει τεχνολογία hyperthreading, άρα κάθε πυρήνας έχει 2 νήματα. Επιπλέον διαθέτει 4 Gb dual channel DDR3 Ram στα 1600 MHz. Τα πειράματα έγιναν με λειτουργικό σύστημα Windows 7 Professional χρησιμοποιώντας τον compiler MinGW και σε Linux Ubuntu με χρήση του gcc.

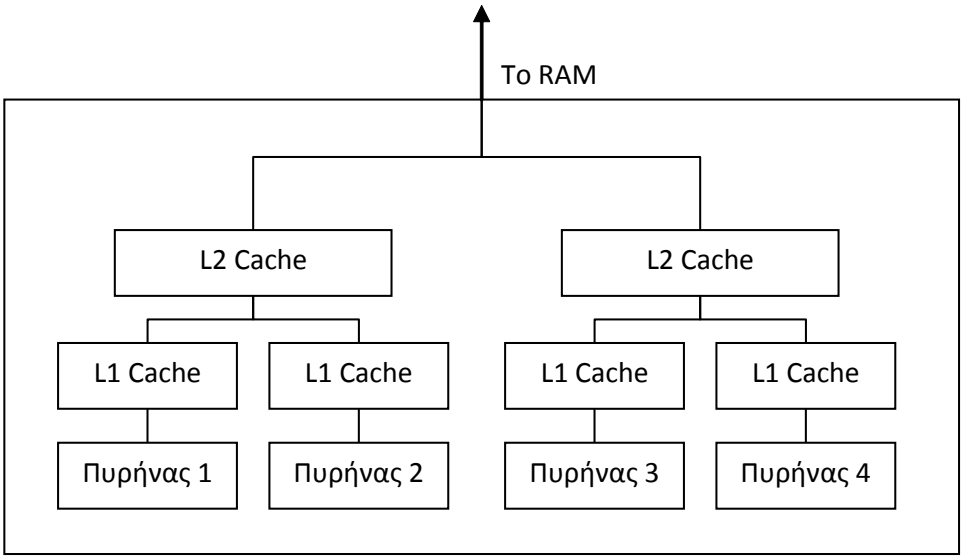
5.1.2 Paralyzer

Ανήκει στην ομάδα παράλληλης επεξεργασίας του τμήματος μας. Διαθέτει επεξεργαστή Intel Core2 Quad CPU Q8400 [16] στα 2,66 GHz με τέσσερις πυρήνες και ταχύτητα διαύλου (FSB) 1333 MHz. Οι πυρήνες ανά δύο μοιράζονται την



Σχήμα 5.1: Επεξεργαστής Intel i7 (PC)

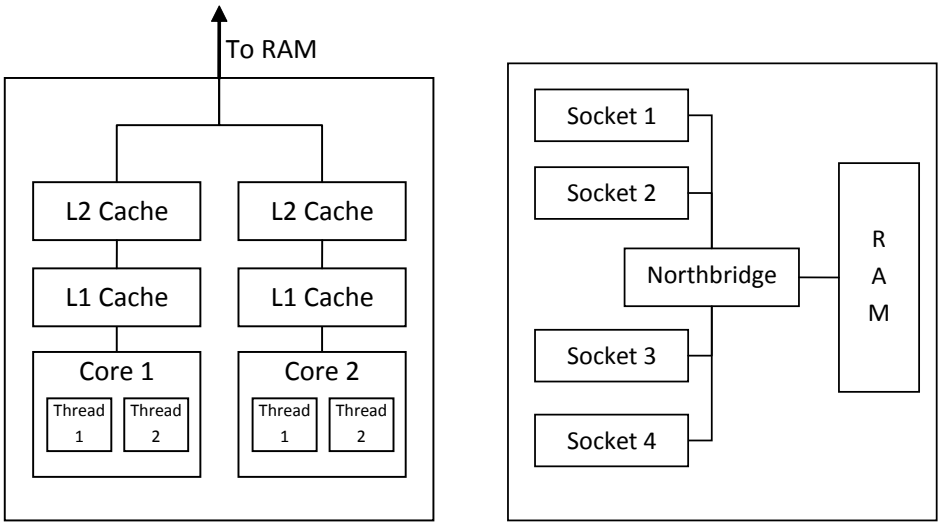
ίδια l2 cache. Επίσης διαθέτει 2 Gb dual channel DDR3 Ram στα 1066 MHz. Οι μετρήσεις έγιναν σε Debian squeeze λειτουργικό με gcc 4.4 compiler.



Σχήμα 5.2: Intel Core2 Quad (Paralyzer)

5.1.3 Paraguay

Το σύστημα paraguay αποτελεί δωρεά της εταιρίας Intel USA στην ομάδα παράλληλης επεξεργασίας του τμήματος μας. Πρόκειται για ένα διακομιστή τύπου SR6850HW4 [11]. Διαθέτει τέσσερις θέσεις επεξεργαστών (sockets) για επεξεργαστές Intel® Xeon® Processor 7040 [12, 13] (Paxville) στα 3GHz με ταχύτητα διαύλου (FSB) 667MHz. Οι επεξεργαστές αυτοί διαθέτουν δύο πυρήνες, καθένας εκ των οποίων διαθέτει τεχνολογία hyperthreading και επομένως μπορεί να χειριστεί δυο νήματα. Επίσης κάτι που δεν φαίνεται στο hwloc παρά μόνο στο datasheet του SR6850HW4 και στα αποτελέσματα των πειραμάτων είναι το γεγονός ότι οι επεξεργαστές ανά δυο βρίσκονται σε ένα δίαυλο. Κατά τη διάρκεια των πειραμάτων είχαμε απενεργοποιημένη τη λειτουργία hyperthreading οπότε είχαμε στη διάθεση μας συνολικά 8 πυρήνες, με τον καθένα να εκτελεί ένα νήμα τη φορά. Τέλος διαθέτει 8Gb Ram. Οι μετρήσεις έγιναν σε Debian λειτουργικό με τους compilers gcc, icc και suncc.



(α′) Intel Xeon 7020 (β′) SR6850HW4

Σχήμα 5.3: Paraguay

Μετά από μία αναβάθμιση του λειτουργικού σε debian squeeze, σταμάτησε να δουλεύει το binding. Αφού επικοινωνήσαμε με το mailing-list [14] της hwloc, καταλήξαμε ότι το πρόβλημα ήταν στον kernel (debian 2.6.32-5). Στη συνέχεια δοκιμάσαμε τα vanilla kernels σε ένα virtual machine και ανακαλύψαμε ότι το πρόβλημα ήταν μόνο στον 2.6.32.15, ενώ οι 2.6.34, 2.6.33.5 και 2.6.31.13 δούλευαν κανονικά. Στον 2.6.32.15 ενώ η sched_setaffinity() δεν επέστρεφε κάποιο σφάλμα και η sched_getaffinity() επέστρεφε το σωστό affinity μετά την κλήση της sched_setaffinity(), το λειτουργικό ανέθετε το κάθε νήμα σε λάθος επεξεργαστή.

