Εκτέλεση πολυ-εργασιακού κώδικα σε συστάδες υπολογιστών

Χρήστος Καραμπεάζης-Παπαδάκης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

Πανεπιστήμιο Ιωαννίνων

Σεπτέμβριος 2019

Περιεχομενα

Κατάλογος Σχημάτων **v**

Κατάλογος Πινάκων **vi**

Κατάλογος Αλγορίθμων **vii**

Περίληψη **viii**

Abstract **ix**

1 Εισαγωγή **1**

1.1 Μια ανασκόπηση της Ιστορίας των Υπολογιστικών Συστημάτων 1

1.2 Οργανώσεις και προγραμματισμός Παράλληλων Συστημάτων 3

1.2.1 Συστήματα Κοινόχρηστης Μνήμης (Shared Memory) 3

1.2.2 Συστήματα κατανεμημένης μνήμης και συστάδες (Distributed Memory and Clusters) 4

1.3 Παράλληλος Προγραμματισμός ανά αρχιτεκτονική 5

1.3.1 Συστήματα κοινόχρηστης μνήμης (shared memory) 5

1.3.2 Συστήματα κατανεμημένης μνήμης (distributed memory) 6

1.3.3 Μοντέλο συντονιστή-εργάτη 7

1.3.4 Μοντέλο προγραμματισμού με tasks (εργασίες) 8

1.4 Αντικείμενο Διπλωματικής Εργασίας 9

1.5 Δομή Διπλωματικής Εργασίας 10

2 Επισκόπηση του HOMPI Framework **11**

2.1 Το προγραμματιστικό μοντέλο HOMPI 11

2.1.1 Επισκόπηση των λειτουργιών και της δομής του HOMPI 12

2.1.2 Ο παραλληλοποιητικός μεταφραστής OMPi 14

2.1.3 Η βιβλιοθήκη TORC 17
2.1.4 Το μοντέλο εκτέλεσης νημάτων δύο επιπέδων 18
2.1.5 Η διεπαφή OMPi-TORC 21

3 Επεκτάσεις του προγραμματιστικού μοντέλου 23

3.1 Επεκτάσεις του μεταγλωττιστή 23
3.1.1 Εκτέλεση task ως απλή συνάρτηση 23
3.1.2 Υλοποίηση της οδηγίας ”If” του προτύπου OpenMP στον HOMPI 24
3.1.3 Οι ειδικές περιπτώσεις ”here” και ”remote” της φράσης atnode 25
3.1.4 Ασύγχρονη εκτέλεση κώδικα αρχικοποίησης 26
3.2 Επεκτάσεις της βιβλιοθήκης εκτέλεσης TORC 27
3.2.1 Ο δρομολογητής εργασιών EFTeQ (Expandable/Flexible TORC enQueuer) 27
3.2.2 Ο αλγόριθμος δρομολόγησης του EFTeQ 30
3.3 Σύνταξη προσθήκων των HOMPI και EFTeQ 33
3.3.1 Οι φράσεις οδηγιών atnode(here) και atnode(remote) 33
3.3.2 Η φράση οδηγιών if 33
3.3.3 Σύνταξη αρχείου ρυθμίσεων συστάδας του EFTeQ 34
3.3.4 Οι φράσεις hint του δρομολογητή EFTeQ 34

4 Εφαρμογές 36

4.1 Εφαρμογές ταξινόμησης μονοδιάστατων πινάκων 36
4.1.1 Merge Sort 36
4.1.2 Παραλληλοποιημένη συγχώνευση (Merge) 39
4.1.3 Bitonic/Quick Sort 41
4.1.4 Γρήγορη Ταξινόμηση (Quicksort) 41
4.2 Κατανεμημένος πολλαπλασιασμός δισδιάστατων πινάκων 42
4.3 Αναγνώριση προσώπων 44

5 Πειραματικές εκτελέσεις και συμπεράσματα 46

5.1 Πολλαπλασιασμός τετραγωνικών πινάκων 47
5.1.1 Συμπεριφορά σε περιβάλλον κοινόχρηστης μνήμης 48
5.1.2 Sun Cluster 52
5.1.3 EC2 Cluster 54
5.2 Τροποποιημένη Merge Sort 57

5.2.1 Sun Cluster 57
5.2.2 EC2 Cluster 58
5.3 Bitonic Sort 59
5.4 Αναγνώριση προσώπων 60
5.4.1 Επιδόσεις κοινόχρηστης μνήμης 60
5.4.2 Επιδόσεις κατανεμημένης εκτέλεσης 65
5.5 Θεωρητική Ανάλυση κατανεμημένης εκτέλεσης 68

6 Αξιολόγηση του HOMPI Framework 76
6.1 Τα εργαλεία προγραμματισμού από την σκοπιά της Τεχνολογίας Λογισμικού 76
6.1.1 Συντηρησιμότητα και Επεκτασιμότητα 76

7 Προγραμματισμός με Tasks για Παράλληλη και Κατανεμημένη επεξεργασία 80
7.1 Δυσκολίες του task based μοντέλου προγραμματισμού σε αλγορίθμους ανάλυσης γραφημάτων 81
7.1.1 Μη βέλτιστες περιπτώσεις κατανεμημένης εκτέλεσης 81
7.1.2 Σειριοποίηση πολύπλοκων δεδομένων 87
7.1.3 Αφαίρεση μη απαραίτητων πράξεων συγχρονισμού 89
7.1.4 Ασύγχρονη εκτέλεση κώδικα αρχικοποίησης - εκτίμηση 91
7.2 Λύσεις στα προβλήματα της κατανεμημένης εκτέλεσης 92
7.2.1 Εκμετάλλευση της τοπολογίας του δικτύου 93
7.3 Θεωρητικά εναντίον αληθινών σεναρίων 94

8 Επίλογος 97
8.1 Σύνοψη διπλωματικής εργασίας 97
8.2 Προτάσεις για μελλοντική εργασία 98

Βιβλιογραφία 99

A Πηγαίος κώδικας εφαρμογών 101
A.1 Merge sort 101
A.2 Παραλληλοποιημένη συγχώνευση (Merge) 104
A.3 Bitonic/Quick Sort 106
A.4 Γρήγορη ταξινόμηση (Quicksort) 108

A.5 Κατανεμημένος πολλαπλασιασμός πινάκων 109
A.6 Αναγνώριση προσώπων 113
A.6.1 main.c 113
A.6.2 lolac.c 116
A.6.3 Convolver.c 120

