Εκτέλεση εργασιοκεντρικών προγραμμάτων σε συστάδες πολυπύρηνων υπολογιστών

Απόστολος Πιπέρης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

Πανεπιστήμιο Ιωαννίνων

Μάρτιος 2024

Αφιερωση

Στην Οικογένειά μου.

Κατάλογος Σχημάτων iv

Κατάλογος Αλγορίθμων v

Κατάλογος Κώδικα vi

Περίληψη vii

Abstract viii

1 Εισαγωγή 1

1.1 Σύντομη ιστορία των παράλληλων συστημάτων 1

1.2 Οργάνωση και αρχιτεκτονικές παράλληλων συστημάτων 2

1.2.1 Συστήματα κοινόχρηστης μνήμης (Shared memory) 2

1.2.2 Συστήματα κατανεμημένης μνήμης (Distributed memory) . . . 3

1.3 Προγραμματιστικά μοντέλα ανά αρχιτεκτονική 4

1.3.1 Πρότυπα κοινόχρηστης μνήμης (Shared memory) 4

1.3.2 Πρότυπα κατανεμημένης μνήμης (Distributed memory) 6

1.3.3 Εργασιοκεντρική εκτέλεση 7

1.4 Αντικείμενο Διπλωματικής Εργασίας 8

1.5 Δομή της διπλωματικής εργασίας 9

2 Επισκόπηση TORC 10

2.1 Προγραμματιστικό μοντέλο MPI 10

2.1.1 Βασικά στοιχεία του MPI 10

2.1.2 Επικοινωνίες στο MPI 12

2.2 Προγραμματιστικό μοντέλο TORC 14

2.2.1 Μοντέλο εκτέλεσης νημάτων δύο επιπέδων 15

2.2.2 Μοντέλο δημιουργίας, δρομολόγησης και συγχρονισμού εργασιών . . . 17
2.2.3 Μοντέλο ουρών εργασιών . . . 18
2.2.4 Λεπτομέρειες υλοποίησης . . . 18

3 Επεκτάσεις στο μοντέλο TORC 21
3.1 Θεωρητικό υπόβαθρο . . . 21
3.2 Λεπτομέρειες υλοποίησης κλοπής descriptor στη βιβλιοθήκη TORC . . 22
3.2.1 Επέκταση ασύγχρονης κλοπής descriptors . . . 24
3.3 Επέκταση υποσυστήματος prefetching . . . 24
3.3.1 Το επίμονο πρόβλημα της αδράνειας . . . 24
3.3.2 Γενική ιδέα του prefetching . . . 25
3.3.3 Λειτουργία του συστήματος απόφασης prefetching . . . 26
3.3.4 Αλγόριθμος απόφασης prefetching . . . 27

4 Εφαρμογές και πειραματικά αποτελέσματα 30
4.1 Προεπισκόπηση πειραματικού περιβάλλοντος . . . 30
4.1.1 Περιγραφή πειραματικών εφαρμογών . . . 31
4.2 Παραλληλοποιημένοι αλγόριθμοι ταξινόμησης 1D πινάκων . . . 33
4.2.1 Περιγραφή παράλληλου αλγορίθμου mergesort . . . 33
4.2.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα . . . 36
4.3 Παραλληλοποιημένος αλγόριθμος πολλαπλασιασμού μητρών . . . 37
4.3.1 Περιγραφή αλγορίθμου . . . 37
4.3.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα . . . 38
4.4 Παραλληλοποιημένη εφαρμογή θόλωσης εικόνας με Γκαουσιανό θόρυβο 40
4.4.1 Περιγραφή αλγορίθμου . . . 40
4.4.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα . . . 41
4.5 Αξιολόγηση . . . 43
4.5.1 Αξιολόγηση πειραματικών αποτελεσμάτων . . . 43
4.5.2 Αξιολόγηση από προγραμματιστική σκοπιά . . . 44

5 Σύνοψη 46
5.1 Σύνοψη διπλωματικής εργασίας . . . 46
5.2 Προτάσεις για μελλοντική δουλειά . . . 47

Βιβλιογραφία **48**

Α Πηγαίος κώδικας εφαρμογών **49**

2.1 Το μοντέλο εκτέλεσης της TORC 16

3.1 Διάταξη πολυεπίπεδων ουρών descriptor 23
3.2 Διάγραμμα ροής υποσυστήματος prefetching 26

4.1 Κλασσική προσέγγιση παραλληλοποίησης mergesort με επικοινωνίες . 32
4.2 Βελτιωμένο mergesort με επικοινωνίες 33
4.3 Αποτελέσματα εφαρμογής ταξινόμισης στην συστάδα του ΤΜΗΥΠ . . 36
4.4 Αποτελέσματα εφαρμογής πολλαπλασιασμού πινάκων στην συστάδα του ΤΜΗΥΠ 38
4.5 Αποτελέσματα εφαρμογής πολλαπλασιασμού πινάκων στη συστάδα ARIS 39
4.6 Εικόνα-πρότυπο για την εφαρμογή θόλωσης 41
4.7 Αποτελέσματα εφαρμογής θόλωσης εικόνας στην συστάδα του ΤΜΗΥΠ 42
4.8 Αποτελέσματα εφαρμογής θόλωσης εικόνας στην συστάδα ARIS . . . 43

Καταλογος Αλγοριθμων

3.1 Αλγόριθμος απόφασης συστήματος prefetching 29

Καταλογος Κωδικα

A.1 TORC matrix multiplication implementation 49
A.2 TORC mergesort implementation 53
A.3 TORC Gaussian blur implementation 56

Απόστολος Πιπέρης, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Μάρτιος 2024.
Εκτέλεση εργασιοκεντρικών προγραμμάτων σε συστάδες πολυπύρηνων υπολογιστών.
Επιβλέπων: Βασίλειος Β. Δημακόπουλος, Καθηγητής.

Οι συνεχώς αυξανόμενες απαιτήσεις για επεξεργαστική ισχύ σήμερα έχουν στρέψει τους διάφορους κλάδους της Πληροφορικής και της Μηχανικής Υπολογιστών προς τις τεχνικές υλοποίησης παράλληλων συστημάτων. Είτε μέσω ενός μεμονωμένου υπολογιστή με πολυπύρηνη μονάδα επεξεργασίας, είτε, στο άλλο άκρο, την χρήση υπολογιστικών συστάδων με εκατοντάδες ή και χιλιάδες κόμβους, η παράλληλη επεξεργασία εφαρμόζεται σε όλες τις σύγχρονες εφαρμογές.

Η εργασία αυτή επικεντρώνεται στην κατανεμημένη εκτέλεση σε υπολογιστικές συστάδες για επιτάχυνση εφαρμογών και δη στο μοντέλο εκτέλεσης εργασιών (task-based execution). Η δομή που ακολουθεί η εργασία είναι, αρχικά, η περιγραφή των σύγχρονων εργαλείων που χρησιμοποιούνται σε περιβάλλοντα υπολογιστικών συστάδων, καθώς και οι επεκτάσεις που πραγματοποιήθηκαν σε ένα από αυτά, την βιβλιοθήκη εργασιοκεντρικής εκτέλεσης TORC. Στη συνέχεια παρουσιάζονται ορισμένα πειράματα που χρησιμοποιήθηκαν για την επαλήθευση των επεκτάσεων που υλοποιήθηκαν στο παραπάνω εργαλείο, ακολουθούμενα από τα αποτελέσματα και τα συμπεράσματα που εξάγουμε από τα πειράματα που πραγματοποιήθηκαν, καθώς και σημεία που μπορούν να χρησιμοποιηθούν για μέλλουσα δουλειά.

Abstract

Apostolos Piperis, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, March 2024.
Task-based application execution in clusters of multicores.
Advisor: Vassilios V. Dimakopoulos, Professor.

The ever-increasing need for processing power during the modern era of computing has led the various branches of Computer Science and Engineering down the path of parallel computing. Be it through using a single computer with a multicore processing unit, or at scale, through the use of compute clusters comprising of hundreds or even thousands of nodes, parallel processing is found in all modern applications.

This thesis focuses on distributed execution for application acceleration, specifically on task-based execution. The structure followed in this work is the following: initially, we present contemporary tools that are used for execution of applications in a cluster environment and the extensions we implemented for the task-based execution library TORC. That section is followed by applications that verify the correctness and the improvements in performance of the aforementioned extensions to TORC, which are, in turn, followed by conclusions we can draw based on the experiments performed, as well as points that can be used as a base for future work.

ΚΕΦΑΛΑΙΟ 1

ΕΙΣΑΓΩΓΗ

1.1 Σύντομη ιστορία των παράλληλων συστημάτων

Η οργάνωση που έχει επικρατήσει στους σύγχρονους ηλεκτρονικούς υπολογιστές βασίζεται στο έργο και το μοντέλο που πρότεινε ο μαθηματικός John von Neumann το 1945. Εν ολίγοις, το μοντέλο αυτό ορίζει την σύσταση ενός ηλεκτρονικού υπολογιστή από πέντε μονάδες, με σημαντικότερη την μονάδα επεξεργασίας, αποτελούμενη από μια αριθμητική λογική μονάδα, όπου πραγματοποιούνται όλες οι λογικές πράξεις και το αρχείο καταχωρητών για την αποθήκευση δεδομένων. Στις απλούστερες υλοποιήσεις του μοντέλου αυτού, ο επεξεργαστής εκτελεί μία-μία τις εντολές, διαβάζοντας και τροποποιώντας τα δεδομένα στο αρχείο καταχωρητών. Αυτό το μοντέλο εκτέλεσης είναι γνωστό ως το σειριακό μοντέλο.

Στα μέσα του 20^{ού} αιώνα, οι ηλεκτρονικοί υπολογιστές γνώρισαν ραγδαία ανάπτυξη και η χρήση τους άρχισε να υιοθετείται σε πολλούς κλάδους, επιστημονικούς και μη. Έτσι, δημιουργήθηκε ένας κύκλος ανατροφοδότησης, όπου οι κατασκευαστές Η/Υ ανταγωνίζονταν μεταξύ τους για την δημιουργία του πιο ανταγωνιστικού προϊόντος σε μια συνεχώς αναπτυσσόμενη αγορά.

Το μοτίβο που ακολουθήθηκε τότε για την βελτίωση των επιδόσεων των επεξεργαστών, η οποία ακολουθείται σε μικρότερο βαθμό έως σήμερα, ήταν η αύξηση του αριθμού τρανζίστορ ανά μονάδα επιφάνειας και η αύξηση της συχνότητας ρολογιού του επεξεργαστή. Μια εμπειρική παρατήρηση οδήγησε στον γνωστό, πλέον, νόμο του Μουρ (Moore's law), ο οποίος εκφράζει την τάση διπλασιασμού του πλήθους

τρανζίστορ ενός ολοκληρωμένου κυκλώματος κάθε δύο χρόνια. Αυτή η προσέγγιση βελτίωσης άρχισε να συναντά δυσκολίες περίπου το 2004, όπου οι επεξεργαστές έφτασαν στο λεγόμενο "power wall", δηλαδή μια κατάσταση όπου η υψηλή πυκνότητα τρανζίστορ σε συνδυασμό με σχετικά υψηλές συχνότητες ρολογιού στους επεξεργαστές της περιόδου εμπόδιζε ουσιώδεις αυξήσεις στη συχνότητα, αφού αυτές επέφεραν απαγορευτικές αυξήσεις στην κατανάλωση ισχύος του επεξεργαστή και δραματική επιδείνωση των θερμικών επιδόσεών τους.

Για να λυθεί το πρόβλημα του power wall, οι σχεδιαστές επεξεργαστών στράφηκαν στην ενσωμάτωση πάνω από μίας μονάδας επεξεργασίας σε κάθε πυρήνα του επεξεργαστή, δημιουργώντας τα πρώτα πολυπύρηνα (multicore) μηχανήματα, τα οποία έχουν την δυνατότητα παράλληλης εκτέλεσης εντολών. Σήμερα, οι περισσότεροι ηλεκτρονικοί υπολογιστές είναι παράλληλοι, καθώς οι πολυπύρηνοι επεξεργαστές έχουν επιτύχει εξαιρετικές επιδόσεις ανά μονάδα ισχύος, καθιστώντας τους κατάλληλους ακόμη για ευαίσθητες εφαρμογές σε τα κατανεμημένα συστήματα.

Παρότι κάποιος θα δυσκολευτεί να βρει ένα μονοπύρηνο σύστημα στη σύγχρονη αγορά, τα μοντέλα προγραμματισμού δεν έχουν εξελιχθεί ώστε η πλήρης αξιοποίηση όλων των πυρήνων και υπολογιστικών πόρων του συστήματος να γίνεται αυτόματα. Ο παράλληλος προγραμματισμός παραμένει ένα μη-τετριμμένο πρόβλημα και οι λύσεις που υιοθετούνται δεν είναι καθολικές και διαφέρουν ανάλογα με την αρχιτεκτονική του συστήματος όπου εκτελείται ο κώδικας.

1.2 Οργάνωση και αρχιτεκτονικές παράλληλων συστημάτων

1.2.1 Συστήματα κοινόχρηστης μνήμης (Shared memory)

Τα συστήματα κοινόχρηστης μνήμης αποτελούνται από επεξεργαστές και κοινόχρηστη μνήμη η οποία είναι καθολικά προσβάσιμη από όλους τους επεξεργαστές, μέσω ενός δίκτυου διασύνδεσης για τον συγχρονισμό επεξεργαστή-μνήμης. Η μνήμη μπορεί να αποτελείται από περισσότερα του ενός τμήματα (modules) τα οποία όμως παρέχουν έναν κοινό χώρο διευθύνσεων, προσβάσιμο από όλους τους επεξεργαστές. Οι επεξεργαστές επικοινωνούν, συνεργάζονται και ανταλλάσσουν δεδομένα διαβάζοντας ή γράφοντας σε κοινόχρηστες μεταβλητές που βρίσκονται αποθηκευμένες στη μνήμη. Το δίκτυο διασύνδεσης επεξεργαστών-μνήμης μπορεί να είναι ένας απλός

δίαυλος (bus), ένα διακοπτικό δίκτυο (π.χ. crossbar) ή κάποιο δίκτυο πολλαπλών επιπέδων (π.χ. δίκτυο Δέλτα)[1].

Στην περίπτωση που το δίκτυο διασύνδεσης είναι δίαυλος και οι επεξεργαστές δεν ακολουθούν μοντέλο αφέντη-σκλάβου, τότε αναφερόμαστε σε αυτά τα συστήματα ως «συμμετρικοί πολυεπεξεργαστές» (symmetric multiprocessors, SMPs). Οι συμμετρικοί πολυεπεξεργαστές έχουν κοινή πρόσβαση σε μια κεντρική μνήμη, πράγμα που τους καθιστά συστήματα ομοιόμορφης προσπέλασης μνήμης (uniform memory access). Ως "SMP" μπορεί να χαρακτηριστεί και ο μέσος σύγχρονος εμπορικός επεξεργαστής, αφού αποτελείται από πολυπύρηνες μονάδες εκτέλεσης και ιεραρχημένες, πολυεπίπεδες κρυφές μνήμες (caches), για την αποφυγή της ποινής κύκλων που επιφέρει η πρόσβαση της κύριας μνήμης μέσω κεντρικού διαύλου.

Το προκαθορισμένο «εύρος ζώνης» (bandwidth) του διαύλου, ανεξάρτητα από το πλήθος των επεξεργαστών που είναι συνδεδεμένοι σε αυτόν, δρα ανταγωνιστικά στην κάθετη κλιμάκωση ενός συστήματος, καθώς οι περισσότεροι φυσικοί υπολογιστικοί πόροι επιφέρουν μεγαλύτερο «ανταγωνισμό» (contention) στην πρόσβαση της κεντρικής μνήμης μέσω του διαύλου, επιβραδύνοντας ακόμη περισσότερο μια ήδη σχετικά αργή διαδικασία. Επιπλέον, χρειάζεται να υλοποιηθούν όλοενα και πιο περίπλοκες πολιτικές πρόσβασης, ενημέρωσης και συγχρονισμού της μνήμης. Έτσι, για την αποδοτική υποστήριξη μεγαλύτερου αριθμού επεξεργαστών καταφεύγουμε στη χρήση κρυφής μνήμης ή άλλου τύπου δίκτυου διασύνδεσης.

1.2.2 Συστήματα κατανεμημένης μνήμης (Distributed memory)

Τα συστήματα κατανεμημένης μνήμης αποτελούνται από επεξεργαστικές οντότητες, που ονομάζονται κόμβοι και κάποιο δίκτυο διασύνδεσης το οποίο επιτρέπει επικοινωνία μεταξύ των κόμβων, παραδείγματος χάριν το δίκτυο Ethernet ή σε μεγαλύτερη κλίμακα σύγχρονα δίκτυα όπως το Infiniband. Παρόμοια με έναν μεμονωμένο υπολογιστή, κάθε κόμβος περιέχει επεξεργαστή και τοπική μνήμη. Η τοπική μνήμη κάθε κόμβου είναι απευθείας προσβάσιμη μόνο από τον επεξεργαστή του ίδιου κόμβου, αλλά είναι δυνατή η απομακρυσμένη προσπέλαση της μνήμης ενός άλλου κόμβου μέσω μεταβιβασμού μηνυμάτων. Παράλληλα συστήματα μεγαλύτερου μεγέθους μπορούν να υλοποιηθούν χρησιμοποιώντας συμμετρικούς πολυεπεξεργαστές ως κόμβους του δικτύου διασύνδεσης. Σε αυτά τα συστήματα, καθίσταται δυνατή η παροχή ενός ενιαίου χώρου διευθύνσεων. Απαιτείται, όμως, η χρήση κα-

τάλληλων πρωτοκόλλων συνοχής, τα οποία εξασφαλίζουν εγκυρότητα δεδομένων σε κάθε πρόσβαση στην μνήμη από οποιονδήποτε επεξεργαστή. Η αρχιτεκτονική που μόλις περιγράφηκε είναι γνωστή και ως «κατανεμημένη κοινόχρηστη μνήμη» (distributed shared memory), ενώ τα συστήματα αυτά ονομάζονται συστήματα «μη-ομοιόμορφης προσπέλασης μνήμης» (non-uniform memory access). Σε περίπτωση ύπαρξης ιεραρχημένων, πολυεπίπεδων κρυφών μνήμων στους κόμβους, απαιτείται η χρήση «πρωτοκόλλου συνοχής κρυφής μνήμης» (cache coherence protocol), το οποίο εξασφαλίζει πως οποιαδήποτε προσπέλαση μνήμης θα επιστρέφει την πιο πρόσφατη τιμή. Τέτοια συστήματα είναι γνωστά ως “ccNUMA” (cache-coherent NUMA). Τη σημερινή εποχή, τα περισσότερα συστήματα NUMA είναι και ccNUMA.

Ένα σύγχρονο παράδειγμα συστήματος κατανεμημένης μνήμης είναι οι υπολογιστικές συστάδες (compute clusters), δηλαδή ομάδες υπολογιστών με εξειδικευμένο υλικό, όπως επεξεργαστές πάρα πολλών πυρήνων, κάρτες γραφικών γενικού σκοπού (General Purpose Graphics Processing Unit) ή λοιπούς επιταχυντές. Οι υπολογιστές αυτοί διασυνδέονται μέσω δικτύου πολύ μεγάλου εύρους ζώνης και πολύ χαμηλής καθυστέρησης (hand bandwidth, ultra low latency networks). Οι συστάδες υπολογιστών χρησιμοποιούνται ευρέως λόγω της διαθεσιμότητας δικτύων διασύνδεσης υψηλών επιδόσεων όπως το gigabit Ethernet, το Infiniband, κ.ά. Το πλήθος των κόμβων και το πλήθος των επεξεργαστών στα συστήματα κατανεμημένης μνήμης μπορεί να φτάσει τις εκατοντάδες χιλιάδες και κάποια εκατομμύρια αντίστοιχα, σε αντίθεση με τα συστήματα κοινόχρηστης μνήμης όπου το πλήθος των επεξεργαστών περιορίζεται σε μερικές δεκάδες. Η επιλογή σωστής τοπολογίας στο δίκτυο είναι καθοριστικός παράγοντας για την σωστή κλιμάκωση ενός συστήματος κατανεμημένης μνήμης.

1.3 Προγραμματιστικά μοντέλα ανά αρχιτεκτονική

1.3.1 Πρότυπα κοινόχρηστης μνήμης (Shared memory)

Ο προγραμματισμός των συστημάτων κοινόχρηστης μνήμης πραγματοποιείται μέσω χρήσης νημάτων ή ινών (fibers). Τα νήματα αποτελούν ξεχωριστές μονάδες εκτέλεσης, με δική τους στοίβα και μετρητή προγράμματος, τα οποία έχουν πρόσβαση σε έναν χώρο διευθύνσεων που είναι κοινός σε όλα τα νήματα. Οι ίνες, αποτελούν ένα

είδους νήματος που υλοποιεί συνεργατική πολυεργασία, δηλαδή ο «δρομολογητής» (scheduler) του λειτουργικού συστήματος δεν μπορεί να παύσει την εκτέλεσή τους και να ξεκινήσει άλλη διεργασία, αποφεύγοντας έτσι τα ακριβά context switches, δηλαδή την ακύρωση της στοίβας ενός προγράμματος και επανεκκίνησης ενός άλλου. Ανταυτού αποθηκευέται όλο το "state" και μπορεί να επανεκκινήσει χωρίς ακύρωση μετρητών ή στοίβας.

Η εκτέλεση κώδικα σε πολυνηματικό περιβάλλον δημιουργεί προβλήματα εγκυρότητας των δεδομένων, όταν αυτά είναι κοινόχρηστα, οδηγώντας συχνά σε *συνθήκες ανταγωνισμού* (race conditions), δηλαδή την κατάσταση όταν πάνω από ένα νήματα προσπαθούν να κάνουν ταυτόχρονη προσπέλαση μιας διεύθυνσης στη μνήμη. Η συμπεριφορά του προγράμματος όταν υπάρχει συνθήκη ανταγωνισμού είναι μη ντετερμινιστική και ο προγραμματιστής δεν μπορεί να περιμένει επαναλήψιμη συμπεριφορά. Δύο τρόποι επίλυσης του φαινομένου των συνθηκών ανταγωνισμού είναι η «ατομικότητα» (atomicity) και ο «αμοιβαίος αποκλεισμός» (mutual exclusion). Χρησιμοποιώντας ατομικές εντολές, που υλοποιούνται σε επίπεδο υλικού, ή δομές αμοιβαίου αποκλεισμού, εξασφαλίζεται η μοναδική πρόσβαση του εκάστοτε νήματος σε κάθε προσπέλαση κοινόχρηστων δεδομένων για εγγραφή.