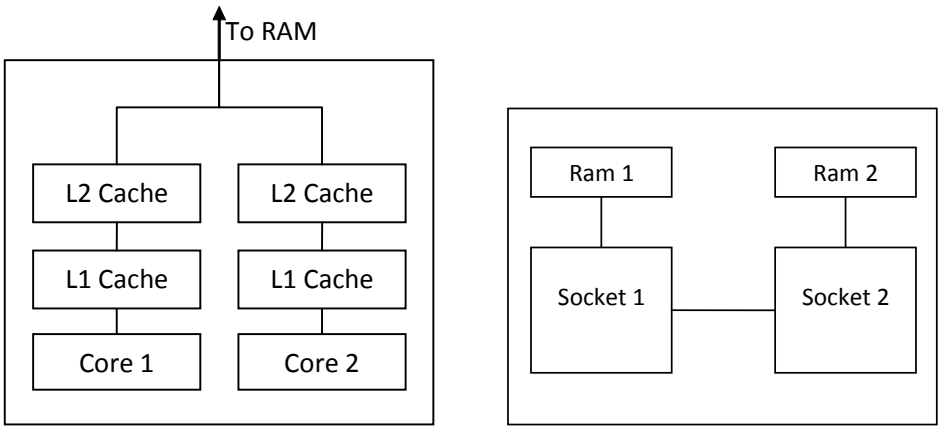
Στη συνέχεια αφού επικοινωνήσαμε ξανά με τη mailing-list της hwloc μας πρότειναν να χρησιμοποιήσουμε το git bisect για να βρούμε πότε και σε ποιο

commit εμφανίστηκε το πρόβλημα. Αφού βρήκαμε ότι εμφανίστηκε στην έκδοση 2.6.32.12 σε ένα commit που αλλάζει κάτι στο sched.c το στείλαμε στο mailing list [15] του kernel και στους maintainers του sched.c και ελπίζουμε ότι το πρόβλημα θα διορθωθεί σε επόμενη έκδοση του kernel. Εντωμεταξύ αντικαταστήσαμε τον ελαττωματικό kernel με παλιότερη έκδοση. Παρακάτω βρίσκεται το προβληματικό commit:

```
commit c6fc81afa2d7ef2f775e48672693d8a0a8a7325d
Author: John Wright <john.wright@hp.com>
Date:   Tue Apr 13 16:55:37 2010 -0600

    sched: Fix a race between ttwu() and migrate_task()

    Based on commit e2912009fb7b715728311b0d8fe327a1432b3f79 up-
    stream, but done differently as this issue is not present in
    .33 or .34 kernels due to rework in this area.

    If a task is in the TASK_WAITING state, then try_to_wake_up()
    is working on it, and it will place it on the correct cpu.

    This commit ensures that neither migrate_task() nor __migra-
    te_task() calls set_task_cpu(p) while p is in the TASK_WAKING
    state.  Otherwise, there could be two concurrent calls to
    set_task_cpu(p), resulting in the task's cfs_rq being inconsistent with its cpu.

    Signed-off-by: John Wright <john.wright@hp.com>
    Cc: Ingo Molnar <mingo@elte.hu>
    Cc: Peter Zijlstra <peterz@infradead.org>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/kernel/sched.c b/kernel/sched.c
index 2591562..3261c19 100644
--- a/kernel/sched.c
+++ b/kernel/sched.c
@@ -2116,12 +2116,10 @@ migrate_task(struct task_struct *p, int dest_cpu, struct migration_req *req)

     /*
      * If the task is not on a runqueue (and not running), then
-     * it is sufficient to simply update the task's cpu field.
+     * the next wake-up will properly place the task.
      */
-    if (!p->se.on_rq && !task_running(rq, p)) {
-        set_task_cpu(p, dest_cpu);
+    if (!p->se.on_rq && !task_running(rq, p))
         return 0;
-    }

     init_completion(&req->done);
     req->task = p;
@@ -7167,6 +7165,9 @@ static int __migrate_task(struct task_struct *p, int src_cpu, int dest_cpu)
     /* Already moved. */
     if (task_cpu(p) != src_cpu)
         goto done;
+    /* Waking up, don't get in the way of try_to_wake_up(). */
+    if (p->state == TASK_WAKING)
+        goto fail;
     /* Affinity changed (again). */
     if (!cpumask_test_cpu(dest_cpu, &p->cpus_allowed))
         goto fail;
```

5.1.4 Master Private

Στο τμήμα μας βρίσκεται ένα cluster [18] το οποίο είναι μια συστάδα υπολογιστών. Αποτελείται από 17 κόμβους Sun Fire x4100 [19] τοποθετημένους σε rack. Κάθε κόμβος διαθέτει 2 dual Opterons επεξεργαστές στα 2.14 Ghz και μνήμη 4 Gb. Εκτελεί λειτουργικό Debian lenny/sid με compiler gcc 4.3.1. Διαφέρει από τα προηγούμενα συστήματα γιατί κάθε επεξεργαστής διαθέτει τη δική του μνήμη Ram, είναι δηλαδή μηχάνημα NUMA (βλέπε κεφάλαιο 4). Ένας επεξεργαστής έχει πρόσβαση στη μνήμη ενός άλλου κόμβου αλλά όχι άμεσα, με αποτέλεσμα να χρειάζεται περισσότερο χρόνο απ' ότι για την προσπέλαση της μνήμης του κόμβου όπου βρίσκεται. Οι μετρήσεις έγιναν σε έναν κόμβο (τον Master Private) άρα είχαμε στη διάθεση μας τέσσερις πυρήνες.



