Παράλληλη εκτέλεση κώδικα σε απομακρυσμένες συσκευές

Ηλίας Κλεφτάκης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πολυτεχνική Σχολή
Πανεπιστήμιο Ιωαννίνων

Σεπτέμβριος 2018

Περιεχομενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Περιληψη

Ηλίας Κλεφτάκης, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος 2018.
Παράλληλη εκτέλεση κώδικα σε απομακρυσμένες συσκευές.
Επιβλέπων: Βασίλειος Δημακόπουλος, Αναπληρωτής Καθηγητής.

Η εκτέλεση κώδικα σε άλλους υπολογιστές στα πλαίσια ενός κατανεμημένου υπολογισμού κρύβει δυσκολίες και μπορεί να είναι δύσκολη στη διαχείριση και επιρρεπής σε σφάλματα για πολλούς προγραμματιστές. Σε αυτή την εργασία εξετάζουμε την οδηγία *target* του OpenMP API που μας επιτρέπει να φορτώνουμε κώδικα και δεδομένα σε άλλες συσκευές. Στη συνέχεια, κάνουμε μία επισκόπηση του OMPi, ενός μεταφραστή C ανοιχτού κώδικα που υποστηρίζει το OpenMP και του τρόπου που χειρίζεται την οδηγία *target*. Επεκτείνουμε τον OMPi προσθέτοντας υποστήριξη για έναν νέο τύπο συσκευών, τους κόμβους από ένα cluster. Έτσι, η γνώση παράλληλου προγραμματισμού με το OpenMP είναι αρκετή για ένα προγραμματιστή ώστε να πραγματοποιήσει κατανεμημένους υπολογισμούς, κατάλληλους για ένα cluster αλλά χωρίς να περιορίζονται σε αυτό. Εσωτερικά, χρησιμοποιούμε το πρότυπο του MPI για να διαχειριστούμε την επικοινωνία μεταξύ του κύριου υπολογιστή και των συσκευών αξιόπιστα και γρήγορα. Κάνουμε μία ενδελεχή συζήτηση για την υλοποίηση αυτού του νέου τύπου συσκευής, καλύπτοντας όλες τις σημαντικές λεπτομέρειες. Τέλος, παρουσιάζουμε και σχολιάζουμε τα αποτελέσματα μίας κατάλληλα τροποποιημένης έκδοσης των πειραμάτων επίδοσης BOTS, τα οποία χρησιμοποιήσαμε προκειμένου να αποτιμήσουμε τη συμπεριφορά και τις επιδόσεις των νέων μας "συσκευών".

Abstract

Ilias Kleftakis, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, September 2018.
Parallel code execution at remote devices.
Advisor: Vassilios Dimakopoulos, Associate Professor.

Offloading code to other computers for a distributed computation can be cumbersome, tricky and error prone for many programmers. In this thesis we examine the *target* directive of the OpenMP API that allows us to offload code and data to other devices. Then, we take a look at OMPi, an open source C compiler with OpenMP support and the way it handles the *target* directive. We extend OMPi by adding support a novel type of devices: the nodes of a cluster. Thus, knowledge of OpenMP is enough for a programmer to accomplish distributed computations, suitable for but not limited to clusters. Under the hood, we use the MPI standard to handle communication between the host and the devices in a reliable and efficient way. We thoroughly discuss the implementation of this new device type, while covering all of its important details. Finally, we report and comment on the results of a properly modified version of the BOTS benchmark suite, which was used to assess the behavior and the performance of our new "devices".

Κεφαλαιο 1

Εισαγωγη

1.1 Η εξέλιξη των Η/Υ

Ο άνθρωπος προσπαθούσε να εφεύρει νέους τρόπους ώστε να βελτιώσει τη ζωή του σε όλη τη διάρκεια της ιστορίας του. Οι πρώτες μαρτυρίες για ύπαρξη υπολογιστών χρονολογούνται ήδη από το 2200 π.Χ. περίπου και δεν ήταν άλλοι από τα γνωστά αριθμητήρια (ή άβακες) που χρησιμοποιούν τα περισσότερα παιδιά σήμερα στην πρώτη τάξη του σχολείου. Πιο πολύπλοκες μηχανές υπολογισμών άρχισαν να κατασκευάζονται σε μεγαλύτερο ρυθμό τον 17ο αιώνα. Αρχικά χρησιμοποιούνταν μόνο για μαθηματικούς υπολογισμούς αλλά η επινόηση της άλγεβρας Boole και των διάτρητων καρτών έθεσαν τις βάσεις για μία νέα, επαναστατική εποχή.

Η πρώτη περιγραφή ενός υπολογιστή με τη μορφή που έχει σε γενικές γραμμές σήμερα συναντάται σε μία εργασία του *John von Neumann* το 1945, ενώ ο πρώτος τέτοιος υπολογιστής εμφανίζεται το 1946 και έχει το όνομα ENIAC. Φυσικά δεν είχε καμία σχέση με αυτό που πάει στο μυαλό μας όταν ακούμε τη λέξη υπολογιστής. Αποτελούνταν από περισσότερες από 18.000 λυχνίες κενού που καίγονταν συχνά

και 1500 ηλεκτρονόμους. Ζύγιζε 30 τόνους, καταλάμβανε 163 τετραγωνικά μέτρα χώρο και κατανάλωνε 140 κιλοβάτ ισχύ. Παρόλα αυτά, ήταν αδιαμφισβήτητα ένα μεγάλο βήμα προς το σήμερα.

Από τότε ο άνθρωπος προσπαθούσε διαρκώς να βελτιστοποιήσει κάθε πτυχή αυτού του εγχειρήματος. Είτε αυτό ήταν το μέγεθος, είτε η κατανάλωση ενέργειας, είτε η χωρητικότητα μνήμης, είτε ακόμα σημαντικότερο η επεξεργαστική ισχύς. Ενώ ο ENIAC μπορούσε να εκτελέσει 5000 προσθέσεις ανά δευτερόλεπτο, ένας σημερινός υπολογιστής μπορεί να εκτελέσει δισεκατομμύρια τέτοιες πράξεις στον ίδιο χρόνο. Έτσι, ξεκίνησε ένας αγώνας ταχύτητας με σκοπό την κατασκευή όλο και πιο γρήγορων υπολογιστών, δημιουργώντας όλο και πιο γρήγορους επεξεργαστές. Η εξέλιξη αυτή, όμως, προσέκρουσε σε έναν τοίχο: ο άνθρωπος πλέον περιοριζόταν από τους νόμους της φύσης και τα όρια του υλικού και έτσι θα έπρεπε να βρει νέους τρόπους.

Οι προσπάθειες για βελτίωση άρχισαν να επικεντρώνονται κυρίως στη βελτίωση της αρχιτεκτονικής. Η πλέον διαδεδομένη τεχνική είναι η παραλληλοποίηση στο επίπεδο των επεξεργαστών. Χρησιμοποιώντας περισσότερους από έναν επεξεργαστές μπορούμε να αυξήσουμε την επεξεργαστική ισχύ μέχρι και N φορές (όπου N 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. Ο αριθμός των πυρήνων που συναντάμε στους επεξεργαστές όλο και αυξάνεται. Πλέον πολυπύρηνοι επεξεργαστές χρησιμοποιούνται ευρέως και σε μικρότερα συστήματα όπως τα smartphones 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 την αυξημένη υπολογιστική ισχύ. Αυτό καθιστά τον παράλληλο προγραμματισμό απαραίτητο στη σημερινή εποχή.

1.2 Αρχιτεκτονικές παράλληλων συστημάτων

Όταν μιλάμε για παράλληλα συστήματα ουσιαστικά αναφερόμαστε σε συστήματα με πολλές επεξεργαστικές μονάδες (CPUs) που συνεργάζονται και επικοινωνούν για

την επίτευξη του τελικού αποτελέσματος. Οι επεξεργαστές δηλαδή αναλαμβάνουν διάφορα μέρη του προγράμματος και στη συνέχεια επικοινωνούν για να συγκεντρώσουν και να συνδυάσουν τα επιμέρους αποτελέσματα ώστε να συμπληρώσουν την τελική λύση.

Υπάρχουν δύο βασικές κατηγορίες παράλληλων αρχιτεκτονικών: τα συστήματα κοινής μνήμης και τα συστήματα κατανεμημένης μνήμης. Ανάλογα με το σύστημα που διαθέτει, ο προγραμματιστής θα πρέπει να ακολουθήσει διαφορετικές προγραμματιστικές λογικές και μεθόδους. Το ποιο είναι πιο αποδοτικό εξαρτάται ως ένα βαθμό και από το εκάστοτε πρόβλημα.

1.2.1 Συστήματα κοινής μνήμης (Shared memory)

Στα συστήματα κοινής μνήμης οι επεξεργαστές διασυνδέονται μεταξύ τους μέσω της κύριας μνήμης, η οποία είναι άμεσα προσπελάσιμη από όλους. Οι επεξεργαστές έχουν πρόσβαση σε ένα κοινό χώρο διευθύνσεων, από όπου μπορούν να μάθουν τυχόν αλλαγές σε μεταβλητές που επιθυμούν να χρησιμοποιήσουν. Με άλλα λόγια, οι επεξεργαστές επικοινωνούν μεταξύ τους τροποποιώντας κοινές μεταβλητές που είναι αποθηκευμένες στην κοινόχρηστη μνήμη και άρα ορατές σε όλους τους επεξεργαστές. Αυτό συνήθως επιτυγχάνεται με τη χρήση ενός διαύλου.

Στην περίπτωση που έχουμε μικρό αριθμό επεξεργαστών, το σύστημα μπορεί να είναι αποδοτικό. Αν όμως αυξήσουμε τον αριθμό των επεξεργαστών, θα αυξηθεί και ο αριθμός των επικοινωνιών, αλλά η χωρητικότητα του διαύλου μένει σταθερή με αποτέλεσμα η απόδοση να πέφτει, αφού ένα μεγάλο μέρος του χρόνου της εκτέλεσης θα καταναλώνεται μόνο στην επικοινωνία με την μνήμη. Επομένως, δεν μπορούμε να έχουμε τέτοια συστήματα με μεγάλο αριθμό επεξεργαστών. Αυτό έχει ως αποτέλεσμα να καταφύγουμε σε άλλου τύπου διασυνδέσεις, όπως για παράδειγμα τα διακοπτικά δίκτυα πολλαπλών επιπέδων.

Η πιο διαδεδομένη βιβλιοθήκη για τον προγραμματισμό κοινού χώρου διευθύνσεων είναι τα νήματα Posix (Pthreads), τα οποία παρέχουν κλήσεις για τη δημιουργία και το συντονισμό νημάτων. Ο προγραμματιστής πρέπει να λάβει υπόψιν του τις παράλληλες προσπελάσεις στην μνήμη και να χρησιμοποιήσει μηχανισμούς προστασίας, όπως για παράδειγμα κλειδαριές αμοιβαίου αποκλεισμού, για να διασφαλίσει την ακεραιότητα των δεδομένων και την ορθότητα του προγράμματος. Μία άλλη πλατφόρμα που διευκολύνει πολύ τη δημιουργία παράλληλων προγραμμάτων είναι

το OpenMP. Χρησιμοποιώντας το OpenMP ο προγραμματιστής μπορεί να μετατρέψει ένα σειριακό πρόγραμμα σε παράλληλο με τη χρήση μερικών απλών οδηγιών (directives) χωρίς να χρειάζεται να ασχοληθεί ιδιαίτερα με το συντονισμό των νημάτων και την προστασία των κοινόχρηστων μεταβλητών, αφού αυτά τα αναλαμβάνει σε μεγάλο βαθμό αυτόματα το OpenMP.

1.2.2 Συστήματα κατανεμημένης μνήμης (Distributed memory)

Στα συστήματα κατανεμημένης μνήμης, ο κάθε επεξεργαστής διαθέτει τη δική του ιδιωτική μνήμη και η επικοινωνία με τους άλλους επεξεργαστές πραγματοποιείται κάνοντας χρήση ενός δικτύου. Κάθε επεξεργαστής μπορεί άμεσα να προσπελάσει την ιδιωτική του μνήμη, δηλαδή να επεξεργαστεί και να τροποποιήσει τα δεδομένα της, αλλά δεν έχει πρόσβαση στην ιδιωτική μνήμη των υπόλοιπων επεξεργαστών. Στην περίπτωση που επιθυμεί να προσπελάσει δεδομένα που βρίσκονται εκεί, θα χρειαστεί να γίνει επικοινωνία μέσω μηνυμάτων, μεταξύ αυτών των δύο επεξεργαστών.

Ένα τέτοιο σύστημα μπορεί να είναι περισσότερο κλιμακώσιμο σε σχέση με ένα σύστημα κοινής μνήμης, με την έννοια ότι η εισαγωγή πολύ περισσότερων κόμβων και επεξεργαστών είναι εφικτή, ανάλογα πάντα και με την τοπολογία του δικτύου διασύνδεσης. Το πρόβλημα που μπορεί να προκύψει εδώ είναι η αύξηση του φόρτου του δικτύου. Αν πολλά δεδομένα πρέπει να είναι προσπελάσιμα από πολλές διεργασίες τότε ο φόρτος επικοινωνίας μπορεί να γίνει σημαντικός και να έχει αρνητικές επιπτώσεις στο χρόνο εκτέλεσης του προγράμματος.

Το πιο διαδεδομένο πρότυπο για προγραμματισμό συστημάτων κατανεμημένης μνήμης είναι το MPI [1]. Το MPI προσφέρει έτοιμες ρουτίνες για την αποστολή και λήψη μηνυμάτων, καθώς και πληθώρα άλλων λειτουργιών, χωρίς να αναγκάζει τον χρήστη να γνωρίζει τα πάντα για την αρχιτεκτονική του δικτύου που χρησιμοποιείται για τη διασύνδεση των επεξεργαστών στο εσωτερικό του συστήματος. Μπορεί να προσφέρει σημαντικές επιδόσεις αλλά θεωρείται από πολλούς σχετικά "χαμηλού" επιπέδου.

Στην πράξη, τα συστήματα που συναντάμε στις μέρες μας είναι συνήθως συνδυασμός των δύο κατηγοριών: συστήματα κοινής μνήμης, τα οποία αποτελούν τις επεξεργαστικές οντότητες για συστήματα κατανεμημένης μνήμης. Για παράδειγμα, οι σύγχρονες συστάδες (clusters) είναι ένα σύστημα κατανεμημένης μνήμης που

στηρίζεται σε ένα δίκτυο διασύνδεσης που ενώνει ανεξάρτητους επεξεργαστικούς κόμβους. Κάθε κόμβος όμως είναι ένα μικρό σύστημα κοινής μνήμης (π.χ. με πολυπύρηνους επεξεργαστές).

1.3 Αντικείμενο διπλωματικής εργασίας

Το αντικείμενο της παρούσας διπλωματικής εργασίας είναι η επέκταση του μεταφραστή OMPi ώστε οι κόμβοι ενός cluster να υποστηρίζονται ως συσκευές (devices) του OpenMP 4.0. Το OpenMP (Κεφάλαιο 2) από την έκδοση 4.0 και μετά υποστηρίζει την εκτέλεση τμημάτων κώδικα που επισημαίνει ο προγραμματιστής σε συσκευές/επιταχυντές (devices/accelerators) πέραν της βασικής CPU του συστήματος. Τέτοιες συσκευές/επιταχυντές είναι, για παράδειγμα, οι κάρτες γραφικών. Το πρότυπο επισημαίνει ότι δεν είναι απαραίτητη η ύπαρξη κοινής μνήμης ανάμεσα στο κύριο σύστημα (host) και τη συσκευή. Η εκτέλεση του προγράμματος ξεκινά στην ή στις CPUs του συστήματος. Όταν ένα νήμα συναντά μια κατάλληλα επισημασμένη περιοχή κώδικα, επικοινωνεί με την αντίστοιχη συσκευή ώστε να εκτελεστούν εκεί οι εντολές που την αποτελούν. Στις περισσότερες περιπτώσεις, είναι απαραίτητη και η αποστολή και λήψη δεδομένων (μεταβλητών ή πινάκων) από και προς τη συσκευή ώστε να επιτευχθεί ο εκάστοτε υπολογισμός. Μόλις ο υπολογισμός ολοκληρωθεί, η συσκευή ενημερώνει το νήμα στη βασική CPU το οποίο συνεχίζει τη δουλειά του.

Ο OMPi (Κεφάλαιο 3) είναι ένας παραλληλοποιητικός μεταφραστής source–to–source, ο οποίος δέχεται ως είσοδο ένα σειριακό πρόγραμμα που περιέχει εντολές OpenMP, και εξάγει ένα πρόγραμμα σε C, παραλληλοποιημένο αποδοτικά και έτοιμο να μεταφραστεί στο εκάστοτε σύστημα από τον τοπικό μεταφραστή. Ήδη υποστηρίζει το πρότυπο OpenMP 4.0 καθώς και κάποιες συσκευές, όπως είναι για παράδειγμα το Parallella Board [2]. Στόχος της εργασίας, λοιπόν, είναι:

- Η υποστήριξη μίας νέας συσκευής από τον OMPi, η οποία αναπαριστά ένα κόμβο σε ένα cluster. Για να επιτευχθεί αυτό, απαιτήθηκαν αλλαγές στην αρχιτεκτονική του API που παρέχει ο OMPi για τη διαχείριση συσκευών.
- Η ανάπτυξη και χρονομέτρηση προγραμμάτων (benchmarks) ώστε να αναλυθεί η απόδοση και η χρησιμότητα μίας τέτοιας συσκευής, καθώς και να προσδιοριστούν οι περιορισμοί και οι αδυναμίες της.

1.4 Δομή διπλωματικής εργασίας

Στη συνέχεια ακολουθεί μια περιληπτική περιγραφή της δομής των επόμενων κεφαλαίων:

- Κεφάλαιο 2: Παρουσίαση και περιγραφή του προτύπου OpenMP για παράλληλο προγραμματισμό σε μοντέλο κοινού χώρου διευθύνσεων. Γίνεται επίσης ανάλυση των εντολών που είναι απαραίτητες για την εκτέλεση κώδικα σε συσκευές.

- Κεφάλαιο 3: Περιγραφή του παραλληλοποιητικού μεταφραστή OMPi και ανάλυση του τρόπου λειτουργίας του για την εκτέλεση κώδικα σε συσκευές.

- Κεφάλαιο 4: Αναλυτική παρουσίαση της νέας μονάδας (module) που δημιουργήσαμε για να κάνουμε δυνατή την παράλληλη εκτέλεση κώδικα στους κόμβους ενός cluster. Δίνεται έμφαση στα κομβικά σημεία της υλοποίησης που παρουσιάζουν ενδιαφέρον.

- Κεφάλαιο 5: Περιγραφή της υλοποίησης πειραματικών προγραμμάτων (benchmarks) και ανάλυση των αποτελεσμάτων από την εκτέλεσή τους. Αποτίμηση τους κέρδους που έχουμε από τη χρήση της νέας συσκευής, καθώς και τους περιορισμούς της.

- Κεφάλαιο 6: Επισκόπηση της διπλωματικής εργασίας και προτάσεις για μελλοντικές βελτιώσεις σχετικά με την υποστήριξη κόμβων ενός cluster ως συσκευές στον OMPi.

Κεφαλαιο 2

Το προτυπο OpenMP

2.1 Εισαγωγή στο OpenMP

Η συνεχώς αυξανόμενη χρήση των παράλληλων συστημάτων, κυρίως των συστημάτων κοινής μνήμης, είχε ως αποτέλεσμα νέες απαιτήσεις για τους προγραμματιστές τους. Πιο συγκεκριμένα, ο προγραμματιστής θα πρέπει να δώσει ιδιαίτερη προσοχή στη διαχείριση των θεμελιωδών συστατικών του προγράμματος (όπως συναρτήσεις και μεταβλητές) αφού αυτά τώρα προσπελάζονται ταυτόχρονα από περισσότερες οντότητες και πρέπει να προστατευθούν για να είναι το αποτέλεσμα συνεπές. Είναι επίσης γεγονός ότι ένα παράλληλο πρόγραμμα αποτελείται από πολύ μεγαλύτερο όγκο εντολών από ότι ένα σειριακό πρόγραμμα που κάνει την ίδια δουλειά. Εξαιτίας αυτών των δυσκολιών, προέκυψε η ανάγκη για τη δημιουργία ενός προτύπου για τη διευκόλυνση της συγγραφής παράλληλων προγραμμάτων. Το πρότυπο αυτό ονομάζεται OpenMP (Open Multi-Processing).