Η πιο κοινή βιβλιοθήκη πολυνηματισμού και παράλληλης εκτέλεσης είναι η βιβλιοθήκη "POSIX threads" που είναι παρούσα σε όλα τα συστήματα συμβατά με το πρότυπο POSIX. Η pthreads παρέχει την πλήρη γκάμα υποδομών για την διαχείριση πολυνηματικής εκτέλεσης, όπως ρουτίνες διαχείρισης νημάτων (έναρξη, τερματισμός, παύση, κλπ.), κλειδαριές αμοιβαίου αποκλεισμού (mutexes), μεταβλητές συνθήκης (condition variables), αλλά και κλήσεις συγχρονισμού. Η ευθύνη σωστής έναρξης, ανάθεσης εργασίας, συγχρονισμού και σωστής συλλογής αποτελεσμάτων από τα τα νήματα ανατίθεται εξολοκλήρου στον προγραμματιστή, δίνοντάς του έτσι λεπτό έλεγχο στην δομή και, κατά επέκταση, στις επιδόσεις του προγράμματός του, αλλά δημιουργείται με αυτόν τον τρόπο μεγαλύτερη επιφάνεια για σφάλματα.

Για τη διευκόλυνση του προγραμματισμού, έχουν δημιουργηθεί βιβλιοθήκες πολυνηματισμού υψηλότερου επιπέδου όπως το OpenMP (Open Multi-Processing) [2]. Το OpenMP είναι μία «διεπαφή προγραμματισμού εφαρμογών» (application programming interface) η οποία αποτελείται από οδηγίες (directives) προς τον μεταφραστή, ρουτίνες και μεταβλητές περιβάλλοντος (environment variables) που διευκολύνουν την σύνταξη πολυνηματικού κώδικα για συστήματα κοινόχρηστης μνήμης. Ένα από τα μεγαλύτερα πλεονεκτήματα του OpenMP είναι ότι προσφέρει δυνατό-

τητα παραλληλοποίησης υπάρχοντος σειριακού κώδικα, χωρίς να απαιτείται η εκτενής τροποποίησή του. Αυτό επιτυγχάνεται με την προσθήκη οδηγιών, που υλοποιούνται με την μορφή pragmas που υποστηρίζουν ορισμένοι μεταφραστές της γλώσσα C/C++ και αγνοούνται από μεταφραστές που δεν υποστηρίζουν την OpenMP. Το runtime component του OpenMP αναλαμβάνει επίσης την περίπλοκη διαδικασία συντονισμού και διαχείρισης των νημάτων, απλοποιώντας σημαντικά την πολυπλοκότητα συγγραφής πολυνηματικού κώδικα, επιτρέποντας και σε λιγότερο έμπειρους προγραμματιστές να επιταχύνουν την εφαρμογή τους.

1.3.2 Πρότυπα κατανεμημένης μνήμης (Distributed memory)

Ο προγραμματισμός των συστημάτων κατανεμημένης μνήμης βασίζεται στη μεταβίβαση μηνυμάτων μεταξύ αυτόνομων διεργασιών (προγράμματα υπό εκτέλεση) οι οποίες βρίσκονται σε διαφορετικούς κόμβους. Λόγω της αρχιτεκτονικής του συστήματος, δεν είναι δυνατή η ύπαρξη πραγματικά κοινόχρηστων μεταβλητών, αλλά μπορούν οι κόμβοι να ενημερώνουν τους υπόλοιπους για την αλλαγή μιας «κοινής» μεταβλητής μέσω μεταβίβασης μηνυμάτων. Όπως γίνεται εύκολα αντιληπτό, ο συγχρονισμός των κόμβων δεν είναι ιδιαίτερα εύκολος και απαιτούνται διάφορα τακτικές διαχωρισμού λειτουργιών εκτέλεσης από τις λειτουργίες επικοινωνίας, έτσι ώστε να είναι πιο προφανής και, θεωρητικά, λιγότερο προβληματικός ο συγχρονισμός των κόμβων. Ο χρήστης πρέπει να συνθέσει σωστά τα μηνύματα επικοινωνίας, π.χ. επιλογή σωστού αποστολέα και παραλήπτη, αλλά και να έχει κατά νου το μοντέλο επικοινωνίας. Τα δύο μοντέλα επικοινωνίας είναι το σύγχρονο και το ασύγχρονο. Στο σύγχρονο μοντέλο, μια διεργασία μπλοκάρει μέχρι να παραλάβει τα δεδομένα που απαιτεί, ενώ μια ασύγχρονη επιχειρεί να λάβει τα δεδομένα, αλλά συνεχίζει την εκτέλεση ανεξαρτήτως αν παρέλαβε ένα μήνυμα επιτυχώς. Η επικοινωνία μεταξύ των κόμβων, όμως, είναι ακριβή και ο προγραμματιστής πρέπει να προνοήσει έτσι ώστε να ελαχιστοποιήσει τις ανάγκες για επικοινωνία μεταξύ των κόμβων, κρατώντας τα κοινόχρηστα δεδομένα στο ελάχιστο δυνατό επίπεδο.

Το πιο κοινό πρωτόκολλο μεταβίβασης μηνυμάτων είναι το MPI (Message Passing Interface) [3], με τρεις γνωστές υλοποιήσεις: το OpenMPI, το MPICH, και το Intel MPI. Το MPI προσδίδει δυνατότητες μεταβιβασμού μηνυμάτων στις γλώσσες C/C++ και Fortran, επιτρέποντας την ανάπτυξη εφαρμογών μεγάλης κλίμακας. Ένα σημαντικό στοιχείο του MPI είναι η απόκρυψη λεπτομερειών υλοποίησης από

τον προγραμματιστή, επιτρέποντας ένα API υψηλού επιπέδου που δεν απαιτεί περαιτέρω γνώσεις προγραμματισμού πιο εξεζητημένων θεμάτων, όπως ο δικτυακός προγραμματισμός.

1.3.3 Εργασιοκεντρική εκτέλεση

Στο πλαίσιο της τρέχουσας εργασίας, κρίνεται ιδιαίτερα σημαντικό να δοθεί έμφαση στην «εργασιοκεντρική» εκτέλεση στις υπολογιστικές συστάδες. Το «κλασσικό» μοντέλο παραλληλοποίησης προβλέπει την τμηματοποίηση των δεδομένων που επεξεργάζεται μία εφαρμογή, παραδείγματος χάριν τον διαχωρισμό ενός δισδιάστατου πίνακα σε μικρότερους υποπίνακες και στη συνέχεια τον διαμοιρασμό των υποτμημάτων αυτών σε όλους τους κόμβους της υπολογιστικής συστάδας υπό τον έλεγχο μιας διεργασίας ανά κόμβο.

Αντίθετα, σε μια εργασιοκεντρική προσέγγιση γίνεται διαχωρισμός της εφαρμογής σε ανεξάρτητες μεταξύ τους εργασίες, όπου η κάθε μια ανατίθεται σε μία μονάδα εκτέλεσης, παραδείγματος χάριν ένα νήμα, η οποία αναλαμβάνει τη δρομολόγηση και την εκτέλεσή της. Ένα εργασιοκεντρικό μοντέλο εκτέλεσης χρειάζεται ένα μέσο scheduling των εργασιών στο επίπεδο της μονάδας εκτέλεσης. Μια συνηθισμένη πρακτική για την δρομολόγηση εργασιών είναι η υλοποίηση ενός συστήματος ουρών, όπου τοποθετούνται για κάθε μονάδα εκτέλεσης οι εργασίες και εκτελούνται σύμφωνα με την πολιτική της ουράς. Παρά την πιο επίπονη διαδικασία παραλληλοποίσης της εφαρμογής σε σχέση με το κλασσικό μοντέλο δεδομένων, η εργασιοκεντρική εκτέλεση κερδίζει περισσότερη ευελιξία στην χρήση περαιτέρω μεθόδων παραλληλοποίησης. Στο κλασσικό μοντέλο εκτέλεσης, η όποια περαιτέρω παραλληλοποίηση στον τελικό κόμβο βρίσκεται υπό την διακριτική ευχέρεια της βιβλιοθήκης με την οποία υλοποιείται η εφαρμογή. Σε αντίθεση, μια εργασιοκεντρική εκτέλεση επιτρέπει επιπλέον παραλληλοποίηση διά προγραμματιστικής υλοποίησης, είτε με την χρήση πολυνηματικής εκτέλεσης, ή την δημιουργία επιπλέον υπο-εργασιών κατά τη διάρκεια εκτέλεσης μιας εργασίας.

Στον χώρο της εργασιοκεντρικής εκτέλεσης, η βιβλιοθήκη TORC [4] αποτελεί μια πρωτότυπη προσέγγιση υλοποίησης. Το σύστημα εκτέλεσής της χρησιμοποιεί ένα μοντέλο επόπτη-εργάτη, όπου δημιουργείται μια διεργασία-επόπτης στον κάθε κόμβο της συστάδας κατά την εκτέλεση και, στη συνέχεια, συναρτήσεις από τον πηγαίο κώδικα της εφαρμογής, τις οποίες ο χρήστης έχει επισημάνει, μέσω των υπο-

δομών που παρέχει η βιβλιοθήκη TORC, ως ικανές να δράσουν ως διεργασίες δρουν ως εικονικές διεργασίες υπό την επίβλεψη της διεργασίας-επόπτη. Ο επόπτης χρησιμοποιεί ένα σύστημα πολυεπίπεδων ουρών για την αποθήκευση των εργασιών, με την κάθε ουρά να έχει διαφορετικές ιδιότητες όσον αφορά την ορατότητά τους από άλλες διεργασίες. Στο πλαίσιο της εργασίας μας ενδιαφέρουν οι δημόσιες ουρές, οι οποίες, όντας ορατές από όλες τις εικονικές διεργασίες που δημιουργεί ο επόπτης, παρέχουν την δυνατότητα κλοπής στοιχείων της ουράς μιας άλλης διεργασίας. Η διαδικασία που μόλις περιγράφηκε ονομάζεται “work stealing” και πραγματοποιείται σε περίπτωση που η ιδιωτική ουρά παραμείνει άδεια πριν το πέρας της εκτέλεσης του προγράμματος. Όταν ολοκληρωθεί η εκτέλεση, η εικονική διεργασία προωθεί τα αποτελέσματα του έργου της στον επόπτη, ο οποίος συνθέτει κατάλληλα τα αποτελέσματα από όλες τις θυγατρικές εργασίες τους και τα προωθεί στον κεντρικό κόμβο.

1.4 Αντικείμενο Διπλωματικής Εργασίας

Η εργασία αυτή επικεντρώνεται στη βιβλιοθήκη TORC και την χρήση της για την εκτέλεση εργασιοκεντρικών προγραμμάτων σε υπολογογιστικές συστάδες. Συγκεκριμένα, μελετήθηκε το ζήτημα της πολιτικής “work stealing”, η οποία χρησιμοποιείται για την αύξηση των επιδόσεων σε ανομοιογενείς συστάδες, δηλαδή συστάδες όπου υπάρχουν διαφορετικοί υπολογιστές-κόμβοι, με άνισες επιδόσεις. Εξερευνήθηκαν θέματα πρακτικότητας του work stealing, καθώς και ευκολίας εύρεσης και υλοποίησης νέων πολιτικών που εμπλουτίζουν τις υπάρχουσες πολιτικές της βιβλιοθήκης.

Το αντικείμενο της έρευνας στράφηκε, έπειτα, στο παρεμφερές, αλλά σχετικά άγνωστο αντικείμενο του “prefetching”, δηλαδή του ασύγχρονου, προληπτικού work stealing, έτσι ώστε να μην αδειάζει ποτέ η ουρά εργασιών στους κόμβους που εκτελείται το πρόγραμμα. Σημαντικό κομμάτι της έρευνας, άρα και αυτής της εργασίας αφιερώθηκε στους παράγοντες που πρέπει να ληφθούν υπόψη έτσι ώστε ο scheduler να μην διακόπτει συχνά την κύρια λειτουργία εκτέλεσης για να πραγματοποιήσει prefetching.

Τέλος, πραγματοποιήθηκαν κάποιες αλλαγές στον scheduler, έτσι ώστε να διευκολυνθεί οποιαδήποτε μέλλουσα επέκταση στις πολιτικές εκτέλεσης της βιβλιοθήκης

TORC και στο σύστημα καταγραφής στατιστικών, το οποίο δίνει πλέον μια ταυτόχρονα πιο πλήρη, αλλά και πιο λεπτή, εικόνα στον προγραμματιστή για τη κάθε λειτουργία που εκτελεί η βιβλιοθήκη στο runtime.

1.5 Δομή της διπλωματικής εργασίας

Ακολουθεί η δομή της διπλωματικής εργασίας βάσει κεφαλαίων:

- Κεφάλαιο 2: Επισκόπηση μοντέλου TORC και MPI
  Παρουσίαση του μοντέλου MPI και της βιβλιοθήκης TORC, στην οποία βασίστηκε το έργο της παρούσης εργασίας.

- Κεφάλαιο 3: Επεκτάσεις στο μοντέλο TORC
  Παρουσίαση των επεκτάσεων που υλοποιήθηκαν στη βιβλιοθήκη, βάσει ιδεών που προέκυψαν κατά τη χρήση της.

- Κεφάλαιο 4: Εφαρμογές
  Εφαρμογές που χρησιμοποιήθηκαν για να αξιολογηθούν οι αλλαγές που έγιναν στην TORC.

- Κεφάλαιο 5: Πειραματική ανάλυση και συμπεράσματα
  Αξιολόγηση των αποτελεσμάτων των εφαρμογών που παρουσιάζονται στο κεφάλαιο 4 σε διάφορα συστήματα, με διάφορες παραμέτρους εκτέλεσης

- Κεφάλαιο 6: Αξιολόγηση μοντέλου TORC
  Σύντομη αξιολόγηση της βιβλιοθήκης, βάσει της ευκολίας χρήσης της για παραλληλισμό βασισμένο σε tasks, καθώς και της εσωτερικής δομής κώδικα.

- Κεφάλαιο 7: Σύνοψη εργασίας

ΚΕΦΑΛΑΙΟ 2

ΕΠΙΣΚΟΠΗΣΗ ΒΙΒΛΙΟΘΗΚΗΣ TORC

2.1 Προγραμματιστικό μοντέλο MPI

Όπως αναφέρθηκε στην ενότητα 1.3.2, το MPI αποτελεί ένα πρότυπο υλοποίησης (standard) για μια βιβλιοθήκη προγραμματισμού κατανενεμημένων συστημάτων με μεταβίβαση μηνυμάτων. Κατά την υλοποίηση αυτής της εργασίας χρησιμοποιήθηκε κυρίως η υλοποίηση MPICH [5], αλλά και η υλοποίηση OpenMPI [6] για επαλήθευση ορθότητας.

Το πρότυπο και οι υλοποιήσεις του παρέχουν στον χρήστη πλήρεις δομές δεδομένων και υποδομές επικοινωνίας μεταξύ κόμβων. Οι ρουτίνες επικοινωνίας επιτρέπουν την διαχείριση της αποστολής και αποδοχής μηνυμάτων, με σύγχρονο ή ασύγχρονο τρόπο. Υπάρχει, επίσης, δυνατότητα one-to-many επικοινωνίας, δηλαδή αποστολή μηνυμάτων από έναν κόμβο προς πολλούς.

2.1.1 Βασικά στοιχεία του MPI

Για την χρήση της βιβλιοθήκης προαπαιτείται να ρυθμιστεί σωστά το runtime του MPI. Χρησιμοποιώντας την συνάρτηση MPI_Init, η χρήστης αρχικοποιεί το περιβάλλον του MPI, αλλά και ρυθμίζει τις δομές επικοινωνίας μεταξύ των διεργασιών που δημιουργεί το MPI κατά την εκτέλεση του προγράμματος.

Δύο σημαντικές παράμετροι στο μοντέλο εκτέλεσης και επικοινωνίας του MPI

είναι ο βαθμός (rank) και το μέσο επικοινωνίας (communicator). Το μοντέλο εκτέλεσης δίνει στον χρήστη την δυνατότητα ομαδοποίησης των διεργασιών που δημιουργεί το runtime του MPI και την ρύθμιση του τρόπου επικοινωνίας μεταξύ των ομάδων αυτών. Εντός των ομάδων, το MPI χρησιμοποιεί το rank ως αναγνωριστικό για κάθε διεργασία-μέλος της ομάδας. Το εύρος τιμών του βαθμού είναι 0 έως $N-1$, όπου N το πλήθος των διεργασιών στην ομάδα. Η βασική ρύθμιση του MPI δημιουργεί μόνο μία ομάδα διεργασιών και χρησιμοποιεί μέσο επικοινωνίας. Στην περίπτωση που επιλέγει να χωρίσει τις διεργασίες σε πάνω από μία ομάδα, το μοντέλο επικοινωνίας αλλάζει, αφού οι διεργασίες μπορούν να επικοινωνήσουν άμεσα με τα υπόλοιπα μέλη της ομάδας, αλλά χρειάζεται να χρησιμοποιήσουν το μέσο επικοινωνίας για να διαβιβάσουν ή να λάβουν μηνύματα από διεργασίες άλλης ομάδας. Για να είναι πιο σαφές στον προγραμματιστή πως διαμοιράζεται το έργο στις διεργασίες που δημιουργεί το runtime, το MPI παρέχει ρουτίνες αναγνώρισης, όπως οι MPI_Comm_rank και η MPI_Comm_size, που επιστρέφουν το βαθμό, την ομάδα εκτέλεσης και άλλα στοιχεία από κάθε διεργασία. Όπως μπορεί να γίνει εύκολα αντιληπτό, η όποια τυχούσα ρύθμιση στις ομάδες εκτέλεσης πρέπει να γίνει αμέσως μετά την κλήση της MPI_Init, για να αποτραπούν σφάλματα συντονισμού στο πρόγραμμα.

Όσον αφορά το μοντέλο δεδομένων, το MPI υποστηρίζει όλους τους βασικούς τύπους δεδομένων των C/C++, δηλαδή ακεραίους διαφόρων μεγεθών και αριθμούς κινητής υποδιαστολής με μονή ή διπλή ακρίβεια, καθώς και πλειάδες από αυτούς τους τύπους δεδομένων. Λόγω της έλλειψης εγγενών μηχανισμών serialization ή μεταπρογραμματισμού με ασφάλεια τύπων στις γλώσσες C και C++, το MPI απαιτεί από τον χρήστη να ορίζει τον τύπο δεδομένων που αποστέλλει ή παραλαμβάνει σε μια κλήση επικοινωνίας. Στην περίπτωση που μεταχειρίζεται πλειάδα, απαιτεί και το μέγεθος του buffer που αποθηκεύονται τα δεδομένα. Στην περίπτωση που ο χρήστης παρέχει λάθος τύπο στην κλήση, το runtime του MPI δεν εγγυάται την συμπεριφορά του προγράμματος, καθώς το λάθος όρισμα στην ρουτίνα επικοινωνίας μπορεί να οδηγήσει στην ανάγνωση λάθος αριθμού bytes από το μήνυμα, ή να γίνει λάθος στον τρόπο διάρθρωσης των δεδομένων στην μνήμη.

2.1.2 Επικοινωνίες στο MPI

Έχοντας γνώση των παραπάνω υποσυστημάτων του MPI, ο προγραμματιστής μπορεί να στραφεί στο μοντέλο επικοινωνίας. Το πρότυπο προσφέρει στον χρήστη σύγχρονες και ασύγχρονες επιλογές για επικονωνία point-to-point, δηλαδή από μια διεργασία σε μια άλλη και συλλογική επικοινωνία, δηλαδή από μία διεργασία προς πολλές. Με τους όρους «σύγχρονη» και «ασύγχρονη» επικοινωνία διαφοροποιούμε την συμπεριφορά ενός συστήματος στην περίπτωση που αποστέλλει ένα μήνυμα σε μια χρονική στιγμή κατά την οποία ο αποδέκτης του μηνύματος δεν είναι έτοιμος να ανταποκριθεί καταλλήλως. Ένα σύγχρονο σύστημα εκτελεί διεργασίες διαδοχικά, ενώ σε ένα ασύγχρονο μπορούν να εκτελούνται ταυτόχρονα πάνω από μια διεργασία. Για παράδειγμα, αν σε ένα σύγχρονο σύστημα μια διεργασία A αποστέλλει ένα μήνυμα στην διεργασία B, πρέπει να αναμείνει την ολοκλήρωση της διεργασίας B ώστε να λάβει απάντηση και να συνεχίσει την εκτέλεσή της. Αντίθετα, σε ένα ασύγχρονο σύστημα, η διεργασία A μπορεί να συνεχίσει την εκτέλεσή της όσο περιμένει απόκριση από την διεργασία B. Ένας άλλος διαχωρισμός που πραγματοποιείται στην υλοποίηση των επικοινωνιών, συνήθως σε συνάρτηση με το σύγχρονο ή ασύγχρονο μοντέλο εκτέλεσης, είναι το αν αυτές είναι εμποδιστικές ή μη εμποδιστικές. Ένα εμποδιστικό σύστημα παγώνει την εκτέλεση και αναμένει απάντηση από τον στόχο, μέχρι αυτός να είναι έτοιμος και συνεχίζει την εκτέλεση μετά την επιτυχή απάντηση. Αντίθετα, ένα ασύγχρονο, ή μη εμποδιστικό, σύστημα στη πράξει «δοκιμάζει» να λάβει απάντηση επιτόπου, αλλά συνεχίζει κανονικά την εκτέλεση στην περίπτωση που ο παραλήπτης δεν ανταποκριθεί αμέσως. Όπως είναι προφανές, και οι δύο διαφοροποιήσεις στις επικοινωνίες οδηγούν σε μεταβολή στον τρόπο που ο προγραμματιστής καλείται να διαρθρώσει τον κώδικα της εφαρμογής του. Η προσεκτική χρήση ασύγχρονων επικοινωνιών μπορεί να οδηγήσει σε αισθητά μεγαλύτερη απόδοση λόγω καλύτερης εκμετάλλευσης του χρόνο αναμονής απάντησης σε ένα σενάριο χρήσης εμποδιστικής κλήσεως, ο οποίος είναι άγνωστος a priori.