B Περιβάλλον πειραμάτων **122**
B.1 Απαιτήσεις λογισμικού 122
B.2 Sun Cluster 123
B.3 EC2 Cluster 124
B.4 Opti7020 124
B.5 Προσωπικός Η/Υ 124
B.6 Raspberry Pi 4 125

Καταλογος Σχηματων

2.1 Η διαδικασία μεταγλώττισης του HOMPI. 15
2.2 Το μοντέλο εκτέλεσης δύο επιπέδων νημάτων 19

3.1 Προσομοιωμένη κατάσταση συστήματος 31

4.1 Το δίκτυο ταξινόμησης του αλγορίθμου σε cluster 8 κόμβων. 39

5.1 Επιτάχυνση πολλαπλασιασμού πινάκων σε διαφορετικά συστήματα . 47
5.2 Επιδόσεις πολλαπλασιασμού πινάκων στον Sun Cluster 52
5.3 Επιδόσεις πολλαπλασιασμού πινάκων στον EC2 Cluster 55
5.4 Επιδόσεις Merge Sort στον Sun Cluster 56
5.5 Επιδόσεις Merge Sort στον EC2 Cluster 58
5.6 Επιδόσεις Bitonic Sort στον Sun Cluster 59
5.7 Επιδόσεις Bitonic Sort στον EC2 Cluster 60
5.8 Επιδόσεις σε Batch processing 160 εικόνων 61
5.9 Επιτάχυνση στην εικόνα ”Class57” 62
5.10 Επιτάχυνση στην εικόνα ”The many faces of Anthony Hopkins” 62
5.11 Επιτάχυνση ανά αριθμό πυρήνων 64
5.12 Επιδόσεις Face Detection στον Sun Cluster 65
5.13 Αποδοτικότητα ανά αριθμό κόμβων 65
5.14 Επιδόσεις Face Detection στον EC2 Cluster 66
5.15 Αποδοτικότητα ανά αριθμό κόμβων 66
5.16 Σχηματική ανάλυση παράλληλης εκτέλεσης σε κοινόχρηστη μνήμη . . 68
5.17 Σχηματική ανάλυση κατανεμημένης εκτέλεσης με δύο κόμβους 69

Καταλογος Πινακων

Καταλογος Αλγοριθμων

3.1 Παραγωγή κώδικα υπό την ύπαρξη If clause 25

7.1 Αλγόριθμος του Prim 82

7.2 Κατανεμημένος εσωτερικός βρόγχος 86

7.3 Ο αλγόριθμος κάθε εργασίας 87

7.4 Ο αλγόριθμος των callbacks 87

7.5 Διαδικασία προετοιμασίας προγράμματος 91

7.6 Κώδικας στην έναρξη εργασίας 91

Περιληψη

Τα παράλληλα συστήματα υπολογιστών έχουν ενσωματωθεί πλέον πλήρως στον τομέα της Πληροφορικής και της Μηχανικής Ηλεκτρονικών Υπολογιστών. Η εργασία αυτή παρουσιάζει μία μελέτη πάνω στην εκτέλεση εφαρμογών που χρησιμοποιούν τη μέθοδο παραλληλισμού με τη χρήση tasks (εργασιών) σε πολυπύρηνα συστήματα και συστάδες πολυπύρηνων υπολογιστικών συστημάτων, καθώς και τη μελέτη και επέκταση των εργαλείων προγραμματισμού πάνω στα οποία αναπτύχθηκαν οι εφαρμογές αυτές. Στην αρχή της εργασίας παρουσιάζονται τα εργαλεία αυτά και η δομή τους, συνεχίζοντας με τις νέες προσθήκες. Στη συνέχεια, παρουσιάζεται μία συλλογή από εφαρμογές οι οποίες αναπτύχθηκαν χρησιμοποιώντας τα εργαλεία αυτά, μαζί με τα πλεονεκτήματα και μειονεκτήματά που προέκυψαν κατά την ανάπτυξή τους συνοδευόμενα από μία αξιολόγηση των εργαλείων και της μεθόδου παραλληλισμού. Τέλος, δίνουμε τις παρατηρήσεις και τα συμπεράσματά μας από την πειραματική εκτέλεση των εφαρμογών αυτών σε πολλαπλά διαφορετικά συστήματα και περιβάλλοντα εκτέλεσης.

Abstract

Christos Karampeazis-Papadakis, Diploma, Department of Computer Science and Engineering, University of Ioannina, Greece, September 2019.
Execution of task-based code on computer clusters.
Advisor: Vassilios Dimakopoulos, Associate Professor.

Parallel computing systems have become ubiquitous in the world of Computer Science and Engineering in the modern age. In this thesis we present a study on the execution of applications utilizing task-based parallelism in multi-core and clusters of multi-core systems, as well as an evaluation and some extensions of the framework on which these applications were developed. In the beginning of this work we shall present said framework and its components, followed by the aforementioned extensions. Next, we present a collection of applications developed utilizing this framework along with the advantages and disadvantages which came up during the development phase, accompanied by an evaluation of the framework itself. Finally, we will present our observations and conclusions on the experimental execution of this set of applications in a multitude of systems and configurations.

Κεφαλαιο 1

Εισαγωγη

1.1 Μια ανασκόπηση της Ιστορίας των Υπολογιστικών Συστημάτων

1.2 Οργανώσεις και προγραμματισμός Παράλληλων Συστημάτων

1.3 Παράλληλος Προγραμματισμός ανά αρχιτεκτονική

1.4 Αντικείμενο Διπλωματικής Εργασίας

1.5 Δομή Διπλωματικής Εργασίας

1.1 Μια ανασκόπηση της Ιστορίας των Υπολογιστικών Συστημάτων

Η περιέργεια του ανθρώπου για το άγνωστο ξεκίνησε από τη στιγμή που ανέπτυξε την ίδια τη λογική, περιέργεια η οποία δεν φαίνεται να σταματά να αναζητά ποτέ το άγνωστο και την αλήθεια. Σπουδαιότερο εργαλείο αυτής της αναζήτησης ήταν πάντα οι επιστήμες, σε οποιαδήποτε μορφή και στάδιο εξέλιξής τους. Οι επιστήμες έδωσαν στον άνθρωπο δυνατότητες από το να βελτιώσει τη ζωή του σε καθημερινή βάση, έως το να προσγειωθεί στο φεγγάρι, πριν από πέντε δεκαετίες.