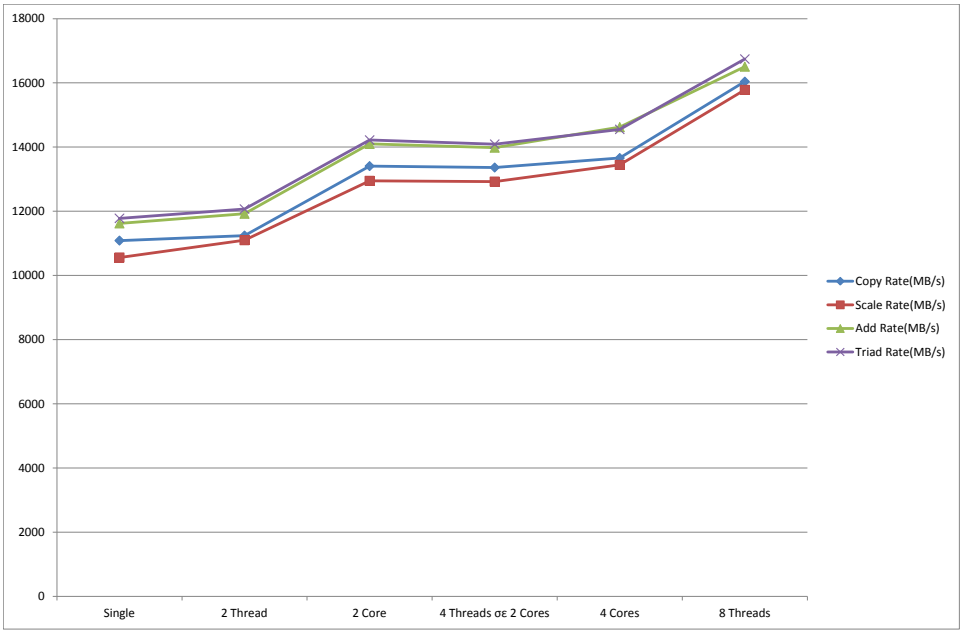
(α΄) Opteron (β΄) Sun Fire x4100

Σχήμα 5.4: Master Private, κόμβος του Cluster

5.2 Αποτελέσματα μετρήσεων

5.2.1 PC

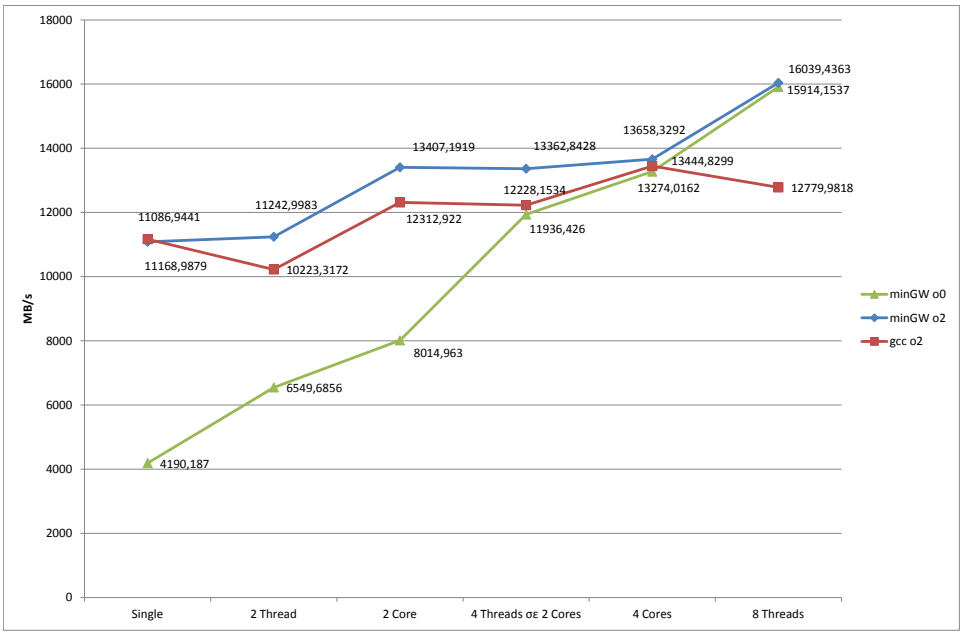
Στο σχήμα 5.5 βλέπουμε τα αποτελέσματα του Stream στο PC. Βλέπουμε ότι υπάρχει μία αύξηση της τάξεως του 21% όταν χρησιμοποιούμε περισσότερους από έναν πυρήνες. Την μέγιστη τιμή για το bandwidth την παίρνουμε όταν αξιοποιούμε όλα τα νήματα του επεξεργαστή και είναι περίπου 45% μεγαλύτερη από αυτήν της σειριακής εκτέλεσης. Όπως θα δούμε στο σχήμα 5.6 ο gcc όταν χρησιμοποιούμε όλα τα νήματα σε αντίθεση με το MinGW έχει πτώση στο bandwidth.



Σχήμα 5.5: Stream στο PC

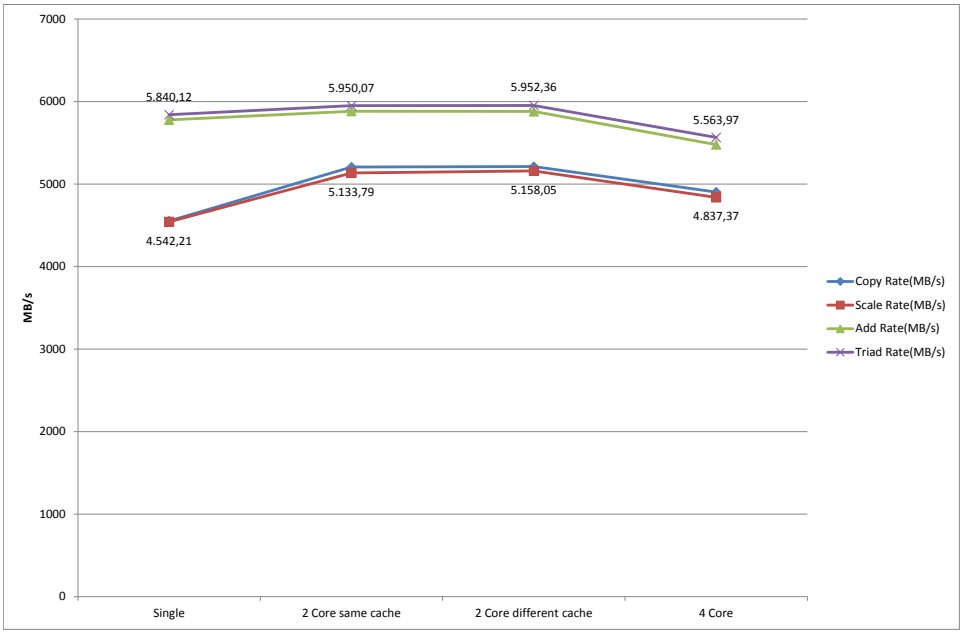
Αρχικά τα προγράμματα έγιναν compile χωρίς optimizations. Στα Linux με τη χρήση του gcc η διαφορά των αποτελεσμάτων με optimizations με αυτών χωρίς ήταν πολύ μικρή. Στα Windows όμως με τη χρήση του MinGW η διαφορά είναι πολύ μεγάλη όπως φαίνεται στο σχήμα 5.6 όταν χρησιμοποιούμε λίγα threads και μικραίνει όσο ο αριθμός τους αυξάνει. Αυτό κατά πάσα πιθανότητα συμβαίνει γιατί ο μη-optimized κώδικας δεν είναι αρκετά καλός με αποτέλεσμα να χάνεται χρόνος στην πραγματοποίηση των όποιων πράξεων και να υπάρχουν χρονικά διαστήματα στα οποία δεν υπάρχουν καθόλου requests για δεδομένα από τους επεξεργαστές προς τη μνήμη.