Το OpenMP είναι μια διεπαφή δημιουργίας παράλληλων εφαρμογών σε συστήματα κοινής μνήμης. Υποστηρίζει τις γλώσσες προγραμματισμού C, C++ και Fortran. Αποτελείται από:

- Οδηγίες προς το μεταγλωττιστή: τα λεγόμενα pragmas που εισάγει ο προγραμματιστής στον κώδικά του για να υποδείξει ποια σημεία θέλει να παραλληλοποιήσει και πώς.

- Συναρτήσεις βιβλιοθήκης: σύνολο συναρτήσεων που μπορεί να καλέσει κατευθείαν ο προγραμματιστής για να καθορίσει ή να ζητήσει να μάθει τον αριθμό των νημάτων που θα εκτελέσουν μία παράλληλη περιοχή, να χρησιμοποιήσει τις κλειδαριές του OpenMP κ.α.

- Μεταβλητές περιβάλλοντος: θέτουν τις προεπιλεγμένες τιμές για διάφορες μεταβλητές που καθορίζουν τον τρόπο εκτέλεσης του προγράμματος.

Η ανάπτυξη ενός προγράμματος χρησιμοποιώντας το OpenMP θα μπορούσαμε να ισχυριστούμε ότι είναι κλιμακωτή διαδικασία. Ο προγραμματιστής μπορεί να πάρει ένα σειριακό πρόγραμμα και εισάγοντας οδηγίες OpenMP σε αυτό, σταδιακά να το κάνει παράλληλο χωρίς μεγάλη δυσκολία. Βέβαια, η απόκρυψη των λεπτομερειών της παραλληλίας και της διαχείρισης των νημάτων από τον προγραμματιστή έχει και μειονεκτήματα. Το OpenMP δεν μπορεί να εξασφαλίσει τη μέγιστη αποδοτικότητα της παραλληλίας σε συστήματα κοινής μνήμης και επιπλέον δεν είναι εφικτή η διαχείριση ενός υποσυνόλου των νημάτων από αυτά που θα εκτελέσουν ένα τμήμα του κώδικα. Παρόλα αυτά, στα θετικά του OpenMP θα μπορούσαμε να αναφέρουμε τη δυνατότητα υποστήριξης δυναμικού ορισμού του αριθμού των νημάτων που θα εκτελέσουν μια παράλληλη περιοχή κατά τη διάρκεια εκτέλεσης του προγράμματος καθώς και την υποστήριξη λεπτού και αδρού καταμερισμού του παραλληλισμού.

2.2 Προγραμματιστικό μοντέλο OpenMP

Το προγραμματιστικό μοντέλο του OpenMP είναι βασισμένο σε παραλληλισμό με νήματα. Τα νήματα αυτά επικοινωνούν τροποποιώντας κοινές μεταβλητές. Η δημιουργία και η λειτουργία τους είναι πολύ κοντά στην λογική *fork-join* που συναντάμε στις διεργασίες.

Αρχικά το πρόγραμμα ξεκινάει να εκτελείται με ένα μόνο νήμα. Όταν αυτό συναντήσει μια περιοχή που ο προγραμματιστής έχει ζητήσει να παραλληλοποιηθεί, δημιουργεί μια ομάδα νημάτων των οποίων ο αριθμός ορίζεται είτε δυναμικά, είτε

στατικά. Αυτά με την σειρά τους θα εκτελέσουν το τμήμα παραλληλίας που τους αντιστοιχεί. Το αρχικό νήμα περιμένει στο τέλος της παράλληλης περιοχής (που αποτελεί σημείο συντονισμού) μέχρι τα νήματα που δημιούργησε να τελειώσουν τη δουλειά τους, οπότε καταστρέφεται και η ομάδα. Στη συνέχεια, το αρχικό νήμα εκτελεί τον κώδικα που ακολουθεί σειριακά. Τα νήματα, δηλαδή, συγχρονίζονται και επικοινωνούν σε ένα πρόγραμμα OpenMP αυτόματα. Βέβαια, η κοινή χρήση των πόρων απαιτεί και την προστασία αυτών, αλλά το OpenMP προσφέρει πληθώρα οδηγιών και παραμέτρων για να διευκολύνει όσο γίνεται τη διαδικασία αυτή.

Λόγω της παραπάνω λογικής που ακολουθείται από το αρχικό νήμα για τη δημιουργία νημάτων, η συγγραφή ενός OpenMP προγράμματος θα μπορούσαμε να πούμε ότι είναι κλιμακωτή. Γράφουμε το παράλληλο πρόγραμμα αρχικά όπως θα γράφαμε ένα αντίστοιχο σειριακό και στη συνέχεια προσθέτουμε τις οδηγίες στις περιοχές που θέλουμε να παραλληλοποιήσουμε. Έτσι, βαθμιαία το πρόγραμμα από σειριακό μετατρέπεται σε παράλληλο με χρήση νημάτων.

2.3 Οδηγίες OpenMP για C/C++

Σε αυτή την ενότητα θα ασχοληθούμε με τις οδηγίες του OpenMP [3] για τις γλώσσες C/C++. Θα δούμε συνοπτικά την οδηγία *parallel*, που κατά πάσα πιθανότητα αποτελεί την πιο γνωστή και πολύ χρησιμοποιημένη οδηγία του OpenMP. Στη συνέχεια, θα μελετήσουμε την οδηγία *task*, που χρησιμοποιείται κατά κόρον στα πειράματα επίδοσης (benchmarks) που δοκιμάσαμε. Τέλος, θα αναλύσουμε τη σύνταξη και τις πιο συχνές παραμέτρους για το υποσύνολο των οδηγιών που μας επιτρέπουν να στείλουμε κώδικα και μεταβλητές σε άλλες συσκευές, μιας και αυτό είναι το αντικείμενο της παρούσας εργασίας.

2.3.1 Γενική σύνταξη οδηγιών στο OpenMP

Μία οδηγία στο OpenMP αποτελείται από 3 μέρη:

1. Θα πρέπει να ξεκινά υποχρεωτικά με το `#pragma omp`.

2. Ακολουθεί το όνομα της οδηγίας. Υπάρχουν οδηγίες για τη δημιουργία παράλληλης ομάδας, επίτευξη συγχρονισμού μεταξύ των νημάτων, αποστολή δεδομένων και εκτέλεση κώδικα σε συσκευές και πολλές άλλες.

3. Προαιρετικά, μπορεί να υπάρχουν φράσεις οδηγιών, που ρυθμίζουν τις παραμέτρους των οδηγιών. Για παράδειγμα, στην οδηγία της παράλληλης εκτέλεσης μερικές φράσεις οδηγιών είναι το πλήθος των νημάτων που θα χρησιμοποιηθούν, ποιες μεταβλητές είναι κοινόχρηστες και χρειάζονται προστασία, ποιες είναι ιδιωτικές κ.α.

2.3.2 Η οδηγία parallel

Η οδηγία `#pragma omp parallel` χρησιμοποιείται για τον ορισμό μίας παράλληλης περιοχής. Παράλληλη περιοχή είναι ένα τμήμα κώδικα που θα εκτελεστεί από πολλαπλά νήματα. Όταν ένα νήμα συναντήσει αυτή την οδηγία, δημιουργεί μία ομάδα νημάτων, της οποίας το μέγεθος, δηλαδή το πλήθος των νημάτων που την αποτελούν, καθορίζεται από τον προγραμματιστή. Ο κώδικας της παράλληλης περιοχής εκτελείται αυτούσιος από όλα τα νήματα, ωστόσο η σχετική ταχύτητα κάθε νήματος είναι απρόβλεπτη, δηλαδή δεν μπορούμε να ξέρουμε με ποια σειρά τα νήματα θα τελειώσουν τη δουλειά που τους έχει ανατεθεί. Στο τέλος της παράλληλης περιοχής υπονοείται ένας *barrier*, εκτός και αν ο προγραμματιστής ζητήσει το αντίθετο. Με άλλα λόγια, πρώτα όλα τα νήματα τελειώνουν τη δουλειά τους και στη συνέχεια καταστρέφονται *όλα*, πλην του αρχικού, που συνεχίζει μετά το τέλος της παράλληλης περιοχής. Αυτό αποκτά περισσότερη σημασία στην περίπτωση που έχουμε φωλιασμένες οδηγίες *parallel*.

2.3.3 Η οδηγία task

Η οδηγία `#pragma omp task` ορίζει μία εργασία ευέλικτου τύπου. Όταν ένα νήμα τη συναντήσει, "πακετάρει" τον κώδικα που την αποτελεί και τις μεταβλητές που ο προγραμματιστής δηλώνει ότι χρειάζεται σε μία νέα εργασία. Η εκτέλεσή της μπορεί να ξεκινήσει αμέσως ή να καθυστερήσει λιγάκι και το σύστημα χρόνου εκτέλεσης είναι δυνατόν να την εκτελέσει παράλληλα με κώδικα που δεν ανήκει σε task. Επίσης, είναι δυνατή η εναλλαγή της εκτέλεσης ενός task μεταξύ νημάτων, που σημαίνει ότι ένα νήμα μπορεί να ξεκινήσει να εκτελεί το task και μετά από ένα σημείο να αναλάβει την εκτέλεση ένα άλλο νήμα. Οδηγίες `#pragma omp task` μπορεί να βρίσκονται και φωλιασμένες η μία μέσα στην άλλη.

Κάποιες από τις μεταβλητές του task ενδέχεται να χρειάζεται να λαμβάνουν αρχικές τιμές πριν αρχίσει η εκτέλεση του task ή να είναι το αποτέλεσμα των υπο-

λογισμών που πρέπει να μεταφερθεί σε μεταβλητές που βρίσκονται εκτός του task. Ο προγραμματιστής μπορεί να επισημάνει τέτοιες μεταβλητές χρησιμοποιώντας τις ακόλουθες φράσεις στην οδηγία `#pragma omp task`:

- *shared (λίστα μεταβλητών)*: Αυτές οι μεταβλητές είναι κοινόχρηστες μεταξύ των νημάτων, δηλαδή τροποποιήσεις στις τιμές τους που γίνονται μέσα από το task "φαίνονται" στον κώδικα που βρίσκεται εκτός του task και το αντίθετο.

- *private (λίστα μεταβλητών)*: Αυτές οι μεταβλητές είναι ιδιωτικές σε κάθε task, δηλαδή τροποποιήσεις στις τιμές τους που γίνονται μέσα από το task δε "φαίνονται" στον κώδικα που βρίσκεται εκτός του task.

- *firstprivate (λίστα μεταβλητών)*: Αυτές οι μεταβλητές είναι ιδιωτικές σε κάθε task, επομένως ισχύει ό,τι προηγουμένως. Επιπλέον, αρχικοποιούνται με την τιμή της αρχικής μεταβλητής.

Οι μεταβλητές μέσα στη λίστα μεταβλητών διαχωρίζονται με κόμμα (,).

Το πρότυπο OpenMP παρέχει δύο οδηγίες που βοηθούν στο συγχρονισμό της εκτέλεσης των tasks. Πρώτον, η οδηγία `#pragma omp barrier` εξασφαλίζει ότι η εκτέλεση όλων των tasks που δημιουργήθηκαν από οποιοδήποτε νήμα της τρέχουσας ομάδας νημάτων έχει ολοκληρωθεί μόλις εκτελεστεί η οδηγία. Δεύτερον, ένα task που συναντά την οδηγία `#pragma omp taskwait` σταματά προσωρινά την εκτέλεσή του, μέχρι την ολοκλήρωση της εκτέλεσης όλων των task παιδιών του, δηλαδή των tasks που έχει δημιουργήσει (αφορά μόνο τα άμεσα παιδιά και όχι τους υπόλοιπους απογόνους).

2.3.4 Η οδηγία target

Η `#pragma omp target` είναι η οδηγία που μας επιτρέπει να εκτελέσουμε κώδικα σε συσκευή. Οι εντολές του προγράμματος που υπάρχουν μέσα στο μπλοκ κώδικα που την ακολουθεί εκτελούνται στη συσκευή και όχι στον κύριο επεξεργαστή.

Επίσης, η οδηγία μας επιτρέπει να ορίσουμε μία αντιστοίχιση ανάμεσα στις μεταβλητές του κύριου επεξεργαστή (host) και της συσκευής (device). Έτσι, μπορούμε να μεταφέρουμε δεδομένα από και προς μία συσκευή με τους ακόλουθους 3 τρόπους:

- Με τον προσδιορισμό *to* η τιμή μιας μεταβλητής αντιγράφεται από το βασικό σύστημα στη συσκευή *πριν* την εκτέλεση του τμήματος κώδικα. Είναι χρήσιμο για αρχικοποίηση δεδομένων.

- Με τον προσδιορισμό *from* η τιμή μιας μεταβλητής αντιγράφεται από τη συσκευή στο βασικό σύστημα *μετά* την εκτέλεση του τμήματος κώδικα. Είναι χρήσιμο για να πάρουμε πίσω τα αποτελέσματα.

- Με τον προσδιορισμό *tofrom* επιτυγχάνουμε το συνδυασμό των 2 παραπάνω επιλογών. Είναι χρήσιμο όταν στέλνουμε δεδομένα των οποίων η τιμή θα χρησιμοποιηθεί και θα τροποποιηθεί και χρειαζόμαστε την τελική τιμή.

- Με τον προσδιορισμό *alloc* δημιουργείται μια αντιστοιχία δεδομένων ανάμεσα σε βασικό σύστημα και συσκευή, αλλά δεν πραγματοποιείται καμία μεταφορά δεδομένων. Είναι χρήσιμο για βοηθητικά δεδομένα που δεν έχουν προκαθορισμένη αρχική τιμή, ούτε και χρειαζόμαστε την τελική τους τιμή.

Για να γίνουν τα παραπάνω πιο κατανοητά, ας θεωρήσουμε το ακόλουθο παράδειγμα. Ο κώδικας στο Πρόγραμμα 2.1 εκτελεί πρόσθεση πινάκων σε μία συσκευή. Αρχικά, οι μεταβλητές `a`, `b`, `c` και `size` βρίσκονται μόνο στη μνήμη του βασικού μηχανήματος. Μόλις συναντηθεί η οδηγία *target* δεσμεύεται χώρος στη μνήμη της συσκευής για τις μεταβλητές `a`, `b` και `c` μεγέθους size στοιχείων ξεκινώντας από τη θέση 0, εξαιτίας του `[0:size]`. Επίσης δεσμεύεται χώρος για τη μεταβλητή `size`. Τα δεδομένα των πινάκων `a` και `b` καθώς και της μεταβλητής `size` που είναι επισημασμένα με την επιλογή *to* αντιγράφονται από τη μνήμη του βασικού υπολογιστή στη μνήμη της συσκευής.

Πρόγραμμα 2.1: Πρόσθεση πινάκων σε συσκευή.

```
1  float a[1024];
2  float b[1024];
3  float c[1024];
4  int size;
5
6  void add(float *a, float *b, float *c, int size)
7  {
8    #pragma omp target map(to:a[0:size],b[0:size],size) map(from: c[0:size])
9    {
10     int i;
```

```
11      #pragma omp parallel for
12      for (i = 0; i < size; i++)
13        c[i] = a[i] + b[i];
14    }
15  }
```

Στη συνέχεια, η πρόσθεση πινάκων εκτελείται στη συσκευή. Η οδηγία `#pragma omp parallel for` χρησιμοποιείται για την κατανομή των επαναλήψεων του βρόγχου for στους πυρήνες του επεξεργαστή της συσκευής. Έτσι πετυχαίνουμε καλύτερη χρησιμοποίηση των πόρων της συσκευής. Μόλις ο υπολογισμός της πρόσθεσης ολοκληρωθεί, τα δεδομένα του πίνακα `c` που είναι επισημασμένα με την επιλογή *from* αντιγράφονται από τη μνήμη της συσκευής στη μνήμη του κύριου επεξεργαστή.

2.3.5 Η οδηγία declare target

Για να χρησιμοποιήσουμε καθολικές (global) μεταβλητές και για να καλέσουμε συναρτήσεις μέσα από κώδικα που εκτελείται σε συσκευή, θα πρέπει πρώτα να τις έχουμε επισημάνει. Αυτό γίνεται πολύ εύκολα δηλώνοντας τις μεταβλητές και τα πρωτότυπα των συναρτήσεων ανάμεσα στις οδηγίες `#pragma omp declare target` και `#pragma omp end declare target`. Στο Πρόγραμμα 2.2 δηλώνουμε ότι θα χρησιμοποιήσουμε τη συνάρτηση `myfunc` και την καθολική μεταβλητή `myglobal` στον κώδικα που εκτελείται στη συσκευή.

Πρόγραμμα 2.2: Χρήση της οδηγίας declare target.

```
1 #pragma omp declare target
2 int myfunc(void);
3 float myglobal = 1.5;
4 #pragma omp end declare target
```

2.3.6 Η οδηγία target data

Με την οδηγία `#pragma omp target data` ορίζουμε μία περιοχή μετακίνησης δεδομένων από και προς τη συσκευή. Κώδικας μπορεί να εκτελεστεί αν υπάρχει οδηγία `#pragma omp target` φωλιασμένα. Στη διάθεσή του θα έχει και τις μεταβλητές που ορίστηκαν πριν με την οδηγία `#pragma omp target data`, οι οποίες θα είναι έγκυρες και διαθέσιμες για όλο το μπλοκ κώδικα της οδηγίας. Ο λόγος ύπαρξης αυτής της

οδηγίας είναι να βοηθήσει στη μείωση του όγκου των δεδομένων που μεταφέρονται από και προς τη συσκευή όταν υπάρχουν διαδοχικές οδηγίες `#pragma omp target` που χρησιμοποιούν τα ίδια δεδομένα. Οδηγίες `#pragma omp target data` μπορούν να εμφανίζονται και φωλιασμένα η μία μέσα στην άλλη.

Πρόγραμμα 2.3: Χρήση της οδηγίας target data.

```
1 #pragma omp target data map(to:a[0:size], b[0:size]) map(from:c[0:size])
2 {
3   #pragma omp target
4   {
5     int i;
6     #pragma omp parallel for
7     for (i = 0; i < size; i++)
8       c[i] += a[i] + b[i];
9   }
10 }
```

Στο Πρόγραμμα 2.3 βλέπουμε ξανά το πρόγραμμα πρόσθεσης δύο πινάκων σε συσκευή, που παρουσιάστηκε πρώτα στο Πρόγραμμα 2.1, αυτή τη φορά με χρήση της οδηγίας `#pragma omp target data`. Στη γραμμή 1 δεσμεύεται χώρος στη συσκευή για τους πίνακες `a`, `b` και `c`, ενώ γίνεται και μεταφορά δεδομένων των πινάκων `a` και `b` από το κύριο σύστημα στη συσκευή. Στη γραμμή 3 ορίζεται μία περιοχή κώδικα που πρέπει να εκτελεστεί στη συσκευή. Δεν γίνεται μεταφορά δεδομένων σε αυτό το σημείο, αφού οι πίνακες `a`, `b` και `c` βρίσκονται ήδη στη συσκευή. Τέλος, στη γραμμή 10 τα δεδομένα του πίνακα `c` μεταφέρονται από τη συσκευή στο κύριο σύστημα.

2.3.7 Η οδηγία target update

Όπως έχουμε δει μέχρι τώρα, μεταφορά δεδομένων από ή προς τη συσκευή μπορεί να πραγματοποιηθεί μόνο στην αρχή ή μόνο στο τέλος της οδηγίας `#prgma omp target`. Με την οδηγία `#pragma omp target update` επιτυγχάνουμε συγχρονισμό δεδομένων με το κύριο σύστημα ή τη συσκευή οποιαδήποτε στιγμή το επιθυμούμε.