Όπως όλες οι επιστήμες, έτσι και η επιστήμη της Πληροφορικής έχει αναπτύξει μέσα που τη βοηθούν να επιτύχει τους στόχους της. Τα μέσα αυτά μπορούν να συναντηθούν χιλιάδες χρόνια πίσω στην ιστορία, όταν δεν υπήρχε καν η έννοια των επιστημών, σε πρώιμα στάδια των εργαλείων αριθμητικής όπως το Ishango Bone. Έπειτα από αιώνες ανάπτυξης των επιστημών και συλλογικής ανθρώπινης

προσπάθειας, η επιστήμη της Πληροφορικής άρχισε να θεμελιώνεται κατά τα μέσα του 19ου αιώνα, χάρη σε σπουδαίους επιστήμονες όπως ο Alan Turing, του οποίου το μαθηματικό μοντέλο έθεσε τη βάση για τους σύγχρονους υπολογιστές, όπως αυτό περιγράφθηκε στην εργασία του εν ονόματι "On Computable Numbers, with an application to the Entscheidungsproblem".

Έπειτα από τα απαραίτητα πρώτα βήματα, άρχισαν να εμφανίζονται οι πρώτοι ηλεκτρονικοί υπολογιστές. Η πρώτη γενιά αυτών χρησιμοποιούσε λυχνίες κενού, οι οποίες τους καθιστούσαν εξαιρετικά απαιτητικούς σε ενέργεια, ογκώδεις και αναξιόπιστους. Στην αρχή, όλες οι βασικές ιδέες που είναι δεδομένες εδώ και δεκαετίες ήταν ανύπαρκτες στην εποχή αυτήν. Ο προγραμματισμός γινόταν απευθείας στο υλικό, η είσοδος και η έξοδος των προγραμμάτων ήταν κατά κάποιαν έννοια ακόμα χειροκίνητη και δεν υπήρχε ακόμα η έννοια των απαραίτητων πια λειτουργικών συστημάτων. Η έννοια του παραλληλισμού δεν υπήρχε ακόμη, για αυτόν τον λόγο αυτά τα συστήματα έπρεπε να δεσμεύουν εκ των προτέρων τον διαθέσιμο υπολογιστικό χρόνο τους.

Χάρη στην εφεύρεση του τρανσίστορ το 1947, οι υπολογιστές έγιναν μικρότεροι, πιο αξιόπιστοι και η εξέλιξη που ακολούθησε επέτρεψε έννοιες όπως αυτή των μεταγλωττιστών και των λειτουργικών συστημάτων. Παρόλο που η έννοια της παράλληλης επεξεργασίας δεν ήταν πλήρως άγνωστη χάρη σε διάφορους μαθηματικούς εδώ και αιώνες, ο πρώτος παράλληλος υπολογιστής έκανε την εμφάνισή του σε αυτή τη γενιά, το 1962, με άλλους να ακολουθούν. Σε αυτή την γενιά επίσης ο Gordon Moore εξέφρασε τον γνωστό νόμο του Moore, ο οποίος δηλώνει ότι κάθε περίπου ενάμιση έτος, το πλήθος των τρανζίστορ ανά επεξεργαστική μονάδα διπλασιάζεται. Ο νόμος του Moore είναι άμεση συνεπαγωγή του γεγονότος ότι άλλοι τομείς των επιστημών όπως η Φυσική και η Χημική Μηχανική καταφέρνουν να σμικρύνουν το μέγεθος των τρανζίστορ σε μεγέθη τα οποία κατά τη συγγραφή αυτής της εργασίας βρίσκονται στα 7 νανόμετρα σε εμπορικό επίπεδο, με αρχιτεκτονικές των 5 νανομέτρων να είναι σύντομα εμπορικά διαθέσιμες.

Η σμίκρυνση των τρανσίστορ έχουν ως επιπλέον πλεονέκτημα σε γενικές γραμμές τη μειωμένη κατανάλωση ισχύος, με αποτέλεσμα να μπορεί να αυξηθεί η υπολογιστική ισχύς ανά επεξεργαστική μονάδα αυξάνοντας τον ρυθμό ρολογιού, πέρα από την αύξηση του υλικού ανά επεξεργαστική μονάδα. Δυστυχώς όμως, υπάρχει ένα όριο στους ρυθμούς ρολογιού που μπορούν να επιτευχθούν, καθώς υψηλοί τέτοιοι ρυθμοί συνεπάγονται μεγάλη κατανάλωση ισχύος, ασταθές και αναξιόπιστο υλικό

και υψηλότερους λόγους ρευμάτων διαρροής προς ενεργή ισχύ στα μικροηλεκτρονικά κυκλώματα αυτά.

Ως αποτέλεσμα, έπειτα από δεκαετίες σταθερής αύξησης του ρυθμού ρολογιού και τρανζίστορ ανά κύκλωμα, ο πρώτος παράγοντας πιθανόν έχει φτάσει το ανώτατο όριό του, τουλάχιστον για το προβλέψιμο μέλλον. Καθώς ο δεύτερος παράγοντας συνέχισε και συνεχίζει να αυξάνεται, η λύση για αυτό το πρόβλημα ήταν τα παράλληλα συστήματα, συστήματα τα οποία μπορούν να εκτελέσουν ταυτόχρονα κώδικα σε πολλαπλές επεξεργαστικές μονάδες.

Τα παράλληλα συστήματα υπήρχαν σε διάφορες μορφές τους στην ιστορία των υπολογιστών, αλλά σήμερα πια συναντώνται σε καθημερινή βάση και είναι εξαιρετικά δύσκολο έως και απίθανο να συναντήσει κάποιος σύγχρονα συστήματα με μονοπύρηνες επεξεργαστικές μονάδες. Ενώ όμως χάρη στην έννοια του πολυπρογραμματισμού δέσμης και του χρονομερισμού οι σημερινοί υπολογιστές ευνοούνται από τις πολλαπλές επεξεργαστικές μονάδες εκτελώντας διαφορετικές διεργασίες πολλαπλών χρηστών στο ίδιο λειτουργικό σύστημα, η εκμετάλλευση πολλαπλών πυρήνων για μία συγκεκριμένη εφαρμογή είναι ένα μεγάλο ζήτημα, το οποίο δεν έχει συγκεκριμένη, καθολική λύση.

1.2 Οργανώσεις και προγραμματισμός Παράλληλων Συστημάτων

1.2.1 Συστήματα Κοινόχρηστης Μνήμης (Shared Memory)

Τα συστήματα κοινόχρηστης μνήμης είναι συστήματα που αποτελούνται από πολλαπλές επεξεργαστικές μονάδες, οι οποίες επικοινωνούν με τη μνήμη μέσω ενός κοινόχρηστου μέσου, στο οποίο είναι άμεσα συνδεδεμένοι. Στην αρχιτεκτονική αυτή όλοι οι επεξεργαστές έχουν διαθέσιμη ολόκληρη τη μνήμη του συστήματος, την οποία μοιράζονται μεταξύ τους. Η μνήμη, για λόγους που θα περιγράψουμε παρακάτω, ενδεχομένως να είναι χωρισμένη σε τμήματα.

