Υποστήριξη σύγχρονων εκδόσεων του OpenMP σε συστάδες πολυπύρηνων υπολογιστών

Η Μεταπτυχιακή Διπλωματική Εργασία

υποβάλλεται στην ορισθείσα

από την Συνέλευση

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής

Εξεταστική Επιτροπή

από τον

Ηλία Κλεφτάκη

ως μέρος των υποχρεώσεων για την απόκτηση του

ΔΙΠΛΩΜΑΤΟΣ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΣΤΗ ΜΗΧΑΝΙΚΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΜΕ ΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΠΡΟΗΓΜΕΝΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Πανεπιστήμιο Ιωαννίνων

Ιούνιος 2020

Εξεταστική Επιτροπή:

- **Βασίλειος Δημακόπουλος**, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Επιβλέπων)
- **Γεώργιος Μανής**, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Ευαγγελία Πιτουρά**, Καθηγήτρια, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Αφιερωση

Αφιερωμένο στην οικογένειά μου.

Ευχαριστιες

Αρχικά θα ήθελα να ευχαριστήσω τους γονείς μου για την πολύτιμη στήριξη και βοήθεια που μου παρείχαν σε κάθε επίπεδο όλα αυτά τα χρόνια. Επιπλέον, θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή της μεταπτυχιακής αυτής διατριβής, κ. Βασίλειο Δημακόπουλο Αναπληρωτή Καθηγητή, για την καθοδήγηση και τις συμβουλές του κατά την διάρκεια της εκπόνησης της διατριβής.

Η παρούσα εργασία στηρίχθηκε με υπολογιστικό χρόνο από τις Εθνικές Υποδομές για την Έρευνα και την Τεχνολογία (National Infrastructures for Research and Technology – GRNET) και πιο συγκεκριμένα από την Εθνική υπηρεσία HPC (High Performance Computing) – ARIS.

Περιεχομενα

Κατάλογος Σχημάτων

Κατάλογος Προγραμματων

Γλωσσαρι

ArgoDSM Η βιβλιοθήκη sDSM που χρησιμοποιούμε στην παρούσα εργασία.

EELIB (Execution Entity Library) Βιβλιοθήκη οντότητας εκτέλεσης: Βιβλιοθήκη "χαμηλού επιπέδου" που υλοποιεί τη διεπαφή του ORT. Είναι υπεύθυνη για την δημιουργία και διαχείριση οντοτήτων εκτέλεσης (νημάτων, διεργασιών ή συνδυασμών τους) και άλλων θεμελιωδών στοιχείων του παράλληλου προγραμματισμού, όπως barriers και κλειδαριές.

MPI (Message Passing Interface) Δημοφιλής διεπαφή για τον προγραμματισμό συστημάτων κατανεμημένης μνήμης.

OMPi Παραλληλοποιητικός μεταφραστής (compiler) ανοιχτού κώδικα που υποστηρίζει προγράμματα σε C που εμπεριέχουν οδηγίες OpenMP.

OpenMP (Open Multi-Processing) Δημοφιλής διεπαφή για τον προγραμματισμό συστημάτων κοινόχρηστης μνήμης.

ORT (OMPi Runtime) Η βιβλιοθήκη υποστήριξης χρόνου εκτέλεσης του μεταφραστή OMPi.

sDSM (software Distributed Shared Memory) Κατανεμημένη κοινόχρηστη μνήμη υλοποιημένη σε λογισμικό: βιβλιοθήκη που δίνει την ψευδαίσθηση της ύπαρξης κοινόχρηστης μνήμης σε συστήματα που αυτή δεν υπάρχει, όπως για παράδειγμα οι συστάδες (clusters).

TORC Η βιβλιοθήκη κατανεμημένου tasking που χρησιμοποιούμε στην παρούσα εργασία.

Περιληψη

Ηλίας Κλεφτάκης, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2020.
Υποστήριξη σύγχρονων εκδόσεων του OpenMP σε συστάδες πολυπύρηνων υπολογιστών.
Επιβλέπων: Βασίλειος Δημακόπουλος, Αναπληρωτής Καθηγητής.

Οι σύγχρονες υπολογιστικές συστάδες (clusters) είναι κατανεμημένα συστήματα που αποτελούνται από επιμέρους κόμβους, καθένας από τους οποίους είναι ένα μικρό σύστημα κοινόχρηστης μνήμης. Η αποτελεσματική χρήση τους, επομένως, απαιτεί από τον προγραμματιστή να κατανείμει τα δεδομένα και τους υπολογισμούς στους επιμέρους κόμβους σε πρώτο στάδιο και στους πυρήνες και τα νήματα που αποτελούν κάθε κόμβο σε δεύτερο στάδιο. Ως αποτέλεσμα, η διαδικασία είναι δύσκολη και επιρρεπής σε σφάλματα.

Το OpenMP είναι ένα απλό και εύχρηστο πρότυπο για τον παράλληλο προγραμματισμό συστημάτων κοινόχρηστης μνήμης. Στο παρελθόν είχαν γίνει προσπάθειες επέκτασής του ώστε να είναι εφικτή η χρήση του σε κατανεμημένα περιβάλλοντα, αλλά σταμάτησαν στις αρχικές εκδόσεις του προτύπου, πολύ πριν το OpenMP αποκτήσει τις δυνατότητες που το έκαναν δημοφιλές και ισχυρό, όπως είναι τα tasks.

Στην παρούσα εργασία, παρουσιάζουμε την πρώτη προσπάθεια προς την πλήρη και διάφανη εκτέλεση προγραμμάτων OpenMP, τα οποία χρησιμοποιούν όλα τα προηγμένα προγραμματιστικά χαρακτηριστικά του, σε clusters.

Αναλύουμε τον τρόπο λειτουργίας της υλοποίησής μας, δίνοντας έμφαση στα σημαντικότερα προβλήματα που αντιμετωπίσαμε. Αυτά αφορούν τον τρόπο διαχείρισης των κοινόχρηστων μεταβλητών μεταξύ των κόμβων, την ενσωμάτωση των βιβλιοθηκών ArgoDSM και TORC για υποστήριξη sDSM (software Distributed Shared Memory) και κατανεμημένου tasking, αντίστοιχα, και την παροχή κατανεμημένων

κλειδαριών και barriers. Για την αντιμετώπισή τους απαιτήθηκαν τόσο συνολικές αλλαγές στο σύστημα υποστήριξης εκτέλεσης του παραλληλοποιητικού μεταφραστή OMPi, όσο και τροποποιήσεις στον πηγαίο κώδικα των ArgoDSM και TORC.

Στόχος μας είναι να παρέχουμε ένα εύχρηστο προγραμματιστικό μοντέλο για κατανεμημένα περιβάλλοντα και όχι να υποκαταστήσουμε τη χρήση του MPI. Παρόλο αυτά, τα πειράματά μας αποδεικνύουν ότι μπορούμε να πετύχουμε αξιόλογες επιδόσεις, αρκεί τα προγράμματα να μην κάνουν εκτεταμένη χρήση λειτουργιών συγχρονισμού για την εξασφάλιση της συνέπειας μνήμης και, αν χρειάζονται *tasks*, αυτά να δημιουργούνται από όλους τους συμμετέχοντες κόμβους.

Extended Abstract

Ilias Kleftakis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, June 2020.
Deploying recent versions of OpenMP onto multicore clusters.
Advisor: Vassilios Dimakopoulos, Associate Professor.

Modern clusters are distributed systems consisting of individual nodes, that are small shared memory systems. Thus, to program them efficiently, one would need to distribute their data and computations among the cluster's nodes at first phase and then among the cores and threads of each node at second phase. Even though this is a fairly common way to deal with these machines, it is considered difficult and error prone for many programmers, as it requires the use of different techniques and APIs.

OpenMP is a simple and easy to use API for programming shared memory systems. In the past, several studies tried to expand it in order to be used in distributed environments, but they ceased at version 2.5, long before OpenMP obtained the functionalities that made it popular, such as tasks.

In this thesis, we expand OMPi, a research OpenMP to C compiler, to support the transparent execution of parallel programs that contain OpenMP directives in clusters, while requiring little or even no modification to their source code. Moreover, to the best of our knowledge, our implementation is the first to support programs with OpenMP *task* directives, but with some limitations.

Furthermore, we discuss how software Distributed Shared Memory (sDSM) libraries can be used to provide the illusion of a shared memory environment where such does not exist, as for example in clusters. We present the inner workings of the ArgoDSM library and its advantages that made us intergrade it in our implementation. Similarly, we discuss how tasking can be used to allow the execution of

independent functions, or tasks, in a distributed system, how the TORC library works and why we preferred to use it.

We thoroughly analyze our implementation and focus on the main problems we faced. These include managing shared variables across nodes, incorporating ArgoDSM and TORC libraries and providing distributed locks and barriers that confront to the OpenMP requirements. We also show how every component coordinates together when executing parallel regions with reduction operations and when creating and executing tasks. To manage these problems, many changes to OMPi's runtime system were required, as well as modifications to the source code of ArgoDSM and TORC. However, our implementation is still open for extension; we provide two APIs so these libraries can be easily substituted in the future, if need arises.

Our goal is to provide an easy to use programming model for distributed environments and not to substitute the use of MPI. Nonetheless, the experimental results we show prove that we can achieve significant performance, as long as the programs do not make extensive use of synchronization operations to ensure memory coherency and, if OpenMP tasks are needed, they are created by every participant node and not by just one of them.

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 για να αυξήσει την επεξεργαστική ισχύ.

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 smartwatches, 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 και προγραμματισμός 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.

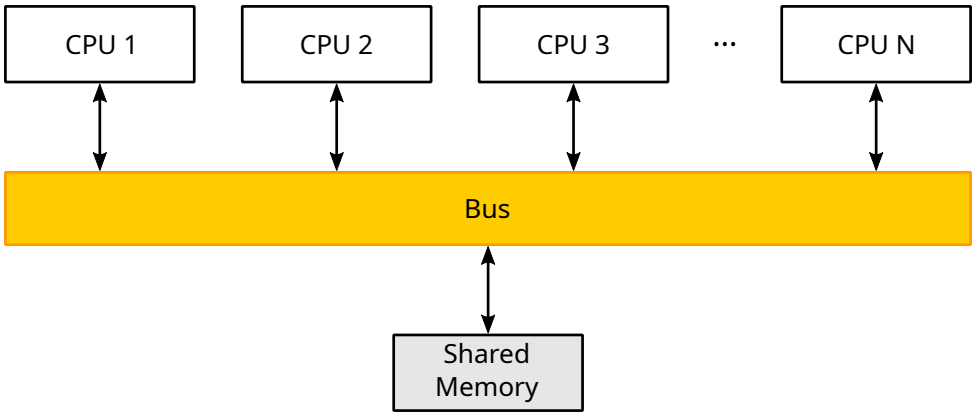
Υπάρχουν δύο βασικές κατηγορίες παράλληλων αρχιτεκτονικών: τα συστήματα κοινόχρηστης μνήμης και τα συστήματα κατανεμημένης μνήμης. Ανάλογα με το σύστημα που διαθέτει, ο προγραμματιστής θα πρέπει να ακολουθήσει διαφορετικές προγραμματιστικές λογικές και μεθόδους. Το ποιο είναι πιο αποδοτικό εξαρτάται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

1.2.1 Συστήματα κοινόχρηστης μνήμης (Shared memory)

Στα συστήματα κοινόχρηστης 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 (bus). Η αρχιτεκτονική κοινόχρηστης μνήμης φαίνεται γραφικά στο Σχήμα 1.1.

Ένας πολυπύρηνος επεξεργαστής είναι περίπου σαν να έχουμε ανεξάρτητους επεξεργαστές. Αν και υπάρχουν διαφορές στην οργάνωση της ιεραρχίας της μνήμης, ένα σύστημα με πολυπύρηνο επεξεργαστή είναι ίσως το πιο συνηθισμένο σύστημα κοινόχρηστης μνήμης που συναντάμε στις μέρες μας.

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 απο-



Σχήμα 1.1: Τυπική αρχιτεκτονική κοινόχρηστης μνήμης. Οι επεξεργαστές επικοινωνούν με τη μνήμη, η οποία είναι κοινόχρηστη από όλους, με τη χρήση ενός διαύλου.

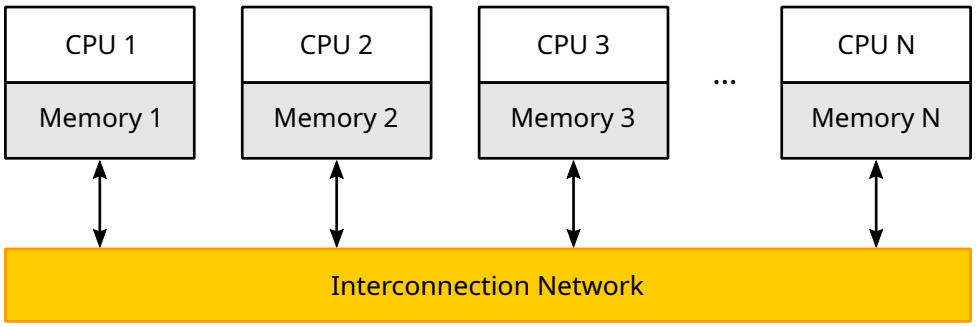
τέλεσμα να καταφύγουμε σε άλλου τύπου διασυνδέσεις, όπως για παράδειγμα τα διακοπτικά δίκτυα πολλαπλών επιπέδων.

Η πιο διαδεδομένη βιβλιοθήκη για τον προγραμματισμό κοινού χώρου διευθύνσεων είναι τα νήματα Posix (Pthreads), τα οποία παρέχουν κλήσεις για την 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 διασύνδεσης (interconnection network), όπως φαίνεται στο Σχήμα 1.2. 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.



Σχήμα 1.2: Αρχιτεκτονική κατανεμημένης μνήμης. Κάθε επεξεργαστής έχει τη δική του ιδιωτική μνήμη. Η επικοινωνία με άλλους επεξεργαστές γίνεται μέσω ενός δικτύου διασύνδεσης.

Ένα τέτοιο σύστημα μπορεί να είναι περισσότερο κλιμακώσιμο σε σχέση με ένα σύστημα κοινόχρηστης 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 και σχετικά δύσκολο στον προγραμματισμό.

Στην πράξη, τα συστήματα που συναντάμε στις μέρες μας είναι συνήθως συνδυασμός των δύο κατηγοριών: συστήματα κοινόχρηστης 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 κοινόχρηστης μνήμης (π.χ. με πολυπύρηνους επεξεργαστές).

1.3 Αντικείμενο διπλωματικής εργασίας

Όπως αναφέραμε στην προηγούμενη ενότητα, οι σύγχρονες συστάδες υπολογιστών (clusters) είναι συστήματα κατανεμημένης μνήμης που αποτελούνται από υπολογιστικούς κόμβους, που είναι μικρά συστήματα κοινόχρηστης μνήμης. Η εκμετάλλευσή τους, επομένως, απαιτεί συνδυασμό πολύπλοκων τεχνικών και προτύπων. Στην πράξη σήμερα είναι πολύ συνηθισμένο να προγραμματίζονται με ένα συνδυασμό από MPI και OpenMP. Αυτό, όμως, έχει ένα σοβαρό πρόβλημα: κάποιος που θέλει να προγραμματίσει επιτυχώς μια συστάδα θα πρέπει να έχει σημαντικές γνώσεις *και των δύο* προτύπων. Θα πρέπει να σκεφτεί πώς θα κατανείμει τα δεδομένα και τους υπολογισμούς στους επιμέρους κόμβους σε πρώτο στάδιο και στους πυρήνες και τα νήματα που αποτελούν κάθε κόμβο σε δεύτερο στάδιο.

Θα ήταν πολύ βολικό, επομένως, αν μπορούσαμε να βγάλουμε το MPI (που είναι πιο πολύπλοκο) από την εξίσωση, ώστε να προγραμματίζουμε αυτά τα μεγαλύτερα συστήματα σαν να ήταν συστήματα κοινόχρηστης μνήμης, δηλαδή χρησιμοποιώντας *μόνο* το OpenMP. Το OpenMP, ωστόσο, αν και πολύ δημοφιλές, *δεν* έχει σχεδιαστεί για clusters και γενικότερα για συστήματα κατανεμημένης μνήμης. Είναι, όμως, τόσο απλό και εύχρηστο, ακόμα και από μη έμπειρους προγραμματιστές, που έγινε προσπάθεια να χρησιμοποιηθεί αυτούσιο και σε clusters. Δηλαδή, ένα πρόγραμμα που είναι γραμμένο σε OpenMP να μπορεί να εκτελεστεί χωρίς αλλαγές (διαφανώς) και σε ένα cluster.

Οι προσπάθειες αυτές σταμάτησαν στην έκδοση 2.5 του προτύπου, που επικεντρωνόταν στην παραλληλοποίηση απλών βρόχων for. Όμως, στην επόμενη έκδοση του προτύπου εισήχθησαν πολύ σημαντικές προγραμματιστικές δομές που σήμερα χρησιμοποιούνται ευρέως, όπως είναι οι εργασίες (tasks), οι οποίες καθιστούν δυνατή την παραλληλοποίηση εφαρμογών που δεν στηρίζονται μόνο σε βρόχους.

Στόχος της παρούσας εργασίας είναι να διερευνήσει αν είναι δυνατή η χρήση των σύγχρονων εκδόσεων του OpenMP σε clusters, μέσα από μία πλήρη υλοποίηση. Πιο συγκεκριμένα:

- Η επέκταση του παραλληλοποιητικό μεταφραστή OMPi με την δημιουργία μιας καινούργιας βιβλιοθήκης που εκτελεί εφαρμογές OpenMP που περιέχουν tasks διαφανώς σε clusters. Μάλιστα, πρόκειται για τον πρώτο γνωστό σχεδιασμό και υλοποίηση OpenMP που το επιτυγχάνει αυτό.

- Η χρήση της βιβλιοθήκης sDSM ArgoDSM και της βιβλιοθήκης tasking TORC

ώστε να επιτευχθεί ο παραπάνω σκοπός.

- Ο πειραματισμός και χρονομέτρηση προγραμμάτων (benchmarks) σε πραγματικό cluster, ώστε να αναλυθεί η απόδοση και η χρησιμότητα μίας τέτοιας προσέγγισης, καθώς και να προσδιοριστούν οι περιορισμοί και οι αδυναμίες της.

Προφανώς, η υλοποίησή μας δεν μπορεί να υποκαταστήσει τη χρήση του MPI για προγραμματισμό clusters. Αντίθετα, το βασικό της πλεονέκτημα είναι ότι μπορεί να αντλήσει επιδόσεις εκτελώντας σε clusters προγράμματα που δεν είχαν δημιουργηθεί με αυτή την απαίτηση κατά νου, και μάλιστα το καταφέρνει αυτό χωρίς να χρειάζεται η τροποποίησή τους.

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 βιβλιογραφική αναφορά σε προηγούμενες προσπάθειες για εκτέλεση προγραμμάτων OpenMP σε κατανεμημένο περιβάλλον.

- Κεφάλαιο 3: Περιγραφή του παραλληλοποιητικού μεταφραστή OMPi και ανάλυση του τρόπου λειτουργίας του για την παραλληλοποίηση βρόχων *for* και οδηγιών *task*. Γίνεται ακόμη παρουσίαση της υπάρχουσας υποδομής του για την εκτέλεση προγραμμάτων OpenMP σε clusters.

- Κεφάλαιο 4: Σύντομη αναφορά σε υπάρχουσες βιβλιοθήκες sDSM και tasking. Εξήγηση των λόγων που μας οδήγησαν στην επιλογή της ArgoDSM και της TORC, αντίστοιχα, ανάλυση των βασικών πλεονεκτημάτων και του τρόπου λειτουργίας τους.

- Κεφάλαιο 5: Αναλυτική παρουσίαση της νέας βιβλιοθήκης που δημιουργήσαμε για να κάνουμε δυνατή την εκτέλεση εφαρμογών OpenMP που περιέχουν tasks διαφανώς σε clusters. Δίνεται έμφαση στα κομβικά σημεία της υλοποίησης που παρουσιάζουν ενδιαφέρον.

- Κεφάλαιο 6: Περιγραφή των πειραματικών προγραμμάτων (benchmarks) και ανάλυση των αποτελεσμάτων από την εκτέλεσή τους. Αποτίμηση τους κέρδους που έχουμε από τη χρήση της νέας υλοποίησης, καθώς και τους περιορισμούς της.
- Κεφάλαιο 7: Επισκόπηση της διπλωματικής εργασίας και προτάσεις για μελλοντικές βελτιώσεις σχετικά με την εκτέλεση εφαρμογών OpenMP που περιέχουν tasks διαφανώς σε clusters, χρησιμοποιώντας την υποδομή του OMPi.

Κεφαλαιο 2

OpenMP και clusters

2.1 Εισαγωγή στο OpenMP

Η συνεχώς αυξανόμενη χρήση των παράλληλων συστημάτων, κυρίως των συστημάτων κοινόχρηστης μνήμης, τα οποία, με την επικράτηση των πολυπύρηνων επεξεργαστών, βρίσκονται πλέον παντού, 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 είναι αρκετά πιο απαιτητικό στην υλοποίησή του σε σχέση με ένα αντίστοιχο σειριακό πρόγραμμα που κάνει την ίδια δουλειά. Εξαιτίας αυτών των δυσκολιών, έχουν γίνει πολλές προσπάθειες για τη διευκόλυνση συγγραφής παράλληλων προγραμμάτων. Μία από αυτές, ίσως η σημαντικότερη, μιας και στις μέρες μας αποτελεί τον πλέον δημο-

φιλή τρόπο προγραμματισμού συστημάτων κοινόχρηστης μνήμης, είναι το OpenMP (OpenMulti-Processing).

Το OpenMP είναι μια διεπαφή δημιουργίας παράλληλων εφαρμογών σε συστήματα κοινόχρηστης 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. Για παράδειγμα, το OpenMP δεν μπορεί *πάντα* να εξασφαλίσει τη μέγιστη αποδοτικότητα της παραλληλίας σε συστήματα κοινόχρηστης μνήμης. 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 κοινόχρηστες 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 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 τη διαδικασία αυτή.

Η συγγραφή ενός προγράμματος OpenMP θα μπορούσαμε να πούμε ότι είναι κλιμακωτή διαδικασία. Αν από το παράλληλο πρόγραμμα αφαιρέσουμε τις οδηγίες OpenMP, θα προκύψει ένα έγκυρο σειριακό πρόγραμμα, χωρίς να έχει αλλάξει η λειτουργία του. Επομένως, γράφουμε το παράλληλο πρόγραμμα αρχικά όπως θα γράφαμε ένα αντίστοιχο σειριακό και στη συνέχεια προσθέτουμε σιγά–σιγά τις οδηγίες στις περιοχές που θέλουμε να παραλληλοποιήσουμε.

2.3 Οδηγίες OpenMP για C/C++

Σε αυτή την ενότητα θα ασχοληθούμε με τις οδηγίες του OpenMP [2] για τις γλώσσες C/C++. Θα ξεκινήσουμε με τις οδηγίες που υπήρχαν ήδη στην έκδοση 2.5 του προτύπου. Πιο συγκεκριμένα, θα δούμε αναλυτικά την οδηγία *parallel*, που χρησιμοποιείται για την δημιουργία μιας παράλληλης περιοχής, δηλαδή ενός τμήματος κώδικα που εκτελείται ταυτόχρονα από πολλαπλά νήματα. Θα μελετήσουμε τις οδηγίες για διαμοιρασμό έργου, που εξασφαλίζουν ότι κάθε νήμα θα εκτελέσει ένα υποσύνολο της δουλειάς, καθώς και τις οδηγίες για επίτευξη συγχρονισμού μεταξύ των νημάτων. Ακόμη, θα δούμε πώς ορίζουμε ποιες μεταβλητές θα είναι κοινόχρηστες και ποιες ιδιωτικές μέσα στην παράλληλη περιοχή.

Στη συνέχεια, θα μελετήσουμε την οδηγία *task*, που είναι η πιο σημαντική αλλαγή στην έκδοση 3.0 του OpenMP. Μας επιτρέπει να ορίζουμε εργασίες ευέλικτου τύπου, δηλαδή τμήματα κώδικα που μπορούν να εκτελεστούν οποιαδήποτε χρονική στιγμή από οποιοδήποτε νήμα. Είναι ιδιαίτερα χρήσιμες για να παραλληλοποιούμε πιο

πολύπλοκο κώδικα, που δεν αποτελείται απλώς από ένα βρόχο for, του οποίου οι επαναλήψεις μπορούν να υπολογιστούν ανεξάρτητα.

Στις πιο πρόσφατες εκδόσεις του OpenMP, συγκεκριμένα τις 4.0/4.5 και 5.0, έγιναν επιπλέον προσθήκες, που όμως δεν μας απασχολούν άμεσα στην παρούσα εργασία. Η πιο σημαντική είναι η οδηγία *target*, που μας επιτρέπει να εκτελέσουμε ένα μέρος του προγράμματος σε μια συσκευή (device) ή επιταχυντή (accelerator). Οι συσκευές/επιταχυντές που χρησιμοποιούνται κυρίως είναι οι κάρτες γραφικών [3] και ο συνεπεξεργαστής Intel Xeon Phi [4], αν και υποστήριξη υπάρχει για το Parallella board [5], ακόμη και τους κόμβους ενός cluster [6]. Σκοπός της οδηγίας *target* είναι να παρέχει ένα κοινό και ενιαίο πρότυπο προγραμματισμού που αποκρύπτει την ετερογένεια που επικρατεί στα σύγχρονα υπολογιστικά συστήματα, δηλαδή την ύπαρξη υλικού (hardware) που εξ ορισμού προγραμματίζεται τελείως διαφορετικά από ότι ένας κλασσικός επεξεργαστής. Επιπλέον, στις πρόσφατες εκδόσεις υπάρχει η οδηγία *simd* (Single Instruction Multiple Data), που έχει ως σκοπό την εκμετάλλευση των πολλαπλών επιπέδων παραλληλισμού, που υπάρχουν σε έναν σύγχρονο πολυπύρηνο επεξεργαστή.

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`. Γενικά τα `#pragma` είναι εντολές του προεπεξεργαστή της C, οι οποίες κατά βάση αγνοούνται, εκτός και αν μεταφέρουν κάποια πληροφορία προς τον μεταφραστή. Συγκεκριμένα, τα `#pragma omp` αγνοούνται από όποιον μεταφραστή δεν χειρίζεται OpenMP, ενώ τις αντιλαμβάνεται όποιος μεταφράζει OpenMP.

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

3. Προαιρετικά, μπορεί να υπάρχουν φράσεις, 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*. 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*.

Υπάρχουν διαφορετικοί τρόποι για να καθορίσουμε το πλήθος των νημάτων που θα εκτελέσουν μια παράλληλη περιοχή, γεγονός που αναδεικνύει την ευελιξία που προσφέρει το πρότυπο OpenMP. Αρχικά, μπορούμε να ορίσουμε την μεταβλητή περιβάλλοντος (environmental variable) `OMP_NUM_THREADS` από το κέλυφος (shell) που χρησιμοποιούμε πριν την εκτέλεση του προγράμματος. Επίσης, πριν την δημιουργία της παράλληλης περιοχής το πρόγραμμα μπορεί να καλέσει την συνάρτηση `omp_set_num_threads`. Τέλος, κατά την δημιουργία της παράλληλης περιοχής, μπορούμε να χρησιμοποιήσουμε τη φράση `num_threads` της οδηγίας `parallel`. Ακόμη, μπορούμε να αφήσουμε το σύστημα να αποφασίσει αυτόματα ποιο είναι το βέλτιστο πλήθος νημάτων, ορίζοντας την μεταβλητή περιβάλλοντος `OMP_DYNAMIC` ή καλώντας την συνάρτηση `omp_set_dynamic`.

2.3.3 Οδηγίες διαμοίρασης έργου (workshare)

Μέσα σε μια παράλληλη περιοχή, συνήθως δεν είναι θεμιτό να εκτελούν όλα τα νήματα τις ίδιες ακριβώς εντολές στα ίδια δεδομένα. Αυτό επιτυγχάνεται με τις οδηγίες διαμοίρασης έργου, οι πιο σημαντικές από τις οποίες είναι οι παρακάτω:

- **Οδηγία for**: Οι επαναλήψεις του βρόχου for που την ακολουθεί μοιράζονται μεταξύ των νημάτων που εκτελούν την παράλληλη περιοχή. Το ποιο νήμα θα εκτελέσει ποια επανάληψη εξαρτάται από την φράση οδηγίας **schedule**. Πιο συγκεκριμένα, με την επιλογή *static*, που είναι η προεπιλογή, οι επαναλήψεις

χωρίζονται στατικά και διαμοιράζονται κυκλικά στα διαθέσιμα νήματα. Επομένως, κάθε νήμα γνωρίζει εξαρχής ποιες επαναλήψεις του έχουν ανατεθεί. Με την επιλογή *dynamic* οι επαναλήψεις χωρίζονται σε ισομεγέθη κομμάτια και όταν ένα νήμα τελειώσει με ένα κομμάτι, συναγωνίζεται με τα υπόλοιπα για το επόμενο. Αυτό φυσικά απαιτεί συγχρονισμό μεταξύ των νημάτων και δεν συνίσταται σε κατανεμημένα περιβάλλοντα, όπου ο συγχρονισμός είναι ακριβός. Η επιλογή *guided* είναι παρόμοια με την *dynamic*, με την διαφορά ότι το μέγεθος κάθε κομματιού μειώνεται με το χρόνο, ώστε να μειωθεί το κόστος των συχνών συγχρονισμών.

- **Οδηγία sections**: Χρησιμοποιείται ώστε ο χρήστης να μοιράσει δουλειά σε επιμέρους τμήματα. Κάθε τμήμα θα εκτελεστεί από ένα μόνο νήμα. Αν το πλήθος των τμημάτων υπερβαίνει το πλήθος των διαθέσιμων νημάτων, τότε κάποια νήματα αναγκαστικά θα εκτελέσουν πάνω από ένα τμήμα.

- **Οδηγία single**: Μόνο το πρώτο νήμα που θα συναντήσει αυτή την οδηγία θα εκτελέσει τον κώδικα που ακολουθεί. Τα υπόλοιπα θα τον αγνοήσουν, αλλά θα περιμένουν μέχρι το πρώτο νήμα να τελειώσει με την εκτέλεσή του.

- **Οδηγία master**: Είναι παρόμοια με την οδηγία *single*, με τη διαφορά ότι το αρχικό νήμα (δηλαδή το νήμα master, που υπήρχε πριν την παράλληλη περιοχή) θα εκτελέσει τον κώδικα που ακολουθεί. Επίσης, τα υπόλοιπα νήματα δεν θα το περιμένουν μέχρι να τελειώσει.

Τέλος, θα κάνουμε μια εκτενέστερη αναφορά στη **φράση reduction**. Πολλές φορές παραλληλοποιούμε κώδικα που έχει ως σκοπό την παραγωγή μιας ποσότητας που πρέπει να αθροιστεί ή να συνδυαστεί με κάποιον άλλο τρόπο. Τέτοια παραδείγματα αποτελούν ο υπολογισμός ενός αθροίσματος ή η εύρεση ενός μεγίστου. Η οδηγία *reduction* μας βοηθάει να παραλληλοποιήσουμε τέτοιες περιπτώσεις κώδικα. Η σύνταξή της είναι: **reduction(πράξη : λίστα μεταβλητών)**. Κάθε νήμα δημιουργεί ένα τοπικό αντίγραφο κάθε μεταβλητής στη λίστα μεταβλητών. Στη συνέχεια, πραγματοποιεί τη ζητούμενη πράξη (π.χ. πρόσθεση) στο τοπικό του αντίγραφο για το μέρος της δουλειάς που εκτελεί. Στο τέλος, τα τοπικά αντίγραφα συνδυάζονται και ενημερώνεται η τιμή της καθολικής μεταβλητής. Οι επιτρεπόμενες πράξεις είναι: `+`, `-`, `*`, `/`, `and` (λογικό και bit προς bit), `or` (λογικό και bit προς bit), `xor`, `min` και `max`.

Προκειμένου να γίνουν πιο κατανοητές τόσο οι οδηγίες διαμοίρασης έργου γενι-

κότερα, όσο και η οδηγία *reduction* ειδικότερα, σε αυτό το σημείο θα παραθέσουμε ένα απλό παράδειγμα. Ο κώδικας στο Πρόγραμμα 2.1 υπολογίζει το μέγιστο στοιχείο του πίνακα που ορίζεται στη γραμμή 6 (και υποτίθεται πως αρχικοποιείται στη γραμμή 9), χρησιμοποιώντας μια μόνο γραμμή OpenMP.

Πιο συγκεκριμένα, στη γραμμή 11 ορίζουμε μια παράλληλη περιοχή με την οδηγία *parallel* και αφήνουμε το σύστημα χρόνου εκτέλεσης να αποφασίσει το βέλτιστο πλήθος νημάτων για να την εκτελέσουν. Με τη φράση οδηγίας *for* ορίζουμε ότι τα νήματα θα πρέπει να μοιράσουν στατικά τις επαναλήψεις του βρόχου for που ακολουθεί και καθένα να εκτελέσει από ένα κομμάτι. Με τη φράση οδηγίας *reduction* ορίζουμε ότι κάθε νήμα θα δημιουργήσει ένα τοπικό αντίγραφο της μεταβλητής `max_value`, το οποίο θα αρχικοποιηθεί αυτόματα στο μικρότερο δυνατό ακέραιο, και στο τέλος της παράλληλης περιοχής θα κρατήσουμε το αντίγραφο με τη μεγαλύτερη τιμή. Η γραμμή 13, που εκτελείται παράλληλα από όλα τα νήματα, υπολογίζει το μέγιστο στο εύρος στοιχείων του πίνακα που αντιστοιχεί σε κάθε νήμα και το αποθηκεύει στο τοπικό αντίγραφο. Στη γραμμή 15, που βρίσκεται μετά τον βρόχο for και συνεπώς μετά την παράλληλη περιοχή, τυπώνουμε το τελικό αποτέλεσμα. Ο πίνακας `array` είναι κοινόχρηστος μέσα στην παράλληλη περιοχή, παρόλο που δεν τον δηλώνουμε ως *shared*. Σύμφωνα με τους κανόνες του OpenMP, οι μεταβλητές που δηλώνονται πριν από μία παράλληλη περιοχή, είναι κοινόχρηστες σε αυτή, εκτός και αν ο προγραμματιστής υποδείξει κάτι διαφορετικό.

Πρόγραμμα 2.1: Παράλληλος υπολογισμός μέγιστου στοιχείου πίνακα με χρήση του OpenMP.

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(void)
5 {
6   int array[1024];
7   int i, max_value = 0;
8
9   /* Initialize the array here ... */
10
11  #pragma omp parallel for reduction(max:max_value)
12  for (i = 0; i < 1024; ++i)
13    max_value = (array[i] > max_value) ? array[i] : max_value;
14
15  printf("The max_value is %d\n", max_value);
```

```
16   return 0;
17 }
```

2.3.4 Κοινόχρηστες μεταβλητές

Κατά την δημιουργία μιας παράλληλης περιοχής ο προγραμματιστής μπορεί να ορίσει ρητά την πολιτική κοινοχρησίας των μεταβλητών, μέσω κατάλληλων φράσεων. Οι πιο συνηθισμένες πολιτικές είναι οι παρακάτω:

- **Shared (λίστα μεταβλητών)**: Οι μεταβλητές που βρίσκονται στη λίστα μεταβλητών θα είναι κοινόχρηστες, δηλαδή όλα τα νήματα που εκτελούν την παράλληλη περιοχή θα έχουν την δυνατότητα να τις προσπελάζουν και να τις τροποποιούν. Ο ορισμός τους ως *shared*, ωστόσο, δεν εξασφαλίζει την ατομικότητα κατά την ανάγνωση ή την εγγραφή τους.

- **Private (λίστα μεταβλητών)**: Οι μεταβλητές που βρίσκονται στη λίστα μεταβλητών θα είναι ιδιωτικές, δηλαδή κάθε νήμα θα έχει το δικό του αντίγραφο, το οποίο θα μπορεί να βλέπει και να επεξεργάζεται ανεξάρτητα των άλλων νημάτων. Οι μεταβλητές αυτές, όμως, δεν θα είναι αρχικοποιημένες.

- **Firstprivate (λίστα μεταβλητών)**: Οι μεταβλητές που βρίσκονται στη λίστα μεταβλητών θα είναι ιδιωτικές μεταξύ των νημάτων, όπως και στην περίπτωση των *private* μεταβλητών. Η διαφορά τους είναι ότι θα αρχικοποιηθούν με την τιμή που είχε το αρχικό νήμα (νήμα master) κατά την δημιουργία της παράλληλης περιοχής.

- **Lastprivate (λίστα μεταβλητών)**: Οι μεταβλητές που βρίσκονται στη λίστα μεταβλητών θα είναι ιδιωτικές μεταξύ των νημάτων, όπως και στην περίπτωση των *private* μεταβλητών. Η διαφορά τους είναι ότι τα ιδιωτικά αντίγραφα θα αντιγράψουν τις τιμές που έχουν από την τελευταία επανάληψη ή τμήμα της παράλληλης περιοχής που εκτέλεσαν στις αντίστοιχες καθολικές μεταβλητές. Η φράση μπορεί να χρησιμοποιηθεί μόνο σε οδηγίες *for* και *sections*.

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

2.3.5 Οδηγίες συγχρονισμού

Οι οδηγίες συγχρονισμού που παρέχει το OpenMP χρησιμοποιούνται για τον συγχρονισμό των νημάτων κατά την εκτέλεση μιας παράλληλης περιοχής, καθώς και την εξασφάλιση της ατομικότητας των ενεργειών ανάγνωσης και εγγραφής κοινόχρηστων μεταβλητών. Οι πιο χρήσιμες οδηγίες συγχρονισμού είναι οι ακόλουθες:

- **Critical**: Χρησιμοποιείται για να ορίσει μια *κρίσιμη περιοχή*, δηλαδή ένα τμήμα κώδικα το οποίο μπορεί να εκτελείται μόνο από ένα νήμα κάθε φορά. Τα υπόλοιπα νήματα υποχρεωτικά περιμένουν στην αρχή της, έως ότου τελειώσει το νήμα που έχει εξασφαλίσει την αποκλειστική πρόσβαση σε αυτήν. Συνήθως αυτό επιτυγχάνεται με τη χρήση μιας κλειδαριάς αμοιβαίου αποκλεισμού (mutual exclusion lock). Χρησιμοποιείται για να εξασφαλίσει την ακεραιότητα των τιμών των κοινόχρηστων μεταβλητών και είναι απαραίτητη όταν μια κοινόχρηστη μεταβλητή πρέπει να τροποποιηθεί από όλα τα νήματα που συμμετέχουν στην παράλληλη περιοχή.

- **Atomic**: Μπορεί να χρησιμοποιηθεί αντί της οδηγίας *critical* για να εξασφαλίσει την ατομικότητα στην περίπτωση όπου μια κοινόχρηστη μεταβλητή πρέπει να τροποποιηθεί από πολλαπλά νήματα εντός μιας παράλληλης περιοχής. Αντί να βασίζεται σε μια κλειδαριά, βασίζεται σε ειδικές εντολές μηχανής που υποστηρίζουν οι σύγχρονοι επεξεργαστές, για αυτό και οι πράξεις που μπορεί να γίνουν σε μία μεταβλητή είναι απλές και περιορισμένες, για παράδειγμα πρόσθεση και αφαίρεση. Είναι οδηγία ειδικού σκοπού, σε αντίθεση με την *critical* που μας επιτρέπει να κάνουμε ό,τι υπολογισμούς θέλουμε.

- **Barrier**: Χρησιμοποιείται για να πετύχουμε συγχρονισμό των νημάτων που εκτελούν μια παράλληλη περιοχή. Όλα τα νήματα θα πρέπει πρώτα να φτάσουν στον barrier, όσα νήματα φτάνουν περιμένουν έως ότου φτάσει και το τελευταίο. Σε εκείνο το σημείο συνεχίζουν όλα μαζί να εκτελούν τον κώδικα που βρίσκεται παρακάτω. Ένας περιορισμός του OpenMP είναι ότι ο barrier πρέπει υποχρεωτικά να εφαρμοστεί σε όλα τα νήματα που συμμετέχουν στην παράλληλη περιοχή, δηλαδή δεν μπορούμε να τον περιορίσουμε σε ένα υποσύνολο νημάτων.

- **Flush**: Χρησιμοποιείται για να πετύχουμε συγχρονισμό μνήμης. Επιβάλει την άμεση εγγραφή των κοινόχρηστων μεταβλητών, με αποτέλεσμα το νήμα που

την καλεί να αποκτήσει ξανά συνεπή εικόνα της κοινόχρηστης μνήμης. Το πρότυπο του OpenMP έχει ένα από τα χαλαρότερα μοντέλα συνέπειας μνήμης και για αυτό θα πρέπει να χρησιμοποιείται η οδηγία *flush* στις περιπτώσεις που μας ενδιαφέρει οι αλλαγές στις μεταβλητές να είναι άμεσα ορατές στα υπόλοιπα νήματα.

- **Ordered**: Χρησιμοποιείται για ορίσουμε τμήματα κώδικα που, ενώ βρίσκονται εντός ενός παράλληλου βρόχου *for*, επιθυμούμε να εκτελεστούν ακολουθιακά, δηλαδή όπως θα εκτελούνταν αν τρέχαμε το πρόγραμμα εντελώς σειριακά. Είναι χρήσιμη σε περιπτώσεις όπου μπορούμε να παραλληλοποιήσουμε το μεγαλύτερο μέρος ενός βρόχου for, για παράδειγμα, αλλά ένα μικρό τμήμα του θα πρέπει να εκτελεστεί ακολουθιακά.

2.3.6 Η οδηγία task

Η οδηγία `#pragma omp task` ορίζει μια εργασία ευέλικτου τύπου. Όταν ένα νήμα τη συναντήσει, "πακετάρει" τον κώδικα που την ακολουθεί και τις μεταβλητές που ο προγραμματιστής δηλώνει ότι χρειάζεται σε μία νέα εργασία. Η εκτέλεσή της μπορεί να ξεκινήσει αμέσως ή να καθυστερήσει 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 εκτέλεσή του 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 υπολογισμών που πρέπει να μεταφερθεί σε μεταβλητές που βρίσκονται εκτός του task. Ο προγραμματιστής μπορεί να επισημάνει τέτοιες μεταβλητές χρησιμοποιώντας τις φράσεις `shared`, `private`, `firstprivate` και `lastprivate`, όπως ακριβώς και με τις μεταβλητές των παράλληλων περιοχών, που εξετάσαμε πιο πάνω.

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).

Για να γίνουν τα παραπάνω πιο κατανοητά, ας μελετήσουμε το ακόλουθο παράδειγμα. Η συνάρτηση `fib` στο Πρόγραμμα 2.2 υπολογίζει έναν αριθμό Fibonacci αναδρομικά, χρησιμοποιώντας tasks του OpenMP. Κάθε αριθμός στην ακολουθία Fibonacci προκύπτει αθροίζοντας τους δύο προηγούμενούς του, ξεκινώντας από το 0 και το 1.

Αρχικά, στη γραμμή 8 ελέγχουμε για το τέλος της αναδρομής. Στη γραμμή 10 με την οδηγία *task* δημιουργούμε ένα νέο task, το οποίο αποτελείται από την αναδρομική κλήση και την εκχώρηση της γραμμής 11. Με τις φράσεις οδηγίας *shared* και *firstprivate* ορίζουμε ότι οι μεταβλητές `x` και `n` θα είναι κοινόχρηστη και ιδιωτική, αντίστοιχα, μέσα στο task. Με τον ίδιο ακριβώς τρόπο δημιουργούμε ένα ακόμη task στη γραμμή 12. Αυτά τα δύο tasks φροντίζουν να υπολογίσουν τους δύο προηγούμενους όρους της ακολουθίας, τους οποίους πρέπει απλώς να προσθέσουμε προκειμένου να υπολογίσουμε τον τρέχων όρο. Πριν, όμως, να μπορέσουμε να το κάνουμε αυτό, θα πρέπει να βεβαιωθούμε ότι τα tasks έχουν τελειώσει την εκτέλεσή τους, μιας και το OpenMP δεν μας εξασφαλίζει ότι ένα task θα ξεκινήσει να εκτελείται αμέσως τη στιγμή που δημιουργείται. Για να το εξασφαλίσουμε αυτό, χρησιμοποιούμε την οδηγία *taskwait* στη γραμμή 15. Τέλος, πρέπει να τονίσουμε ότι για να μπορέσουν τα tasks να εκτελεστούν από ένα νήμα εκτός του δημιουργού τους, πρέπει όλη η διαδικασία να γίνεται μέσα σε μια παράλληλη περιοχή, πράγμα που επιτυγχάνεται στη γραμμή 21. Επειδή όμως θέλουμε μόνο ένα νήμα να τα δημιουργεί, χρησιμοποιούμε την οδηγία `#pragma omp single` στη γραμμή 22.

Τέλος, αξίζει να επισημάνουμε ότι το σύστημα υποστήριξης χρόνου εκτέλεσης (runtime) θα πρέπει να είναι ικανό να υποστηρίζει αποδοτικά πολύ μεγάλο αριθμό από tasks, μιας και ο διαμερισμός ενός προβλήματος σε πολλά μικρότερα και ανεξάρτητα tasks είναι συνηθισμένη τακτική στον παράλληλο προγραμματισμό. Για παράδειγμα, ο υπολογισμός του `fib(30)` στο πρόγραμμα που εξετάζουμε θα έχει ως αποτέλεσμα την δημιουργία 2.692.536 tasks.

Πρόγραμμα 2.2: Υπολογισμός αριθμών Fibonacci αναδρομικά χρησιμοποιώντας tasks.