Πρόγραμμα 2.4: Χρήση της οδηγίας target update.

```
1 #pragma omp target data map(to:a) map(from:b)
2 {
```

```
3   #pragma omp target
4   {
5     /* Device modifies variable b... */
6   }
7
8   #pragma omp target update from(b)
9
10  /* Host needs to perform some processing on variable b... */
11
12  #pragma omp target update to(b)
13
14  #pragma omp target
15  {
16    /* Device modifies variable b... */
17  }
18 }
```

Ας δούμε για παράδειγμα το Πρόγραμμα 2.4. Στη γραμμή 1 δεσμεύεται χώρος για τις μεταβλητές `a` και `b` στη συσκευή και αντιγράφονται τα περιεχόμενα της μεταβλητής `a`. Στη γραμμή 5 υποθέτουμε ότι η συσκευή κάνει ενέργειες που θα επηρεάσουν την τιμή της μεταβλητής `b`. Στη γραμμή 8 μεταφέρουμε την αλλαγμένη μεταβλητή `b` στο κύριο σύστημα, το οποίο υποθέτουμε ότι την επεξεργάζεται με εντολές στη γραμμή 10. Στη γραμμή 12 μεταφέρουμε ξανά την αλλαγμένη μεταβλητή `b` στη συσκευή, η οποία υποθέτουμε ότι την επηρεάζει με τις εντολές στη γραμμή 16. Τέλος, στη γραμμή 18 η τιμή της μεταβλητής `b` μεταφέρεται πίσω στο κύριο σύστημα.

2.4 Συναρτήσεις βιβλιοθήκης χρόνου εκτέλεσης και μεταβλητές περιβάλλοντος

Το OpenMP παρέχει στον προγραμματιστή μία σειρά συναρτήσεων που επιτρέπουν την παραμετροποίηση των οδηγιών του, ενδεχομένως και δυναμικά. Για να μπορέσει να τις χρησιμοποιήσει, θα πρέπει να έχει κάνει πρώτα `#include <omp.h>` στον κώδικά του. Μερικές παραμετροποιήσεις μπορούν να γίνουν και στατικά, πριν την εκτέλεση του προγράμματος, θέτοντας τις κατάλληλες μεταβλητές περιβάλλοντος. Στις επόμενες ενότητες θα εξετάσουμε τις συναρτήσεις και μεταβλητές περιβάλλοντος που

αφορούν την εκτέλεση κώδικα σε συσκευές, μιας και αυτό είναι το αντικείμενο της παρούσας εργασίας.

Πριν όμως δούμε αυτό, θα απαντήσουμε το ακόλουθο ερώτημα: Αν ο προγραμματιστής έχει στη διάθεσή του πάνω από μία συσκευές, πώς καθορίζει το ποια θα χρησιμοποιηθεί κάθε φορά; Η απάντηση είναι ότι ο εκάστοτε μεταφραστής (compiler) αντιστοιχίζει με αυθαίρετο τρόπο τις συσκευές σε διαδοχικούς ακέραιους. Για παράδειγμα, αν υποθέσουμε ότι ο προγραμματιστής διαθέτει δύο κάρτες γραφικών τις οποίες ο μεταφραστής μπορεί να αναγνωρίσει και να χρησιμοποιήσει ως συσκευές, θα δώσει στη μία τον αριθμό συσκευής 1 και στην άλλη τον αριθμό συσκευής 2. Σε όλες τις οδηγίες που είδαμε παραπάνω, μπορεί να δοθεί ως φράση οδηγίας ο αριθμός συσκευής στην οποία θα πραγματοποιηθεί η ενέργεια. Για παράδειγμα, αν υποθέσουμε ότι υπάρχει η συσκευή 2, η οδηγία `#pragma omp target device(2)` θα έχει ως αποτέλεσμα την εκτέλεση κώδικα στη συσκευή 2. Αν δεν υπάρχει η φράση οδηγίας *device*, η οδηγία θα εκτελεστεί στην *προεπιλεγμένη συσκευή (default device)*, που ορίζεται αυθαίρετα από τον κάθε μεταφραστή.

2.4.1 Συναρτήσεις βιβλιοθήκης χρόνου εκτέλεσης

Οι συναρτήσεις βιβλιοθήκης που αφορούν τη μεταφορά δεδομένων και την εκτέλεση κώδικα σε συσκευές είναι οι:

- `omp_set_default_device()`: Δέχεται ως όρισμα ένα θετικό ακέραιο και θέτει τη συσκευή που του αντιστοιχεί ως προεπιλεγμένη συσκευή.
- `omp_get_default_device()`: Επιστρέφει έναν ακέραιο που αντιστοιχεί στον αριθμό της προεπιλεγμένης συσκευής.
- `omp_get_num_devices()`: Επιστρέφει το πλήθος των συσκευών που έχουν αναγνωριστεί και μπορούν να χρησιμοποιηθούν.
- `omp_is_initial_device()`: Επιστρέφει true (1) αν κληθεί από κώδικα που εκτελείται στο κυρίως σύστημα και false (0) αν κληθεί από κώδικα που εκτελείται σε συσκευή.

2.4.2 Μεταβλητές περιβάλλοντος

Η προεπιλεγμένη συσκευή μπορεί να οριστεί και στατικά πριν την εκτέλεση του προγράμματος με τη μεταβλητή περιβάλλοντος `OMP_DEFAULT_DEVICE`. Η σύνταξη που ακολουθείται για τον ορισμό μεταβλητών περιβάλλοντος διαφέρει ανάλογα με το κέλυφος (shell) που είναι σε χρήση. Για παράδειγμα, για να εκτελέσουμε το πρόγραμμα `a.out` με προεπιλεγμένη τη συσκευή 2 (με την προϋπόθεση ότι η συσκευή 2 υπάρχει) σε κέλυφος bash θα γράφαμε: `OMP_DEFAULT_DEVICE=2 ./a.out`.

Κεφαλαιο 3

Ο παραλληλοποιητικος μεταφραστης OMPi

3.1 Επισκόπηση του OMPi

Ο OMPi [4] είναι ένας παραλληλοποιητικός μεταφραστής ανοιχτού κώδικα για προγράμματα που έχουν γραφτεί σε γλώσσα C και υποστηρίζει το πρότυπο OpenMP. Αποτελείται από δύο τμήματα:

- Το μεταφραστή (compiler) που δέχεται ως είσοδο προγράμματα γραμμένα σε γλώσσα προγραμματισμού C με εντολές OpenMP. Ο συντακτικός αναλυτής διατρέχει το πρόγραμμα και παράγει το συντακτικό δέντρο, το οποίο στη συνέχεια επεξεργάζεται κατάλληλα ώστε να αφαιρέσει τις οδηγίες OpenMP. Παράγει ως έξοδο προγράμματα σε C όπου οι οδηγίες OpenMP έχουν αντικατασταθεί με κλήσεις συναρτήσεων της βιβλιοθήκης χρόνου εκτέλεσης. Τέλος, ο τοπικός μεταφραστής C που είναι εγκατεστημένος στο σύστημα (για παράδειγμα ο gcc) αναλαμβάνει τη δημιουργία του τελικού εκτελέσιμου από τον

κώδικα C.

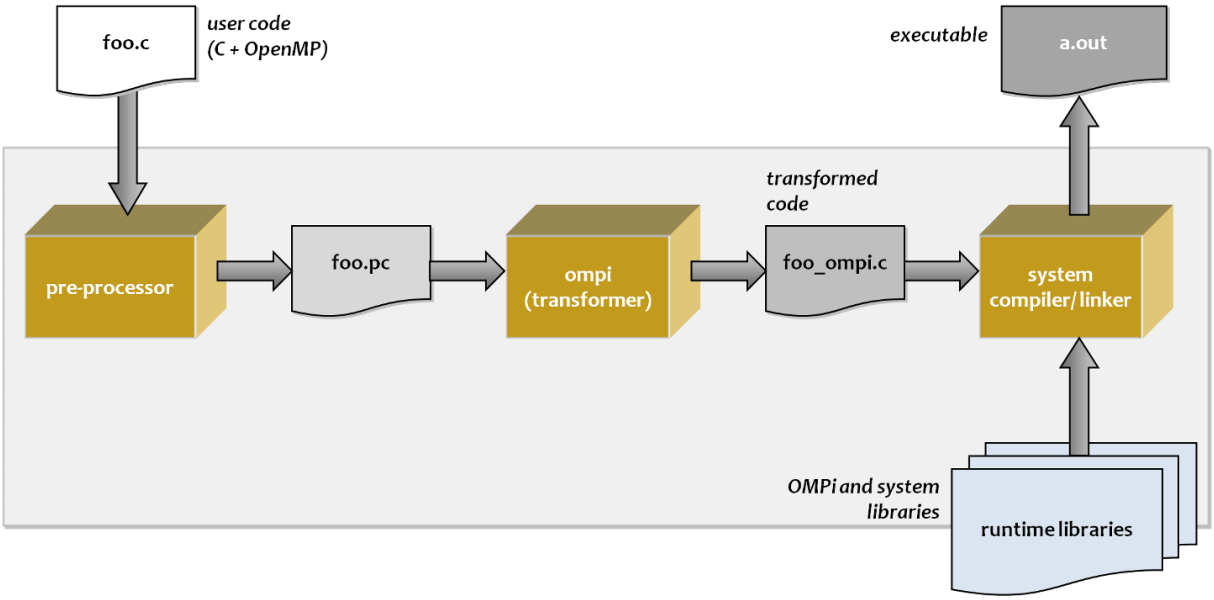
- Τη βιβλιοθήκη χρόνου εκτέλεσης που παρέχει συναρτήσεις για τη δημιουργία υπολογιστικών οντοτήτων (όπως νήματα και διεργασίες) που αναλαμβάνουν την εκτέλεση του κώδικα που έχει παραλληλοποιηθεί, συναρτήσεις που διαχειρίζονται κλειδαριές και άλλες βοηθητικές δυνατότητες, καθώς και συναρτήσεις που διαχειρίζονται την επικοινωνία με συσκευές και την εκτέλεση τμημάτων κώδικα σε αυτές.

Η διαδικασία της μετάφρασης φαίνεται γραφικά στο Σχήμα 3.1. Αξίζει να επισημάνουμε ότι οι εντολές που βρίσκονται μέσα σε μία οδηγία *target* εξάγονται σε *συναρτήσεις πυρήνα (kernel functions)*. Οι συναρτήσεις πυρήνα (που ουσιαστικά είναι ο κώδικας που πρέπει να εκτελεστεί στη συσκευή) βρίσκονται σε δύο σημεία ώστε η κάθε συσκευή να τις εκτελέσει με τον τρόπο που της είναι πιο εύκολος:

- Όλες οι συναρτήσεις πυρήνα βρίσκονται στο αρχείο `foo_ompi.c` του Σχήματος 3.1, το οποίο περιέχει όλο τον κώδικα του χρήστη. Αυτός είναι ο τρόπος που χρησιμοποιούμε στην περίπτωσή μας ώστε να στείλουμε το εκτελέσιμο αρχείο σε όλους τους κόμβους του cluster και ο καθένας να εκτελεί τη συνάρτηση πυρήνα που χρειάζεται κάθε φορά.

- Στο βήμα του μετασχηματισμού του κώδικα του χρήστη (ompi transformer στο Σχήμα 3.1) πέρα από το αρχείο `foo_ompi.c` δημιουργούνται και τα αρχεία `foo_d00.c`, `foo_d01.c`, κ.τ.λ. καθένα από τα οποία περιέχει μόνο μία συνάρτηση πυρήνα. Για κάποιες συσκευές (όπως η Parallella) είναι πιο εύκολο να μεταφράζονται και αυτά τα αρχεία μαζί με τον κώδικα του χρήστη και να αποστέλλονται στη συσκευή μικρότερα εκτελέσιμα που περιέχουν μόνο τον κώδικα που πρέπει να εκτελέσει κάθε φορά.

Στο υπόλοιπο του κεφαλαίου θα εξετάσουμε τις πτυχές της λειτουργίας του OMPi που αφορούν τη μεταφορά δεδομένων και την εκτέλεση κώδικα σε συσκευές, καθώς αυτό αποτελεί το αντικείμενο της παρούσας εργασίας. Επίσης, θα αναλύσουμε τη διεπαφή (API) που παρέχει σε όποιον θελήσει να υποστηρίξει ένα καινούργιο τύπο συσκευής.



Σχήμα 3.1: Η διαδικασία μετάφρασης με τον μεταφραστή OMPi.

3.2 Διευθύνσεις

Μερικές συσκευές εξ ορισμού μοιράζονται κάποια μνήμη με τον κεντρικό επεξεργαστή. Έτσι, δεδομένα που γράφονται σε αυτό το τμήμα μνήμης έχουν την ίδια διεύθυνση, είτε προσπελάζονται από τη CPU, είτε από τη συσκευή. Υπάρχουν, όμως, πιο πολύπλοκες περιπτώσεις. Ο OMPi είναι δομημένος ώστε να υποστηρίζει και συσκευές που δεν μοιράζονται καθόλου μνήμη με τον κεντρικό επεξεργαστή. Ένα παράδειγμα τέτοιων συσκευών είναι οι κόμβοι ενός cluster, που εμείς θέλουμε να υποστηρίξουμε.

Για να επιλύσει το πρόβλημα όπου μία μεταβλητή έχει διαφορετική διεύθυνση στη βασική CPU και διαφορετική στη συσκευή, ο OMPi χρησιμοποιεί *ενδιάμεσες διευθύνσεις*. Κάθε μεταβλητή που αποστέλλεται σε μία συσκευή, έχει τις ακόλουθες 3 διευθύνσεις:

1. Διεύθυνση στο βασικό επεξεργαστή (host address): είναι η διεύθυνση που είχε η μεταβλητή και πριν τη μεταφορά της σε κάποια συσκευή. Προφανώς είναι έγκυρη μόνο όταν χρησιμοποιείται από το βασικό επεξεργαστή.

2. Διεύθυνση στη συσκευή (device address): είναι η διεύθυνση που αποκτά η μεταβλητή στη συσκευή, μετά τη μεταφορά της. Προφανώς είναι έγκυρη μόνο όταν χρησιμοποιείται από τη συσκευή.

3. Ενδιάμεση διεύθυνση (mediary address): Είναι μία επιπλέον διεύθυνση που ο

OMPi δίνει σε κάθε μεταβλητή που μεταφέρεται σε συσκευή. Είναι απαραίτητη προκειμένου το κυρίως σύστημα και η συσκευή να συγχρονίζονται και να καταλαβαίνουν σε ποια μεταβλητή αναφέρονται κάθε φορά, μιας και όπως είδαμε οι μεταβλητές μπορεί να έχουν διαφορετική διεύθυνση στο κυρίως σύστημα και διαφορετική στη συσκευή.

Μάλιστα, για να είμαστε ακριβείς, κάθε μεταβλητή δεν έχει 1, αλλά 2 ενδιάμεσες διευθύνσεις:

- Χρήσιμη ενδιάμεση διεύθυνση (usuable mediary address): Είναι η ενδιάμεση διεύθυνση με την έννοια που την περιγράψαμε πιο πάνω. Είναι αυτή που, όπως θα δούμε στη συνέχεια, χρησιμοποιείται στον κώδικα του χρήστη.

- Εσωτερική ενδιάμεση διεύθυνση (internal mediary address): Χρησιμοποιείται κατά την κλήση συναρτήσεων "εσωτερικά" στον OMPi. Μερικές φορές χρειαζόμαστε επιπλέον πληροφορίες πέραν μιας χρήσιμης ενδιάμεσης διεύθυνσης για να προσδιορίσουμε μια μεταβλητή, για παράδειγμα μπορεί να μας είναι απαραίτητο το μέγεθός της σε bytes. Μπορούμε να τοποθετήσουμε αυτές τις πληροφορίες μέσα σε μία δομή και να ορίσουμε τη διεύθυνση της δομής ως εσωτερική ενδιάμεση διεύθυνση.

Η εσωτερική και η χρήσιμη ενδιάμεση διεύθυνση μπορούν και να ταυτίζονται, όπως για παράδειγμα στη δική μας συσκευή. Αυτό επαφίεται στην ευχέρεια του προγραμματιστή που αναπτύσσει την υποστήριξη για τη συσκευή.

3.3 Η διεπαφή (API) για την υποστήριξη συσκευών

Μία βασική αρχή που ακολουθείται στην ανάπτυξη του OMPi είναι η ευκολία στην επεκτασιμότητα. Έτσι, για να υποστηρίξει κάποιος μία καινούργια συσκευή στον OMPi, αρκεί να φτιάξει μία μονάδα (module). Μία μονάδα υποστηρίζει πολλές συσκευές του ίδιου τύπου, για παράδειγμα η δική μας μονάδα μπορεί να υποστηρίξει όσους κόμβους cluster (συσκευές) ζητήσει ο χρήστης. Επίσης, παρέχει υλοποίηση για τις ακόλουθες συναρτήσεις, που αποτελούν τη διεπαφή εφαρμογών (API) για την υποστήριξη συσκευών:

- `void hm_set_module_name(char *modname)`: Θέτει το όνομα της μονάδας, όπως αυτό θα φαίνεται στον τελικό χρήστη.

- `int hm_get_num_devices(void)`: Επιστρέφει το πλήθος των συσκευών που υποστηρίζονται από αυτή τη μονάδα.

- `void hm_print_information(int device_offset)`: Τυπώνει πληροφορίες σχετικά με τη μονάδα αλλά και τις επί μέρους συσκευές που υποστηρίζει. Εμφανίζει την αντιστοίχιση μεταξύ ονομάτων συσκευών και θετικών ακεραίων στο χρήστη. Καλείται μόνο αν το ζητήσει ο χρήστης.

- `void hm_register_ee_calls(...)`: Κάθε μονάδα μεταφράζεται ως κοινόχρηστη βιβλιοθήκη (shared library). Προκειμένου να έχει πρόσβαση στις συναρτήσεις του runtime του OMPi [5] που χρειάζεται, θα πρέπει να ξέρει τη διεύθυνσή τους. Αυτές οι διευθύνσεις των συναρτήσεων δίνονται ως ορίσματα. Καλείται μία φορά, κατά την αρχικοποίηση της μονάδας.

- `void *hm_initialize(int devid, ...)`: Αρχικοποιεί τη συσκευή που αντιστοιχεί στον ακέραιο που δίνεται ως όρισμα. Επιστρέφει ένα γενικό δείκτη, στον οποίο αποθηκεύει τις πληροφορίες που χρειάζεται για κάθε συσκευή. Αυτός ο δείκτης δίνεται σαν όρισμα στις επόμενες συναρτήσεις, ώστε η μονάδα να είναι σε θέση να καταλάβει σε ποια συσκευή πρέπει να δράσει κάθε φορά. Καλείται την πρώτη φορά που ο χρήστης θέλει να χρησιμοποιήσει μία συσκευή.

- `void hm_finalize(void *devinfo)`: Τερματίζει τη συσκευή που της δίνεται ως όρισμα. Καλείται στο τέλος του προγράμματος, όταν η συσκευή δε χρειάζεται πια.

- `void hm_offload(...)`: "Φορτώνει" τη συνάρτηση που δίνεται ως όρισμα μαζί με τα ορίσματά της στη συσκευή που ζητείται και στη συνέχεια την εκτελεί εκεί. Επιστρέφει μόνο όταν η εκτέλεση στη συσκευή έχει ολοκληρωθεί. Καλείται κάθε φορά που ο χρήστης ζητά την εκτέλεση κώδικα σε συσκευή, με την οδηγία *target*.

- `void *hm_dev_alloc(void *device_info, size_t size, ...)`: Δεσμεύει την ποσότητα μνήμης που δίνεται ως όρισμα στη συσκευή που ζητείται και επιστρέφει την εσωτερική ενδιάμεση διεύθυνση της μνήμης που δεσμεύτηκε.

- `void hm_dev_free(void *devinfo, void *imedaddr, ...)`: Αποδεσμεύει τη μνήμη που αντιστοιχεί στην εσωτερική ενδιάμεση διεύθυνση που δίνεται ως όρισμα στη συσκευή που ζητείται.

- `void hm_todev(void *devinfo, void *hostaddr, size_t hostoffset, void *imedaddr, size_t devoffset, size_t size)`: Μεταφέρει την ποσότητα των δεδομένων που περιγράφεται με τα ορίσματα από τη μνήμη του κύριου συστήματος στη συσκευή που ζητείται.

- `void hm_fromdev(void *devinfo, void *hostaddr, size_t hostoffset, void *imedaddr, size_t devoffset, size_t size)`: Μεταφέρει την ποσότητα των δεδομένων που περιγράφεται με τα ορίσματα από τη συσκευή που ζητείται στη μνήμη του κύριου συστήματος.