Το κοινόχρηστο μέσο είναι συνήθως ένας δίαυλος, ή ένα διακοπτικό δίκτυο διασύνδεσης όπως ένα δίκτυο baseline ή διασταυροτικοί διακόπτες [1]. Η πιο συνηθισμένη περίπτωση είναι αυτή του διαύλου (bus), όπου στον κοινό δίαυλο μεταφέρονται όλα τα απαραίτητα δεδομένα όπως διευθύνσεις μνήμης και σήματα ελέγχου και κάθε επεξεργαστής μπορεί να παρακολουθεί την κίνηση σε αυτό τον δίαυλο.

Αυτή η συνδεσμολογία, παρά το γεγονός ότι έχει κυριαρχήσει, φέρει ένα σημαντικό μειονέκτημα που της απαγορεύει να κλιμακωθεί για χρήση με μεγάλο αριθμό επεξεργαστών. Σε οποιαδήποτε δεδομένη χρονική στιγμή, μόνο ένας επεξεργαστής μπορεί να επικοινωνεί με ένα τμήμα της μνήμης. Ως συνέπεια, οποιοσδήποτε άλλος επεξεργαστής επιθυμεί να εκτελέσει κάποια πρόσβαση μνήμης, θα πρέπει να περιμένει μέχρις ότου να ελευθερωθεί ο δίαυλος. Ως αποτέλεσμα αυτού, η συνδεσμολογία διαύλου συναντάται κυρίως σε συστήματα με σχετικά μικρό αριθμό επεξεργαστών (το πολύ 10).

Για να υλοποιηθεί ένα σύστημα με μεγαλύτερο αριθμό επεξεργαστών, συνήθως καταφεύγουμε σε άλλες συνδεσμολογίες που έχουν υψηλότερο ποσοστό αποδοχής των αιτημάτων των επεξεργαστών, όπως διακοπτικών δικτύων (π.χ. δίκτυα Δέλτα). Σε τέτοια δίκτυα είναι δυνατόν παραπάνω από ένα ζεύγος επεξεργαστή-τμήματος μνήμης να επικοινωνούνε ταυτόχρονα, με αποτέλεσμα να επιτυγχάνεται πολύ μικρότερο ποσοστό απόρριψης αιτημάτων των επεξεργαστών και κατά συνέπεια μεγαλύτερα ποσοστά αξιοποίησης των πολλαπλών επεξεργαστών.

1.2.2 Συστήματα κατανεμημένης μνήμης και συστάδες (Distributed Memory and Clusters)

Στα συστήματα κατανεμημένης μνήμης, κάθε επεξεργαστής είναι συνδεδεμένος άμεσα με ένα μονάχα τμήμα της συνολικής μνήμης του συστήματος, με την οποία μπορεί να επικοινωνήσει μόνο αυτός ο επεξεργαστής. Σε αυτή την αρχιτεκτονική, για να προσπελάσει ένας επεξεργαστής κάποιο τμήμα μνήμης διαφορετικό από αυτό στο οποίο είναι άμεσα συνδεδεμένος, πρέπει να το κάνει μεταφέροντας ένα μήνυμα-αίτηση στον επεξεργαστή ο οποίος είναι συνδεδεμένος με το τμήμα μνήμης το οποίο επιθυμεί να προσπελάσει, και στη συνέχεια να λάβει μία απάντηση. Σε αυτή την αρχιτεκτονική αποκαλούμε κάθε ζεύγος επεξεργαστή-μνήμης *κόμβο*.

Στις υλοποιήσεις αυτής της αρχιτεκτονικής κάθε κόμβος μπορεί να είναι ένα ανεξάρτητο σύστημα, ή ένα ζεύγος επεξεργαστή-μνήμης μέσα σε ένα ενιαίο σύστημα. Η μεταβίβαση των μηνυμάτων γίνεται μέσω του δικτύου, το οποίο μπορεί είτε να είναι κάποιο συνηθισμένο δίκτυο γενικού σκοπού όπως ένα δίκτυο Ethernet, είτε κάποιο ειδικού σκοπού δίκτυο όπως τα σύγχρονα δίκτυα υψηλών επιδόσεων Myrinet και Infiniband.

Στον πραγματικό κόσμο των συστημάτων κατανεμημένης μνήμης, κυριαρχεί ένας

συνδυασμός αυτής της αρχιτεκτονικής και της προαναφερθείσας αρχιτεκτονικής κοινόχρηστης μνήμης. Κάθε κόμβος είναι ένα ανεξάρτητο σύστημα κοινόχρηστης μνήμης, όπου κάθε επεξεργαστής είναι μία πολυπύρηνη επεξεργαστική μονάδα, η οποία επικοινωνεί με τη δική της ανεξάρτητη μνήμη, με οποιαδήποτε από τις συνδεσμολογίες που αναφέρθηκαν παραπάνω. Τέτοια συστήματα πολλές φορές έχουν παραπάνω από μία τέτοια επεξεργαστική μονάδα σε κάθε κόμβο, ενώ ως επεξεργαστικές μονάδες μπορεί να χρησιμοποιούνται και κάρτες γραφικών γενικού σκοπού (GPGPUs).

Παραδείγματα τέτοιων παράλληλων συστημάτων είναι οι γνωστές συστάδες (clusters), όπου κάθε κόμβος είναι ένας συμβατικός ή μη συμβατικός πολυπύρηνος Η/Υ και οι κόμβοι είναι συνδεδεμένοι όπως αναφέρθηκε παραπάνω μέσω ενός δικτύου. Μεγάλο πλεονέκτημα αυτών των συστημάτων είναι το χαμηλό κόστος τους, καθώς οι κόμβοι μπορεί να αποτελούνται από υλικό γενικού σκοπού και το δίκτυο που τους διασυνδέει μπορεί επίσης να είναι υλικό χαμηλού κόστους.

Η αρχιτεκτονική αυτή συναντάται και σε σημερινά υπερυπολογιστικά συστήματα, με τη σημαντική διαφορά ότι τώρα πια πρόκειται για συστήματα των οποίων το υλικό αποτελείται από ειδικά σχεδιασμένο, ειδικού σκοπού υλικό, με χιλιάδες κόμβους, καθένας από τους οποίους περιέχει δεκάδες πυρήνες και δεκάδες κάρτες γραφικών, για ένα σύνολο συχνά δεκάδων χιλιάδων επεξεργαστικών μονάδων.