```
1 #include <stdio.h>
2 #include <omp.h>
3
```

```
4 int fib(int n)
5 {
6   int x, y;
7
8   if (n < 2) return n;
9
10  #pragma omp task shared(x) firstprivate(n)
11  x = fib(n - 1);
12  #pragma omp task shared(y) firstprivate(n)
13  y = fib(n - 2);
14
15  #pragma omp taskwait
16  return (x + y);
17 }
18
19 int main(void)
20 {
21  #pragma omp parallel
22  #pragma omp single
23    printf("The fibonacci of 10 is %d\n", fib(10));
24  return 0;
25 }
```

2.4 Συναρτήσεις βιβλιοθήκης χρόνου εκτέλεσης και μεταβλητές περιβάλλοντος

Το OpenMP παρέχει στον προγραμματιστή μια 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.

Οι πιο συχνές συναρτήσεις βιβλιοθήκης για χρήση από προγράμματα C/C++ είναι:

- `void omp_set_num_threads(int)`: Η ακέραια παράμετρος που δέχεται ορίζει το πλήθος των νημάτων που θα συμμετέχουν στην επόμενη παράλληλη περιοχή.
- `int omp_get_num_threads(void)`: Επιστρέφει το πλήθος των νημάτων που εκτε-

λούν την παράλληλη περιοχή στην οποία βρισκόμαστε. Αν κληθεί εκτός παράλληλης περιοχής, θα επιστρέψει 1, αφού το πρόγραμμα εκτελείται σειριακά από ένα νήμα.

- `int omp_get_thread_num(void)`: Επιστρέφει το αναγνωριστικό (ακέραιος αριθμός) του νήματος από το οποίο κλήθηκε. Το αρχικό νήμα που εκτελεί το σειριακό τμήμα του κώδικα έχει πάντα το αναγνωριστικό 0, ενώ τα αναγνωριστικά των υπολοίπων παίρνουν τιμή από το 1 έως το συνολικό πλήθος των νημάτων μείον 1.

- `int omp_in_parallel(void)`: Επιστρέφει 1 αν το κληθεί εντός παράλληλης περιοχής και 0 σε διαφορετική περίπτωση.

- `int omp_num_procs(void)`: Επιστρέφει το πλήθος των επεξεργαστών που υπάρχουν στο σύστημα.

- `void omp_init_lock(omp_lock_t*)`: Αρχικοποιεί την κλειδαριά αμοιβαίου αποκλεισμού (mutual exclusion lock) στην οποία δείχνει το όρισμά της. Η αρχικοποίηση θα πρέπει να γίνει από ένα μόνο νήμα.

- `void omp_set_lock(omp_lock_t*)`: Αν η κλειδαριά στην οποία δείχνει το όρισμά της είναι ελεύθερη, τότε το νήμα που την καλεί την κλειδώνει. Αν, όμως, το νήμα που την καλεί βρει την κλειδαριά κλειδωμένη από ένα άλλο νήμα, τότε μπλοκάρει την εκτέλεσή του περιμένοντας μέχρι η κλειδαριά να ελευθερωθεί, οπότε και την κλειδώνει.

- `int omp_test_lock(omp_lock_t*)`: Αν η κλειδαριά στην οποία δείχνει το όρισμά της είναι ελεύθερη, τότε το νήμα που την καλεί την κλειδώνει. Αν, όμως, το νήμα που την καλεί βρει την κλειδαριά κλειδωμένη από ένα άλλο νήμα, τότε επιστρέφει αμέσως, χωρίς να μπλοκάρει την εκτέλεσή του.

- `void omp_unset_lock(omp_lock_t*)`: Ξεκλειδώνει την κλειδαριά στην οποία δείχνει το όρισμά της, ώστε να μπορέσει να κλειδωθεί ξανά από κάποιο άλλο νήμα.

Οι πιο συχνές μεταβλητές περιβάλλοντος που χρησιμοποιούνται από προγράμματα OpenMP είναι οι:

- `OMP_NUM_THREADS`: Ορίζει τον προεπιλεγμένο αριθμό νημάτων που θα συμμετέχουν στις παράλληλες περιοχές. Η τιμή αυτή αγνοείται αν ο αριθμός των νημάτων ορίζεται ρητά κατά την δημιουργία της παράλληλης περιοχής, με την φράση οδηγίας `num_threads`. Επιπλέον, η τιμή αυτή μπορεί να αλλάξει αν το πρόγραμμα καλέσει την συνάρτηση `omp_set_num_threads`.

- `OMP_DYNAMIC`: Αν οριστεί με την τιμή 1, τότε το σύστημα χρόνου εκτέλεσης αποφασίζει για το μέγιστο πλήθος νημάτων που μπορούν να δημιουργηθούν για την εκτέλεση μιας παράλληλης περιοχής, ανάλογα με τα τεχνικά χαρακτηριστικά του εκάστοτε υπολογιστικού συστήματος. Δηλαδή, υπάρχει η περίπτωση να δημιουργηθούν λιγότερα νήματα από όσα ζητάει το πρόγραμμα. Αν, όμως, οριστεί με την τιμή 0, τότε το σύστημα χρόνου εκτέλεσης θα πρέπει να δημιουργεί ακριβώς το πλήθος των νημάτων που ζητούνται σε κάθε παράλληλη περιοχή.

- `OMP_SCHEDULE`: Ορίζει τον προεπιλεγμένο τρόπο διαμοιρασμού των επαναλήψεων ενός παραλληλοποιημένου βρόχου for στα νήματα που εκτελούν την παράλληλη περιοχή.

2.5 Εκτέλεση προγραμμάτων OpenMP σε clusters

Σε αυτήν την ενότητα αναφέρουμε με συντομία προηγούμενη δουλειά που είχε γίνει για την εκτέλεση προγραμμάτων που είναι γραμμένα σε OpenMP (άρα σχεδιασμένα να τρέχουν σε συστήματα κοινόχρηστης μνήμης) σε συστάδες υπολογιστών (clusters), που είναι κατεξοχήν συστήματα κατανεμημένης μνήμης. Δεν θα μιλήσουμε, ωστόσο, για την υπάρχουσα υποδομή που υπάρχει στον μεταφραστή OMPi, μιας και εξετάζουμε αναλυτικά αυτό το θέμα στην Ενότητα 3.3.

Το πρότυπο OpenMP έχει σχεδιαστεί για συστήματα κοινόχρηστης μνήμης, επομένως, για να εκτελεστεί ένα πρόγραμμα OpenMP θα πρέπει να υπάρχει κοινόχρηστη μνήμη όπου θα αποθηκεύονται οι κοινές μεταβλητές, κάτι το οποίο δεν ισχύει σε ένα cluster.

Από τα μέσα της δεκαετίας του 1990, υπήρξε μεγάλη δραστηριότητα στο σχεδιασμό και υλοποίηση λογισμικού κατανεμημένης κοινόχρηστης μνήμης (software Distributed Shared Memory - sDSM). Τα συστήματα αυτά δίνουν την ψευδαίσθηση

της ύπαρξης κοινόχρηστης μνήμης σε περιπτώσεις που αυτή δεν υπάρχει, όπως είναι τα clusters. Ο τρόπος λειτουργίας τους είναι συνοπτικά και απλουστευμένα ο εξής: η μνήμη χωρίζεται σε σελίδες και κάθε κόμβος είναι υπεύθυνος για ένα υποσύνολο αυτών. Κάθε φορά που ένας κόμβος χρειάζεται να τροποποιήσει μια σελίδα που δεν του ανήκει, ζητάει ένα αντίγραφό της. Ένα πρωτόκολλο συνέπειας φροντίζει για τον συγχρονισμό των αντιγράφων.

Με αυτά τα συστήματα μπορούσε κάποιος να γράψει ένα πρόγραμμα που αποτελούνταν από διεργασίες σε διαφορετικούς κόμβους οι οποίες, όμως, χρησιμοποιούσαν, μέσω συγκεκριμένων κλήσεων, "κοινόχρηστες" μεταβλητές. Από πίσω φυσικά, υπήρχε κίνηση μηνυμάτων στο δίκτυο, είτε χρησιμοποιώντας το MPI, είτε τις υποδοχές (sockets) που παρέχει το λειτουργικό σύστημα, είτε κάποια άλλη βιβλιοθήκη. Ένα από τα πρώτα και πιο δημοφιλή συστήματα sDSM είναι το TreadMarks [7], που αποτέλεσε την αφετηρία για πολλές από τις επόμενες προσπάθειες. Αναλύουμε σε μεγαλύτερο βάθος τις βιβλιοθήκες sDSM στο Κεφάλαιο 4.

Τέτοιες βιβλιοθήκες ήταν που χρησιμοποιήθηκαν προκειμένου να γίνει εφικτή η εκτέλεση προγραμμάτων OpenMP σε clusters. Οι συγγραφείς της εργασίας [8] υλοποίησαν έναν μεταφραστή που χρησιμοποιούσε την TreadMarks για να υποστηρίξει ένα υποσύνολο των εντολών του OpenMP σε clusters, ενώ στην εργασία [9] χρησιμοποιήθηκε μια τροποποιημένη έκδοση της TreadMarks, με σκοπό την καλύτερη επίδοση σε clusters που αποτελούνταν από πολυπύρηνους (SMPs) κόμβους.

Σε εμπορικό πλαίσιο, η εταιρία Intel είχε κυκλοφορήσει το πρόγραμμα Cluster OpenMP (επίσης γνωστό και ως CLOMP) [10], που ήταν η μόνη εμπορική λύση για εκτέλεση προγραμμάτων OpenMP σε clusters, μάλιστα χρησιμοποιούσε και αυτό μια τροποποιημένη έκδοση της TreadMarks. Αυτό το πρόγραμμα έχει πάψει να κυκλοφορεί στην αγορά για άγνωστο λόγο εδώ και αρκετά χρόνια. Από την άλλη πλευρά, στο ερευνητικό περιβάλλον έχουν προταθεί αρκετές δουλειές για την ενσωμάτωση άλλων βιβλιοθηκών sDSM σε ερευνητικούς μεταφραστές. Πιο συγκεκριμένα, στις εργασίες [11] και [12] περιγράφονται υλοποιήσεις που αφορούν την ενσωμάτωση βιβλιοθηκών sDSM στον μεταφραστή NANOS.

Ιδιαίτερη αναφορά πρέπει να κάνουμε στο σύστημα ParADE (Parallel Application Development Environment) [13], που αποτελείται από έναν μεταφραστή OpenMP που συνεργάζεται με ένα σύστημα χρόνου εκτέλεσης (runtime system). Υλοποιεί μια βιβλιοθήκη sDSM με χαλαρό πρωτόκολλο συνέπειας μνήμης πάνω στην οποία τρέχει ένα περιβάλλον OpenMP. Μάλιστα, για να ενισχύσει τις επιδόσεις του,

χρησιμοποιεί μια υβριδική προσέγγιση: το σύστημα χρόνου εκτέλεσης μεταφέρει τα δεδομένα του από τον έναν κόμβο στον άλλο χρησιμοποιώντας ρητή μεταβίβαση μηνυμάτων με το MPI, αντί να βασίζεται στην μνήμη sDSM.

Όμως, όλες οι παραπάνω δουλειές υποστηρίζουν μέχρι και την έκδοση 2.5 του OpenMP, που επικεντρώνεται στην παραλληλοποίηση απλών βρόχων for. Δεν υποστηρίζουν, δηλαδή, τα tasks, που αποτελούν ιδιαίτερα σημαντικό κομμάτι του σύγχρονου OpenMP και εμφανίστηκαν για πρώτη φορά στην έκδοση 3.0 του προτύπου.

Κεφαλαιο 3

Ο OMPi και η υπαρχουσα υποδομη για clusters

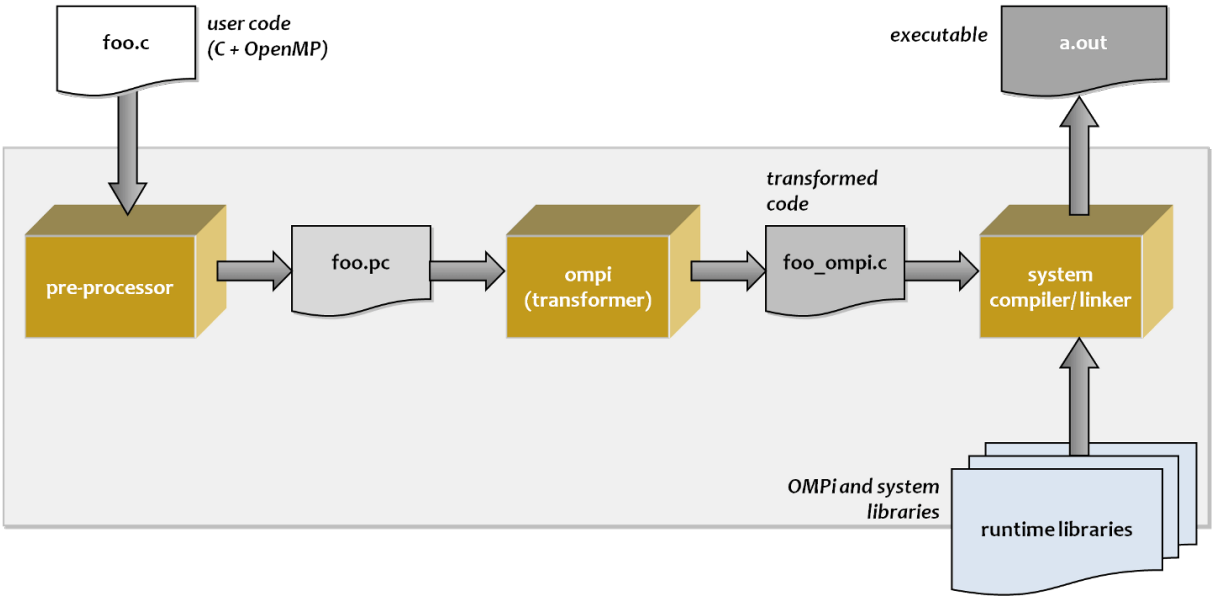
3.1 Επισκόπηση του OMPi

Ο OMPi [14] είναι ένας παραλληλοποιητικός μεταφραστής ανοιχτού κώδικα για προγράμματα που έχουν γραφτεί σε γλώσσα C και υποστηρίζει το πρότυπο OpenMP. Αποτελείται από δύο τμήματα:

- Τον μεταφραστή (compiler) που δέχεται ως είσοδο προγράμματα γραμμένα σε γλώσσα προγραμματισμού C με οδηγίες OpenMP. Ο συντακτικός αναλυτής διατρέχει το πρόγραμμα και παράγει το συντακτικό δέντρο (AST – abstract syntax tree), το οποίο αναπαριστά το αρχικό πρόγραμμα. Στη συνέχεια, προσπελάζει τους κόμβους του δέντρου και τροποποιεί αυτούς που περιέχουν οδηγίες OpenMP: μερικοί μετασχηματισμοί είναι σχετικά απλοί, ενώ άλλοι (όπως για παράδειγμα ο *parallel*) απαιτούν σημαντικές αλλαγές. Το αποτέλεσμα αυτής της διαδικασίας είναι η παραγωγή ενός νέου κώδικα σε C, ο οποίος είναι ισοδύναμος με τον αρχικό, με την διαφορά ότι οι οδηγίες OpenMP έχουν

αντικατασταθεί με μετασχηματισμένο κώδικα που εμπεριέχει 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.

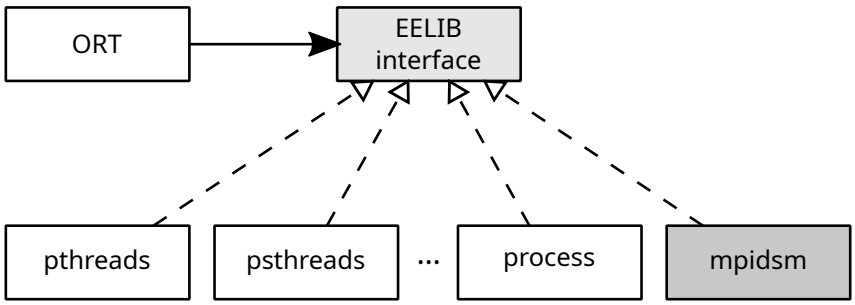
- Τη βιβλιοθήκη χρόνου εκτέλεσης που παρέχει συναρτήσεις για την δημιουργία υπολογιστικών οντοτήτων (όπως νήματα και διεργασίες) που αναλαμβάνουν την εκτέλεση του κώδικα που έχει παραλληλοποιηθεί, συναρτήσεις που διαχειρίζονται κλειδαριές και άλλες βοηθητικές δυνατότητες, καθώς και συναρτήσεις που διαχειρίζονται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



Σχήμα 3.1: Η διαδικασία μετάφρασης του προγράμματος του χρήστη `foo.c` με τον OMPi.

Η διαδικασία μετάφρασης του κώδικα του χρήστη `foo.c`, που είναι γραμμένος σε C και περιέχει οδηγίες OpenMP με τον OMPi φαίνεται γραφικά στο Σχήμα 3.1. Όπως συμβαίνει με κάθε κώδικα C, το πρώτο βήμα είναι να τον περάσουμε από τον προ-επεξεργαστή (pre-processor). Δουλειά του είναι να αφαιρέσει τα σχόλια του χρήστη, να αντικαταστήσει τις σταθερές (`#define`) με τις τιμές τους και να εισάγει τον κώδικα των αρχείων που προσδιορίζονται με οδηγίες `#include`. Στη συνέχεια, καλείται ο OMPi που αναλύει συντακτικά και μετασχηματίζει τον κώδικα, αντικαθιστώντας τις οδηγίες του OpenMP με κλήσεις στις συναρτήσεις της βιβλιοθήκης

χρόνου εκτέλεσης. Σε αυτό το σημείο, ο μεταφραστής C που βρίσκεται στο σύστημα, όπως για παράδειγμα ο GCC, είναι σε θέση να παράξει το τελικό εκτελέσιμο αρχείο (`a.out`).



Σχήμα 3.2: Οργάνωση της βιβλιοθήκης χρόνου εκτέλεσης στον OMPi.

Ένας από τους βασικούς στόχους που ακολουθείται κατά την ανάπτυξη του OMPi είναι η επεκτασιμότητα, δηλαδή η δυνατότητα να εισάγεται καινούργια λειτουργικότητα, χωρίς να απαιτούνται τροποποιήσεις στον υπάρχων κώδικα. Ως εκ τούτου, η βιβλιοθήκη χρόνου εκτέλεσης χωρίζεται σε δύο τμήματα, όπως φαίνεται και στο Σχήμα 3.2. Από τη μια πλευρά, το ORT (OMPi runtime) είναι υπεύθυνο για την υλοποίηση της λειτουργικότητας που απαιτεί το OpenMP και δουλεύει σε πιο "υψηλό" επίπεδο. Από την άλλη πλευρά, έχουμε τις βιβλιοθήκες οντοτήτων εκτέλεσης (execution entity libraries – EELIBs), που δουλεύουν σε πιο "χαμηλό" επίπεδο και είναι υπεύθυνες για την δημιουργία και καταστροφή οντοτήτων εκτέλεσης (όπως είναι τα νήματα), την παροχή κλειδαριών, barriers και άλλων στοιχείων που απαιτούνται στον παράλληλο προγραμματισμό. Αυτή που χρησιμοποιείται πιο συχνά είναι η *pthreads*, που δημιουργεί και χειρίζεται νήματα Posix, ενώ υπάρχει η *process* που χειρίζεται διεργασίες, η *psthreads* που χειρίζεται νήματα επιπέδου χρήστη κ.α. Αυτές παρέχουν την λειτουργικότητα που χρειάζεται το ORT μέσω μιας διεπαφής, εξού και η ευκολία στην επεκτασιμότητα. Η βιβλιοθήκη που δημιουργήσαμε εμείς ονομάζεται *mpidsm* και χειρίζεται διεργασίες σε διαφορετικούς κόμβους ενός cluster που καθεμιά αποτελείται από επιμέρους νήματα.

3.2 Παραδείγματα μετασχηματισμού κώδικα

Για να γίνει η παραπάνω διαδικασία πιο κατανοητή, σε αυτή την ενότητα θα εξετάσουμε σε βάθος τον κώδικα που παράγεται από τη μετάφραση των δύο απλών

προγραμμάτων που είδαμε στην Ενότητα 2.3 με τον OMPi. Το ένα πρόγραμμα δημιουργεί μια παράλληλη περιοχή, ενώ το άλλο δημιουργεί tasks. Όπως είδαμε στην προηγούμενη παράγραφο, ο OMPi σε "υψηλό" επίπεδο βλέπει οντότητες εκτέλεσης και παρέχει υποστήριξη τόσο για νήματα, όσο και για διεργασίες. Στη δική μας περίπτωση, έχουμε διεργασίες που τρέχουν σε διαφορετικούς κόμβους, επομένως ο μετασχηματισμένος κώδικας που θα δείξουμε παρακάτω είναι ο κώδικας που παράγεται όταν η οντότητα εκτέλεσης χειρίζεται διεργασίες. Ο κώδικας θα ήταν λίγο διαφορετικός αν η οντότητα εκτέλεσης χειριζόταν μόνο νήματα σε μία διεργασία.

3.2.1 Μετασχηματισμός παράλληλης περιοχής

Σε αυτό το σημείο θα εξετάσουμε τους μετασχηματισμούς που λαμβάνουν χώρα κατά τη μετάφραση του Προγράμματος 2.1, που βρίσκεται στην σελίδα 15. Στόχος του είναι να υπολογίσει παράλληλα το μέγιστο στοιχείο ενός πίνακα. Στο Πρόγραμμα 3.1 φαίνεται ο μετασχηματισμένος κώδικας που προκύπτει όταν το μεταφράζουμε με τον OMPi.

Πρόγραμμα 3.1: Μετασχηματισμένο πρόγραμμα εύρεσης μέγιστου στοιχείου πίνακα.

```
1 int __original_main(int _argc_ignored, char ** _argv_ignored)
2 {
3   int array [1024];
4   int i, max_value;
5
6   struct __shvt__ {
7     int (* array)[ 1024];
8     int (* max_value);
9   } _shvars;
10
11  _shvars.array = &array;
12  _shvars.max_value = &max_value;
13  ort_execute_parallel(_thrFunc0_, (void *) &_shvars, -1, 1, 0);
14  printf("The max_value is %d\n", max_value);
15  return (0);
16 }
17
18 static void * _thrFunc0_(void * __arg)
19 {
20  struct __shvt__ {
```

```
21      int (* array)[ 1024];
22      int (* max_value);
23    };
24    struct __shvt__ * _shvars = (struct __shvt__ *) __arg;
25
26    int (* array) [1024] = _shvars->array;
27    int max_value = INT_MIN;
28
29    int i;
30    struct _ort_gdopt_ gdopt_;
31    unsigned long niters_ = 0, iter_ = 0, fiter_, liter_ = 0;
32    niters_ = (long) (((1024) >= (0)) ? ((1024) - (0)) : 0);
33    ort_entering_for(0, 0, &gdopt_);
34    if (ort_get_static_default_chunk(niters_, &fiter_, &liter_))
35    {
36      for (iter_ = fiter_, i = (0) + fiter_ * 1; iter_ < liter_; iter_++, i +=
            1)
37        max_value = ((*array)[i] > max_value) ? (*array)[i] : max_value;
38    }
39    ort_fence();
40    ort_reduction_begin(&(*_paredlock0));
41    if (*(_shvars->max_value) < max_value)
42      *(_shvars->max_value) = max_value;
43    ort_reduction_end(&(*_paredlock0));
44    ort_taskwait(2);
45    return ((void *) 0);
46  }
```

Αρχικά, παρατηρούμε ότι η συνάρτηση `main` έχει μετονομαστεί σε `__original_main` και ο κώδικας που αποτελούσε την παράλληλη περιοχή έχει εξαχθεί σε μια καινούργια συνάρτηση, την `_thrFunc0_`. Στο αρχικό πρόγραμμα, ο πίνακας `array` ήταν κοινόχρηστος στην παράλληλη περιοχή. Επίσης, η μεταβλητή `max_value` θα πρέπει να είναι κοινόχρηστη ώστε η κάθε διεργασία να μπορεί να διαπιστώσει αν το τοπικό μέγιστο που υπολόγισε είναι μεγαλύτερο και από το ολικό μέγιστο που έχει υπολογιστεί από τις άλλες διεργασίες ως εκείνη τη στιγμή. Για κάθε μεταβλητή που πρέπει να είναι κοινόχρηστη, τοποθετούμε έναν δείκτη στην δομή `_shvars`, τόσο στη συνάρτηση που καλεί την παράλληλη περιοχή (εδώ η `main`), όσο και στην συνάρτηση που υλοποιεί την παράλληλη περιοχή (εδώ η `_thrFunc0_`), όπως φαίνεται στις γραμμές 6-9 και 20-23. Πριν την κλήση (γραμμές 11-12) ενημερώνουμε την δομή ώστε

οι δείκτες να δείχνουν στις αντίστοιχες μεταβλητές της αρχικής διεργασίας. Στην παράλληλη περιοχή (γραμμή 26) ο δείκτης προς τον πίνακα ενημερώνεται ώστε να δείχνει στον πίνακα της αρχικής διεργασίας.

Για να εκτελεστεί η παράλληλη περιοχή, θα πρέπει να κληθεί η συνάρτηση `ort_execute_parallel` στην γραμμή 13. Ως ορίσματα μεταξύ άλλων δέχεται το όνομα της συνάρτησης που αντιστοιχεί στην παράλληλη περιοχή, τη δομή με τις μεταβλητές που θα χρησιμοποιηθούν στην παράλληλη περιοχή και το πλήθος των νημάτων που θα την εκτελέσουν (-1 σημαίνει ότι το σύστημα θα πρέπει να αποφασίσει για το βέλτιστο αριθμό αυτόματα).

Ο βρόχος for έχει μετασχηματιστεί προκειμένου να μοιράζει τις επαναλήψεις στο κάθε νήμα (γραμμές 30-43). Οι επαναλήψεις μοιράζονται στατικά μεταξύ των διεργασιών, επομένως δεν απαιτείται κανένας συγχρονισμός μεταξύ τους (γραμμές 33-38). Το σώμα του βρόχου for φαίνεται στη γραμμή 39, με τη διαφορά ότι ο πίνακας `array` έχει μετασχηματιστεί σε δείκτη που δείχνει στον αρχικό πίνακα. Το τοπικό αντίγραφο της μεταβλητής `max_value` αρχικοποιείται στον μικρότερο δυνατό ακέραιο στην γραμμή 27. Μετά την εκτέλεση του βρόχου for κάθε διεργασία πρέπει να ελέγξει αν το τοπικό μέγιστο που υπολόγισε είναι και το ολικό μέγιστο. Αυτό συμβαίνει στις γραμμές 43-46. Επειδή πάνω από μια διεργασίες μπορεί να χρειαστεί να τροποποιήσουν το ολικό μέγιστο ταυτόχρονα, χρειαζόμαστε μια κλειδαριά αμοιβαίου αποκλεισμού (mutual exclusion lock) για να εξασφαλίσουμε την ατομικότητα. Πίσω από τις κλήσεις `ort_reduction_begin` και `ort_reduction_end` στις γραμμές 43 και 46 κρύβεται το κλείδωμα και το ξεκλείδωμα (αντίστοιχα) μιας κλειδαριάς. Το ολικό μέγιστο βρίσκεται στην αρχική διεργασία και μέσα στην παράλληλη περιοχή μπορεί να προσπελαστεί μέσω του δείκτη που έχουμε σε αυτό, όπως φαίνεται στις γραμμές 44 και 45.

3.2.2 Μετασχηματισμός tasks

Τώρα θα δούμε τους μετασχηματισμούς που λαμβάνουν χώρα κατά τη μετάφραση του Προγράμματος 2.2, που βρίσκεται στην σελίδα 19. Δουλειά του είναι να υπολογίσει την ακολουθία Fibonacci χρησιμοποιώντας μόνο αναδρομικά tasks. Στο Πρόγραμμα 3.2 φαίνεται ο μετασχηματισμένος κώδικας που προκύπτει όταν το μεταφράζουμε με τον OMPi.

Πρόγραμμα 3.2: Μετασχηματισμένο πρόγραμμα υπολογισμού της ακολουθίας Fibonacci.

```
1 int __original_main(int _argc_ignored, char ** _argv_ignored)
2 {
3   ort_execute_parallel(_thrFunc0_, (void *) 0, -1, 0, 0);
4   return (0);
5 }
6
7 static void * _thrFunc0_(void * __arg)
8 {
9   if (ort_mysingle(1))
10    printf("The fibonacci of 10 is %d\n", fib(10));
11  ort_leaving_single();
12  ort_taskwait(2);
13  return ((void *) 0);
14 }
15
16 int fib(int n)
17 {
18  int x, y;
19
20  if (n < 2) return (n);
21
22  struct __taskenv__ {
23    int (* x);
24    int n;
25  } * _tenv;
26
27  _tenv = (struct __taskenv__ *) ort_taskenv_alloc(sizeof(struct __taskenv__
        ), _taskFunc0_);
28  _tenv->x = &x;
29  _tenv->n = n;
30  ort_new_task(_taskFunc0_, (void *) _tenv, 0, 0, 0, 0, (void *) 0, 0, 0, 0)
        ;
31
32  /* Similar code for the other task */
33  ort_taskwait(0);
34  return ((x + y));
35 }
36
37 static void * _taskFunc0_(void * __arg)
38 {
39  struct __taskenv__ {
```