- `void *hm_imed2umed_addr(void *devinfo, void *imedaddr)`: Επιστρέφει τη χρήσιμη ενδιάμεση διεύθυνση που αντιστοιχεί στην εσωτερική ενδιάμεση διεύθυνση που δίνεται ως όρισμα στη συσκευή που ζητείται. Στην περίπτωση που η χρήσιμη και η εσωτερική ενδιάμεση διεύθυνση ταυτίζονται, επιστρέφει απλώς το όρισμά της.

- `void *hm_umed2imed_addr(void *devinfo, void *umedaddr)`: Είναι η αντίστροφη της προηγούμενης συνάρτησης. Επιστρέφει την εσωτερική ενδιάμεση διεύθυνση που αντιστοιχεί στη χρήσιμη ενδιάμεση διεύθυνση που δίνεται ως όρισμα στη συσκευή που ζητείται. Στην περίπτωση που η εσωτερική και η χρήσιμη ενδιάμεση διεύθυνση ταυτίζονται, επιστρέφει απλώς το όρισμά της.

- `void *devpart_med2dev_addr(void *uaddr, unsigned long size)` : Μετατρέπει τη χρήσιμη ενδιάμεση διεύθυνση που δίνεται ως όρισμα σε διεύθυνση συσκευής.

3.4 Ένα παράδειγμα

Για να γίνουν τα παραπάνω πιο κατανοητά, σε αυτή την ενότητα θα εξετάσουμε σε βάθος τον κώδικα που παράγεται από τη μετάφραση ενός απλού προγράμματος με τον OMPi. Στο Πρόγραμμα 3.1 βλέπουμε ολοκληρωμένο το πρόγραμμα που προσθέτει δύο πίνακες στη συσκευή και μεταφέρει το αποτέλεσμα στο κυρίως σύστημα.

Πρόγραμμα 3.1: Πρόγραμμα πρόσθεσης δύο πινάκων σε συσκευή.

```
1 #include <omp.h>
2
3 float a[1024];
```

```
4 float b[1024];
5 float c[1024];
6 int size = 1024;
7
8 int main(void)
9 {
10   #pragma omp target map(to:a[0:size],b[0:size],size) map(from: c[0:size])
11   {
12     int i;
13     for (i = 0; i < size; i++)
14       c[i] = a[i] + b[i];
15   }
16 }
```

Το μεταφράζουμε με τον OMPi με την εντολή `ompicc -k add.c`. Το όρισμα `-k` είναι προαιρετικό και έχει ως αποτέλεσμα τη διατήρηση των ενδιάμεσων αρχείων που παράχθηκαν. Στο Πρόγραμμα 3.2 βλέπουμε τη μετασχηματισμένη συνάρτηση `main` που θα εκτελεστεί από τη CPU του βασικού συστήματος και βρίσκεται στο αρχείο `add_ompi.c`, που είναι το αρχείο εισόδου όπου οι εντολές OpenMP έχουν αντικατασταθεί με κλήσεις συναρτήσεων στη βιβλιοθήκη χρόνου εκτέλεσης του OMPi.

Πρόγραμμα 3.2: Η μετασχηματισμένη συνάρτηση `main`.

```
1 int __original_main(int _argc_ignored, char ** _argv_ignored)
2 {
3   /* (l15) #pragma omp target map(to: a[0 : size], b[0 : size], size) map(
        from: c[0 : size])
4  */
5   {
6     int __ompi_devID = (-1);
7     void * __ort_denv = ort_start_target_data(4, __ompi_devID);
8
9     /* map to */
10    ort_map_tdvar(&size, sizeof(size), &size, 0, 1);
11    ort_map_tdvar(&a, (size) * sizeof((a[0])), &(a[0]), 0, 1);
12    ort_map_tdvar(&b, (size) * sizeof((b[0])), &(b[0]), 0, 1);
13    /* map from */
14    ort_map_tdvar(&c, (size) * sizeof((c[0])), &(c[0]), 0, 0);
15    struct __dev_struct {
16        int (* size);
17        unsigned long _size_offset;
```

```
18          float (* a)[ 1024];
19          unsigned long _a_offset;
20          float (* b)[ 1024];
21          unsigned long _b_offset;
22          float (* c)[ 1024];
23          unsigned long _c_offset;
24        } * _dev_data;
25
26      _dev_data = (struct __dev_struct *) ort_devdata_alloc(sizeof(struct
            __dev_struct), __ompi_devID);
27      /* mapto variables */
28        /* moved to device data environment */
29      _dev_data->size = (int (*)) ort_host2med_addr(&size, __ompi_devID);
30      _dev_data->_size_offset = 0;
31        /* moved to device data environment */
32      _dev_data->a = (float (*)[ 1024]) ort_host2med_addr(&a, __ompi_devID);
33      _dev_data->_a_offset = 0;
34        /* moved to device data environment */
35      _dev_data->b = (float (*)[ 1024]) ort_host2med_addr(&b, __ompi_devID);
36      _dev_data->_b_offset = 0;
37      /* mapfrom variables */
38        /* moved to device data environment */
39      _dev_data->c = (float (*)[ 1024]) ort_host2med_addr(&c, __ompi_devID);
40      _dev_data->_c_offset = 0;
41      ort_offload_kernel(_kernelFunc0_, (void *) _dev_data, (void *) 0, "
            add2_d00", __ompi_devID, &size, &a, &b, &c, (void *) 0);
42      ort_devdata_free(_dev_data, __ompi_devID);
43      /* unmap from */
44      ort_unmap_tdvar(&c, 1);
45      ort_end_target_data(__ort_denv);
46    }
47  }
```

Στη γραμμή 6 επιλέγεται η προεπιλεγμένη συσκευή (-1), αφού δεν ορίσαμε εμείς κάποια άλλη και στη συνέχεια η συσκευή ξεκινά. Στις γραμμές 10-14 δεσμεύεται η κατάλληλη ποσότητα μνήμης στη συσκευή για τους πίνακες a, b και c, καθώς και για τη μεταβλητή size. Επίσης, μεταφέρονται τα δεδομένα των a, b και size στη συσκευή. Στη συνέχεια, ορίζεται και δεσμεύεται χώρος στη συσκευή για τη δομή _dev_data. Στις γραμμές 29-40 τοποθετούνται οι ενδιάμεσες διευθύνσεις των μεταβλητών που θα χρησιμοποιηθούν στη συσκευή στη δομή _dev_data. Στη γραμμή

41 ενημερώνουμε τη συσκευή να εκτελέσει τη συνάρτηση _kernelFunc0_ χρησιμοποιώντας τις διευθύνσεις των δεδομένων που θα βρει στη δομή _dev_data και μόλις τελειώσει καταστρέφεται η δομή. Στη γραμμή 44 αποδεσμεύεται ο πίνακας c, αφού πρώτα τα δεδομένα του αντιγραφούν στο κύριο σύστημα.

Στο Πρόγραμμα 3.3 βλέπουμε τη μετασχηματισμένη συνάρτηση _kernelFunc0_ από το αρχείο add_ompi.c. Αυτή η συνάρτηση περιέχει τον κώδικα που πρέπει να εκτελεστεί στη συσκευή.

Πρόγραμμα 3.3: Η μετασχηματισμένη συνάρτηση _kernelFunc0_.

```
1 static void * _kernelFunc0_(void * __arg)
2 {
3   struct __dev_struct {
4       int (* size);
5       unsigned long _size_offset;
6       float (* a)[ 1024];
7       unsigned long _a_offset;
8       float (* b)[ 1024];
9       unsigned long _b_offset;
10      float (* c)[ 1024];
11      unsigned long _c_offset;
12    };
13  struct __dev_struct * _dev_data = (struct __dev_struct *) __arg;
14
15  /* byresult variables */
16  /* mapto variables */
17  int size = *((int (*)) devpart_med2dev_addr(_dev_data->size, sizeof(*(
        _dev_data->size))));
18  float (* a)[ 1024] = devpart_med2dev_addr(_dev_data->a, sizeof(*(_dev_data
        ->a))) - _dev_data->_a_offset;
19  float (* b)[ 1024] = devpart_med2dev_addr(_dev_data->b, sizeof(*(_dev_data
        ->b))) - _dev_data->_b_offset;
20
21  /* mapfrom variables */
22  float (* c)[ 1024] = devpart_med2dev_addr(_dev_data->c, sizeof(*(_dev_data
        ->c))) - _dev_data->_c_offset;
23
24  /* (l15) #pragma omp target map(to: a[0 : size], b[0 : size], size) map(
        from: c[0 : size])
25 -- body moved below */
```

```
26   # 15 "add2.c"
27   # 11 "add2.c"
28   {
29     int i;
30
31     for (i = 0; i < size; i++)
32       (*c)[i] = (*a)[i] + (*b)[i];
33   }
34   /* copy back the results */
35   *((int (*)) devpart_med2dev_addr(_dev_data->size, sizeof(*(_dev_data->size
         )))) = size;
36   return ((void *) 0);
37 }
```

Αρχικά ορίζεται η δομή _dev_data, η οποία είναι ακριβώς ίδια με αυτή που είδαμε πριν για το κυρίως σύστημα. Στις γραμμές 15-22 γίνεται η μετατροπή των πινάκων a, b και c, καθώς και της μεταβλητής size από ενδιάμεσες διευθύνσεις σε διευθύνεις συσκευής (οι ενδιάμεσες διευθύνσεις περιέχονται στη δομή _dev_data που στάλθηκε από το βασικό σύστημα). Στη συνέχεια, στις γραμμές 24-33 εκτελείται ο κώδικας του χρήστη. Τέλος ενημερώνεται η τιμή της μεταβλητής size.

Κεφαλαιο 4

Υποστηριξη κομβων cluster ως συσκευων στο OpenMP

4.1 Εισαγωγή

Για να υποστηρίξουμε μία καινούργια συσκευή στον OMPi, πρέπει να δημιουργήσουμε μία νέα μονάδα (module) που υλοποιεί τη διεπαφή του OMPi για επικοινωνία με συσκευές. Μία μονάδα μπορεί να υποστηρίζει πάνω από μία συσκευή, αρκεί να είναι του ίδιου τύπου. Ονομάζουμε τη δική μας *mpinode*.

Ως συσκευή θεωρούμε ένα υπολογιστή που έχει εγκατεστημένο το MPI. Στόχος μας είναι να έχουμε πολλούς υπολογιστές, για παράδειγμα τους κόμβους από τους οποίους αποτελείται ένα cluster, να θεωρήσουμε καθένα από αυτούς ως διαφορετική συσκευή και να επιτρέψουμε στο χρήστη να τους χρησιμοποιήσει ταυτόχρονα ώστε να εκτελέσει μέρος του προγράμματός του σε αυτούς. Επειδή δεν υπάρχει κάποιος

γεωγραφικός ή άλλος περιορισμός που να καθορίζει τη θέση των υπολογιστών που θα χρησιμοποιηθούν, δεν έχουμε ένα εύκολο τρόπο να τους ανακαλύψουμε όλους αυτόματα. Έτσι, απαιτούμε από το χρήστη να εισάγει τις διευθύνσεις IP τους στο αρχείο ρυθμίσεων `~/.ompi_mpi_nodes`.

Ο αρχικός κόμβος από τον οποίο ξεκινά να εκτελείται το πρόγραμμα ονομάζεται κόμβος–host, ενώ οι υπόλοιποι κόμβοι–συσκευές. Η επικοινωνία γίνεται μόνο ανάμεσα στον κόμβο–host και τους κόμβους–συσκευές, δηλαδή δύο κόμβοι–συσκευές δεν μπορούν να επικοινωνήσουν μεταξύ τους χωρίς να παρεμβληθεί ο κόμβος–host. Η επίτευξη της επικοινωνίας πραγματοποιείται χρησιμοποιώντας τη βιβλιοθήκη MPI, και πιο συγκεκριμένα τις κλήσεις `MPI_Send()` και `MPI_Recv()`. Οι κόμβοι–συσκευές περιμένουν για εντολή από τον κόμβο–host, την εκτελούν και στη συνέχεια επιστρέφουν σε κατάσταση αναμονής. Μία εντολή μπορεί να αποτελείται από παραπάνω από ένα μηνύματα, για παράδειγμα ο κόμβος–host στέλνει ένα μήνυμα ζητώντας την τιμή μίας μεταβλητής και ο κόμβος–συσκευή απαντά με ένα μήνυμα που την περιέχει. Για να γνωρίζει ο κόμβος–συσκευή τον τύπο της εντολής που πρέπει να εκτελέσει κάθε φορά, κάθε εντολή κωδικοποιείται με έναν ακέραιο ο οποίος αποστέλλεται πάντα από τον κόμβο–host στο πρώτο μήνυμα.

Σε κάθε κόμβο, είτε είναι ο κόμβος–host είτε είναι κόμβος–συσκευή, εκτελείται μία διεργασία με το πρόγραμμα του χρήστη. Όλες οι διεργασίες εντάσσονται σε ένα *communicator* του MPI, με τη διεργασία που εκτελείται στον κόμβο–host να λαμβάνει την *τάξη (rank)* 0, ενώ οι υπόλοιπες τις τάξεις $1, 2, 3 \ldots$ Αυτό είναι σημαντικό ώστε οι διεργασίες που εκτελούνται στους κόμβους–συσκευές να ξέρουν πάντα πώς μπορούν να επικοινωνήσουν με τη διεργασία στον κόμβο–host.

Στις επόμενες ενότητες θα εξετάσουμε λεπτομερώς τα κομβικά σημεία της υλοποίησης που χρήζουν ιδιαίτερης προσοχής.

4.2 Εκκίνηση συσκευών

Υπάρχουν δύο τρόποι ώστε να επιτευχθεί η έναρξη διεργασιών σε απομακρυσμένα μηχανήματα (που για εμάς ισοδυναμεί με εκκίνηση των αντίστοιχων συσκευών) χρησιμοποιώντας το MPI. Ο πρώτος τρόπος είναι να ξεκινήσει ο χρήστης το πρόγραμμά του δίνοντας τα κατάλληλα ορίσματα: `mpirun --hostfile hostfile -np num_of_procs ./a.out`. Πέρα από το ότι αυτός ο τρόπος δεν είναι βολικός για τον τελικό χρήστη

που θα προτιμούσε να πληκτρολογήσει απλά `./a.out` για να τρέξει το πρόγραμμά του, υπάρχει και ένα ακόμη πρόβλημα. Το αρχείο `hostfile` που περιέχει τις IP διευθύνσεις των κόμβων που θα χρησιμοποιηθούν θα πρέπει να έχει ως πρώτη διεύθυνση την `localhost`, ώστε το τοπικό μηχάνημα να αποτελέσει το βασικό σύστημα (host) και οι υπόλοιποι κόμβοι της συσκευές. Αυτό είναι ιδιαίτερα σημαντικό αν ο χρήστης έχει συνδέσει και άλλες συσκευές στο βασικό σύστημα (για παράδειγμα κάρτες γραφικών) τις οποίες επιθυμεί να χρησιμοποιήσει[1]. Επίσης, με βάση το πλήθος των διευθύνσεων στο αρχείο `hostfile`, ο χρήστης θα πρέπει να υπολογίσει μόνος του την παράμετρο `num_of_procs`. Για να λύσουμε αυτά τα προβλήματα, καταφεύγουμε στο δεύτερο τρόπο που είναι η χρήση της συνάρτησης `MPI_Comm_spawn()` για τη δυναμική δημιουργία επιπλέον διεργασιών μέσα από τη βασική εν ώρα εκτέλεσης.

Η εύρεση της κατάλληλης χρονικής στιγμής για την εκκίνηση των συσκευών είναι ένα ζήτημα που πρέπει να αντιμετωπιστεί με προσοχή. Τόσο ο OMPi, όσο και το MPI θέτουν αρκετούς περιορισμούς που ελαττώνουν τις επιλογές μας:

- Η πρώτη κλήση MPI που πρέπει να καλέσει κάθε διεργασία είναι η `MPI_Init()`.
- Η κλήση `MPI_Comm_spawn()` είναι συλλογική: για να ολοκληρωθεί κάθε διεργασία που ανήκει στον υπάρχον communicator πρέπει να καλέσει `MPI_Comm_spawn()` και κάθε διεργασία που δημιουργείται πρέπει να καλέσει `MPI_Init()`.
- Η κλήση `MPI_Comm_spawn()` επιστρέφει ένα communicator για την επικοινωνία των νέων διεργασιών. Εμείς συγχωνεύουμε αυτό τον communicator με τον αρχικό (που περιέχει μόνο την αρχική διεργασία) με την κλήση `MPI_Intercomm_merge()` και πρέπει να αποθηκεύσουμε τον communicator που προκύπτει σε μία καθολική (global) μεταβλητή για να τον χρησιμοποιήσουμε σε επόμενες κλήσεις του MPI.
- Στον OMPi ο κώδικας για τη διαχείριση συσκευών (module) φορτώνεται ως δυναμική βιβλιοθήκη κατά την εκκίνηση του προγράμματος και καλείται η συνάρτηση `hm_get_num_devices()`, που πρέπει να εξετάσει το αρχείο ρυθμίσεων για να αποφανθεί για το πλήθος των συσκευών. Στη συνέχεια, η κοινόχρηστη βιβλιοθήκη κλείνει και ανοίγει ξανά μόνο αν χρειαστεί να χρησι-

[1]Το πρότυπο OpenMP επιτρέπει μόνο στο βασικό σύστημα (host) να στείλει κώδικα σε συσκευές (devices). Μία συσκευή δεν μπορεί να στείλει κώδικα σε μία άλλη συσκευή. Για να χρησιμοποιηθεί μία κάρτα γραφικών ως συσκευή, θα πρέπει να είναι συνδεδεμένη στον κόμβο–host και όχι σε ένα κόμβο–συσκευή.

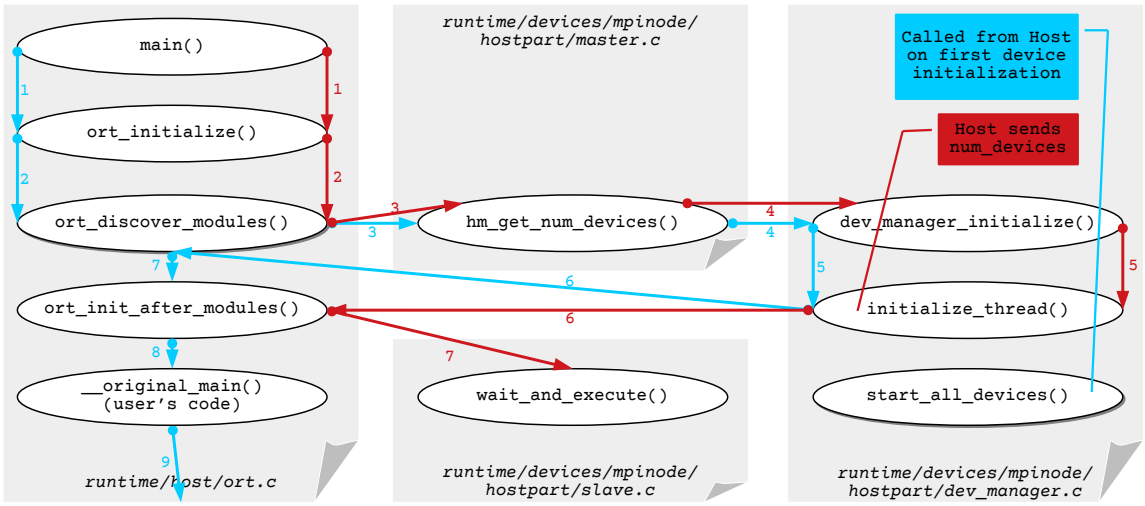
μοποιηθεί μία συσκευή που υποστηρίζει. Αυτό σημαίνει ότι δεν μπορούμε να έχουμε καθολικές μεταβλητές για διατήρηση κατάστασης μεταξύ των κλήσεων `hm_get_num_devices()` και `hm_initialize()`. Επίσης, ένα μέρος της αρχικοποίησης πρέπει να πραγματοποιηθεί αναγκαστικά δύο φορές σε αυτά τα δύο σημεία.