Τα πειράματα που θα παρουσιάσουμε στα υπόλοιπα μηχανήματα έγιναν όλα με χρήση του -O2.



Σχήμα 5.6: Optimization στο PC (Copy Rate)

5.2.2 Paralyzer

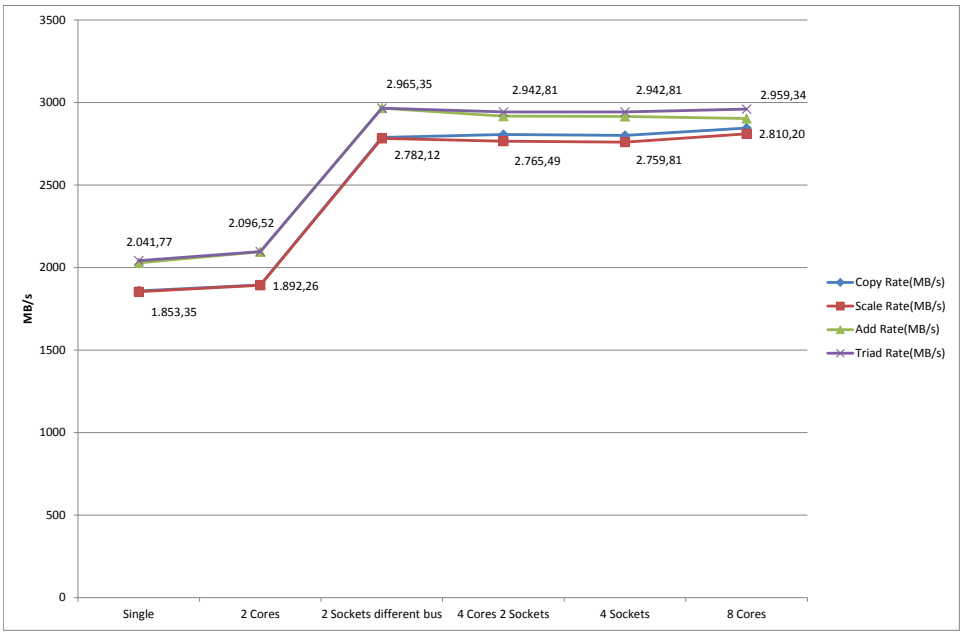


Σχήμα 5.7: Stream στον Paralyzer

Όπως βλέπουμε στο σχήμα 5.7 σε αυτό το μηχάνημα παρατηρούμε ελάχι-

στη αύξηση όταν χρησιμοποιούμε δυο πυρήνες. Όταν χρησιμοποιούμε τέσσερις όχι μόνο δεν παρατηρούμε περαιτέρω αύξηση αλλά λόγω της συμφόρησης παρατηρούμε πτώση στα αποτελέσματα. Στην περίπτωση των Add και Triad τα αποτελέσματα είναι μικρότερα και από αυτά που είχαμε με τη χρήση μονάχα ενός επεξεργαστή. Στα αρχικά πειράματα όπου δεν είχαμε ενεργοποιήσει το optimization υπήρχε αύξηση μόνο εφόσον χρησιμοποιούσαμε 2 πυρήνες που διέθεταν διαφορετική l2 cache.

5.2.3 Paraguay

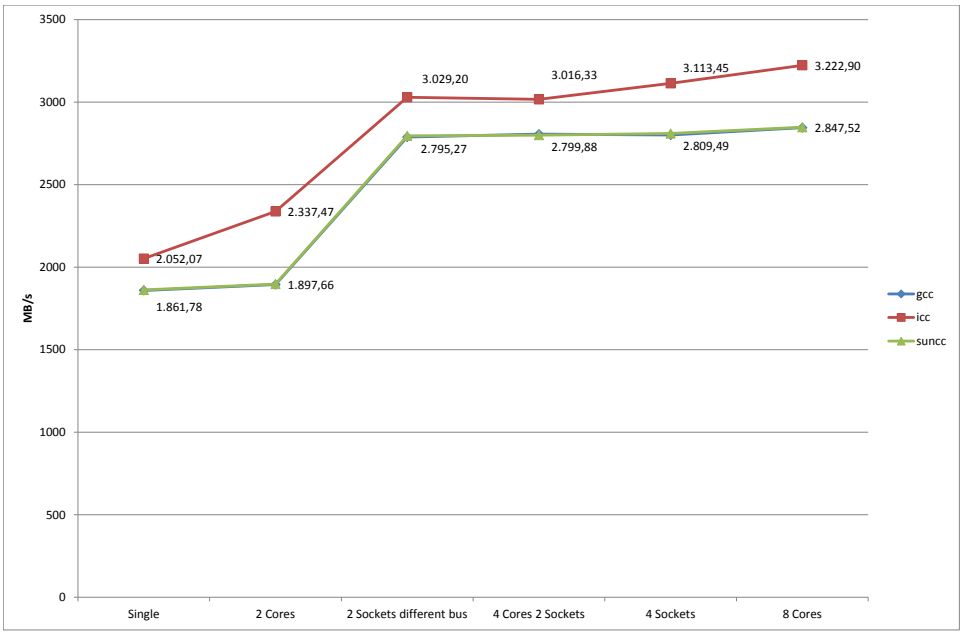


Σχήμα 5.8: Stream στην Paraguay με gcc

Όπως βλέπουμε στο σχήμα 5.8 το bandwidth αυξάνεται ελάχιστα όταν χρησιμοποιούμε δυο πυρήνες που βρίσκονται στο ίδιο socket. Το ίδιο ισχύει και για όταν χρησιμοποιούμε δύο πυρήνες σε sockets που βρίσκονται πάνω στον ίδιο δίαυλο (δεν φαίνεται στο γράφημα αλλά οι μετρήσεις είναι ίδιες με την περίπτωση όπου χρησιμοποιούμε δύο πυρήνες στο ίδιο socket). Βελτίωση παρατηρείται μόνο όταν χρησιμοποιούμε πυρήνες όπου ο καθένας βρίσκεται σε διαφορετικό δίαυλο.