```
40      int (* x);
41      int n;
42    };
43    struct __taskenv__ * _tenv = (struct __taskenv__ *) __arg;
44
45    int (* x) = _tenv->x;
46    int n = _tenv->n;
47
48    (*x) = fib(n - 1);
49    ort_taskenv_free(_tenv, _taskFunc0_);
50    return ((void *) 0);
51  }
```

Όπως στο πρόγραμμα της προηγούμενης ενότητας, έτσι και εδώ παρατηρούμε ότι η συνάρτηση `main` έχει μετονομαστεί σε `__original_main`. Η μοναδική δουλειά που κάνει είναι να δημιουργεί οντότητες εκτέλεσης και να τις βάζει να εκτελέσουν την παράλληλη περιοχή, που βρίσκεται στην συνάρτηση `_thrFunc0_`. Δεν υπάρχουν κοινόχρηστες μεταβλητές στην παράλληλη περιοχή, επομένως, σε αντίθεση με την προηγούμενη ενότητα, δεν χρειάζεται η ύπαρξη δομής με δείκτες σε αυτές.

Στην παράλληλη περιοχή οι κλήσεις `ort_mysingle` και `ort_leaving_single` στις γραμμές 9 και 11, αντίστοιχα, εξασφαλίσουν ότι μία μόνο διεργασία θα εκτελέσει τον κώδικα που βρίσκεται στη γραμμή 10. Η συνάρτηση `fib` είναι αυτή που δημιουργεί τα αναδρομικά tasks. Στις γραμμές 22-30 δείχνουμε τον κώδικα που χρειάζεται για να παραχθεί το ένα task. Δεν δείχνουμε τον κώδικα που χρειάζεται για το άλλο task, αλλά είναι παρόμοιος.

Το task έχει μια κοινόχρηστη μεταβλητή (την `x`) και μια ιδιωτική (την `n`). Όπως με τις παράλληλες περιοχές, έτσι και με τα tasks, δημιουργούμε μια δομή με δείκτες (γραμμές 22-25) όπου αποθηκεύουμε ένα δείκτη που δείχνει στην αντίστοιχη μεταβλητή της διεργασίας που παράγει το task (γραμμή 28). Η μεταβλητή `n` έχει οριστεί ως *firstprivate*, που σημαίνει ότι το task θα έχει ένα ιδιωτικό αντίγραφο με την τιμή που έχει η `n` στην διεργασία που δημιουργεί το task τη στιγμή που το δημιουργεί. Για να το πετύχουμε αυτό, κατά την δημιουργία του task αποθηκεύουμε την τιμή της `n` στην δομή (γραμμή 29), ενώ κατά την εκτέλεση του task, αρχικοποιούμε το τοπικό αντίγραφο με την τιμή που διαβάζουμε από την δομή (γραμμή 46). Σε αντίθεση με τις παράλληλες περιοχές που η δομή είναι τοπική μεταβλητή της συνάρτησης που τις δημιουργεί, η μνήμη για αυτή δεσμεύεται με την συνάρτηση

`ort_taskenv_alloc` (γραμμή 27) πριν την δημιουργία του task και αποδεσμεύεται με την συνάρτηση `ort_taskenv_free` (γραμμή 49) μέσα στο task.

Το task έχει εξαχθεί στην συνάρτηση `_taskFunc0_` και ο κώδικας που αποτελούσε το σώμα του βρίσκεται στην γραμμή 48, όπου η κοινόχρηστη μεταβλητή `x` έχει αντικατασταθεί με δείκτη. Η δημιουργία του task γίνεται καλώντας την συνάρτηση `ort_new_task`, που, μεταξύ άλλων, δέχεται ως ορίσματα την συνάρτηση του task και την δομή με τους δείκτες για τις κοινόχρηστες μεταβλητές. Τέλος, η εκτέλεση των tasks που έχουν δημιουργηθεί γίνεται καλώντας την συνάρτηση `ort_taskwait` στην γραμμή 33 για την διεργασία που δημιουργεί τα tasks και στην γραμμή 12 για τις υπόλοιπες διεργασίες που συμμετέχουν στην παράλληλη περιοχή αλλά δεν δημιουργούν tasks.

Είναι καλό να σημειώσουμε εδώ ότι ο κώδικας στο Πρόγραμμα 3.2 έχει παραχθεί στοχεύοντας στο μικρότερο δυνατό μέγεθος κώδικα. Ο OMPi μπορεί να παράγει, εφόσον ζητηθεί, και εναλλακτικό κώδικα για tasks, ο οποίος υπό προϋποθέσεις εξασφαλίζει καλύτερες επιδόσεις, με το μειονέκτημα ότι αυξάνεται αρκετά το μέγεθος του παραγόμενου κώδικα.

3.3 Η υπάρχουσα υποδομή για clusters στον OMPi

Σε αυτή την ενότητα θα περιγράψουμε την υπάρχουσα υποδομή που υπήρχε στον OMPi για την εκτέλεση προγραμμάτων OpenMP σε clusters [15], πάνω στην οποία βασίστηκε και η δική μας δουλειά. Κίνητρό της ήταν η παρατήρηση ότι το MPI, που χρησιμοποιείται ευρέως και είναι ιδιαίτερα αποδοτικό για τον προγραμματισμό clusters, δεν είναι προσιτό για τον μέσο προγραμματιστή, μιας και απαιτεί ο διαμοιρασμός των δεδομένων αλλά και όλες οι επικοινωνίες να γίνονται "με το χέρι", δηλαδή με ρητές κλήσεις από τον προγραμματιστή.

Από την άλλη πλευρά, οι βιβλιοθήκες sDSM (software Distributed Shared Memory – κατανεμημένη κοινόχρηστη μνήμη υλοποιημένη σε λογισμικό) δίνουν στον προγραμματιστή την ψευδαίσθηση της ύπαρξης κοινόχρηστης μνήμης, ακόμη και αν αυτή δεν υπάρχει, όπως για παράδειγμα στο κατανεμημένο περιβάλλον που λειτουργούν τα clusters, με αποτέλεσμα την διευκόλυνση του προγραμματισμού τους. Τα μειονεκτήματά τους είναι ότι έχουν χαμηλότερη απόδοση σε σχέση με το MPI, καθεμιά παρέχει διαφορετικό API και συνήθως χρησιμοποιούν χαλαρά πρωτόκολλα

συνοχής, που σημαίνει ότι ο προγραμματιστής είναι αναγκασμένος να εισάγει ρητές κλήσεις συγχρονισμού για να εγγυηθεί την ορθότητα του προγράμματός του.

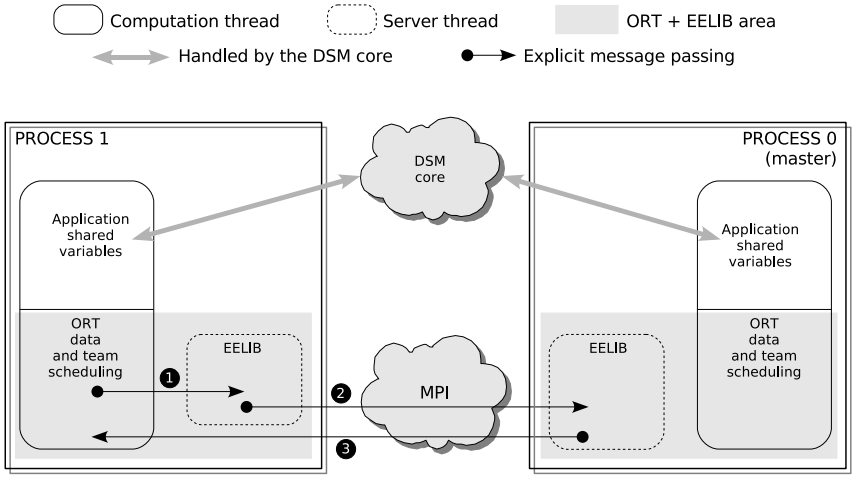
Έτσι, προέκυψε η ιδέα μιας υβριδικής προσέγγισης που συνδυάζει το MPI με βιβλιοθήκες sDSM, ως έναν αποδοτικότερο τρόπο για την αξιοποίηση ενός cluster. Μάλιστα, για πειραματικούς σκοπούς, ενσωματώθηκε και χρησιμοποιήθηκε μια πληθώρα βιβλιοθηκών sDSM που ήταν διαθέσιμες τότε. Για την υποστήριξη πολλαπλών βιβλιοθηκών δημιουργήθηκε μια διεπαφή υψηλού επιπέδου (API) με συναρτήσεις που καθεμία έπρεπε να υλοποιεί. Κατά την ενσωμάτωσή τους λήφθηκαν υπόψιν και οι ιδιαιτερότητές τους: σε αυτές που χρησιμοποιούσαν χαλαρό πρωτόκολλο συνοχής μνήμης, έπρεπε να εισαχθούν επιπρόσθετες κλήσεις συγχρονισμού σε καίρια σημεία του προγράμματος για να εξασφαλιστεί η απαιτούμενη συνοχή. Η άλλη ιδιαιτερότητα αφορούσε τη δέσμευση κοινόχρηστης μνήμης: οι βιβλιοθήκες στις οποίες η δέσμευση κοινόχρηστης μνήμης γινόταν συλλογικά, δηλαδή από όλους τους κόμβους που εκτελούν τον πρόγραμμα, χρειαζόταν επιπλέον συγχρονισμό των κόμβων, σε σχέση με αυτές που η δέσμευση κοινόχρηστης μνήμης γινόταν από έναν μόνο κόμβο. Περισσότερες λεπτομέρειες για τις λειτουργίες αυτές θα δούμε στο Κεφάλαιο 5.

Για να μπορούν όλοι οι κόμβοι (και συνεπώς οι διαφορετικές διεργασίες που τρέχουν σε αυτούς) να προσπελάσουν και να τροποποιήσουν τις κοινόχρηστες μεταβλητές που ορίζονται στο πρόγραμμα του χρήστη, αυτές θα έπρεπε να βρίσκονται στην κοινόχρηστη μνήμη sDSM. Οι καθολικές (global) μεταβλητές μετασχηματίζονταν σε δείκτες κατά την διαδικασία της μετάφρασης του προγράμματος. Κατά την εκκίνηση του προγράμματος και πριν την εκτέλεση της συνάρτησης `main` του χρήστη, δεσμεύονταν η κατάλληλη ποσότητα μνήμης στον κοινόχρηστο χώρο και οι δείκτες τροποποιούνταν ώστε να δείχνουν στην κατάλληλη θέση (offset) στον κοινόχρηστο χώρο.

Η υλοποίηση δεν υποστήριζε εμφωλευμένο (nested) παραλληλισμό, δηλαδή την δυνατότητα να δημιουργείται μια παράλληλη περιοχή μέσα σε μια άλλη. Επομένως, οι τοπικές μεταβλητές που έπρεπε να είναι κοινόχρηστες στην παράλληλη περιοχή βρίσκονταν στην στοίβα της αρχικής διεργασίας που εκτελούσε το σειριακό μέρος του κώδικα. Για να γίνει αυτό, ολόκληρη η στοίβα της αρχικής διεργασίας μεταφερόταν στον κοινόχρηστο χώρο κατά την εκκίνηση του προγράμματος. Πιο συγκεκριμένα, η αρχική διεργασία δημιουργούσε ένα νήμα επιπέδου χρήστη, η μνήμη για την στοίβα του οποίου δεσμευόταν στον κοινόχρηστο χώρο και στη συνέχεια εκτελούσε

αυτό το νήμα.

Από την άλλη πλευρά, για τις κοινόχρηστες μεταβλητές που χρειαζόταν η ίδια η υλοποίηση του ORT, ακολουθήθηκε μια διαφορετική προσέγγιση. Για λόγους απόδοσης, αντί να τις διαχειριζόταν η βιβλιοθήκη sDSM, αυτές οι μεταβλητές αντιγράφονταν στον εκάστοτε κόμβο με ρητές κλήσεις του MPI. Κατά την εκκίνηση του προγράμματος, κάθε κόμβος δημιουργούσε ένα νήμα server, του οποίου δουλειά ήταν να λαμβάνει και να στέλνει αιτήματα από και προς τους άλλους κόμβους χρησιμοποιώντας το MPI. Αυτή η υβριδική οργάνωση φαίνεται γραφικά στο Σχήμα 3.3.



Σχήμα 3.3: Η οργάνωση της υπάρχουσας υποδομής για clusters στον OMPi. Οι μεταβλητές του προγράμματος του χρήστη γίνονται κοινόχρηστες με τη βοήθεια των βιβλιοθηκών sDSM, όπως φαίνεται στο επάνω μισό. Αντίθετα, οι μεταβλητές της υλοποίησης του ORT γίνονται κοινόχρηστες μέσω ρητών κλήσεων στο MPI, όπως φαίνεται στο σκιασμένο μέρος. Το Σχήμα αυτό προέρχεται από το [15].

Στη συνέχεια, και συγκεκριμένα στην εργασία [16], η υλοποίηση επεκτάθηκε. Αντί κάθε διεργασία να αποτελείται από ένα νήμα server που επικοινωνεί με τις υπόλοιπες διεργασίες και ένα νήμα εργάτη που κάνε όλη την δουλειά, κάθε διεργασία αποτελούταν από ένα νήμα server και $n-1$ νήματα εργάτες, όπου n το πλήθος των επεξεργαστικών πυρήνων του κάθε κόμβου. Με άλλα λόγια, κάθε διεργασία υποστήριζε πια πολλαπλά νήματα εργάτες.

Μετά από αυτό το σημείο δεν προέκυψε κάποια νεώτερη βελτίωση. Έτσι, η υλοποίηση έμεινε να υποστηρίζει την έκδοση 2.5 του OpenMP, που επικεντρωνόταν στην παραλληλοποίηση απλών βρόχων for και μέχρι σήμερα δεν υποστήριζε tasks, που εμφανίστηκαν στην έκδοση 3.0 του OpenMP.

Κεφαλαιο 4

Βιβλιοθηκεσ sDSM και tasking σε clusters

4.1 Βιβλιοθήκες sDSM

Οι βιβλιοθήκες sDSM (software Distributed Shared Memory – κατανεμημένη κοινόχρηστη μνήμη υλοποιημένη σε λογισμικό) χρησιμοποιούνται για να δώσουν την ψευδαίσθηση κοινόχρηστης μνήμης σε συστήματα κατανεμημένης μνήμης, όπως οι υπολογιστικές συστάδες (clusters), με σκοπό να διευκολύνουν τον προγραμματισμό τους. Η βασική αρχή λειτουργίας τους είναι η εξής: κάθε κόμβος είναι υπεύθυνος για ένα τμήμα της συνολικής κοινόχρηστης μνήμης, που συνήθως το μέγεθός του είναι πολλαπλάσιο του μεγέθους μιας σελίδας (page). Από αυτές τις σελίδες αφαιρείται το δικαίωμα εγγραφής και ανάγνωσης, επομένως όταν προσπαθεί να τις προσπελάσει το πρόγραμμα του χρήστη προκαλείται σήμα segmentation fault. Τότε καλείται ο χειριστής σφάλματος (που έχει εγκαταστήσει η βιβλιοθήκη sDSM), ο οποίος φροντίζει να εκτελέσει το πρωτόκολλο συνοχής μνήμης και να κάνει τα

δεδομένα που ζήτησε το πρόγραμμα του χρήστη διαθέσιμα. Το πρωτόκολλο συνοχής (coherency protocol) καθορίζει *ποια* τιμή κάθε μεταβλητής θα γίνει ορατή στους υπόλοιπους κόμβους. Συνήθως, προκειμένου να αυξήσουν την αποδοτικότητά τους, οι βιβλιοθήκες sDSM χρησιμοποιούν χαλαρά πρωτόκολλα συνέπειας μνήμης (consistency protocol), τα οποία καθορίζουν *πότε* οι αλλαγμένες τιμές θα γίνουν ορατές στους υπόλοιπους κόμβους. Αυτό βέβαια επιβαρύνει τον προγραμματιστή, αναγκάζοντάς τον να κάνει ρητές κλήσεις συγχρονισμού μνήμης για εξασφαλίσει την ορθότητα του προγράμματός του.

Η έρευνα για βιβλιοθήκες sDSM ξεκίνησε στις αρχές της δεκαετίας του 90. Μία από τις πρώτες και πιο ιστορικές είναι η TreadMarks [7]. Αποτέλεσε σημείο εκκίνησης για πολλές από τις προσπάθειες που έγιναν στην συνέχεια. Η JIAJIA [17] σχεδιάστηκε με σκοπό να είναι όσο πιο απλή γίνεται, για αυτό και το πρωτόκολλο συνοχής μνήμης που χρησιμοποιεί βασίζεται σε κλειδαριές. Το μέγεθος της κοινόχρηστης μνήμης μπορεί να είναι τόσο μεγάλο, όσο και το άθροισμα των τοπικών μνημών κάθε μηχανήματος.

Η Mocha [18] βασίστηκε στην παρατήρηση ότι βιβλιοθήκες όπως η TreadMarks και η JIAJIA χρησιμοποιούν το πρωτόκολλο UDP για μεταφορά δεδομένων, το οποίο δεν παρέχει αξιόπιστη επικοινωνία και επομένως, χρειάζεται να στέλνουν επιβεβαίωση (acknowledgment) μετά από κάθε μήνυμα. Υποστηρίζει ότι η επιβεβαίωση δεν είναι απαραίτητη σε όλες τις περιπτώσεις, ενώ υλοποιεί μια μέθοδο για την μείωση της επιβάρυνσης από τις επιβεβαιώσεις.

Η Mome [19], όπως και οι περισσότερες βιβλιοθήκες sDSM δουλεύει σε επίπεδο χρήστη (user level). Το βασικό χαρακτηριστικό της είναι ότι κάθε κόμβος μπορεί να επιλέξει ανάμεσα σε δύο μοντέλα συνέπειας για τον τρόπο που βλέπει την κοινόχρηστη μνήμη: το κλασσικό μοντέλο σειριακής συνέπειας που είναι αργό και σε ένα απλό και πολύ βασικό μοντέλο χαλαρής συνέπειας, που είναι όμως πιο γρήγορο. Επίσης, η βιβλιοθήκη ParADE (Parallel Application Development Environment) [13] έχει σχεδιαστεί ειδικά για χρήση από το OpenMP. Αποτελείται από έναν μεταφραστή OpenMP και ένα σύστημα χρόνου εκτέλεσης (runtime system), το οποίο υλοποιεί μια βιβλιοθήκη sDSM.

Τα τελευταία χρόνια εμφανίστηκε μια καινούργια βιβλιοθήκη, η ArgoDSM [20], την οποία αποφασίσαμε να ενσωματώσουμε στην υλοποίησή μας. Χρησιμοποιεί ένα χαλαρό πρωτόκολλο συνοχής και πετυχαίνει υψηλές επιδόσεις, όταν το cluster στο οποίο χρησιμοποιείται διαθέτει σύγχρονο και αποδοτικό εξοπλισμό δικτύωσης.

Επιπρόσθετα, μιας και έχει δημιουργηθεί έχοντας υπόψη της σύγχρονα μηχανήματα, επιτρέπει σε πολλαπλά νήματα μέσα σε κάθε διεργασία να προσπελάζουν την μνήμη sDSM και να καλούν τις συναρτήσεις της, είναι δηλαδή μια thread safe βιβλιοθήκη.

4.2 Η βιβλιοθήκη ArgoDSM

4.2.1 Εισαγωγή

Τα περισσότερα συστήματα sDSM αντιμετωπίζουν δύο βασικά προβλήματα που περιορίζουν την κλιμάκωσή τους. Πρώτον, ο έλεγχος συνέπειας στα κοινόχρηστα δεδομένα γίνεται συνήθως κεντρικά σε ένα σημείο του συστήματος. Δεύτερον, το σειριακό κομμάτι του προγράμματος (π.χ. κρίσιμες περιοχές) εκτελείται χωρίς κάποια σειρά προτεραιότητας, με αποτέλεσμα να χρειάζεται μετακίνηση δεδομένων στον κόμβο που εκτελεί κάθε φορά τον κώδικα. Όταν ειδικά για την διασύνδεση των κόμβων χρησιμοποιείται κοινός εξοπλισμός δικτύωσης (π.χ. Ethernet), η κατάσταση γίνεται ακόμη χειρότερη: η καθυστέρηση (latency) του δικτύου είναι μεγάλη και οι χειριστές μηνυμάτων (message handlers) που εκτελούνται σε λογισμικό αυξάνουν την καθυστέρηση. Αυτό έχει ως αποτέλεσμα ο συγχρονισμός σε κρίσιμες περιοχές γίνεται σημείο συμφόρησης (bottleneck).

Ιστορικά, όταν είχαν κυκλοφορήσει τα πρώτα συστήματα sDSM, η καθυστέρηση και το εύρος ζώνης (bandwidth) του δικτύου ήταν τάξεις μεγέθους χειρότερα από την καθυστέρηση και το εύρος ζώνης της μνήμης RAM, επομένως η επιβάρυνση για την εκτέλεση των χειριστών μηνυμάτων ήταν σχετικά μικρή. Ωστόσο, χάρη στην πρόοδο στην τεχνολογία δικτύων, η απόσταση αυτή σήμερα έχει μειωθεί σημαντικά.

Με βάση αυτά, η ArgoDSM [20] έχει τρεις σχεδιαστικές αρχές: να θυσιάσει κάποιο εύρος ζώνης ώστε να πετύχει μικρότερη καθυστέρηση, να ελαχιστοποιήσει, ή αν γίνεται να αποφύγει τελείως την εκτέλεση των χειριστών μηνυμάτων και να περιορίσει την μετακίνηση δεδομένων για εξαρτώμενους υπολογισμούς, όπως είναι η εκτέλεση μιας κρίσιμης περιοχής.

Η ArgoDSM είναι, λοιπόν, μια βιβλιοθήκη sDSM που εκτελείται σε επίπεδο χρήστη (user space), χρησιμοποιεί το MPI και χωρίζει την μνήμη σε σελίδες των 4 KB. Κατά την εκκίνησή της, όλοι οι κόμβοι δεσμεύουν το ίδιο εύρος εικονικών διευθύνσεων. Κάθε εικονική σελίδα εκχωρείται σε έναν κόμβο (home-based DSM) με

ακολουθιακή σειρά. Αυτό σημαίνει ότι σε ένα σύστημα με N κόμβους ο κόμβος 0 θα είναι υπεύθυνος για τις χαμηλότερες διευθύνσεις, ενώ ο κόμβος $N-1$ θα είναι υπεύθυνος για τις υψηλότερες διευθύνσεις. Ακόμη, κάθε κόμβος κρατά αντίγραφα απομακρυσμένων σελίδων που χρησιμοποιεί στην τοπική του κρυφή μνήμη (page cache).

4.2.2 Το πρωτόκολλο συνοχής

Ένα πρωτόκολλο συνοχής (coherence) μνήμης είναι απαραίτητο για να εξασφαλίσει την ενημέρωση των αντιγράφων. Τυπικά αυτό γίνεται με ένα πρωτόκολλο καταλόγου (directory), που παρακολουθεί τους αναγνώστες και τους εγγραφείς κάθε σελίδας και στέλνει αιτήματα ακύρωσης όταν είναι απαραίτητο. Στόχος της ArgoDSM είναι να εξαλείψει την συμφόρηση που προκαλούν οι κεντρικοποιημένοι (centralized) κατάλογοι, επομένως χρησιμοποιεί μια κατανεμημένη προσέγγιση.

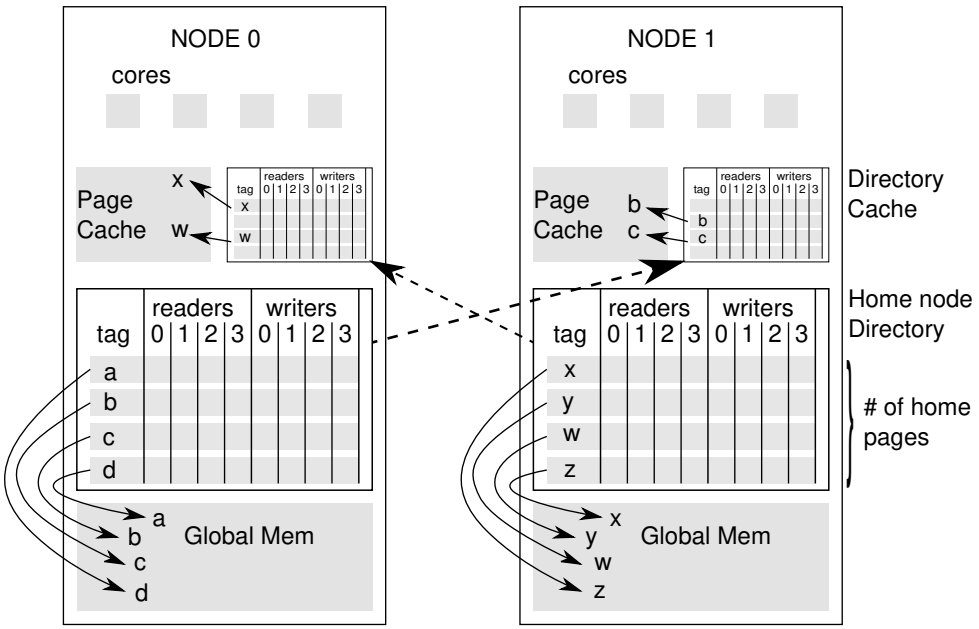
Το πρωτόκολλο συνοχής στην ArgoDSM αποτελείται από δύο μηχανισμούς. Πρώτον, την *αυτο-ακύρωση (self-invalidate)* που σημαίνει ότι κάθε κόμβος μπορεί να διαβάζει όποιο μπλοκ δεδομένων επιθυμεί, αρκεί να το ακυρώσει στο σημείο συγχρονισμού. Δεύτερον, την *αυτο-υποβάθμιση (self-downgrade)* που σημαίνει ότι κάθε κόμβος μπορεί να γράψει σε όποιο μπλοκ δεδομένων επιθυμεί, χωρίς να ζητήσει κάποια άδεια πριν, αρκεί να κάνει την εγγραφή ορατή στους άλλους κόμβους στο σημείο συγχρονισμού. Αυτή η προσέγγιση έχει δύο σημαντικά πλεονεκτήματα. Πρώτον, δεν απαιτείται ο κατάλογος να καταγράφει όλους τους κόμβους που τροποποιούν τα κοινόχρηστα δεδομένα και δεύτερον, η αυτο-υποβάθμιση εξαλείφει την ανάγκη της αναζήτησης στον κατάλογο για την εύρεση του κόμβου με την πιο πρόσφατη έκδοση των κοινόχρηστων δεδομένων στην περίπτωση αστοχίας ανάγνωσης (read miss).

Συνεπώς, το πρωτόκολλο συνοχής αποτελείται από παθητικούς καταλόγους (passive directory protocol), δηλαδή δεν χρησιμοποιεί χειριστές μηνυμάτων για την δημιουργία αιτήσεων και την απάντηση σε αυτές, αλλά αποκλειστικά κλήσεις άμεσης απομακρυσμένης προσπέλασης μνήμης (RDMA - Remote Direct Memory Access), που γίνονται από τους αιτούντες κόμβους και εκτελούν όλες τις ενέργειες του πρωτοκόλλου. Οι κατάλογοι είναι δομές μεταδεδομένων (metadata structures) που διαβάζονται και γράφονται απομακρυσμένα με χρήση RDMA από τους αιτούντες κόμβους. Για κάθε σελίδα μνήμης αποθηκεύουν τους κόμβους που διαβάζουν και

τους κόμβους που γράφουν την σελίδα, χρησιμοποιώντας πλήρη απεικόνιση (full map). Μάλιστα, η χρήση RDMA και η απουσία εκτέλεσης κώδικα στην πλευρά του παραλήπτη είναι μια από τις σημαντικότερες διαφορές ανάμεσα στην ArgoDSM και σε άλλα συστήματα sDSM.

Δύο ειδών φράχτες (fences) είναι διαθέσιμοι στην ArgoDSM. Ο φράχτης `SI_fence` για αυτο-ακύρωση, στον οποίο όλες οι σελίδες που βρίσκονται στην κρυφή μνήμη σελίδων ακυρώνονται και ο φράχτης `SD_fence` για αυτο-υποβάθμιση, στον οποίο όλες οι εγγραφές σε τροποποιημένες σελίδες (χρησιμοποιούνται *diffs* για να βρεθούν οι αλλαγές) γίνονται ορατές σε όλους τους κόμβους. Το μοντέλο μνήμης της ArgoDSM αντιστοιχεί σε μοντέλο χαλαρής συνέπειας (weak memory model) αν σε κάθε σημείο συγχρονισμού εφαρμοστούν και οι δύο φράχτες.

Η οργάνωση των καταλόγων φαίνεται γραφικά στο Σχήμα 4.1. Κάθε κόμβος διαθέτει έναν κατάλογο για τις σελίδες που έχει υπό την ευθύνη του (home directory) και έναν κατάλογο για τα προσωρινά αντίγραφα (cache) των απομακρυσμένων σελίδων που χρησιμοποιεί.



Σχήμα 4.1: Η οργάνωση των καταλόγων στην ArgoDSM όταν χρησιμοποιούνται δύο κόμβοι. Το Σχήμα προέρχεται από το [20].

4.2.3 Κατάταξη σελίδων

Αν, όμως, η αυτο-ακύρωση (και σε μικρότερο βαθμό η αυτο-υποβάθμιση) γίνονται ανεξέλεγκτα, τότε αυτό μπορεί να έχει σοβαρές επιπλοκές στην απόδοση του συστήματος. Στην αυτο-ακύρωση όλα τα δεδομένα που βρίσκονται στην κρυφή μνήμη του κόμβου ακυρώνονται, ακόμη και αν δεν έχουν τροποποιηθεί. Αντίστοιχα, στην αυτο-υποβάθμιση όλες οι εγγραφές πρέπει να μεταφερθούν στον κόμβο που είναι υπεύθυνος για την σελίδα (home node), ακόμη και αν τα δεδομένα δεν θα διαβαστούν στην συνέχεια από κάποιον άλλο κόμβο. Για αυτό τον λόγο, οι σελίδες κατατάσσονται σε κατηγορίες και η κατηγορία κάθε σελίδας αποθηκεύεται στα μεταδεδομένα του καταλόγου της.

Υπάρχουν δύο ειδών κατατάξεις και ο χρήστης της βιβλιοθήκης μπορεί να επιλέξει ποια θα χρησιμοποιηθεί. Στην πρώτη, κάθε σελίδα μπορεί να είναι ιδιωτική (private), αν μόνο ένας κόμβος την προσπελάζει ή κοινόχρηστη (shared), αν την προσπελάζουν πάνω από ένας κόμβοι. Στην δεύτερη κατάταξη, κάθε σελίδα μπορεί να είναι μόνο για ανάγνωση, μοναδικού εγγραφέα (single writer), αν ένας μόνο κόμβος την γράφει ή πολλαπλών εγγραφέων (multiple writers), αν πάνω από ένας κόμβοι την γράφουν. Στην τρέχουσα υλοποίηση οι κατηγορίες πηγαίνουν προς τη μια μόνο κατεύθυνση και δεν μπορούν να αναιρεθούν. Δηλαδή, στο πρώτο είδος κατάταξης, μια σελίδα από ιδιωτική μπορεί να γίνει δημόσια, αλλά στην συνέχεια δεν μπορεί να γίνει πάλι ιδιωτική.

Χρησιμοποιώντας αυτήν την πληροφορία, γίνεται φιλτράρισμα των σελίδων κατά την αυτο-ακύρωση, δηλαδή εξαιρούνται από αυτήν (παραμένοντας στην ιδιωτική μνήμη του κόμβου) οι ιδιωτικές σελίδες ή οι σελίδες χωρίς εγγραφείς (μόνο για ανάγνωση). Ο λόγος που στο ένα είδος κατάταξης οι κοινόχρηστες σελίδες χωρίζονται σε δύο κατηγορίες (με έναν ή πολλούς εγγραφείς) είναι ότι αυτός ο διαχωρισμός δίνει επιπρόσθετες ευκαιρίες για βελτιστοποίηση.

Αρχικά όλες οι σελίδες ανήκουν στην κατηγορία μόνο για ανάγνωση. Οποιοσδήποτε κόμβος που έχει αντίγραφο μιας σελίδας στην κρυφή μνήμη του μπορεί να το τροποποιήσει χωρίς ζητήσει άδεια από τους άλλους κόμβους[1]. Ο μοναδικός εγγραφέας στον συγχρονισμό πρέπει να κάνει αυτο-υποβάθμιση, δηλαδή να ενημερώσει τον κόμβο που είναι υπεύθυνος για την σελίδα (home node) για τις αλλαγές που

[1] Αυτό ισχύει με την προϋπόθεση ότι στο πρόγραμμα δεν υπάρχουν συνθήκες συναγωνισμού (data race free), δηλαδή δεν υπάρχει περίπτωση δύο κόμβοι να τροποποιήσουν τα ίδια δεδομένα χωρίς ρητή κλήση συγχρονισμού (φράχτη).

έκανε. Ωστόσο, δεν χρειάζεται να κάνει αυτο-ακύρωση, δηλαδή να ακυρώσει το τοπικό του αντίγραφο: αφού είναι ο μοναδικός εγγραφέας, δεν υπάρχει περίπτωση άλλοι κόμβοι να έχουν τροποποιήσει την σελίδα. Αν στην συνέχεια εμφανιστούν και άλλοι εγγραφείς, η σελίδα θα ανήκει στην κατηγορία των πολλαπλών εγγραφέων, οπότε όλοι οι κόμβοι θα πρέπει να αυτο-ακυρώνουν την σελίδα και να την αυτο-υποβαθμίζουν (αν φυσικά την τροποποιούν) σε κάθε συγχρονισμό. Για την μετάβαση, ωστόσο, θα πρέπει να ενημερωθεί μόνο ο αρχικός εγγραφέας (από τον δεύτερο), μιας και για όλους τους υπόλοιπους κόμβους οι κατηγορίες μονός εγγραφέας και πολλαπλοί εγγραφείς είναι ισοδύναμες.

Επιπλέον της κατάταξης των σελίδων, χρησιμοποιούνται δύο ακόμη τεχνικές για να βελτιώσουν την απόδοση του συστήματος. Πρώτον, η ενημέρωση των τροποποιημένων σελίδων μόνο στα σημεία συγχρονισμού μπορεί να προκαλέσει αρκετή κίνηση στο δίκτυο. Καθώς οι σελίδες τροποποιούνται, τοποθετούνται σε μια ουρά FIFO με προσαρμόσιμο μέγεθος. Όταν η ουρά γεμίσει, οι αλλαγές της σελίδας που βρίσκεται στην αρχή στης ουράς μεταφέρονται στον κόμβο που είναι υπεύθυνος για αυτήν (home node). Δεύτερον, ένας περιορισμός που προκύπτει από την χρήση του MPI παθητικά (passive one-side communication) είναι ότι μόνο ένα νήμα μπορεί να χρησιμοποιήσει το δίκτυο κάθε στιγμή. Για αυτόν το λόγο, σε κάθε αστοχία της κρυφής μνήμης η ArgoDSM δεν φέρνει μόνο την ζητούμενη σελίδα, αλλά μια "γραμμή" από διαδοχικές σελίδες.

4.2.4 Συγχρονισμός

Η επίτευξη γρήγορου συγχρονισμού είναι ζήτημα ζωτικής σημασίας για κάθε σύστημα sDSM. Στην ArgoDSM, ιδιαίτερα, ο συγχρονισμός καθορίζει τα σημεία που εκτελείται το πρωτόκολλο συνοχής με αυτο-ακυρώσεις και αυτο-υποβαθμίσεις. Προσφέρονται δύο είδη συγχρονισμού: οι barriers και οι κλειδαριές αμοιβαίου αποκλεισμού (mutual exclusion locks).

Για τους barriers χρησιμοποιείται ένα κατανεμημένο πρωτόκολλο που εξασφαλίζει ότι πρώτα κάθε κόμβος ενημερώνει για τις αλλαγές που έκανε και, στην συνέχεια, διαβάζει τα τροποποιημένα δεδομένα άλλων κόμβων. Πιο συγκεκριμένα, τα νήματα κάθε κόμβου περνούν από έναν τοπικό barrier και μετά ένα νήμα κάθε κόμβου κάνει αυτο-υποβάθμιση. Στην συνέχεια, χρησιμοποιείται ένας MPI barrier για να εξασφαλιστεί ότι όλοι οι κόμβοι έκαναν την αυτο-υποβάθμιση. Μετά ένα νήμα σε

κάθε κόμβο κάνει αυτο-ακύρωση και όλα τα νήματά του περιμένουν σε έναν ακόμη τοπικό barrier.

Ο συγχρονισμός με κλειδαριές αμοιβαίου αποκλεισμού είναι ακριβός για κάθε βιβλιοθήκη sDSM, μιας και τις περισσότερες φορές χρησιμοποιείται σε κρίσιμες περιοχές, όπου όλα τα νήματα σε όλους κόμβους χρειάζεται να τροποποιήσουν το ίδιο υποσύνολο κοινόχρηστων μεταβλητών. Αν ο έλεγχος της κλειδαριάς μεταφερθεί από έναν κόμβο σε έναν άλλο, τότε τα δεδομένα της κρίσιμης περιοχής πρέπει να μεταναστεύσουν (migrate) στον νέο κόμβο.

Το γεγονός που κάνει τα πράγματα χειρότερα είναι ότι στην ArgoDSM η κρυφή μνήμη για τις σελίδες είναι σε επίπεδο κόμβου, αντί να είναι σε επίπεδο νήματος ή πυρήνα, για παράδειγμα. Αυτή η προσέγγιση έχει πλεονεκτήματα τοπικής και χρονικής τοπικότητας σε περιόδους εκτέλεσης που δεν υπάρχει συγχρονισμός, αφού ένα νήμα μπορεί να φέρει στην κρυφή μνήμη μια σελίδα που θα χρησιμοποιήσει και ένα άλλο νήμα στον ίδιο κόμβο, οπότε το δεύτερο νήμα θα την βρει έτοιμη. Ωστόσο, αν ένα νήμα αποκτήσει ή χάσει τον έλεγχο μιας κλειδαριάς, ο συγχρονισμός θα επηρεάσει (για παράδειγμα ακυρώνοντας) όλες τις σελίδες που βρίσκονται στην κρυφή μνήμη του κόμβου, συμπεριλαμβανομένων και αυτών που χρησιμοποιούνται αποκλειστικά από τα υπόλοιπα νήματα του κόμβου.

Ως αποτέλεσμα, οι κλειδαριές στην ArgoDSM χρησιμοποιούν έναν ιεραρχικό αλγόριθμο για να περιορίσουν τις άσκοπες μετακινήσεις δεδομένων από τον έναν κόμβο στον άλλο. Προτεραιότητα για την απόκτηση μιας κλειδαριάς έχουν τα νήματα που βρίσκονται στον ίδιο κόμβο με τον προηγούμενο ιδιοκτήτη της κλειδαριάς.