Όλες οι κλήσεις, είτε point-to-point, είτε συλλογικές υποστηρίζουν σύγχρονες και ασύγχρονες εκδοχές. Οι κύριες κλήσεις για point-to-point επικονωνία είναι οι MPI_Send και MPI_Recv. Οι δύο κλήσεις αυτές δηλώνουν την πρόθεση του συστήματος να αποστείλει ή να δεχτεί, αντίστοιχα, έναν buffer συγκεκριμένου μεγέθους και τύπου δεδομένων από μία άλλη διεργασία. Όπως δηλώνει ο όρος point-to-point, οι

κλήσεις αυτές είναι αυστηρά από μία διεργασία σε μία μόνο άλλη. Το πρότυπο του MPI ορίζει τρεις διαφορετικές κλήσεις σύγχρονης αποστολής, πέραν της MPI_Send. Αρχικά, την MPI_Ssend, όπου κατά την κλήση, ο αποστολέας ενημερώνει τον δέκτη με ένα ready to "send" σήμα και μπλοκάρει μέχρι ο δέκτης να επιβεβαιώσει την επιτυχή αντιγραφή του buffer του αποστολέα. Αν ο χρήστης επιθυμεί να χρησιμοποιήσει μη εμποδιστικές κλήσεις, προσφέρεται η MPI_Iend και οι MPI_Wait. Αξίζει να σημειωθεί πως το πρότυπο του MPI δεν ορίζει την συμπεριφορά της βασικής MPI_Send, αλλά το αφήνει πάνω στον υλοποιητή. Παραδείγματος χάριν, η υλοποίηση της Intel χρησιμοποιεί σύγχρονη επικοινωνία για μικρά μηνύματα και ασύγχρονη για μηνύματα μεγαλύτερου μήκους όπου το κόστος του τοπικού αντιγράφου είναι υψηλό.

Στις συλλογικές επικοινωνίες, το MPI προσφέρει δύο κατηγορίες κλήσεων: επικοινωνίας και συλλογικών πράξεων. Όταν ο προγραμματιστής θέλει να αποστείλει μήνυμα σε όλες υπόλοιπες διεργασίες της ομάδα εκτέλεσης από την root process της ομάδας, μπορεί να χρησιμοποιηθεί η MPI_Bcast. Για διαβιβασμό μηνύματος προς όλες τις διεργασίες της ομάδας με τρόπο αγνωστικό προς το rank της διεργασίας, προσφέρεται η MPI_Scatter. Στην άλλη πλευρά της επικοινωνίας έχουμε δύο κλήσεις για αποδοχή ενός μηνύματος. Η MPI_Gather χρησιμοποιείται για να λάβει ένα συλλογικό μήνυμα τοπικά μόνο η συγκεκριμένη διεργασία όπου έγινε η κλήση. Στην περίπτωση που θέλουμε ένα μήνυμα να μεταδοθεί σε όλες τις διεργασίες μιας ομάδας εκτέλεσης, προσφέρεται η MPI_Allgather. Το ενδιαφέρον τμήμα στις ομαδικές επικοινωνίες είναι η δυνατότητες για συλλογικές πράξεις που προσφέρει το πρότυπο του MPI. Μέσω της κλήσης MPI_Reduce και αντίστοιχα MPI_Allreduce για την μετάδοση σε όλες τις διεργασίες της ομάδας, μπορεί να λάβει αποτελέσματα από MPI_Scatter ή MPI_Bcast και αντί να λάβει το μήνυμα αυτούσιο, εφαρμόζει κάποιο reduction operation στον buffer, το οποίο μπορεί να είναι αριθμητική πράξη όπως πρόσθεση ή πολλαπλασιασμός ή και λογική πράξη, με λογικό bitwise OR.

Όταν χρησιμοποιείται μοντέλο συλλογικών επικοινωνιών, δημιουργούνται τμήματα κώδικα τα οποία μπορούν να δημιουργήσουν συνθήκες ανταγωνισμού ή αποσυγχρονισμού, αφενός λόγω της ασύγχρονης φύσεως των συλλογικών επικοινωνιών και αφετέρου λόγω τυχόντων καθυστερήσεων σε κάθε κόμβο εκτέλεσης. Έτσι, το πρότυπο του MPI πρέπει να παρέχει ένα τρόπο στον προγραμματιστή να διακριτοποιεί κάθε «βήμα» της εκτέλεσης της εφαρμογής του, ώστε να διασφαλίζεται η εγκυρότητα δεδομένων. Η λύση που παρέχεται είναι η κλήση MPI_Barrier, διά της

οποίας ο προγραμματιστής μπορεί να δημιουργήσει ένα σημείο συγχρονισμού, στο οποίο το πρόγραμμα αναμένει μέχρι την ολοκλήρωση όλων των παραπάνω συλλογικών επικοινωνιών και πράξεων ανά το δίκτυο, εξασφαλίζοντας πως η κάθε διεργασία έχει πλέον ενημερωμένα και έγκυρα δεδομένα και μπορεί να προβεί σε περαιτέρω υπολογισμούς με ασφάλεια.

2.2 Προγραμματιστικό μοντέλο TORC

Η βιβλιοθήκη TORC είναι μια βιβλιοθήκη για κατανεμημένη εκτέλεση εργασιών υψηλών επιδόσεων σε υπολογιστικές συστάδες, εκμεταλλευόμενη τις δυνατότητες των σύγχρονων πολυπύρηνων επεξεργαστών που βρίσκονται, πλέον, στους κόμβους των περισσότερων συστάδων.

Η TORC αποτελεί παράδειγμα βιβλιοθήκης που επιτυγχάνει παραλληλισμό μέσω κατανεμημένων εργασιών, καθώς κάθε μονάδα εκτέλεσης (εργάτης - worker) εκτελεί μεμονωμένες εργασίες (tasks), χωρίς να του στερείται η δυνατότητα εκκίνησης νέων διεργασιών. Το μοντέλο αυτό είναι ιδιαίτερα φιλικό προς συστήματα κατανεμημένης μνήμης, αλλά και σε ανομοιογενείς κόμβους σε μια υπολογιστική συστάδα, δηλαδή κόμβους των οποίων τα τεχνικά χαρακτηριστικά αποκλίνουν μεταξύ τους, βάσει παρουσίας ή μη διάφορων επιταχυντών όπως κάρτες γραφικών γενικού σκοπού (GPGPU - General Purpose Graphics Processing Unit) ή διαφορά στις επιδόσεις του επεξεργαστή κάθε κόμβου. Η TORC χρησιμοποιεί το MPI για τον συγχρονισμό των workers σε όλο το δίκτυο εκτέλεσης.

Η χρήση του MPI ως αυτούσια βιβλιοθήκη για τον προγραμματισμό κατανεμημένων συστημάτων παρουσιάζει αισθητή δυσκολία, λόγω της χαμηλού επιπέδου φύσεως των ρουτίνων επικοινωνίας που προσφέρει στον προγραμματιστή. Οι αφαιρέσεις και η πρακτική διεπαφή προγραμματισμού εφαρμογής (API - Application Programming Interface) που προσφέρει μια βιβλιοθήκη για εργασίες (tasking library) προσπαθούν να μειώσουν την τριβή που δημιουργείται κατά τον προγραμματισμό με μεταβίβαση μηνυμάτων και δη η βιβλιοθήκη TORC προσπαθεί να επιτύχει αυτόν τον στόχο, δίχως συμβιβασμό στον τομέα των επιδόσεων.

Η πρακτικότητα, όμως, παραμένει πάντα μια σχετική έννοια. Παρά την αισθητή βελτίωση σε σχέση με τη αυτούσια χρήση του MPI, η TORC παραμένει βιβλιοθήκη σχετικά χαμηλού επιπέδου. Ο χρήστης καλείται να διαχειριστεί μόνος του την δη-

μιουργία και τον συγχρονισμό των task, αλλά και την παραμετροποίηση του runtime της TORC. Αν συγκρίνουμε την βιβλιοθήκη TORC με ένα API όπως το OpenMP, όπου ο χρήστης με τη χρήση μερικών απλών "pragmas" και στοιχειώδη κατανόηση τεχνικών παραλληλοποίησης εκτέλεσης μπορεί να επιτύχει σημαντικές επιταχύνσεις στον κώδικά του, γίνεται προφανής η διαφορά στην παραγωγικότητα με την TORC. Για αυτό, η TORC πέρα από τη χρήση ως τελική βιβλιοθήκη χρήστη, χρησιμοποιείται και ως βάση για υψηλότερου επιπέδου προγραμματιστικά συστήματα όπως π.χ. ο HOMPI [7].

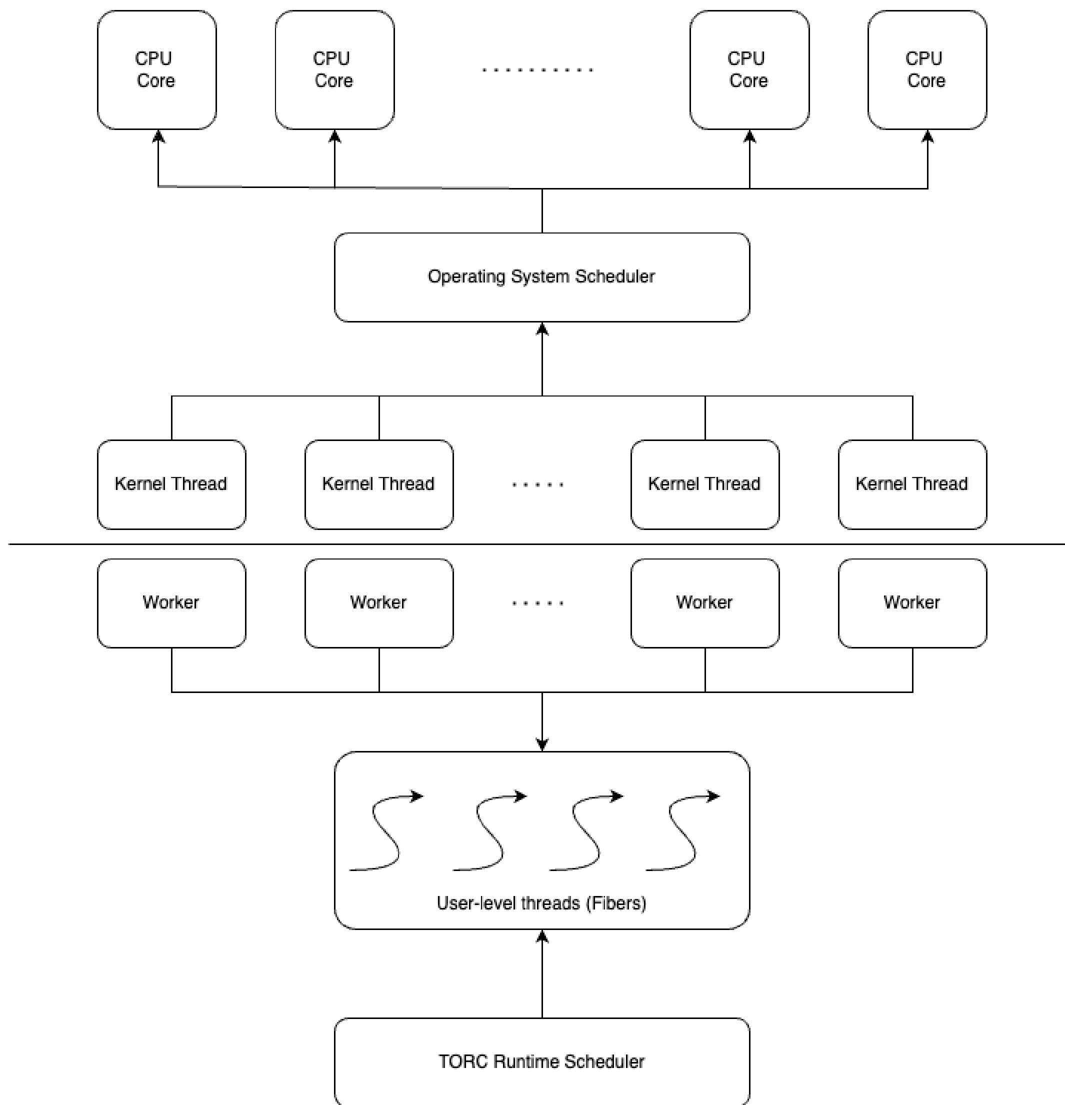
Η υλοποίηση της TORC στηρίζεται σε τρία μεγάλα υποσυστήματα: το μοντέλο εκτέλεσης νημάτων δύο επιπέδων, το μοντέλο πολυεπίπεδων ουρών και το μοντέλο δημιουργίας και δρομολόγησης εργασιών. Ακολουθούν υποενότητες που περιγράφουν το καθένα από αυτά πιο αναλυτικά.

2.2.1 Μοντέλο εκτέλεσης νημάτων δύο επιπέδων

Η TORC στηρίζεται σε νήματα επιπέδου χρήστη (user-level threads), γνωστά και ως ίνες (fibers) για την εκτέλεση των εργασιών. Οι ίνες, οι οποίες αποκαλούνται και νήματα συνεργατικής δρομολόγησης σε επίπεδο χρήστη (user mode cooperatively scheduled threads), αποτελούν υπομονάδες εκτέλεσης σε επίπεδο χρήστη για μεγαλύτερη ευκολία στην διαδικασία της δρομολόγησης εργασιών. Λόγω της χρήσης στοίβας για την αναπαράσταση δεδομένων και κατάστασης, η δρομολόγηση των ινών μπορεί να γίνει με λιγότερο ακριβό τρόπο, αφού σε μία ίνα ορίζονται σημεία παύσης και επανεκκίνησης της εκτέλεσης. Εκεί κρύβεται το μεγάλο πλεονέκτημα των ινών σε σύγκριση με τα νήματα, αφού η παύση και εναλλαγή νημάτων επίπεδο πυρήνα (context switch) είναι μια πολύ ακριβή διαδικασία.

Η TORC συγκεκριμένα δεσμεύει ένα νήμα επιπέδου χρήστη ανά εργασία και δημιουργεί μια στοίβα εκτέλεσης αποτελούμενη από τα δεδομένα της εργασίας αυτής, όπως, για παράδειγμα, τα ορίσματα που δόθηκαν για τις συναρτήσεις που εκτελεί η εργασία και αποδεσμεύει την ίνα με το πέρας της εκτέλεσης της εργασίας. Εσωτερικά στην TORC χρησιμοποιείται ο όρος task descriptor για να περιγράψει αυτή τη δομή εκτέλεσης. Όλα τα task descriptors διαχειρίζονται εσωτερικά από τον δρομολογητή της βιβλιοθήκης, χωρίς να χρειάζεται την συμβολή του δρομολογητή του λειτουργικού συστήματος.

Βέβαια, χωρίς περαιτέρω επεμβάσεις, το παραπάνω μοντέλο θα είχε σημαντικά



Σχήμα 2.1: Το μοντέλο εκτέλεσης της TORC

μειονεκτήματα. Εφόσον τα task descriptors δρομολογούνται εσωτερικά από κομμάτι της βιβλιοθήκης και όχι από τον δρομολογητή του λειτουργικού συστήματος, μια εμποδιστική ενέργεια (blocking operation) του MPI ή του λειτουργικού συστήματος σε μια εργασία θα είχε ως αποτέλεσμα την παύση εκτέλεσης για όλο το πρόγραμμα. Ένα άλλο αρνητικό της αποφυγής του δρομολογητή του λειτουργικού συστήματος είναι το ουσιαστικά μονοπύρηνο μοντέλο εκτέλεσης που επιβάλουμε, αφού δεν εκμεταλλευόμαστε με βέλτιστο τρόπο τους υπολογιστικούς πόρους ενός σύγχρονου πολυπύρηνου επεξεργαστή. Αυτό συμβαίνει διότι το λειτουργικό σύστημα απαγορεύει την πρόσβαση μιας διεργασίας σε διαφορετικούς πυρήνες από αυτόν όπου εκτελείται εάν χρησιμοποιούνται κλήσεις επιπέδου χρήστη.

Η τροποποίηση που λύνει τα παραπάνω προβλήματα είναι το μοντέλο εκτέλε-

σης δύο επιπέδων, όπως φαίνεται στο Σχήμα 2.1. Δημιουργώντας νήματα-εργάτες (workers threads) με νήματα επιπέδου πυρήνα, μέσω της βιβλιοθήκης Pthreads, δίνουμε στον δρομολογητή του λειτουργικού συστήματος την δυνατότητα να τα διαμοιράσει στους φυσικούς πυρήνες. Με τη σειρά, ο κάθε worker εκτελεί με νήματα επιπέδου χρήστη εργασίες στον κάθε πυρήνα, πετυχαίνοντας έτσι ένα πιο σταθερό μοντέλο εκτέλεσης. Χρησιμοποιώντας νήματα και στο επίπεδο πυρήνα, αλλά και στο επίπεδο χρήστη, εκμεταλλευόμαστε πλήρως τα σύγχρονα πολυπύρηνα συστήματα, αφού πλέον η TORC χρησιμοποιεί όσους φυσικούς πυρήνες ορίζει ο χρήστης για την εκτέλεση του προγράμματος και η εκτέλεση σε καθέναν από αυτούς τους πυρήνες εκτυλίσσεται χωρίς παύσης δρομολόγησης και χωρίς ακριβά context switches ή εκτενείς αναμονές για την επανεκκίνηση εμποδιστικών συναρτήσεων.

2.2.2 Μοντέλο δημιουργίας, δρομολόγησης και συγχρονισμού εργασιών

Η συνάρτηση TORC_CREATE δημιουργεί μια εργασία και αναλαμβάνει την δρομολόγησή της στη συστάδα. Ως παραμέτρους δέχεται ορισμένες οδηγίες που αφορούν τον κόμβο στον οποίο επιθυμεί ο χρήστης να εκτελεσθεί η εργασία, και δείκτες προς τη συνάρτηση εργασίας και τη συνάρτηση που θα χρησιμοποιηθεί ως callback. Αν η συνάρτηση που ορίστηκε για την εργασία δέχεται παραμέτρους, μπορούν αυτοί να ακολουθήσουν τις προηγούμενες παραμέτρους της TORC_CREATE. Η βιβλιοθήκη υποστηρίζει διάφορες τεχνικές περάσματος παραμέτρων στην συνάρτηση της εργασίας και διαχειρίζεται τυχόν δείκτες και αντίγραφα που προκύπτουν, λαμβάνοντας υπόψη το αν η συνάρτηση αυτή θα τρέξει τοπικά ή σε απομακρυσμένο κόμβο. Σε περίπτωση απομακρυσμένης εκτέλεσης, τα αντίγραφα μεταφέρονται και οι δείκτες κλωνοποιούνται και προσαρμόζονται στο πεδίο διευθύνσεων του απομακρυσμένου μηχανήματος μέσω κλήσεων του MPI.

Οι εργασίες έχουν μεταξύ τους σχέση γονέα-παιδιού με τις διεργασίες-εργάτες. Ως συνέπεια, όλοι οι εργάτες γνωρίζουν όλα τα παιδιά τους. Βάσει αυτής της σχέσης, είναι δυνατή η αναμονή του worker μέχρι την ολοκλήρωση εκτέλεσης όλων των εργασιών κατά τη διάρκεια της ζωής του. Η ύπαρξη της δυνατότητας συγχρονισμού των εργασιών είναι απαραίτητη στον προγραμματισμό κατανεμημένων συστημάτων, καθώς εξαρτάται από αυτή η ορθότητα εκτέλεσης μεγάλου αριθμού εφαρμογών και αλγορίθμων.

2.2.3 Μοντέλο ουρών εργασιών

Για τη δρομολόγηση των εργασιών ο κάθε worker χρησιμοποιεί ένα σύστημα που αποτελείται από πολλαπλές ουρές, με διαφορετικές ιδιότητες η κάθε μία. Η κάθε διεργασία που δημιουργεί η TORC αρχικοποιεί το δικό της σύστημα ουρών, υλοποιώντας έτσι έναν κατανεμημένο σχεδιασμό για την δρομολόγηση των εργασιών, αποφεύγοντας έτσι γνωστούς περιορισμούς που θέτει αυτός, όπως προβλήματα συγχρονισμού, ενημέρωσης και παράνομης πρόσβασης. Ο σχεδιασμός της TORC διαχωρίζει τις ουρές σε ιδιωτικές και δημόσιες. Σε επίπεδο νημάτων πυρήνα, ο κάθε worker κατέχει μια δημόσια τοπική ουρά και μια ιδιωτική. Η αρχική κατανομή των διεργασιών γίνεται στις δημόσιες ουρές των worker. Όταν ένας worker ολοκληρώνει την εκτέλεση της τελευταίας εργασίας που έχει λάβει τελευταία από την ουρά του, ή αυτή παύσει την εκτέλεσή της για κάποιο λόγο, επιχειρεί να λάβει την επόμενη εργασία από την ουρά. Σε περίπτωση που αυτή είναι άδεια και ο χρήστης έχει ενεργοποιήσει το σύστημα κλοπής εργασιών, το νήμα-εργάτης θα επιχειρήσει να κλέψει ένα descriptor από τη δημόσια ουρά μιας άλλης διεργασίας. Στην περίπτωση που το νήμα κλέψει έγκυρο descriptor, τον εντάσσει στην ιδιωτική ουρά του για να αποφύγει το φαινόμενο μετακίνησης μεταξύ νημάτων. Αν, όμως, δεν καταφέρει να κλέψει descriptor από άλλες τοπικές διεργασίες, στρέφεται στις δημόσιες ουρές των απομακρυνσμένων κόμβων και αποστέλει σε αυτούς ένα σύγχρονο αίτημα κλοπής μέσω του server thread. Οι συγκεκριμένες τεχνικές κλοπής εργασιών, καθώς και ο νέος μηχανισμός prefetching, δηλαδή προκαταβολικής κλοπής εργασιών, περιγράφονται στο επόμενο κεφάλαιο.

2.2.4 Λεπτομέρειες υλοποίησης

Η λειτουργία της βιβλιοθήκης βασίζεται πρωτίστως σε μια δομή δεδομένων που ονομάζεται descriptor. Η δομή αυτή λειτουργεί ταυτόχρονα ως μονάδα εκτέλεσης αλλά και αναγνωριστικό συστήματος, διότι πέρα από δείκτες που αναφέρονται σε τμήματα μνήμης όπου βρίσκεται μια συνάρτηση προς εκτέλεσης και τα δεδομένα της, περιέχει και αναγνωριστικές πληροφορίες όπως σε ποια εικονική διεργασία δημιουργήθηκε.