- Όλες οι διεργασίες που δημιουργούνται με το MPI εκτελούν το ίδιο πρόγραμμα, όμως μόνο αυτή που τρέχει στο βασικό σύστημα (host) πρέπει να εκτελέσει τον κώδικα του χρήστη. Οι υπόλοιπες θα πρέπει να καταλάβουν ότι βρίσκονται σε συσκευή με την κλήση `MPI_Comm_get_parent()` και να σταματήσουν περιμένοντας για εντολές από τη βασική διεργασία. Επειδή το σύστημα υποστήριξης χρόνου εκτέλεσης (runtime support system) πρέπει να είναι ανεξάρτητο από τις συσκευές (device-agnostic), δεν θέλουμε να υπάρχουν διάσπαρτες κλήσεις του MPI στον κώδικα του OMPi, αλλά να είναι όλες συγκεντρωμένες στον κώδικα της μονάδας (module), υπάρχει μόνο ένα σημείο που μπορεί να γίνει η διαφοροποίηση στη ροή εκτέλεσης. Αυτό είναι στην κλήση της συνάρτησης `hm_get_num_devices()` της μονάδας μας κατά την εκκίνηση του προγράμματος. Δεν πρέπει όμως να ξεχάσουμε να ολοκληρώσουμε την αρχικοποίηση που έχει μείνει στη μέση για τις διεργασίες που εκτελούνται σε συσκευές.

Λαμβάνοντας υπόψιν τους παραπάνω περιορισμούς, καταλήγουμε στην ακόλουθη λύση για την εκκίνηση των συσκευών, η οποία φαίνεται και γραφικά στο Σχήμα 4.1. Την πρώτη φορά που καλείται η συνάρτηση `hm_initialize()` εκκινούμε όλες τις συσκευές τύπου mpinode[2]. Μέσα από τη συνάρτηση `dev_manager_initialize()` η διεργασία που εκτελείται στον κόμβο–host διαβάζει το αρχείο ρυθμίσεων, καλεί την `start_all_devices()` που εκκινεί όλες τις συσκευές που υποστηρίζονται από τη μονάδα mpinode και συγχωνεύει τους δύο communicators. Στη συνέχεια, εκπέμπει (broadcast) το πλήθος όλων των συσκευών που έχουν αναγνωριστεί από τον OMPi στις διεργασίες που εκτελούνται στους κόμβους–συσκευές.

Αυτές ξεκινούν να τρέχουν το πρόγραμμα του χρήστη και από τη συνάρτηση `ort_initialize()` (που αρχικοποιεί τη βιβλιοθήκη χρόνου εκτέλεσης OMPi runtime)

[2]Ο αρχικός σχεδιασμός ήταν να εκκινούμε μόνο τη συσκευή που ζητήθηκε σε κάθε κλήση της `hm_initialize()`. Όμως, ένα πρόβλημα (bug) στην υλοποίηση της `MPI_Comm_spawn()` στο OpenMPI, που είναι η έκδοση του MPI που χρησιμοποιούμε, απέτρεψε αυτή τη δυνατότητα.



Σχήμα 4.1: Γραφική αναπαράσταση της ακολουθίας των κλήσεων για την εκκίνηση των συσκευών. Με μπλε χρώμα είναι σημειωμένη η ακολουθία των κλήσεων του κόμβου–host, ενώ με κόκκινο χρώμα των κόμβων–συσκευών.

μεταβαίνουν στην `ort_discover_modules()` (που ψάχνει στο σύστημα για συσκευές που αναγνωρίζονται από τον OMPi), στη συνέχεια στην `hm_get_num_devices()` της μονάδας mpinode (που επιστρέφει το πλήθος των συσκευών που αναγνωρίζονται από τη συγκεκριμένη μονάδα), από εκεί στην `dev_manager_initialize()` (που διαβάζει το αρχείο ρυθμίσεων `.ompi_mpi_nodes` και υπό συνθήκες αρχικοποιεί τη διεργασία) και τέλος στην `initialize_thread()` (που αρχικοποιεί τη διεργασία). Αφού γίνει η αρχικοποίηση του MPI, καλώντας την `MPI_Comm_get_parent()` διαπιστώνουν ότι δεν είναι η αρχική διεργασία που τρέχει στον κόμβο–host, οπότε συγχωνεύουν και αυτές με τη σειρά τους τους δύο communicators. Έπειτα, λαμβάνουν το πλήθος όλων των συσκευών που έχουν αναγνωριστεί και ολοκληρώνουν την αρχικοποίηση τους καλώντας την `ort_init_after_modules()` (που είναι ο κώδικας της `ort_initialize()` που βρίσκεται μετά την `ort_discover_modules()`). Ας θυμηθούμε ότι βρισκόμαστε μέσα στη συνάρτηση `ort_discover_modules()`, σε διαδικασία αρχικοποίησης. Το πλήθος όλων των συσκευών υπολογίζεται στη συνάρτηση `ort_discover_modules()` και άρα είναι γνωστό μόνο μετά το τέλος της, δηλαδή μόνο στην αρχική διεργασία που είχε τελειώσει κανονικά την αρχικοποίηση της. Όμως, είναι απαραίτητο σε όλες τις διεργασίες για να εκτελεστεί σωστά η `ort_init_after_modules()`, αυτός είναι και ο λόγος ύπαρξης της εκπομπής. Τέλος, καλούν την `wait_and_execute()` ώστε να περιμένουν για εντολές από τη διεργασία στον κόμβο–host, αντί να εκτελέσουν και αυτές τον

κώδικα του χρήστη.

4.3 Η δομή kerneltable

Ο OMPi δημιουργεί μία συνάρτηση για κάθε τμήμα κώδικα που περιέχεται σε μία οδηγία *target*. Οι συναρτήσεις αυτές ονομάζονται *συναρτήσεις πυρήνα*. Στον παραγόμενο κώδικα δηλώνονται ως `_kernelFunc0_`, `_kernelFunc1_` και ούτω καθεξής και βρίσκονται μέσα στο τελικό πρόγραμμα του χρήστη. Το MPI φροντίζει να αντιγράψει το εκτελέσιμο αρχείο σε όλους τους κόμβους που θέλουμε να χρησιμοποιήσουμε. Όμως πώς μπορεί να πει ο κόμβος–host σε ένα κόμβο–συσκευή να εκτελέσει μία συνάρτηση πυρήνα;

Η απλή λύση θα ήταν να στείλει τη διεύθυνση της συνάρτησης στον κόμβο–συσκευή, αλλά αυτό θα δούλευε μόνο υπό αυστηρές προϋποθέσεις. Στη γενική περίπτωση ο κόμβος–host και ο κόμβος–συσκευή είναι δύο διαφορετικά μηχανήματα με ενδεχομένως διαφορετικά τεχνικά χαρακτηριστικά και έτσι μία διεργασία που τρέχει στον ένα κόμβο δε χρησιμοποιεί τις ίδιες διευθύνσεις με μία διεργασία που τρέχει στον άλλο κόμβο, ακόμα και αν οι διεργασίες προέρχονται από το ίδιο ακριβώς εκτελέσιμο αρχείο.

Για να λύσουμε αυτό το πρόβλημα καταφεύγουμε στη δομή *kerneltable*, που υλοποιείται στο αρχείο `runtime/rt_common.c`. Αυτή η δομή αντιστοιχίζει κάθε συνάρτηση πυρήνα σε έναν ακέραιο. Σε αντίθεση με τις διευθύνσεις των συναρτήσεων που διαφέρουν από μηχάνημα σε μηχάνημα, οι ακέραιοι αριθμοί θα είναι ίδιοι αν εξασφαλίσουμε ότι κάθε μηχάνημα πραγματοποιεί την ίδια αντιστοίχιση. Εσωτερικά, η δομή αυτή είναι ένας πίνακας, το μέγεθος του οποίου αυξάνεται δυναμικά, καθώς εισάγονται περισσότερα στοιχεία. Κάθε στοιχείο του πίνακα αποτελείται από δύο μέρη: το όνομα της συνάρτησης πυρήνα ως αλφαριθμητικό και τη διεύθυνση της συνάρτησης πυρήνα ως δείκτης σε συνάρτηση. Αντιστοιχίζουμε κάθε συνάρτηση πυρήνα με τον αριθμοδείκτη του στοιχείου του πίνακα στο οποίο είναι αποθηκευμένη. Δηλαδή, η πρώτη συνάρτηση που εισάγεται στη δομή kerneltable αντιστοιχίζεται στον ακέραιο 0, η δεύτερη στον ακέραιο 1, κ.ο.κ.

Το API που είναι διαθέσιμο προς χρήση βρίσκεται στο αρχείο `runtime/rt_common.h` και αποτελείται από τις ακόλουθες συναρτήσεις:

- `void ort_kerneltable_add(char *name, void *(*kernel_function)(void *))`: Προ-

σθέτει τη συνάρτηση πυρήνα με όνομα `name` (για παράδειγμα `"_kernelFunc0_"`) που βρίσκεται στη διεύθυνση `kernel_function` στη δομή kerneltable. Πρέπει να κληθεί μία φορά για κάθε συνάρτηση πυρήνα σε κάθε κόμβο. Αυτό αναλαμβάνει να το κάνει αυτόματα ο OMPi αμέσως πριν την εκτέλεση της συνάρτησης `main` του προγράμματος του χρήστη.

- `int get_kernel_id_from_name(char *name)`: Επιστρέφει τον ακέραιο αριθμοδείκτη που αντιστοιχεί στη συνάρτηση `name` που δίνεται ως όρισμα. Καλείται από τον κόμβο–host, που αντί να στείλει τη διεύθυνση της συνάρτησης πυρήνα προς εκτέλεση στον κόμβο–συσκευή, στέλνει τον ακέραιο αριθμοδείκτη που αντιστοιχεί σε αυτή τη συνάρτηση. Φυσικά, θα μπορούσε να στείλει το όνομα της συνάρτησης ως αλφαριθμητικό αντί να στείλει τον αριθμοδείκτη, αλλά η αποστολή ενός ακεραίου είναι πιο αποδοτική από την αποστολή ενός αλφαριθμητικού.

- `void *(*get_kernel_function_from_id(int id))(void *)`: Επιστρέφει ένα δείκτη στη συνάρτηση που αντιστοιχεί στον αριθμοδείκτη `id` που δίνεται ως όρισμα. Καλείται από τον κόμβο–συσκευή μόλις λάβει έναν αριθμοδείκτη από τον κόμβο–host για να βρει ποια συνάρτηση πρέπει να εκτελέσει. Ο δείκτης σε συνάρτηση που επιστρέφεται είναι έγκυρος για χρήση από το μηχάνημα που έκανε την κλήση, επομένως ο κόμβος–συσκευή μπορεί να καλέσει χωρίς άλλες μετατροπές και δεύτερες σκέψεις τη σωστή συνάρτηση πυρήνα.

- `void free_kerneltable(void)`: Απελευθερώνει όλη τη μνήμη που χρησιμοποιείται από τη δομή kerneltable. Για την αποφυγή διαρροών μνήμης, καλείται μία φορά από κάθε διεργασία κατά τον τερματισμό του προγράμματος. Δεν είναι δυνατή η κλήση άλλων συναρτήσεων από αυτό το API μετά από την κλήση αυτής της συνάρτησης.

Το παραπάνω API καθώς και η δομή kerneltable για την ώρα χρησιμοποιούνται μόνο από τη μονάδα (module) mpinode, αλλά δεν υπάρχει τίποτα που να εμποδίζει τη χρήση τους και σε άλλα σημεία στο μέλλον.

Για τη σωστή λειτουργία της δομής kerneltable, όλες οι διεργασίες θα πρέπει να εισάγουν τις συναρτήσεις πυρήνα στη δομή με την ίδια σειρά. Αυτό το εξασφαλίζουμε επειδή όλες οι διεργασίες (είτε αυτή που τρέχει στον κόμβο–host, είτε αυτές που τρέχουν στους κόμβους–συσκευές) εκτελούν το ίδιο πρόγραμμα. Επομένως,

εκτελούν τις κλήσεις στη συνάρτηση `ort_kerneltable_add()` με την ίδια σειρά.

4.4 Χειρισμός μεταβλητών

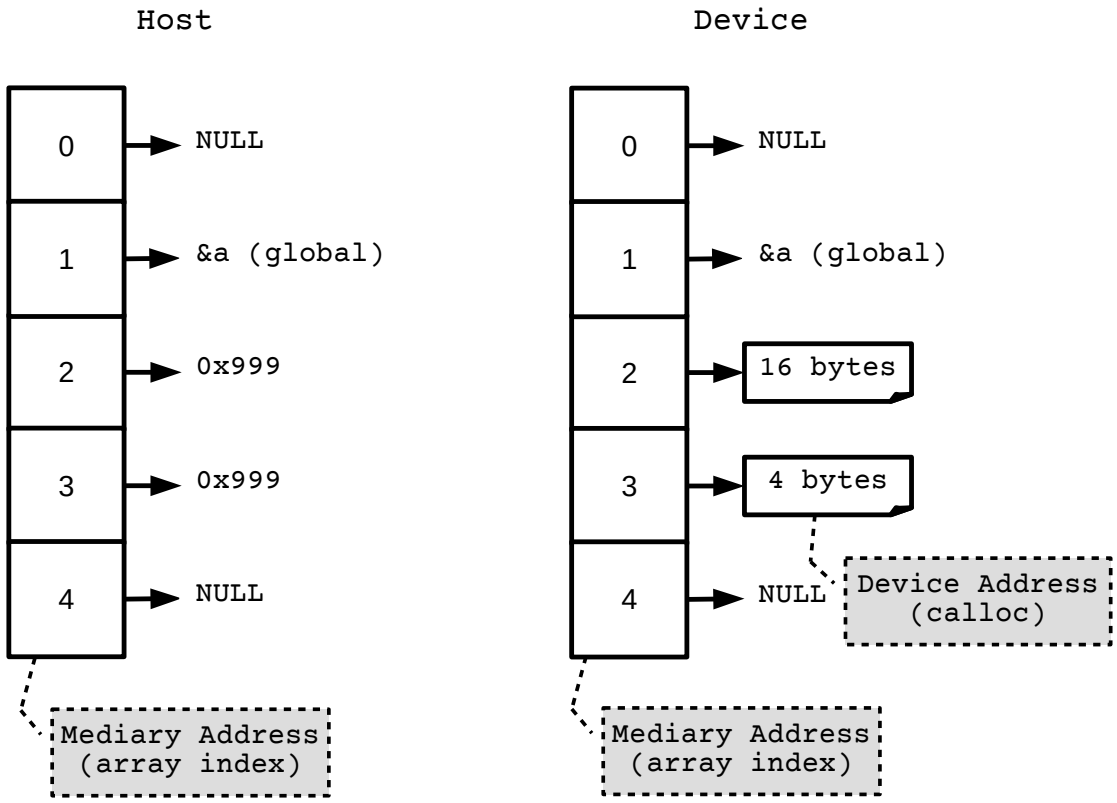
Ο χειρισμός των μεταβλητών που πρέπει να αποσταλούν σε ένα κόμβο–συσκευή παρουσιάζει παρόμοιες προκλήσεις με την αποστολή συναρτήσεων πυρήνα, πρόβλημα που μελετήσαμε στην Ενότητα 4.3. Για αυτό το λόγο, υιοθετούμε παρόμοιες λύσεις τις οποίες παρουσιάζουμε στις ακόλουθες υποενότητες. Αρχικά θα δούμε πώς χειριζόμαστε τοπικές μεταβλητές που πρέπει να αποσταλούν σε κόμβους–συσκευές και στη συνέχεια θα εξετάσουμε τις καθολικές μεταβλητές. Για να γίνουν τα παρακάτω πιο κατανοητά, στο Σχήμα 4.2 παρουσιάζεται ένα παράδειγμα που δείχνει τον πίνακα των ενδιάμεσων μεταβλητών τόσο στον κόμβο–host, όσο και στον κόμβο–συσκευή μετά την εισαγωγή δύο μεταβλητών σε αυτόν.

4.4.1 Ενδιάμεσες διευθύνσεις

Σε κάθε οδηγία *target* ο κόμβος–host τοποθετεί την ενδιάμεση διεύθυνση κάθε μεταβλητής που θέλουμε να χρησιμοποιήσουμε σε μία συσκευή σε μία δομή `_dev_data`. Στη συνέχεια, ο κόμβος–συσκευή που αναλαμβάνει την εκτέλεση της οδηγίας χρησιμοποιεί τη δομή `_dev_data` ώστε να εξάγει τη διεύθυνση συσκευής για κάθε μεταβλητή που χρειάζεται. Ο λόγος που οι ενδιάμεσες διευθύνσεις είναι απαραίτητες είναι ότι διαφορετικές διεργασίες που εκτελούνται σε διαφορετικούς κόμβους του cluster τρέχουν το ίδιο εκτελέσιμο αρχείο. Επομένως, παρόλο που χρησιμοποιούν τις ίδιες μεταβλητές, η διεύθυνση κάθε μεταβλητής είναι διαφορετική σε κάθε κόμβο.

Ο τρόπος που υλοποιούνται οι ενδιάμεσες διευθύνσεις είναι ο εξής:

- Κάθε κόμβος–συσκευή διατηρεί ένα πίνακα, το μέγεθος του οποίου μεταβάλλεται δυναμικά. Όταν ο κόμβος–host ζητήσει δέσμευση μνήμης για μία μεταβλητή που πρέπει να μεταφερθεί στη συσκευή, βρίσκουμε το πρώτο στοιχείο του πίνακα που δε χρησιμοποιείται. Στη συνέχεια, με μία κλήση `calloc()` δεσμεύουμε την κατάλληλη ποσότητα μνήμης και αποθηκεύουμε τη διεύθυνσή της σε αυτό το στοιχείο του πίνακα. Τα στοιχεία που δε χρησιμοποιούνται έχουν αποθηκευμένη τη διεύθυνση `NULL`. Έτσι, η διεύθυνση που είναι αποθηκευμένη σε ένα στοιχείο του πίνακα αποτελεί τη διεύθυνση συσκευής της



Σχήμα 4.2: Παράδειγμα πίνακα ενδιάμεσων διευθύνσεων στον κόμβο–host (αριστερά) και στον κόμβο–συσκευή (δεξιά). Υποθέτουμε ότι στο πρόγραμμα υπάρχει μόνο μία καθολική μεταβλητή με όνομα a. Επίσης, ο κόμβος–host έχει ζητήσει από τον κόμβο–συσκευή δύο φορές δέσμευση μνήμης: την πρώτη για ένα πίνακα μεγέθους 16 bytes και τη δεύτερη για μία μεταβλητή μεγέθους 4 bytes.

αντίστοιχης μεταβλητής, ενώ ο αριθμοδείκτης του στοιχείου αποτελεί την ενδιάμεση διεύθυνση της μεταβλητής, που χρησιμοποιείται κατά την επικοινωνία με τον κόμβο–host.

- Ο κόμβος–host δεν είναι αναγκαίο να διατηρεί ένα πίνακα για να γνωρίζει ποιες ενδιάμεσες διευθύνσεις είναι σε χρήση σε κάθε συσκευή. Κάθε φορά που ζητάει από ένα κόμβο–συσκευή να δεσμεύσει μνήμη για μία μεταβλητή θα μπορούσε να περιμένει ένα μήνυμα απάντησης που να περιέχει την ενδιάμεση διεύθυνση της καινούργιας μεταβλητής. Όμως, προκειμένου να αποφύγουμε την άσκοπη επικοινωνία και να μεγιστοποιήσουμε την απόδοση, χρησιμοποιούμε ένα πίνακα για τη διαχείριση των ενδιάμεσων διευθύνσεων και στον κόμβο–host. Φυσικά ο κόμβος–host δε χρειάζεται να δεσμεύει πραγματική μνήμη με κλήση της `calloc()`, αρκεί να "θυμάται" ποια στοιχεία χρησιμοποιούνται. Έτσι, κάθε φορά που πρέπει να δεσμευτεί μνήμη για μία μεταβλητή, βρίσκουμε το πρώτο στοιχείο του πίνακα που δε χρησιμοποιείται, το σημαδεύουμε με την ειδική (και αυθαίρετη) τιμή `0x999`, ενημερώνουμε την κατάλληλη συσκευή να προχωρήσει σε δέσμευση μνήμης και επιστρέφουμε στη βιβλιοθήκη χρόνου εκτέλεσης (runtime) του OMPi την ενδιάμεση διεύθυνση της μεταβλητής, την οποία γνωρίζουμε εκ των προτέρων. Τα στοιχεία του πίνακα που δε χρησιμοποιούνται περιέχουν και πάλι την ειδική διεύθυνση `NULL`. Επειδή είναι δυνατόν να υπάρχει πάνω από ένας κόμβος–συσκευή και επειδή θέλουμε να αποφύγουμε τη σπατάλη μνήμης, ο κόμβος–host διατηρεί έναν ανεξάρτητο τέτοιο πίνακα για κάθε συσκευή.