4.3 Βιβλιοθήκες tasking σε clusters

Με την έννοια *task* εννοούμε ένα τμήμα κώδικα που μπορεί να εκτελεστεί ανεξάρτητα από την βασική ροή του προγράμματος. Δέχεται δεδομένα ως είσοδο, κάνει κάποιους υπολογισμούς και επιστρέφει τα αποτελέσματα στο υπόλοιπο πρόγραμμα. Σε πολλές βιβλιοθήκες ένα task αναπαρίσταται με μια συνάρτηση. Είναι ιδιαίτερα χρήσιμα γιατί διευκολύνουν το μοντέλο προγραμματισμού αφέντη–εργάτη (master–worker) που συναντάται στον παράλληλο προγραμματισμό. Σε αυτό το μοντέλο η οντότητα αφέντης δημιουργεί εργασίες (tasks), ενώ οι οντότητες εργάτες (σχεδόν πάντα είναι πάνω από ένας) φροντίζουν για την εκτέλεσή τους. Η οντότητα

αφέντης, ίσως αφού σπαταλήσει λίγο χρόνο για να κάνει δικούς της υπολογισμούς, ζητάει από τους εργάτες τα αποτελέσματα των tasks. Η παραλληλία έγκειται στο γεγονός ότι τόσο ο αφέντης, όσο και οι εργάτες μπορούν να εκτελέσουν εργασίες ταυτόχρονα. Στα συστήματα κοινόχρηστης μνήμης οι οντότητες είναι νήματα, ενώ στα συστήματα κατανεμημένης μνήμης (που μας ενδιαφέρουν εδώ) οι οντότητες είναι διεργασίες που εκτελούνται σε διαφορετικά μηχανήματα. Επίσης, μπορεί να έχουμε συνδυασμό των δύο, δηλαδή διεργασίες που εκτελούνται σε διαφορετικά μηχανήματα και καθεμιά από τις οποίες αποτελείται από επιμέρους νήματα.

Μια από τις πιο δημοφιλείς βιβλιοθήκες για δημιουργία και εκτέλεση tasks σε παράλληλο περιβάλλον είναι η KAAPI [21], που αργότερα εξελίχθηκε και μετονομάστηκε σε XKaapi [22]. Στην αρχική της έκδοση υποστήριζε συστήματα κατανεμημένης μνήμης, ωστόσο, η εξελιγμένη έκδοση περιορίζεται σε συστήματα κοινόχρηστης μνήμης, αλλά χρησιμοποιεί και τις κάρτες γραφικών (GPUs) που ενδεχομένως βρίσκονται σε αυτά. Πάντως, οι συγγραφείς της υποστηρίζουν ότι είναι στα πλάνα τους η επαναφορά της υποστήριξης για αρχιτεκτονικές κατανεμημένης μνήμης. Το κύριο πλεονέκτημα αυτής της βιβλιοθήκης είναι ο αποδοτικός αλγόριθμος που χρησιμοποιεί για κλέψιμο tasks (task stealing).

Σε άλλες αξιόλογες βιβλιοθήκες περιλαμβάνεται η Task Pool Teams [23], που υλοποιεί ένα υβριδικό προγραμματιστικό περιβάλλον που προσφέρει εξισορρόπηση φόρτου (load balancing) και πολυνηματική (multi-threaded) ασύγχρονη επικοινωνία κατά την εκτέλεση των tasks. Επίσης, η Tlib [24] είναι χτισμένη πάνω στο MPI και παρέχει ιεραρχικά δομημένα tasks για συστήματα κατανεμημένης μνήμης. Με αυτόν τον τρόπο προσπαθεί να μειώσει το κόστος της επικοινωνίας και να βελτιώσει την κλιμάκωση των λειτουργιών.

Μια άλλη κατηγορία βιβλιοθηκών χρησιμοποιούν ένα κατευθυνόμενο άκυκλο γράφημα, στο οποίο τα tasks αποτελούν τους κόμβους, ενώ οι ακμές αναπαριστούν τη σχέση πρέπει-να-εκτελεστεί-πριν. Οι πιο γνωστές βιβλιοθήκες tasking που συμπεριλαμβάνονται σε αυτή την κατηγορία είναι η Intel Threading Building Blocks [25], η OmpSs [26] και η StarPU [27].

Επιπρόσθετα, μια πιο σύγχρονη βιβλιοθήκη είναι η DuctTeip [28], που προσφέρει API σε C++ για την εκτέλεση tasks σε κατανεμημένο περιβάλλον. Χρησιμοποιεί το MPI για επικοινωνία μεταξύ των κόμβων και την βιβλιοθήκη SuperGlue [29] για την εκτέλεση tasks σε περιβάλλον κοινόχρηστης μνήμης.

Ο σημαντικότερος λόγος που τελικά καταλήξαμε να χρησιμοποιήσουμε την βι-

βλιοθήκη TORC [30] στην υλοποίησή μας είναι η απλότητα στο API της, καθώς και το γεγονός ότι υποστηρίζει απευθείας την γλώσσα C και όχι την C++, με αποτέλεσμα να υπάρχει άριστη συμβατότητα με την δική μας δουλειά. Στη συνέχεια, ωστόσο, διαπιστώσαμε ότι η σημασιολογία κάποιων λειτουργιών είναι διαφορετική στην TORC και διαφορετική στο OpenMP, με αποτέλεσμα να προβούμε σε κάποιες τροποποιήσεις του πηγαίου κώδικα της βιβλιοθήκης.

4.4 Η βιβλιοθήκη TORC

Η βιβλιοθήκη TORC [30] μας παρέχει την δυνατότητα να δημιουργούμε και να εκτελούμε tasks κατανεμημένα. Στόχος της είναι να επιτρέπει τον εύκολο προγραμματισμό clusters, που αν και είναι ευρέως διαδεδομένα, εξακολουθούν να δυσκολεύουν τους προγραμματιστές τους. Το βασικό της πλεονέκτημα έναντι άλλων βιβλιοθηκών είναι η δυνατότητά της να λειτουργεί σε clusters που παρουσιάζουν ετερογένεια, δηλαδή οι επιμέρους κόμβοι που τα αποτελούν διαφέρουν στην αρχιτεκτονική του υλικού, στην δικτύωση, στο λειτουργικό σύστημα και στην υπολογιστική ισχύ. Η TORC αντιμετωπίζει την ετερογένεια χωρίς ωστόσο να θυσιάζει την απόδοση, επομένως δεν βασίζεται σε λύσεις που χρησιμοποιούν εικονικές μηχανές, αλλά ούτε και σε γλώσσες προγραμματισμού, όπως η Java, που εκτελούνται σε έναν εικονικό επεξεργαστή με εικονικές εντολές μηχανής.

Το άλλο σημαντικό πλεονέκτημα της TORC είναι η διαφανής εξισορρόπηση φόρτου (load balancing) για εφαρμογές που χρησιμοποιούν το προγραμματιστικό μοντέλο αφέντη–εργάτη. Αυτό σημαίνει ότι στις εφαρμογές που μόνο μια διεργασία παράγει tasks, αυτά κατανέμονται και σε όλες τις υπόλοιπες που δεν εκτελούν κάποια δουλειά. Μάλιστα, αυτή η διαδικασία συμβαίνει αυτόματα, δηλαδή ο χρήστης της βιβλιοθήκης δεν χρειάζεται να μεριμνήσει για την μεταφορά των δεδομένων που χρησιμοποιούν τα tasks από και προς τον κόμβο που τελικά θα τα εκτελέσει. Επομένως, αν και ο προγραμματιστής μπορεί να καθορίσει ρητά τον κόμβο στον οποίο θα εκτελεστεί ένα task κατά την δημιουργία του, η υποστήριξη για κλέψιμο των tasks (task stealing), τον απαλλάσσει από αυτήν την υποχρέωση.

Η TORC παρέχει διεπαφή (API) για τις γλώσσες προγραμματισμού C και Fortran και μπορεί να λειτουργήσει τόσο σε συστήματα κοινόχρηστης, όσο και κατανεμημένης μνήμης. Στη δεύτερη περίπτωση, οι εφαρμογές που τη χρησιμοποιούν

αποτελούνται από πολλαπλές διεργασίες που δημιουργούνται με το MPI και καθεμιά εκτελείται σε διαφορετικό κόμβο του cluster. Επίσης, η αρχιτεκτονική της αποτελείται από νήματα δύο επιπέδων. Στο πρώτο επίπεδο, κάθε διεργασία αποτελείται από νήματα επιπέδου πυρήνα (kernel level thread), καθένα από τα οποία είναι ένας εργάτης που συνεχώς εκτελεί tasks που βρίσκονται στις ουρές εκτέλεσης. Στο δεύτερο επίπεδο, κάθε task ενσωματώνεται σε ένα νήμα επιπέδου χρήστη (user level thread) και εισάγεται σε μια από τις ουρές εκτέλεσης. Αυτή η αρχιτεκτονική επιτρέπει την ύπαρξη εμφωλευμένων tasks (nested tasks), δηλαδή tasks που με την σειρά τους δημιουργούν άλλα tasks, σε πολλαπλά επίπεδα.

Όπως αναφέραμε πριν, τα tasks μπορεί να εκτελεστούν είτε τοπικά, δηλαδή στον κόμβο που τα παρήγαγε, είτε απομακρυσμένα. Εσωτερικά, κάθε task αναπαρίσταται με μια δομή δεδομένων, που ονομάζεται *περιγραφέας task (task descriptor)*. Όλες οι μεταφορές δεδομένων μεταξύ των κόμβων πραγματοποιούνται με χρήση του MPI. Για τον σκοπό αυτό, κατά την εκκίνηση του προγράμματος, η TORC δημιουργεί ένα νήμα επιπέδου πυρήνα σε κάθε κόμβο, που είναι υπεύθυνο για την διαχείριση των ουρών εκτέλεσης και τις μεταφορές δεδομένων. Αυτές οι δύο λειτουργίες συμβαίνουν διαφανώς, δηλαδή χωρίς να τις καταλαβαίνει και χωρίς να χρειάζεται να κάνει κάτι για αυτές ο χρήστης της βιβλιοθήκης.

Η TORC υποστηρίζει δύο μοντέλα εκτέλεσης. Στο πρώτο, που ονομάζεται μοντέλο αφέντη–εργάτη (master–worker), μετά την αρχικοποίηση της βιβλιοθήκης, μόνο ένα νήμα σε μια διεργασία συνεχίζει με την εκτέλεση του κώδικα. Όλα τα υπόλοιπα νήματα σε όλους τους κόμβους μπλοκάρουν, μετατρέπονται, δηλαδή, σε εργάτες που περιμένουν για δουλειά. Στο δεύτερο μοντέλο εκτέλεσης, που ονομάζεται μοντέλο SPMD (Single Program Multiple Data), ένα νήμα σε κάθε διεργασία συνεχίζει με την εκτέλεση του κώδικα. Το δεύτερο μοντέλο χρησιμοποιείται σε προγράμματα που κάνουν ρητή χρήση του MPI. Μάλιστα, η TORC παρέχει την δυνατότητα με μια κλήση του API της να μεταφερθούμε από το ένα μοντέλο στο άλλο, γεγονός που διευκολύνει την ενσωμάτωση υπάρχοντος κώδικα MPI σε εφαρμογές που λειτουργούν με το μοντέλο αφέντη–εργάτη.

Για να λύσει τα προβλήματα που προκύπτουν από την ετερογένεια των κόμβων που απαρτίζουν το cluster, η TORC βασίζεται τόσο σε λύσεις που παρέχουν οι υλοποιήσεις του MPI, όσο και σε δικές της τεχνικές. Πιο συγκεκριμένα, για να επιλύσει ζητήματα που προκύπτουν με την ευθυγράμμιση μνήμης (memory alignment) μεταξύ διαφορετικών επεξεργαστών, τοποθετεί τον περιγραφέα του task σε

κατάλληλη θέση στην μνήμη και, αν αυτό δεν είναι δυνατόν, χρησιμοποιεί επιπλέον χώρο (padding). Για να διορθώσει το διαφορετικό endianness, δηλαδή ότι σε διαφορετικούς επεξεργαστές η αναπαράσταση των bits των αριθμών είναι με διαφορετική σειρά, βασίζεται στο MPI για τα δεδομένα και τα αποτελέσματα των tasks. Όμως, χειρίζεται μόνη της τη δομή του περιγραφέα του task: αν γίνει μεταφορά του περιγραφέα του task σε κόμβο με διαφορετικό endianness, τα bytes που αποτελούν κάθε πεδίο της δομής του αντιμετατίθενται.

Ακόμη, αν και όλοι οι κόμβοι τρέχουν το ίδιο εκτελέσιμο αρχείο, οι εικονικές διευθύνσεις των συναρτήσεων (κάθε task αναπαριστάται με μια συνάρτηση) είναι διαφορετικές σε κάθε κόμβο. Ο προγραμματιστής είναι υποχρεωμένος να καταχωρήσει όλα τα tasks πριν τα δημιουργήσει. Εσωτερικά, η TORC αντιστοιχεί κάθε συνάρτηση task σε έναν ακέραιο και αυτή η αντιστοίχιση είναι η ίδια σε κάθε κόμβο. Για τον σκοπό αυτό, κάθε κόμβος διατηρεί έναν πίνακα με τις αντιστοιχίσεις του. Ένα παρόμοιο πρόβλημα προκύπτει κατά την εκπομπή (broadcast) δεδομένων σε πολλαπλούς κόμβους, αλλά και αυτό λύνεται με παρόμοιο τρόπο.

Τέλος, αξίζει να αναφέρουμε ότι η TORC υποστηρίζει την εκτέλεση tasks σε κάρτες γραφικών (GPUs) χρησιμοποιώντας την βιβλιοθήκη OpenCL. Ο προγραμματιστής, χρησιμοποιώντας το API της TORC, μπορεί να κατανείμει τα tasks σε συγκεκριμένους κόμβους του cluster. Στη συνέχεια, η συνάρτηση του task μπορεί να εκτελεστεί είτε στον κεντρικό επεξεργαστή του συστήματος (CPU), είτε στην κάρτα γραφικών (GPU). Μάλιστα, η TORC μπορεί να αποφασίσει δυναμικά ποια είναι η καλύτερη επιλογή, ελέγχοντας αν εκείνη τη στιγμή η κάρτα γραφικών είναι ήδη απασχολημένη με την εκτέλεση κάποιου άλλου task.

Κεφαλαιο 5

Σχεδιασμος και υλοποιηση του ee_mpidsm

Στο κεφάλαιο αυτό περιγράφουμε τον σχεδιασμό και την υλοποίηση του μηχανισμού που αποτελεί κεντρικό αντικείμενο της εργασίας. Όπως ειπώθηκε και στην εισαγωγή, η εργασία μας βασίζεται, και αποτελεί συνέχεια προηγούμενων εργασιών [15, 16]. Παρότι η βασική ιδέα πίσω από κάποια τμήματα του μηχανισμού παραμένει, εντούτοις:

(α) έπρεπε να επανασχεδιαστούν τα περισσότερα τμήματα και να εισαχθεί επιπλέον η έννοια του task.

(β) επειδή οι προηγούμενες εργασίες έγιναν πριν από μια δεκαετία, η όποια υλοποίηση υπήρχε ήταν ουσιαστικά μη λειτουργική, μιας και το πρότυπο του OpenMP έχει μεταβληθεί σε τεράστιο βαθμό, και το ίδιο και ο μεταφραστής OMPi.

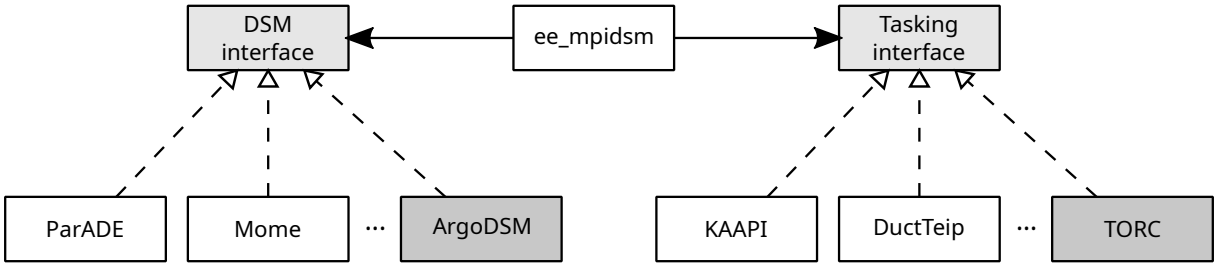
Ουσιαστικά, οι εργασίες [15, 16] αποτέλεσαν απλά έναν οδηγό για την δική μας εργασία. Πρόκειται για εξ ολοκλήρου νέο σχεδιασμό και νέα υλοποίηση.

Στις ακόλουθες ενότητες αναλύουμε διεξοδικά την υλοποίησής μας, μαζί με τα προβλήματα που αντιμετωπίσαμε κατά την ανάπτυξή της, τις εναλλακτικές επιλογές που είχαμε για την επίλυσή τους, καθώς και τις αποφάσεις που πήραμε τελικά.

5.1 Διεπαφή με βιβλιοθήκες sDSM και tasking

Ένας από τους βασικούς μας στόχους είναι να παρέχουμε μια υλοποίηση που δεν εξαρτάται άμεσα από μια συγκεκριμένη βιβλιοθήκη sDSM ή tasking. Με αυτό τον τρόπο επιτυγχάνουμε *επεκτασιμότητα*, δηλαδή την ικανότητα να αντικαταστήσουμε αυτές τις βιβλιοθήκες, αν το θελήσουμε στο μέλλον, χωρίς να χρειαστούν αλλαγές στον υπάρχων κώδικα. Επομένως, το `ee_mpidsm` βασίζεται σε δύο διεπαφές (interfaces) υψηλού επιπέδου, που εξασφαλίζουν λειτουργικότητα για sDSM και tasks, χωρίς να επικοινωνεί απευθείας με τις βιβλιοθήκες που υλοποιούν αυτές τις δυνατότητες. Στο Σχήμα 5.1 φαίνεται η γραφική αναπαράσταση της οργάνωσής μας. Ωστόσο, για να έχουμε μία λειτουργική υλοποίηση, προχωρήσαμε σε πλήρη ενσωμάτωση μιας σύγχρονης sDSM, της ArgoDSM, και της βιβλιοθήκης TORC για tasking, για του λόγους που αναλύσαμε στις Ενότητες 4.1 και 4.3, αντίστοιχα.

Αξίζει να σημειώσουμε ότι διεπαφή για τις βιβλιοθήκες sDSM υπήρχε και στην προηγούμενη υλοποίηση [15, 16], ωστόσο δεν παρείχε κάποιες λειτουργίες που χρειαστήκαμε, επομένως προχωρήσαμε σε μερικό επανασχεδιασμό της. Επίσης, εισάγαμε τη διεπαφή για βιβλιοθήκες tasking. Στις ακόλουθες υποενότητες περιγράφουμε λεπτομερώς αυτές τις διεπαφές.



Σχήμα 5.1: Η οργάνωση του `ee_mpidsm`. Για να κάνουμε την υλοποίησή μας εύκολα επεκτάσιμη, το `ee_mpidsm` δεν επικοινωνεί απευθείας με τις βιβλιοθήκες sDSM και tasking, αλλά με δύο διεπαφές (interfaces) υψηλού επιπέδου.

Πριν συνεχίσουμε με την περιγραφή των διεπαφών, είναι χρήσιμο να κάνουμε μια σύντομη παρουσίαση της οργάνωσης της υλοποίησής μας. Στοχεύουμε σε συστάδες (clusters) που αποτελούνται από πολλαπλούς κόμβους. Εκκινούμε μια διεργασία σε κάθε κόμβο. Κάθε κόμβος, ωστόσο, είναι ένα μικρό σύστημα κοινόχρηστης μνήμης και, προκειμένου να τον αξιοποιήσουμε πλήρως, χρησιμοποιούμε πολλαπλά νήματα σε κάθε κόμβο. Στην πλειοψηφία τους είναι νήματα εργάτες (worker threads) που εκτελούν το παράλληλο τμήμα της εφαρμογής. Σε κάθε κόμβο υπάρχει ακόμη ένα νήμα server, σκοπός του οποίου είναι να επικοινωνεί με τους υπόλοιπους κόμβους. Τέλος, το αρχικό νήμα στον αρχικό κόμβο είναι υπεύθυνο για την εκτέλεση του σειριακού τμήματος του προγράμματος. Στο βήμα (3) του Σχήματος 5.4 φαίνεται ένα παράδειγμα με 2 κόμβους.

5.1.1 Διεπαφή με βιβλιοθήκες sDSM

Η διεπαφή με βιβλιοθήκες sDSM αποτελείται από τις ακόλουθες συναρτήσεις:

- `void dsm_init(int *argc, char ***argv)`: Αρχικοποιεί τη βιβλιοθήκη sDSM. Καλείται μια φορά σε κάθε κόμβο κατά τη διαδικασία της αρχικοποίησης. Δέχεται ως παραμέτρους δείκτες στα ορίσματα `argc` και `argv` της συνάρτησης main του προγράμματος του χρήστη. Η βιβλιοθήκη MPI είναι ήδη αρχικοποιημένη σε αυτό το σημείο.

- `void dsm_finalize(void)`: Τερματίζει τη βιβλιοθήκη sDSM. Καλείται μια φορά σε κάθε κόμβο κατά τη διαδικασία του τερματισμού. Δεν πρέπει να τερματίσει το MPI.

- `int oprc_pid(void)`: Επιστρέφει την ταυτότητα (0 έως $n-1$) της διεργασίας (κόμβου) που την κάλεσε.

- `int oprc_procs(void)`: Επιστρέφει το συνολικό πλήθος των διεργασιών (κόμβων) που εκτελούν το πρόγραμμα.

- `void oprc_shmalloc(void **p, size_t size, int *memid)`: Δεσμεύει μνήμη μεγέθους size στον κοινόχρηστο χώρο και τοποθετεί έναν δείκτη σε αυτή τη μνήμη στον p. Η κλήση είναι συλλογική, δηλαδή καλείται από κάθε κόμβο.

- `void oprc_shmfree(int *p)`: Απελευθερώνει μνήμη που είχε προηγουμένως δεσμευτεί στον κοινόχρηστο χώρο και στην οποία δείχνει ο p. Καλείται από έναν

μόνο κόμβο.

- `void oprc_barrier(void)`: Κλήση barrier με συγχρονισμό μνήμης. Είναι συλλογική, δηλαδή καλείται από κάθε κόμβο.
- `int oprc_lock_init(void)`: Αρχικοποιεί μία νέα κλειδαριά και επιστρέφει το αναγνωριστικό (id) ως ακέραιο.
- `void oprc_lock_destroy(int lock)`: Καταστρέφει την κλειδαριά στην οποία αντιστοιχεί το αναγνωριστικό που δέχεται ως παράμετρο.
- `void oprc_lock_acquire(int lock)`: Κλειδώνει την κλειδαριά στην οποία αντιστοιχεί το αναγνωριστικό που δέχεται ως παράμετρο.
- `void oprc_lock_release(int lock)`: Ξεκλειδώνει την κλειδαριά στην οποία αντιστοιχεί το αναγνωριστικό που δέχεται ως παράμετρο.

Οποιαδήποτε βιβλιοθήκη sDSM υποστηρίζει αυτή τη λειτουργικότητα μπορεί να χρησιμοποιηθεί από την υλοποίησή μας, χωρίς καμία αλλαγή στον υπάρχων κώδικα. Η λειτουργικότητα που ζητάμε είναι τυπική για βιβλιοθήκες sDSM, με εξαίρεση ίσως τις συναρτήσεις για χειρισμό κατανεμημένων barriers και κλειδαριών. Επίσης, χρειάζεται προσοχή με τη σημασιολογία των συναρτήσεων. Πιο συγκεκριμένα, η συνάρτηση για δέσμευση μνήμης θα κληθεί από ένα νήμα σε κάθε κόμβο, ενώ για αποδέσμευση μνήμης και δημιουργία κλειδαράς θα κληθούν από ένα νήμα σε ένα μόνο κόμβο.

Για παράδειγμα, αν σε μια άλλη βιβλιοθήκη sDSM η δέσμευση μνήμης δεν είναι συλλογική, τότε κατά την υλοποίηση της συνάρτησης `oprc_shmalloc` μόνο ένα νήμα (π.χ. αυτό που βρίσκεται στον πρώτο κόμβο) θα πρέπει να καλέσει τη συνάρτηση της βιβλιοθήκης. Αντίθετα, αν η συνάρτηση αποδέσμευσης μνήμης είναι συλλογική, τότε κατά την υλοποίηση της `oprc_shmfree` θα πρέπει να στέλνεται ένα μήνυμα στα νήματα server των υπόλοιπων κόμβων, ώστε να κληθεί η συνάρτηση της βιβλιοθήκης από όλους τους κόμβους.

5.1.2 Τροποποιήσεις στην ArgoDSM

Για να χρησιμοποιήσουμε την ArgoDSM στην υλοποίησή μας, χρειάστηκε να κάνουμε ορισμένες μικρές αλλαγές στον πηγαίο κώδικά της. Αυτές συνοψίζονται στην ακόλουθη λίστα.

- Κάναμε τον χειριστή του σήματος segmentation fault να εκτελείται σε εναλλακτική στοίβα, την οποία δεσμεύουμε στην ιδιωτική μνήμη του εκάστοτε κόμβου. Αυτό χρειάστηκε για να αποφύγουμε προβλήματα που αναλύουμε στις ακόλουθες ενότητες.

- Προσθέσαμε τις συναρτήσεις `argo_acquire` και `argo_release` (που αφορούν το συγχρονισμό μνήμης) στο API που εξάγει η ArgoDSM για προγράμματα C. Μέχρι τώρα, αυτές οι συναρτήσεις ήταν διαθέσιμες μόνο από το API της C++.

- Σβήσαμε τις κλήσεις `MPI_Init_thread` και `MPI_Finalize`. Τώρα η αρχικοποίηση και ο τερματισμός του MPI γίνεται από την TORC. Φροντίζουμε να αρχικοποιήσουμε την TORC πριν την ArgoDSM και να την τερματίσουμε μετά την ArgoDSM, ώστε το MPI να είναι πάντα διαθέσιμο στην ArgoDSM.

5.1.3 Διεπαφή με βιβλιοθήκες tasking

Η διεπαφή με βιβλιοθήκες tasking αποτελείται από τις ακόλουθες συναρτήσεις:

- `void tasking_init(int *argc, char ***argv)`: Αρχικοποιεί τη βιβλιοθήκη tasking, η οποία θα πρέπει να δημιουργήσει τα νήματα workers σε κάθε κόμβο. Καλείται μια φορά σε κάθε κόμβο κατά την αρχικοποίηση του προγράμματος. Δέχεται ως παραμέτρους δείκτες στα ορίσματα `argc` και `argv` της συνάρτησης main του προγράμματος του χρήστη. Πρέπει να φροντίσει να αρχικοποιήσει τη βιβλιοθήκη MPI.

- `void oprc_create_initial_task(void (*worker_func)(), int id)`: Δημιουργεί το αρχικό task που τα νήματα workers θα πρέπει να εκτελούν σε όλη τη διάρκεια του προγράμματος. Δουλειά του είναι να εξασφαλίζει ότι τα νήματα workers περιμένουν για οδηγίες από το νήμα server του εκάστοτε κόμβου και τις εκτελούν. Το task αυτό τελειώνει λίγο πριν τον τερματισμό του προγράμματος. Δέχεται ως παράμετρο το τοπικό αναγνωριστικό (id) του νήματος worker που θα αναλάβει να το εκτελέσει ως ακέραιο. Το task αυτό πρέπει να είναι δεμένο (tied) στο νήμα που το εκτελεί, δηλαδή πρέπει να εκτελεστεί από την αρχή έως το τέλος του από το ίδιο νήμα.

- `int oprc_get_num_local_workers(void)`: Επιστρέφει το πλήθος των νημάτων worker στον κόμβο από τον οποίο καλείται, δηλαδή το πλήθος των νημάτων που μπορούν να εκτελούν tasks.

- `void oprc_new_task(void *(*func)(void *arg), void *arg)`: Δημιουργεί ένα νέο task που αποτελείται από τη συνάρτηση func. Το task δέχεται ως μοναδικό όρισμα ένα δείκτη σε δομή, η οποία θα πρέπει να είναι προσπελάσιμη από κόμβο στον οποίο τελικά θα εκτελεστεί το task.

- `void oprc_taskwait(int how, void *info, int thread_num)`: Εκτελεί όλα τα tasks που έχουν δημιουργηθεί ως αυτό το σημείο και είναι εκκρεμή. Μπορεί να κληθεί από νήμα worker που προηγουμένως δεν έχει δημιουργήσει κανένα task. Σε αυτή την περίπτωση, θα πρέπει να εκτελέσει tasks που έχουν δημιουργηθεί από άλλα νήματα workers. Η βιβλιοθήκη δεν χρειάζεται να ανησυχεί για τις παραμέτρους how και info μιας και στην τρέχουσα υλοποίησή μας δεν χρησιμοποιούνται.

- `void *oprc_taskenv_alloc(int size, void *task_func)`: Δεσμεύει μνήμη μεγέθους size για την παράμετρο του task που περιγράφεται από την συνάρτηση task_func. Είτε αυτή η συνάρτηση, είτε η `oprc_new_task` θα πρέπει να φροντίσει ώστε η μνήμη που δεσμεύεται να είναι διαθέσιμη σε όλους τους κόμβους, αν η βιβλιοθήκη tasking υποστηρίζει κλέψιμο (stealing) tasks από διαφορετικούς κόμβους. Η συνάρτηση επιστρέφει ένα δείκτη στη μνήμη που δέσμευσε.

- `void oprc_taskenv_free(void *arg)`: Απελευθερώνει τη μνήμη που είχε δεσμευτεί σε μια προηγούμενη κλήση της `oprc_taskenv_alloc`. Αν κατά τη δημιουργία του task η παράμετρος που δέχεται το task *δεν* αντιγράφτηκε από τη βιβλιοθήκη, αλλά χρησιμοποιήθηκε αυτούσια, τότε ο δείκτης που δέχεται ως όρισμα αυτή η συνάρτηση, θα είναι ίδιος με κάποιο δείκτη που είχε επιστρέψει η `oprc_taskenv_alloc`. Σε διαφορετική περίπτωση, όμως, ο δείκτης θα είναι εντελώς διαφορετικός, μιας και αυτή η συνάρτηση καλείται μέσα από το task και χρειαζόμαστε διαφορετικό τρόπο για να αποδεσμεύσουμε αυτή τη μνήμη (πιθανώς αμέσως μετά τη δημιουργία του task). Αναλύουμε περισσότερο αυτό το ζήτημα στην Ενότητα 5.5.

- `void tasking_finalize(void)`: Τερματίζει τη βιβλιοθήκη tasking. Καλείται μία φορά σε κάθε κόμβο κατά τον τερματισμό του προγράμματος. Πρέπει να φροντίσει να τερματίσει τη βιβλιοθήκη MPI, καθώς και τα νήματα workers.

Οποιαδήποτε βιβλιοθήκη tasking υποστηρίζει αυτή την λειτουργικότητα μπορεί να χρησιμοποιηθεί στην υλοποίησή μας. Μπορεί να φαίνονται πολλές, αλλά οι βα-

σικές συναρτήσεις είναι δύο: αυτή που δημιουργεί και αυτή που εκτελεί τα tasks. Η μοναδική δυσκολία που μπορεί να προκύψει στην επεκτασιμότητα είναι η ιδιαιτερότητα της TORC που περιγράφουμε στην Ενότητα 5.3.1. Συνοπτικά, το πρόβλημα είναι ότι η βιβλιοθήκη tasking θα πρέπει να είναι υπεύθυνη για την δημιουργία των νημάτων workers κατά την αρχικοποίηση του προγράμματος και την καταστροφή τους στον τερματισμό του. Ίσως να μην είναι πολύ συνηθισμένο για μια βιβλιοθήκη tasking να παρέχει αυτή τη δυνατότητα.

5.1.4 Τροποποιήσεις στην TORC

Για να χρησιμοποιήσουμε την TORC στην υλοποίησή μας, χρειάστηκε να κάνουμε ορισμένες αλλαγές στον πηγαίο κώδικά της, που απαίτησαν εμβάθυνση στον τρόπο λειτουργίας της και στην οργάνωση του κώδικα. Σκοπός των αλλαγών ήταν να υποστηρίξουμε λειτουργικότητα που χρειαζόμασταν και δεν υπήρχε ήδη στην TORC και η οποία είναι χρήσιμη να υπάρχει σε βιβλιοθήκες tasking. Οι αλλαγές αυτές συνοψίζονται στην ακόλουθη λίστα.

- Η TORC αποθηκεύει τα tasks προς εκτέλεση σε διάφορες ουρές. Δημιουργήσαμε μια επιπλέον ιδιωτική ουρά για κάθε νήμα κάθε κόμβου, από την οποία δεν μπορούν να κλαπούν tasks (task stealing). Σκοπός αυτής της προσθήκης ήταν να επιτρέψουμε την ύπαρξη δεμένων (tied) tasks και για την πραγματοποίησή της απαιτήθηκε επέμβαση στον τρόπο λειτουργίας των ουρών.

- Κάναμε όλα τα νήματα workers να ψάχνουν και να εκτελούν εκκρεμή tasks όταν καλούν την συνάρτηση `torc_waitall` με σκοπό να αυξήσουμε την αποδοτικότητα της υλοποίησής μας. Μέχρι τώρα, τα νήματα workers που δεν δημιουργούσαν tasks δεν έμπαιναν στην διαδικασία να εκτελέσουν κανένα task. Για αυτήν την αλλαγή, χρειάστηκαν αλλαγές στον πυρήνα της TORC που αφορά το μηχανισμό εκτέλεσης tasks (που γίνεται με εναλλαγές νημάτων επιπέδου χρήστη) και τις μεταβάσεις που γίνονται από την εκτέλεση ενός task στον δρομολογητή (scheduler) των tasks και αντίστροφα.

- Για λόγους που θα εξηγήσουμε σε επόμενη ενότητα, τελικά χρειάστηκε να απενεργοποιήσουμε το κλέψιμο (stealing) tasks μεταξύ κόμβων. Για να το πετύχουμε αυτό, χρειάστηκε και μια αλλαγή στον κώδικα που δημιουργεί τα

tasks: τα καινούργια tasks έπρεπε να εισάγονται μόνο στις ουρές των τοπικών νημάτων workers, όχι των απομακρυσμένων.

5.2 Κοινόχρηστες μεταβλητές

Το πρότυπο OpenMP έχει σχεδιαστεί με συστήματα κοινόχρηστης μνήμης κατά νου και συνεπώς υποθέτει ότι κάποιες μεταβλητές, όπως για παράδειγμα οι καθολικές (global), είναι εξ ορισμού κοινόχρηστες μεταξύ των νημάτων. Προκειμένου η υλοποίηση μας να είναι συμβατή με το πρότυπο OpenMP, θα πρέπει να εξασφαλίσουμε ότι αυτές οι μεταβλητές συμπεριφέρονται σαν να είμαστε σε σύστημα κοινόχρηστης μνήμης, παρόλο που βρισκόμαστε σε κατανεμημένο περιβάλλον. Στις ακόλουθες υποενότητες αναλύουμε τις επιλογές που έχουμε για κάθε είδος μεταβλητής, μαζί με τα πλεονεκτήματα και τα μειονεκτήματά τους, καθώς και τις αποφάσεις που πήραμε τελικά.

5.2.1 Καθολικές (global) μεταβλητές, στατικές (static) και δηλωμένες ως extern