Κάτι άλλο που μπορούμε να δούμε από το γράφημα είναι ότι ενώ το Copy έχει σχεδόν τα ίδια αποτελέσματα με το Scale και το Triad με το Add υπάρχει μια μικρή διαφορά μεταξύ των αποτελεσμάτων που έχουν τα Copy και Scale με αυτά των Add και Triad. Αυτό κατά πάσα πιθανότητα οφείλεται στο ότι τα Copy και Scale προσπελαύνουν δυο πίνακες ενώ τα Add και Triad προσπελαύνουν τρεις πίνακες. Το ίδιο φαινόμενο παρατηρείται και στα άλλα μηχανήματα.

Όπως είπαμε και στην ενότητα 5.1.3 στην Paraguay εκτός από τον gcc κάναμε μετρήσεις και με τον icc της Intel και τον suncc της Sun. Στο σχήμα 5.9 βλέπουμε συγκριτικά τα αποτελέσματα αυτών των compilers. Τα αποτελέσματα του gcc και του icc είναι σχεδόν πανομοιότυπα. Ο icc όμως παρουσιάζει μια αύξηση ≈ 10% (στην περίπτωση των δυο πυρήνων στο ίδιο socket η αύξηση είναι της τάξεως του 24%) σε σχέση με τους άλλους δύο.

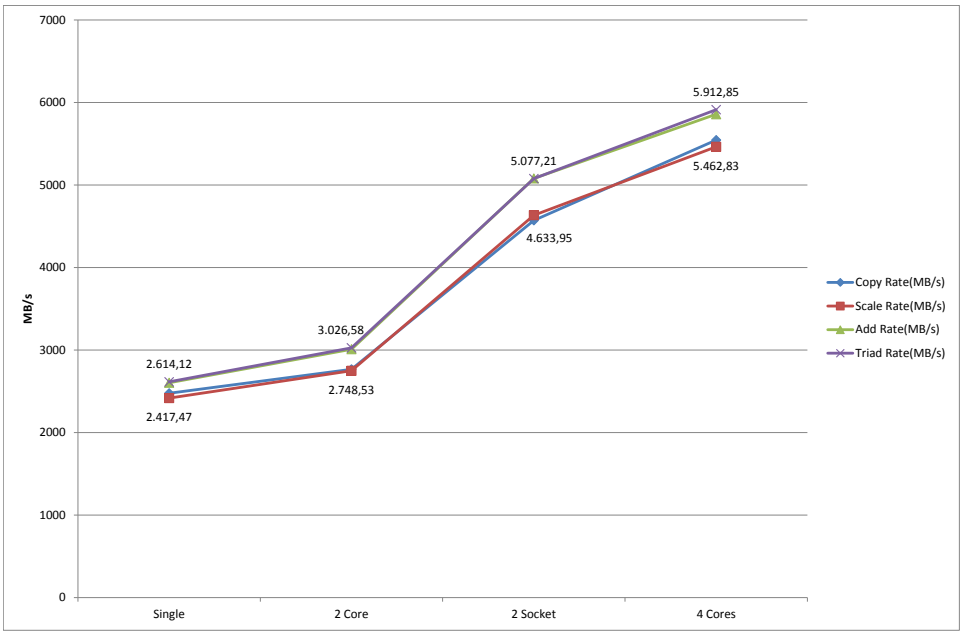


Σχήμα 5.9: Σύγκριση compilers στην Paraguay (Copy Rate)

5.2.4 Master Private

Λόγω της αρχιτεκτονικής αυτού του συστήματος στο σχήμα παρατηρούμε τις μεγαλύτερες αυξήσεις σε bandwidth σε σχέση με τα άλλα συστήματα. Όπως είπαμε και στην ενότητα 5.1.4 κάθε socket έχει τη δικιά του μνήμη Ram με αποτέλεσμα, όταν χρησιμοποιούμε δυο πυρήνες σε διαφορετικά socket, το bandwidth να διπλασιάζεται σε σχέση με αυτό του ενός thread. Σημαντική όμως είναι και η αύξηση όταν χρησιμοποιούμε και τους τέσσερις πυρήνες.

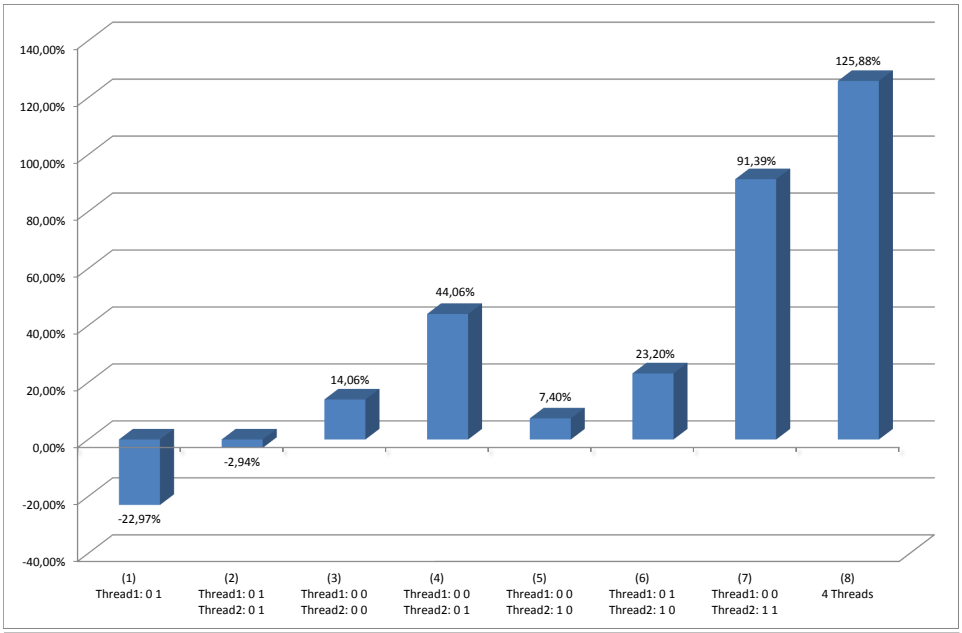
Στο σχήμα 5.11 παρουσιάζονται τα Copy Rates διάφορων διατάξεων των νημάτων και των δεδομένων τους σε σχέση με το Copy Rate που έχει το σειριακό Stream. Στην πρώτη στήλη του σχήματους αυτού μπορούμε να δούμε ότι η χρήση δεδομένων που βρίσκονται σε απομακρυσμένο κόμβο έχει περίπου 23% καθυστέρηση στο συγκεκριμένο σύστημα. Όταν χρησιμοποιούμε δύο νήματα (στήλη 2) για να προσπελάσουμε την απομακρυσμένη μνήμη έχουμε περίπου ίσο bandwidth με αυτό του σειριακού προγράμματος. Από αυτήν καθώς και από την επόμενη στήλη του γραφήματος, η οποία δείχνει το bandwidth όταν χρη-