4.4.2 Καθολικές (global) μεταβλητές

Οι καθολικές μεταβλητές, όπως και οι τοπικές, μπορεί να έχουν διαφορετικές διευθύνσεις όταν οι διεργασίες που τις περιέχουν εκτελούνται σε διαφορετικά μηχανήματα. Όταν πρόκειται να εκτελεστεί μία συνάρτηση πυρήνα, ο κόμβος–host λαμβάνει ως παράμετρο της συνάρτησης `hm_offload()` τη δομή `_decl_data`, που περιέχει τις διευθύνσεις των καθολικών μεταβλητών που θα χρησιμοποιηθούν. Παρ' όλα αυτά, οι συναρτήσεις πυρήνα που εκτελούνται στους κόμβους–συσκευές δέχονται μόνο μία παράμετρο για τη δομή `_dev_data`, επομένως χρειαζόμαστε ένα διαφορετικό τρόπο για να διαχειριστούμε τις καθολικές μεταβλητές, πέρα από το να τις μεταβιβάζουμε στον κόμβο–συσκευή κατά την κλήση της `hm_offload()`.

Η λύση στην οποία οδηγηθήκαμε είναι να αποθηκεύουμε τη διεύθυνση των καθολικών μεταβλητών στην αρχή του πίνακα που αποθηκεύουμε τις ενδιάμεσες διευθύνσεις των μεταβλητών και πριν αποθηκεύσουμε εκεί τίποτα άλλο. Προσθέτουμε μία επιπλέον παράμετρο στην κλήση `hm_dev_alloc()` που είναι η διεύθυνση στον κόμβο–host της μεταβλητής που πρέπει να δεσμευτεί στον κόμβο–συσκευή. Με αυτό τον τρόπο ελέγχουμε αν η μεταβλητή που πρέπει να δεσμευτεί είναι καθολική και σε αυτή την περίπτωση δε χρειάζεται να κάνουμε τίποτα, αφού σε όλους τους κόμβους δεσμεύεται εξ ορισμού μνήμη για τις καθολικές μεταβλητές του προγράμματος. Για να συνοψίσουμε, μέσα στις συναρτήσεις πυρήνα οι καθολικές μεταβλητές χρησιμοποιούνται με τη διεύθυνσή τους, που είναι η διεύθυνση συσκευής εξ ορισμού. Όταν ο κόμβος–host χρειάζεται να αναφερθεί σε αυτές χρησιμοποιεί την ενδιάμεση διεύθυνσή τους, που στον κόμβο–συσκευή είναι αντιστοιχισμένη με την πραγματική τους διεύθυνση στη συσκευή.

Το τελευταίο μας πρόβλημα είναι πώς γίνεται αρχικά η εισαγωγή των καθολικών μεταβλητών στον πίνακα. Ο OMPi διατηρεί μία δομή κατακερματισμού (hash) στην οποία διατηρεί αρκετές χρήσιμες πληροφορίες για κάθε μεταβλητή, όπως για παράδειγμα αν είναι καθολική ή όχι. Κατά την αρχικοποίηση του πίνακα, διατρέχουμε αυτή τη δομή, βρίσκουμε ποιες μεταβλητές είναι καθολικές και τις εισάγουμε με τη σειρά στον πίνακα. Όμως τόσο ο κόμβος–host όσο και ο κόμβος–συσκευή πρέπει να εισάγουν τις μεταβλητές με την ίδια σειρά και η δομή κατακερματισμού δεν το υποστηρίζει αυτό, δηλαδή δύο διεργασίες που διατρέχουν διαφορετικές δομές με τα ίδια στοιχεία ενδέχεται να δώσουν τα στοιχεία με διαφορετική σειρά. Για να το λύσουμε αυτό ακολουθούμε την εξής προσέγγιση: οι καθολικές μεταβλητές εισάγονται μία–μία στη δομή κατά την εκκίνηση του προγράμματος με κλήσεις της συνάρτησης `ort_decltarg_register()`. Επειδή όλοι οι κόμβοι τρέχουν το ίδιο εκτελέσιμο αρχείο, συμπεραίνουμε ότι οι εισαγωγές γίνονται με την ίδια σειρά. Έτσι, κατά την εισαγωγή μίας καθολικής μεταβλητής αποθηκεύουμε ακόμη ένα πεδίο στις πληροφορίες της μεταβλητής, που είναι η σειρά εισαγωγής. Στη συνέχεια, όταν διατρέχουμε τη δομή κατακερματισμού, εισάγουμε τις καθολικές μεταβλητές στον πίνακα με τις ενδιάμεσες διευθύνσεις στη θέση που υποδεικνύει η σειρά εισαγωγής και όχι στην επόμενη θέση που είναι διαθέσιμη. Με αυτό τον τρόπο εξασφαλίζουμε ότι οι καθολικές μεταβλητές είναι αποθηκευμένες με την ίδια σειρά στον πίνακα για όλους τους κόμβους.

4.4.3 Το API για τη διαχείριση μνήμης

Το API για τη διαχείριση των μεταβλητών που πρέπει να μεταφερθούν σε μία συσκευή βρίσκεται στο αρχείο `runtime/devices/mpinode/hostpart/memory.h`, ενώ η υλοποίησή του είναι στο αρχείο `runtime/devices/mpinode/hostpart/memory.c`. Οι συναρτήσεις που το αποτελούν είναι:

- `void alloc_items_init(int number_of_devices)`: Αρχικοποιεί τον πίνακα που αποθηκεύονται οι ενδιάμεσες διευθύνσεις των μεταβλητών που χρησιμοποιούνται σε συσκευές. Πρέπει να κληθεί μία φορά πριν από τις υπόλοιπες συναρτήσεις αυτού του API. Ο κόμβος–host την καλεί δίνοντας ως παράμετρο το πλήθος των κόμβων–συσκευών, ενώ οι κόμβοι–συσκευές με παράμετρο 1.

- `void alloc_items_init_global_vars(int devid, int is_master)`: Προσθέτει στον πίνακα τις καθολικές μεταβλητές του προγράμματος του χρήστη. Πρέπει να κληθεί μετά από την προηγούμενη συνάρτηση. Ο κόμβος–host την καλεί μία φορά για κάθε κόμβο–συσκευή δίνοντας ως παράμετρο τον κατάλληλο αριθμό συσκευής και την τιμή `is_master` 1, ενώ οι κόμβοι–συσκευές μία φορά δίνοντας και στις δύο παραμέτρους την τιμή 0.

- `size_t alloc_items_get_global(int devid, void *addr)`: Επιστρέφει την ενδιάμεση διεύθυνση που αντιστοιχεί στη διεύθυνση της καθολικής μεταβλητής που δίνεται ως παράμετρος για τη συσκευή που ζητείται.

- `size_t alloc_items_register(int devid)`: Προσθέτει μία μεταβλητή στον πίνακα της συσκευής που ζητείται χωρίς να δεσμεύει μνήμη για αυτή. Επιστρέφει την ενδιάμεση διεύθυνση της μεταβλητής. Καλείται από τον κόμβο–host.

- `void *alloc_items_add(size_t maddr, size_t size)`: Δεσμεύει την ποσότητα μνήμης που δίνεται ως παράμετρος για τη μεταβλητή με την ενδιάμεση διεύθυνση που ζητείται. Επιστρέφει τη διεύθυνση συσκευής της μεταβλητής. Καλείται από τους κόμβους–συσκευές.

- `void alloc_items_unregister(int devid, size_t maddr)`: Αφαιρεί τη μεταβλητή με την ενδιάμεση διεύθυνση που δίνεται ως παράμετρος από τον πίνακα της συσκευής που ζητείται χωρίς να αποδεσμεύσει τη μνήμη που της αντιστοιχεί. Καλείται από τον κόμβο–host.

- `void alloc_items_remove(size_t maddr)`: Αφαιρεί τη μεταβλητή με την ενδιάμεση διεύθυνση που δίνεται ως παράμετρος από τον πίνακα αποδεσμεύοντας τη μνήμη που της αντιστοιχεί. Καλείται από τους κόμβους–συσκευές.

- `void *alloc_items_get(size_t maddr)`: Επιστρέφει την πραγματική διεύθυνση της μεταβλητής με την ενδιάμεση διεύθυνση που δίνεται ως παράμετρος. Καλείται τόσο από τον κόμβο–host όσο και από τους κόμβους–συσκευές.

- `void alloc_items_free_all(void)`: Αποδεσμεύει τη μνήμη για τα στοιχεία που δεν έχουν ακόμα αφαιρεθεί και στη συνέχεια τη μνήμη για τον πίνακα. Καλείται μία φορά από όλους τους κόμβους κατά τον τερματισμό του προγράμματος. Καμία άλλη συνάρτηση από αυτό το API δεν πρέπει να κληθεί μετά.

4.5 Ζητήματα παραλληλίας

Ο κώδικας της μονάδας (module) μας καλείται όταν συναντάται μία οδηγία *target* και αυτό ενδέχεται να συμβαίνει ταυτόχρονα από διαφορετικά νήματα. Για παράδειγμα, μία οδηγία *target* μπορεί να βρίσκεται μέσα σε μία οδηγία *parallel for*. Αυτός μάλιστα είναι ο ενδεδειγμένος τρόπος χρήσης της μονάδας για την επίτευξη των καλύτερων δυνατών επιδόσεων. Δηλαδή, διαφορετικά νήματα στον κόμβο–host επικοινωνούν και στέλνουν δουλειά σε διαφορετικούς κόμβους–συσκευές ταυτόχρονα. Όμως, για να συμβεί αυτό η μονάδα μας θα πρέπει να υποστηρίζει παράλληλες κλήσεις, να είναι δηλαδή *thread safe*.

Ας ξεκινήσουμε από τις καθολικές (global) μεταβλητές που χρησιμοποιούμε για τη διατήρηση κατάστασης μεταξύ κλήσεων. Στη γενική περίπτωση αυτές θα έπρεπε να προστατεύονται από ταυτόχρονη επεξεργασία με δομές αμοιβαίου αποκλεισμού, όπως είναι για παράδειγμα οι κλειδαριές (mutexes). Όμως, εμείς χρειάζεται να τροποποιήσουμε αυτές τις μεταβλητές μόνο κατά την αρχικοποίηση και τον τερματισμό κάποιας συσκευής, στις συναρτήσεις `hm_initialize()` και `hm_finalize()`. Αυτές οι συναρτήσεις καλούνται από τη βιβλιοθήκη χρόνου εκτέλεσης (runtime) του OMPi εγγυημένα σειριακά, επομένως είμαστε καλυμμένοι χωρίς να κάνουμε τίποτα.

Ας εξετάσουμε τώρα αν οι συναρτήσεις που καλούμε από τη μονάδα μας, που είναι οι συναρτήσεις του MPI, είναι thread safe. Αν η αρχικοποίηση του MPI γίνει με τη συνάρτηση `MPI_Init()`, τότε οι κλήσεις στο MPI δεν πρέπει να γίνονται παράλ-

ληλα. Για να έχουμε υποστήριξη για παράλληλες κλήσεις, θα πρέπει να κάνουμε την αρχικοποίηση με τη συνάρτηση `MPI_Init_thread()` δίνοντάς της την παράμετρο `MPI_THREAD_MULTIPLE`. Οι έγκυρες παράμετροι είναι:

- `MPI_THREAD_SINGLE`: Χρησιμοποιείται όταν η εφαρμογή αποτελείται από ένα μόνο νήμα (δηλαδή δεν είναι παράλληλη) και είναι ισοδύναμη με την κλήση `MPI_Init()`.

- `MPI_THREAD_FUNNELED`: Αν η εφαρμογή αποτελείται από πολλά νήματα, μόνο ένα νήμα πρέπει να καλέσει την `MPI_Init_thread()` με αυτή την παράμετρο. Στη συνέχεια, μόνο αυτό το νήμα μπορεί να καλέσει συναρτήσεις του MPI.

- `MPI_THREAD_SERIALIZED`: Αν η εφαρμογή αποτελείται από πολλά νήματα, θα πρέπει να εξασφαλίσουμε με μία λύση αμοιβαίου αποκλεισμού ότι μόνο ένα νήμα κάθε φορά καλεί συναρτήσεις του MPI, δηλαδή να σειριοποιήσουμε μόνοι μας τις κλήσεις στο MPI.

- `MPI_THREAD_MULTIPLE`: Πολλαπλά νήματα μπορούν να καλούν συναρτήσεις του MPI ταυτόχρονα.

Στις πρώτες εκδόσεις του OpenMPI (που είναι η υλοποίηση του MPI που χρησιμοποιούμε) η υποστήριξη για παράλληλες κλήσεις υπήρχε μόνο αν ο χρήστης έδινε την παράμετρο `--enable-mpi-thread-multiple` όταν έκανε διαμόρφωση της μετάφρασης (compile configuration) του OpenMPI. Σε διαφορετική περίπτωση, η αρχικοποίηση του MPI με την παράμετρο `MPI_THREAD_MULTIPLE` θα επέστρεφε ένα κωδικό σφάλματος. Παρόλο αυτά, από την έκδοση 3.0 και μετά η υποστήριξη για παράλληλες κλήσεις συμπεριλαμβάνεται σε κάθε περίπτωση, άσχετα με το πώς μεταφράστηκε το OpenMPI, γι αυτό και εμείς τη θεωρούμε δεδομένη.

Η αρχικοποίηση του MPI με την επιλογή `MPI_THREAD_MULTIPLE` δε λύνει όμως όλα μας τα προβλήματα. Σε αρκετές περιπτώσεις η επικοινωνία με μία συσκευή γίνεται σε δύο φάσεις. Στην πρώτη καλούμε την `MPI_Send()` για να ενημερώσουμε τη συσκευή ποια δεδομένα θέλουμε να μας στείλει και στη δεύτερη καλούμε την `MPI_Recv()` για να λάβουμε αυτά τα δεδομένα. Αν δύο νήματα στον κόμβο–host κάνουν αυτή την επικοινωνία με την *ίδια* συσκευή ταυτόχρονα, το ένα μπορεί να λάβει τα δεδομένα που προορίζονταν για το άλλο με ολέθριες συνέπειες. Θα μπορούσαμε να λύσουμε αυτό το πρόβλημα χρησιμοποιώντας την παράμετρο *tag* στις

κλήσεις `MPI_Send()` και `MPI_Recv()`, αλλά αποφασίσαμε να την αφήσουμε για μελλοντική χρήση. Αντί αυτού, κάθε φορά που θέλουμε να καλέσουμε μία συνάρτηση του MPI ελέγχουμε αν βρισκόμαστε σε παράλληλη περιοχή και, αν αυτό ισχύει, κλειδώνουμε μία κλειδαριά αμοιβαίου αποκλεισμού (mutex). Μόλις ολοκληρώσουμε τις κλήσεις στο MPI για τη δουλειά που θέλουμε, ξεκλειδώνουμε την κλειδαριά. Αυτό γίνεται με τις μακροεντολές (macros) `MPI_PAR_LOCK` και `MPI_PAR_UNLOCK`. Έχουμε μία κλειδαριά για κάθε συσκευή, μιας και το πρόβλημα προκύπτει μόνο αν θέλουμε να χρησιμοποιήσουμε ταυτόχρονα την ίδια συσκευή. Διαφορετικά, τα μηνύματα δεν μπλέκονται μεταξύ τους γιατί όταν καλούμε την `MPI_Recv()` θέτουμε ως πηγή την *τάξη (rank)* της διεργασίας που αντιστοιχεί στη συσκευή που θέλουμε.

Το τελευταίο σημείο που χρήζει προσοχής είναι οι κοινόχρηστοι πίνακες στους οποίους αποθηκεύουμε τις καθολικές μεταβλητές και τις μεταβλητές που αποστέλλονται στις συσκευές, όπως είδαμε στην Ενότητα 4.4. Δύο νήματα στον κόμβο–host μπορεί να θέλουν να προσθέσουν ή να αφαιρέσουν μεταβλητές στον ίδιο πίνακα, αν επικοινωνούν με τον ίδιο κόμβο–συσκευή, δημιουργώντας συνθήκη ανταγωνισμού. Ευτυχώς αυτό το πρόβλημα δεν είναι δύσκολο να αντιμετωπιστεί. Επεκτείνουμε τη χρήση της κλειδαριάς που χρησιμοποιούμε για τις κλήσεις του MPI ώστε η περιοχή που κλειδώνεται να συμπεριλαμβάνει και τις κλήσεις που προσθέτουν ή αφαιρούν μεταβλητές από αυτούς τους πίνακες.

4.6 Παρόμοια έρευνα

Η παρούσα διπλωματική εργασία δεν είναι η μοναδική που πραγματεύεται τη χρήση κόμβων ενός cluster ως συσκευές στο OpenMP. Στη βιβλιογραφία εντοπίσαμε ακόμα δύο εργασίες τις οποίες θα παρουσιάσουμε συνοπτικά σε αυτή την ενότητα. Η πρώτη [6] θεωρεί ολόκληρο το cluster ως μία συσκευή στην οποία ο προγραμματιστής μπορεί να αποστείλει κώδικα και δεδομένα. Πρόκειται για μία προσέγγιση εντελώς διαφορετική από τη δική μας, αφού χρησιμοποιεί την τεχνολογία MapReduce για το διαμοιρασμό της δουλειάς στους επιμέρους κόμβους, ενώ η αποστολή και λήψη μεταβλητών γίνεται γράφοντας και διαβάζοντας αρχεία στο κατανεμημένο σύστημα αρχείων Hadoop Distributed File System (HDFS).

Η δεύτερη εργασία [7] έχει την ίδια προσέγγιση με εμάς, δηλαδή θεωρεί κάθε κόμβο του cluster ως ξεχωριστή συσκευή, ενώ για την επικοινωνία και το διαμοιρα-

σμό της δουλειάς χρησιμοποιεί το MPI. Η ανάπτυξη έγινε ως επέκταση για το μεταφραστή clang. Το βασικό μειονέκτημα της εργασίας είναι ότι αποτελεί εσωτερικό πρότζεκτ της IBM και ο πηγαίος κώδικας δεν είναι διαθέσιμος για το ευρύ κοινό ώστε να προχωρήσουμε σε σύγκριση απόδοσης. Επίσης, οι δημιουργοί της επισημαίνουν ότι δεν έχουν προβεί σε καμία προσπάθεια βελτιστοποιήσεων και μείωσης των επικοινωνιών. Τέλος, δεν μπορέσαμε να βρούμε κάποια νεώτερη ενημέρωση για την εργασία, πέρα από την αρχική δημοσίευση του 2015.

Κεφαλαιο 5

Πειραματικα αποτελεσματα

5.1 Εισαγωγή και μεθοδολογία

Έχοντας ολοκληρώσει την υλοποίηση της μονάδας μας, προχωρήσαμε στην εκτέλεση πειραμάτων, τα οποία χωρίζονται σε δύο κατηγορίες. Από τη μία πλευρά τρέξαμε πειράματα ορθότητας, τα οποία ελέγχουν αν η επικοινωνία με τη συσκευή πραγματοποιείται σωστά και σύμφωνα με αυτά που ορίζει το πρότυπο OpenMP, ακόμα και στις πιο οριακές περιπτώσεις.

Αφού βεβαιωθήκαμε ότι όλα δουλεύουν σωστά ασχοληθήκαμε με τη δεύτερη κατηγορία πειραμάτων: τα πειράματα επίδοσης. Χρησιμοποιήσαμε τα προγράμματα που παρέχονται στο πακέτο BOTS[1] [8] και συγκεκριμένα τα alignment, fib, και sparselu. Το fib έχει μία σημαντική διαφορά σε σχέση με τα υπόλοιπα και αυτή είναι ότι χρησιμοποιεί αναδρομή για να υπολογίσει την τελική λύση.