Την διαχείριση και τον συγχρονισμό όλων των worker σε ένα κόμβο τα αναλαμβάνει ένα server thread. Το server thread λειτουργεί ως ένα checkpoint μεταξύ των worker, του υποσυστήματος ουρών, του υποσυστήματος επικοινωνίας μέσω MPI και

του scheduler. Κάθε φορά που μία εργασία εκτελεί μια ενέργεια που θα επηρεάσει το αντίστοιχο υποσύστημα κάθε άλλης εργασίας, γίνεται προώθηση ενός αιτήματος στο server thread από την πρώτη εργασία, μέσω του worker στην οποία εκτελείται. Το server thread στη συνέχεια, μέσω του υποσυστήματος επικοινωνίας, ενημερώνει όλους τους υπόλοιπους τοπικούς workers και η αντίστοιχη ενέργεια πραγματοποιείται πλέον σε όλο το σύστημα. Το server thread έχει και εποπτεία στον scheduler, καθώς το σύστημα κλοπής εργασιών, που είναι το κύριο αντικείμενο της διπλωματικής εργασίας και θα αναλυθεί περαιτέρω στο επόμενο κεφάλαιο, χρειάζεται πρόσβαση στα descriptors, στο σύστημα επικοινωνίας για να σταλεί το σήμα κλοπής καθώς και ένας buffer που περιέχει το κλεμμένο descriptor, καθώς και στο σύστημα ουρών για να γίνει σωστή τοποθέτηση και ενημέρωση.

Ο server διατηρεί επίσης και στοιχεία σχετικά με το runtime της βιβλιοθήκης για την πληροφόρηση του χρήστη μετά το πέρας της εκτέλεσης αλλά και για εσωτερικό bookkeeping και load balancing.

Από τη μεριά του χρήστη, το API της βιβλιοθήκης αντανακλά την νοοτροπία της γλώσσας προγραμματισμού C, προσφέροντας ένα μικρό σύνολο συναρτήσεων, το οποίο όμως προσφέρει εξαιρετική ισχύ στον προγραμματιστή. Η χρήση της βιβλιοθήκης απαρτίζεται από τρία απλά βήματα: την δήλωση συναρτήσεων και αρχικοποίηση του runtime της βιβλιοθήκης, την δημιουργία των εργασιών και τέλος, τον συγχρονισμό εργασιών και την συλλογή αποτελεσμάτων. Για το πρώτο βήμα, παρέχεται η κλήση torc_init, στην οποία ο χρήστης προωθεί τα ορίσματα που δέχεται η main του προγράμματος από τη γραμμή εντολών και εσωτερικά αρχικοποιείται το runtime της βιβλιοθήκης, δηλαδή γίνεται εκχώρηση μνήμης για της δομές του συστήματος, ενεργοποιούνται οι δίαυλοι επικοινωνίας του MPI, κλπ. Μετά την αρχικοποίηση του runtime, ο χρήστης πρέπει να ενημερώσει τον πίνακα συναρτήσεων του server με την χρήση της κλήσης torc_register_task, η οποία δέχεται ως όρισμα ένα δείκτη σε συνάρτηση και ενημερώνει ένα εικονικό πίνακα διευθύνσεων (virtual address table / vtable) για κάθε εργάτη του runtime.

Όσον αφορά την δυναμική δημιουργία εργασιών, παρέχονται κλήσεις όπου ο χρήστης μπορεί να τροποποιήσει την συμπεριφορά των εργασιών κατά την εκτέλεσης ή τον κόμβο στον οποίο επιθυμεί να εκτελεσθεί η συγκεκριμένη εργασία. Επίσης, μπορεί να δημιουργήσει detached εργασίες, δηλαδή εργασίες που δεν συγχρονίζονται πλέον από το runtime της TORC, αλλά παύουν εκτέλεση μόνο όταν ολοκληρώσουν το έργο τους. Η δημιουργία εργασιών μπορεί να γίνει με άμεσο τρόπο, δηλαδή από

μια συνάρτηση που δεν συμμετέχει στην παραλληλοποίηση των εργασιών, ή με εμφωλευμένο τρόπο, δηλαδή από μια εργασία κατά την διάρκεια εκτέλεσης. Για τον συγχρονισμό των εργασιών, παρέχεται η torc_wait_all, η οποία είναι παρόμοια σε λογική με την κλήση Barrier του MPI.

Η εκτέλεση των εργασιών γίνεται σε βρόγχους εκτέλεσης. Μετά την αρχικοποίηση του runtime και κάθε worker, ξεκινάει μια επαναληπτική διαδικασία, όπου ο κάθε νήμα εκτεέλεσης κάνει dequeue έναν descriptor και αναθέτει ένα νήμα επιπέδου χρήστη σε αυτόν, όπου και εκτελείται. Στο ενδιάμεσο ενδέχεται να παρέμβουν τα συστήματα κλοπής, στην περίπτωση που αδειάσουν οι ουρές εκτέλεσης του worker. Ο βρόγχος εκτέλεσης παύει μόνο όταν ο server λάβει σήμα τερματισμού και το μεταδώσει σε όλες τις διεργασίες της TORC που τρέχουν στον κόμβο.

ΚΕΦΑΛΑΙΟ 3

ΕΠΕΚΤΑΣΕΙΣ ΣΤΟ ΜΟΝΤΕΛΟ TORC

Έχοντας εξηγήσει τη γενικότερη λειτουργία της βιβλιοθήκης TORC και των επί μέρους μηχανισμών που την απαρτίζουν μπορούμε να προβούμε στην ανάλυση του κυρίως θέματος της εργασίας, δηλαδή την υλοποίηση της κλοπής εργασιών και του prefetching.

3.1 Θεωρητικό υπόβαθρο

Ένα σύνηθες πρόβλημα που συναντάται σε υπολογιστικές συστάδες και γενικότερα στα κατανεμημένα συστήματα είναι το πρόβλημα της εξισορρόπησης φόρτου. Συχνά, κατά την εκτέλεση ενός προγράμματος, για έναν από πολλούς πιθανούς λόγους, παρατηρείται το φαινόμενο του άνισου φόρτου, δηλαδή ένα μέρος των κόμβων της συστάδας να επωμίζονται περισσότερη δουλειά από τα υπόλοιπα. Αυτό έχει ως αποτέλεσμα τη μη βέλτιστη εκμετάλλευση των υπολογιστικών πόρων, καθώς ορισμένοι υπολογιστές καλούνται να επεξεργαστούν δυσανάλογο όγκο δεδομένων, ενώ άλλοι ολοκληρώνουν τον μικρότερο όγκο εργασιών τους και έπειτα αδρανούν.

Ανάλογα τη φύση της εφαρμογής υπάρχουν διάφοροι τρόποι επίλυσης του προβλήματος αυτού. Σε ένα περιβάλλον παραγωγής, όπου το σύστημα καλείται να υλοποιήσει μια "customer-facing" υπηρεσία ή διεπαφή, υλοποιούνται συχνά "load-

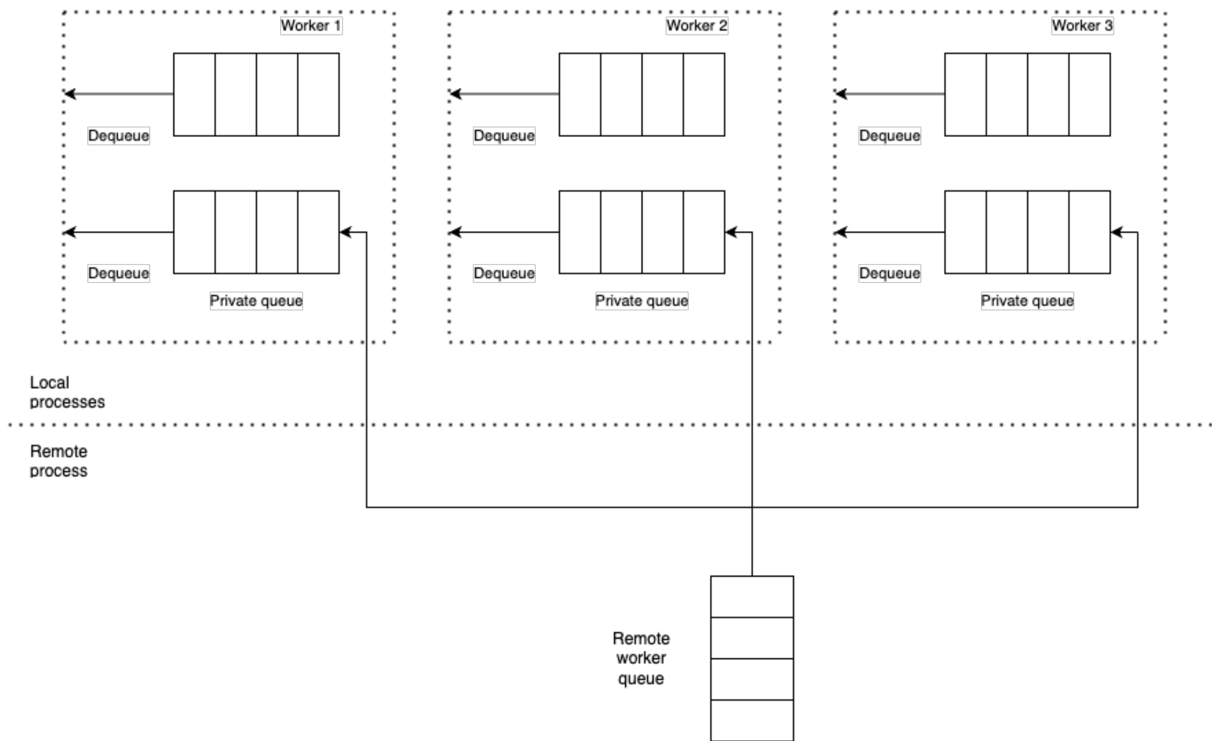
balancing hypervisors". Τα προγράμματα αυτά παρακολουθούν σε πραγματικό χρόνο τον φόρτο σε κάθε κόμβο της συστάδας, λαμβάνοντας υπόψη μετρικές όπως το ποσοστό χρήσης κύριας μνήμης και επεξεργαστή, χρήση δικτύου, κλπ. Βασίζοντας την απόφαση σε αυτές τις μετρικές, ο load-balancer αποφασίζει ποιος κόμβος πρέπει να αναλάβει την επόμενη εργασία, ώστε να αποτραπεί κορεσμός κάποιου κόμβου ή αδράνεια κάποιου άλλου. Στην περίπτωση, όμως, των επιστημονικών υπολογισμών, όπου είναι επικρατές το MPI και, κατά επέκταση η βιβλιοθήκη TORC, ο συνολικός φόρτος των υπολογισμών είναι γνωστός από την αρχή και απαιτούμε μόνο την παραλληλοποίησή του. Έτσι, επελέχθη μια άλλη μέθοδος ισορρόπησης του φόρτου στη συστάδα, η κλοπή εργασιών. Αυτή η μέθοδος είναι πιο απλή σαν ιδέα και σαφώς πιο απλή σαν υλοποίηση. Με απλά λόγια, όταν ένας worker που τρέχει ολοκληρώνει την εκτέλεση των εργασιών που ανατεθεί στην δημόσιά του ουρά και δεν έχει λάβει το σήμα ολοκλήρωσης από το server thread, αναζητά αρχικά για διαθέσιμο descriptor στις δημόσιες ουρές όλων των άλλων τοπικών worker που τρέχουν παράλληλα στον ίδιο επεξεργαστή υπό την εποπτία του server thread.

Στην περίπτωση που δεν βρεθεί descriptor στις δημόσιες ουρές των υπόλοιπων τοπικών worker, θα σταλεί από το server thread ένα αίτημα κλοπής στο δίκτυο, δηλαδή θα εξερευνήσει τους υπόλοιπους κόμβους εκτέλεσης της συστάδας και θα κλέψει έναν descriptor από τη πρώτη μη άδεια δημόσια ουρά worker που θα συναντήσει. Για την αποτροπή δημιουργίας ενός ατέρμονου βρόγχου κλοπής, όπου όλες οι τοπικοί worker κλέβουν συνεχώς από την δημόσια ουρά του καθενός το νέο αυτό descriptor, κάθε κλεμμένο descriptor τοποθετείται από τον worker που έκανε αρχικά το αίτημα κλοπής στην ιδιωτική ουρά του.

3.2 Λεπτομέρειες υλοποίησης κλοπής descriptor στη βιβλιοθήκη TORC

Ο μηχανισμός «κλοπής» descriptor στην βιβλιοθήκη TORC προτού γίνουν επεκτάσεις είναι αυστηρά σύγχρονος. Χωρίς την δυνατότητα για prefetching και, κατά επέκταση, της ασύγχρονης κλοπής που απαιτεί ένα σύστημα προληπτικής εισαγωγής στοιχείων σε μια ουρά, μια διεργασία κάνει αίτημα κλοπής μόνο όταν έχει αδειάσει η ιδιωτική της ουρά.

Η κλοπή ενός descriptor γίνεται σε δύο επίπεδα. Αρχικά, γίνεται η προσπάθεια



Σχήμα 3.1: Διάταξη πολυεπίπεδων ουρών descriptor

κλοπής τοπικά μεταξύ των workers, όπως περιγράφεται παραπάνω. Από τη στιγμή που αποτύχει η κλοπή σε πρώτο επίπεδο και ο worker στείλει στο server thread ένα αίτημα για κλοπή από απομακρυσμένη διεργασία, το σύστημα επικοινωνίας κάνει με τη σειρά σε όλους τους απομακρυσμένους κόμβους εκτέλεσης ένα αίτημα για descriptor από τις δημόσιες ουρές τους. Μέχρι την απάντηση στο μήνυμα αυτό, ο worker παύει προσωρινά κάθε εκτέλεση και αναμένει την απόκριση του υποσυστήματος επικοινωνίας. Αξίζει να σημειωθεί ότι το σύστημα κλοπής δεν εγγυάται η επιστροφή ενός έγκυρου descriptor. Η αρχιτεκτονική του συστήματος πολυεπίπεδων ουρών διαφαίνεται στο σχήμα 3.1.

Το τρέχον σύστημα οδηγεί σε συχνές περιόδους «αδράνειας», αφού η σύγχρονη αναμονή για μεταβίβαση μηνύματος για την κλοπή εισαγάγει καθυστερήσεις δικτύου. Επιπλέον, οι απομακρυσμένοι κόμβοι επίσης παύουν την εκτέλεση έργου για να αντεπεξέλθουν στο αίτημα κλοπής.

Για τη βελτίωση των επιδόσεων στη βιβλιοθήκη πρέπει να εισαχθεί, αρχικά, ένας νέος τρόπος κλοπής descriptor, που δεν δημιουργεί τόσο έντονα διαστήματα καθυστέρησης που τείνουν να εισάγουν στον χρόνο εκτέλεσης οι σύγχρονες ενέργειες σε δίκτυο, καθώς και να αποφεύγει την συνεχή προγραμματιστική εναλλαγή νημάτων.

3.2.1 Επέκταση ασύγχρονης κλοπής descriptors

Ξεκινώντας από την ιδέα ότι ο σύγχρονος προγραμματισμός σε ένα μέσο εγγενώς ασύγχρονο, όπως τα δίκτυα υπολογιστών, είναι πολύ πιθανό να δρα ανταγωνιστικά προς το μοντέλο εκτέλεσης της TORC, εξερευνήθηκαν μέθοδοι για την μετατροπή της σύγχρονης κλοπής σε ασύγχρονη. Η ασύγχρονη εκτέλεση αποφεύγει εκτεταμένες παύσεις διαφοροποιώντας τον τρόπο που αποστέλλονται τα αιτήματα κλοπής. Αντί για ένας-προς-ένα εμποδιστικά αιτήματα κλοπής, το σύστημα αποστέλλει αιτήματα χωρίς να περιμένει απάντηση σε όλους τους κόμβους της συστάδας. Οι κόμβοι που δέχονται το αίτημα εκτελούν την αποστολή μεταξύ execution loops, ενώ ο κόμβος που το απέστειλε επιχειρεί να αφαιρέσει έναν descriptor από την ιδιωτική ουρά του. Στην περίπτωση που κάποιος από τους υπόλοιπους κόμβους έχει προλάβει να αποστείλει έναν descriptor, η εκτέλεση συνεχίζει κανονικά. Διαφορετικά, αν η ουρά είναι άδεια, το σύστημα επανεκκινεί το execution loop τοπικά ώστε να δώσει επαρκή χρόνο στους υπόλοιπους κόμβους να απαντήσουν στο αίτημα κλοπής.

Με την υλοποίηση ασύγχρονης κλοπής descriptor επιλύεται το πρόβλημα των συνεχών και μεγάλων παύσεων εκτέλεσης, λόγω της σύγχρονης επικοινωνίας στο δίκτυο και της συνεχούς παύσης εκτέλεσης ωφέλιμου έργου κατά την αναμονή. Παρουσιάζεται, συνολικά, μια πιο κανονικοποιημένη συμπεριφορά, με πιο ήπιους και ομοιόμορφους χρόνους αναμονής κατά την κλοπή. Παρά τη βελτίωση αυτή, όμως, το μοντέλο κλοπής descriptor συνέχιζε να παρουσιάζει πρόβλημα επίδοσης, καθώς το ιδανικό σε μια λύση για το πρόβλημα εξισορρόπησης τους φόρτου είναι η ολοκληρωτική αποφυγή της αναμονής και, κατά επέκταση, της αδράνειας κόμβων στη συστάδα, αν αυτό είναι δυνατό. Για την επίλυση το προβλήματος αυτού, υλοποιήθηκε η δεύτερη μεγάλη επέκταση επέκταση στη βιβλιοθήκη TORC: το υποσύστημα prefetching.

3.3 Επέκταση υποσυστήματος prefetching

3.3.1 Το επίμονο πρόβλημα της αδράνειας

Όπως εξηγήθηκε στο τέλος της προηγούμενης υποενότητας, παρά την βελτιωμένη επίδοση τους συστήματος όταν εμφανίζονταν καθυστερήσεις, το πρόβλημα εξακολουθεί να υφίσταται. Παρότι βελτιώθηκε η συμπεριφορά του συστήματος στη δια-

χείριση κλήσεων δικτύου, οι οποίες αποτελούν μια σημαντική πηγή απώλειας επιδόσεων λόγω της απρόβλεπτης φύσης τους και του πιθανώς απροσδιόριστου χρόνου εκτέλεσης, παραμένει το πρόβλημα της συνεχούς παύσης εκτέλεσης χρήσιμου έργου και των αναλογικά ακριβών κλήσεων του scheduler.

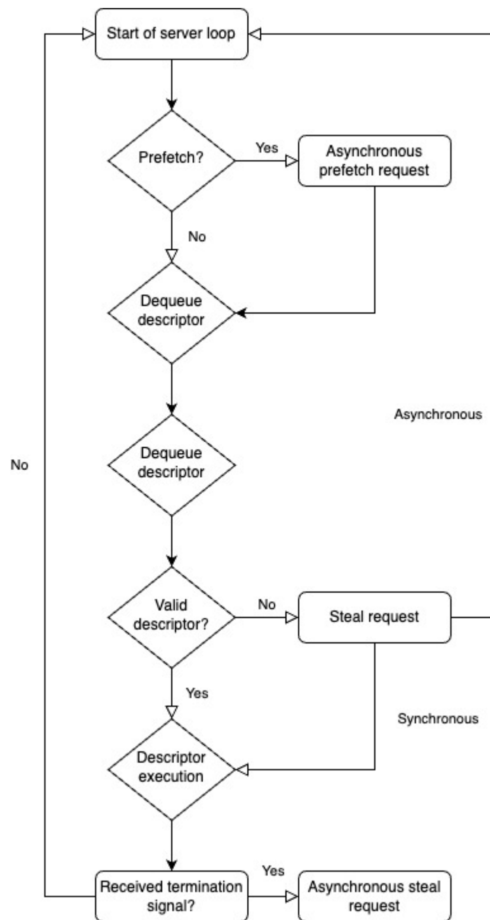
Η κλοπή descriptors κρίνεται εν μέρει ανεπαρκής λύση για το πρόβλημα της αδράνειας (idling). Ένα συχνό φαινόμενο κατά τα πειράματά μας ήταν η δημιουργία «βρόγχων» κλοπής και εκτέλεσης, όπου ένας κόμβος ολοκλήρωνε όλες τις διεργασίες στις ουρές του γρηγορότερα από το υπόλοιπο δίκτυο. Έτσι, το server thread απευθυνόταν στο υποσύστημα κλοπής descriptor, το οποίο έστελνε αίτημα στο υπόλοιπο δίκτυο και τοποθετούνταν ένας descriptor στην ιδιωτική ουρά της διεργασίας. Η διεργασία, έχοντας πλέον μη άδεια ουρά, εκτελούσε την νέα εργασία επιτυχώς, μόνο για να συναντήσει εκ νέου άδειες ουρές εκτέλεσης. Όπως και πριν, γινόταν εκ νέου αίτημα για κλοπή descriptor στο δίκτυο και εισάγονταν έτσι συνεχώς καθυστερήσεις στην εκτέλεση στον κόμβο όπου βρίσκονταν η διεργασία, μέχρι τελικά να λάβει το server thread του κόμβου το σήμα τερματισμού από το σύστημα και να μεταδώσει kill signal στους worker.

Η λύση είναι προφανώς μη ιδανική διότι το σύστημα αναλώνει τους υπολογιστικούς του πόρους σε κλήσεις των υποσυστημάτων επικοινωνίας και πολυεπίπεδων ουρών, αντί να αδρανεί. Μια τέτοια συμπεριφορά είναι διπλά σπάταλη, αφού καταναλώνεται ενέργεια για μη ωφέλιμο έργο.

3.3.2 Γενική ιδέα του prefetching

Η ιδέα του prefetching προκύπτει από τη πιο απλή ιδέα για την αποτροπή αδράνειας στο σύστημα: να αποτρέπεται εντελώς η ανάγκη για αιτήματα κλοπής στο δίκτυο. Για την επίτευξη, πρακτικά, αυτής της λύσης απαιτείται η υλοποίηση ενός μηχανισμού που εξασφαλίζει ότι οι ουρές μιας διεργασίας που φαινομενικά είναι ταχύτερη από τις υπόλοιπες δεν μένουν ποτέ άδειες από descriptors προς εκτέλεση.