Σε αυτή την κατηγορία ανήκουν οι μεταβλητές που έχουν δηλωθεί έξω από οποιαδήποτε συνάρτηση είτε στον τρέχον αρχείο κώδικα (καθολικές), είτε σε κάποιο άλλο (δηλωμένες ως extern), καθώς και οι μεταβλητές που έχουν δηλωθεί μέσα σε κάποια συνάρτηση με τον προσδιορισμό static. Σε ένα σύστημα κοινόχρηστης μνήμης όπου μια διεργασία αποτελείται από νήματα, το λειτουργικό σύστημα φροντίζει ώστε οι μεταβλητές αυτές να βρίσκονται σε τμήμα της μνήμης που είναι προσπελάσιμο και εγγράψιμο από όλα τα νήματα.

Μια προσέγγιση θα ήταν να αναπαράγουμε αυτή τη συμπεριφορά. Με τη βοήθεια του συνδέτη (linker), μπορούμε να μεταφράσουμε το πρόγραμμα του χρήστη χρησιμοποιώντας ένα τροποποιημένο σενάριο σύνδεσης (linker script), το οποίο φροντίζει να απομονώσει τις ενότητες (sections) `bss` και `data` που περιέχουν αυτές τις μεταβλητές, σε μια καινούργια ενότητα. Αυτή μπορεί να είναι ευθυγραμμισμένη ως προς το μέγεθος μιας σελίδας μνήμης (page aligned) για να μην έχει πρόβλημα η συνάρτηση `mprotect` που χρησιμοποιεί η ArgoDSM για να αλλάζει τα δικαιώματα των σελίδων. Επίσης, με το τροποποιημένο σενάριο σύνδεσης είναι εύκολο να γνω-

ρίζουμε τόσο την αρχή, όσο και το μέγεθος του νέου τμήματος.

Η δυσκολία με αυτή την προσέγγιση είναι ότι η βιβλιοθήκη ArgoDSM θα χρειαζόταν ριζικές αλλαγές, αν όχι ολική επανεγγραφή. Εκτός από την ποσότητα μνήμης που δεσμεύει, κατανέμει στους κόμβους, ενημερώνει κατάλληλα και της αλλάζει τα δικαιώματα, θα έπρεπε να διαχειρίζεται και ένα επιπλέον τμήμα μνήμης το οποίο δε θα χρειαζόταν να δεσμεύει, αλλά θα έπρεπε να κατανείμει στους κόμβους και να τροποποιήσει κατάλληλα. Έτσι, αποφασίσαμε να εγκαταλείψουμε αυτή την προσέγγιση.

Η εναλλακτική λύση είναι να αφήσουμε το μεταφραστή να κάνει όλη τη δύσκολη δουλειά. Κατά τη μετάφραση του προγράμματος του χρήστη, μετράμε το συνολικό μέγεθος που χρειάζεται για την αποθήκευση όλων των μεταβλητών που ανήκουν σε αυτή την κατηγορία. Επίσης, μετασχηματίζουμε αυτές τις μεταβλητές σε δείκτες. Κατά την εκτέλεση του προγράμματος και πριν την εκτέλεση της συνάρτησης `main` του χρήστη, δεσμεύουμε την απαιτούμενη ποσότητα στην κοινόχρηστη μνήμη και τροποποιούμε τους δείχτες ώστε να δείχνουν στην κοινόχρηστη μνήμη. Παρότι αυτή προσέγγιση δεν είναι ιδιαίτερα κομψή, προϋπήρχε στην υλοποίηση του OMPi και δουλεύει χωρίς προβλήματα, οπότε αποφασίσαμε να την ακολουθήσουμε.

5.2.2 Στοίβα του αρχικού νήματος

Ας σκεφτούμε τώρα την ακόλουθη περίπτωση: στην αρχή του προγράμματος του χρήστη υπάρχει η δήλωση `int x`. Το πρόγραμμα του χρήστη εκτελείται σειριακά, δηλαδή από ένα νήμα, έως ότου βρεθεί παράλληλη περιοχή. Η τοπική μεταβλητή `x` θα αποθηκευτεί, όπως όλες οι τοπικές μεταβλητές, στη στοίβα του αρχικού αυτού νήματος (initial thread, σύμφωνα με την ορολογία του OpenMP). Έστω ότι στη συνέχεια υπάρχει η οδηγία `#pragma omp parallel shared(x)`, που σημαίνει ότι δημιουργείται μια παράλληλη περιοχή στην οποία η `x` θα είναι κοινόχρηστη, άρα όλα τα νήματα θα πρέπει να μπορούν να τη διαβάζουν και να τη γράφουν. Σε ένα σύστημα κοινόχρηστης μνήμης όπου όλα τα νήματα ανήκουν σε μια διεργασία, ο μεταφραστής θα αντικαταστήσει τις αναφορές στη μεταβλητή `x` μέσα στην παράλληλη περιοχή με ένα δείκτη που δείχνει στη `x` στη στοίβα του πρώτου νήματος. Εκεί έχουν δικαίωμα ανάγνωσης και εγγραφής όλα τα νήματα της διεργασίας. Στο τέλος της παράλληλης περιοχής, το αρχικό νήμα θα έχει τη σωστή τιμή της μεταβλητής στη στοίβα του, όπως θα έπρεπε.

Όμως, σε ένα κατανεμημένο περιβάλλον τα νήματα είναι μοιρασμένα σε διαφορετικές διεργασίες που μάλιστα εκτελούνται σε διαφορετικούς κόμβους. Έτσι, ένα νήμα δεν είναι δυνατόν να έχει πρόσβαση στην στοίβα ενός άλλου νήματος, αν αυτό εκτελείται σε διαφορετικό κόμβο (και συνεπώς σε διαφορετική διεργασία). Όπως στην προηγούμενη περίπτωση, έτσι και εδώ, υπάρχουν δύο τρόποι να αντιμετωπίσουμε αυτό το πρόβλημα.

Η πρώτη προσέγγιση είναι να βάλουμε το μεταφραστή να λύσει το πρόβλημα. Κατά τη μεταγλώττιση του προγράμματος του χρήστη, θα πρέπει να μετράμε το συνολικό μέγεθος των τοπικών μεταβλητών κάθε συνάρτησης και να μετασχηματίζουμε τις μεταβλητές σε δείκτες. Πριν από την εκτέλεση κάθε συνάρτησης πρέπει να δεσμεύουμε χώρο στην κοινόχρηστη μνήμη για τις τοπικές μεταβλητές και να ενημερώνουμε κατάλληλα τους δείκτες. Αυτό πρέπει να το κάνουμε κάθε φορά που θέλουμε να εκτελέσουμε τη συνάρτηση, διότι η συνάρτηση μπορεί να καλείται από διαφορετικά μέρη στον κώδικα ή από το ίδιο μέρος αναδρομικά. Αυτή η προσέγγιση θα απαιτούσε σημαντικές αλλαγές στον μεταφραστή, οπότε την αγνοήσαμε.

Η εναλλακτική (και πιο κομψή) προσέγγιση είναι να αποθηκεύουμε ολόκληρη τη στοίβα του νήματος που εκτελεί το σειριακό κομμάτι του προγράμματος του χρήστη στην κοινόχρηστη μνήμη. Αυτό μπορεί να γίνει εύκολα, αρκεί να δημιουργήσουμε ένα καινούργιο νήμα. Οι βιβλιοθήκες διαχείρισης νημάτων μας δίνουν τη δυνατότητα να δεσμεύσουμε μόνοι μας τη μνήμη που θα χρησιμοποιηθεί ως στοίβα για τα νήματα που δημιουργούμε. Η δημιουργία ενός νήματος επιπέδου πυρήνα (kernel level thread) μέσω της βιβλιοθήκης pthreads θα ήταν υπερβολή, αφού το υπάρχον νήμα πρέπει απλά να κάθεται και να περιμένει το καινούργιο να τελειώσει. Έτσι, αποφασίσαμε να βασιστούμε σε ένα νήμα επιπέδου χρήστη (user level thread) χρησιμοποιώντας τις συναρτήσεις `getcontext`, `makecontext` και `swapcontext`, το μέγεθος της στοίβας του οποίου καθορίζεται από τη σταθερά `SHARED_STACK_SIZE`, που έχει προεπιλεγμένη τιμή 8 MB και είναι εύκολα τροποποιήσιμη.

Στην υλοποίησή μας *δεν* υποστηρίζουμε ένθετο παραλληλισμό (nested parallelism), δηλαδή *δεν* επιτρέπεται να δημιουργήσουμε μια παράλληλη περιοχή μέσα σε μια παράλληλη περιοχή. Αν θέλαμε να υποστηρίξουμε κάτι τέτοιο, θα έπρεπε να τοποθετούμε στην κοινόχρηστη μνήμη τις στοίβες όλων των νημάτων από όλους τους κόμβους. Από την άλλη, θα μπορούσαμε, ως μελλοντική εργασία, να υποστηρίξουμε ένθετο παραλληλισμό εφόσον όμως τα εσωτερικά επίπεδα νημάτων βρίσκονταν όλα στον ίδιο κόμβο-διεργασία και άρα είχαν απευθείας πρόσβαση στις στοίβες των

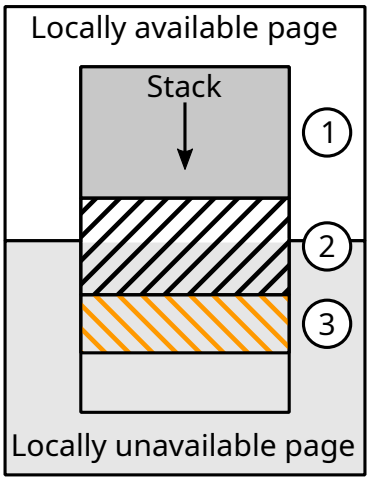
υπόλοιπων νημάτων.

Η απόφασή μας να αποθηκεύσουμε τη στοίβα του πρώτου νήματος στην κοινόχρηστη μνήμη κρύβει δύο πολύ καλά κρυμμένα και άκρως καταστροφικά προβλήματα. Όπως εξηγήσαμε στην Ενότητα 4.1, οι βιβλιοθήκες sDSM χωρίζουν τη μνήμη σε επιμέρους τμήματα (συνήθως ίσα με το μέγεθος μιας σελίδας) και δουλεύουν εκμεταλλευόμενες το χειριστή του σήματος segmentation fault.

Ας σκεφτούμε την ακόλουθη περίπτωση, που φαίνεται γραφικά στο Σχήμα 5.2: καθώς το αρχικό νήμα καλεί συναρτήσεις και χρησιμοποιεί όλο και περισσότερη από τη στοίβα του (βήμα (1)), κάποια στιγμή φτάνει στο όριο μιας σελίδας (βήμα (2)). Η επόμενη σελίδα, αν και αποθηκεύει (μεταξύ άλλων) την στοίβα του νήματος, δεν είναι διαθέσιμη τοπικά, επομένως προκαλείται segmentation fault και καλείται ο χειριστής του σήματος. Ο χειριστής του σήματος, όμως, είναι και αυτός μια συνάρτηση και χρειάζεται χώρο στη στοίβα για να αποθηκεύσει τις τοπικές μεταβλητές του, όπως φαίνεται στο βήμα (3). Μόλις πάει να γράψει στη στοίβα, "πέφτει" σε μη διαθέσιμη διεύθυνση και έτσι ο χειριστής του segmentation fault προκαλεί και αυτός segmentation fault. Το λειτουργικό σύστημα καταλαβαίνει ότι αν αφήσει αυτή την κατάσταση να εξελιχθεί, θα δημιουργηθεί μια ακολουθία από άπειρα segmentation faults και έτσι τερματίζει βίαια το πρόγραμμά μας.

Ευτυχώς, οι σχεδιαστές του λειτουργικού συστήματος έχουν προνοήσει και για αυτή την περίπτωση και μας δίνουν τη δυνατότητα να χρησιμοποιήσουμε εναλλακτική στοίβα για την κλήση ενός χειριστή σήματος. Αρχικά, τροποποιούμε την ArgoDSM προσθέτοντας την επιλογή `SA_ONSTACK` στις σημαίες (flags) που παίρνει ως όρισμα η συνάρτηση `sigaction` για να καθορίσουμε ότι ο χειριστής του σήματος segmentation fault της ArgoDSM πρέπει να κληθεί σε εναλλακτική στοίβα. Στη συνέχεια, δεσμεύουμε μνήμη τοπικά (με χρήση της `malloc`) για τη στοίβα του χειριστή. Τέλος, με τη συνάρτηση `sigaltstack` ενημερώνουμε το λειτουργικό σύστημα για την ύπαρξη της εναλλακτικής στοίβας για τους χειριστές σημάτων.

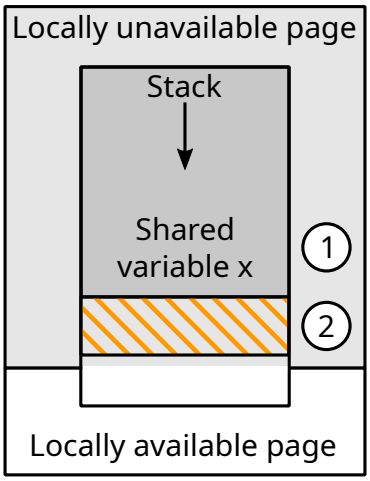
Το δεύτερο πρόβλημα σχετίζεται με το πρώτο. Όταν το αρχικό νήμα προσπαθεί να προσπελάσει τα κοινόχρηστα δεδομένα μιας σελίδας μνήμης που δεν είναι διαθέσιμη τοπικά, παράγεται ένα σήμα segmentation fault και καλείται ο χειριστής σφάλματος, που φροντίζει να ενημερώσει τα περιεχόμενα της ζητούμενης σελίδας με την πιο πρόσφατη έκδοση. Όμως, για να μπορέσει να συνεχιστεί η εκτέλεση του προγράμματος του χρήστη από το σημείο που βρισκόταν πριν συμβεί το segmentation fault, ο χειριστής του σφάλματος πρέπει να αποθηκεύσει κάποιες πληροφορίες



Σχήμα 5.2: Πρώτο πρόβλημα με την κοινόχρηστη στοίβα. Βήμα (1): η κοινόχρηστη στοίβα αποθηκεύεται σε διαδοχικές σελίδες μνήμης, μια από τις οποίες δεν είναι διαθέσιμη τοπικά. Βήμα (2): Τα δεδομένα που μια συνάρτηση θέλει να αποθηκεύσει στη στοίβα ξεπερνούν τα όρια της σελίδας, με αποτέλεσμα να προκληθεί segmentation fault. Βήμα (3): Ο χειριστής του σήματος είναι συνάρτηση και χρειάζεται να αποθηκεύσει τα δεδομένα του στη στοίβα, προκαλώντας νέο segmentation fault, αφού η στοίβα βρίσκεται σε μια σελίδα που δεν είναι διαθέσιμη τοπικά.

στη στοίβα του νήματος. Αν τύχει αυτές οι πληροφορίες να πρέπει να γραφούν στην ίδια μη διαθέσιμη σελίδα που περιέχει τα δεδομένα που ζήτησε αρχικά το νήμα, προκαλείται segmentation fault μέσα στον χειριστή του σήματος, με τις συνέπειες που περιγράψαμε πιο πάνω.

Αυτό μπορεί να συμβεί στην ακόλουθη περίπτωση, που φαίνεται γραφικά στο Σχήμα 5.3: το αρχικό νήμα αποθηκεύει στην κοινόχρηστη στοίβα του τη μεταβλητή x και στη συνέχεια δημιουργεί μια παράλληλη περιοχή, στην οποία η μεταβλητή x είναι κοινόχρηστη μεταξύ των νημάτων. Μέσα στην παράλληλη περιοχή ένα νήμα που βρίσκεται σε διαφορετικό κόμβο τροποποιεί την μεταβλητή x (χρησιμοποιώντας συγχρονισμό με κλειδαριές). Τότε η σελίδα μνήμης του αρχικού νήματος που περιέχει τη μεταβλητή x ακυρώνεται, όπως φαίνεται στο βήμα (1) του Σχήματος 5.3. Αν τώρα το αρχικό νήμα θελήσει να διαβάσει τη x, θα προκληθεί segmentation fault και, αφού η x βρίσκεται στην ίδια σελίδα μνήμης με την τρέχουσα θέση της στοίβας του νήματος, θα προκληθεί segmentation fault και στο χειριστή του σήματος, που θα προσπαθήσει να γράψει πληροφορίες στην στοίβα του αρχικού νήματος, όπως φαίνεται στο βήμα (2) του Σχήματος 5.3. Υπάρχουν δύο τρόποι για να αποτρέψουμε να συμβεί αυτό το πρόβλημα.



Σχήμα 5.3: Δεύτερο πρόβλημα με την κοινόχρηστη στοίβα. Βήμα (1): Η κοινόχρηστη μεταβλητή x αποθηκεύεται στην στοίβα του αρχικού νήματος, που είναι στην κοινόχρηστη μνήμη. Ένας άλλος κόμβος τροποποιεί την x, άρα η σελίδα του αρχικού νήματος δεν είναι διαθέσιμη τοπικά. Βήμα (2): Το αρχικό νήμα θέλει να προσπελάσει την x, προκαλείται segmentation fault και χειριστής του σήματος αποθηκεύει κάποια δεδομένα στην στοίβα ώστε να μπορεί να συνεχίσει η εκτέλεση μετά. Επειδή η σελίδα δεν είναι διαθέσιμη, προκαλείται segmentation fault μέσα στον χειριστή.

Η πρώτη λύση είναι να εξασφαλίσουμε ότι οι μεταβλητές που αποθηκεύονται στην στοίβα του αρχικού νήματος βρίσκονται σε διαφορετική σελίδα από την υπόλοιπη στοίβα του νήματος. Αυτό μπορεί να επιτευχθεί αν βάλουμε το μεταφραστή να εισάγει έναν επιπλέον πίνακα, με μέγεθος όσο και το μέγεθος μιας σελίδας μνήμης, μετά το τέλος των δηλώσεων των τοπικών μεταβλητών στο πρόγραμμα του χρήστη (padding). Τα μειονεκτήματα αυτής της προσέγγισης είναι ότι απαιτεί αλλαγές στον μεταφραστή και θυσιάζει πολύτιμες σελίδες μνήμη, επομένως δεν την ακολουθήσαμε.

Η εναλλακτική λύση, την οποία και ακολουθήσαμε, είναι να βάζουμε το αρχικό νήμα να εκτελεί την παράλληλη περιοχή σε ιδιωτική στοίβα, δηλαδή στοίβα που έχουμε δεσμεύσει στην τοπική και όχι την κοινόχρηστη μνήμη. Η αλλαγή στοίβας είναι εύκολο και υπολογιστικά φθηνό να γίνει, αφού χρησιμοποιούμε νήματα επιπέδου χρήστη. Με αυτό τον τρόπο, αν προκληθεί segmentation fault, ο χειριστής του σήματος θα μπορεί πάντα να γράψει τις πληροφορίες που χρειάζεται στη στοίβα του αρχικού νήματος χωρίς πρόβλημα.

5.2.3 Μνήμη στο σωρό (heap)

Σε αυτή την κατηγορία ανήκει η μνήμη που δεσμεύεται δυναμικά κατά την εκτέλεση του προγράμματος, χρησιμοποιώντας συναρτήσεις όπως η `malloc`. Σε ένα σύστημα κοινόχρηστης μνήμης όπου μια διεργασία αποτελείται από νήματα, αυτή η μνήμη δεσμεύεται στο σωρό, όπου έχουν δικαίωμα ανάγνωσης και εγγραφής όλα τα νήματα. Αντίθετα, σε ένα σύστημα κατανεμημένης μνήμης, κάθε διεργασία σε κάθε κόμβο έχει δικό της ιδιωτικό σωρό στον οποίον έχουν προφανώς πρόσβαση μόνο τα δικά της νήματα.

Η πρώτη προσέγγιση για να κάνουμε την μνήμη που δεσμεύεται στο σωρό κοινόχρηστη σε ένα κατανεμημένο περιβάλλον είναι να αποκρύψουμε εντελώς την ύπαρξή της από τον τελικό χρήστη. Με τη βοήθεια του συνδέτη (linker) μπορούμε να ανακατευθύνουμε όλες τις συναρτήσεις που καλούνται από το πρόγραμμα του χρήστη και δεσμεύουν μνήμη στο σωρό. Δηλαδή, όταν το πρόγραμμα του χρήστη καλεί μια συνάρτηση που δεσμεύει μνήμη στο σωρό, όπως η `malloc`, αντί να κληθεί η `malloc` του συστήματος, μπορεί να κληθεί μια δική μας έκδοση, για παράδειγμα η `my_malloc`, η οποία φροντίζει να δεσμεύσει τη ζητούμενη ποσότητα μνήμης στην κοινόχρηστη μνήμη αντί για το σωρό. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι το πρόγραμμα του χρήστη θα δουλεύει εγγυημένα, χωρίς να χρειάζεται η παραμικρή αλλαγή στον κώδικά του.

Το μειονέκτημα αυτής της προσέγγισης είναι ότι γίνεται πολύ δραστική. Δεσμεύει πάντα χώρο στην κοινόχρηστη μνήμη για να είναι σίγουρη, ακόμη και όταν αυτό δεν είναι θεμιτό. Για παράδειγμα, ένα νήμα μπορεί να χρειάζεται να δεσμεύσει δυναμικά προσωρινή μνήμη για να εκτελέσει τους υπολογισμούς του. Αν τα υπόλοιπα νήματα δεν έχουν σκοπό να προσπελάσουν αυτή την προσωρινή μνήμη, τότε δεν υπάρχει λόγος να τη δεσμεύουμε στον κοινόχρηστο χώρο, που είναι υπολογιστικά πιο ακριβός.

Έτσι, στην προσέγγιση που ακολουθήσαμε αφήνουμε περισσότερη ελευθερία, αλλά και υποχρεώσεις, στον τελικό χρήστη. Πιο συγκεκριμένα, του παρέχουμε τη συνάρτηση `oprc_dsm_alloc`, που παίρνει ως μοναδικό όρισμα το μέγεθος μνήμης που πρέπει να δεσμεύσει στον κοινόχρηστο χώρο και επιστρέφει ένα δείκτη σε αυτό το χώρο. Τώρα είναι ευθύνη του τελικού χρήστη να ελέγξει τον κώδικά του για συναρτήσεις που δεσμεύουν μνήμη στο σωρό και να αποφασίσει αν αυτή η μνήμη χρειάζεται να είναι κοινόχρηστη μεταξύ των νημάτων. Γενικά αναμένουμε ότι αυτές

οι κλήσεις θα είναι λίγες στα περισσότερα προγράμματα, άρα ο τελικός χρήστης δε θα επιβαρυνθεί με πολλή επιπλέον δουλειά.

Το πρόβλημα με τη συνάρτησή μας είναι ότι *δεν* είναι ασφαλής αν κληθεί ταυτόχρονα από πολλαπλά νήματα (thread safe). Αυτό συμβαίνει διότι η συνάρτηση δέσμευσης κοινόχρηστης μνήμης στην ArgoDSM είναι συλλογική, δηλαδή πρέπει να κληθεί μια φορά σε κάθε κόμβο για να ολοκληρωθεί. Αν δύο νήματα σε διαφορετικούς κόμβους ζητήσουν ταυτόχρονα δέσμευση μνήμης, δεν είναι βέβαιο ότι όλοι οι κόμβοι θα δεσμεύσουν πρώτα τη μνήμη του ενός και μετά του άλλου. Αυτό μπορεί να διορθωθεί εύκολα χρησιμοποιώντας μια κατανεμημένη κλειδαριά για να εξασφαλίσουμε την ατομικότητα, αλλά θα εισάγει επιπρόσθετες καθυστερήσεις. Μιας και αναμένουμε αυτές οι κλήσεις να γίνονται στην αρχή του προγράμματος (που ο κώδικας εκτελείται σειριακά), πριν ξεκινήσουν οι παράλληλοι υπολογισμοί, αποφασίσαμε να αφήσουμε την κατάσταση ως έχει. Επομένως, ο τελικός χρήστης, αν απαιτείται, θα πρέπει να φροντίσει για τον σχετικό αμοιβαίο αποκλεισμό μέσα στην εφαρμογή του, στις περιπτώσεις που οι αιτήσεις για δέσμευση μνήμης γίνονται ταυτόχρονα.

5.3 Αρχικοποίηση

Σε αυτή την ενότητα συζητούμε τη διαδικασία της αρχικοποίησης, δηλαδή τα βήματα που απαιτούνται προκειμένου να ξεκινήσει να εκτελείται σωστά το πρόγραμμα του χρήστη.

5.3.1 Προβλήματα με τις αρχικοποιήσεις των βιβλιοθηκών

Για να μπορέσει να ξεκινήσει το πρόγραμμά του σε διαφορετικούς κόμβους, ο χρήστης χρησιμοποιεί την εντολή `mpirun`. Αυτή φροντίζει να αντιγράψει το εκτελέσιμο στους κόμβους που καθόρισε ο χρήστης και στη συνέχεια να το ξεκινήσει. Συνεπώς, όλοι οι κόμβοι καταλήγουν να τρέχουν το ίδιο εκτελέσιμο αρχείο.

Όπως στην αρχική σχεδίαση [15, 16], έτσι και εδώ, στόχος μας είναι να έχουμε όσο το δυνατόν περισσότερα νήματα workers, δηλαδή νήματα που εκτελούν τις παράλληλες περιοχές του προγράμματος του χρήστη, και ένα νήμα server, που επικοινωνεί με τους υπόλοιπους κόμβους και δίνει εντολές στα νήματα workers του τοπικού κόμβου. Για να έχουμε πολλαπλά νήματα workers σε κάθε κόμβο, όμως,

οι βιβλιοθήκες ArgoDSM και TORC θα πρέπει να υποστηρίζουν να καλούμε τις συναρτήσεις τους από πολλαπλά νήματα ταυτόχρονα. Η ArgoDSM δεν έχει πρόβλημα σε αυτόν τον τομέα: αρκεί να την αρχικοποιήσει ένα νήμα σε κάθε κόμβο και μετά μπορεί να κληθεί από οποιοδήποτε νήμα.

Από την άλλη πλευρά, η κατάσταση με την TORC είναι πιο πολύπλοκη. Θα πρέπει και αυτή να αρχικοποιηθεί από ένα νήμα σε κάθε κόμβο, αλλά μόνο το νήμα που την αρχικοποίησε μπορεί να καλέσει τις συναρτήσεις της. Για τη δημιουργία ενός task αυτό δεν είναι πρόβλημα: μπορεί το νήμα server να την αρχικοποιήσει και κάθε φορά που θέλουμε να δημιουργήσουμε task να προωθούμε την αίτηση στο νήμα server. Όμως, ακριβώς το ίδιο θα πρέπει να συμβαίνει και με την εκτέλεση των tasks (τα tasks εκτελούνται καλώντας την συνάρτηση `torc_waitall`): μόνο το νήμα server τα εκτελεί όλα, ενώ τα υπόλοιπα νήματα workers κάθονται και περιμένουν. Φυσικά, κάτι τέτοιο δεν είναι καθόλου αποδοτικό.

Ωστόσο, η TORC μας παρέχει μια άλλη δυνατότητα: κατά την εκτέλεση του προγράμματος με το `mpirun`, ο χρήστης μπορεί να καθορίσει πόσοι TORC workers θα δημιουργηθούν σε κάθε κόμβο, θέτοντας τη μεταβλητή περιβάλλοντος (environmental variable) `TORC_WORKERS`. Οι TORC workers είναι νήματα που δημιουργεί εσωτερικά η TORC κατά την αρχικοποίησή της και τα τερματίζει κατά τον τερματισμό της. Σε όλη τη διάρκεια της εκτέλεσης του προγράμματος, περιμένουν μέχρι να δημιουργηθούν tasks και τα εκτελούν. Τα tasks που εκτελούν μπορούν με τη σειρά τους να παράγουν καινούργια tasks, ή να περιμένουν έως ότου εκτελεστούν όλα τα tasks που έχουν δημιουργηθεί ως αυτό το σημείο πριν συνεχίσουν. Η σχεδίασή μας θα πρέπει να λαμβάνει υποχρεωτικά υπόψιν αυτή την ιδιαιτερότητα της TORC.

5.3.2 Αρχικοποίηση των νημάτων

Ο χρήστης ξεκινάει το πρόγραμμά του ορίζοντας τη μεταβλητή περιβάλλοντος `TORC_WORKERS` και χρησιμοποιώντας το πρόγραμμα `mpirun`. Κατά τη διαδικασία της μετάφρασης, ο OMPi έχει φροντίσει να μετονομάσει τη συνάρτηση `main` του προγράμματος του χρήστη σε `__original_main`, επομένως το πρόγραμμα ξεκινάει να εκτελείται από τη `main` που έχουμε ορίσει εμείς.

Το πρώτο βήμα που κάνουμε είναι να αρχικοποιήσουμε τις βιβλιοθήκες που χρησιμοποιούμε: πρώτα την TORC και μετά την ArgoDSM. Ο λόγος για αυτή τη σειρά είναι ότι η TORC φροντίζει να αρχικοποιήσει και το MPI, που το χρειάζεται

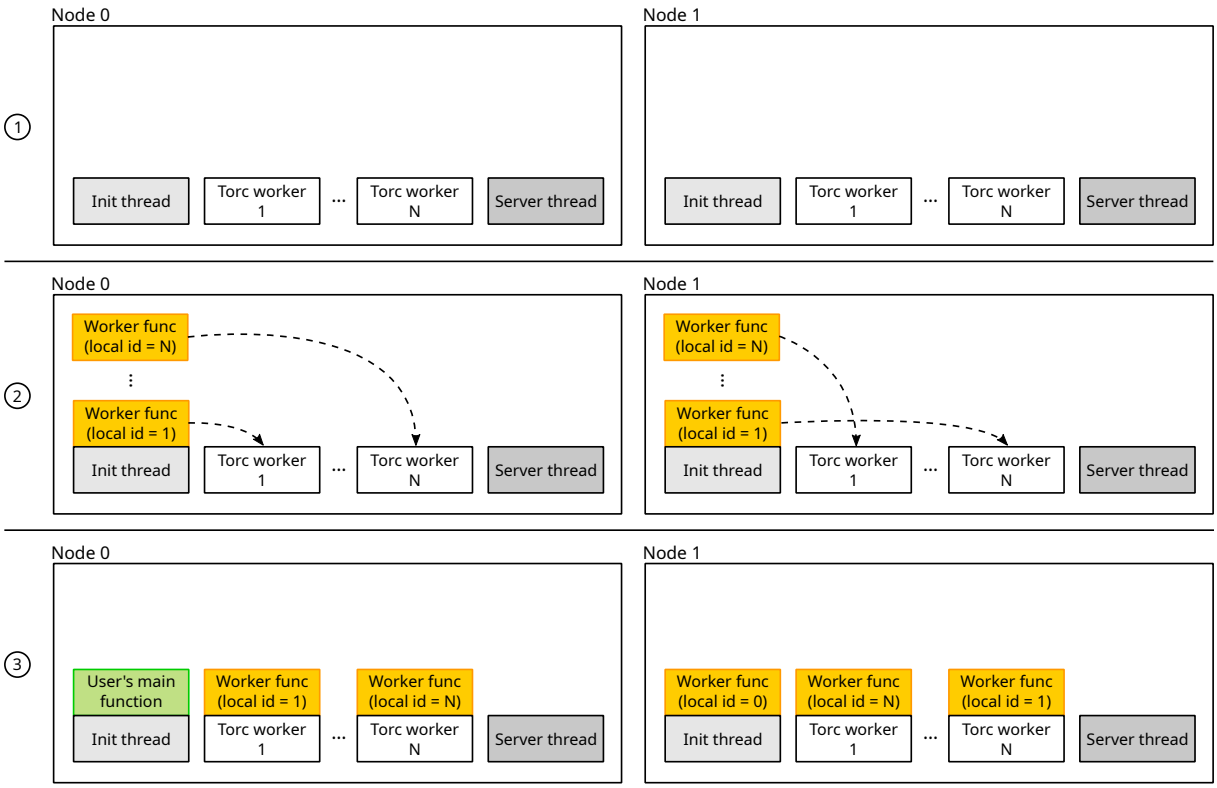
η TORC, η ArgoDSM και η δική μας υλοποίηση. Επίσης, δεσμεύουμε χώρο στην κοινόχρηστη μνήμη για τη στοίβα του αρχικού νήματος και εγκαθιδρυθούμε την εναλλακτική στοίβα για την εκτέλεση του χειριστή σήματος του segmentation fault, για τους λόγους που αναλύσαμε στην Ενότητα 5.2.2.

Συνεχίζουμε αρχικοποιώντας τη δική μας βιβλιοθήκη. Αρχικά, πρέπει να υπολογίσουμε πόσα νήματα workers έχουμε αθροιστικά. Όλοι οι κόμβοι στέλνουν στον αρχικό το πλήθος των TORC workers νημάτων που έχουν. Ο αρχικός κόμβος εξασφαλίζει ότι αυτό το άθροισμα ικανοποιεί τυχόν περιορισμούς που έχουν τεθεί από το χρήστη (η μεταβλητή περιβάλλοντος `OMP_THREAD_LIMIT` ορίζει το μέγιστο αριθμό νημάτων που μπορούν να χρησιμοποιηθούν) και ενημερώνει τους υπόλοιπους, ώστε κάθε κόμβος να γνωρίζει πόσους TORC workers πρέπει να χρησιμοποιήσει τελικά.

Ακολούθως, κάθε κόμβος δημιουργεί ένα νήμα server (βήμα (1) στο Σχήμα 5.4). Αυτό αναλαμβάνει να δέχεται και να μεταβιβάζει αιτήματα στους υπόλοιπους κόμβους. Για παράδειγμα, κατά την δημιουργία μιας καινούργιας παράλληλης περιοχής, το νήμα server του αρχικού κόμβου ενημερώνει όλα τα νήματα server των υπόλοιπων κόμβων να εκτελέσουν τη ζητούμενη παράλληλη περιοχή. Επιπλέον, προωθεί αιτήματα στα τοπικά νήματα TORC workers. Για παράδειγμα, μόλις λάβει αίτημα από έναν απομακρυσμένο κόμβο για την εκτέλεση μιας παράλληλης περιοχής, βάζει τα τοπικά νήματα TORC workers να εκτελέσουν την παράλληλη περιοχή.

Όμως, υπάρχει ένα πρόβλημα: οι TORC workers περιμένουν να δημιουργηθούν tasks μέσω της TORC για να τα εκτελέσουν, δεν περιμένουν για οδηγίες που προέρχονται από το δικό μας νήμα server. Για να το αντιμετωπίσουμε, κάθε κόμβος δημιουργεί ειδικά tasks, τα οποία για να τα ξεχωρίζουμε από τα tasks που δημιουργεί το πρόγραμμα του χρήστη, τα ονομάζουμε *task εκκίνησης*. Δημιουργούνται τόσα task εκκίνησης όσα και τα TORC worker νήματα που έχει ο κόμβος (βήμα (2) στο Σχήμα 5.4). Στην ουσία, το αρχικό task είναι η συνάρτηση που πρέπει να εκτελούν οι workers σε όλη τη διάρκεια εκτέλεσης του προγράμματος, η οποία φροντίζει να διεκπεραιώνει τις εντολές που δέχεται από το νήμα server, εκφρασμένη ως task. Ωστόσο, αυτά τα tasks εκκίνησης κρύβουν δύο κινδύνους.

Πρώτον, θα πρέπει να περιμένουμε μέχρι να βεβαιωθούμε ότι όλοι οι TORC workers ξεκίνησαν να εκτελούν το task εκκίνησής τους. Στην αντίθετη περίπτωση μπορεί να συμβεί το εξής σενάριο: Δημιουργούμε το task εκκίνησης και συνεχίζουμε εκτελώντας τον κώδικα του χρήστη, ο οποίος παράγει και αυτός ένα task. Τώρα ο TORC worker βλέπει δύο tasks που εκκρεμούν. Όμως το αρχικό task είναι υπεύθυνο



Σχήμα 5.4: Διαδικασία αρχικοποίησης. Βήμα (1): Το αρχικό νήμα έχει καλέσει την αρχικοποίηση της TORC, που δημιούργησε τα νήματα TORC workers (το πλήθος τους καθορίζεται από τη μεταβλητή περιβάλλοντος `TORC_WORKERS`), ενώ έχουμε δημιουργήσει και ένα νήμα server. Βήμα (2): Το αρχικό νήμα σε κάθε κόμβο δημιουργεί τόσα αρχικά tasks, όσοι και οι TORC workers, οι οποίοι τα εκτελούν (δε μας νοιάζει ποιος worker αναλαμβάνει ποιο task). Βήμα (3): Το αρχικό νήμα στον αρχικό κόμβο εκτελεί τον κώδικα του χρήστη. Στους υπόλοιπους κόμβους, γίνεται και αυτό worker, καλώντας την κατάλληλη συνάρτηση.

για σημαντικές αρχικοποιήσεις. Έτσι, αν αποφασίσει να εκτελέσει πρώτα το task του χρήστη, το πρόγραμμα θα καταρρεύσει, αφού δεν θα έχουν γίνει οι απαραίτητες αρχικοποιήσεις.

Για να το λύσουμε αυτό χρησιμοποιούμε δύο barriers, έναν τοπικό και έναν κατανεμημένο. Ο τοπικός barrier, που μας παρέχει η βιβλιοθήκη pthreads, εξασφαλίζει ότι τα νήματα TORC workers του τοπικού κόμβου έχουν αρχίσει να εκτελούν το task εκκίνησης πριν συνεχίσουμε την αρχικοποίηση. Ο κατανεμημένος barrier, που μας παρέχει η βιβλιοθήκη MPI, εξασφαλίζει ότι όλοι οι κόμβοι έχουν περάσει τον τοπικό barrier, δηλαδή ότι όλα τα νήματα TORC workers σε όλους τους κόμβους έχουν αρχίσει να εκτελούν το task εκκίνησης, πριν συνεχίσουμε την αρχικοποίηση.

Το δεύτερο πρόβλημα είναι λίγο πιο "πονηρό" και δύσκολο να γίνει αντιληπτό. Το task εκκίνησης δέχεται ως παράμετρο το τοπικό αναγνωριστικό (local id) του νήματος worker. Επίσης, κάθε νήμα worker αποθηκεύει στην ιδιωτική του μνήμη (thread private storage) το μπλοκ ελέγχου του (control block). Για κάποιες λειτουργίες, όπως είναι για παράδειγμα ο barrier, πρέπει να συνδυαστούν πληροφορίες και από τις δύο πηγές. Ας σκεφτούμε τώρα την εξής περίπτωση: δύο νήματα workers εκτελούν το task εκκίνησής τους. Στη συνέχεια, εκτελούν τα tasks που έχει παράγει το πρόγραμμα του χρήστη. Όταν τελειώσουν με αυτά, ποιος μας εξασφαλίζει ότι θα συνεχίσουν με το task εκκίνησης που είχαν πριν, και δεν θα πάρει το ένα νήμα το task εκκίνησης του άλλου; Αυτό μπορεί να συμβεί διότι η TORC επιτρέπει στους workers να κλέβουν μη ολοκληρωμένα tasks μεταξύ τους και να συνεχίζει κάποιο νήμα την εκτέλεση ενός task που το είχε ξεκινήσει ένα άλλο νήμα. Αν συμβεί αυτό, τότε το τοπικό αναγνωριστικό των νημάτων workers (που είναι τοπική μεταβλητή στη συνάρτηση του task εκκίνησης) δεν θα σχετίζεται με το μπλοκ ελέγχου τους (που είναι συσχετισμένο με το εκάστοτε νήμα).