1.3 Παράλληλος Προγραμματισμός ανά αρχιτεκτονική

1.3.1 Συστήματα κοινόχρηστης μνήμης (shared memory)

Τα πιο διαδεδομένα μοντέλα προγραμματισμού κοινόχρηστης μνήμης σήμερα είναι το πρότυπο OpenMP [2] και το μοντέλο εκτέλεσης POSIX Threads (pthreads). Στο πρότυπο OpenMP ο προγραμματιστής δίνει κάποιες οδηγίες στον μεταγλωττιστή υπό τη μορφή pragmas στον πηγαίο κώδικα τα οποία στη συνέχεια χρησιμοποιούνται από τον μεταγλωττιστή για την παραγωγή ενός παράλληλου προγράμματος. Στο μοντέλο εκτέλεσης των POSIX Threads δίνεται στον προγραμματιστή η δυνατότητα να ελέγχει διαφορετικές οντότητες εκτέλεσης με τη χρήση ενός Application Programming Interface (API), υπό τη μορφή κλήσεων συστήματος.

Τα κύρια μειονεκτήματα των pthreads αφορούν τη δυσκολία για τον προγραμμα-

τιστή να συντονίσει τα νήματα, να διαχειριστεί θέματα που αφορούν την ταυτόχρονη προσπέλαση κοινόχρηστης μνήμης, καθώς και την ίδια τη διαχείριση των νημάτων (δημιουργία, καταστροφή). Από την άλλη πλευρά, το πρότυπο OpenMP όχι μόνο δίνει στον προγραμματιστή τη δυνατότητα να παραλληλοποιήσει ένα ήδη υπάρχων σειριακό πρόγραμμα, δίχως να το τροποποιήσει, αλλά επίσης τα θέματα που αναφέρθηκαν πιο πάνω απλοποιούνται σημαντικά, ή δεν πέφτουν πια στην ευθύνη του προγραμματιστή.

1.3.2 Συστήματα κατανεμημένης μνήμης (distributed memory)

Στον προγραμματισμό των μοντέλων κατανεμημένης μνήμης, το κυρίαρχο πρότυπο για την ανταλλαγή μηνυμάτων είναι το MPI (Message Passing Interface) [3]. Στο μοντέλο αυτό, ο προγραμματιστής επιλέγει πόσες διεργασίες θέλει να δημιουργήσει σε πόσους κόμβους. Το πρότυπο δημιουργεί N διεργασίες – αντίγραφα του κύριου προγράμματος και τα αναθέτει στους διαθέσιμους κόμβους σύμφωνα με κάποια χαρτογράφηση που επιλέχθηκε από τον προγραμματιστή.

Ο προγραμματιστής είναι ο υπεύθυνος της διαχείρισης κάθε κόμβου ξεχωριστά, καθώς και κάθε μεταβίβασης μηνυμάτων "με το χέρι". Είναι επίσης ευθύνη του να γνωρίζει τι τύπους δεδομένων θέλει να μεταφέρει, το πλήθος τους, καθώς και το αν θα πραγματοποιήσει σύγχρονη ή ασύγχρονη επικοινωνία. Στην πρώτη περίπτωση, για την επικοινωνία με άλλες διεργασίες, η αποστέλουσα διεργασία "μπλοκάρει" μέχρις ότου λάβει απάντηση από τη διεργασία-παραλήπτη ότι έλαβε επιτυχώς τα δεδομένα. Αντίθετα, στην ασύγχρονη αποστολή, η διεργασία στέλνει τα δεδομένα, και συνεχίζει την εκτέλεσή της, δίχως να περιμένει για κάποιο είδος απάντησης από τη λαμβάνουσα διεργασία.

Πέραν αυτών των δυσκολιών, η ειδοποιός διαφορά με τον προγραμματισμό κοινόχρηστης μνήμης είναι η εξής. Ενώ στο μοντέλο κοινόχρηστης μνήμης όλα τα δεδομένα είναι προσβάσιμα από κάθε επεξεργαστή, στην περίπτωση αυτήν η τοποθέτηση και διαχείριση αυτών είναι καθαρά ευθύνη του προγραμματιστή. Η διαδικασία αυτή δεν είναι ούτε εύκολη, ούτε συγκεκριμένη. Αντίθετα, εξαρτάται από την εφαρμογή, ενώ αυτός ο διαμοιρασμός πρέπει να γίνει σοφά από τον προγραμματιστή.

Στο προγραμματιστικό μοντέλο αυτό, δεν γίνεται εκμετάλλευση των πολλαπλών πυρήνων ή/και επεξεργαστικών μονάδων κάθε κόμβου. Όπως αναφέρθηκε παραπάνω, δημιουργούνται αντίγραφα της κύριας διεργασίας τα οποία μοιράζο-

νται στους διαθέσιμους κόμβους. Όμως, από σχεδιασμού, εάν η λογική του προγράμματος ήταν σειριακή, θα παραμείνει σειριακή σε επίπεδο κόμβου. Μία απλή λύση για αυτό το πρόβλημα ώστε να αξιοποιηθούν όλοι οι φυσικοί πυρήνες από το κατανεμημένο πρόγραμμα αυτό είναι η ανάθεση μίας διεργασίας ανά επεξεργαστική μονάδα κόμβου, αφήνοντας την ευθύνη στο λειτουργικό σύστημα του κόμβου να αναθέσει τις διεργασίες σε κάποιον φυσικό πυρήνα, έτσι ώστε να γίνει έμμεση παραλληλοποίηση αυτών, καθώς ο κάθε πυρήνας θα εκτελεί μία ξεχωριστή διεργασία. Ο προγραμματισμός όπου εκμεταλλευόμαστε πολλαπλούς κόμβους με το μοντέλο κατανεμημένης μνήμης, καθώς και πολλαπλούς πυρήνες σε κάθε κόμβο, με το μοντέλο κοινόχρηστης μνήμης σε καθέναν από αυτούς, ονομάζεται υβριδικός προγραμματισμός.

1.3.3 Μοντέλο συντονιστή-εργάτη

Υπάρχουν δύο γενικότερα μοντέλα προγραμματισμού που ακολουθούνται κατά τον προγραμματισμό τέτοιων συστημάτων. Το πρώτο είναι αυτό του συντονιστή-εργατών, ενώ το άλλο είναι αυτού με εργασίες. Στο πρώτο, υπάρχει μία εργασία συντονιστής, η οποία συνήθως υλοποιεί την αρχική (σειριακή) λογική του προγράμματος, ενώ οι εργάτες εκτελούν κάποια τμήματα του προγράμματος κατανεμημένα, όπως αυτά ανατίθενται σε αυτούς από τον συντονιστή.