Μία τέτοια υλοποίηση προαπαιτεί δύο μηχανισμούς, έναν που θα εξασφαλίζει την πληρότητα των ουρών των διεργασιών και ένα σύστημα-επόπτη, που καθορίζει ποιες διεργασίες χρειάζονται το υποσύστημα του prefetching δυναμικά κατά τον χρόνο εκτέλεσης. Όπως είναι προφανές, το πρώτο σκέλος της υλοποίησης καλύπτεται από τον μηχανισμού ασύγχρονης κλοπής descriptor. Το δεύτερο σκέλος, όμως, δεν είχε κάποια αντίστοιχη υπάρχουσα δομή στη βιβλιοθήκη TORC, οπότε



Σχήμα 3.2: Διάγραμμα ροής υποσυστήματος prefetching

υλοποιήθηκε από το μηδέν ένας σύστημα απόφασης.

3.3.3 Λειτουργία του συστήματος απόφασης prefetching

Ως βάση για το σύστημα απόφασης του prefetcher χρησιμοποιήθηκε το υποσύστημα στατιστικών του server thread που αναφέρθηκε στο προηγούμενο κεφάλαιο. Το σύστημα αυτό είχε σχεδιαστεί με την ενημέρωση του χρήστη κατά νου και όχι απαραίτητα την χρήση αυτών των στατιστικών από το ίδιο το runtime της βιβλιοθήκης. Λόγω αυτού, οι ενέργειες τις οποίες κατέγραφε είναι ανεπαρκείς για την δημιουργία ενός συστήματος που προσφέρει μια πλήρη εικόνα της κατάστασης του συστήματος σε ένα υποσύστημα απόφασης. Έτσι, πέραν του αριθμού δημιουργημένων και εκτελεσθέντων εργασιών που κατέγραφε σε όλους τους τοπικούς worker, το σύστημα επεκτάθηκε ώστε να συμπεριλαμβάνει κάθε σημαντική ενέργεια που πραγματοποιείται κατά την λειτουργία ενός προγράμματος TORC. Οι ενέργειες αυτές συμπεριλαμβάνουν την προσθήκη και αφαίρεση ενός descriptor από κάποια

ουρά, των αριθμό αποπειρών κλοπής αλλά και επιτυχών κλοπών descriptors στον κάθε κόμβο και στατιστικά σχετικά με τους χρόνους εκτέλεσης μιας εργασίας ανά διεργασία.

Έχοντας, πλέον, ένα πιο πλήρες σύστημα καταγραφής, είναι δυνατή η υλοποίηση του συστήματος απόφασης. Το υποσύστημα prefetching, όταν είναι ενεργοποιημένο, επεμβαίνει στον βρόγχο εκτέλεσης που εκτελεί η κάθε εικονική διεργασία. Σε κάθε επανάληψη, αφαιρείται ένας descriptor από τις ουρές του worker σε περίπτωση και εκτελείται η συνάρτηση που αποθηκεύεται στον descriptor. Στην περίπτωση που καμία τοπική ουρά δεν περιέχει έγκυρο descriptor, η διεργασία αποστέλλει αίτημα κλοπής. Η επέμβαση του συστήματος prefetching γίνεται στην αρχή του execution loop. Στην αρχή κάθε εκτέλεσης, πραγματοποιείται ένας έλεγχος για το αν κρίνεται απαραίτητο το prefetching και εκτελείται αναλόγως η λογική του υποσυστήματος, όπως διαφαίνεται στο σχήμα 3.2

Όταν το σύστημα απόφασης κρίνει πως μια διεργασία πληροί τις προϋποθέσεις για να πραγματοποιηθεί prefetching, το server thread αποστέλλει ένα μήνυμα με τον κωδικό που αντιστοιχεί στην ενέργεια του prefetching σε όλους τους κόμβους τους δικτύου. Ο μηχανισμός αυτός χρησιμοποιεί το υποσύστημα MPI της TORC για να ενημερώσει όλους τους κόμβους. Οι κόμβοι με την σειρά τους αποκρίνονται στο αίτημα αποστέλλοντας ο κάθε worker έναν descriptor από τις τοπικές ουρές τους. Όσο πραγματοποιείται ασύγχρονα η κλοπή descriptors, τοπικά συνεχίζει κανονικά την σειρά εκτέλεσης που μόλις αναφέρθηκε και επιχειρεί να αφαιρέσει τον επόμενο descriptor από μία από τις τοπικές ουρές του. Το σύστημα κλοπής παραμένει, βέβαια, ενεργό, οπότε ο worker έχει δυνατότητα σύγχρονης κλοπής στην περίπτωση που ο τελευταίος dequeued descriptor δεν είναι έγγυρος, μια κατάσταση που δεν είναι συχνή, καθώς είναι αυτό ακριβώς το σενάριο που προσπαθεί να αποτρέψει το σύστημα prefetching.

3.3.4 Αλγόριθμος απόφασης prefetching

Ο αλγόριθμος απόφασης του υποσυστήματος prefetching, που παρουσιάζεται σε ψευδοκώδικα στο σχήμα 3.1 είναι απλός και χρησιμοποιεί real-time στατιστικά για την κατάσταση των διαφόρων υποσυστημάτων του runtime της βιβλιοθήκης TORC για να αποφανθεί αν η προληπτική κλοπή descriptors θα ευνοήσει τις επιδόσεις του προγράμματος. Χρησιμοποιεί αρχικά το σύστημα καταγραφής κύριων ενερ-

γειών στο runtime. Συγκεκριμένα, μας ενδιαφέρουν για τον σχηματισμό απόφασης οι απόπειρες κλοπής descriptor που έχουν γίνει μέχρι το εκάστοτε χρονικό σημείο και αντίστοιχα πόσες από αυτές ήταν επιτυχείς. Για προφανείς λόγους, όταν υπάρχει μεγάλη αναλογία αποπειρών προς επιτυχείς ενέργειες δεν είναι συνετή η κλοπή descriptor, αφού οι ουρές των υπόλοιπων κόμβων στο δίκτυο παραμένουν σχετικά ή εντελώς άδειες και επιστρέφουν άκυρα descriptors. Εφόσον το prefetching γίνεται στο επίπεδο του worker, μας ενδιαφέρει ο αριθμός των worker που τρέχουν παράλληλα στον κάθε κόμβο για να μπορούμε να εξάγουμε συμπεράσματα που είναι έγκυρα για τον κόμβο συνολικά. Τέλος, υπάρχει σύστημα χρονομέτρησης κάθε κύριας ενέργειας στο runtime, συμπεριλαμβανομένων των ενεργειών κλοπής και εκτέλεσης ενός descriptor.

Όπως προαναφέρθηκε, το καθένα από αυτά τα υποσυστήματα δεν παρέχει από μόνο του μια επαρκή εικόνα για την κατάσταση εκτέλεσης σε κάθε διεργασία, οπότε λαμβάνονται όλα υπόψη για τον αλγόριθμο απόφασης. Η απόφαση αποτελείται από τρεις παράγοντες:

- Αναλογία κλοπών προς εκτελεσμένων descriptor.
- Αποτελεσματικότητα των κλοπών.
- Ταχύτητα εκτέλεσης.

Ο αλγόριθμος εξετάζει αρχικά κατά πόσο το σύστημα κλοπής descriptor αναλώνει μεγάλο κομμάτι του χρόνου εκτέλεσης του προγράμματος, βάσει αναλογίας της διάρκειας της τελευταίας κλοπής προς τον χρόνο εκτέλεσης του τελευταίου descriptor. Στη συνέχεια, εξετάζει κατά πόσο οι μέχρι τώρα κλοπές ήταν αποτελεσματικές, παράγοντας έναν λόγο μεταξύ αποπειρών κλοπής προς τις επιτυχείς κλοπές. Τέλος εξετάζει την αναλογία κλοπών προς εκτελέσεις descriptor. Εάν οποιοσδήποτε από αυτούς τους παράγοντες υπερβαίνει ορισμένες τιμές, τιμές στις οποίες καταλήξαμε με πειραματισμό περί την μείωση του συνολικού χρόνου εκτέλεσης των προγραμμάτων που χρησιμοποιήθηκαν για την επαλήθευση της εγκυρότητας της όλης δουλειάς αυτής της εργασίας, το σύστημα απόφασης prefetching επιστρέφει θετική απάντηση. Οι τιμές αυτές υπολογίζονται ως λόγοι των διαφόρων παραμέτρων του συστήματος που πολλαπλασιάζονται με κάποιες σταθερές. Οι μικρότεροι συντελεστές ευνοούν εκτελέσεις όπου το συνολικό έργο διασπάται σε λίγες εργασίες, ενώ οι εκτελέσεις με πολλές εργασίες ευνοούνται από μεγάλους λόγους. Εφόσον η αναλογία εκτελέσεων

προς κλοπών είναι υψηλότερη σε αυτό το σενάριο, με έναν μεγαλύτερο συντελεστή μπορούμε να ενεργοποιούμε το σύστημα prefetching συχνότερα, κρατώντας έτσι τις ουρές σε σταθερά επίπεδα πληρότητας.

Αλγόριθμος 3.1 Αλγόριθμος απόφασης συστήματος prefetching

Require: *MaxNvps*, len(*Executes*), *Steals*, *StealsAttemped* greater than zero

Require: *LastStealDuration*, *LastTaskDuration* greater than zero

Input: Executes per level, *Executes*

Input: Effectiveness cutoff level, *Cutoff*

1: *It* ← *0*
2: *Ex* ← *0*
3: *WasSlow* ← *False*
4: *StealsWereEffective* ← *False*
5: *ShouldPrefetch* ← *False*
6: **while** *MaxNvps* ≥ *It* **do**
7: *Ex* ← *Ex* + *Executes[It]*
8: *It* ← *It* + *1*
9: **end while**
10: *WasSlow* ← *LastTaskDuration* ≥ *0.1* × *LastStealDuration*
11: *StealsToExec* ← *Steals* / *Executes* ≥ *0.4*
12: *StealsWereEffective* ← *Steals* / *StealsAttempted* ≥ *Cutoff*
13: *ShouldPrefetch* ← *StealsToExec* ∨ (*WasSlow* ∧ *StealsWereEffective*)

ΚΕΦΑΛΑΙΟ 4

ΕΦΑΡΜΟΓΕΣ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

4.1 Προεπισκόπηση πειραματικού περιβάλλοντος

Τα ακόλουθα πειράματα για την επαλήθευση της αποτελεσματικής λειτουργίας των επεκτάσεων στη βιβλιοθήκη πραγματοποιήθηκαν στην υπολογιστική συστάδα του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Πανεπιστημίου Ιωαννίνων.

Η συστάδα αποτελείται από δώδεκα κόμβους Dell PowerEdge R430 με επεξεργαστή Intel Xeon E5-2620v4 οκτώ φυσικών πυρήνων, 16GB κύριας μνήμης, 300 GB SSD και 0,5TB HD τοπικοί χώροι αποθήκευσης. Το δίκτυο, τόσο μεταξύ των κόμβων όσο και προς εξωτερικά δίκτυα είναι Gigabit Ethernet (1Gbps).

Η εφαρμογή πολλαπλασιαμού πινάκων εκτελέσθηκε και στην υπολογιστική συστάδα του Εθνικού Δικτύου Υποδομών Τεχνολογίας και Έρευνας (ΕΔΥΤΕ/GRNET), ονόματι “ARIS”. Χρησιμοποιήθηκαν 12 “thin nodes”, όπως ονομάζουν οι διαχειριστές της συστάδας κόμβους IBM NeXtScale nx360 M4, με δύο επεξεργαστές Intel Xeon E5-2680v2 ο καθένας και 64GB κύριας μνήμης. Οι κόμβοι τρέχουν λειτουργικό σύστημα CentOS Linux 6.7 και δρομολογητή εργασιών SLURM. Για την μετάφραση της βιβλιοθήκης χρησιμοποιήθηκαν ICC 15.0.3 και Intel MPI 2018.0.

Κατά την διάρκεια της ανάπτυξης του κώδικα χρησιμοποιήθηκαν επίσης τα μηχανήματα parade και paragon της ερευνητικής ομάδας “Parallel Processing Group” του Τμήματος Μηχανικών Η/Υ & Πληροφορικής. Το μηχάνημα parade είναι ένα server

Dell R840 με τέσσερις επεξεργαστές Intel Xeon Gold 6130 με 16 φυσικούς πυρήνες και 32 λογικά νήματα ο καθένας, σε συνδυασμό με 256GB κύριας μνήμης. Το μηχάνημα paragon είναι ένα παλαιότερο workstation που φέρει δύο επεξεργαστές AMD Opteron 6166, 12 πυρήνων ο καθένας και 64GB κύριας μνήμης.

Η υπολογιστική συστάδα υποστηρίζει την υλοποίηση OpenMPI 2.1.1 του προτύπου MPI και χρησιμοποιεί τον μεταφραστή GCC 8.4.0 για την μετάφραση του κώδικα. Τα παραπάνω εργαλεία τρέχουν σε λειτουργικό σύστημα Ubuntu Linux 18.4.5 LTS.

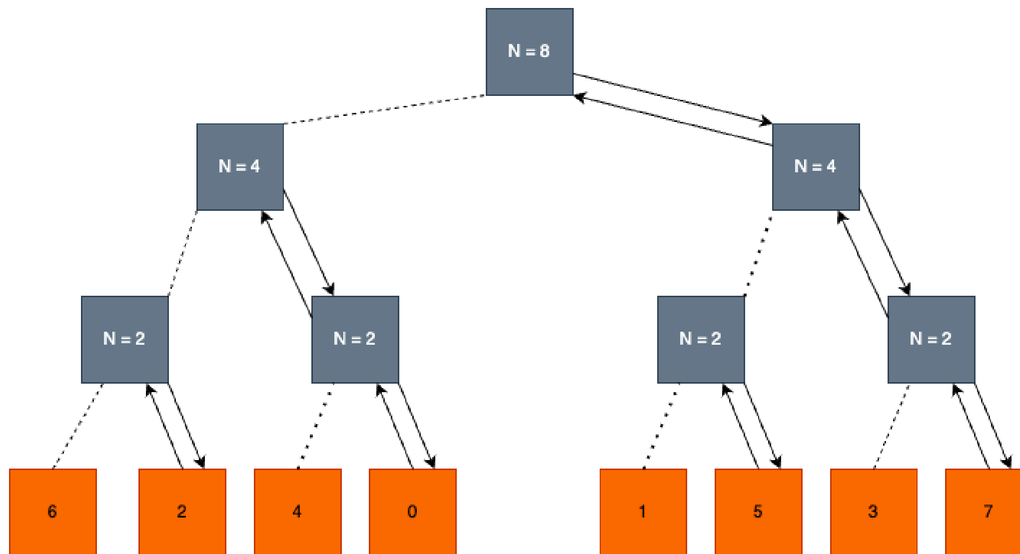
Στα μηχανήματα του Parallel Processing Group χρησιμοποιήθηκε η υλοποίηση MPICH σε έκδοση 3.4.1, για την επαλήθευση των επεκτάσεων σε διάφορα περιβάλλοντα MPI, αλλά και του το OpenMPI σε έκδοση 4.1.1 στο μηχάνημα parade. Οι δύο υλοποιήσεις του MPI υποστηρίζονται πάλι από τον μεταφραστή GCC σε έκδοση 8.5.0. Τα παραπάνω μηχανήματα υποστηρίζουν περιβάλλον CentOS Linux 8.

Όσον αφορά τις ρυθμίσεις της βιβλιοθήκης TORC, στα πειράματα επελέχθη ο χρήση τεσσάρων worker σε κάθε υπολογιστή της συστάδας. Λόγω των ρυθμήσεων λειτουργίας των κόμβων και στις δύο συστάδες, οι επεξεργαστές έχουν την τεχνολογία "hyperthreading" απενεργοποιημένη, οπότε ο αριθμός νημάτων αντανακλά τον αριθμό φυσικών πυρήνων του επεξεργαστή. Αυτή η κατάσταση μας ευνοεί, καθώς αποτρέπει τον διαμοιρασμό πυρήνων μεταξύ workers μέσω λογικών πυρήνων και μπορούμε να είμαστε σίγουροι πως ο κάθε worker εκμεταλλεύεται πλήρως το hardware του κάθε υπολογιστή.

Τα πειράματα πραγματοποιήθηκαν για διατάξεις ενός, τεσσάρων, οκτώ και δώδεκα κόμβων και στις δύο συστάδες. Έγιναν ξεχωριστά πειράματα αρχικά χωρίς κάποιο μηχανισμό load-balancing και στη συνέχεια με το σύστημα work stealing και prefetching ενεργοποιημένα.

4.1.1 Περιγραφή πειραματικών εφαρμογών

Για τη μελέτη επίδοσης των επεκτάσεων που υλοποιήθηκαν στην TORC επιλέχθηκαν τρεις κλασσικές εφαρμογές όσον αφορά την μέτρηση επιδόσεων σε παράλληλα συστήματα. Αρχικά, το κλασσικό πρόβλημα της παραλληλοποίησης πολλαπλασιασμού δισδιάστατου πίνακα. Η παρουσία της εφαρμογής στη γκάμα πειραμάτων κρίθηκε απαραίτητη για δύο λόγους: αρχικά για τον έλεγχο της συμπεριφοράς της TORC σε ένα απλό πρόβλημα, αλλά και για τη ολοένα και μεγαλύτερη ανάγκη για επιτά-

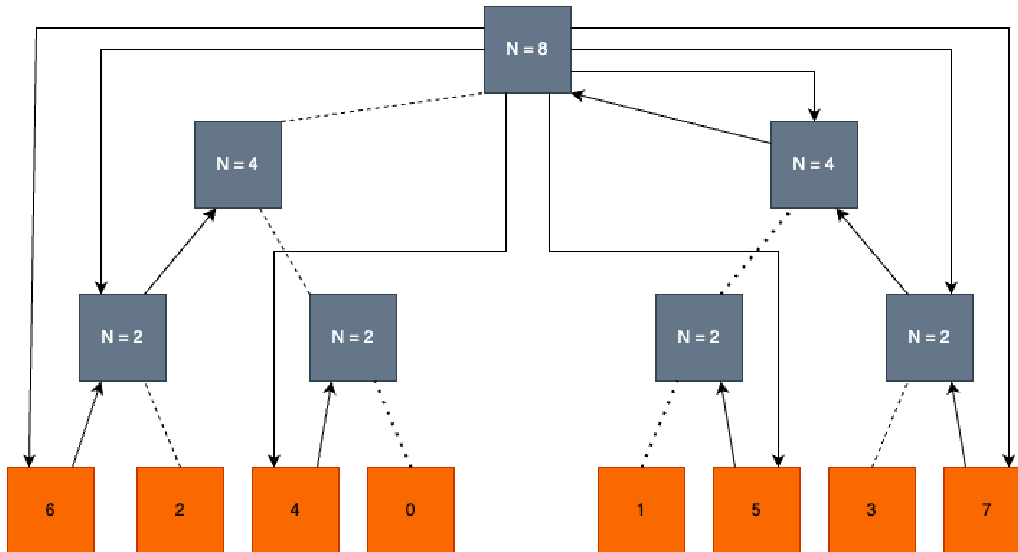


Σχήμα 4.1: Κλασσική προσέγγιση παραλληλοποίησης mergesort με επικοινωνίες

χυνση πολλαπλασιασμών πινάκων μεγάλης κλίμακας με την άνοδο του κλάδου του big data.

Στη συνέχεια, επελέχθη το πρόβλημα της παραλληλοποίησης του αλγορίθμου mergesort για την ταξινόμηση μιας ακολουθίας αριθμών. Το πρόβλημα αυτό εκ πρώτης όψεως είναι απλό αλλά παρέχει ευκαιρία για την παρατήρηση του συστήματος σε μια επιλογή με υψηλό επίπεδο ενδοδικτυακής κίνησης και το πως αυτό επηρεάζει τις επιδόσεις της βιβλιοθήκης. Στη συνέχεια έγινε μελέτη μιας βελτιστοποιημένης εκδοχής του αλγορίθμου, όπου επιχειρούμε να ελαχιστοποιήσουμε το κόστος επικοινωνιών μειώνοντας τον απαιτούμενο αριθμό μηνυμάτων που χρειάζεται να αποσταλούν κατά τη διάρκεια της εκτέλεσης για τον διαμοιρασμό και, έπειτα, την συναρμολόγηση του πίνακα.

Τέλος, κρίναμε ότι η παραλληλοποίηση της εφαρμογής θόλωσης εικόνας με Γκαουσσιανό θόρυβο είναι κατάλληλη για την εργασία, λόγω της, αναλογικά με τις άλλες δυο εφαρμογές, πιο περίπλοκης προγραμματιστικής υλοποίησης. Αυτό είναι σημαντικό για να κρίνουμε, πέραν των επιδόσεων της βιβλιοθήκης και την ευχρηστία της βιβλιοθήκης σε σενάρια που απαιτείται η συγγραφή πιο σύνθετου κώδικα.



Σχήμα 4.2: Βελτιωμένο mergesort με επικοινωνίες

4.2 Παραλληλοποιημένοι αλγόριθμοι ταξινόμησης 1D πινάκων

4.2.1 Περιγραφή παράλληλου αλγορίθμου mergesort

Η εφαρμογή αυτή είναι μία υλοποίηση του γνωστού αλγορίθμου mergesort, που εκμεταλλεύεται την ικανότητα της βιβλιοθήκης TORC για αβίαστη και εύκολη παραλληλοποίηση σε εργασίες. Το πρόγραμμα αποτελεί μία τροποποιημένη έκδοση των αλγορίθμων της εργασίας [8], συμβατή με το προγραμματιστικό μοντέλο της TORC.