[1]Barcelona OpenMP Task Suite, διαθέσιμα δωρεάν στη διεύθυνση `https://github.com/bsc-pm/bots`

5.1.1 Τροποποίηση των εφαρμογών BOTS

Όλες οι εφαρμογές που περιλαμβάνονται στο πακέτο BOTS είναι σχεδιασμένες έτσι ώστε να χρησιμοποιούν *tasks* του OpenMP [9] για να κάνουν παράλληλους υπολογισμούς σε έναν μόνο υπολογιστή. Εμείς θέλαμε να τις εκτελέσουμε σε πολλούς υπολογιστές και για να γίνει αυτό ήταν αναγκαίο να τις τροποποιήσουμε εισάγοντας και οδηγίες *target*. Με άλλα λόγια, αυτό που κάναμε ήταν να προσθέσουμε οδηγίες *target* ώστε να "σπάσουμε" το αρχικό πρόβλημα σε ίσα υποπροβλήματα τα οποία μοιράζουμε στους διαθέσιμους κόμβους–συσκευές. Ως αποτέλεσμα, έχουμε παραλληλοποίηση σε δύο επίπεδα, πράγμα απαραίτητο για να επιτύχουμε τη μέγιστη δυνατή απόδοση. Πρώτων, ο κόμβος–host χωρίζει τη δουλειά σε ίσα μέρη και ζητάει παράλληλα από όλους τους κόμβους–συσκευές να υπολογίσουν το κομμάτι που τους αντιστοιχεί (κάνοντας χρήση της οδηγίας *target*). Δεύτερον, κάθε κόμβος–συσκευή εκτελεί τους kernels παράλληλα δημιουργώντας πολλαπλά νήματα για να υπολογίσει το αποτέλεσμα που του ανατέθηκε (κάνοντας χρήση της οδηγίας *task*).

Τα προγράμματα alignment και sparselu παράγουν ως έξοδο έναν πίνακα. Κάναμε αλλαγές ώστε αν έχουμε n συσκευές και ο πίνακας αποτελείται από m στοιχεία, κάθε συσκευή να υπολογίζει $\frac{m}{n}$ στοιχεία του αποτελέσματος. Επίσης πριν από τον υπολογισμό, στέλνουμε σε όλες τις συσκευές τις βοηθητικές μεταβλητές και πίνακες που χρειάζονται. Δώσαμε ιδιαίτερη έμφαση στον περιορισμό των επικοινωνιών όπου αυτό είναι δυνατόν. Στέλνουμε στους κόμβους–συσκευές μόνο τις μεταβλητές και τα στοιχεία του πίνακα που πραγματικά χρειάζονται για να υπολογίσουν το μέρος του προβλήματος που τους έχει ανατεθεί.

Το πρόγραμμα fib υπολογίζει έναν αριθμό Fibonacci αναδρομικά. Εδώ ακολουθήσαμε μία διαφορετική προσέγγιση. Ο περιορισμός που είχαμε προέρχονταν από το γεγονός ότι το OpenMP επιτρέπει μόνο στον κόμβο–host να στέλνει δουλειά σε ένα κόμβο–συσκευή. Μία συσκευή δεν μπορεί να στείλει δουλειά σε μία άλλη συσκευή, άρα ο κόμβος–host είναι αναγκασμένος να εκτελέσει τις πρώτες αναδρομικές κλήσεις. Μόλις η αναδρομή ξετυλιχτεί αρκετά ώστε ο αριθμός των αναδρομικών κλήσεων που προκύπτουν να είναι ίσος με το πλήθος των κόμβων–συσκευών, ο κόμβος–host ζητάει από τις συσκευές να τις εκτελέσουν και περιμένει για το αποτέλεσμα.

5.1.2 Τροποποίηση της εφαρμογής Mandelbrot

Επίσης χρησιμοποιήσαμε το πρόγραμμα mandelbrot[2], το οποίο δημιουργεί μία εικόνα με το Mandelbrot fractal, οι διαστάσεις της οποίας δηλώνονται ως σταθερές. Η αρχική υλοποίηση ήταν εντελώς σειριακή. Η προσέγγισή μας ήταν να χωρίσουμε την εικόνα, που είναι ένας διδιάστατος πίνακας, σε ίσους διδιάστατους υποπίνακες με μέγεθος τέτοιο, ώστε το πλήθος των υποπινάκων να ισούται με το πλήθος των διαθέσιμων κόμβων–συσκευών. Επιπρόσθετα, παραλληλοποιήσαμε τους υπολογισμούς ώστε το αποτέλεσμα κάθε υποπίνακα να υπολογίζεται παράλληλα.

5.2 Τεχνικά χαρακτηριστικά συστήματος

Όλες οι μετρήσεις πραγματοποιήθηκαν στο cluster του τμήματος. Το cluster διαθέτει 16 κόμβους (μαζί με τον master node) SUN Fire X4100 Servers με 2 CPUs ο καθένας (Dual Core AMD Opteron στα 2193 MHz η κάθε CPU) και 12GB μνήμης. Το δίκτυο μεταξύ των κόμβων είναι στα 1000Mbps ενώ η εξωτερική σύνδεση είναι στα 100Mbps. Επίσης, κάθε κόμβος τρέχει Ubuntu 16.04.4 LTS με λειτουργικό σύστημα GNU/Linux 4.4.0. Ο μεταφραστής C που χρησιμοποιήθηκε είναι ο GCC 5.4.0.

Για να μετρήσουμε την απόδοση της συσκευής μας χρησιμοποιήσαμε από 2 μέχρι 12 κόμβους του cluster, ανάλογα με το πείραμα. Μετρήσαμε το όφελος που έχουμε σε σχέση με την παράλληλη εκτέλεση σε ένα μόνο υπολογιστή (όπου εκτελέσαμε κάθε πείραμα χρησιμοποιώντας μόνο ένα τυχαίο κόμβο του cluster), δηλαδή όπως ήταν αρχικά σχεδιασμένα τα πειράματα BOTS. Εκτελέσαμε κάθε πείραμα 5 φορές και κρατήσαμε τον ελάχιστο χρόνο. Στις επόμενες ενότητες παρουσιάζουμε και σχολιάζουμε τα αποτελέσματα των μετρήσεων για κάθε πείραμα.

5.3 Protein Alignment

Το πρόγραμμα align πραγματοποιεί ευθυγράμμιση πρωτεΐνης. Από προγραμματιστικής άποψης, παράγει ως αποτέλεσμα ένα μονοδιάστατο πίνακα, κάθε στοιχείο του οποίου μπορεί να υπολογιστεί ανεξάρτητα από τα υπόλοιπα. Για τον υπολογισμό απαιτούνται εκτεταμένες πράξεις με βοηθητικούς πίνακες και μεταβλητές, που

[2]διανέμεται δωρεάν στη διεύθυνση `www2.imm.dtu.dk/~bd/DCAMM09/Labs/mandelbrot.zip`

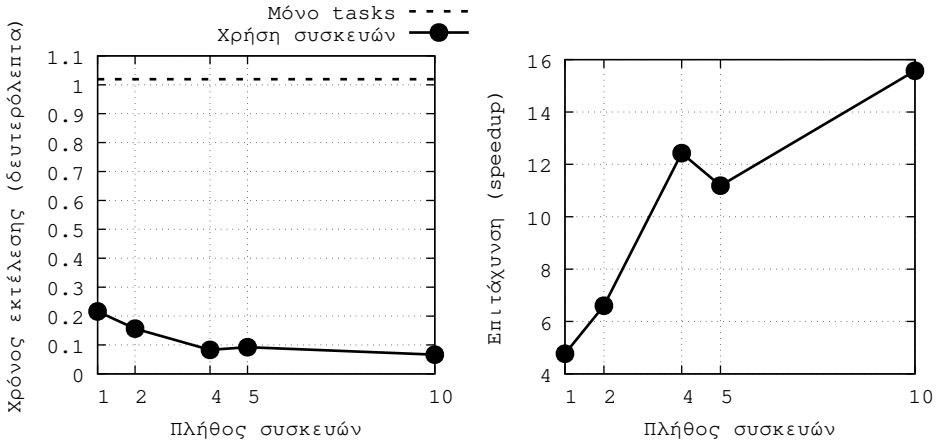
δε μεταβάλλονται κατά τη διάρκεια της εκτέλεσης και άρα μπορούν να σταλούν μία φορά σε κάθε συσκευή.

Στον Πίνακα 5.1 φαίνεται ο χρόνος εκτέλεσης του προγράμματος align για το αρχείο εισόδου `prot20.aa` και στο Σχήμα 5.1 η αντίστοιχη γραφική παράσταση. Στον πίνακα υπάρχει και ο χρόνος της σειριακής εκτέλεσης, που χρησιμοποιεί ένα νήμα σε ένα κόμβο, καθώς και της χρήσης μόνο OpenMP tasks, που χρησιμοποιεί 4 νήματα σε ένα κόμβο. Στις υπόλοιπες περιπτώσεις, το συνολικό πλήθος νημάτων που χρησιμοποιούνται για τους υπολογισμούς είναι 4 επί το πλήθος των συσκευών που έχουμε. Αυτή η είσοδος παράγει ως έξοδο ένα πίνακα με 400 στοιχεία. Η καθυστέρηση που έχουμε λόγω επικοινωνιών είναι μικρή αφού η ποσότητα των επικοινωνιών είναι μικρή, αλλά κάθε κόμβος–συσκευή έχει αρκετή δουλειά να εκτελέσει. Επιπλέον, το μεγαλύτερο τμήμα των επικοινωνιών συμβαίνει μία φορά στην αρχή του προγράμματος. Ως αποτέλεσμα, η επιτάχυνση (speedup) που παίρνουμε, την οποία ορίζουμε ως τον ελάχιστο χρόνο εκτέλεσης κάνοντας χρήση μόνο tasks διά τον ελάχιστο χρόνο εκτέλεσης κάνοντας χρήση 10 συσκευών, είναι εντυπωσιακή και αγγίζει το 17.16.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Σειριακό	0.891605	0.891578	1.779725	1.779232	1.779649	0.891578
Μόνο tasks	1.033148	1.112053	1.13175	1.128268	1.077235	1.033148
1 συσκευή	0.217113	0.216566	0.219556	0.21729	0.220294	0.216566
2 συσκευές	0.15739	0.156893	0.156479	0.157376	0.158667	0.156479
4 συσκευές	0.083901	0.092261	0.093545	0.08311	0.093519	0.08311
5 συσκευές	0.107739	0.092345	0.093189	0.103673	0.099331	0.092345
10 συσκευές	0.069328	0.066352	0.067413	0.08382	0.068268	0.066352

Πίνακας 5.1: Χρόνος εκτέλεσης προγράμματος alignment για μέγεθος εισόδου 20.

Στον Πίνακα 5.2 φαίνεται ο χρόνος εκτέλεσης του προγράμματος align για το αρχείο εισόδου `prot100.aa` και στο Σχήμα 5.2 η αντίστοιχη γραφική παράσταση. Αυτή η είσοδος παράγει ως έξοδο ένα πίνακα με 10000 στοιχεία. Σε σχέση με πριν, το μέγεθος των επικοινωνιών αυξάνεται αρκετά, αλλά το ίδιο συμβαίνει και με το φόρτο εργασίας που πρέπει να υπολογιστεί. Όπως και πριν, ένα σημαντικό μέρος των επικοινωνιών συμβαίνει μόνο μία φορά στην αρχή του προγράμματος. Η επιτάχυνση που παίρνουμε σε σχέση με τη χρήση μόνο tasks σε ένα κόμβο είναι

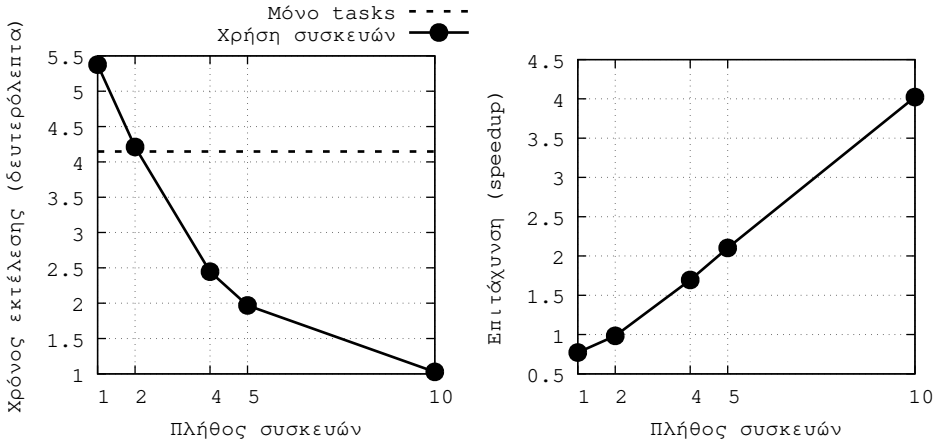


Σχήμα 5.1: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος alignment για μέγεθος εισόδου 20.

4.02.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Σειριακό	22.578211	45.156249	45.139232	22.581157	45.156177	22.578211
Μόνο tasks	5.003553	4.146409	5.161094	4.951626	4.985391	4.146409
1 συσκευή	5.375776	5.378743	5.378781	5.378778	5.37818	5.375776
2 συσκευές	4.212972	4.212003	4.214385	4.215502	4.212514	4.212003
4 συσκευές	2.450787	2.446177	2.452094	2.466572	2.472211	2.446177
5 συσκευές	1.971655	1.97657	1.974328	1.97287	1.97121	1.97121
10 συσκευές	1.030623	1.033209	1.035346	1.042028	1.034387	1.030623

Πίνακας 5.2: Χρόνος εκτέλεσης προγράμματος alignment για μέγεθος εισόδου 100.



Σχήμα 5.2: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος alignment για μέγεθος εισόδου 100.

5.4 Mandelbrot

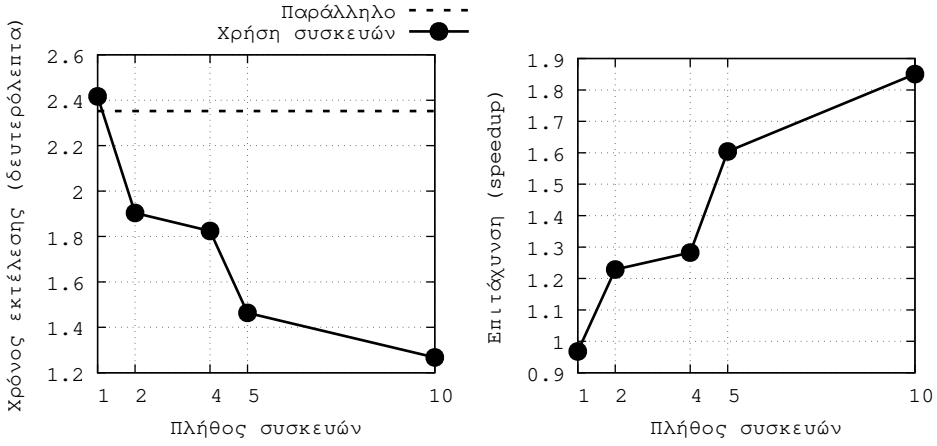
Το πρόγραμμα mandelbrot δημιουργεί μία εικόνα διαστάσεων που ορίζονται από τον χρήστη με το Mandelbrot fractal. Για τον υπολογισμό της τιμής κάθε pixel απαιτείται μόνο η θέση (x, y) του pixel. Στην υλοποίησή μας, κάθε κόμβος–συσκευή κατασκευάζει μία λωρίδα (πλήθος από συνεχόμενες γραμμές) της εικόνας, άρα οι μόνες πληροφορίες που χρειάζεται να γνωρίζει κάθε συσκευή είναι η θέση και το μέγεθος της λωρίδας που της αντιστοιχεί. Αντίθετα, μόλις οι υπολογισμοί τελειώσουν, πρέπει να μεταφερθεί πίσω στον κόμβο–host ολόκληρη η εικόνα και αυτό μπορεί να έχει ως αποτέλεσμα σημαντικό όγκο επικοινωνιών, ανάλογα με τις διαστάσεις της εικόνας.

Στον Πίνακα 5.3 φαίνεται ο χρόνος εκτέλεσης του προγράμματος mandelbrot για μέγεθος εικόνας 2600x2600 pixels και στο Σχήμα 5.3 η αντίστοιχη γραφική παράσταση. Η επιτάχυνση που παίρνουμε σε σχέση με την παράλληλη εκτέλεση σε έναν κόμβο, λόγω των συνθηκών που αναφέραμε παραπάνω, είναι 1.85.

Στον Πίνακα 5.4 φαίνεται ο χρόνος εκτέλεσης του προγράμματος mandelbrot για μέγεθος εικόνας 4600x4600 pixels και στο Σχήμα 5.4 η αντίστοιχη γραφική παράσταση. Παρατηρούμε ότι ο διπλασιασμός των διαστάσεων της εικόνας έχει ως αποτέλεσμα μεγάλη αύξηση στο φόρτο εργασίας που πρέπει να υπολογιστεί (ο χρόνος της σειριακής εκτέλεσης τριπλασιάζεται), ενώ η ποσότητα των επικοινωνιών

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Σειριακό	5.76556	11.4494	5.76506	5.76805	5.76466	5.76466
Παράλληλο	2.37015	2.36878	2.35209	2.36129	2.37241	2.35209
1 συσκευή	2.44902	2.61205	2.43441	2.43485	2.4304	2.4304
2 συσκευές	1.91906	2.055	1.91469	1.93736	2.02725	1.91469
4 συσκευές	1.84204	1.98811	1.89932	1.83494	1.83412	1.83412
5 συσκευές	1.4682	1.60799	1.46788	1.47396	1.46615	1.46615
10 συσκευές	1.29617	1.332	1.2952	1.31829	1.27127	1.27127

Πίνακας 5.3: Χρόνος εκτέλεσης προγράμματος mandelbrot για μέγεθος εικόνας 2600x2600 pixels.

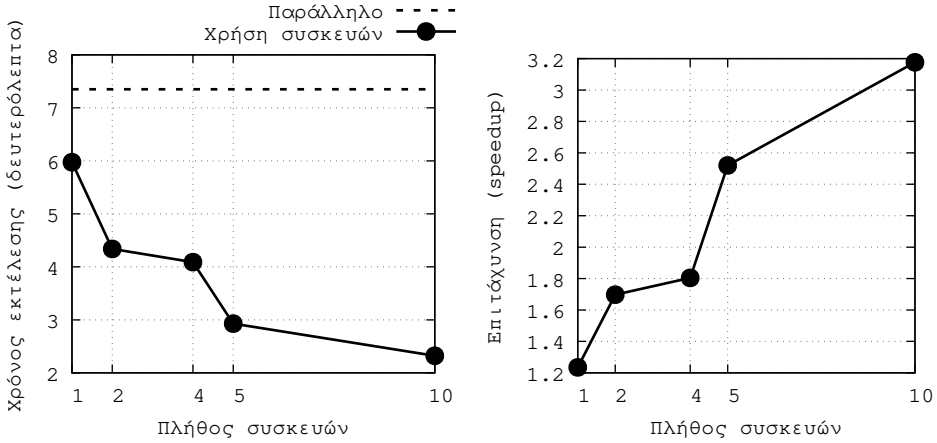


Σχήμα 5.3: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος mandelbrot για μέγεθος εικόνας 2600x2600 pixels.

δεν αυξάνεται τόσο δραματικά. Ως αποτέλεσμα, η επιτάχυνση που παίρνουμε σε σχέση με την παράλληλη εκτέλεση σε έναν κόμβο είναι 3.18.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Σειριακό	18.0148	35.7955	35.8223	35.8096	18.0192	18.0148
Παράλληλο	7.39534	7.43085	7.4048	7.45224	7.37848	7.37848
1 συσκευή	6.00775	6.15459	6.05141	5.98428	5.97394	5.97394
2 συσκευές	4.34595	4.50685	4.44217	4.41002	4.40999	4.34595
4 συσκευές	4.0953	4.25475	4.17118	4.08857	4.25118	4.08857
5 συσκευές	2.95509	3.10169	3.02879	2.92814	2.94559	2.92814
10 συσκευές	2.47345	2.45489	2.32746	2.38808	2.32282	2.32282

Πίνακας 5.4: Χρόνος εκτέλεσης προγράμματος mandelbrot για μέγεθος εικόνας 4600x4600 pixels.



Σχήμα 5.4: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος mandelbrot για μέγεθος εικόνας 4600x4600 pixels.