Με άλλα λόγια, θέλουμε το νήμα που ξεκινάει να εκτελεί ένα task εκκίνησης, να το εκτελεί μέχρι το τέλος του προγράμματος, χωρίς να το ανταλλάξει για το αρχικό task κάποιου άλλου νήματος. Δηλαδή, θέλουμε το task εκκίνησης να είναι δεμένο (tied) με το νήμα που το ξεκινάει. Η έκδοση της TORC που είχαμε δεν υποστήριζε tied tasks, επομένως, για να το πετύχουμε αυτό, χρειάστηκε η ακόλουθη αλλαγή τον πηγαίο κώδικά της: δημιουργήσαμε μια καινούργια ιδιωτική ουρά για κάθε worker, στην οποία εισάγουμε τα αρχικά tasks. Όταν ένας worker θέλει να εκτελέσει ένα task, πρώτα ψάχνει σε αυτή την ουρά. Ο υπόλοιπος κώδικας της TORC δεν γνωρίζει για την ύπαρξη της καινούργιας ουράς, άρα κατά τη διαδικασία του κλεψίματος (stealing) κανένας worker δεν θα πάρει task που βρίσκεται στην ιδιωτική ουρά ενός άλλου worker.

5.3.3 Συνέχεια της αρχικοποίησης

Μετά την αρχικοποίηση των νημάτων, η διαδικασία συνεχίζει ως εξής. Το αρχικό νήμα στον πρώτο κόμβο θα πρέπει να εκτελέσει τον κώδικα του χρήστη, όπως φαίνεται στο βήμα (3) του Σχήματος 5.4. Δημιουργούμε ένα νήμα επιπέδου χρήστη του οποίου η στοίβα έχει δεσμευτεί στην κοινόχρηστη μνήμη και το βάζουμε να καλέσει τη συνάρτηση `__original_main`. Το αρχικό νήμα στους υπόλοιπους κόμβους,

ωστόσο, θα πρέπει να γίνει και αυτό νήμα worker, επομένως καλεί τη συνάρτηση που εκτελούν και τα υπόλοιπα νήματα workers (ως task εκκίνησης). Επειδή το αρχικό νήμα ήταν αυτό που έκανε την αρχικοποίηση της TORC, μπορεί να καλεί κανονικά τις συναρτήσεις της, παρόλο που δεν είναι νήμα τύπου TORC worker.

Συνοψίζοντας, μετά την αρχικοποίηση, έχουμε τους ακόλουθους τρεις τύπους νημάτων:

- **Νήμα server** Ένα νήμα σε κάθε κόμβο αναλαμβάνει το ρόλο του server. Δουλειά του είναι να λαμβάνει και να προωθεί μηνύματα στους υπόλοιπους κόμβους, καθώς και να δίνει εντολές στα νήματα workers του τοπικού κόμβου, όπως είναι για παράδειγμα η εκτέλεση μιας παράλληλης περιοχής.

- **Νήμα worker** Είναι τα νήματα που δημιουργεί η TORC κατά την αρχικοποίησή της (TORC workers) και το πλήθος τους καθορίζεται από τη μεταβλητή περιβάλλοντος `TORC_WORKERS`, καθώς και το αρχικό νήμα σε όλους τους κόμβους πλην του πρώτου. Δέχονται εντολές από το νήμα server του κόμβου, για παράδειγμα την εκτέλεση μιας παράλληλης περιοχής, και τις διεκπεραιώνουν.

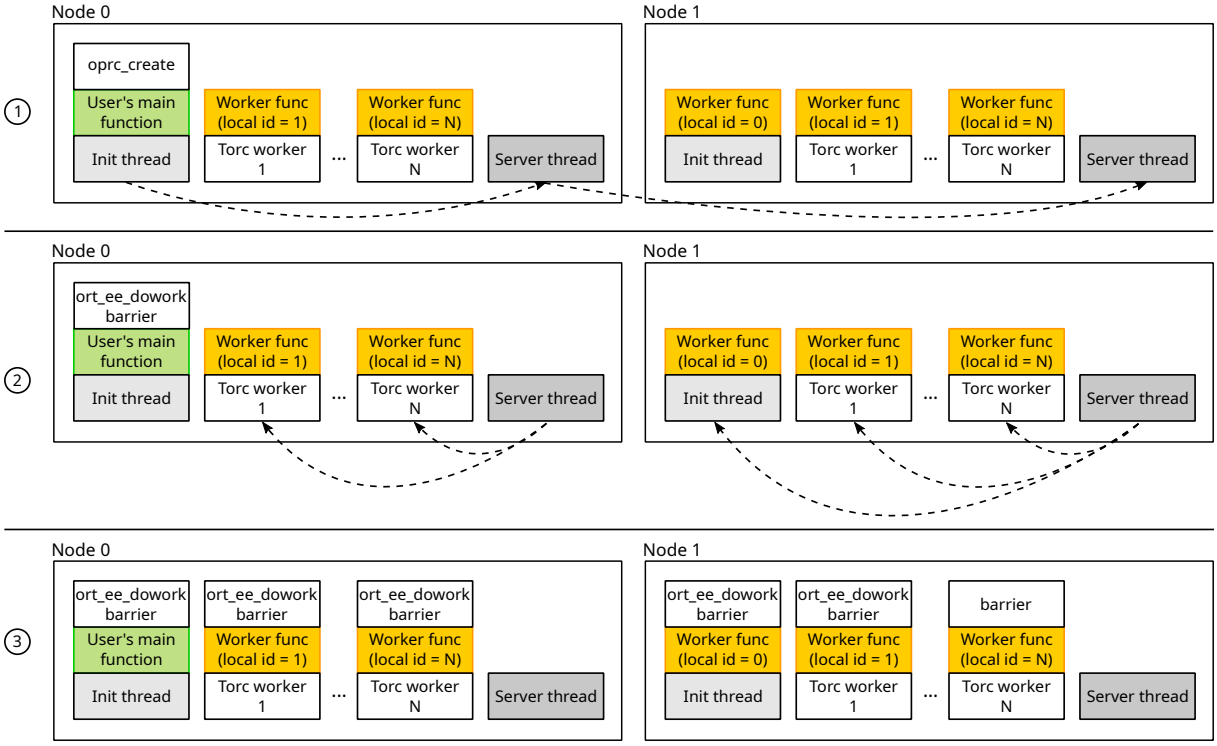
- **Νήμα χρήστη** Είναι το αρχικό νήμα στον αρχικό κόμβο. Εκτελεί τον κώδικα του χρήστη σειριακά, μέχρι να βρει παράλληλη περιοχή.

5.4 Εκτέλεση παράλληλης περιοχής

Σε αυτήν την ενότητα αναλύουμε την διαδικασία εκτέλεσης μιας παράλληλης περιοχής, η οποία παρουσιάζεται γραφικά στο Σχήμα 5.5.

5.4.1 Η διαδικασία εκτέλεσης παράλληλης περιοχής

Το αρχικό νήμα στον αρχικό κόμβο εκτελεί τον κώδικα του χρήστη σειριακά, μέχρι να βρει μια παράλληλη περιοχή. Τότε, γίνεται εναλλαγή του νήματος επιπέδου χρήστη, ώστε να χρησιμοποιεί στοίβα που βρίσκεται στην ιδιωτική μνήμη (για τον λόγο που εξηγήσαμε στην Ενότητα 5.2.2) και στη συνέχεια καλεί τη συνάρτηση `ort_execute_parallel`. Ξεκινώντας, βεβαιώνουμε ότι μπορούμε να παρέχουμε το πλήθος των νημάτων που ζήτησε ο χρήστης (ο χρήστης μπορεί να καθορίσει ρητά το πλήθος των νημάτων που συμμετέχουν σε μια παράλληλη περιοχή, για παρά-



Σχήμα 5.5: Εκτέλεση παράλληλης περιοχής. Βήμα (1): μόλις το αρχικό νήμα που εκτελεί τον κώδικα σειριακά βρει παράλληλη περιοχή, ενημερώνει το νήμα server του και συνεχίζει με την εκτέλεσή της. Το νήμα server του πρώτου κόμβου φροντίσει να ενημερώσει όλα τα υπόλοιπα νήματα server. Βήμα (2): Το νήμα server κάθε κόμβου ενημερώνει όλα τα τοπικά νήματα workers. Από αυτή τη διαδικασία εξαιρείται το πρώτο νήμα του πρώτου κόμβου. Βήμα (3): Τα νήματα workers εκτελούν την παράλληλη περιοχή και συγχρονίζονται στο τέλος της. Τα νήματα που δεν συμμετέχουν στην παράλληλη περιοχή, όπως το n-οστό νήμα worker του δεύτερου κόμβου εδώ, συμμετέχουν μόνο στο συγχρονισμό.

δειγμα μέσω της συνάρτησης `omp_set_num_threads`). Επίσης, το αρχικό νήμα τροποποιεί το μπλοκ ελέγχου του (control block), ώστε να ενημερώσει πληροφορίες, όπως η ταυτότητα (identity) της παράλληλης περιοχής που πρόκειται να εκτελεστεί.

Στη συνέχεια, όπως φαίνεται στο βήμα (1) του Σχήματος 5.5, το αρχικό νήμα καλεί τη συνάρτηση `oprc_create`, η οποία ενημερώνει το νήμα server του πρώτου κόμβου για την παράλληλη περιοχή και ακολούθως αρχίζει να εκτελεί την παράλληλη περιοχή. Το νήμα server στον αρχικό κόμβο στέλνει ένα μήνυμα τύπου `PARALLEL` στα νήματα server των υπόλοιπων κόμβων. Το μήνυμα περιλαμβάνει το πλήθος των νημάτων που πρέπει να εκτελέσουν την παράλληλη περιοχή, αλλά και ένα δείκτη

στο μπλοκ ελέγχου του αρχικού νήματος, που μεταξύ άλλων περιέχει την ταυτότητα της παράλληλης περιοχής που πρέπει να εκτελεστεί. Για να μπορούν να έχουν όλοι οι κόμβοι πρόσβαση στο μπλοκ ελέγχου του αρχικού νήματος, αυτό θα πρέπει να είναι αποθηκευμένο στην κοινόχρηστη μνήμη. Φροντίζουμε για αυτό κατά τη διαδικασία της αρχικοποίησης των νημάτων.

Μόλις τα νήματα server λάβουν το μήνυμα `PARALLEL`, θα πρέπει να ενημερώσουν τα νήματα workers (που άλλωστε κάνουν όλη τη δουλειά) να εκτελέσουν τη ζητούμενη παράλληλη περιοχή (βήμα (2) του Σχήματος 5.5). Από αυτή τη διαδικασία εξαιρείται φυσικά το αρχικό νήμα στον αρχικό κόμβο, που φροντίζει να ξεκινήσει μόνο του την εκτέλεση της παράλληλης περιοχής. Επιπλέον, αν ο χρήστης έχει καθορίσει ρητά το πλήθος των νημάτων που θα εκτελέσουν την παράλληλη περιοχή και αυτό είναι μικρότερο από το συνολικό πλήθος νημάτων workers που διαθέτουμε, τα νήματα που περισσεύουν θα πρέπει απλά να περιμένουν για συγχρονισμό στο τέλος της. Στο Σχήμα 5.5, αυτό συμβαίνει με το n-οστό νήμα worker του δεύτερου κόμβου.

Τα υπόλοιπα νήματα workers καλούν τη συνάρτηση `ort_ee_work`, όπως φαίνεται στο βήμα (3) του Σχήματος 5.5. Εκεί, όλα τα νήματα δεσμεύουν χώρο στην ιδιωτική τους μνήμη για το μπλοκ ελέγχου τους, σε περίπτωση που δεν έχουν ήδη ένα. Επιπλέον, από το μπλοκ ελέγχου του αρχικού νήματος, αντιγράφουν στο δικό τους πληροφορίες που χρειάζονται, όπως είναι ο συνολικός αριθμός των νημάτων που συμμετέχουν στην παράλληλη περιοχή και η ταυτότητά της. Στη συνέχεια, εκτελούν τη ζητούμενη παράλληλη περιοχή.

Το αρχικό νήμα στον αρχικό κόμβο καθώς και όλα τα νήματα workers, είτε συμμετείχαν είτε όχι στην παράλληλη περιοχή, συγχρονίζονται στο τέλος της. Αυτό επιτυγχάνεται με δύο barriers, έναν τοπικό και έναν κατανεμημένο. Πρώτα όλα τα νήματα κάθε κόμβου συγχρονίζονται τοπικά (χρησιμοποιώντας τον τοπικό barrier) και στη συνέχεια το πρώτο νήμα κάθε κόμβου καλεί τον κατανεμημένο barrier για να επιτευχθεί ολικός συγχρονισμός. Αυτή η διαδικασία είναι απαραίτητη για να εξασφαλίσουμε ότι μόλις το αρχικό νήμα στον αρχικό κόμβο συνεχίσει να εκτελεί τον κώδικα του χρήστη μετά την παράλληλη περιοχή, όλα τα νήματα σε όλους τους κόμβους θα έχουν τελειώσει. Τέλος, εναλλάσσουμε το πρώτο νήμα στον πρώτο κόμβο, ώστε να συνεχίσει τη σειριακή εκτέλεση του προγράμματος του χρήστη χρησιμοποιώντας την στοίβα που έχουμε δεσμεύσει στην κοινόχρηστη μνήμη.

Σε αυτό το σημείο αξίζει να σημειώσουμε κάτι ακόμη. Τα νήματα workers σε

όλους τους κόμβους, όταν δεν εκτελούν μια παράλληλη περιοχή, περιμένουν για εντολές από το τοπικό νήμα server. Αυτό το επιτυγχάνουμε χρησιμοποιώντας μια μεταβλητή συνθήκης (condition variable) για κάθε νήμα worker, οι οποίες παρέχονται από τη βιβλιοθήκη pthreads. Οι μεταβλητές συνθήκης μας εξασφαλίζουν ότι τα νήματα που περιμένουν σε αυτές, δεν θα δρομολογηθούν για εκτέλεση από το λειτουργικό σύστημα και επομένως δεν θα σπαταλούν κύκλους του επεξεργαστή όσο περιμένουν. Αυτό αποτελεί αλλαγή σε σχέση με την προηγούμενη υλοποίηση, η οποία όσο ένα νήμα περίμενε, ζητούσε από το δρομολογητή (scheduler) του λειτουργικού συστήματος την παραχώρηση της προτεραιότητάς του (yield), ξοδεύοντας κύκλους του επεξεργαστή. Γενικά, στις πρόσφατες εκδόσεις του Linux δε συνίσταται η κλήση συναρτήσεων που δίνουν συμβουλές στον δρομολογητή, όπως η yield.

5.4.2 Προσδιορισμός παράλληλων περιοχών

Κατά τη διάρκεια της μετάφρασης του προγράμματος του χρήστη, ο OMPi εξάγει τον κώδικα κάθε παράλληλης περιοχής σε μια ξεχωριστή συνάρτηση, με όνομα `_thrFunc0_`, `_thrFunc1_`, κ.ο.κ. Για να εκτελέσει μια παράλληλη περιοχή ένα νήμα worker θα πρέπει απλά να καλέσει την κατάλληλη συνάρτηση. Πώς όμως το αρχικό νήμα θα ενημερώσει τα υπόλοιπα νήματα workers για το *ποια* συνάρτηση θα πρέπει να εκτελέσουν;

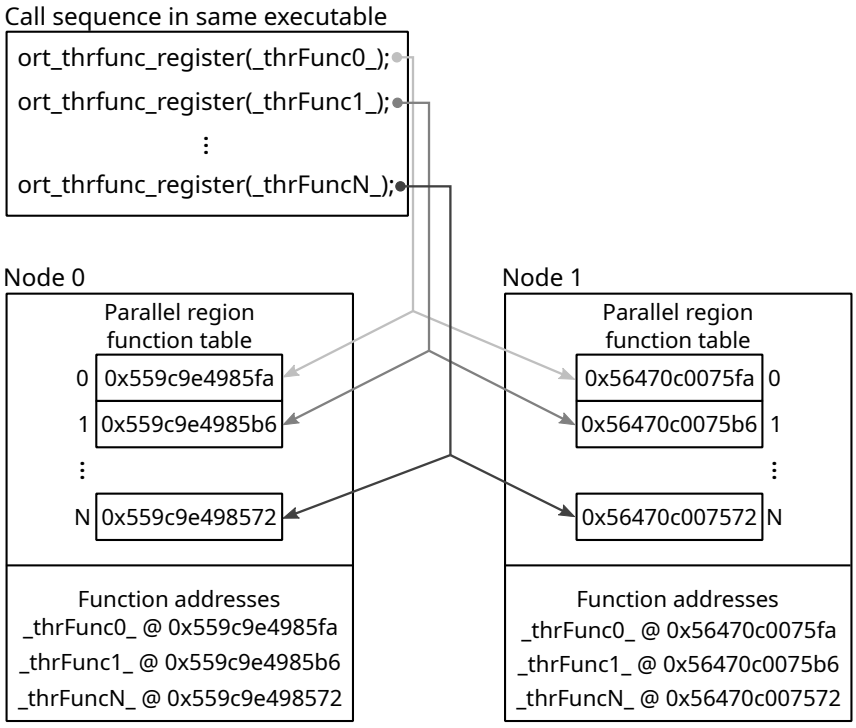
Η απλή λύση θα ήταν να αποθηκεύει στο μπλοκ ελέγχου του έναν δείκτη στην κατάλληλη συνάρτηση. Ωστόσο, κάποια νήματα εκτελούνται σε διαφορετικούς κόμβους και συνεπώς σε διαφορετικές διεργασίες. Ακόμη και αν οι κόμβοι έχουν ακριβώς τα ίδια τεχνικά χαρακτηριστικά, ορισμένες τεχνολογίες που βρίσκονται στα σύγχρονα λειτουργικά συστήματα, όπως η τυχαιοποίηση της διάταξης του χώρου διευθύνσεων (address space layout randomization – ASLR), θα έχουν ως αποτέλεσμα η ίδια συνάρτηση να έχει διαφορετική διεύθυνση όταν εκτελείται σε διαφορετικές διεργασίες.

Επομένως, χρειαζόμαστε έναν άλλο τρόπο για να προσδιορίσουμε τις συναρτήσεις παράλληλων περιοχών. Η λύση μας είναι να αντιστοιχίζουμε κάθε συνάρτηση σε έναν διαδοχικό ακέραιο, ξεκινώντας από το μηδέν. Κατά τη διαδικασία της μετάφρασης του προγράμματος του χρήστη, φροντίζουμε ώστε ο OMPi να προσθέτει μια κλήση στη συνάρτηση `ort_thrfunc_register`, κάθε φορά που συναντά μια παράλληλη περιοχή, με όρισμα τη συνάρτηση παράλληλης περιοχής, για παράδειγμα `ort_thrfunc_register(_thrFunc0_)`. Αυτές οι κλήσεις θα γίνουν σε κάθε κόμβο στην

αρχή της εκτέλεσης του προγράμματος, πριν κληθεί η συνάρτηση `main` του χρήστη.

Κάθε κόμβος αποθηκεύει ένα πίνακα με δείκτες σε συναρτήσεις παράλληλων περιοχών, το μέγεθος του οποίου αυξάνεται δυναμικά. Κάθε φορά που καλείται η `ort_thrfunc_register`, το όρισμά της προστίθεται στο τέλος του πίνακα. Στο τέλος αυτής της διαδικασίας, ο πίνακας θα έχει τις διευθύνσεις όλων των συναρτήσεων παράλληλων περιοχών. Επειδή, όμως, όλοι οι κόμβοι τρέχουν το ίδιο εκτελέσιμο αρχείο, η σειρά των κλήσεων της `ort_thrfunc_register` είναι ίδια σε κάθε κόμβο, επομένως οι συναρτήσεις αποθηκεύονται με την ίδια σειρά στον πίνακα κάθε κόμβου.

Έτσι, το αρχικό νήμα αποθηκεύει στο μπλοκ ελέγχου του τον δείκτη (index) του πίνακα που βρίσκεται η ζητούμενη συνάρτηση παράλληλης περιοχής. Τα νήματα workers, καλούν την συνάρτηση παράλληλης περιοχής που βρίσκεται στην θέση που καθορίζεται από τον δείκτη (index) του δικού τους πίνακα. Η διαδικασία αυτή φαίνεται γραφικά στο Σχήμα 5.6.



Σχήμα 5.6: Διαδικασία για τον προσδιορισμό των παράλληλων περιοχών. Επειδή η σειρά των κλήσεων της `ort_thrfunc_register` είναι ίδια σε κάθε κόμβο (αφού όλοι οι κόμβοι τρέχουν το ίδιο εκτελέσιμο), οι πίνακες των κόμβων περιέχουν τις συναρτήσεις παράλληλων περιοχών με την ίδια σειρά. Προσδιορίζουμε κάθε συνάρτηση χρησιμοποιώντας τη θέση της (index) στον πίνακα.

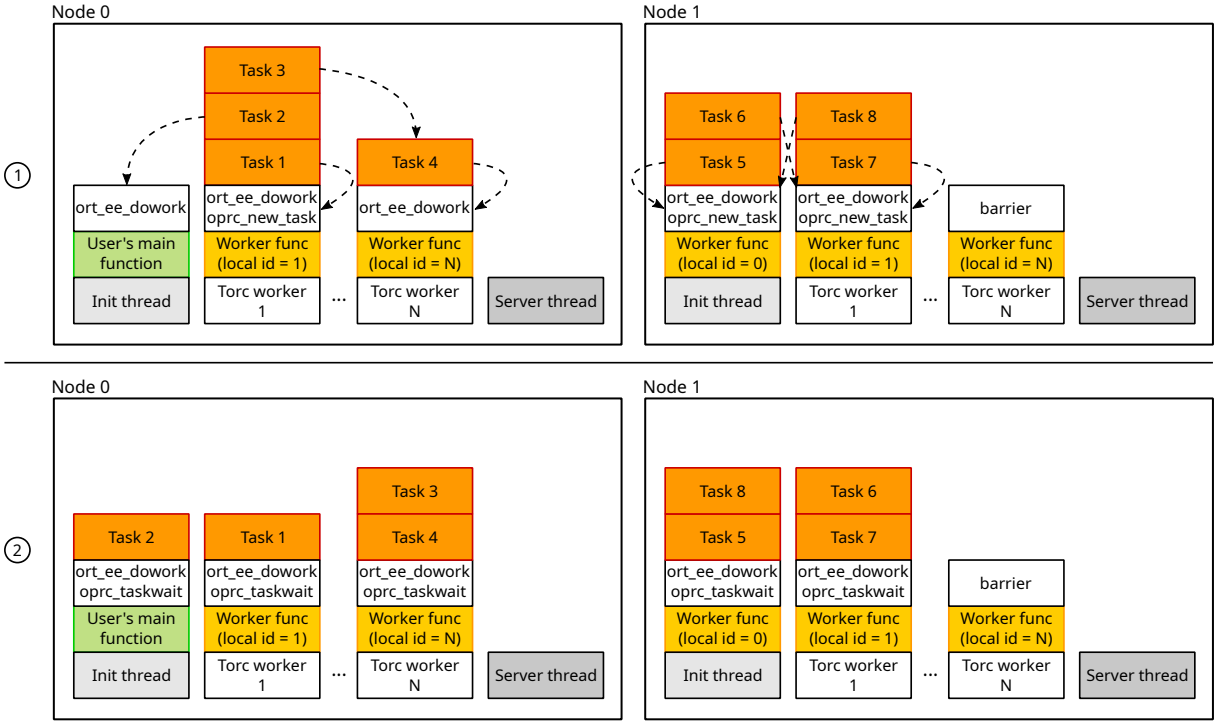
5.4.3 Κοινόχρηστες μεταβλητές εντός της παράλληλης περιοχής

Κάποιες μεταβλητές θα πρέπει να είναι κοινόχρηστες εντός της παράλληλης περιοχής, είτε επειδή έτσι τις όρισε ρητά ο χρήστης, είτε επειδή αυτό συνεπάγεται από τους κανόνες του OpenMP: μεταβλητές που ορίζονται πριν από μια παράλληλη περιοχή είναι κοινόχρηστες μέσα σε αυτή. Όλες αυτές οι μεταβλητές έχουν δηλωθεί πριν την παράλληλη περιοχή, δηλαδή στο σειριακό μέρος του προγράμματος, το οποίο εκτελείται εξ ολοκλήρου από το αρχικό νήμα του αρχικού κόμβου. Η μνήμη για τη στοίβα αυτού του νήματος είναι δεσμευμένη στον κοινόχρηστο χώρο, επομένως όλες αυτές οι μεταβλητές βρίσκονται στην κοινόχρηστη μνήμη και συνεπώς όλα τα νήματα σε όλους του κόμβους μπορούν να τις προσπελάσουν και να τις τροποποιήσουν.

Επιπλέον, κατά τη μετάφραση του προγράμματος του χρήστη ο OMPi φροντίζει να αντικαταστήσει όλες τις αναφορές που γίνονται εντός της παράλληλης περιοχής σε αυτές τις μεταβλητές με δείκτες. Η συνάρτηση της παράλληλης περιοχής δέχεται ως όρισμα μια δομή (struct) που περιέχει τις τιμές για αυτούς τους δείκτες. Πριν την έναρξη της παράλληλης περιοχής, το αρχικό νήμα στον αρχικό κόμβο δημιουργεί στην κοινόχρηστη μνήμη (τοπική μεταβλητή που αποθηκεύεται στην κοινόχρηστη στοίβα) αυτή τη δομή και την αρχικοποιεί ώστε οι δείκτες που περιέχει να δείχνουν στις αντίστοιχες μεταβλητές στη στοίβα του. Στην αρχή της παράλληλης περιοχής, όλα τα νήματα ενημερώνουν τους δείκτες τους με βάση τις τιμές που διαβάζουν από τη δομή (την οποία μπορούν να προσπελάσουν γιατί είναι αποθηκευμένη στην κοινόχρηστη μνήμη), δηλαδή κάνουν τους δείκτες τους να δείχνουν στην κοινόχρηστη στοίβα του αρχικού νήματος. Αναλυτικό παράδειγμα που εξηγεί λεπτομερώς αυτή η διαδικασία βρίσκεται στην Ενότητα 3.2.1.

5.5 Παραγωγή και εκτέλεση tasks

Στην παρούσα ενότητα αναλύουμε την διαδικασία της δημιουργίας και εκτέλεσης tasks μέσα σε μία παράλληλη περιοχή, η οποία φαίνεται στο Σχήμα 5.7.



Σχήμα 5.7: Παραγωγή και εκτέλεση tasks. Βήμα ①: τα νήματα που συμμετέχουν σε μια παράλληλη περιοχή μπορεί να παράγουν tasks, χωρίς να είναι υποχρεωτικό όλα τους να παράγουν tasks. Βήμα ②: Όλα τα νήματα που συμμετέχουν στην παράλληλη περιοχή πρέπει να εκτελέσουν κάποια από τα tasks. Για την ώρα, δεν μπορούν να κλέψουν tasks από άλλους κόμβους.

5.5.1 Διαδικασία δημιουργίας και εκτέλεσης tasks

Η βιβλιοθήκη TORC που χρησιμοποιούμε για τα tasks απαιτεί να της δηλώσουμε όλα τα tasks που ενδέχεται να δημιουργήσουμε πριν την αρχικοποίησή της. Επομένως, κατά τη μετάφραση του προγράμματος του χρήστη, ο OMPi εξάγει κάθε task που συναντά σε μια νέα συνάρτηση με όνομα `_taskFunc0_`, `_taskFunc1_` κ.ο.κ. (όμοια με τις παράλληλες περιοχές). Επίσης, τον τροποποιήσαμε ώστε να παράγει μια κλήση στη συνάρτηση `torc_register_task`, που παίρνει ως όρισμα μια συνάρτηση task και φροντίζει να καταχωρίσει αυτό το task στην TORC, για παράδειγμα `torc_register_task(_taskFunc0_)`. Αυτή η συνάρτηση θα κληθεί από όλους τους κόμβους στην αρχή της εκτέλεσης του προγράμματος. Επίσης, χρησιμοποιούμε αυτή τη συνάρτηση για να δηλώσουμε στην TORC το αρχικό task που τρέχουν τα νήματα workers σε όλη τη διάρκεια εκτέλεσης του προγράμματος και είναι υπεύθυνο να δέχεται εντολές από το τοπικό νήμα server του κόμβου.

Κατά την αρχικοποίηση της TORC, που γίνεται από ένα νήμα σε κάθε κόμβο,

πρέπει να καθορίσουμε το μοντέλο εκτέλεσης. Η TORC υποστηρίζει δύο μοντέλα εκτέλεσης: το master–worker (`MODE_MW`), όπου μόνο το νήμα στον αρχικό κόμβο συνεχίζει μετά την αρχικοποίηση της TORC, ενώ τα νήματα στους υπόλοιπους κόμβους μετατρέπονται σε TORC workers. Αντιθέτως, στο single process multiple data (`MODE_SPMD`) μοντέλο, τα νήματα όλων των κόμβων συνεχίζουν μετά την αρχικοποίηση της TORC. Φυσικά, εμείς χρησιμοποιούμε το δεύτερο, αφού όλοι οι κόμβοι πρέπει να κάνουν και άλλες αρχικοποιήσεις, πέραν της TORC.

Κατά τη δημιουργία ενός task, απλώς καλούμε τη συνάρτηση `torc_create`. Ανάλογα το πρόγραμμα του χρήστη, μπορεί όλα τα νήματα να παράγουν tasks, αλλά είναι αρκετά συνηθισμένη και η περίπτωση όπου όλα τα tasks παράγονται από ένα μόνο νήμα. Η εκτέλεση των tasks γίνεται με την οδηγία `#pragma omp taskwait`, η οποία για εμάς αντιστοιχεί σε κλήση της `torc_waitall`. Το νήμα που την καλεί ψάχνει και εκτελεί τα tasks που εκκρεμούν, ακόμη και αν το ίδιο δεν έχει παράξει κανένα task. Για αυτό τον λόγο, έχουμε ενεργοποιήσει την επιλογή τοπικού κλεψίματος (local stealing) που παρέχει η TORC, με αποτέλεσμα ένα νήμα να μπορεί να κλέψει και να εκτελέσει ένα task που έχει παραχθεί από ένα άλλο νήμα στον ίδιο κόμβο. Ιδανικά θα θέλαμε να ενεργοποιήσουμε και την δυνατότητα της TORC για απομακρυσμένο κλέψιμο (remote stealing), αλλά ένα πρόβλημα που αναλύουμε στην Ενότητα 5.5.3 για την ώρα μας αναγκάζει να απορρίψουμε αυτήν την επιλογή. Ένα νήμα αναλαμβάνει την εκτέλεση tasks και όταν περιμένει σε κάποιον barrier, διαδικασία που αναλύουμε στην Ενότητα 5.7.

5.5.2 Κοινόχρηστες μεταβλητές εντός των tasks

Τα tasks, όπως και οι παράλληλες περιοχές, μπορεί να έχουν κοινόχρηστες μεταβλητές, οι οποίες είτε προκύπτουν από ρητές οδηγίες του χρήστη, είτε από τους κανόνες του OpenMP. Η διαδικασία που χρησιμοποιεί ο OMPi για να χειριστεί τις κοινόχρηστες μεταβλητές των tasks είναι σχεδόν ίδια με τη διαδικασία που χρησιμοποιεί για τις κοινόχρηστες μεταβλητές των παράλληλων περιοχών, την οποία περιγράψαμε στην Ενότητα 5.4.3. Το μοναδικό σημείο στο οποίο διαφέρουν είναι ότι στην περίπτωση μιας παράλληλης περιοχής, η δομή με τους δείκτες είναι τοπική μεταβλητή του νήματος που τη δημιουργεί. Αντίθετα, στην περίπτωση ενός task, η μνήμη για τη δομή με τους δείκτες πρέπει να δεσμευτεί μέσω της συνάρτησης `ort_taskenv_alloc`, που καλείται από το νήμα που δημιουργεί το task και δέχεται ως παραμέτρους ένα

δείκτη στη συνάρτηση του task (που εμείς δεν χρησιμοποιούμε) και το απαιτούμενο μέγεθος.

Η δομή αυτή θα πρέπει να είναι προσπελάσιμη από τα νήματα όλων των κόμβων, μιας και δεν γνωρίζουμε ποιο νήμα θα αναλάβει να εκτελέσει τελικά το task. Μια λύση θα ήταν, λοιπόν, να την δεσμεύσουμε στην κοινόχρηστη μνήμη. Αυτό, ωστόσο, δεν είναι εφικτό με την τρέχουσα υλοποίησή μας. Η συνάρτηση δέσμευσης μνήμης στον κοινόχρηστο χώρο είναι συλλογική (δηλαδή πρέπει να κληθεί από ένα νήμα σε κάθε κόμβο) και δεν είναι ασφαλές να κληθεί από διαφορετικά νήματα ταυτόχρονα (thread safe). Δηλαδή, αν δύο νήματα θελήσουν να δημιουργήσουν από ένα task το καθένα και άρα να δεσμεύσουν μνήμη για τη δομή τους ταυτόχρονα, δεν είναι βέβαιο ότι όλοι οι κόμβοι θα δεσμεύσουν πρώτα τη μνήμη του ενός και μετά του άλλου. Θα μπορούσαμε να λύσουμε αυτό το πρόβλημα χρησιμοποιώντας μια κατανεμημένη κλειδαριά, αλλά αυτό θα εισήγαγε νέες καθυστερήσεις. Έτσι, ακολουθήσαμε μια διαφορετική προσέγγιση.

Δεσμεύουμε τη μνήμη για τη δομή στον ιδιωτικό χώρο (χρησιμοποιώντας τη `malloc`) και κατά τη δημιουργία του task, ενημερώνουμε την TORC ότι το task χρειάζεται μια παράμετρο, η οποία θα πρέπει να αντιγραφεί στον κόμβο που τελικά θα αναλάβει να το εκτελέσει. Για να το πετύχουμε αυτό, χρησιμοποιούμε την επιλογή `CALL_BY_COP` της TORC όταν δηλώνουμε την παράμετρο του task στη συνάρτηση `torc_create`. Η TORC εσωτερικά χρησιμοποιεί το MPI για να αντιγράφει δεδομένα μεταξύ των κόμβων. Αυτή η προσέγγιση έχει το ακόλουθο πρόβλημα: η TORC δέχεται ως παραμέτρους ενός task ακεραίους, floats, doubles ή πίνακες αυτών, αλλά όχι δομές.

Η εύκολη (και ταυτόχρονα μη μεταφέρσιμη) λύση για αυτό το πρόβλημα είναι καμουφλάρουμε τη δομή μας ως πίνακα ακεραίων. Όταν δεσμεύουμε μνήμη για τη δομή, δεσμεύουμε επιπλέον χώρο, αν χρειάζεται (padding), ώστε το τελικό μέγεθός της να είναι πολλαπλάσιο του μεγέθους ενός ακεραίου. Στη συνέχεια, όταν δημιουργούμε το task, ενημερώνουμε την TORC ότι το task χρειάζεται ως παράμετρο έναν πίνακα ακεραίων, τον οποίο θα πρέπει να αντιγράψει στον κόμβο που τελικά θα εκτελέσει το task. Δεν υπάρχει κάποιο πρόβλημα, πέρα από το αισθητικό, με αυτή την προσέγγιση: ο επιπλέον χώρος που ίσως χρειαστεί να δεσμεύσουμε είναι το πολύ 3 bytes, αν υποθέσουμε ότι το μέγεθος ενός ακεραίου είναι 4 bytes (που στην πλειοψηφία των συστημάτων ισχύει). Επίσης, βρίσκεται μετά το τέλος της δομής, άρα δεν πρόκειται να προκύψει πρόβλημα με τον κώδικα που προσπαθεί να

προσπελάσει τα στοιχεία της δομής.

5.5.3 Το πρόβλημα με τις κοινόχρηστες μεταβλητές

Η ύπαρξη κοινόχρηστων μεταβλητών σε ένα task, όμως, κρύβει ένα πρόβλημα που δεν είναι εύκολα αντιληπτό, στην περίπτωση που το task καταλήξει να εκτελεστεί σε έναν απομακρυσμένο κόμβο, λόγω κλεψίματος (stealing). Σε αντίθεση με τις ιδιωτικές μεταβλητές (*private* και *firstprivate*), που στη δομή του task αντιγράφουμε απλά την αρχική τιμή τους, στην περίπτωση των κοινόχρηστων μεταβλητών αποθηκεύουμε την διεύθυνσή τους.

Αν η μεταβλητή που θα χρησιμοποιηθεί ως κοινόχρηστη στο task έχει δηλωθεί εκτός της παράλληλης περιοχής, τότε δεν υπάρχει πρόβλημα. Σε αυτήν την περίπτωση είναι αποθηκευμένη στην στοίβα του πρώτου νήματος του πρώτου κόμβου, η οποία όπως έχουμε πει βρίσκεται ολόκληρη στην κοινόχρηστη μνήμη, επομένως θα μπορεί να προσπελαστεί από όλους τους κόμβους. Αν, όμως, η μεταβλητή είναι δηλωμένη εντός μιας παράλληλης περιοχής, τότε υπάρχει πρόβλημα. Οι στοίβες όλων των νημάτων βρίσκονται στην ιδιωτική μνήμη του εκάστοτε κόμβου, επομένως τα νήματα των υπόλοιπων κόμβων δεν μπορούν να προσπελάσουν μεταβλητές που είναι αποθηκευμένες εκεί. Όταν εκτελεί μια παράλληλη περιοχή, ακόμη και το αρχικό νήμα χρησιμοποιεί ιδιωτική στοίβα.

Ακόμη, υπάρχει και μια τρίτη περίπτωση. Αν ένα task δημιουργήσει ένα νέο task αναδρομικά, τότε οι μεταβλητές που είναι δηλωμένες εντός του πρώτου task θα αποθηκευτούν στην στοίβα του worker που εκτελεί το πρώτο task. Στην TORC κάθε task υλοποιείται ως ένα νήμα επιπέδου χρήστη (user level thread), επομένως οι μεταβλητές αυτές θα βρίσκονται σε μνήμη που διαχειρίζεται η TORC. Η TORC δεν γνωρίζει για την ύπαρξη κοινόχρηστης μνήμης sDSM και επομένως οι στοίβες όλων των νημάτων workers βρίσκονται σε ιδιωτική μνήμη, άρα δεν μπορούν να προσπελαστούν από νήματα άλλων κόμβων.

Ο προφανής τρόπος για να λυθεί αυτό το πρόβλημα θα ήταν να τοποθετήσουμε τις στοίβες *όλων* των νημάτων στην κοινόχρηστη μνήμη. Τότε, από οποιοδήποτε σημείο δημιουργήσουμε ένα task, οι μεταβλητές στις οποίες θα αναφερόμαστε ως κοινόχρηστες θα βρίσκονται στην κοινόχρηστη μνήμη και θα είναι προσπελάσιμες από όλους τους κόμβους. Το πρόβλημα με αυτή την προσέγγιση είναι ότι οι στοίβες που πρέπει να μεταφερθούν στην κοινόχρηστη μνήμη καταλήγουν να είναι πάρα

πολλές. Αν εκτελούμε ένα πρόγραμμα σε 10 κόμβους και κάθε κόμβος έχει 4 νήματα, τότε έχουμε 40 στοίβες. Αν δώσουμε σε κάθε στοίβα 8 MB χώρο (που είναι το προεπιλεγμένο μέγεθος σε σύγχρονα συστήματα Linux), τότε θα χρειαστούμε 320 MB επιπλέον χώρο στην κοινόχρηστη μνήμη.

Ο υπολογισμός της προηγούμενης παραγράφου λαμβάνει υπόψιν μόνο τις στοίβες των νημάτων και όχι τις στοίβες των tasks[1], που πρέπει να επίσης να βρίσκονται στην κοινόχρηστη μνήμη, για να καλύψουμε την περίπτωση της δημιουργίας αναδρομικών tasks με κοινόχρηστες μεταβλητές. Στην TORC τα tasks υλοποιούνται ως νήματα επιπέδου χρήστη, επομένως το καθένα έχει την δική του στοίβα. Στον αριθμό των στοιβών που υπολογίσαμε στην προηγούμενη παράγραφο, επομένως, θα πρέπει να προσθέσουμε το πλήθος των tasks που έχουν δημιουργηθεί, αλλά δεν έχουν ακόμη εκτελεστεί, σε κάθε σημείο του προγράμματος. Φυσικά, αν ένα πρόγραμμα δημιουργεί τα tasks κατά εκατοντάδες ή και χιλιάδες (που δεν είναι τόσο περίεργο όσο μπορεί να ακούγεται αρχικά), ο συνολικός αριθμός των στοιβών μπορεί να γίνει απαγορευτικά μεγάλος.

Ωστόσο, ακόμη και αν μπαίναμε στην διαδικασία να δεσμεύσουμε το χώρο κάθε στοίβας στην κοινόχρηστη μνήμη, αυτό από μόνο του δεν θα ήταν αρκετό. Η ArgoDSM χρησιμοποιεί χαλαρό πρωτόκολλο συνοχής μνήμης, που σημαίνει ότι για να μεταφερθούν οι τροποποιήσεις που έκανε ένας κόμβος στην κοινόχρηστη μνήμη στους υπόλοιπους, απαιτείται ρητή κλήση συγχρονισμού. Για να είμαστε καλυμμένοι σε κάθε δυνατό τρόπο δημιουργίας και χρήσης ενός task, απαιτείται να έχουμε κλήση συγχρονισμού κάθε φορά που ξεκινάμε την εκτέλεση ενός task που είχε δημιουργηθεί σε έναν απομακρυσμένο κόμβο. Στην περίπτωση των προγραμμάτων που μόνο ένα νήμα δημιουργεί όλα τα tasks και τα υπόλοιπα απλώς τα εκτελούν, αυτή η προσέγγιση είναι απαγορευτική.

Λαμβάνοντας τα παραπάνω υπόψιν, καταλήξαμε ότι ο πιο αποδοτικός τρόπος για να υποστηρίξουμε tasks με κοινόχρηστες μεταβλητές είναι να απενεργοποιήσουμε την δυνατότητα κλεψίματος tasks (tasks stealing) μεταξύ κόμβων. Για να συμβεί αυτό, χρειάζονται δύο βήματα. Πρώτον, καλούμε την συνάρτηση `torc_disable_stealing` κατά την αρχικοποίηση του προγράμματος, ώστε κατά τον έλεγχο για εκκρεμή tasks, να γίνεται αναζήτηση μόνο στις τοπικές ουρές του κόμβου. Δεύτερον, τροποποιήσαμε τον πηγαίο κώδικα της TORC ώστε κατά την δη-