Σχήμα 5.10: Stream στο Master Private

σιμοποιούμε δύο νήματα με τα δεδομένα τους να βρίσκονται στον ίδιο κόμβο, παρατηρούμε ότι ένα νήμα μπορεί να εκμεταλλευτεί γύρω στο 80-85% του bandwidth ενός κόμβου. Στην τέταρτη στήλη φαίνεται ότι η τοποθέτηση δεδομένων και στους δύο κόμβους αποδίδει ακόμα και όταν τα νήματα μας βρίσκονται σε ένα μόνο κόμβο.

Στην πέμπτη στήλη είναι η περίπτωση όπου οι αρχικοποιήσεις γίνονται από ένα κεντρικό νήμα. Παρατηρούμε ότι το bandwidth είναι ελάχιστα μεγαλύτερο απ' ότι με τη χρήση μονάχα ενός νήματος. Αν το συγκρίνουμε και με την έβδομη στήλη στην οποία φαίνεται το bandwidth όταν χρησιμοποιούμε δύο νήματα σε διαφορετικούς κόμβους με τα δεδομένα να βρίσκονται στην τοπική μνήμη, συμπεραίνουμε ότι για να εκμεταλλευτούμε στο έπακρον τις δυνατότητες του συστήματος δεν πρέπει να αφήσουμε στην τύχη τις αρχικοποιήσεις. Στην έκτη στήλη βλέπουμε την ταχύτητα στην περίπτωση που έχουμε τα δεδομένα κάθε ενός εκ των δύο νημάτων στον κόμβο του άλλου νήματος. Μπορεί να φαίνεται σαν ακραία περίπτωση και bug του προγράμματος, αλλά τόσο αυτή όσο και οι άλλες περιπτώσεις είναι πολύ πιθανόν να συμβούν κατά την διάρκεια εκτέλεσης του προγράμματος μας. Ο δρομολογητής του συστήματος, είτε λόγω κατανάλωσης/θερμοκρασίας είτε διότι κάποιοι πυρήνες είναι άεργοι, πολλές φορές αλλάζει τον πυρήνα στον οποίο τρέχει ένα νήμα. Κάτι τέτοιο θα μπορούσε να αποβεί καταστροφικό για τις επιδόσεις του προγράμματος μας αν έτρεχε σε ένα τέτοιο σύστημα χωρίς cpu binding.



№Στήλης	Διάταξη	Επεξήγηση
1	Thread1: 0 1	Ένα νήμα στον κόμβο 0 με τα δεδομένα στον κόμβο 1
2	Thread1: 0 1 Thread2: 0 1	Δύο νήματα στον κόμβο 0 με τα δεδομένα τους στον κόμβο 1
3	Thread1: 0 0 Thread2: 0 0	Δύο νήματα στον κόμβο 0 με τα δεδομένα τους στον κόμβο 0
4	Thread1: 0 0 Thread2: 0 1	Δύο νήματα στον κόμβο 0 με τα δεδομένα του πρώτου στον κόμβο 0 και τα δεδομένα του δεύτερου στον κόμβο 1
5	Thread1: 0 0 Thread2: 1 0	Ένα νήμα στον κόμβο 0 και ένα στον κόμβο 1 με τα δεδομένα και των δύο στον κόμβο 0
6	Thread1: 0 1 Thread2: 1 0	Ένα νήμα στον κόμβο 0 με τα δεδομένα του στον κόμβο 1 και ένα νήμα στον κόμβο 1 με τα δεδομένα του στον κόμβο 0
7	Thread1: 0 0 Thread2: 1 1	Ένα νήμα στον κόμβο 0 και ένα στον κόμβο 1 με τα δεδομένα τους στους αντίστοιχους κόμβους
8	4 Threads	Τέσσερα νήματα, δύο στον πρώτο κόμβο και δύο στον δεύτερο με τα δεδομένα τους στην τοπική μνήμη

Σχήμα 5.11: Copy rate συγκριτικά με ένα νήμα στο Master Private

5.2.5 Συμπεράσματα

Όπως είδαμε από τα πειράματα σε συστήματα κοινόχρηστης μνήμης ένας επεξεργαστής μπορεί να αντλήσει σχεδόν όλο το bandwidth της μνήμης. Σε συστήματα όπως η Paraguay (σχήμα 5.8) όπου υπάρχουν περισσότεροι από ένας δίαυλοι επικοινωνίας με την μνήμη παρατηρούμε μία μέτρια αύξηση του bandwidth της τάξεως του 45-60%. Σε άλλα όπως το PC που διαθέτει γρήγορη μνήμη σε σχέση με άλλα συστήματα χωρίς όμως κάποια άλλη ιδιαιτερότητα όπως η Paraguay η αύξηση ήταν μικρότερη (περίπου 20-30%, βλέπε σχήμα 5.5). Τέλος σε συστήματα όπως το Paralyzer συναντάμε ακόμα και μείωση (γύρω στο

5%, σχήμα 5.7) όταν χρησιμοποιούμε πολλούς επεξεργαστές η οποία λογικά θα ήταν ακόμα μεγαλύτερη αν είχαμε στην διάθεση μας περισσότερους.

Αντιθέτως σε συστήματα NUMA όπου κάθε ομάδα επεξεργαστών έχει πρόσβαση σε μία ιδιωτική μνήμη, κάθε μνήμη μπορεί να επεξεργάζεται αιτήσεις ανεξάρτητα από τις άλλες μνήμες με αποτέλεσμα να αποφεύγεται (ή έστω να μην είναι τόσο έντονη) η συμφόρηση της. Σε αυτά τα συστήματα η αύξηση του bandwidth είναι ανάλογη της αύξησης των κόμβων του συστήματος (σχήμα 5.10).