Το μοντέλο αυτό όμως στη γενικότερή του μορφή, π.χ. όταν χρησιμοποιηθούν μονάχα οι δυνατότητες που παρέχει το πρότυπο MPI, υποφέρει από ορισμένα προβλήματα. όπως τα εξής.

- Οι περισσότερες υλοποιήσεις που εκμεταλλεύονται αυτή την ιδέα με τη χρήση κάποιου προτύπου μεταβίβασης μηνυμάτων ακολουθούν μία κεντρικοποιημένη υλοποίηση, όπου ο συντονιστής είναι ο μόνος κόμβος που δημιουργεί και αναθέτει εργασίες, καθιστώντας τον σημαντικό κώλυμα στην επίτευξη επιδόσεων.

- Η έκφραση εμφωλευμένου παραλληλισμού σε αυτό το μοντέλο είναι όχι μόνο περιορισμένη, αλλά επίσης σημαντικά δυσκολότερη να υλοποιηθεί, από κάθε σκοπιά της ανάπτυξης λογισμικού και του παράλληλου προγραμματισμού. Για τους ίδιους λόγους, η υλοποίηση πολλαπλών συντονιστών αποτελεί μία εξίσου δύσκολη ιδέα. Σε κάποιες περιπτώσεις μάλιστα, ακόμα και εάν βρεθούν λύσεις σε αυτά τα προβλήματα, αυτά ενδέχεται να μην αποτελούν αποδεκτές

λύσεις, καθώς υπάρχουν κρυφοί κίνδυνοι όπως τον συγχρονισμό αυτών, οι οποίοι μπορούν να κρύβουν θέματα όπως deadlocks ή την έλλειψη της δυνατότητας κλιμάκωσης και προσαρμοστικότητας (ανά συστάδα) των εφαρμογών αυτών, τα οποία είναι δύσκολο έως αδύνατο να αντιμετοπισθούν.

- Όπως αναφέρθηκε και την περιγραφή του μοντέλου προγραμματισμού με μεταφορά μηνυμάτων, το μοντέλο αυτό βασίζεται σε διεργασίες έναντι άλλων οντοτήτων εκτέλεσης όπως νήματα, με συνέπεια η εκμετάλλευση πολλαπλών πυρήνων (επεξεργαστικών μονάδων γενικότερα) να απαιτεί πιο πολύπλοκη λογική, αλλά και να επιφέρει επιπλέον υπολογιστικά κόστη.

1.3.4 Μοντέλο προγραμματισμού με tasks (εργασίες)

Ο προγραμματισμός του μοντέλου με εργασίες αφορά την τεχνική σχεδίασης λογισμικού όπου στο πρόγραμμα δημιουργείται ένα πλήθος συνήθως (ιδανικά) ανεξάρτητων μεταξύ τους λογικών εργασιών (tasks), οι οποίες στη συνέχεια εκτελούνται από κάποια κατάλληλη οντότητα εκτέλεσης.

Μία οντότητα εκτέλεσης αυτού του μοντέλου μπορεί να είναι ένα νήμα POSIX, ή μία (ενδεχομένως απομακρυσμένη) συσκευή. Εάν αυτές οι οντότητες τρέχουν σε διαφορετικές επεξεργαστικές μονάδες, ταυτόχρονα, τότε επιτυγχάνεται παραλληλισμός. Υπό το μοντέλο αυτό μπορούν να υλοποιηθεί με ευκολία αυτό του συντονιστήεργατών, όπου ως συντονιστής ορίζεται μία οντότητα εκτέλεσης η οποία δημιουργεί εργασίες, ενώ ως εργάτης οποιαδήποτε τέτοια οντότητα. Έναντι των παραδοσιακών υλοποιήσεων, τώρα πια οι οντότητες εκτέλεσης μπορούν να ενεργούν ως συντονιστές και ως εργάτες ταυτόχρονα, ενώ μπορούν να υπάρχουν πολλαπλοί συντονιστές, ταυτόχρονα, σε κάθε εκτέλεση.

Σε πολλές περιπτώσεις εφόσον η χρήση αυτής της τεχνικής έναντι κάποιας παραδοσιακής όπως του απλού πολυνηματισμού, μπορεί να αποδειχθεί επιφέρει κέρδος στις επιδόσεις [4]. Μία τέτοια υλοποίηση, όπου χρησιμοποιούνται νήματα επιπέδου χρήστη (user-level threads), ως οντότητες εκτέλεσης, χρησιμοποιεί η βιβλιοθήκη TORC.

Παρόλο η ιδέα του προγραμματισμού (και παραλληλισμού) με εργασίες μπορεί να φανεί ανήκουστη ακόμα και σε κάποιον επιστήμονα της Πληροφορικής, δεν είναι κάτι τελείως ξένο. Ακόμα και η δρομολόγηση διεργασιών επιπέδου χρήστη από το λειτουργικό σύστημα μπορεί να χρησιμοποιεί αυτή την τεχνική, όπου κάθε διεργασία

του χρήστη είναι ένα task, τα οποία διαχειρίζονται και δρομολογούνται σε κάποιο χαμηλότερο επίπεδο του λειτουργικού συστήματος, στο επίπεδο του δρομολογητή.

1.4 Αντικείμενο Διπλωματικής Εργασίας

Το πρώτο τμήμα της εργασίας αφορά τη μελέτη του HOMPI framework. Κατά τη μελέτη αυτήν εξετάζονται τα πλεονεκτήματα και μειονεκτήματα της χρήσης του framework αυτού, καθώς και του μοντέλου προγραμματισμού με χρήση εργασιών (tasks), από πλευράς επιδόσεων αλλά και προγραμματιστικής ευκολίας. Τα θέματα αυτά καθώς και της πρακτικής της μετατροπής σειριακών (ή πολυνηματικών) εφαρμογών σε αντίστοιχες κατανεμημένες εκδόσεις τους θα αποτελέσουν σημαντικό κομμάτι της παρούσας εργασίας. Επίσης, προστέθηκαν νέες λειτουργίες στο εν λόγω framework, ώστε να γίνει μία αξιολόγηση της οργάνωσης του συστήματος αυτού από τη σκοπιά της τεχνολογίας λογισμικού. Οι λειτουργίες αυτές αφορούν τη βελτίωση της διεπαφής του χρήστη με το framework, καθώς και την προσθήκη ενός νέου δρομολογητή εργασιών για χρήση σε ανομοιογενή συστήματα.