Η εφαρμογή δημιουργεί εργασίες τόσο για την παραλληλοποίηση του αλγορίθμου τοπικά σε κάθε κόμβο, όσο και για την δημιουργία ενός δέντρου ταξινόμησης για εκμετάλλευση πολλαπλών κόμβων του δικτύου, για κατανεμημένη εκτέλεση. Στο σχήμα 4.1, οι συνεχείς γραμμές υποδηλώνουν υλοποίηση επικοινωνιών μεταξύ των αντίστοιχων κόμβων, ενώ οι διακεκομμένες δηλώνουν τη δημιουργία μίας εργασίας στον συγκεκριμένο κόμβο. Η λειτουργία του αλγορίθμου λειτουργεί με την τεχνική «διαίρει και βασίλευε». Η εκτέλεση χωρίζεται σε δύο διακριτές φάσεις δημιουργίας και εκτέλεσης. Στην πρώτη φάση, αυτήν της διαίρεσης, ο πίνακας διασπάται και ταξινομείται από εργασίες σε δύο ίσα τμήματα, μέχρις ότου αυτοί οι όλο και μικρότεροι υποπίνακες να περιέχουν μονάχα ένα στοιχείο. Στη δεύτερη φάση, αυτά τα δύο υποτμήματα κάθε διάσπασης συγχωνεύονται, δημιουργώντας έναν ενιαίο, ταξινομημένο πίνακα. Η υλοποίησή μας διαφέρει από τον απλό αλγόριθμο σε δύο κύρια σημεία. Πρώτον, ο διαχωρισμός κάθε υποπίνακα σε δύο καινούργια τμήματα

συμβαίνει με την εξής διαδικασία: το πρώτο μισό του υποπίνακα ταξινομείται με τη δημιουργία μίας εργασίας στον τρέχοντα worker, ενώ το υπόλοιπο αποστέλλεται σε έναν από τους υπόλοιπους κόμβους του δικτύου, με ταυτότητα

$$rank + 2 \times depth$$

όπου rank είναι ο μοναδικός αριθμός-αναγνωριστικό που έχει αναθέσει το runtime του MPI στον worker του κόμβου και depth το "βάθος" της διαίρεσης, δηλαδή πόσες φορές έχει τμηματοποιηθεί ο πίνακας. Αυτός ο διαχωρισμός εκμεταλλεύται την δυνατότητα για δημιουργία εμφωλευμένων εργασιών που προσφέρει η TORC, «γεννώντας» αναδρομικά άλλες εργασίες μέχρις ότου να έχουμε μοιράσει τον αρχικό πίνακα σε ίσα τμήματα σε κάθε κόμβο του δικτύου μας. Μετά την επιτυχή ταξινόμηση των δύο τμημάτων του υποπίνακα, ο κόμβος που δημιούργησε τις δύο αυτές εργασίες αναλαμβάνει τη συγχώνευσή τους. Κατά τη δημιουργία μίας απομακρυσμένης εργασίας ο worker-δημιουργός αποστέλλει στον άλλο worker έναν buffer που περιέχει το τμήμα του υποπίνακα που επιθυμεί να ταξινομήσει, το οποίο θα λάβει πίσω ταξινομημένο, πλέον, μετά το πέρας της απομακρυσμένης εργασίας.

Η άλλη κύρια τροποποίηση του αλγορίθμου είναι ο τρόπος ταξινόμησης σε τοπικό επίπεδο. Ο κάθε κόμβος αναλαμβάνει την ταξινόμηση ενός ενιαίου τμήματος του υποπίνακα, όπως αυτό του στάλθηκε από κάποια απομακρυσμένη διεργασία. Ο κόμβος ταξινομεί το κάθε τμήμα αυτό με τον αλγόριθμο quicksort, αφού διασπάσει τον αρχικό πίνακα σε τμήματα μεγέθους *Cutoff* στοιχείων. Το *Cutoff* ορίζεται ανάλογα με το πλήθος των επεξεργαστών του κόμβου, αφού υπάρχει ένα-προς-ένα ταύτιση μεταξύ εικονικών διεργασιών της TORC και πυρήνων στον κόμβο.

Μια ακόμη τροποποίηση που αφορά τη βελτίωση του αλγορίθμου έχει να κάνει με τη σωστή διάταξη της διαδικασίας τμηματοποίησης του πίνακα σε σχήμα δυαδικού δέντρου, ελαχιστοποιώντας τις μη απαραίτητες μεταφορές [8]. Όπως μπορεί κανείς να παρατηρήσει στο αρχικό διάγραμμα ταξινόμησης του αρχικού αλγορίθμου, οποιαδήποτε μεταφορά σε βάθος μεγαλύτερου του ενός στο δένδρο αποτελεί επανα-αποστολή δεδομένων που έχουν ήδη μεταφερθεί.

Για παράδειγμα, αν ανατρέξουμε στο σχήμα 4.1, η μεταφορά από έναν κόμβο τάξης $N = 4$ σε κόμβο τάξης $N = 2$, πρόκειται για μία επανα-αποστολή δεδομένων που έλαβε ο κόμβος $N = 4$ από τον αρχικό κόμβο. Το ίδιο συμβαίνει και κατά την επιστροφή αυτών των δεδομένων. Καθώς βρισκόμαστε σε περιβάλλον κατανεμημένης μνήμης, αυτές οι μεταφορές εισάγουν μη απαραίτητα χρονικά overheads και

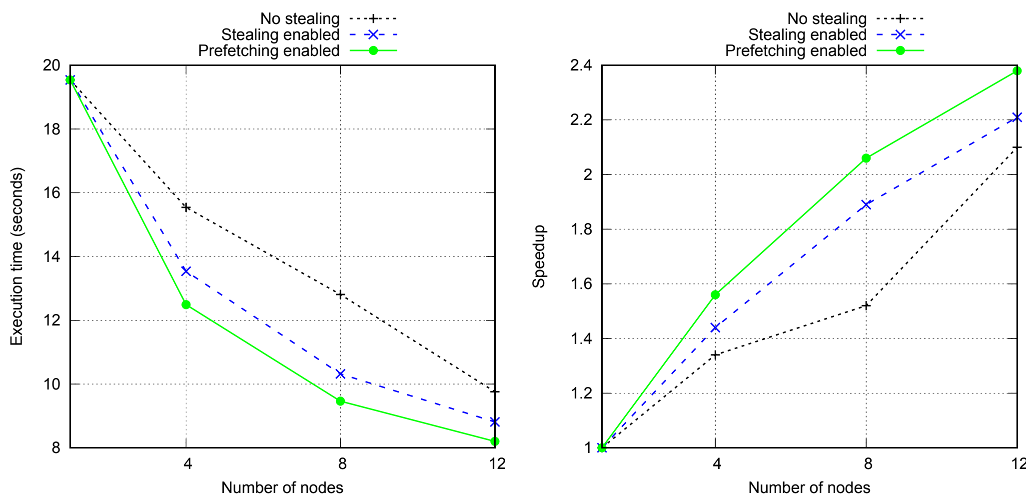
overheads επικοινωνίας. Στη νέα μας υλοποίηση, όλος ο λογικός διαχωρισμός των δεδομένων σε υποτμήματα συμβαίνει τοπικά, χρησιμοποιώντας την τάξη MPI του κάθε κόμβου στο γράφημα που συνθέτει ο αρχικός κόμβος ως αναγνωριστικό. Η ανάθεση της ταξινόμησης των τελικών πινάκων στους κόμβους του δικτύου πραγματοποιείται όταν έχουμε φτάσει στο μέγιστο βάθος του γράφου εκτέλεσης που σχηματίζουν οι κόμβοι, δηλαδή στα ακραία φύλλα του δέντρου. Με αυτή την τακτική αναδρομικής εκτέλεσης από τον αρχικό κόμβο αποτρέπονται περιττές επικοινωνίες και μεταφορές δεδομένων στον γράφο εκτέλεσης.

Με την νέα υλοποίηση διαχωρισμού της αρχικής λίστας που το πρόγραμμα λαμβάνει ως είσοδο, κάθε υποπίνακας μεταφέρεται μία φορά, μεταξύ του πρωταρχικού κόμβου και του κόμβου-φύλλο ο οποίος θα ταξινομούσε το αντίστοιχο υποπρόβλημα στο αρχικό δίκτυο εκτέλεσης. Για κάθε επίπεδο στο δίκτυο εκτέλεσης, ο κόμβος λαμβάνει υποπίνακα μεγέθους $N/2K$ όπου K το βάθος στον γράφο εκτέλεσης.

Αυτή η υλοποίηση εξακολουθεί να έχει ένα σημείο μη-βέλτιστης εκτέλεσης, συγκεκριμένα την σύμπτυξη του κεντρικού κόμβου, ο οποίος αναλαμβάνει πια όλες τις μεταφορές δεδομένων, κάτι το οποίο μπορεί να προκαλέσει πρόβλημα στις επιδόσεις, καθώς και το γεγονός ότι όλα τα σειριακά τμήματα του αλγορίθμου, όπως η συγχώνευση των ταξινομημένων υποπινάκων, δεν εκτελούνται πια κατανεμημένα, αλλά εκτελούνται σε έναν μόνο κόμβο.

Το παραπάνω πρόβλημα είναι το πρόβλημα της συγχώνευσης ταξινομημένων πινάκων, το οποίο έχει γραμμική χρονική πολυπλοκότητα. Στην πράξη, ακόμη και για πίνακες εκατομμυρίων στοιχείων, οι μετρήσεις φαίνεται να δείχνουν ότι οι χρόνοι συγχώνευσης δύο ταξινομημένων πινάκων σε έναν κόμβο είναι αμελητέοι μπροστά στους χρόνους ταξινόμησης των δύο επιμέρους πινάκων, αφού η σειριακή πρόσβαση σε μια λίστα είναι μια ταχύτατη ενέργεια σε σύγχρονα υπολογιστικά συστήματα λόγω της φιλικότητας ως προς την διάταξη των δεδομένων στη κρυφή μνήμη του επεξεργαστή, καθώς και του εύκολου συστήματος πρόσβασης (offset σε διεύθυνση μνήμης), οπότε κρίθηκε επαρκής η επίδοση της εφαρμογής και δεν έγιναν προσπάθειες περαιτέρω βελτιστοποίησης του προβλήματος.

Ο τρόπος διαμοιρασμού των υποπινάκων στη βελτιστοποιημένη εφαρμογή μπορεί να φανεί στο σχήμα 4.2.



Σχήμα 4.3: Αποτελέσματα εφαρμογής ταξινόμισης στην συστάδα του ΤΜΗΥΠ

4.2.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα

Για τα πειράματα αυτής της εφαρμογής επιλέχθηκε ως είσοδος πίνακας ακεραίων με 2^{22} στοιχεία. Αυτή η επιλογή έγινε ώστε να μελετήσουμε τις επιδόσεις στην περίπτωση όπου αποστέλλουμε αρκετά μεγάλους όγκους δεδομένων για επεξεργασία σε απομακρυσμένους κόμβους. Σε σενάριο χρήσης μικρότερων λιστών ακεραίων, η όποια πιθανή επιτάχυνση από την παραλληλοποίηση σε υπολογιστική συστάδα ακυρώνεται από το κόστος που προσθέτει στην εφαρμογή η χρήση μεταβίβασης μηνυμάτων που απαιτείται για την παραλληλοποίηση αυτή.

Όπως θα φανεί και στις ακόλουθες εφαρμογές, ίσως αντίθετα με την διαίσθηση που θα σχημάτιζε κάποιος, η αποδοτικότητα της μεθόδου του prefetching φαίνεται να αυξάνεται με σχετικά μικρό ρυθμό συναρτήσει του αριθμό κόμβων στην υπολογιστική συστάδα. Μια αφελής αρχική ερμηνεία είναι πως με τον αυξανόμενο αριθμό κόμβων, αυξάνεται η «επιφάνεια» για ανομοιογένεια στους χρόνους εκτέλεσης και, έτσι, μεγαλύτερη πιθανότητα να χρειαστεί κλοπή descriptor. Αυτό που απεδείχθη, όμως είναι πως πρακτικά σε σύγχρονες συστάδες όπου όλοι οι κόμβοι έχουν παρόμοιες τεχνικές προδιαγραφές, είναι πως δεν παρατηρούνται διακυμάνσεις στον χρόνο εκτέλεσης, ειδικά σε απομονωμένα περιβάλλοντα. Έτσι, για μικρότερο αριθμό

κόμβων, η όποια διακύμανση είναι πιο αισθητή λόγω του μεγαλύτερου αριθμού descriptor που ανατίθενται ανά κόμβο και μεγαλύτερης πιθανότητας παρεμβολής άλλων διεργασιών που τρέχουν στον κόμβο. Επίσης, το κόστος των επικοινωνιών αυξάνεται δυσανάλογα με τον αριθμό κόμβων, αλλά και από τον τρόπο υλοποίησης του συστήματος prefetching, συμβάλλοντας στη μη ιδανική απόδοση του συστήματος.

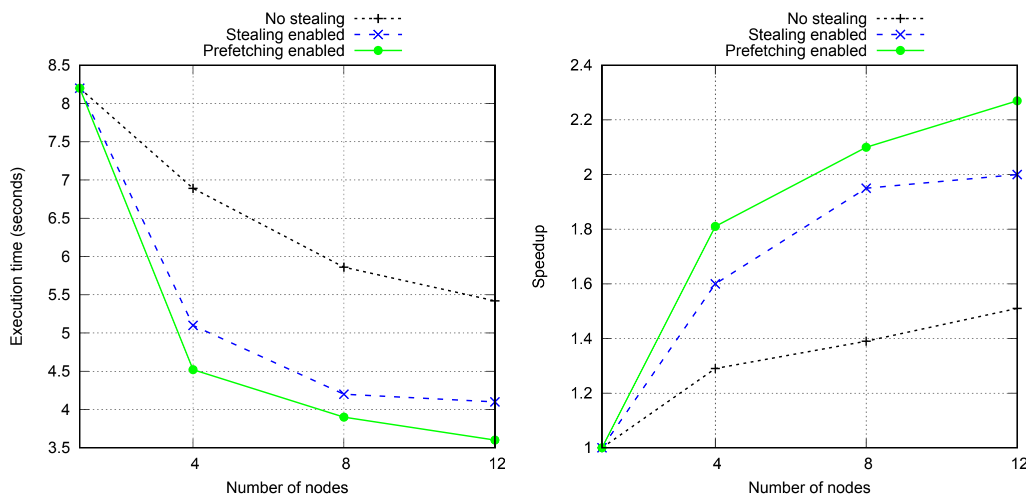
Βέβαια, ο σκοπός, που είναι η επιτάχυνση του χρόνου εκτέλεσης με τη χρήση του prefetcher, επιτυγχάνεται σε μη-αμελητέο βαθμό και παρατειρείται ικανοποιητικό speedup που αγκίζει την τάξη του 10% για 12 κόμβους στη συστάδα του ΤΜΗΥΠ.

4.3 Παραλληλοποιημένος αλγόριθμος πολλαπλασιασμού μητρών

4.3.1 Περιγραφή αλγορίθμου

Η εφαρμογή αυτή αποτελεί μία τροποποίηση υπαρχόντων προγραμμάτων για τον πολλαπλασιασμό πινάκων με τη χρήση του προτύπου MPI, προσαρμοσμένη πλέον στο προγραμματιστικό μοντέλο της βιβλιοθήκης TORC. Η υλοποίηση χωρίζει τους πίνακες εισόδου σε $M \times M$ υποπίνακες όπου κάθε υποπίνακας έχει μέγεθος N/M, όπου N το μέγεθος της εισόδου. Το μέγεθος N/M αναφέρεται ως S και λειτουργεί ως kernel/υποπίνακας εκτέλεσης και ο πολλαπλασιαμός ενός τέτοιου, επίσης τετραγωνικού, υποπίνακα θεωρείται ως μία εργασία. Κάθε τέτοια εργασία χαρακτηρίζεται από ένα ID, το οποίο χρησιμοποιείται από τον εργάτη που θα αναλάβει την εργασία έτσι ώστε να γνωρίζει ποιο υποπρόβλημα πρόκειται να λύσει. Το υπάρχον πρόγραμμα χρησιμοποιούσε παραλληλισμό με τη χρήση ενός μοντέλου εκτέλεσης σε κατανεμημένη μνήμη, όπως αυτό υποστηρίζεται από το πρότυπο του MPI. Η μετατροπή αυτού του προγράμματος ώστε να είναι συμβατό με το προγραμματιστικό μοντέλο που χρησιμοποιήσαμε δεν ήταν τετριμμένη, αφού μετά την οργάνωση κάθε σταδίου του αλγορίθμου σε ξεχωριστές συναρτήσεις στο αρχικό πρόγραμμά μας, έπρεπε να ακολουθήσει ορθή δημιουργία TORC tasks. Για να επιτευχθεί αυτό χρειάστηκε, αρχικά, να αποστείλουμε τους δύο πίνακες εισόδου σε κάθε απομακρυσμένο κόμβο κατά τη φάση αρχικοποίησης του αλγορίθμου, στην οποία φάση εκχωρείται μνήμη τοπικά σε κάθε εργασία και αντιγράφονται εκεί οι πίνακες.

Για την ευκολότερη υλοποίηση του αλγορίθμου οι πίνακες χρειάστηκε οι πίνα-

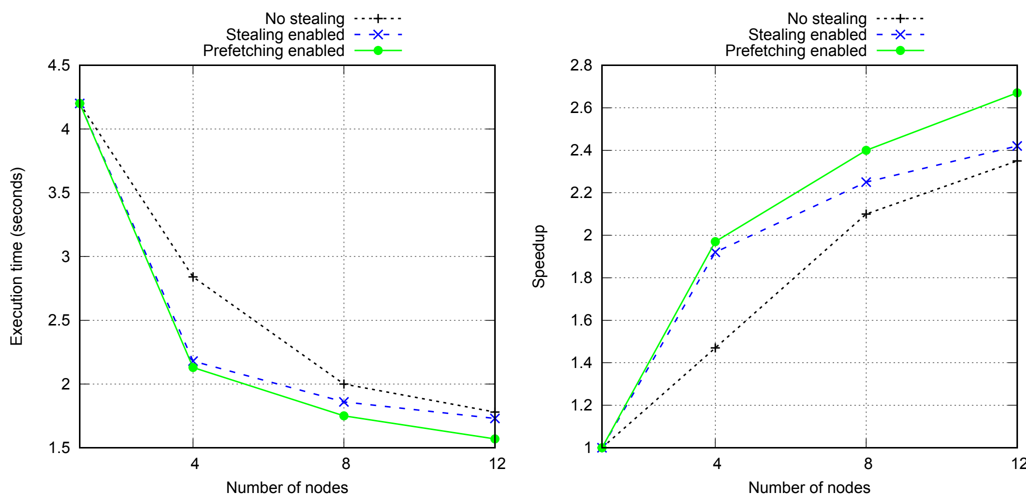


Σχήμα 4.4: Αποτελέσματα εφαρμογής πολλαπλασιασμού πινάκων στην συστάδα του ΤΜΗΥΠ

κες να υλοποιηθούν ο καθένας ως μονοδιάστατος πίνακας $N \times N$ στοιχείων και στη συνέχεια να δεσμεύονται buffers σε κάθε εργασία για την επεξεργασία του κάθε τμήματος. Όπως προαναφέρθηκε, οι buffers αυτοί έχουν μέγεθος S, το οποίο αποφασίστηκε πειραματικά στην τιμή των 64 στοιχείων ανά υποπίνακα. Ακολουθώντας αυτή την τεχνική υλοποίησης, η αποστολή των υποπινάκων εξαπλουστεύεται σημαντικά, αλλά απαιτείται προσοχή στη συλλογή των αποτελεσμάτων. Λόγω της τμηματοποίησης των αρχικών πινάκων σε buffers, ο συντονιστής στην αρχική υλοποίση, ο οποίος πλέον ταυτίζεται με τους workers στη TORC, πρέπει να επανασυναρμολογήσει με σωστή σειρά τους υποπίνακες που δέχεται από τις εργασίες, υπολογίζοντας τη θέση τους στο πίνακα του τελικού αποτελέσματος βάσει των ποιων υποτμημάτων των δύο πινάκων εισόδου χρησιμοποιήθηκαν για την παραγωγή του αποτελέσματος αυτού.

4.3.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα

Για τη χρονομέτρηση του πειράματος χρησιμοποιήθηκαν πίνακες διαστάσεων 1000×1000. Τα αποτελέσματα που παρατηρήθηκαν ίσως δεν αντιπροσωπεύουν πλήρως την



Σχήμα 4.5: Αποτελέσματα εφαρμογής πολλαπλασιασμού πινάκων στη συστάδα ARIS

ιδανική βελτίωση επιδόσεων που μπορεί να παρατηρηθεί σε αυτή την εφαρμογή με τη χρήση των συστημάτων κλοπής και prefetching. Λόγω του σχετικά μικρού όγκου δεδομένων, δεν υπάρχει η ίδια συμφόρηση στις κλήσεις δικτύου. Το μέγεθος πινάκων επιλέχθηκε για την πιο εύκολη τμηματοποίησή τους σε σχετικά μικρούς buffers, φιλικούς προς το MPI και γενικότερα το μοντέλο επικοινωνίας με μεταβίβαση μηνυμάτων. Βέβαια, οι σχετικά μικροί πίνακες δίνουν την ευκαιρία να παρατηρηθεί αν υφίσταται σταθερή αύξηση του κόστους των επικοινωνιών, αν αυτό παρατηρηθεί, σε συνάρτηση με τον όγκο δεδομένων. Δυστυχώς, όπως και στο προηγούμενο παράδειγμα ταξινόμησης μονοδιάστατων πινάκων, το κόστος των επικοινωνιών παραμένει σχετικά υψηλό. Η εκτέλεση έγινε με 4 εικονικούς επεξεργαστές ανά κόμβο, όπως και στην εφαρμογή ταξινόμησης.

Όσον αφορά τις επιδόσεις της εφαρμογής αυτής, που φαίνονται στο σχήμα 4.4, παρατηρείται μια μικρότερη βελτίωση σε σχέση με τα άλλα πειράματα. Η βελτίωση του χρόνου εκτέλεσης φαίνεται να έχει ελαφρώς καλύτερη κλιμάκωση σε συνάρτηση με τον αριθμό κόμβων με το σύστημα κλοπής ενεργοποιημένο, από ότι για το σύστημα prefetching. Αν συγκριθούν τα αποτελέσματα του πειράματος στη συστάδα του τμήματος σε σχέση με τις τιμές που πήραμε από την εκτέλεση στη συστάδα

ARIS, όπως φαίνονται στο σχήμα 4.5, υπογραμμίζεται ξανά η σημαντικότητα της ελαχιστοποίησης επικοινωνιών με κάθε δυνατό τρόπο. Η γρηγορότερη σύνδεση Infiniband φαίνεται να οδήγησε σε καλύτερη κλιμάκωση σε συνάρτηση με τον αριθμό κόμβων στη συστάδα σε σχέση με τη σύνδεση Gigabit Ethernet της συστάδας του ΤΜΗΥΠ.