Τα προγράμματα που συναντάμε στην καθημερινότητα μας συνήθως εκτελούν περισσότερες και πιο πολύπλοκες πράξεις πάνω στα δεδομένα μας απ' ότι το Stream. Αυτό σημαίνει ότι καταναλώνουν περισσότερο χρόνο εκτελώντας πράξεις πάνω στα δεδομένα, και λιγότερο ζητώντας από την μνήμη καινούρια δεδομένα. Εκεί η χρήση περισσότερων επεξεργαστών αποφέρει και καλύτερους χρόνους εκτέλεσης καθώς οι χρόνοι τους δεν εξαρτώνται τόσο πολύ από το bandwidth της μνήμης όσο το πρόγραμμα που χρησιμοποιήσαμε για benchmark. Επιπλέον σε περιπτώσεις όπου περισσότερα από ένα νήματα προσπελαύνουν τις ίδιες διευθύνσεις μνήμης, αν φροντίσουμε να βρίσκονται σε "κοντινούς" επεξεργαστές μπορούμε να εκμεταλλευτούμε την ύπαρξη των κρυφών μνημών. Αυτό έχει ως αποτέλεσμα η προσπέλαση μιας θέσεις μνήμης να γίνεται μόνο μία φορά (ή έστω λιγότερες φορές) αντί να την προσπελαύνει κάθε επεξεργαστής ξεχωριστά.

Παρόλα αυτά η συνεχής αύξηση του αριθμού των επεξεργαστών που συναντάμε στα σημερινά συστήματα αυξάνει τις πιθανότητες οι επιδόσεις ενός προγράμματος να περιορίζονται λόγω συμφόρησης της μνήμης σε συστήματα με μονάχα μία κύρια μνήμη. Γι' αυτό είναι αναγκαία η υιοθέτηση διαφορετικών αρχιτεκτονικών, όπως είναι η NUMA, που δεν παρουσιάζει πρόβλημα στην αύξηση του bandwidth της μνήμης όπως τα συστήματα με φυσικά κοινόχρηστη μνήμη. Πρέπει όμως και οι προγραμματιστές να εξοικειωθούν με τέτοια συστήματα ώστε ο κώδικας που γράφουν να είναι κατάλληλος, για να αποφεύγεται η χρήση της μνήμης μόνο ενός κόμβου και άρα η εμφάνιση του προβλήματος που έχουμε σε συστήματα με κοινόχρηστη μνήμη.

Από τα παραπάνω καταλήγουμε ότι οι προγραμματιστές πρέπει να έχουν γνώση της αρχιτεκτονικής στην οποία θα εκτελεστεί το πρόγραμμα τους και να φροντίζουν να προσαρμόζεται κατάλληλα στην περίπτωση που προορίζεται σε ένα πλήθος αρχιτεκτονικών, ώστε να έχει καλές επιδόσεις.

Κεφάλαιο 6

Επίλογος

6.1 Σύνοψη εργασίας

Η αύξηση των επιδόσεων των επεξεργαστών μέσω της τεχνολογίας φτάνει στα όρια της, καθώς το μέγεθος των transistors έχει γίνει πλέον πολύ μικρό και η αύξηση της συχνότητας τους απαιτεί πολύ μεγάλη αύξηση στην κατανάλωση ενέργειας. Το γεγονός αυτό ώθησε τη δημιουργία και υιοθέτηση πολυπύρηνων επεξεργαστών.

Στις μέρες μας όπου τα συστήματα που κυκλοφορούν στην αγορά είναι πλέον στην πλειοψηφία τους παράλληλα, η γνώση παράλληλου προγραμματισμού είναι απαραίτητο εργαλείο για κάθε προγραμματιστή. Όσο αυξάνεται όμως ο αριθμός των επεξεργαστών που έχουν τα συστήματα και η πολυπλοκότητα τους, τόσο μειώνονται οι αποδόσεις των προγραμμάτων εφόσον αυτά δε λαμβάνουν υπόψιν τους τις ιδιαιτερότητες και τα όρια των συστημάτων πάνω στα οποία εκτελούνται.

Αφού πρώτα μελετήσαμε τις διάφορες αρχιτεκτονικές που συναντάμε στα σημερινά συστήματα και έχοντας υπόψιν μας το πρόβλημα αυτό, παρουσιάσαμε κάποια εργαλεία που έχουμε στην διάθεση μας ως προγραμματιστές για την αναγνώριση της τοπολογίας του συστήματος στο οποίο εκτελείται κάθε φορά ο κώδικας μας, ώστε να έχουμε τη δυνατότητα να μοιράσουμε τις εργασίες και τα δεδομένα τους αποδοτικότερα. Από αυτά ξεχωρίζει η βιβλιοθήκη hwloc για την ευκολία χρήσης της, το πλήθος των πληροφοριών και εργαλείων που μας παρέχει καθώς και για τη μεταφερσιμότητα της. Μας παρέχει μια δενδρική απεικόνιση της τοπολογίας του συστήματος, καθώς και κλήσεις για την ανάθεση νημάτων σε συγκεκριμένους επεξεργαστές. Πλέον διαθέτει και συναρτήσεις για τη διαχείριση της μνήμης σε συστήματα NUMA.

Όσο αυξάνεται ο αριθμός των επεξεργαστών που έχουν τα συστήματα, τόσο αυξάνεται και το χάσμα μεταξύ της ταχύτητας των επεξεργαστών και της μνήμης, αφού περισσότεροι επεξεργαστές προσπαθούν να λάβουν δεδομένα από τη μνήμη στον ίδιο χρόνο. Για το λόγο αυτό στο παρόν κείμενο μελετήσαμε τις επιδόσεις της μνήμης σε πολυπύρηνα συστήματα όταν περισσότεροι

από ένας επεξεργαστές θέλουν να προσπελάσουν την μνήμη. Πρώτα παραλληλοποιήσαμε το μετροπρόγραμμα Stream με χρήση posix νημάτων και χρησιμοποιήσαμε τη βιβλιοθήκη hwloc για την επιλογή σε κάθε πείραμα του επεξεργαστή στον οποίο θα τρέχει το κάθε νήμα του προγράμματος μας. Στη συνέχεια το χρησιμοποιήσαμε για να διεξάγουμε πειράματα σε συστήματα με διαφορές στην αρχιτεκτονική.

Από τα αποτελέσματα των πειραμάτων συμπεραίνουμε ότι το μέγιστο bandwidth της μνήμης σε συστήματα κοινόχρηστης μνήμης μπορεί να περιορίσει τις επιδόσεις του κώδικα μας, ιδιαίτερα όταν ο αριθμός των επεξεργαστών είναι μεγάλος και όταν το πρόγραμμα μας εκτελεί απλές πράξεις πάνω σε μεγάλο όγκο δεδομένων. Λόγω της μεγάλης συμφόρησης της μνήμης και του δίαυλου που την ενώνει με τους επεξεργαστές υπάρχει περίπτωση το πρόγραμμα μας να έχει χειρότερες επιδόσεις απ' ότι το αντίστοιχο σειριακό.