Το δεύτερο κομμάτι της διπλωματικής αυτής εργασίας πραγματεύεται θέματα που αφορούν τον προγραμματισμό συστάδων επεξεργαστικών συστημάτων με τη χρήση του HOMPI Framework, κυρίως με τη χρήση tasks και υβριδικού προγραμματισμού. Ταυτόχρονα, μελετά διάφορα θέματα της παράλληλης επεξεργασίας γενικότερα, όπως τις περιπτώσεις στις οποίες αξίζει να καταφύγουμε σε τέτοιες μεθόδους και όλα τα θέματα που προκύπτουν κατά τον προγραμματισμό τέτοιων συστημάτων για έμπειρους ή μη προγραμματιστές και μηχανικούς τέτοιων συστημάτων υψηλών επιδόσεων. Η μελέτη αυτή έγινε εκτελώντας σε πολλαπλά διαφορετικά μεταξύ τους σε αρχιτεκτονική συστήματα, μία πληθώρα ανεπτυγμένων από εμάς εφαρμογών και διαμορφώσεων στα σχετικά συστήματα. Ένα μέρος αυτών των εφαρμογών γράφθηκαν από την αρχή στοχεύοντας στο HOMPI framework, ενώ οι άλλες πρόκειται για τροποποιημένες εκδόσεις υπάρχων εφαρμογών με σκοπό τη συμβατότητα με το framework αυτό.

1.5 Δομή Διπλωματικής Εργασίας

Παρακάτω δίνεται η δομή των κεφαλαίων της παρούσας διπλωματικής εργασίας:

- Κεφάλαιο 2: Επισκόπηση του HOMPI framework.

  Παρουσίαση της δομής και των λειτουργιών του framework που χρησιμοποιήθηκε στην εργασία μας.

- Κεφάλαιο 3: Επεκτάσεις του προγραμματιστικού μοντέλου

  Παρουσίαση των επεκτάσεων και νέων χαρακτηριστικών που υλοποιήθηκαν στο framework, σύμφωνα με τις ανάγκες και ιδέες που προέκυψαν κατά τη χρήση του.

- Κεφάλαιο 4: Εφαρμογές

  Περιγραφή των εφαρμογών που αναπτύχθηκαν για τις ανάγκες της εργασίας μας, τόσο για συστήματα κοινόχρηστης μνήμης, όσο και κατανεμημένης.

- Κεφάλαιο 5: Πειραματικές εκτελέσεις και συμπεράσματα

  Παρουσίαση και ανάλυση των αποτελεσμάτων εκτέλεσης των παραπάνω εφαρμογών σε διάφορα συστήματα.

- Κεφάλαιο 6: Αξιολόγηση του HOMPI framework

  Σύντομη αξιολόγηση του framework σύμφωνα με τη χρήση του για παράλληλο παραλληλισμό βασισμένο σε tasks, καθώς και της εσωτερικής δομής του.

- Κεφάλαιο 7: Προγραμματισμός με tasks για Παράλληλη και Κατανεμημένη επεξεργασία

  Μία ανάλυση του παράλληλου προγραμματισμού με tasks, ως προς τα σενάρια εφαρμογής του και των δυσκολιών που συναντώνται κατά αυτόν.

- Κεφάλαιο 8: Επίλογος

Κεφαλαιο 2

Επισκοπηση του HOMPI Framework

2.1 Το προγραμματιστικό μοντέλο HOMPI

2.1 Το προγραμματιστικό μοντέλο HOMPI

Το HOMPI framework [5] είναι μία προγραμματιστική πλατφόρμα που στοχεύει στην παραλληλοποίηση εφαρμογών με χρήση των τεχνικών υβριδικού, βασισμένου σε εργασίες παράλληλου προγραμματισμού. Ο στόχος του επιτυγχάνεται δίνοντας στον χρήστη τη δυνατότητα να εκφράσει ανεξάρτητες μεταξύ τους εργασίες, προσφέροντας πλήρη διαφάνεια σχετικά με την κατανομή των εργασιών αυτών, σε μία πληθώρα συστημάτων.

Ένα επιπλέον σημαντικό πλεονέκτημα του framework αυτού είναι η δυνατότητα χρήσης υπαρχόντων μοντέλων προγραμματισμού σε συνδυασμό με τις δυνατότητες που παρέχονται. Μέσα στο πρόγραμμά του, ο χρήστης μπορεί να χρησιμοποιήσει το πρότυπο OpenMP και MPI. Με αυτό τον τρόπο, δίνεται επιπλέον ελευθερία στην έκφραση παραλληλισμού, ενώ ταυτόχρονα υπάρχει πλήρης δυνατότητα της χρήσης του framework για την εκμετάλλευση ή/και εμπλούτιση υπάρχοντος κώδικα OpenMP/MPI.

Τα μειονεκτήματα του μοντέλου προγραμματισμού με tasks καθώς και τα πλεονεκτήματα της παραπάνω δυνατότητας θα αναλυθούν σε επόμενα κεφάλαια της εργασίας.

2.1.1 Επισκόπηση των λειτουργιών και της δομής του HOMPI

Το προγραμματιστικό μοντέλο έχει ως σκοπό την εκμετάλλευση πολλαπλών κόμβων πολυπύρηνων υπολογιστικών συστημάτων. Το μοντέλο προγραμματισμού που ακολουθείται είναι αυτό που περιγράψαμε παραπάνω, δηλαδή η εκτέλεση ανεξάρτητων εργασιών, όμοια με την εκτέλεση απομακρυσμένων κλήσεων συναρτήσεων (Remote Procedure Call). Οι εργασίες αυτές έχουν σχέση γονέα-παιδιού μεταξύ τους και μπορούν να εμφωλευθούν διαφανώς, παρέχοντας έτσι τα πλεονεκτήματα του μοντέλου παραλληλισμού με εργασίες. Ένα απλό παράδειγμα ενός προγράμματος που χρησιμοποιεί το framework δίνεται αμέσως παρακάτω.

```
1  #pragma ompix taskdef in(len) inout(A[len])
2  void quicksort(int *A, int len) {
3      int pivot, i, j;
4
5      if (len < 2) return;
6
7      pivot = A[len >> 1];
8
9      for (i = 0, j = len - 1; ; i++, j--){
10
11         while (A[i] < pivot) i++;
12         while (A[j] > pivot) j--;
13
14         if (i >= j) break;
15
16         int temp = A[i];
17         A[i]     = A[j];
18         A[j]     = temp;
19     }
20
21     #pragma ompix task if (len >= LOCAL_CUTOFF)
22     quicksort(A, i);
23
24     #pragma ompix task if (len >= LOCAL_CUTOFF)
25     quicksort(A + i, len - i);
26
27     #pragma ompix tasksync
28 }
```