[1]στην TORC τα task υλοποιούνται ως νήματα επιπέδου χρήστη και ως νήματα επιπέδου χρήστη έχουν στοίβα.

μιουργία ενός task, αυτό να εισάγεται στην τοπική ουρά του κόμβου και να μην στέλνεται σε έναν απομακρυσμένο. Η δυνατότητα κλεψίματος tasks μέσα στον ίδιο κόμβο εξακολουθεί να είναι ενεργοποιημένη, δηλαδή ένα νήμα μπορεί να εκτελέσει ένα task που βρίσκεται στην ουρά κάποιου άλλου νήματος, αρκεί να είναι και τα δύο στον ίδιο κόμβο.

5.6 Κλειδαριές

Όπως ορίζει το πρότυπο OpenMP, πρέπει να παρέχουμε τριών ειδών κλειδαριές:

- **Απλή (normal)** Είναι ο προεπιλεγμένος τύπος κλειδαριάς και χρησιμοποιείται πολύ περισσότερο από τους υπόλοιπους. Εξασφαλίζει ατομικότητα σε μία κρίσιμη περιοχή: μόνο ένα νήμα[2] μπορεί να την κλειδώσει, αν κάποιο νήμα την βρει ήδη κλειδωμένη, περιμένει μέχρι να ξεκλειδώσει. Αν το νήμα που την έχει κλειδώσει προσπαθήσει να την κλειδώσει ξανά (χωρίς να την ξεκλειδώσει πριν), το αποτέλεσμα είναι απροσδιόριστη συμπεριφορά και αποτελεί προγραμματιστικό σφάλμα.

- **Φωλιασμένη (nested)** Λειτουργεί όπως η απλή, με τη διαφορά ότι ο ιδιοκτήτης μιας ήδη κλειδωμένης κλειδαριάς μπορεί να την κλειδώσει ξανά χωρίς πρόβλημα. Για να θεωρηθεί ξεκλείδωτη, πρέπει να ξεκλειδωθεί από τον ιδιοκτήτη της όσες φορές κλειδώθηκε.

- **Περιστροφική (spin)** Λειτουργεί όπως η απλή, με τη διαφορά ότι αν ένα νήμα που προσπαθεί να την κλειδώσει τη βρει κλειδωμένη, πρέπει να ελέγχει συνεχώς την κατάστασή της μέσα σε ένα βρόχο, απασχολώντας τον επεξεργαστή χωρίς να εκτελεί ουσιαστική δουλειά.

Εφόσον τα προγράμματά μας (και συνεπώς τα νήματα που τα αποτελούν) εκτελούνται σε διαφορετικούς κόμβους, οι κλειδαριές που χρησιμοποιούμε θα πρέπει να είναι κατανεμημένες. Ευτυχώς, η βιβλιοθήκη ArgoDSM παρέχει απλές κατανεμημένες κλειδαριές, ωστόσο, υπάρχουν δύο προβλήματα. Πρώτον, η δημιουργία κλειδαριάς είναι συλλογική, δηλαδή ένα νήμα σε κάθε κόμβο πρέπει να καλέσει τη

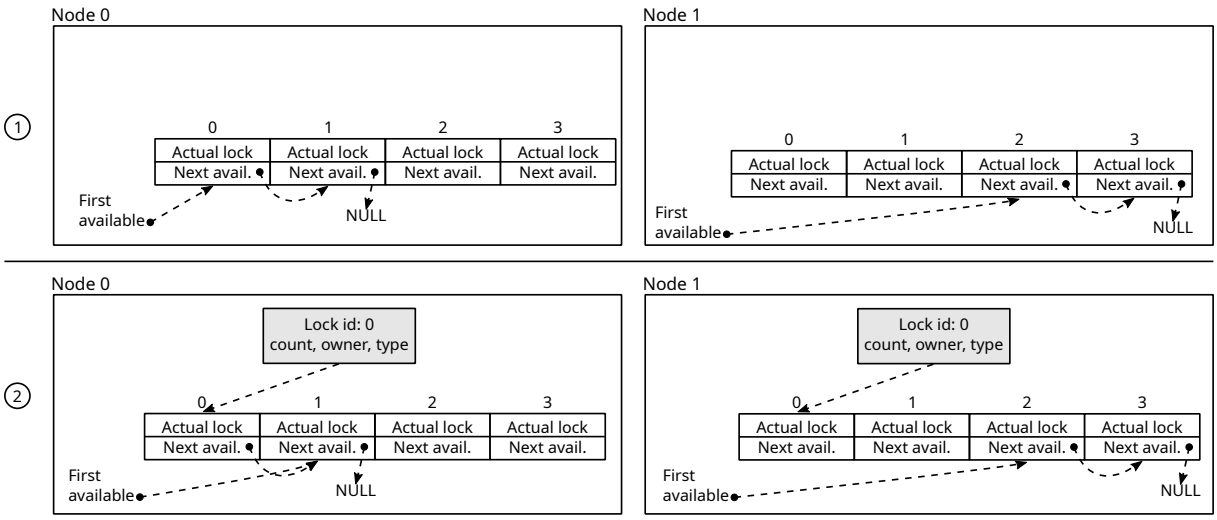
[2]Το πρότυπο OpenMP ορίζει ότι ιδιοκτήτης μιας κλειδαριάς είναι ένα task και όχι ένα νήμα. Χρησιμοποιούμε τον όρο νήμα καταχρηστικά, για λόγους απλότητας.

συνάρτηση δημιουργίας κλειδαριάς. Δεύτερον, η συνάρτηση δημιουργίας επιστρέφει ένα δείκτη στη νέα κλειδαριά, που αργότερα χρησιμοποιείται για το κλείδωμα και το ξεκλείδωμά της, ο οποίος όμως δείχνει σε τοπική μνήμη και συνεπώς είναι διαφορετικός σε κάθε κόμβο.

Για να λύσουμε το πρώτο πρόβλημα, αντί να δημιουργούμε δυναμικά τις κλειδαριές όταν τις χρειαζόμαστε, δημιουργούμε ένα σταθερό πλήθος κλειδαριών στην αρχή της εκτέλεσης του προγράμματος, πριν τρέξουμε τον κώδικα του χρήστη. Κάθε κόμβος έχει στη δικαιοδοσία του, δηλαδή μπορεί να δώσει στον κώδικα του χρήστη, ένα διαφορετικό υποσύνολο των κλειδαριών που παράχθηκαν. Αυτό φαίνεται γραφικά στο βήμα (1) του Σχήματος 5.8. Μάλιστα, επειδή στον αρχικό κόμβο βρίσκεται το νήμα που τρέχει όλο τον σειριακό κώδικα του προγράμματος του χρήστη, δίνουμε σε αυτό περισσότερες κλειδαριές. Το πλήθος των κλειδαριών που έχει στη διάθεσή του κάθε κόμβος ορίζεται από τις παραμέτρους `NODE_ZERO_NLOCKS` και `NODE_OTHER_NLOCKS`, που έχουν τις προεπιλεγμένες τιμές 100 και 50 αντίστοιχα και είναι εύκολα τροποποιήσιμο.

Μια εύλογη απορία θα ήταν: γιατί να το κάνουμε όλο αυτό αφού όταν το πρόγραμμα του χρήστη ζητάει μια καινούργια κλειδαριά, μπορούμε να βάλουμε το νήμα server (που κάθεται) να επικοινωνήσει με τους υπόλοιπους κόμβους και να καλέσουν όλοι μαζί τη συνάρτηση δημιουργίας κλειδαριάς; Η απάντηση είναι ότι αυτή η προσέγγιση προκαλεί συνθήκη συναγωνισμού (race condition). Αν δύο νήματα σε διαφορετικούς κόμβους ζητήσουν ταυτόχρονα μια κλειδαριά, δεν είναι βέβαιο ότι όλοι οι κόμβοι θα δημιουργήσουν πρώτα την κλειδαριά του ενός και μετά του άλλου. Για να το αποτρέψουμε αυτό θα χρειαζόταν να χρησιμοποιούμε μία προϋπάρχουσα κλειδαριά κάθε φορά, το οποίο δεν είναι αποδοτικό.

Το δεύτερο πρόβλημα στην ουσία σημαίνει ότι δεν έχουμε κάποιο τρόπο να προσδιορίσουμε μια κλειδαριά, αφού το μόνο που γνωρίζουμε για αυτή είναι ένας δείκτης, ο οποίος είναι διαφορετικός σε κάθε κόμβο. Για να το λύσουμε αντιστοιχίζουμε κάθε κλειδαριά με έναν ακέραιο, συγκεκριμένα την πρώτη κλειδαριά που δημιουργούμε με τον ακέραιο μηδέν, τη δεύτερη με τον ακέραιο ένα κ.ο.κ. Έτσι, αν πούμε "κλείδωσε την κλειδαριά τέσσερα", όλοι οι κόμβοι ξέρουν ποια κλειδαριά πρέπει να κλειδώσουν. Ωστόσο, όταν το πρόγραμμα του χρήστη ζητάει μια καινούργια κλειδαριά, πρέπει να του επιστρέφουμε έναν ακέραιο που περιέχει αυτή την αντιστοίχιση. Για να μπορεί η κλειδαριά να χρησιμοποιηθεί από νήματα που βρίσκονται σε διαφορετικούς κόμβους, ο ακέραιος θα πρέπει να βρίσκεται στην



Σχήμα 5.8: Δημιουργία και διαχείριση κλειδαριών. Βήμα (1): Στην αρχή της εκτέλεσης του προγράμματος, κάθε κόμβος δημιουργεί έναν πίνακα με κλειδαριές και μία λίστα με τις κλειδαριές που είναι διαθέσιμες. Εδώ έχουμε δύο κόμβους και ο καθένας έχει στη δικαιοδοσία του (δηλαδή μπορεί να δώσει στο πρόγραμμα του χρήστη) δύο κλειδαριές. Βήμα (2): Ένα νήμα στον πρώτο κόμβο ζητάει μια κλειδαριά. Ο OMPi φροντίζει ώστε η μνήμη για τις πληροφορίες της κλειδαριάς (μεταξύ αυτών και η αντιστοίχισή της με την πραγματική κλειδαριά) να βρίσκεται στον κοινόχρηστο χώρο (γκρι φόντο στο Σχήμα) ώστε να μπορεί να χρησιμοποιηθεί από όλους τους κόμβους. Ο κόμβος μηδέν που χρησιμοποίησε μια κλειδαριά φροντίζει να ενημερώσει τη λίστα με τις ελεύθερες κλειδαριές του.

κοινόχρηστη μνήμη. Αυτό φαίνεται γραφικά στο βήμα (2) του Σχήματος 5.8.

Επιπλέον, η ArgoDSM δεν παρέχει φωλιασμένες κλειδαριές και προκειμένου να τις υποστηρίξουμε χρειαζόμαστε δύο πράγματα: τον ιδιοκτήτη της κλειδαριάς (δηλαδή ποιο νήμα την έχει κλειδώσει ή NULL αν είναι ξεκλείδωτη) και έναν ακέραιο που αποθηκεύει το πλήθος των φορών που έχει κλειδωθεί η κλειδαριά. Αυτές οι πληροφορίες θα πρέπει επίσης να βρίσκονται στην κοινόχρηστη μνήμη, για να μπορούν να χρησιμοποιηθούν από νήματα σε διαφορετικούς κόμβους.

Συνοψίζοντας, οι κλειδαριές αποτελούνται από τρία μέρη:

- Έναν πίνακα με δείκτες που παίρνει τιμές καλώντας την συνάρτηση `argo_cohort_lock_create()` στην αρχή της εκτέλεσης του προγράμματος.
- Μια συνδεδεμένη λίστα με τις κλειδαριές του πίνακα που είναι ελεύθερες για χρήση από τον εκάστοτε κόμβο.

- Μια δομή με τον τύπο της κλειδαριάς, τον ιδιοκτήτη της, το πλήθος των φορών που έχει κλειδωθεί και τον ακέραιο με τον οποίο έχει αντιστοιχιστεί. Ο OMPi φροντίζει ώστε ο χώρος για τη δομή αυτή να δεσμεύεται στην κοινόχρηστη μνήμη.

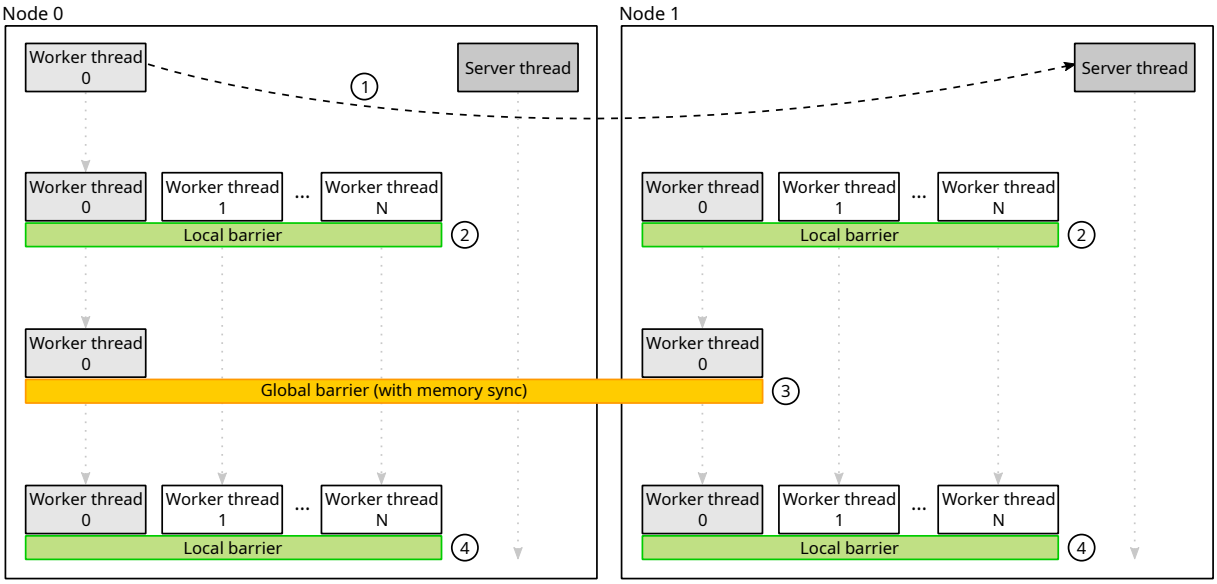
5.7 Barriers

Οι barriers αποτελούν βασική προγραμματιστική δομή στον παράλληλο προγραμματισμό και χρησιμοποιούνται για να εξασφαλίσουν το συγχρονισμό μεταξύ των νημάτων. Πιο συγκεκριμένα, μία κλήση barrier ορίζει ένα σημείο όπου όλα τα νήματα θα σταματήσουν την εκτέλεσή τους μέχρι να την καλέσει και το τελευταίο νήμα. Στη συνέχεια, όλα τα νήματα θα συνεχίσουν κανονικά την εκτέλεσή τους από το σημείο αυτό.

Εφόσον τα προγράμματά μας (και συνεπώς τα νήματα που τα αποτελούν) εκτελούνται σε διαφορετικούς κόμβους, οι barriers που χρησιμοποιούμε θα πρέπει να είναι κατανεμημένοι. Ευτυχώς, η βιβλιοθήκη ArgoDSM παρέχει κατανεμημένους barriers, οι οποίοι μάλιστα πέρα από το συγχρονισμό των νημάτων, εγγυώνται και το συγχρονισμό της μνήμης. Δηλαδή, στο τέλος του barrier όλοι οι κόμβοι θα έχουν την πιο πρόσφατη έκδοση όλων των δεδομένων που βρίσκονται στην κοινόχρηστη μνήμη. Αυτό είναι απαραίτητο αφού δουλεύουμε σε κατανεμημένο περιβάλλον, αλλά όχι αρκετό, μιας και το πρότυπο OpenMP ορίζει ότι τα νήματα που φτάνουν στο barrier, αντί να κάθονται περιμένοντας τα υπόλοιπα, θα πρέπει να εκτελούν εκκρεμή tasks.

Για τον λόγο αυτό, υλοποιήσαμε τον δικό μας κατανεμημένο barrier, ο οποίος φαίνεται γραφικά στο Σχήμα 5.9. Αρχικά, στο βήμα (1) το πρώτο νήμα στον πρώτο κόμβο φροντίζει να ενημερώσει τους τυχών ανενεργούς κόμβους (το πλήθος των νημάτων που ζητούνται σε μια παράλληλη περιοχή μπορεί να καλύπτεται χωρίς να χρησιμοποιούνται όλοι οι κόμβοι), μιας και όλοι οι κόμβοι πρέπει να συμμετέχουν στον barrier. Σε διαφορετική περίπτωση, οι ανενεργοί κόμβοι δεν θα γνώριζαν ότι πρέπει να εκτελέσουν τον κατανεμημένο barrier. Στη συνέχεια, χρησιμοποιούνται δύο τοπικοί barriers και ανάμεσά τους ένας κατανεμημένος: τα νήματα κάθε κόμβου συγχρονίζονται στον τοπικό barrier (βήμα (2)), ώστε να βεβαιωθούμε ότι όλα τα νήματα είναι έτοιμα. Μετά το πρώτο νήμα κάθε κόμβου καλεί τον κατανεμημένο barrier της ArgoDSM που εξασφαλίζει συγχρονισμό μνήμης (βήμα (3)) και τέλος,

όλα τα νήματα κάθε κόμβου συγχρονίζονται ξανά σε έναν τοπικό barrier, με σκοπό να περιμένουν το ένα νήμα που εκτελεί τον κατανεμημένο barrier, όπως φαίνεται στο βήμα (4).



Σχήμα 5.9: Τρόπος λειτουργίας του κατανεμημένου barrier. Βήμα (1): Ενημέρωση τυχών ανενεργών κόμβων ώστε να εκτελέσουν όλοι τον barrier. Βήμα (2): Τοπικός barrier. Βήμα (3): Κατανεμημένος barrier με συγχρονισμό μνήμης. Βήμα (4): Τοπικός barrier.

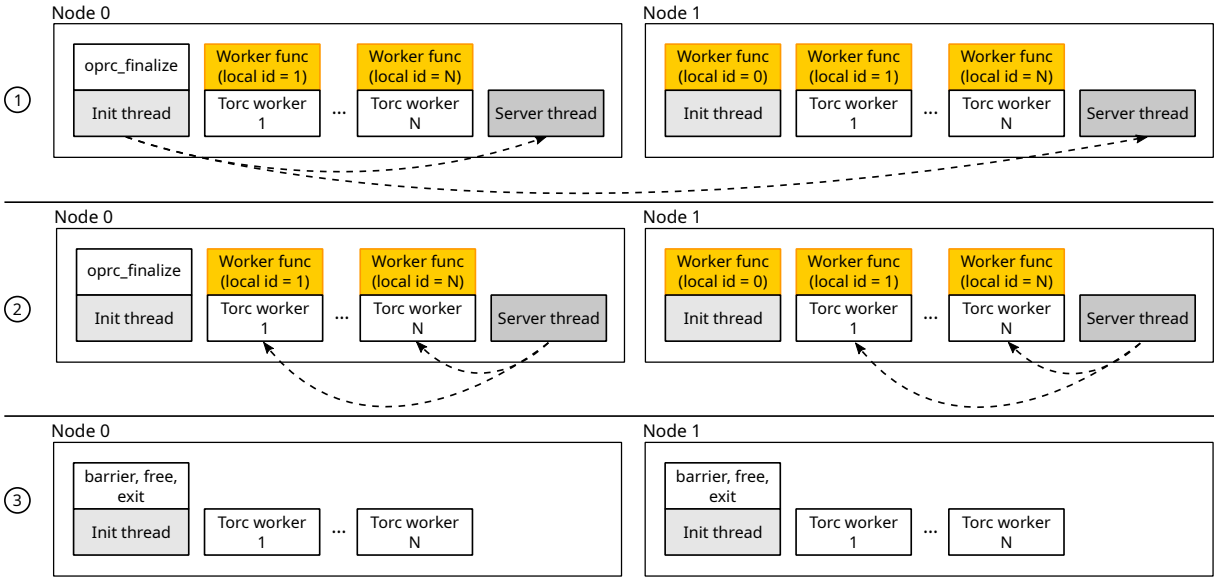
Οι τοπικοί barriers που χρησιμοποιούμε σε αυτήν την περίπτωση είναι βασισμένοι στους barriers του OMPi για την περίπτωση που χρησιμοποιούμε νήματα και εξασφαλίζουν ότι όλα τα tasks που έχουν δημιουργηθεί ως εκείνο το σημείο θα εκτελεστούν. Αρχικά, κάθε νήμα εκχωρεί την τιμή 1 στην θέση του πίνακα `arrived` (που είναι μέρος του barrier) που αντιστοιχεί την ταυτότητά του (id) στην παράλληλη περιοχή. Όσο αυτή η τιμή δεν έχει επιστρέψει σε 0, εκτελεί εκκρεμή tasks και, αν δεν υπάρχουν, περιμένει για λίγο. Το αρχικό νήμα του κόμβου, από την άλλη πλευρά, ελέγχει μια-μια όλες τις θέσεις του πίνακα και, όταν όλες γίνουν 1, τις αντικαθιστά με 0, ώστε τα υπόλοιπα νήματα να μπορούν να συνεχίσουν. Όσο δεν μπορεί να συνεχίσει, εκτελεί και αυτό εκκρεμή tasks. Στη συνέχεια, όλα τα νήματα ελέγχουν μια ακόμη φορά για εκκρεμή tasks και η ίδια διαδικασία επαναλαμβάνεται με τον πίνακα `released`, για να εξασφαλιστεί ότι έχουν εκτελεστεί όλα τα tasks, ακόμη και αυτά που τυχόν δημιουργήθηκαν από το τελευταίο task που εκτελέστηκε.

Στην αρχή του αλγορίθμου του τοπικού barrier υπάρχει ένας γρήγορος έλεγχος ώστε αν στην παράλληλη περιοχή συμμετέχει μόνο ένα νήμα, όλη η διαδικασία να

παρακαμφθεί. Τέλος, αξίζει να τονίσουμε ότι στους barriers, σε αντίθεση με τις κλειδαριές που είδαμε πριν, δεν χρειάζεται να αποθηκεύουμε καμία πληροφορία στην κοινόχρηστη μνήμη.

5.8 Τερματισμός

Σε αυτήν την ενότητα εξετάζουμε τη διαδικασία του τερματισμού, η οποία φαίνεται γραφικά στο Σχήμα 5.10.



Σχήμα 5.10: Διαδικασία τερματισμού.

Όταν το πρόγραμμα του χρήστη τελειώσει, στο αρχικό νήμα (στον πρώτο κόμβο) καλείται η συνάρτηση `oprc_finalize`, στην οποία θα πρέπει να εξασφαλίσουμε ότι όλα τα νήματα σε όλους του κόμβους τερματίζουν ομαλά τη λειτουργία τους. Αρχικά, όπως φαίνεται στο βήμα (1), στέλνουμε στο νήμα server κάθε κόμβου (συμπεριλαμβανομένου και αρχικού) το μήνυμα `FINALIZE`. Στο βήμα (2), το νήμα server ενημερώνει όλα τα νήματα workers του κόμβου για το μήνυμα `FINALIZE` και στη συνέχεια τερματίζει. Μόλις τα νήματα workers λάβουν το μήνυμα `FINALIZE` στο βήμα (3), τερματίζουν το task εκκίνησης που εκτελούν και περιμένουν για καινούργια tasks από την TORC. Από αυτή τη διαδικασία εξαιρείται το πρώτο νήμα worker σε κάθε κόμβο. Μόλις όλοι οι κόμβοι φτάσουν σε αυτό το σημείο (το ένα νήμα που έχει μείνει στον καθένα φροντίζει για το συγχρονισμό), απελευθερώνεται η μνήμη που είχε δεσμευτεί και δε χρειάζεται πια.

Τέλος, φροντίζουμε για τον ομαλό τερματισμό των βιβλιοθηκών που χρησιμοποιούμε. Αυτό το κάνουμε με την αντίστροφη σειρά από την αρχικοποίηση, δηλαδή πρώτα τερματίζουμε την ArgoDSM και μετά την TORC. Στη συνέχεια, καλούμε την `exit` σε κάθε κόμβο για να τερματίσουμε το πρόγραμμα. Η TORC έχει εγκαταστήσει κατά την αρχικοποίησή της ένα χειριστή τερματισμού (exit handler), δηλαδή μια συνάρτηση που καλείται μόλις κληθεί η `exit` και πριν το πρόγραμμα τερματίσει οριστικά, οπότε ο πραγματικός τερματισμός της, μαζί με των νημάτων workers που δεν έχουν άλλα tasks να εκτελέσουν και απλά περιμένουν, γίνεται σε αυτό το σημείο.

Κεφαλαιο 6

Πειραματικα αποτελεσματα

6.1 Τεχνικά χαρακτηριστικά συστήματος και μεθοδολογία

Προκειμένου να ποσοτικοποιήσουμε την απόδοση της υλοποίησής μας, χρησιμοποιήσαμε μια σειρά από πειράματα. Όλες οι μετρήσεις πραγματοποιήθηκαν στο cluster του Τμήματος Μηχανικών Η/Υ και Πληροφορικής καθώς και σε μεμονωμένους υπολογιστές. Το cluster, αν και παλαιότερης τεχνολογίας, διαθέτει 16 κόμβους[1] (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 είναι 1Gb Ethernet, όπου όλοι οι κόμβοι συνδέονται μέσω ενός switch. 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 7.4.0. Σε αυτό το σημείο αξίζει να τονίσουμε ότι η

[1] όταν το χρησιμοποιήσαμε για να τρέξουμε τα πειράματα μόνο 11 κόμβοι (συμπεριλαμβανομένου και του master) ήταν σε λειτουργία.

διασύνδεση μεταξύ των κόμβων είναι τύπου Ethernet. Η βιβλιοθήκη ArgoDSM έχει σχεδιαστεί με πολύ πιο σύγχρονες και αποδοτικές διασυνδέσεις κατά νου, όπως είναι η InfiniBand (IB). Επομένως, είναι αναμενόμενο να μην χρησιμοποιούμε στο έπακρο τις δυνατότητές της.

Τόσο η δική μας υλοποίηση, όσο και οι ArgoDSM και TORC χρησιμοποιούν το MPI για τον συγχρονισμό και την μεταφορά δεδομένων μεταξύ κόμβων. Αρχικά θέλαμε να χρησιμοποιήσουμε την υλοποίηση του Open MPI (μιας και αυτή συνίσταται από τους συγγραφείς της ArgoDSM) και συγκεκριμένα την έκδοση 4.0.2. Ωστόσο, οι συναρτήσεις που καλεί η ArgoDSM έχουν υλοποιηθεί μόνο για clusters που είναι συνδεδεμένα με δίκτυα Infiniband. Σε δίκτυα Ethernet η υλοποίηση των συναρτήσεων δεν παρέχει προστασία όταν καλείται από πολλαπλά νήματα (non thead-safe), κάτι που απαιτεί τόσο η TORC, όσο και η δική μας υλοποίηση. Έτσι, καταλήξαμε να χρησιμοποιήσουμε την υλοποίηση MPIch, και συγκεκριμένα την έκδοση 3.3.2.

Εκτελέσαμε κάθε πείραμα 10 φορές σε 1, 2, 4 και 8 κόμβους και κρατήσαμε όλους τους χρόνους εκτέλεσης. Χρονομετρήσαμε μόνο το παράλληλο κομμάτι του προγράμματος, δηλαδή δεν συμπεριλάβαμε τον χρόνο αρχικοποίησης μεταβλητών, πινάκων και ενδεχομένως ανοίγματος και διαβάσματος αρχείων εισόδου, καθώς και τον χρόνο εγγραφής των αποτελεσμάτων σε αρχεία. Ακόμη, δεν συμπεριλάβαμε τον χρόνο αρχικοποίησης της δικής μας υλοποίησης, που, μαζί με την αρχικοποίηση των βιβλιοθηκών ArgoDSM και TORC, είναι περίπου τέσσερα δευτερόλεπτα σε κάθε εκτέλεση, ανεξάρτητα με το πλήθος κόμβων που χρησιμοποιούνται.

Στα γραφήματα των επόμενων ενοτήτων παρουσιάζουμε τον ενδιάμεσο (median), τον ελάχιστο και τον μέγιστο χρόνο εκτέλεσης σε κάθε περίπτωση. Εκτελούμε, σε όλες τις εφαρμογές, κάθε πείραμα με 2 ειδών εισόδους, μια μικρή που χρειάζεται λιγότερο χρόνο και μια μεγαλύτερη που χρειάζεται περισσότερο, ώστε να διαπιστώσουμε την συμπεριφορά της υλοποίησής μας όταν το μέγεθος του προβλήματος αλλάζει. Υπολογίζουμε επίσης την *επιτάχυνση (speedup)* που παίρνουμε χρησιμοποιώντας τον τύπο $\frac{T_1}{T_n}$, με $n \in \{2, 4, 8\}$, όπου T_i είναι ο ενδιάμεσος (median) χρόνος εκτέλεσης με i κόμβους. Διαισθητικά, η επιτάχυνση μας λέει "πόσες φορές πιο γρήγορα εκτελείται το πρόγραμμα όταν χρησιμοποιούμε n κόμβους αντί για έναν".

Ακόμη, για να έχουμε μία βάση σύγκρισης, στα γραφήματα χρόνου εκτέλεσης παρουσιάζουμε τον χρόνο παράλληλης εκτέλεσης σε έναν κόμβο με μια πράσινη διακεκομμένη γραμμή. Αυτός προκύπτει από τον ενδιάμεσο (median) χρόνο 10

εκτελέσεων του προγράμματος σε έναν κόμβο χρησιμοποιώντας την υλοποίηση του OpenMP που παρέχεται από τον GCC.

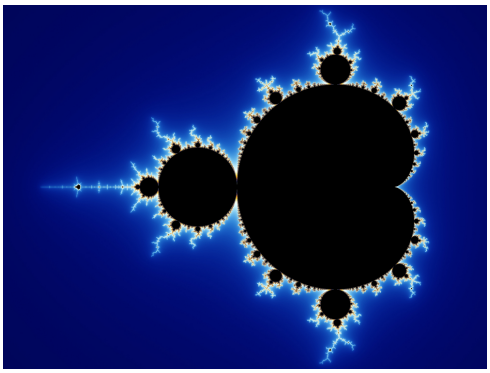
Στην εντολή εκκίνησης κάθε προγράμματος (με το `mpirun`) συμπεριλαμβάνεται το πλήθος νημάτων εργατών (worker threads) κάθε κόμβου (με την μεταβλητή `TORC_WORKERS`). Αποφασίσαμε να έχουμε 4 νήματα εργάτες ανά κόμβο σε κάθε εκτέλεση. Αυτός ο αριθμός αρχικά ακούγεται μεγάλος, αν αναλογιστούμε ότι κάθε κόμβος έχει 4 πυρήνες και πέρα από τα νήματα εργάτες έχουμε 1 νήμα server της υλοποίησής μας, 1 νήμα server της TORC, 3 βοηθητικά νήματα της ArgoDSM και 2 βοηθητικά νήματα που δημιουργεί η βιβλιοθήκη του MPI. Το θετικό στοιχείο που έχουν τα επιπλέον νήματα είναι ότι στην πλειοψηφία του χρόνου εκτέλεσης είναι μπλοκαρισμένα, επομένως δεν απασχολούν την CPU. Τα νήματα εργάτες, ομοίως, όταν δεν έχουν δουλειά να εκτελέσουν, μπλοκάρουν και αυτά. Τα πειράματα έδειξαν ότι μόνο με κάποια προγράμματα και μόνο όταν τα τρέχουμε σε έναν ή δύο κόμβους, έχουμε χρησιμοποίηση CPU (CPU utilization) πάνω από 4, με μέγιστο το 4.1, επομένως κρατήσαμε τα 4 νήματα εργάτες ανά κόμβο σε όλες τις περιπτώσεις.

Η άλλη σταθερά που έπρεπε αν αποφασίσουμε ήταν η μέγιστη μνήμη που ζητάμε από την ArgoDSM. Κατά την αρχικοποίηση, η ArgoDSM δέχεται ως παράμετρο την *μέγιστη* ποσότητα μνήμης sDSM που θέλουμε να χρησιμοποιήσουμε. Τα πειράματα έδειξαν ότι η ποσότητα που ζητάμε έχει ιδιαίτερα σημαντική επίδραση στον χρόνο εκτέλεσης ενός προγράμματος, ακόμη και όταν δεν χρησιμοποιούμε όλη την μνήμη που ζητάμε. Πιο συγκεκριμένα, ο ενδιάμεσος (median) χρόνος εκτέλεσης του προγράμματος `mandelbrot` με μέγεθος εικόνας 4600x4600 pixels ήταν 3.5 δευτερόλεπτα όταν ζητούσαμε μέγιστη ποσότητα μνήμης sDSM 86 MB, αλλά εκτοξευόταν στα 27 δευτερόλεπτα όταν ζητούσαμε μέγιστη μνήμη 256 MB, παρόλο που δεν δεσμεύσαμε και δεν χρησιμοποιούσαμε ποτέ την επιπλέον ποσότητα.

Το συμπέρασμα του παραπάνω πειραματισμού είναι ότι δεν πρέπει να ζητάμε από την ArgoDSM περισσότερη μνήμη από ότι σκοπεύουμε να χρησιμοποιήσουμε. Έτσι, σε όλα τα πειράματα καταλήξαμε να χρησιμοποιούμε 86 MB μνήμης sDSM. Εξαίρεση αποτελεί το πρόγραμμα `mandelbrot` με μέγεθος εικόνας 8600x8600 pixels, όπου χρησιμοποιούμε 300 MB μνήμης sDSM, μιας και αποθηκεύουμε την εικόνα στην μνήμη sDSM και οι μεγαλύτερες διαστάσεις απαιτούν περισσότερη μνήμη. Τέλος, σε κάθε περίπτωση χρησιμοποιούμε 8 MB για την στοίβα του αρχικού νήματος στον αρχικό κόμβο, που επίσης χρειάζεται να αποθηκεύουμε στην μνήμη sDSM.

6.2 Mandelbrot

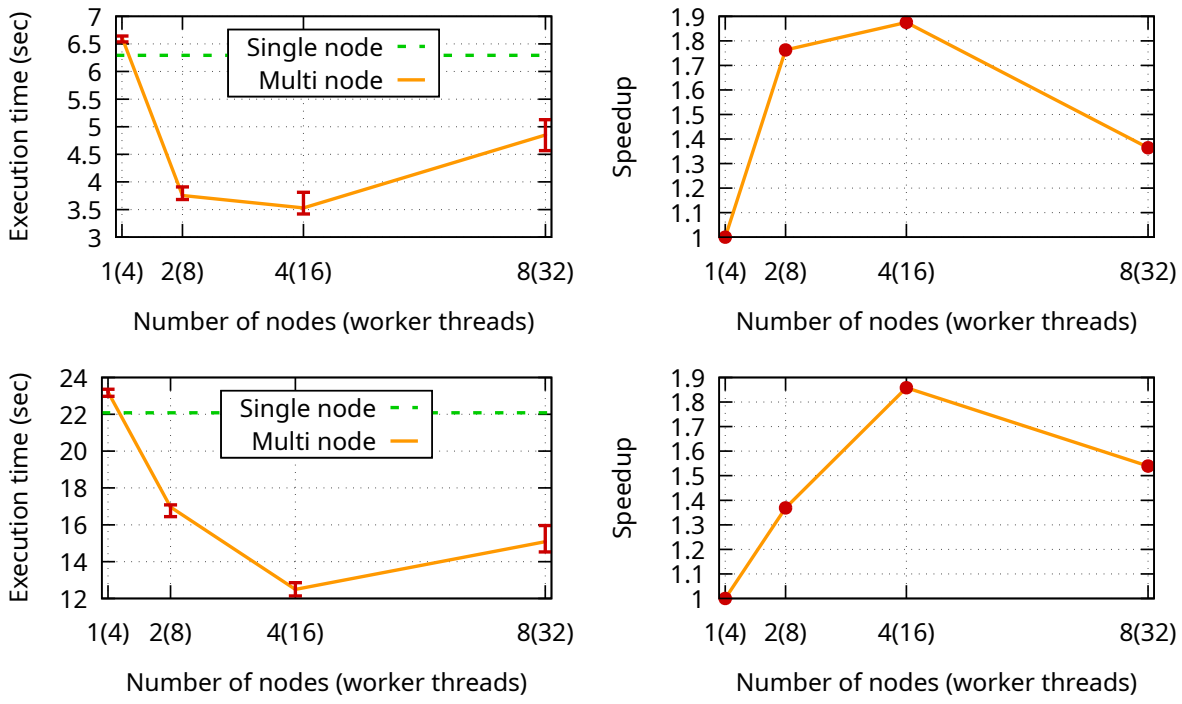
Το πρόγραμμα `mandelbrot` δημιουργεί μια εικόνα με το Mandelbrot fractal, όπως φαίνεται στο Σχήμα 6.1, οι διαστάσεις της οποίας ορίζονται από τον χρήστη. Για τον υπολογισμό της τιμής κάθε pixel απαιτείται μόνο η θέση (x, y) του pixel. Η αρχική υλοποίηση του προγράμματος ήταν εντελώς σειριακή.



Σχήμα 6.1: Παράδειγμα του Mandelbrot fractal.

Η εικόνα αναπαριστάται εσωτερικά ως ένας δισδιάστατος πίνακας, επομένως η παραλληλοποίηση που κάναμε στο πρόγραμμα είναι η εξής: κάθε νήμα υπολογίζει μια λωρίδα (πλήθος από συνεχόμενες γραμμές) της εικόνας, η οποία αποθηκεύεται στην κοινόχρηστη μνήμη. Ο χωρισμός σε λωρίδες γίνεται στατικά (static scheduling), επομένως, για να ξεκινήσει κάθε νήμα τους υπολογισμούς του χρειάζεται μόνο την θέση και το μέγεθος της λωρίδας που του αντιστοιχεί. Στο συγχρονισμό που γίνεται στο τέλος της παράλληλης περιοχής, μεταφέρονται στον αρχικό κόμβο όλες οι λωρίδες, με αποτέλεσμα αυτός να έχει ολόκληρη την εικόνα, την οποία γράφει σε ένα αρχείο png, ώστε να μπορούμε να ελέγξουμε την ορθότητα του προγράμματος.

Στο Σχήμα 6.2 βλέπουμε την γραφική παράσταση του χρόνου εκτέλεσης του προγράμματος καθώς και της επιτάχυνσης για μέγεθος εικόνας 4600x4600 pixels (επάνω) και 8600x8600 pixels (κάτω). Η αύξηση του μεγέθους της εικόνας σημαίνει δύο πράγματα: από τη μια πλευρά κάθε νήμα έχει να κάνει περισσότερη δουλειά, αλλά από την άλλη αυξάνεται και ο επικοινωνιακός φόρτος. Παρατηρούμε ότι και στις δύο περιπτώσεις παίρνουμε τον καλύτερο χρόνο εκτέλεσης όταν χρησιμοποιούμε 4 κόμβους, τότε η επιτάχυνση είναι περίπου 1.85. Αν προσθέσουμε κόμβους και από τους 4 πάμε στους 8, ο χρόνος εκτέλεσης χειροτερεύει. Ακόμη, η χρήση 2 κόμβων αντί για 1, έχει πολύ σημαντικότερη επίπτωση στον χρόνο εκτέλεσης για την μικρή εικόνα, παρά για την μεγάλη.



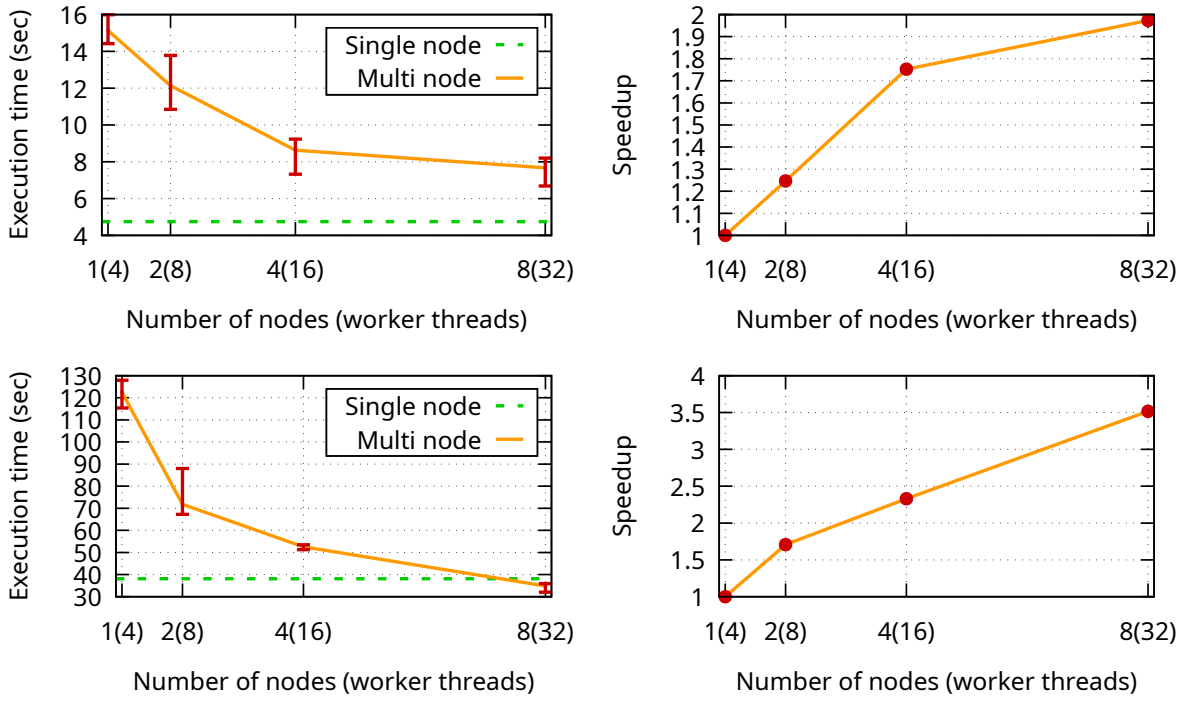
Σχήμα 6.2: Επάνω: Διάγραμμα χρόνου εκτέλεσης του προγράμματος Mandelbrot σε διαφορετικά πλήθη κόμβων (αριστερά) και επιτάχυνσης (δεξιά) για μέγεθος εικόνας 4600x4600 pixels. Κάτω: Διάγραμμα χρόνου εκτέλεσης (αριστερά) και επιτάχυνσης (δεξιά) για μέγεθος εικόνας 8600x8600 pixels.

Όταν χρησιμοποιούμε την κατανεμημένη υλοποίηση σε έναν μόνο κόμβο παίρνουμε λίγο χειρότερο χρόνο σε σχέση με την παράλληλη υλοποίηση του GCC, γεγονός αναμενόμενο αν σκεφτούμε ότι στην κατανεμημένη υλοποίηση χρησιμοποιούμε την μνήμη sDSM, που προσθέτει επιπλέον επιβάρυνση, ακόμη και αν τρέχουμε το πρόγραμμα σε έναν μόνο κόμβο. Σε κάθε περίπτωση, η προσθήκη δεύτερου κόμβου έχει ως αποτέλεσμα να πετυχαίνουμε καλύτερο χρόνο από ότι στην παράλληλη εκτέλεση σε έναν κόμβο.

6.3 Πολλαπλασιασμός πινάκων

Δημιουργήσαμε μόνοι μας το πρόγραμμα `matmul`, το οποίο πολλαπλασιάζει δύο πίνακες που δέχεται ως είσοδο, χρησιμοποιώντας τον παραδοσιακό αλγόριθμο πολυπλοκότητας $O(n^3)$. Τα στοιχεία των πινάκων εισόδου είναι τυχαίοι ακέραιοι στο διάστημα $[0, 100]$, οι οποίο προέρχονται από ομοιόμορφη κατανομή, επομένως οι

πίνακες δεν είναι αραιοί. Το σημαντικό με αυτόν τον αλγόριθμο είναι ότι κάθε στοιχείο του πίνακα γινομένου μπορεί να υπολογιστεί ανεξάρτητα από τα υπόλοιπα. Έτσι, η παραλληλοποίηση που κάναμε είναι η εξής: κάθε νήμα υπολογίζει μια λωρίδα (πλήθος συνεχόμενων γραμμών) του πίνακα γινομένου. Στον συγχρονισμό που γίνεται στο τέλος της παράλληλης περιοχής, όλες οι λωρίδες μεταφέρονται στον αρχικό κόμβο, ο οποίος γράφει τον πίνακα γινόμενο σε ένα αρχείο. Για να συμβεί αυτό, τόσο οι πίνακες εισόδου, όσο και ο πίνακας γινόμενο βρίσκονται στην μνήμη sDSM. Μετά την εκτέλεση του προγράμματος, συγκρίνουμε τα περιεχόμενα του παραγόμενου αρχείου με αυτά που παίρνουμε από την εκτέλεση της σειριακής έκδοσης του προγράμματος (που είναι εγγυημένα σωστά), για να ελέγξουμε την ορθότητά του.



Σχήμα 6.3: Επάνω: Διάγραμμα χρόνου εκτέλεσης του προγράμματος Matmul σε διαφορετικά πλήθη κόμβων (αριστερά) και επιτάχυνσης (δεξιά) για πίνακες μεγέθους 1024x1024. Κάτω: Διάγραμμα χρόνου εκτέλεσης (αριστερά) και επιτάχυνσης (δεξιά) για πίνακες μεγέθους 2048x2048.

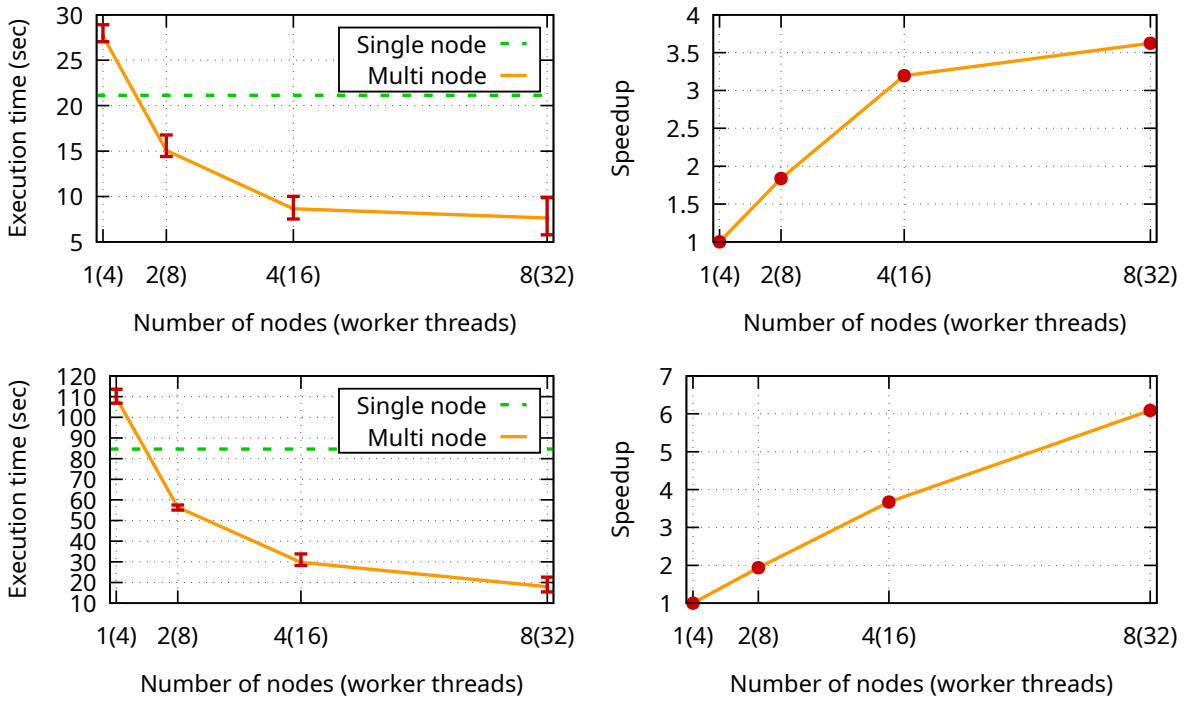
Στο Σχήμα 6.3 φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του προγράμματος καθώς και της επιτάχυνσης για μέγεθος πινάκων εισόδου (συνεπώς και πίνακα γινομένου) 1024x1024 (επάνω) και 2048x2048 (κάτω). Παρατηρούμε ότι και στις δύο περιπτώσεις η αύξηση του πλήθους των κόμβων που χρησιμοποιούμε

έχει ως αποτέλεσμα καλύτερους χρόνους εκτέλεσης. Μάλιστα, αυτό συμβαίνει σε μεγαλύτερο βαθμό στην περίπτωση των μεγάλων πινάκων. Επομένως, παίρνουμε την καλύτερη επιτάχυνση όταν χρησιμοποιούμε 8 κόμβους, που είναι σχεδόν 2 στην πρώτη περίπτωση και 3.5 στην δεύτερη.

Ωστόσο, παρατηρούμε ότι η παράλληλη εκτέλεση του προγράμματος σε έναν μόνο κόμβο με την υλοποίηση του GCC (πράσινη διακεκομμένη γραμμή στα γραφήματα) δίνει μικρότερους χρόνους εκτέλεσης σε όλες τις περιπτώσεις, με εξαίρεση την χρήση 8 κόμβων στους πίνακες 2048x2048. Αυτό μπορεί να εξηγηθεί ως εξής: στο μέρος τους προγράμματος που χρονομετρούμε, γίνονται μεταφορές δεδομένων σε δύο σημεία: στην αρχή της παράλληλης περιοχής, οι πίνακες εισόδου μεταφέρονται από τον αρχικό κόμβο (που τους διάβασε από αρχείο) στους υπόλοιπους κόμβους, ενώ στο τέλος της τα "κομμάτια" του πίνακα γινομένου μεταφέρονται από όλους τους κόμβους στον αρχικό. Θα μπορούσαμε να μην χρονομετρούμε την πρώτη μεταφορά δεδομένων, μιας και η αρχικοποίηση των πινάκων εισόδου μπορεί να θεωρηθεί μέρος της αρχικοποίησης του προγράμματος. Πιστεύουμε ότι μια τέτοια αλλαγή θα έφερνε τους χρόνους κατανεμημένης εκτέλεσης πιο κοντά στον χρόνο της παράλληλης εκτέλεσης και στις δύο περιπτώσεις πινάκων.

6.4 EP (Embarrassingly parallel)