Το γεγονός αυτό αποτελεί έναν από τους κύριους παράγοντες που συντέλεσαν στη διάδοση της αρχιτεκτονικής NUMA όπου κάθε επεξεργαστής ή ομάδα επεξεργαστών διαθέτει μια προσωπική μνήμη, η οποία είναι μεν προσβάσιμη και από τους άλλους αλλά με κάποιο επιπλέον κόστος. Στα μηχανήματα αυτά είναι ανάγκη να φροντίσουμε για μια "καλή" κατανομή της μνήμης και του φόρτου εργασίας του κάθε επεξεργαστή ώστε να ελαχιστοποιήσουμε τον αριθμό των προσβάσεων "απομακρυσμένων" μνημών και παράλληλα το κόστος που συνεπάγονται αυτές οι προσβάσεις.

6.2 Επεκτάσεις

Πιστεύουμε ότι τα εργαλεία που χρησιμοποιήσαμε καθώς και τα συμπεράσματα που βγάλαμε από την πτυχιακή αυτή μπορούν να αξιοποιηθούν στην βελτίωση των παράλληλων εφαρμογών. Η υιοθέτηση τους σε συστήματα χρόνου εκτέλεσης (runtime) διαφόρων προγραμματιστικών μοντέλων θα έχει ως αποτέλεσμα την βελτίωση τους. Για παράδειγμα θα μπορούσαμε να τα εκμεταλλευτούμε στο OpenMP.

Η ομάδα παράλληλης επεξεργασίας (Paragroup) του τμήματος μας διαθέτει έναν compiler για OpenMP σε γλώσσα C ονόματι ompi. Ο ompi είναι ένας source-to-source ανοικτού κώδικα (open source) compiler που υλοποιεί την πλατφόρμα OpenMP 3.1. Μετατρέπει ένα σειριακό πρόγραμμα γραμμένο σε C που διαθέτει οδηγίες (directives) OpenMP σε ένα παράλληλο πρόγραμμα, το οποίο μπορεί στη συνέχεια να γίνει compile χρησιμοποιώντας οποιοδήποτε μεταφραστή C είναι διαθέσιμος στο σύστημα μας χωρίς να χρειάζεται ο χρήστης να γράψει κάποιο κώδικα για το συντονισμό των διάφορων παράλληλων εργασιών.

Ο ompi διαθέτει βιβλιοθήκες runtime που επιτρέπουν την εκτέλεση εφαρμογών σε διάφορες πλατφόρμες, τόσο κοινόχρηστης όσο και κατανεμημένης μνήμης. Στον ompi έχουμε ήδη υλοποιήσει την τοποθέτηση των νημάτων σε επεξεργαστές (binding), γεγονός που απέφερε βελτίωση στις επιδόσεις κάποιων προγραμμάτων. Πιο συγκεκριμένα, κατά την αρχικοποίηση του προγράμματος

δημιουργούνται τόσα νήματα όσοι είναι οι πυρήνες του συστήματος. Στη συνέχεια κάθε ένα από αυτά τοποθετείται στον αντίστοιχο πυρήνα. Τα νήματα αυτά παραμένουν ενεργά καθόλη την διάρκεια του προγράμματος και εκτελούν τον κώδικα του χρήστη.

Ένα δεύτερο κομμάτι που θα μπορούσαμε να βελτιώσουμε είναι η δρομολόγηση των εργασιών του OpenMP (tasks) αξιοποιώντας την γνώση της τοπολογίας του συστήματος. Για παράδειγμα όταν ένας άεργος επεξεργαστής αναζητά εργασίες για να εκτελέσει, μπορεί γνωρίζοντας την τοπολογία να "ψάξει" πρώτα σε "κοντινούς" επεξεργαστές. Αυτού του είδους η ιεραρχική αναζήτηση αποτελεί ένα από τα μελλοντικά σχέδια του Paragroup.

Βιβλιογραφία

[1] Amdahl's law http://en.wikipedia.org/wiki/Amdahl%27s_law

[2] STREAM: Sustainable Memory Bandwidth in High Performance Computers http://www.cs.virginia.edu/stream/

[3] Stream source code http://www.cs.virginia.edu/stream/FTP/Code/stream.c

[4] CPU_SET http://www.kernel.org/doc/man-pages/online/pages/man3/CPU_SET.3.html

[5] Portable Hardware Locality (hwloc) http://www.open-mpi.org/projects/hwloc/http://runtime.bordeaux.inria.fr/hwloc/

[6] hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications http://hal.inria.fr/docs/00/42/98/89/PDF/main.pdf

[7] Libtopology https://gforge.inria.fr/projects/libtopology

[8] Portable Linux Processor Affinity (PLPA) http://www.open-mpi.org/projects/plpa/

[9] Get some Topology Information http://www.open-mpi.org/projects/hwloc/doc/v1.0.1/group__hwlocality__information.php

[10] hwloc_obj Struct Reference http://www.open-mpi.org/projects/hwloc/doc/v1.0.1/structhwloc__obj.php

[11] Intel® Server System SR4850HW4 http://download.intel.com/support/motherboards/server/sr4850hw4/sb/sr6850hw4m_system_tps.pdf

[12] Intel® Xeon® Processor 7040 http://ark.intel.com/Product.aspx?id=27226&processor=7040&spec-codes=SL8UC

[13] Dual-Core Intel® Xeon® Processor 7000 Series Datasheet http://www.intel.com/Assets/PDF/datasheet/309626.pdf

[14] Problem with hwloc_set_thread_cpubind() and pthread_barrier_wait() on new debian http://www.open-mpi.org/community/lists/hwloc-users/2010/06/0153.php

[15] sched_setaffinity not working with kernel 2.6.32.15 http://www.gossamer-threads.com/lists/linux/kernel/1216752?nohighlight=1#1216752

[16] Intel® Core™2 Quad Processor Q8400 http://ark.intel.com/Product.aspx?id=38512

[17] Intel® Core™ i7-860 Processor http://ark.intel.com/Product.aspx?id=41316

[18] Συστάδα Υπολογιστών Sun ® http://gatepc73.cs.uoi.gr:8880/main/

[19] Sun Fire X4100 http://www.sun.com/servers/entry/x4100/specs.xml

[20] Non-Uniform Memory Access http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access

[21] Novell A NUMA API for LINUX* http://developer.amd.com/assets/LibNUMA-WP-fv1.pdf

[22] An NUMA API for Linux, Andi Kleen 2004 http://www.halobates.de/numaapi3.pdf