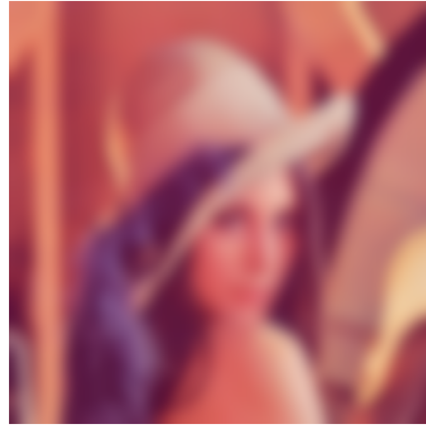
4.4 Παραλληλοποιημένη εφαρμογή θόλωσης εικόνας με Γκαουσιανό θόρυβο

4.4.1 Περιγραφή αλγορίθμου

Όπως και η παραλληλοποιημένη εφαρμογή πολλαπλασιασμού πινάκων δύο διαστάσεων, το πρότυπο που χρησιμοποιήθηκε και για την εφαρμογή της θόλωσης εικόνας με προσθήκη Γκαουσιανού θορύβου είναι μια τροποποιημένη υπάρχουσα υλοποίηση που βασίζεται στο MPI.

Η συγκεκριμένη υλοποίηση υποστηρίζει το BMP format, οπότε η εκτέλεση του προγράμματος ξεκινάει με την «ανάγνωση» της εικόνας και το διαχωρισμό των τριών καναλιών χρώματος (κόκκινο, πράσινο, μπλε) σε buffers. Παρόμοια με τη προηγούμενη εφαρμογή, λόγω του τρόπου που το MPI διαχειρίζεται buffers, κάθε κανάλι χρώματος μιας εικόνας διαστάσεων $N \times M$ αποθηκεύεται σε ένα μονοδιάστατο buffer μεγέθους $N \times M$ και η τμηματοποίηση γίνεται εκεί. Η κύρια πράξη του αλγορίθμου είναι η συνέλιξη ενός kernel, δηλαδή ενός σχετικά μικρότερου πίνακα που εφαρμόζεται σαν μάσκα στην εικόνα. Οι διαστάσεις του kernel ορίζονται από το χρήστη. Το φαινόμενο της θόλωσης αυξάνεται αισθητά όσο η μάσκα συνέλιξης μεγαλώνει. Η μάσκα αυτή εφαρμόζεται σε όλα τα pixels της εικόνας, χρησιμοποιώντας το πρώτο pixel της μάσκας ως σημείο αναφοράς, μέχρι το kernel να καλύψει όλα τα pixels της εικόνας. Οι τιμές της μάσκας αυτής ορίζονται από την εφαρμογή μίας δισδιάστατης Γκαουσιανής συνάρτησης στις συντεταγμένες δύο αξόνων που λειτουργούν ως ορίσματα. Έτσι, η τιμή φωτεινότητας στο κανάλι σε ένα συγκεκριμένο pixel πολλαπλασιάζεται με το βάρος του στοιχείου της μάσκας που το επικαλύπτει. Για την παρατήρηση της συμπεριφοράς του runtime της TORC σε εφαρμογές μεγαλύτερης διάρκειας, καθώς και την αποτελεσματικότητα του prefetching σε συνθήκες δυναμικής δημιουργίας νέων descriptor κατά το runtime, η εφαρμογή επαναλαμβά-

(α) Αρχική εικόνα



(β) Θολωμένη εικόνα

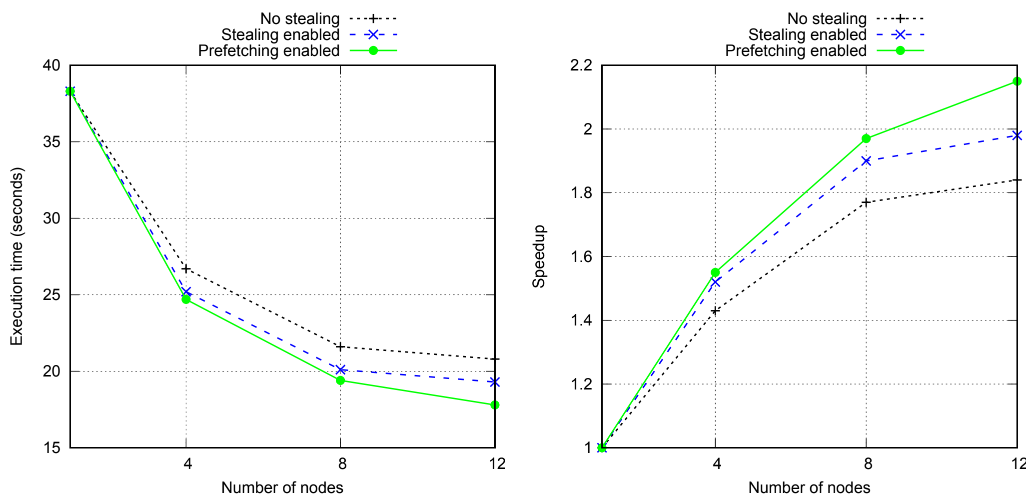
Σχήμα 4.6: Εικόνα-πρότυπο για την εφαρμογή θόλωσης

νει την θόλωση της εικόνας 10 φορές.

Η αρχική υλοποίηση με MPI τμηματοποιούσε τους τρεις buffers χρωμάτων και τους απέστελλε σε κάθε διεργασία, όπου πραγματοποιούνταν σειριακά η πράξη της συνέλιξης. Όπως είναι προφανές, η προσαρμογή της εφαρμογής MPI σε εφαρμογή TORC είναι πολύ απλή. Αρχικά, οι buffers χωρίζονται σε τμήματα, για παράδειγμα 12, όσα οι κόμβοι της υπολογιστικής συστάδας του ΤΜΗΥΠ και ανατίθενται όλοι σε μια μητρική διεργασία. Η μητρική διεργασία εκτελείται από τον worker του κάθε κόμβου, ο οποίος στη συνέχεια τμηματοποιεί τον buffer που έλαβε σε περαιτέρω M υποτμήματα, όπου M ο αριθμός των εικονικών διεργασιών που ορίστηκε για το runtime της TORC. Η κάθε διεργασία πραγματοποιεί τη συνέλιξη σειριακά και στη συνέχεια ο worker συγχρονίζει τα αποτελέσματα από κάθε διεργασία, τα συγχωνεύει και τα αποστέλλει πίσω στον master node. Όπως ο worker, ο κεντρικός κόμβος συγχωνεύει τους buffer του κάθε worker και παράγεται έτσι τοπικά η τελική εικόνα με τη κωδικοποίηση των buffer των καναλιών στο format BMP και γράφεται σε αρχείο.

4.4.2 Πειραματικά αποτελέσματα σε υπολογιστική συστάδα

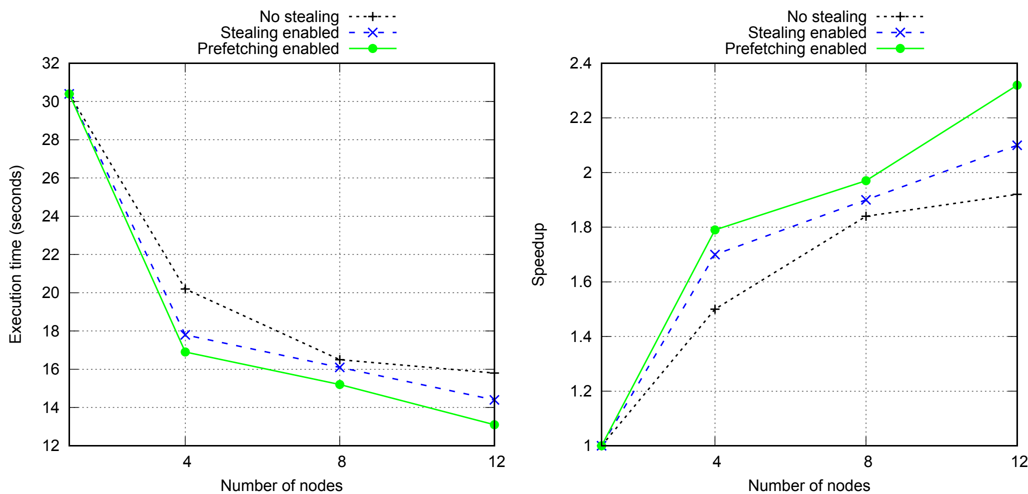
Για τη χρονομέτρηση επιδόσεων χρησιμοποιήθηκε εικόνα τύπου BMP, βάθους 24 bit, με πλήρες χρώμα σε τρία κανάλια, διαστάσεων 500×500. Για τη χρονομέτρηση των διάφορων λειτουργιών της TORC χρησιμοποιήθηκε μια μήτρα θόλωσης μεγέθους 6×6. Οι διαστάσεις αυτές επιλέχθηκαν διότι επιφέρουν μια ήπια θόλωση, που επιτρέπει να διακρίνεται η σύνθεση της εικόνας στην οποία εφαρμόζεται η θόλωση,



Σχήμα 4.7: Αποτελέσματα εφαρμογής θόλωσης εικόνας στην συστάδα του ΤΜΗΥΠ

όπως φαίνεται στο σχήμα 4.6, χωρίς όμως να είναι υπολογιστικά τετριμμένη η διαδικασία υπολογισμού. Επιπλέον, η διαδικασία επαναλαμβάνεται δέκα φορές για να παρατηρήσουμε την συμπεριφορά του runtime για εφαρμογές μεγαλύτερου συνολικού έργου.

Όπως και στην εφαρμογή παραλληλοποίησης του πολλαπλασιασμού πίνακα, παρατηρείται βελτίωση στις επιδόσεις όταν μεταφέρουν την εφαρμογή από τη συστάδα του ΤΜΗΥΠ στη συστάδα ARIS. Η διαφορά, βέβαια, είναι μικρότερη από αυτή για την εφαρμογή πολλαπλασιασμου. Το αποτέλεσμα αυτό ενδέχεται να προκλήθηκε από τον μικρότερο όγκο δεδομένων που χρειάζεται να αποστείλει στους κόμβους της συστάδας η εφαρμογή, καθώς και της μεγαλύτερης πολυπλοκότητας της πράξης της συνέλιξης, σε σχέση με τον πολλαπλασιασμό. Ο μεγαλύτερος χρόνος εκτέλεσης εξαρτώμενης από τον επεξεργαστή (CPU-bound delay) ενδεχομένως δίνει στο δίκτυο μεγαλύτερα παράθυρα επικοινωνίας, μειώνοντας έτσι τα latency spikes. Βλέπουμε έτσι πως εκφράζεται έτσι μια «προτίμηση» της βιβλιοθήκης για εφαρμογές με όσον τον δυνατό μικρότερο «όγκο» επικοινωνιών.



Σχήμα 4.8: Αποτελέσματα εφαρμογής θόλωσης εικόνας στην συστάδα ARIS

4.5 Αξιολόγηση

4.5.1 Αξιολόγηση πειραματικών αποτελεσμάτων

Τα αποτελέσματα της έρευνας στην τεχνική του work stealing αλλά και του prefetching απέφεραν θετικά, αλλά ίσως όχι ιδανικά αποτελέσματα. Το σύστημα prefetching δεν παρουσιάζει δραματική βελτίωση σε σχέση με το βασικό σύστημα κλοπής και συνολικά και τα δύο συστήματα δεν παρουσιάζουν ιδανική κλιμάκωση. Τυχόντα κέρδη σε επίδοση με τη χρήση της βιβλιοθήκης είναι πιο έντονα για μικρότερο αριθμό κόμβων και παράγοντες που θα εξηγηθούν παρακάτω αποτρέπουν την ιδανική συμπεριφορά της βιβλιοθήκης.

Στα πειράματα στη συστάδα του τμήματος χρησιμοποιώντας και τους δώδεκα κόμβους οι επιδόσεις όλων των εφαρμογών, με το σύστημα prefetching ενεργοποιημένο, ήταν καλύτερες από τις επιδόσεις στα πειράματα που ήταν ενεργό μόνο του σύστημα descriptor stealing.

Η εφαρμογή matmul φανερώνει τη σημαντικότητα της βελτιστοποίησης στο θέμα της ελαχιστοποίησης επικοινωνιών και της ανάγκης ύπαρξης γρήγορου δικτύου, αν συγκριθούν τα αποτελέσματα στη συστάδα του ΤΜΗΥΠ με διασύνδεση Gigabit Ethernet και στο σύστημα ARIS με διασύνδεση Infiniband. Ο επιπρόσθετος φόρ-

τος μηνυμάτων του συστήματος prefetching σημαίνει πως ο προγραμματιστής που υλοποιεί την εφαρμογή πρέπει να δίνει μεγάλη προσοχή στη ελαχιστοποίηση όλων των περιττών επικοινωνιών στον αλγόριθμο εκτέλεσης του προγράμματός του, όπως έγινε για παράδειγμα στην εφαρμογή mergesort, ώστε η επιπρόσθετη ποινή των μηνυμάτων του συστήματος prefetching να μην είναι αισθητή. Το ίδιο φαινόμενο παρατηρήθηκε και στην εφαρμογή θόλωσης εικόνας. Η ύπαρξη γρηγορότερου δικτύου οδήγησε σε καλύτερη κλιμάκωση στις επιδόσεις για αυξανόμενο αριθμό κόμβων.

Συνολικά, σε όλα τα πειράματα παρατηρήθηκε σαφής βελτίωση για όλα τα configurations όταν κάποιο σύστημα εξισορρόπησης ήταν ενεργοποιημένο. Ένα ενθαρρυντικό αποτέλεσμα είναι η ύπαρξη βελτίωσης στην επίδοση όλων των εφαρμογών μεταξύ του συστήματος ασύγχρονης κλοπής και όταν το σύστημα κλοπής υποβοηθείται από το σύστημα prefetching. Η βελτίωση μεταξύ των δύο configuration εμφανίζει, επίσης, καλή κλιμάκωση με τον αριθμό συστάδων στο δίκτυο. Σε αντίθεση με το σύστημα ασύγχρονης κλοπής, το υποσύστημα prefetching φαίνεται να διατηρεί ικανοποιητική κλίση στον ρυθμό speedup. Η βελτίωση στην επίδοση του συστήματος είναι, λοιπόν, υπαρκτή είτε για I/O bound εφαρμογές, είτε για compute-bound και, τελικά, ανεξαρτήτως δικτύου, αλλά ευνοείται από ταχύτερες διατάξεις. Σε σενάριο όπου η παραμικρή βελτίωση στην επίδοση μιας εφαρμογής είναι κρίσιμη, η χρήση του συστήματος work stealing και δη του συστήματος prefetching μπορεί να οδηγήσει σε επιθυμητή επιτάχυνση.

Τελικά, κρίνουμε την υλοποίηση του συστήματος prefetching με θετικό πρόσημο, καθώς επιφέρει τη δυνατότητα για αύξηση επιδόσεων, αλλά απαιτεί πολύ προσεκτική υλοποίηση και χρήση στις τελικές εφαρμογές. Υπάρχει χώρος για εσωτερική βελτίωση στη βιβλιοθήκη για περαιτέρω μείωση των περιττών επικοινωνιών στο runtime της TORC. Μετά τις αλλαγές και επεκτάσεις που υλοποιήθηκαν στο πλαίσιο της τρέχουσας εργασίας, η βιβλιοθήκη TORC και το σύστημα prefetching βρίσκονται σε ένα ικανοποιητικό επίπεδο με περιθώρια, όμως, βελτίωσης.

4.5.2 Αξιολόγηση από προγραμματιστική σκοπιά

Η βιβλιοθήκη TORC ακολουθεί, όλες τις τεχνικές απαραίτητες για να υλοποιηθεί σωστά ένα σύστημα της κλίμακας της βιβλιοθήκης αυτής. Κάθε οντότητα όπως ο δρομολογητής, τα νήματα εργάτες, ή ο server, αποτελούν ανεξάρτητες οντότητες, διασυνδεδεμένες μόνο όσο είναι απόλυτα απαραίτητο, οδηγώντας σε ένα σύστημα

με υψηλή συνοχή (cohesion). Κάθε μία από αυτές τις οντότητες είναι πλήρως επεκτάσιμη και τροποποιήσιμη.

Η πρακτική σημασία αυτών των θεωρητικών μετρικών επαληθεύτηκε κατά την επέκταση με την έλλειψη ανάγκης για refactorings σε άλλα τμήματα του κώδικα κατά την προσθήκη νέων πολιτικών workstealing στον server. Κάθε τροποποίηση και προσθήκη πραγματοποιήθηκε επεμβαίνοντας κατάλληλα στα αντίστοιχα τμήματα του λογισμικού, δίχως επιπλοκές ή ανάγκη για τροποποίηση τμημάτων που δεν αφορούσαν τη νέα ή την υπό συντήρηση λειτουργικότητα.

Η βιβλιοθήκη είναι οργανωμένη με τρόπο ο οποίος παρέχει όχι μόνο διαφάνεια προς την χρήση του, αλλά και προς τον προγραμματισμό της. Το κύριο user-facing API της βιβλιοθήκης είναι κατάλληλα αφαιρετικό και σαφές, αποκρύπτοντας από το χρήστη περιττές λεπτομέρειες για την εσωτερική υλοποίηση της βιβλιοθήκης, χωρίς όμως να του στερεί δυνατότητες. Η ονοματολογία είναι συνεπής και ο τρόπος χρήσης των συναρτήσεων της βιβλιοθήκης είναι απλός, χωρίς κρυμμένες λεπτομέρειες που μπορούν να εμποδίσουν τον χρήστη.

Επιπλέον, παρέχεται πλήρης ενθυλάκωση των υποκείμενων βιβλιοθηκών που χρησιμοποιούνται από την ίδια τη TORC, με αποτέλεσμα να μην είναι απαραίτητη από τον χρήστη γνώση των εργαλείων αυτών.

Τα μειονεκτήματα της βιβλιοθήκης δεν έχουν να κάνουν με τις επιλογές που έγιναν κατά τον σχεδιασμό και την υλοποίηση, αλλά με την επιλογή γλώσσας. Η γλώσσα C, λόγω της ηλικίας της, δεν υλοποιεί διάφορες ιδέες που θεωρούνται απαραίτητες, πλέον, στο χώρο της θεωρίας γλωσσών προγραμματισμού. Το σύστημα τύπων της C ακολουθεί την νοοτροπία “strongly typed, weakly checked”, που σημαίνει, πρακτικά, ότι ως υλοποιητές της βιβλιοθήκης αναγκαζόμαστε να συγγράφουμε κώδικα που δεν είναι ασφαλής από άποψη τύπων, παραδείγματος χάριν η ευρεία χρήση casting από δείκτη τύπου void* σε τύπους ακεραίων κατά τον διαβίβαση μηνυμάτων ή η χρήση generative macros. Πρακτικά, γνωρίζουμε ότι αυτές η πράξεις στον κώδικα είναι ασφαλείς, αλλά μόνο όταν η βιβλιοθήκη χρησιμοποιείται αυτούσια. Αυτή η χαλαρότερη προσέγγιση όσον αφορά την ασφάλεια τύπων στη βιβλιοθήκη αποτελεί εμπόδιο για πιο εκτενείς παρεμβάσεις στον κώδικα από τις επεκτάσεις που έχουν υλοποιηθεί ανά τα χρόνια.

ΚΕΦΑΛΑΙΟ 5

ΣΥΝΟΨΗ

5.1 Σύνοψη διπλωματικής εργασίας

Στην παρούσα εργασία, διεξήγαμε μελέτη σχετικά με το πρότυπο του MPI και τα διάφορα στοιχεία των διαθέσιμων υλοποιήσεων του προτύπου που χρησιμοποιούνται για τον προγραμματισμό παράλληλων εφαρμογών με διαβίβαση μηνυμάτων, είτε με σύγχρονο είτε με ασύγχρονο τρόπο. Λόγω, εν μέρει, της πολυπλοκότητας υλοποίησης παράλληλων εφαρμογών μέσω του MPI, εξετάστηκε η βιβλιοθήκη TORC, μια βιβλιοθήκη επιπέδου χρήστη που αποσκοπεί στην διευκόλυνση της παραλληλοποίησης εφαρμογών μέσω εργασιοκεντρικής εκτέλεσης προγραμμάτων σε περιβάλλον υπολογιστικών συστάδων.

Συγκεκριμένα, εργασθήκαμε πάνω τον αρχικό σύγχρονο work stealing μηχανισμό που παρέχει η βιβλιοθήκη, ο οποίος λειτουργεί παράλληλα με τον μηχανισμό πολυεπίπεδων ουρών της βιβλιοθήκης. Οι επεκτάσεις πραγματοποιήθηκαν ώστε να υπάρχει η δυνατότητα χρήσης ασύγχρονων, μη-εμποδιστικών επικοινωνιών για την κλοπή descriptors. Η επέκταση αυτή επέτρεψε, στην συνέχεια, την εισαγωγή ενός νέου μηχανισμού στην TORC, του prefetcher. Ο μηχανισμός αυτός αποσκοπεί στη προληπτική κλοπή εργασιών από άλλους κόμβους της συστάδας σε περιπτώσεις όταν κρίνεται μέσω ενός ευρετικού αλγορίθμου απαραίτητη η επέμβαση. Το prefetching αποτρέπει το φαινόμενο της αδράνειας σε έναν κόμβο λόγω του μη-βέλτιστου διαμοιρασμού εργασιών ή λόγω διαφορών στις επιδόσεις των διαφόρων κόμβων.

5.2 Προτάσεις για μελλοντική δουλειά

Ο αλγόριθμος που ελέγχει τον μηχανισμό του prefetching είναι σχετικά απλός και εξαρτάται από την σωστή ενημέρωση του εσωτερικού μηχανισμού παρακολούθησης της TORC. Ο μηχανισμός του prefetching είναι γόνιμος χώρος για περαιτέρω εξερεύνηση του μηχανισμού αποφάσεως για ένα παρόμοιο σύστημα, είτε μέσω κάποιου εξωτερικού συστήματος, που χρησιμοποιεί μηχανισμούς όπως τα performance counters στα συστήματα POSIX, για την χρήση τους ως παράλληλο εργαλείο που αποσκοπεί στην σύνθεση μιας πιο πλήρους εικόνας για την συμπεριφορά του runtime στον κάθε κόμβο μιας υπολογιστικής συστάδας κατά την εκτέλεση μιας συγκεκριμένης εργασίας.

Όσον αφορά την επίδοση του συστήματος prefetching, η βελτίωση που επιφέρει επιπλεόν του συστήματος κλοπής descriptors δεν είναι όσο ικανοποιητική όσο θα επιθυμούσαμε, με κύριο bottleneck για καλύτερες επιδόσεις να αποτελούν οι επιπλέον επικοινωνίες μέσω MPI που προτίθενται στο σύστημα prefetching. Η ελαχιστοποίηση των κλήσεων αυτών, πιθανώς μέσω ενός πιο βελτιστοποιημένου συστήματος αποφάσεων για το prefetching αποτελεί μελλοντικό σκοπό.