Το πρόγραμμα `EP` (Embarrassingly parallel) αποτελεί μέρος της σουίτας εφαρμογών (benchmark suite) NAS [31], που αναπτύχθηκε από την NASA. Δουλειά του είναι να δημιουργεί ανεξάρτητες, τυχαίες γκαουσιανές (Gaussian) μεταβλητές χρησιμοποιώντας την μέθοδο Marsaglia polar, που είναι μια αποδοτική μέθοδος για την δημιουργία ψευδοτυχαίων αριθμών. Από προγραμματιστικής άποψης, αποτελείται από μια παράλληλη περιοχή, στην οποία υπάρχει μια οδηγία *reduction* καθώς και μια κρίσιμη περιοχή. Αυτό σημαίνει ότι, σε αντίθεση με τα δύο προηγούμενα προγράμματα, εδώ χρησιμοποιείται συγχρονισμός με κλειδαριές και μέσα στην παράλληλη περιοχή. Για τις εφαρμογές της σουίτας εφαρμογών NAS υπάρχουν συνολικά 8 κλάσεις μεγέθους, που καθορίζουν το μέγεθος εισόδου του προγράμματος. Οι χρονομετρήσεις μας έγιναν με τις κλάσεις A και B, όπου η κλάση B είναι περίπου 4 φορές μεγαλύτερη από την A. Όλα τα προγράμματα της σουίτας NAS πραγματοποιούν έλεγχο ορθότητας κατά την εκτέλεσή τους.



Σχήμα 6.4: Επάνω: Διάγραμμα χρόνου εκτέλεσης του προγράμματος EP σε διαφορετικά πλήθη κόμβων (αριστερά) και επιτάχυνσης (δεξιά) για την κλάση εισόδου A. Κάτω: Διάγραμμα χρόνου εκτέλεσης (αριστερά) και επιτάχυνσης (δεξιά) για την κλάση εισόδου B.

Στο Σχήμα 6.4 φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του προγράμματος καθώς και της επιτάχυνσης για την κλάση εισόδου A (επάνω) και B (κάτω). Παρατηρούμε ότι και στις δύο περιπτώσεις, όσο αυξάνουμε το πλήθος των κόμβων που χρησιμοποιούμε, ο χρόνος εκτέλεσης μειώνεται. Ήδη με τη χρήση 2 κόμβων αντί για 1, πετυχαίνουμε καλύτερη απόδοση από την παράλληλη υλοποίηση του GCC. Η επιτάχυνση που παίρνουμε με 8 κόμβους είναι 3.6 στην πρώτη περίπτωση και 6 στην δεύτερη. Μάλιστα, στην δεύτερη περίπτωση, η επιτάχυνση που παίρνουμε χρησιμοποιώντας 2 και 4 κόμβους είναι σχεδόν γραμμική. Σημαντικό ρόλο σε αυτές τις επιδόσεις παίζει φυσικά η υλοποίηση των κλειδαριών στην ArgoDSM: για να μειωθεί το κόστος του συγχρονισμού, προτεραιότητα στην κλειδαριά έχουν τα νήματα του κόμβου στον οποίο βρίσκονταν ο προηγούμενος ιδιοκτήτης της κλειδαριάς.

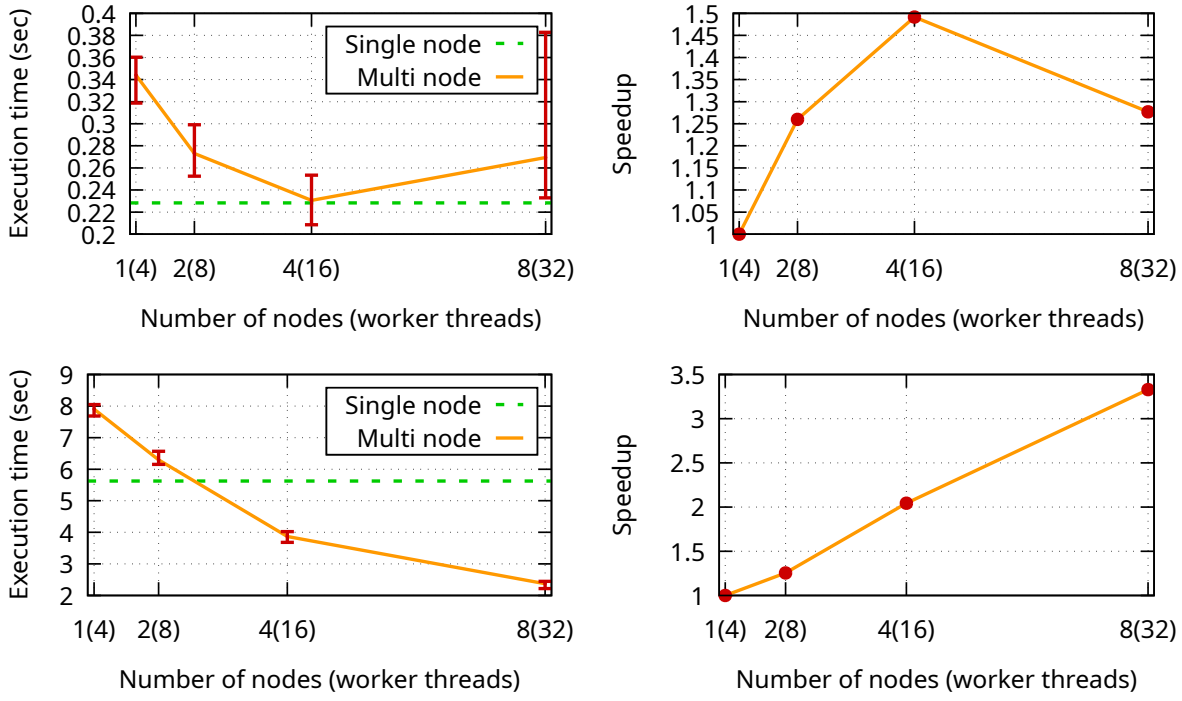
6.5 Ευθυγράμμιση πρωτεΐνης

Το πρόγραμμα `align` αποτελεί μέρος της σουίτας εφαρμογών (benchmark suite) BOTS (Barcelona OpenMP Tasks Suite) [32] και 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 δε μεταβάλλονται κατά τη διάρκεια της εκτέλεσης, επομένως οι τιμές τους μεταφέρονται μία φορά σε όλους τους κόμβους στην αρχή της παράλληλης περιοχής. Όλα τα προγράμματα της σουίτας BOTS πραγματοποιούν έλεγχο ορθότητας κατά την εκτέλεσή τους.

Μετά τις διάφορες αρχικοποιήσεις και την ανάγνωση του αρχείου εισόδου, το πρόγραμμα δημιουργεί μια παράλληλη περιοχή και τα νήματα όλων των κόμβων μοιράζονται τις επαναλήψεις του βρόχου for που ακολουθεί. Σε κάθε επανάληψή του υπολογίζεται ένα στοιχείο του αποτελέσματος ως εξής: αν ισχύει μια συνθήκη, τότε το αποτέλεσμα είναι 0. Διαφορετικά, δημιουργείται ένα καινούργιο task που υπολογίζει το αποτέλεσμα. Επομένως, καταλήγουμε σε δύο σημαντικά συμπεράσματα για την συμπεριφορά του προγράμματος. Πρώτον, δημιουργούνται tasks σε όλους τους κόμβους. Αυτό είναι πολύ σημαντικό επειδή στην τρέχουσα υλοποίηση έχουμε απενεργοποιήσει το κλέψιμο (stealing) των tasks μεταξύ κόμβων, που σημαίνει ότι, αν σε έναν κόμβο δεν δημιουργούταν κανένα task, τότε τα νήματά του δεν θα έκαναν καμία δουλειά. Δεύτερον, επειδή τα tasks παράγονται υπό συνθήκη, υπάρχει το ενδεχόμενο ανισοκατανομής φόρτου, δηλαδή κάποιοι κόμβοι να πρέπει να εκτελέσουν περισσότερα tasks σε σχέση με άλλους.

Η τροποποίηση που κάναμε σε αυτό το πρόγραμμα αφορά τον τρόπο διαμοιρασμού των επαναλήψεων (scheduling) του βρόχου for: αντί να μοιράζονται δυναμικά (dynamic), τώρα μοιράζονται στατικά (static), επομένως κάθε νήμα γνωρίζει από την αρχή ποιες επαναλήψεις του αντιστοιχούν. Αυτή η αλλαγή έχει ως αποτέλεσμα να αποφεύγεται η χρήση κατανεμημένης κλειδαριάς στην παράλληλη περιοχή και πειραματικά είδαμε μείωση του μέσου χρόνου εκτέλεσης, αλλά κυρίως μείωση της διακύμανσης του χρόνου εκτέλεσης σε διαδοχικές επαναλήψεις.

Στο Σχήμα 6.5 φαίνεται η γραφική παράσταση του χρόνου εκτέλεσης του προγράμματος καθώς και της επιτάχυνσης για το αρχείο εισόδου `prot20.aa` που παράγει ως έξοδο έναν πίνακα 400 στοιχείων (επάνω) και για το αρχείο εισόδου `prot100.aa`



Σχήμα 6.5: Επάνω: Διάγραμμα χρόνου εκτέλεσης του προγράμματος align σε διαφορετικά πλήθη κόμβων (αριστερά) και επιτάχυνσης (δεξιά) για το αρχείο εισόδου `prot20.aa`. Κάτω: Διάγραμμα χρόνου εκτέλεσης (αριστερά) και επιτάχυνσης (δεξιά) για το αρχείο εισόδου `prot100.aa`.

που παράγει ως έξοδο έναν πίνακα 10000 στοιχείων (κάτω). Στην πρώτη περίπτωση παρατηρούμε ότι η επίδοση δεν είναι καλή. Πετυχαίνουμε την καλύτερη επίδοση όταν χρησιμοποιούμε 4 κόμβους και ο μέσος χρόνος εκτέλεσης είναι ο ίδιους με αυτόν της παράλληλης υλοποίησης του GCC. Ο χρόνος εκτέλεσης παρουσιάζει μεγάλες διακυμάνσεις, ειδικά στην περίπτωση που χρησιμοποιούμε 8 κόμβους. Θα πρέπει να τονίσουμε, ωστόσο, ότι όλα τα πειράματα χρειάστηκαν κάτω από μισό δευτερόλεπτο για να εκτελεστούν. Υπό αυτές τις συνθήκες, οι διακυμάνσεις δεν είναι ιδιαίτερα μεγάλες και μάλλον είναι αναμενόμενες, δεδομένου ότι χρησιμοποιούμε σύστημα sDSM.

Στην δεύτερη περίπτωση, που το μέγεθος του προβλήματος είναι αισθητά μεγαλύτερο, τα αποτελέσματα είναι εντελώς διαφορετικά. Ο χρόνος εκτέλεσης μειώνεται όσο αυξάνουμε το πλήθος των κόμβων που χρησιμοποιούμε, ενώ με 4 και 8 κόμβους πετυχαίνουμε σημαντικά καλύτερο χρόνο σε σχέση με την παράλληλη υλοποίηση του GCC. Η επιτάχυνση που παίρνουμε είναι 2 όταν χρησιμοποιούμε 4 κόμβους και σχεδόν 3.25 όταν χρησιμοποιούμε 8. Ακόμη και σε αυτή την περίπτωση, όμως, η

επιτάχυνση δεν είναι ιδιαίτερα καλή. Όπως είπαμε προηγουμένως, το πρόβλημα αυτού του προγράμματος είναι η ανισοκατανομή του φόρτου: τα πειράματα έδειξαν ότι όταν χρησιμοποιούμε 4 ή 8 κόμβους, ο πρώτος κόμβος καταλήγει να δημιουργεί και να εκτελεί διπλάσιο αριθμό tasks, σε σχέση με τους υπόλοιπους, με αποτέλεσμα να αυξάνει τον συνολικό χρόνο εκτέλεσης του προγράμματος.

Τέλος, αξίζει να τονίσουμε ότι από τα 12 προγράμματα που δημιουργούν tasks και συμπεριλαμβάνονται στην σουίτα BOTS, το `align` είναι το μοναδικό στο οποίο όλα τα νήματα που συμμετέχουν στην παράλληλη περιοχή δημιουργούν tasks. Σε όλα τα υπόλοιπα, μόνο ένα νήμα δημιουργεί tasks και τα υπόλοιπα τα εκτελούν. Αυτός ο τρόπος χρήσης των tasks (που φαίνεται να είναι ο πιο συχνά χρησιμοποιούμενος στην πράξη) δεν ταιριάζει με την τρέχουσα υλοποίησή μας. Αν τα εκτελούσαμε, όσους κόμβους και αν χρησιμοποιούσαμε, όλα τα tasks θα εκτελούνταν στον έναν κόμβο που τα δημιουργεί. Οι άλλοι κόμβοι θα ήταν σαν να μην υπάρχουν, αφού δεν θα έκαναν καμία δουλειά. Το γεγονός αυτό, σε συνδυασμό με τις παρατηρήσεις που κάναμε στην προηγούμενη παράγραφο, μας οδηγούν στο παρακάτω συμπέρασμα. Η δυνατότητα κλεψίματος (stealing) tasks μεταξύ κόμβων είναι μέγιστης σημασίας για την αποδοτική εκτέλεση προγραμμάτων με tasks. Σε διαφορετική περίπτωση θα πρέπει να δημιουργούνται tasks σε όλους τους κόμβους (που όπως είδαμε δεν είναι η συνηθισμένη τακτική) και τα tasks που δημιουργούνται σε κάθε κόμβο να χρειάζονται περίπου τον ίδιο χρόνο για να εκτελεστούν (που είναι ένας ακόμη περιορισμός).

6.6 Επίδοση σε έναν κόμβο

Στις προηγούμενες ενότητες εστιάσαμε στην συμπεριφορά και την απόδοση της υλοποίησής μας καθώς αυξάνουμε το πλήθος των κόμβων που χρησιμοποιούμε. Σε αυτή την ενότητα εκτελούμε τα ίδια πειράματα σε έναν μόνο κόμβο και παρατηρούμε την απόδοση της υλοποίησής μας καθώς αυξάνουμε το πλήθος των νημάτων εργατών εντός του κόμβου. Όπως και στις προηγούμενες ενότητες, συγκρίνουμε με την υλοποίηση του GCC.

Για αυτό το σετ πειραμάτων χρησιμοποιήσαμε διαφορετικό μηχάνημα και συγκεκριμένα το Dell EMC PowerEdge R840 που ανήκει στην ερευνητική ομάδα παράλληλης επεξεργασίας. Αποτελείται από 4 επεξεργαστές Intel Xeon Gold 6130 με

16 πυρήνες ο καθένας και 256 GB RAM συνολικά.

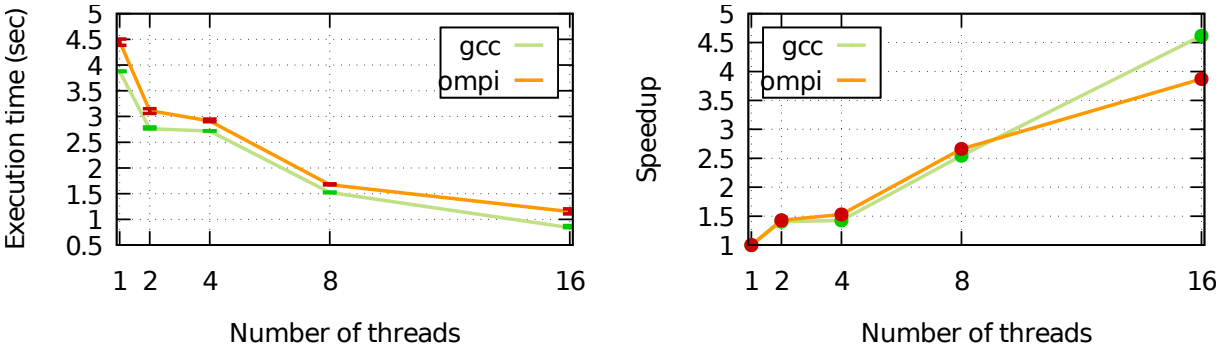
Θα πρέπει να διευκρινίσουμε ότι στην υπόλοιπη Ενότητα, όταν αναφερόμαστε σε πλήθος νημάτων, εννοούμε *το πλήθος των νημάτων εργατών* μόνο. Στη δική μας υλοποίηση πέραν των νημάτων εργατών, τρέχουν και τα ακόλουθα βοηθητικά νήματα: 1 νήμα server της υλοποίησής μας για επικοινωνία με τους υπόλοιπους κόμβους (όταν αυτοί υπάρχουν), 1 νήμα server της TORC για επικοινωνία με τους υπόλοιπους κόμβους (όταν αυτοί υπάρχουν) και 3 βοηθητικά νήματα της ArgoDSM για την διαχείριση της μνήμης sDSM. Αυτά τα νήματα επιτελούν μόνο βοηθητικές λειτουργίες και σε καμία περίπτωση δεν εκτελούν κώδικα του χρήστη, για παράδειγμα παράλληλες περιοχές και tasks.

Αρχικά δοκιμάζουμε το πρόγραμμα Mandelbrot. Στο Σχήμα 6.6 παρατηρούμε τον χρόνο εκτέλεσης και την επιτάχυνσή του σε έναν κόμβο. Η υλοποίησή μας είναι λίγο πιο αργή από αυτήν του GCC, ωστόσο ακολουθούμε την επιτάχυνση του GCC, με εξαίρεση την περίπτωση που χρησιμοποιούμε 16 νήματα, όπου σημειώνουμε αρκετά μικρότερη βελτίωση.

Είναι αξιοπερίεργο, ωστόσο, ότι σε ένα πρόγραμμα που είναι από τα πλέον κατάλληλα για παραλληλοποίηση, καμία από τις δύο υλοποιήσεις δεν καταφέρνει να πετύχει γραμμική επιτάχυνση, όπως θα αναμέναμε. Μάλιστα, στην περίπτωση που αυξάνουμε το πλήθος των νημάτων από 2 σε 4 παρατηρούμε πολύ μικρή βελτίωση, αντί για υποδιπλασιασμό του χρόνου εκτέλεσης. Υποψιαζόμαστε ότι αυτό οφείλεται, τουλάχιστον εν μέρη, στον τρόπο που παραλληλοποιούμε το πρόγραμμα: κάθε νήμα υπολογίζει μία λωρίδα (πλήθος από συνεχόμενες γραμμές) του αποτελέσματος. Επειδή κάθε στοιχείο του πίνακα απαιτεί διαφορετικό πλήθος υπολογισμών για να παραχθεί, μπορεί να προκαλείται ανισοκατανομή φόρτου και κάποια νήματα να πρέπει τελικά να εκτελέσουν σημαντικά περισσότερη δουλειά σε σχέση με άλλα.

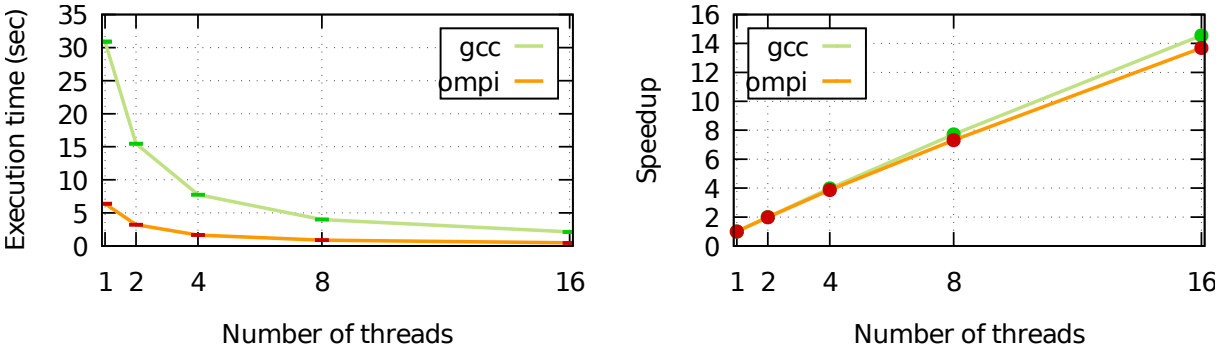
Σε σχέση με τον λίγο πιο μικρό χρόνο εκτέλεσης που δίνει ο GCC, μπορούμε να πούμε πως είναι αναμενόμενο. Αυτό διότι η υλοποίησή μας είναι σημαντικά πιο πολύπλοκη: αποθηκεύουμε τα κοινόχρηστα δεδομένα στην μνήμη sDSM, για το χειρισμό της οποίας απαιτείται διαχειριστική δουλειά από την βιβλιοθήκη ArgoDSM.

Στη συνέχεια, πειραματιστήκαμε με το πρόγραμμα align, στο οποίο όλοι οι κόμβοι δημιουργούν και εκτελούν tasks. Στο Σχήμα 6.7 παρατηρούμε τον χρόνο εκτέλεσης και την επιτάχυνσή του σε έναν κόμβο. Παρατηρούμε ότι η επιτάχυνση σε 2, 4 και 8 νήματα είναι η ιδανική, ενώ με 16 νήματα μειώνεται λίγο. Ωστόσο, στο διάγραμμα του χρόνου εκτέλεσης παρατηρούμε κάτι πολύ ενδιαφέρον: η υλοποίησή



Σχήμα 6.6: Χρόνος εκτέλεσης και επιτάχυνση του προγράμματος Mandelbrot σε έναν κόμβο.

μας είναι αισθητά πιο γρήγορη (μάλιστα πολύ πιο γρήγορη όταν χρησιμοποιούμε μικρό αριθμό νημάτων) από αυτήν του GCC, παρόλο που είναι πιο πολύπλοκη: οι κοινόχρηστες μεταβλητές αποθηκεύονται στην μνήμη sDSM, ενώ τα tasks παράγονται και εκτελούνται μέσω της βιβλιοθήκης TORC. Πιστεύουμε ότι αυτό οφείλεται αφενός στο ότι η βιβλιοθήκη TORC έχει σχεδιαστεί για να προσφέρει υψηλές επιδόσεις και αφετέρου στο ότι η υλοποίηση του GCC δεν είναι βελτιστοποιημένη για την δημιουργία και εκτέλεση εκατομμυρίων tasks.



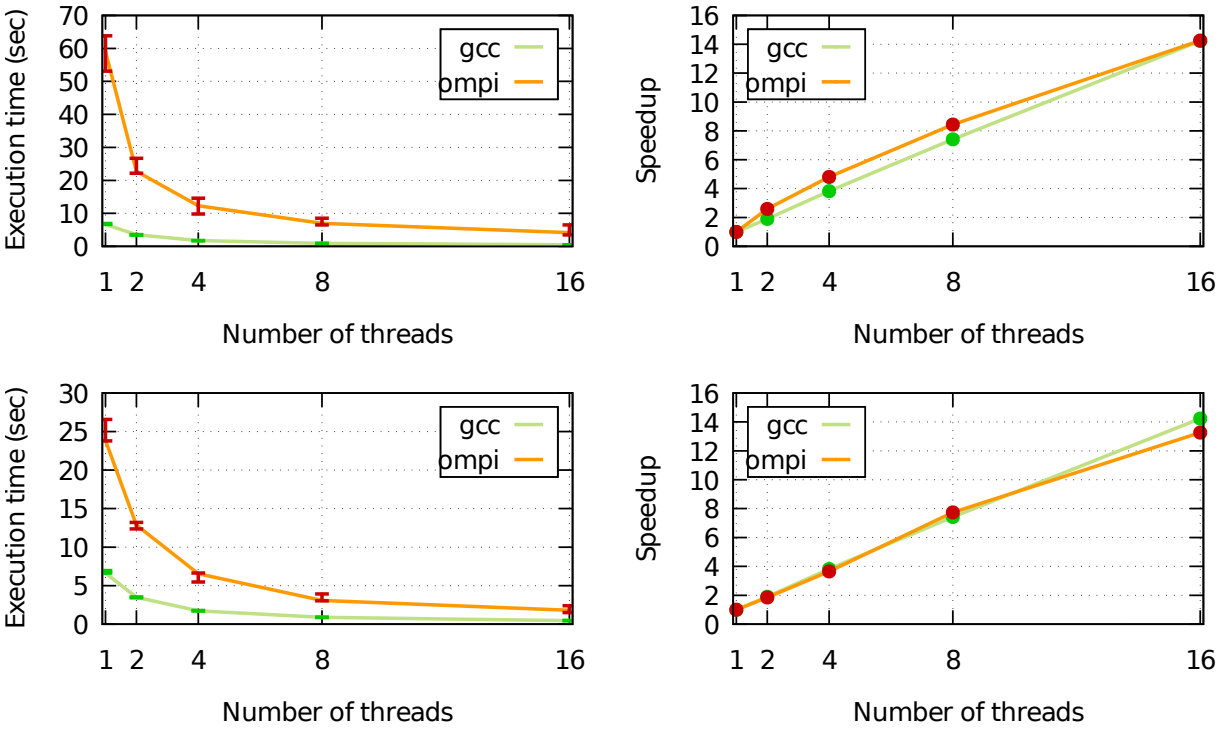
Σχήμα 6.7: Χρόνος εκτέλεσης και επιτάχυνση του προγράμματος align σε έναν κόμβο.

Τέλος, πειραματιστήκαμε με το πρόγραμμα Matmul, που πραγματοποιεί πολλαπλασιασμό πινάκων, με τον κλασσικό αλγόριθμο πολυπλοκότητας $O(n^3)$. Στα επάνω γραφήματα του Σχήματος 6.8 φαίνεται ο χρόνος εκτέλεσης και η επιτάχυνσή του σε έναν κόμβο, ακολουθώντας την διαδικασία μεταγλώττισης που έχουμε περιγράψει στο Κεφάλαιο 5.

Παρατηρούμε ότι η υλοποίησή μας είναι πολύ πιο αργή από αυτήν του GCC και ειδικά όταν χρησιμοποιούμε ένα μόνο νήμα. Πιστεύουμε ότι η ArgoDSM έχει κάποιο

πρόβλημα (bug) και όταν τρέχουμε το πρόγραμμα με ένα μόνο νήμα καθυστερεί περισσότερο από όσο θα έπρεπε. Επειδή χρησιμοποιούμε τον χρόνο εκτέλεσης σε ένα νήμα ως βάση για τον υπολογισμό της επιτάχυνσης, προκύπτει το εξής πρόβλημα: η επιτάχυνση της υλοποίησής μας όταν χρησιμοποιούμε 2 νήματα είναι μεγαλύτερη του 2, όταν χρησιμοποιούμε 4 νήματα είναι μεγαλύτερη του 4 και όταν χρησιμοποιούμε 8 νήματα είναι ελαφρώς μεγαλύτερη του 8. Πρόκειται βέβαια για μια μη φυσιολογική υπεργραμμική επιτάχυνση η οποία οφείλεται στον ανεξήγητα μεγάλο χρόνο εκτέλεσης που παρατηρείται για 1 νήμα.

Όμως, η βιβλιοθήκη ArgoDSM μας παρέχει δύο υλοποιήσεις: μία γενική υλοποίηση που δουλεύει σε πολλαπλούς κόμβους και την οποία χρησιμοποιήσαμε σε όλα τα πειράματα έως τώρα και μία βελτιστοποιημένη υλοποίηση που δουλεύει μόνο σε έναν κόμβο. Στα κάτω γραφήματα του Σχήματος 6.8 χρησιμοποιούμε την δεύτερη υλοποίηση. Αυτή εξακολουθεί να είναι πιο αργή από την υλοποίηση του GCC, ωστόσο είναι δύο φορές γρηγορότερη από την γενική υλοποίηση της ArgoDSM. Επιπλέον, λύνει το πρόβλημα που περιγράψαμε στην προηγούμενη παράγραφο και τώρα πετυχαίνουμε την ιδανική επιτάχυνση όταν χρησιμοποιούμε 2, 4 και 8 νήματα.



Σχήμα 6.8: Χρόνος εκτέλεσης και επιτάχυνση του προγράμματος Matmul σε έναν κόμβο. Επάνω: χρησιμοποιώντας την γενική υλοποίησης της ArgoDSM. Κάτω: χρησιμοποιώντας την υλοποίηση της ArgoDSM που είναι βελτιστοποιημένη για την εκτέλεση σε έναν κόμβο.

Κεφαλαιο 7

Επιλογος

7.1 Σύνοψη διπλωματικής εργασίας

Η προσπάθεια να υποστηριχτούν προγράμματα OpenMP σε clusters έχει ιστορία η οποία άρχισε και τελείωσε την προηγούμενη δεκαετία. Δυστυχώς σε αυτήν δεν συμπεριλαμβάνονται οι σημαντικές εξελίξεις στο προγραμματιστικό μοντέλο του OpenMP που συντελέστηκαν τα τελευταία δέκα χρόνια. Ίσως η σημαντικότερη εξέλιξη ήταν η προσθήκη των tasks, η οποία επέκτεινε τις δυνατότητες και την εφαρμοσιμότητα του OpenMP σε πολλούς τύπους εφαρμογών. Η παρούσα διπλωματική εργασία είναι η πρώτη που προσπαθεί να αναβιώσει και να συμπληρώσει τις προηγούμενες προσπάθειες για εκτέλεση προγραμμάτων OpenMP σε clusters, υποστηρίζοντας όμως όλες τις σύγχρονες ευκολίες που το μοντέλο αυτό παρέχει.

Ξεκινήσαμε μελετώντας τον παραλληλοποιητικό μεταφραστή OMPi και πιο συγκεκριμένα τον τρόπο λειτουργίας του και τις εσωτερικές δομές του που υλοποιούν τις οδηγίες *for* και *task* του προτύπου OpenMP. Επίσης, αναλύσαμε την υπάρχουσα υποδομή του OMPi για εκτέλεση προγραμμάτων OpenMP σε κατανεμημένο περιβάλλον, πάνω στην οποία βασίστηκε η δική μας υλοποίηση. Εξηγήσαμε την βασική αρχή λειτουργίας των βιβλιοθηκών sDSM και tasking σε κατανεμημένο περιβάλλον και αναφέραμε συνοπτικά παραδείγματα τέτοιων βιβλιοθηκών. Εξετάσαμε ενδελε-

χώς τον τρόπο λειτουργίας των ArgoDSM και TORC, καθώς και τους λόγους που μας έκαναν να τις επιλέξουμε για την υλοποίησή μας.

Στο πλαίσιο αυτής της διπλωματικής εργασίας, σχεδιάσαμε και υλοποιήσαμε πλήρως μία νέα οντότητα εκτέλεσης (execution entity) που επιτρέπει την διαφανή εκτέλεση προγραμμάτων OpenMP σε clusters. Με αυτό τον τρόπο, οι προγραμματιστές μπορούν να εκτελέσουν τις παράλληλες εφαρμογές τους σε κατανεμημένα περιβάλλοντα, αυξάνοντας τις επιδόσεις τους, απαιτώντας από ελάχιστες έως και μηδενικές αλλαγές στον πηγαίο κώδικά τους.

Παρουσιάσαμε τη γενική εικόνα και τον τρόπο διαχείρισης και επικοινωνίας μεταξύ των κόμβων χρησιμοποιώντας τη βιβλιοθήκη MPI. Έπρεπε να λύσουμε θέματα επικοινωνιών, διαχείρισης κοινόχρηστων μεταβλητών και διευθυνσιοδότησης, εκκίνησης των κόμβων και αρχικοποίησης των βιβλιοθηκών αλλά και των νημάτων εργατών 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 υλοποίησή μας πέρασε από εξαντλητικά τεστ ορθότητας και καλής λειτουργίας. Χρησιμοποιήσαμε τόσο υπάρχοντα προγράμματα (benchmarks), όσο και προγράμματα που φτιάξαμε οι ίδιοι για να αναλύσουμε τις επιδόσεις της. Καταλήξαμε στο συμπέρασμα ότι μπορούμε να δούμε αξιόλογη επιτάχυνση, αν εξασφαλίσουμε ότι το πρόγραμμα που εκτελούμε δε χρειάζεται να κάνει εκτεταμένη χρήση λειτουργιών συγχρονισμού για την εξασφάλιση συνοχής της κοινόχρηστης μνήμης και αν τα tasks παράγονται από τα νήματα όλων των κόμβων.

7.2 Προτάσεις για μελλοντική εργασία

Όπως διαπιστώσαμε από τα πειραματικά αποτελέσματα μπορούμε να πετύχουμε αξιόλογες επιδόσεις με την τρέχουσα υλοποίησή μας. Ωστόσο, προτείνουμε τις παρακάτω ιδέες για μελλοντική εργασία:

- Επανεξέταση των επιλογών που έχουμε και εύρεση λύσης ώστε να είναι εφικτό το κλέψιμο tasks (task stealing) μεταξύ διαφορετικών κόμβων, που για την ώρα έχουμε απενεργοποιήσει, λόγω ενός προβλήματος με τις κοινόχρηστες μεταβλητές. Με αυτό τον τρόπο θα είναι δυνατή η κατανομή των εργασιών σε όλο το εύρος ενός cluster αντί να περιορίζεται μόνο στους επεξεργαστι-

κούς πυρήνες του κόμβου που παρήγαγε το task, με προφανείς ευεργετικές επιδράσεις στις επιδόσεις.

- Πειραματισμός με επιπλέον βιβλιοθήκες sDSM, tasking, καθώς και εφαρμογές. Παρότι η ArgoDSM είναι ίσως η πιο σύγχρονη sDSM, είχε κάποια σημεία όπου παρουσίαζε μικρότερη από την αναμενόμενη επίδοση. Οι βιβλιοθήκες tasking πιθανώς θέλουν επιπρόσθετη δουλειά για να υποστηρίζουν εγγενώς τα *tasks* του OpenMP. Στη συνέχεια, θα μπορούμε να εκτελέσουμε επιπλέον εφαρμογές που δημιουργούν *tasks* σε όλους τους κόμβους του cluster.

- Επανεκτέλεση των προγραμμάτων (benchmarks) που χρησιμοποιήσαμε σε cluster με σύγχρονες υποδομές δικτύωσης. Η βιβλιοθήκη ArgoDSM έχει σχεδιαστεί με τέτοια συστήματα κατά νου (οι σχεδιαστές της μάλιστα τα συνιστούν ανεπιφύλακτα) και εκτιμούμε ότι θα μπορέσουμε να πάρουμε σημαντικά καλύτερες επιδόσεις σε αυτά.

Τέλος, αξίζει να σημειώσουμε ότι κατόπιν προτάσεως, μας δόθηκαν υπολογιστικοί κόμβοι και χρόνος στο υπερυπολογιστικό σύστημα ARIS του ΕΔΕΤ. Η υποδομή ARIS αποτελείται από συνολικά τέσσερις νησίδες υπολογιστικών συστημάτων βασισμένους σε αρχιτεκτονική Intel x86 διασυνδεμένους σε ένα ενιαίο δίκτυο Infiniband FDR14 που προσφέρουν πολλαπλές δυνατότητες και αρχιτεκτονικές επεξεργασίας. Εμείς χρησιμοποιήσαμε την νησίδα thin nodes, η οποία βασίζεται στην πλατφόρμα IBM NeXtScale και τους επεξεργαστές Intel Xeon E5-2680v2. Διαθέτει 426 υπολογιστικούς κόμβους και προσφέρει συνολικά 8.520 πυρήνες (CPU cores). Ωστόσο, παρά τις προσπάθειές μας, δεν κατέστη δυνατόν να εκτελέσουμε τις εφαρμογές που θέλαμε, λόγω τεχνικών προβλημάτων, τα οποία, λόγω του περιορισμένου χρόνου που είχαμε στην διάθεσή μας, δεν μπορέσαμε να εντοπίσουμε και να επιλύσουμε. Παρόλα αυτά, θα ήταν πολύ ενδιαφέρον να προσπαθήσουμε ξανά στο μέλλον.

Βιβλιογραφια

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

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

[3] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for openmp in clang and llvm," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–11. [Online]. Available: https://doi.org/10.1109/LLVM-HPC.2016.6

[4] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, "Assessing the performance of openmp programs on the intel xeon phi," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 547–558.

[5] S. N. Agathos, A. Papadogiannakis, and V. V. Dimakopoulos, "Targeting the parallella," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 662–674.

[6] 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: Springer International Publishing, 2015, pp. 30–41.

[7] P. Keleher, A. Cox, H. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," 06 1999.

[8] H. Lu, Y. C. Hu, and W. Zwaenepoel, "Openmp on networks of workstations," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. USA: IEEE Computer Society, 1998, p. 1–15.

[9] Y. C. Hu, Honghui Lu, A. L. Cox, and W. Zwaenepoel, "Openmp for networks of smps," in *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, April 1999, pp. 302–310.

[10] J. P. Hoeflinger, "Extending openmp to clusters," *White Paper, Intel Corporation*, 2006.

[11] Costa JJ, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running openmp applications efficiently on an everything-shared sdsm," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 35–.

[12] P. Hadjidoukas, E. Polychronopoulos, and T. Papatheodorou, "Openmp runtime support for clusters of multiprocessors," vol. 2716, 06 2003, pp. 180–194.

[13] Y.-S. Kee, J. Kim, and S. Ha, "Parade: An openmp programming environment for smp cluster systems," 12 2003, pp. 6– 6.

[14] 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.

[15] 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*, July 2008, pp. 189–196.

[16] F. Sitaras, "Αποδοτική εκτέλεση προγραμμάτων openmp σε συστάδες Η/Υ," *Ptychion Thesis (Dept. of Computer Science, Univ. of Ioannina)*, 2009.

[17] W. Hu, W. Shi, Z. Tang, Z. Zhou, and R. Eskicioglu, "Jiajia: An svm system based on a new cache coherence protocol," 02 1998.

[18] K. Kise, T. Katagiri, H. Honda, and T. Yuba, "Evaluation of the acknowledgment reduction in a software-dsm system," in *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, ser. PPAM'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 17–25. [Online]. Available: https://doi.org/10.1007/11752578_3

[19] Y. Jegou, "Implementation of page management in mome, a user-level dsm," 06 2003, pp. 479– 486.

[20] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas, "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 3–14. [Online]. Available: https://doi.org/10.1145/2749246.2749250

[21] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *2007 international workshop on Parallel symbolic computation*. Waterloo, Canada: ACM, Jul. 2007, pp. 15–23. [Online]. Available: https://hal.inria.fr/hal-00684843

[22] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1299–1308.

[23] J. Hippold and G. Rünger, "Task pool teams: A hybrid programming environment for irregular algorithms on smp clusters: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 12, p. 1575–1594, Oct. 2006.

[24] T. Rauber and G. Rünger, "Tlib-a library to support programming with hierarchical multi-processor tasks," *J. Parallel Distrib. Comput.*, vol. 65, no. 3, p. 347–360, Mar. 2005. [Online]. Available: https://doi.org/10.1016/j.jpdc.2004.10.006

[25] "Intel tbb, intel threading building blocks documentation," June 2016. [Online]. Available: https://www.threadingbuildingblocks.org/

[26] E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, pp. 173–193, 06 2011.

[27] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, p. 187–198, Feb. 2011. [Online]. Available: https://doi.org/10.1002/cpe.1631

[28] A. Zafari, E. Larsson, and M. Tillenius, "Ductteip: An efficient programming model for distributed task based parallel computing," *Parallel Computing*, 01 2018.

[29] M. Tillenius, "Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization," *SIAM Journal on Scientific Computing*, vol. 37, pp. C617–C642, 01 2015.

[30] P. E. Hadjidoukas, E. Lappas, and V. V. Dimakopoulos, "A runtime library for platform-independent task parallelism," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2012, pp. 229–236.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 158–165. [Online]. Available: https://doi.org/10.1145/125826.125925

[32] 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*, 2009, pp. 124–131.

Παραρτημα Α

Απαιτησεις λογισμικου

Για την ορθή μετάφραση (compilation) του OMPi με υποστήριξη για την καινούργια οντότητα εκτέλεσης (execution entity) που δημιουργήσαμε, έχουμε τις ακόλουθες απαιτήσεις σε λογισμικό:

- automake: `automake` v1.16
- libtool: `libtool` v2.4.6
- `autoconf`
- `bison`
- `flex`
- `hwloc, libhwloc-dev` (προαιρετικό)
- `gcc`
- OpenMPI: `openmpi` v3.0 ή μεταγενέστερη
- ArgoDSM (την δική μας τροποποιημένη έκδοση)
- TORC (την δική μας τροποποιημένη έκδοση)

Σύντομο Βιογραφικο

Ο Ηλίας Κλεφτάκης γεννήθηκε στα Ιωάννινα το 1995. Το 2013 εγγράφηκε στο προπτυχιακό πρόγραμμα σπουδών του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων, από όπου και αποφοίτησε το 2018. Την ίδια χρονιά ξεκίνησε να φοιτά στο μεταπτυχιακό πρόγραμμα σπουδών του ίδιου Τμήματος. Είναι μέλος της ερευνητικής ομάδας Παράλληλης Επεξεργασίας από το 2017. Τα επιστημονικά του ενδιαφέροντα επικεντρώνονται στα παράλληλα και κατανεμημένα συστήματα, καθώς και στα λειτουργικά συστήματα.