5.5 Fibonacci

Το πρόγραμμα fib υπολογίζει έναν αριθμό Fibonacci που δίνεται ως παράμετρος από το χρήστη χρησιμοποιώντας μόνο αναδρομικές κλήσεις. Η μόνη επικοινωνία

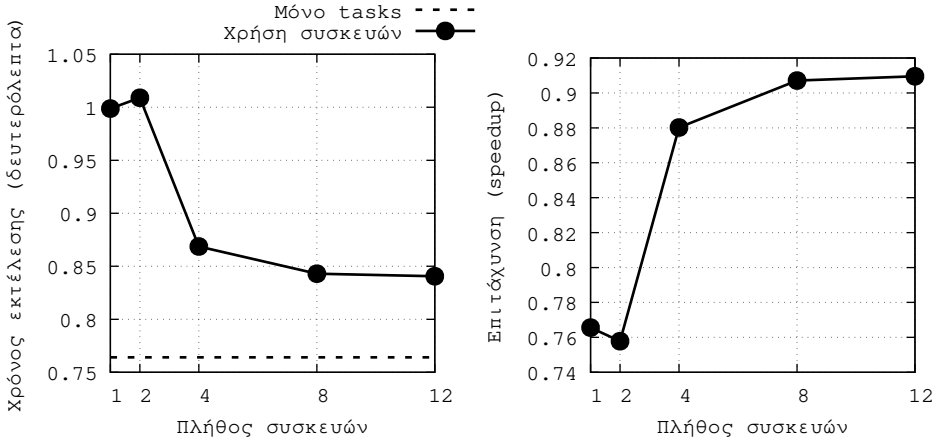
που απαιτείται είναι να σταλεί στον κόμβο–συσκευή ο αριθμός που πρέπει να υπολογιστεί και να μεταφερθεί πίσω ο αριθμός–αποτέλεσμα. Ο κόμβος–συσκευή δημιουργεί ένα OpenMP task για τον υπολογισμό κάθε αναδρομικής κλήσης που προκύπτει. Φυσικά, για να είναι ορθό το τελικό αποτέλεσμα, κάθε task μπορεί να ολοκληρωθεί μόνο όταν ολοκληρωθούν τα επιμέρους tasks που δημιουργεί.

Στον Πίνακα 5.5 φαίνεται ο χρόνος εκτέλεσης του προγράμματος fib για τον αριθμό 35 και στο Σχήμα 5.5 η αντίστοιχη γραφική παράσταση. Παρατηρούμε ότι ο υπολογισμός αυτού του αριθμού στο cluster που διαθέτουμε απαιτεί λιγότερο από ένα δευτερόλεπτο. Η επιτάχυνση που παίρνουμε σε σχέση με τη χρήση μόνο tasks σε ένα κόμβο είναι 0.90, δηλαδή η χρήση tasks σε ένα μόνο υπολογιστή δίνει καλύτερα αποτελέσματα.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Μόνο tasks	0.764696	0.764604	0.856446	0.814474	0.870156	0.764604
1 συσκευή	1.123921	1.272147	1.068866	1.143595	0.998721	0.998721
2 συσκευές	1.051442	1.345602	1.197576	1.169624	1.00901	1.00901
4 συσκευές	0.967558	1.107944	0.868674	0.873337	0.89337	0.868674
8 συσκευές	0.949297	1.023335	0.990892	0.858627	0.842934	0.842934
12 συσκευές	0.930596	1.001042	0.959547	0.845334	0.840655	0.840655

Πίνακας 5.5: Χρόνος εκτέλεσης προγράμματος fibonacci για τον αριθμό 35.

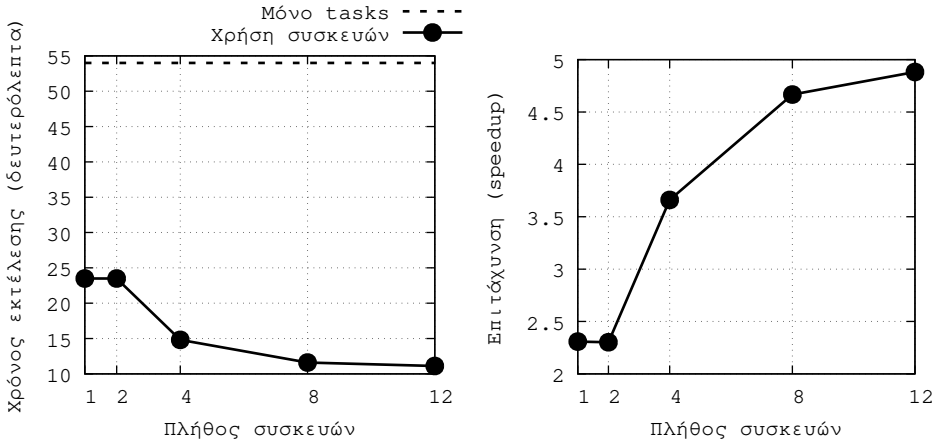
Στον Πίνακα 5.6 φαίνεται ο χρόνος εκτέλεσης του προγράμματος fib για τον αριθμό 45 και στο Σχήμα 5.6 η αντίστοιχη γραφική παράσταση. Σε αυτή την περίπτωση ο υπολογισμός του αποτελέσματος απαιτεί σημαντικά περισσότερο χρόνο. Η επιτάχυνση που παίρνουμε σε σχέση με τη χρήση μόνο tasks σε ένα κόμβο είναι 4.88.



Σχήμα 5.5: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος fibonacci για τον αριθμό 35.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Μόνο tasks	54.602174	55.238012	62.430833	54.218168	57.104851	54.218168
1 συσκευή	25.013764	24.849549	23.635138	24.783836	23.494296	23.494296
2 συσκευές	23.556062	24.738517	23.634319	23.719618	24.685208	23.556062
4 συσκευές	15.691091	15.656713	14.962685	14.811686	15.078353	14.811686
8 συσκευές	12.06595	11.678711	11.770469	11.656473	11.619998	11.619998
12 συσκευές	11.238028	11.186156	11.105935	11.123865	15.696177	11.105935

Πίνακας 5.6: Χρόνος εκτέλεσης προγράμματος fibonacci για τον αριθμό 45.



Σχήμα 5.6: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος fibonacci για τον αριθμό 45.

5.6 Sparse LU

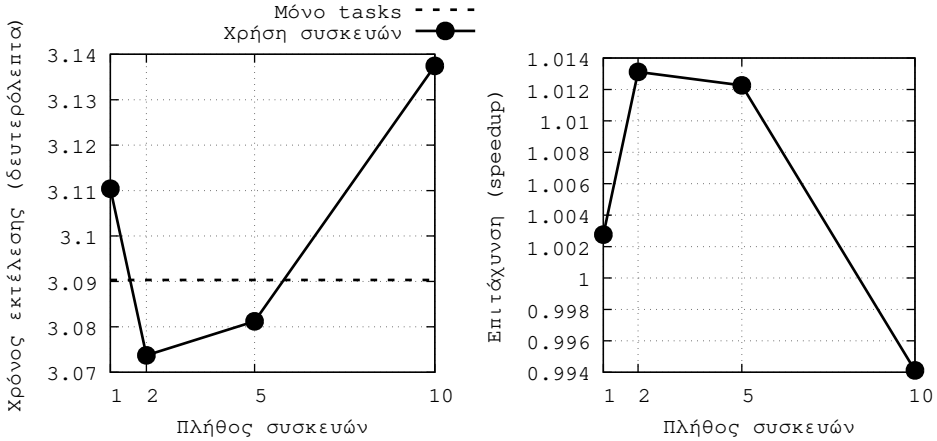
Το πρόγραμμα sparselu πραγματοποιεί ανάλυση LU σε έναν αραιό πίνακα, οι διαστάσεις του οποίου δίνονται ως παράμετροι από το χρήστη. Κάθε συσκευή επεξεργάζεται ένα μέρος του πίνακα. Αυτό έχει ως αποτέλεσμα τον διαχωρισμό του πίνακα σε ίσα μέρη και την αποστολή τους στους επιμέρους κόμβους–συσκευές. Επίσης, όταν τελειώσουν οι υπολογισμοί, τα "κομμάτια" του αποτελέσματος πρέπει να αποσταλούν πίσω στον κόμβο–host. Όλο αυτό προκαλεί μεγάλες ανάγκες για επικοινωνία (αφού στην ουσία μεταφέρεται δύο φορές ολόκληρος ο πίνακας), οι οποίες εντείνονται όσο αυξάνονται οι διαστάσεις του πίνακα.

Στον Πίνακα 5.7 φαίνεται ο χρόνος εκτέλεσης του προγράμματος sparselu για πίνακα διαστάσεων 2500x10000 και στο Σχήμα 5.7 η αντίστοιχη γραφική παράσταση. Λόγω των σοβαρών καθυστερήσεων που έχουμε εξαιτίας των επικοινωνιών, η επιτάχυνση που παίρνουμε σε σχέση με τη χρήση μόνο tasks σε ένα κόμβο είναι 0.99, δηλαδή η χρήση tasks σε ένα μόνο κόμβο δίνει τα ίδια αποτελέσματα με τη χρήση δέκα κόμβων.

Στον Πίνακα 5.8 φαίνεται ο χρόνος εκτέλεσης του προγράμματος sparselu για πίνακα διαστάσεων 3600x14400 και στο Σχήμα 5.8 η αντίστοιχη γραφική παράσταση. Η κατάσταση είναι και σε αυτή την περίπτωση η ίδια, δηλαδή οι εκτενείς επικοινωνίες καταστρέφουν κάθε περιθώριο βελτίωσης. Η επιτάχυνση που παίρ-

	Εκτέλεση 1	Εκτέλεση 2	Εκτέλεση 3	Εκτέλεση 4	Εκτέλεση 5	Ελάχιστη
Σειριακό	4.652363	9.299203	9.29167	9.295705	9.293791	4.652363
Μόνο tasks	3.132892	3.130672	3.200632	3.181747	3.118998	3.118998
1 συσκευή	3.142349	3.159858	3.220691	3.160969	3.110411	3.110411
2 συσκευές	3.120169	3.16728	3.078618	3.12141	3.249875	3.078618
5 συσκευές	3.127315	3.240779	3.178126	3.081225	3.148009	3.081225
10 συσκευές	3.189367	3.164027	3.179948	3.139842	3.137443	3.137443

Πίνακας 5.7: Χρόνος εκτέλεσης προγράμματος sparselu για μέγεθος πίνακα 2500x10000.



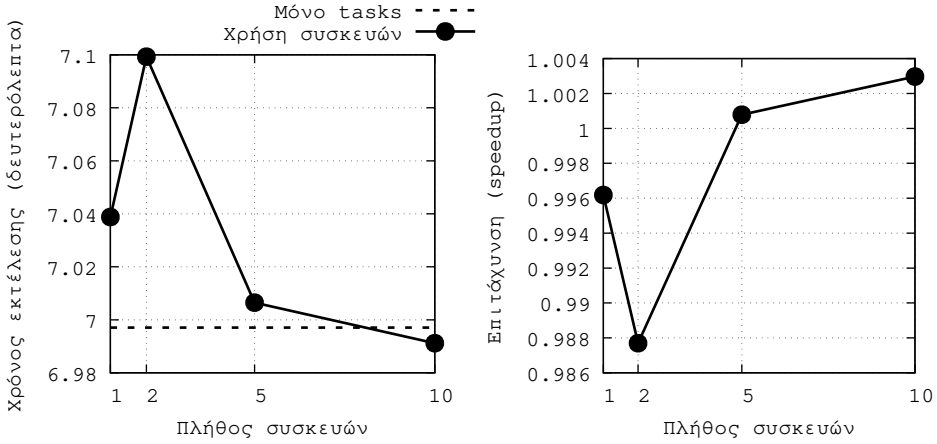
Σχήμα 5.7: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος sparselu για μέγεθος πίνακα 2500x10000.

νουμε σε σχέση με τη χρήση μόνο tasks σε ένα κόμβο είναι 1.00, δηλαδή η χρήση tasks σε ένα μόνο κόμβο δίνει τα ίδια αποτελέσματα με τη χρήση δέκα κόμβων.

Σε αυτό το σημείο, βέβαια, αξίζει να επισημάνουμε ότι λόγω της σχέσης επικοινωνιακού και υπολογιστικού όγκου, γενικότερα η χρήση του cluster δεν ενδείκνυται για τέτοιου είδους εφαρμογές.

	Εκτέλεση 1	**Εκτέλεση 2**	**Εκτέλεση 3**	**Εκτέλεση 4**	**Εκτέλεση 5**	**Ελάχιστη**
Σειριακό	13.023721	26.042456	26.032697	26.038226	26.040044	13.023721
Μόνο tasks	7.196103	7.011933	7.221595	7.092112	7.224736	7.011933
1 συσκευή	7.071719	7.038803	7.111595	7.048105	7.161645	7.038803
2 συσκευές	7.286876	7.166981	7.332088	7.099286	7.117131	7.099286
5 συσκευές	7.109663	7.187128	7.03352	7.249535	7.006455	7.006455
10 συσκευές	7.086498	7.106825	6.991172	7.060363	6.994883	6.991172

Πίνακας 5.8: Χρόνος εκτέλεσης προγράμματος sparselu για μέγεθος πίνακα 3600x14400.



Σχήμα 5.8: Γραφική παράσταση του χρόνου εκτέλεσης και της επιτάχυνσης του προγράμματος sparselu για μέγεθος πίνακα 3600x14400.

Κεφαλαιο 6

Επιλογος

6.1 Σύνοψη διπλωματικής εργασίας

Στην παρούσα διπλωματική εργασία ξεκινήσαμε διαπιστώνοντας την ανάγκη για υπολογιστικά συστήματα με ολοένα και μεγαλύτερη υπολογιστική ισχύ, γεγονός που είχε ως αποτέλεσμα την ευρεία διάδοση παράλληλων και κατανεμημένων συστημάτων. Στη συνέχεια, αναλύσαμε το πρότυπο OpenMP που επιτρέπει εύκολη και σταδιακή παραλληλοποίηση σειριακών προγραμμάτων εστιάζοντας σε μία δυνατότητα που προστέθηκε πρόσφατα: την εκτέλεση τμημάτων κώδικα σε συσκευές πέραν την βασικής CPU, όπως είναι για παράδειγμα οι κάρτες γραφικών. Έπειτα, μελετήσαμε τον παραλληλοποιητικό μεταφραστή OMPi και πιο συγκεκριμένα τον τρόπο λειτουργίας του και τις εσωτερικές δομές του που υποστηρίζουν την προαναφερθείσα δυνατότητα του OpenMP.

Στο πλαίσιο αυτής της διπλωματικής εργασίας, σχεδιάσαμε και υλοποιήσαμε πλήρως μία νέα μονάδα (module) που θεωρεί κάθε κόμβο ενός cluster ως μία ξεχωριστή συσκευή που έχει τη δυνατότητα παράλληλης εκτέλεσης κώδικα, ακόμη και κώδικα που είναι επισημασμένος με επιπλέον οδηγίες OpenMP. Με αυτό τον τρόπο διευκολύνουμε τους προγραμματιστές να "σπάσουν" το πρόγραμμά τους σε κομμάτια, τα οποία θα εκτελούνται παράλληλα στους κόμβους ενός cluster. Παρου-

σιάσαμε τη γενική εικόνα και τον τρόπο διαχείρισης και επικοινωνίας μεταξύ των κόμβων χρησιμοποιώντας τη βιβλιοθήκη MPI. Έπρεπε να λύσουμε θέματα επικοινωνιών, διευθυνσιοδότησης, εκκίνησης των κόμβων–συσκευών κ.λ.π. τα οποία απαίτησαν συνολικές αλλαγές σε όλο το σύστημα υποστήριξης εκτέλεσης του OMPi. Αναλύσαμε σε βάθος κάποια κομβικά σημεία της υλοποίησης που παρουσιάζουν ενδιαφέρον.

Η μονάδα μας πέρασε από εξαντλητικά τεστ ορθότητας και καλής λειτουργίας. Επιπλέον, επανασχεδιάσαμε εφαρμογές οι οποίες κάνουν χρήση tasks εκφράζοντας τες με συναρτήσεις kernels. Οι τελικές εφαρμογές χρησιμοποιήθηκαν για ανάλυση των επιδόσεων της μονάδας. Καταλήξαμε στο συμπέρασμα ότι μπορούμε να δούμε σημαντική επιτάχυνση, εάν εξασφαλίσουμε ότι το πρόγραμμα που εκτελούμε δε χρειάζεται να κάνει εκτεταμένη χρήση επικοινωνιών μεταξύ του κόμβου–host και των κόμβων–συσκευών και αν κάθε κόμβος–συσκευή έχει αρκετούς και χρονοβόρους υπολογισμούς να εκτελέσει.

6.2 Προτάσεις για μελλοντική εργασία

Όπως διαπιστώσαμε, η ανάγκη για εντατική επικοινωνία μεταξύ του κόμβου–host και των κόμβων συσκευών είναι ο βασικός παράγοντας που μειώνει την απόδοση της μονάδας μας. Ως μελλοντική εργασία προτείνουμε τη διεξοδική ανάλυση και ακριβή χρονομέτρηση των επικοινωνιών και την εύρεση τρόπων για την ελάττωση και βελτιστοποίηση ή την αποφυγή τους, όπου αυτό είναι δυνατό.

Επίσης, για να χρησιμοποιηθεί η συσκευή μας, τα υπάρχοντα προγράμματα θα πρέπει να τροποποιηθούν αρκετά, ακόμα και αν είναι ήδη παράλληλα με οδηγίες OpenMP. Όπως είδαμε, ο προγραμματιστής θα πρέπει να μοιράσει "χειροκίνητα" τη δουλειά σε ίσα κομμάτια και να προσέξει ιδιαίτερα ποιες μεταβλητές και ποια στοιχεία κάθε πίνακα χρειάζονται σε κάθε συσκευή. Ως μελλοντική βελτίωση, θα ήταν ιδιαίτερα χρήσιμο να σκεφτούμε επεκτάσεις του OpenMP που να καθιστούν ακόμα πιο εύκολη τη δουλειά του προγραμματιστή εφαρμογών στον τομέα του διαμοιρασμού των δεδομένων. Έτσι, ενδεχομένως θα μπορούσαν να δίνουν τη δυνατότητα αξιοποίησης των συλλογικών επικοινωνιών από τον κόμβο–host στους κόμβους–συσκευές που προσφέρονται από το MPI.

Βιβλιογραφια

[1] S. Huss-Lederman, D. Walker, J. Dongarra, M. Snir, and S. Otto, *MPI: The Complete Reference*. 1996.

[2] S. N. Agathos, V. V. Dimakopoulos, A. Mourelis, and A. Papadogiannakis, "Deploying openmp on an embedded multicore accelerator," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 180–187, July 2013.

[3] "Openmp application programming interface, version 4.5," https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[4] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable c compiler for openmp v.2.0," *In Proc. of the 5th European Workshop on OpenMP (EWOMP '03)*, 2003.

[5] G. C. Philos, V. V. Dimakopoulos, and P. E. Hadjidoukas, "A runtime system architecture for ubiquitous support of openmp," in *2008 International Symposium on Parallel and Distributed Computing*, pp. 189–196, July 2008.

[6] H. Yviquel and G. Araújo, "The cloud as an openmp offloading device," in *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 352–361, Aug 2017.

[7] A. C. Jacob, R. Nair, A. E. Eichenberger, S. F. Antao, C. Bertolli, T. Chen, Z. Sura, K. O'Brien, and M. Wong, "Exploiting fine- and coarse-grained parallelism using a directive based approach," in *OpenMP: Heterogenous Execution and Data Movements* (C. Terboven, B. R. de Supinski, P. Reble, B. M. Chapman, and M. S. Müller, eds.), (Cham), pp. 30–41, Springer International Publishing, 2015.

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in

openmp," in *2009 International Conference on Parallel Processing*, pp. 124–131, Sept 2009.

[9] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Design and implementation of openmp tasks in the ompi compiler," in *2011 15th Panhellenic Conference on Informatics*, pp. 265–269, Sept 2011.

Παραρτημα Α

Απαιτησεις λογισμικου

Για την ορθή μετάφραση (compilation) του OMPi με υποστήριξη για την καινούργια μονάδα (module) που δημιουργήσαμε, έχουμε τις ακόλουθες απαιτήσεις σε λογισμικό:

- automake: `automake` v1.11
- libtool: `libtool` v2.4.6
- `autoconf`
- `bison`
- `flex`
- `hwloc, libhwloc-dev` (προαιρετικό)
- `gcc`
- OpenMPI: `openmpi` v3.0 ή μεταγενέστερη