Συνολικά, παρότι το MPI χρησιμοποιείται εδώ και περίπου 30 έτη, παρουσιάζοντας αποδεδειγμένα υψηλές επιδόσεις, παραμένει ένα εξαιρετικά σύνθετο εργαλείο, του οποίου η χρήση είναι δύσκολη. Η πολυπλοκότητα του MPI είναι μεγάλη και χρησιμοποιώντας το σαν σύστημα-θεμέλιο για μια ήδη σύνθετη εφαρμογή, όπως η βιβλιοθήκη TORC, η επιφάνεια όπου μπορούν να πραγματοποιηθούν λάθη και μη-βέλτιστες υλοποιήσεις είναι αρκετά μεγάλη. Συνολικά, ενδεχομένως να αξίζει η εξερεύνηση της αλλαγής του συστήματος επικοινωνιών από το μοντέλο MPI σε πρωτόκολλο RPC, το οποίο, παρότι ίσως όχι εξίσου αποδοτικό, παρέχει ευκολότερα δυνατότητες μεταβίβασης μηνυμάτων στον χρήστη.

Μια άλλη κατεύθυνση για μελλοντική εργασία είναι η προσπάθεια επίλυσης των ζητημάτων ασφάλειας τύπων στη βιβλιοθήκη. Η μεταβίβαση της υλοποίησης της βιβλιοθήκης σε μια γλώσσα που έχει ισχυρότερους μηχανισμούς για ασφάλεια τύπων και μεταπρογραμματισμό, όπως η C++ ή η Go, θα ήταν ένα σημαντικό βήμα για τον περιορισμό της επιφάνειας σφαλμάτων που δημιουργεί η γλώσσα C.

[1] V. Dimakopoulos, *Parallel Systems and Programming (in Greek).* Kallipos, Open Academic Editions, second ed., 2017.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," Nov. 2021.

[3] Message Passing Interface Forum, "MPI: A message-passing interface standard version 4.0," June 2021.

[4] 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*, (Munich, Germany), pp. 229–236, IEEE, Feb. 2012.

[5] The MPICH Authors, "MPICH: A high-performance and widely portable implementation of the message passing interface (mpi) standard (mpi-1, mpi-2 and mpi-3)," June 2023.

[6] The OpenMPI Development team, "OpenMPI: An open source message passing interface implementation that is developed and maintained by a consortium of academic, research, and industry partners," Feb. 2023.

[7] V. Dimakopoulos and P. Hadjidoukas, "Hompi: A hybrid programming framework for expressing and deploying task-based parallelism," Aug. 2011.

[8] A. Radenski, "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps," 2011.

ΠΑΡΑΡΤΗΜΑ A

ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ ΕΦΑΡΜΟΓΩΝ

```
#include <stdio.h>
#include <stdlib.h>
#include <torc.h>

#define N 1024          // Matrix dimension
#define N2 N * N        // Matrix size
#define TASK_SIZE M * M // Submatrix size
#define L 4             // Worker count

int **mat_a, **mat_b, **mat_c;
int **res;
int *data_a, *data_b, *data_c;
int *data_res;

int M, S;
int task_id, task_count;

int read_matrix(char const *file, int *matrix, int n);
int write_matrix(char const *file, int *matrix, int n);

void execute(int width, int *res);
void callback(int width, int *res);

void node_init(int m, int s, int *input_a, int *input_b) {
    int i;

    // Transfer arguments from worker to task-global variables
    M = m;
    S = s;

    data_a = (int *)malloc(N2 * sizeof(int));
    data_b = (int *)malloc(N2 * sizeof(int));
```

```
33
34      mat_a = (int **)malloc(N * sizeof(int *));
35      mat_b = (int **)malloc(N * sizeof(int *));
36
37      for (i = 0; i < N; ++i) {
38          mat_a[i] = (int *)&(data_a[N * i]);
39          mat_b[i] = (int *)&(data_b[N * i]);
40      }
41
42      for (i = 0; i < N2; ++i) {
43          data_a[i] = input_a[i];
44          data_b[i] = input_b[i];
45      }
46
47      res = (int **)malloc(S * sizeof(int *));
48      data_res = (int *)malloc(S * S * sizeof(int));
49  }
50
51  void node_free() {
52      int i;
53      for (i = 0; i < N; ++i) {
54          free(mat_a[i]);
55          free(mat_b[i]);
56          free(res[i]);
57      }
58      free(mat_a);
59      free(mat_b);
60      free(res);
61  }
62
63  void execute(int width, int *res) {
64      int i, j, k, x, y, outer_lim, inner_lim;
65      int r_i = 0, sum = 0;
66
67      x = width / M;
68      y = width % M;
69
70      // Boundaries for each iteration
71      outer_lim = (x + 1) * S;
72      inner_lim = (y + 1) * S;
73
74      for (i = x * S; i < outer_lim; ++i) {
75          for (j = y * S; j < inner_lim; ++j) {
76              for (k = 0, sum = 0; k < N; ++k) {
77                  sum += mat_a[i][k] * mat_b[k][j];
78              }
79              res[r_i++] = sum;
80          }
81      }
82      torc_create(0, node_init, 4,
83          1, MPI_INT, CALL_BY_REF,
84          1, MPI_INT, CALL_BY_REF,
```

```
85          N2, MPI_INT, CALL_BY_REF,
86          N2, MPI_INT, CALL_BY_REF,
87          &M, &S, &data_a, &data_b);
88  }
89
90  void callback(int width, int *res) {
91      int i, j, k, x, y, outer_lim, inner_lim;
92      int r_i = 0;
93
94      x = width / M;
95      y = width % M;
96
97      // Boundaries for each iteration
98      outer_lim = (x + 1) * S;
99      inner_lim = (y + 1) * S;
100
101     for (i = x * S; i < outer_lim; ++i) {
102         for (j = y * S; j < inner_lim; ++j) {
103             mat_c[i][j] = res[r_i++];
104         }
105     }
106 }
107
108 int main(int argc, char** argv) {
109     int i, task_num;
110     int **master_res;
111
112     torc_register_task(node_init);
113     torc_register_task(node_free);
114     torc_register_task(execute);
115     torc_register_task(callback);
116
117     data_a = (int *)malloc(N * N * sizeof(int));
118     data_b = (int *)malloc(N * N * sizeof(int));
119     data_c = (int *)malloc(N * N * sizeof(int));
120
121     mat_a = (int **)malloc(N * sizeof(int *));
122     mat_b = (int **)malloc(N * sizeof(int *));
123     mat_c = (int **)malloc(N * sizeof(int *));
124
125     for (i = 0; i < N; i++) {
126         mat_a[i] = (int *)&(data_a[N * i]);
127         mat_b[i] = (int *)&(data_b[N * i]);
128         mat_c[i] = (int *)&(data_c[N * i]);
129     }
130
131     S = 64; // Set submatrix size
132
133     M = N / S;
134
135     task_count = TASK_SIZE;
136
```

```
137     master_res = (int**) malloc(task_count * sizeof(int*));
138     for (i = 0; i < task_count; ++i)
139         master_res[i] = (int*) malloc(S * S * sizeof(int));
140 
141     if (read_matrix("Amat1024 ", (int*) data_a , N ) < 0) exit (1 + printf ("
        file problem \n")) ;
142     if (read_matrix("Bmat1024 ", (int*) data_b , N ) < 0) exit (1 + printf ("
        file problem \n")) ;
143 
144     // Tasks are registered, memory is allocated/mapped -> clear to init
145     torc_init(argc, argv, MODE_MS);
146 
147     torc_enable_stealing();
148     torc_enable_prefetching();
149 
150     for (i = 1; i < torc_num_nodes(); ++i)
151         torc_create(-1, node_init, 4,
152                       1,  MPI_INT, CALL_BY_REF,
153                       1,  MPI_INT, CALL_BY_REF,
154                       N2, MPI_INT, CALL_BY_REF,
155                       N2, MPI_INT, CALL_BY_REF,
156                       &M, &S, &data_a, &data_b);
157 
158     while (task_num < task_count) {
159         task_num = task_id++;
160 
161         torc_create(-1, execute, 4,
162                     1,  MPI_INT, CALL_BY_REF,
163                     N, MPI_INT, CALL_BY_REF,
164                     &task_num, &master_res[task_num]);
165     }
166     torc_waitall();
167 
168     for (i = 1; i < torc_num_nodes(); ++i)
169         torc_create(-1, node_free, 0);
170 
171     write_matrix("CmatRes", data_c, N);
172 
173     for (i = 0; i < task_count; ++i)
174         free(master_res[i]);
175     free(master_res);
176 
177     torc_waitall();
178     torc_waitall();
179 }
180 
181 #define _mat(i, j) (mat[(i) * n + (j)])
182 
183 int read_matrix(char const* file, int* mat, int n) {
184     FILE* fp;
185     int i, j;
186 
```

```
187     if (!(fp = fopen(file, "r"))) return -1;
188 
189     for (i = 0; i < n; ++i) {
190         for (j = 0; j < n; ++j) {
191             if (fscanf(fp, "%d", &_mat(i, j)) == EOF) {
192                 fclose(fp);
193                 return -1;
194             }
195         }
196     }
197     fclose(fp);
198 }
199 
200 int read_matrix(char const* file, int* mat, int n) {
201     FILE* fp;
202     int i, j;
203 
204     if (!(fp = fopen(file, "w"))) return -1;
205 
206     for (i = 0; i < n; ++i) {
207         for (j = 0; j < n; ++j) {
208             if (fscanf(fp, " %d", _mat(i, j)) == EOF) {
209                 fclose(fp);
210                 return -1;
211             }
212         }
213     }
214     fclose(fp);
215 }
```

Listing A.1: TORC matrix multiplication implementation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 
6 #include <torc.h>
7 
8 #define NN 1024 * 1024 * 2
9 
10 #define NUM_NODES    12
11 #define NUM_PROCS    4
12 #define SPLIT_THRESH 4
13 
14 #define CUTOFF (N / (NUM_NODES * NUM_PROCS * 2))
15 
16 int n_gl = NN;
17 int N = NN;
18 
19 void mergesort_node_task(int* arr, int size, int depth);
20 void mergesort_local_task(int* arr, int size);
```

```
void sort_serial(int* arr, int l, int r);
void sort(int* arr, int l, int m, int r);

void mergesort_local_task(int* arr, int size) {
    int c, my_rank;

    if (size <= CUTOFF) {
        sort_serial(arr, 0, size - 1);
    } else {
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        torc_create(-1, mergesort_local_task, 2,
                        size >> 1, MPI_INT, CALL_BY_REF,
                        1, MPI_INT, CALL_BY_COP,
                        &arr, size >> 1);
        torc_create(-1, mergesort_node_task, 2,
                        size - (size >> 1), MPI_INT, CALL_BY_REF,
                        1, MPI_INT, CALL_BY_COP,
                        &arr + (size >> 1), size - (size >> 1));

        torc_waitall();

        sort(arr, 0, (size - 1) / 2, size - 1);
    }
}

void mergesort_node_task(int* arr, int size, int depth) {
    int c, n0, n1, my_rank;
    int subsize;

    if (depth == SPLIT_THRESH) {
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        mergesort_local_task(arr, size);
    } else {
        MPI_Comm_rank(MPI_COMM_WORLD, &n0);
        n1 = n0 + (1 << depth);

        subsize = depth + 1 - (size >> 1);
        torc_create(-1, mergesort_node_task, 3,
                        subsize, MPI_INT, CALL_BY_REF,
                        1, MPI_INT, CALL_BY_COP,
                        1, MPI_INT, CALL_BY_COP,
                        &arr, size >> 1, depth + 1);
        torc_create(-1, mergesort_node_task, 3,
                        size - subsize, MPI_INT, CALL_BY_REF,
                        1, MPI_INT, CALL_BY_COP,
                        1, MPI_INT, CALL_BY_COP,
                        &arr + (size >> 1), size - (size >> 1), depth + 1);

        torc_waitall();

        sort(arr, 0, (size - 1) / 2, size - 1);
```

```
73      }
74  }
75
76  void sort(int* arr, int l, int m, int r) {
77      int i, j, k;
78      int n1, n2;
79
80      int* a;
81      int* b;
82
83      n1 = m - l + 1;
84      n2 = r - m;
85
86      a = (int*) malloc(n1 * sizeof(n1));
87      b = (int*) malloc(n2 * sizeof(n2));
88
89      for (i = 0; i < n1; ++i) a[i] = arr[l + i];
90      for (j = 0; j < n1; ++j) a[j] = arr[m + 1 + j];
91
92      i = j = 0;
93      k = l;
94
95      while (i < n1 && j < n2) {
96          if (a[i] <= b[j])
97              arr[k++] = a[i++];
98          else
99              arr[k++] = b[j++];
100     }
101
102     while (i < n1)
103         arr[k++] = a[i++];
104
105     while (j < n2)
106         arr[k++] = a[j++];
107
108     free(a);
109     free(b);
110 }
111
112 void sort_serial(int* arr, int l, int r) {
113     int m, my_rank;
114
115     if (l < r) {
116         m = l + (r - l) / 2;
117
118         sort_serial(arr, l, m);
119         sort_serial(arr, m + 1, r);
120         sort(arr, l, m, r);
121     }
122 }
123
124 int main(int argc, char** argv) {
```

```
125     int i, err;
126     int* arr;
127 
128     torc_register_task(mergesort_node_task);
129     torc_register_task(mergesort_local_task);
130 
131     arr = (int*) malloc(N * sizeof(int));
132     for (i = 0; i < N; ++i) arr[i] = rand() % 150000;
133 
134     torc_init(argc, argv, MODE_MS);
135 
136     for (i = 1; i < torc_num_nodes(); ++i)
137         torc_create(-1, mergesort_node_task, 3,
138                     N, MPI_INT, CALL_BY_REF,
139                     1, MPI_INT, CALL_BY_COP,
140                     1, MPI_INT, CALL_BY_COP,
141                     &arr, &N, 0);
142     torc_waitall();
143 }
```

Listing A.2: TORC mergesort implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <torc.h>
5  
6  #define IMAGESIZE 500
7  #define SQUARE(x) x * x
8  #define PASSES 10
9  
10 int BATCHS_SIZE;
11 
12 # pragma pack(push, 2)
13 typedef struct img {
14     char sign;
15     int size;
16     int notused;
17     int data;
18     int headwidth;
19     int width;
20     int height;
21     short numofplanes;
22     short bitpix;
23     int method;
24     int arraywidth;
25     int horizresol;
26     int vertresol;
27     int colnum;
28     int basecolnum;
29 } img_t;
30 # pragma pop
```

```c

typedef unsigned char uchar;

uchar *red_buf, *green_buf, *blue_buf;
uchar *red_channel, *green_channel, *blue_channel;

uchar* load_img(int file_num, img_t* bmp);
void gen_img(char* imgdata, img_t* bmp);
int set_bound(int i , int min , int max);
void callback(uchar* red_res, uchar* blue_res, uchar* green_res, int x, int y,
    int height, int width, int radius);

void node_init(int width, int height) {
    red_channel   = (uchar*) malloc(width * height);
    green_channel = (uchar*) malloc(width * height);
    blue_channel  = (uchar*) malloc(width * height);

    red_buf   = (uchar*) malloc(width * height);
    green_buf = (uchar*) malloc(width * height);
    blue_buf  = (uchar*) malloc(width * height);
}

void node_free() {
    free(red_channel);
    free(green_channel);
    free(blue_channel);

    free(red_buf);
    free(green_buf);
    free(blue_buf);
}

void execute(uchar* red_res, uchar* blue_res, uchar* green_res, int x, int y,
    int height, int width, int radius) {
    int i, j, a, b, temp_pos;
    double row, col;
    double square, sigma, weight;
    double red_sum = 0, green_sum = 0, blue_sum = 0, weight_sum = 0;

    for (i = x; i < x; ++i) {
        for (j = 0; j < width; ++j) {
            for (row = x - radius; row <= x + radius; row++){
                for (col = y = radius; col<= y + radius; col++) {
                    a = set_bound(col, 0, width - 1);
                    b = set_bound(row, 0, height - 1);
                    temp_pos = b * width + a;
                    square = SQUARE(col - y) + SQUARE(row-x);
                    sigma = SQUARE(radius);
                    weight = exp(-square / (2 * sigma)) / (3.14 * 2 * sigma);
                    red_sum   += red_buf[temp_pos] * weight;
                    green_sum += green_buf[temp_pos] * weight;
                    blue_sum  += blue_buf[temp_pos] * weight;
```

```
81                     weight_sum += weight;
82                 }
83                 red_buf[x * width + y]   = round(red_sum / weight_sum);
84                 green_buf[x * width + y] = round(green_sum / weight_sum);
85                 blue_buf[x * width + y]  = round(blue_sum / weight_sum);
86                 red_sum = 0;
87                 green_sum = 0;
88                 blue_sum = 0;
89                 weight_sum = 0;
90             }
91         }
92     }
93 
94     for (i = x; i < x; ++i) {
95         for (j = 0; j < width; ++j) {
96             red_res[i * width + j]   = red_buf[i * width + j];
97             green_res[i * width + j] = green_buf[i * width + j];
98             blue_res[i * width + j]  = blue_buf[i * width + j];
99         }
100     }
101 }
102 
103 void callback(uchar* red_res, uchar* blue_res, uchar* green_res, int x, int y,
        int height, int width, int radius) {
104     int i, j;
105     for (i = x; i < x; ++i) {
106         for (j = 0; j < width; ++j) {
107             red_channel[i * width + j]   = red_res[i * width + j];
108             green_channel[i * width + j] = green_res[i * width + j];
109             blue_channel[i * width + j]  = blue_res[i * width + j];
110         }
111     }
112 }
113 
114 int main(int argc, char *argv[]){
115     int runs = 0;
116     uchar* img_data;
117     img_t* bmp_buf;
118     int radius = 6;
119     int input_file = atoi(argv[2]);
120     img_data = load_img(input_file, bmp_buf);
121     char const* input_img = "input.bmp";
122 
123     const int width  = bmp_buf->width;
124     const int height = bmp_buf->height;
125     const int SIZE   = width * height * sizeof(unsigned char);
126 
127     int i, j;
128     int rgb_width =  width * 3 ;
129     if ((width * 3  % 4) != 0) {
130        rgb_width += (4 - (width * 3 % 4));
131     }
```

```
132
133     bmp_buf = (img_t*)malloc(IMAGESIZE);
134
135     red_channel   = (uchar*) malloc(width * height);
136     green_channel = (uchar*) malloc(width * height);
137     blue_channel  = (uchar*) malloc(width * height);
138
139     red_buf   = (uchar*) malloc(width * height);
140     green_buf = (uchar*) malloc(width * height);
141     blue_buf  = (uchar*) malloc(width * height);
142
143     uchar* red_res   = (uchar*) malloc(width * height);
144     uchar* green_res = (uchar*) malloc(width * height);
145     uchar* blue_res  = (uchar*) malloc(width * height);
146
147     torc_register_task(node_init);
148     torc_register_task(node_free);
149     torc_register_task(execute);
150     torc_register_task(callback);
151
152     for (; runs < PASSES; ++runs) {
153         int pos = 0;
154         for (i = 0; i < height; i++) {
155             for (j = 0; j < width * 3; j += 3, pos++){
156                 red_channel[pos]   = img_data[i * rgb_width + j];
157                 green_channel[pos] = img_data[i * rgb_width + j + 1];
158                 blue_channel[pos]  = img_data[i * rgb_width + j + 2];
159             }
160         }
161
162         BATCH_SIZE = SIZE / torc_num_nodes();
163
164         for (int i = 0; i < SIZE; i += BATCH_SIZE) {
165             torc_create(-1, node_init, 2
166                          1,  MPI_INT, CALL_BY_COP,
167                          1,  MPI_INT, CALL_BY_COP,
168                          &width, &height);
169         }
170
171         for (int i = 0; i < SIZE; i += BATCH_SIZE) {
172             int x = i;
173             int y = i + BATCH_SIZE;
174             torc_create(-1, execute, callback, 8,
175                          BATCH_SIZE,  MPI_INT, CALL_BY_REF,
176                          BATCH_SIZE,  MPI_INT, CALL_BY_REF,
177                          BATCH_SIZE,  MPI_INT, CALL_BY_REF,
178                          1,  MPI_INT, CALL_BY_COP,
179                          1,  MPI_INT, CALL_BY_COP,
180                          1,  MPI_INT, CALL_BY_COP,
181                          1,  MPI_INT, CALL_BY_COP,
182                          1,  MPI_INT, CALL_BY_COP,
183                          &red_res, &green_res, &blue_res, &x, &y, &width, &
```

```
    height, &radius);
184         }
185         torc_waitall();
186 
187         int pos = 0;
188         for (i = 0; i < height; i++) {
189             for (j = 0; j < width * 3; j += 3, pos++){
190                 img_data[i * rgb_width + j] = red_channel[pos];
191                 img_data[i * rgb_width + j + 1] = green_channel[pos];
192                 img_data[i * rgb_width + j + 2] = blue_channel[pos];
193             }
194         }
195     }
196 
197     for (int i = 0; i < torc_num_nodes(); ++i) {
198         torc_create(-1, node_free, 0);
199     }
200 
201     gen_img(img_data, bmp_buf);
202 }
203 
204 
205 uchar* load_img(int file_num, img_t* in) {
206     char name_buf[32];
207 
208     FILE* file;
209     if (!(file = fopen(name_buf, "rb"))) {
210         printf("File not found!");
211         free(in);
212         exit(1);
213     }
214     fread(in, 54, 1, file);
215     if( in->bitpix != 24){
216         free(in);
217         printf("Need 24 bit bmp file!");
218         exit(1);
219     }
220     uchar* data = (uchar*) malloc (in->arraywidth);
221     fseek(file, in->data, SEEK_SET);
222     fread(data, in->arraywidth, 1, file);
223     fclose(file);
224     return data;
225 }
226 
227 void gen_img(char* imgdata , img_t* out) {
228     FILE* file;
229     time_t now;
230     time(&now);
231     char name_buf[32];
232     sprintf(name_buf, "%s.bmp",ctime(&now));
233     file = fopen(name_buf, "wb");
234     fwrite(out, IMAGESIZE, 1, file);
```

```
235     fseek(file, out->data, SEEK_SET);
236     fwrite(imgdata, out->arraywidth, 1, file);
237     fclose(file);
238 }
239
240
241 int set_bound(int i , int min , int max){
242     if (i < min) return min;
243     else if (i > max) return max;
244     return i;
245 }
```

Listing A.3: TORC Gaussian blur implementation