Στο παράδειγμα αυτό βλέπουμε μία κατανεμημένη εκδοχή του γνωστού αλγορίθμου quicksort. Όπως θα αναλυθεί περαιτέρω, η αξιοποίηση των λειτουργιών του framework γίνεται με την προσθήκη επιπλέον οδηγιών (βλέπε κώδικα pragma ompix) προς τον μεταγλωττιστή. Στην γραμμή 1 βλέπουμε την δήλωση μίας συνάρτησης για εκτέλεση ως task, μαζί με την δήλωση των παραμέτρων της, στις γραμμές 21 και 24 βλέπουμε τη δημιουργία τέτοιων task, ενώ στην γραμμή 27 την χρήση ενός μηχανισμού συγχρονισμού. Αξίζει να σημειώσουμε ένα ιδιαίτερα χρήσιμο χαρακτηριστικό του framework. Ο αρχικός κώδικας που βλέπουμε δεν έχει τροποποιηθεί, έχει παρά μόνο εμπλουτιστεί με γραμμές οι οποίες θα αγνοηθούν από άλλους μεταγλωττιστής πέραν αυτού του HOMPI. Ως αποτέλεσμα, ο κώδικας αυτός μπορεί να μεταγλωττιστεί και να εκτελεσθεί ως απλό πρόγραμμα της γλώσσας C. Επιπλέον, σε αρκετές εφαρμογές, οι εργασίες μπορούν να εκτελεσθούν είτε στον κόμβο που τις

δημιούργησε, είτε σε κάποιον απομακρυσμένο κόμβο, δίχως κάποια τροποποίηση στο πρόγραμμα. Θα σημειώσουμε εν συντομία ότι η ορθή χρήση αυτής της λειτουργίας όπως και όλων των άλλων του framework είναι ευθύνη του προγραμματιστή, τόσο για την επίτευξη επιδόσεων, όσο και για την ορθότητα του προγράμματος.

Το προγραμματιστικό μοντέλο αποτελείται από το τμήμα του μεταγλωττιστή από πηγαίο σε πηγαίο κώδικα της γλώσσας C και τη βιβλιοθήκη χρόνου εκτέλεσης, οι οποίες θα περιγραφθούν πιο αναλυτικά παρακάτω. Σημαντικό πλεονέκτημα του συνόλου εργαλείων είναι η εκμετάλλευση παραλληλισμού κοινόχρηστης μνήμης σε κάθε κόμβο, το οποίο περιγράψαμε ως μειονέκτημα σε υλοποιήσεις που χρησιμοποιούν μονάχα το μοντέλο MPI. Ο HOMPI όχι μόνο υλοποιεί χάρη στον σχεδιασμό του παραλληλισμό σε τοπικό επίπεδο με πλήρη διαφάνεια χάρην στην υλοποίηση των νημάτων εργατών που θα περιγραφούν παρακάτω, αλλά υποστηρίζει πλήρως εκτέλεση κώδικα με οδηγίες του προτύπου OpenMP μέσα στις εργασίες που μπορεί να δημιουργήσει ο χρήστης. Ο χρήστης μπορεί επίσης να εκτελέσει reductions με τη χρήση tasks, όμοια με αυτά του προτύπου OpenMP.

Μία άλλη χρήσιμη λειτουργία είναι αυτή των callbacks. Ένα callback ορίζεται ως μία συνάρτηση που ορίζεται αμέσως μετά από τον κώδικα ενός task. Αυτή η συνάρτηση χρησιμοποιεί τις ίδιες παραμέτρους με το task με το οποίο συνδέεται και εκτελείται κατά το πέρας της εκτέλεσης του task αυτού, στον κόμβο που το δημιούργησε. Τα callbacks είναι ιδιαίτερα χρήσιμα στην περίπτωση όπου κάποια εργασία (ενδεχομένως απομακρυσμένη) υπολογίζει κάποιο μερικό αποτέλεσμα το οποίο πρέπει να επεξεργασθεί ο δημιουργός της εργασίας τοπικά. Ένα σενάριο εφαρμογής αυτής της λειτουργίας θα δείξουμε στην εφαρμογή του πολλαπλασιασμού πινάκων. Θα σημειώσουμε επίσης ότι στην τυπική ορολογία τα callbacks έχουν αιτιακή σχέση με τα task που συνδέονται, παρέχοντας τη δυνατότητα υλοποίησης της έννοιας αυτής, κάτι το οποίο δεν παρέχεται από τη βιβλιοθήκη με άλλο τρόπο.

Τα νήματα-εργάτες έχουν επίσης τη δυνατότητα να κλέψουν κάποιο task. Για τη ρύθμιση της λειτουργίας αυτής παρέχεται μία διεπαφή. Οι οδηγίες "tied" και "untied" παρέχουν τη δυνατότητα δημιουργίας tasks τα οποία στην πρώτη περίπτωση δεν επιτρέπεται να κλαπούν, ενώ οι εντολές `torc_{disable, enable}_stealing()` ενεργοποιούν και απενεργοποιούν αυτόν τον μηχανισμό, αντίστοιχα.

Ο χρήστης μπορεί επίσης ρητά να δηλώσει σε ποιόν κόμβο επιθυμεί να εκτελεσθεί η εργασία, με χρήση της φράσης "atnode" της οδηγίας "task", καθώς και ένα συγκεκριμένο νήμα-εργάτη, με την οδηγία "atworker".

Κατά την εκτέλεση μίας εφαρμογής, δημιουργείται ένα αντίγραφο της διεργασίας σε κάθε κόμβο του δικτύου, ακολουθώντας δηλαδή αρχικά το μοντέλο του συντονιστή-εργάτη, όπου ο συντονιστής μπορεί επίσης να εκτελέσει εργασίες. Αυτό το μοντέλο μπορεί να τροποποιηθεί με ευέλικτο τρόπο, καθώς κάθε εργάτης μπορεί στη συνέχεια να αναθέσει εργασίες στον εαυτό του ή σε άλλους κόμβους, ή χρησιμοποιώντας το μοντέλο εκτέλεσης SPMD που παρέχεται από το μοντέλο προγραμματισμού.

Το περιβάλλον προγραμματισμού

Το περιβάλλον προγραμματισμού ακολουθεί τη μορφή του γνωστού προτύπου OpenMP, την ενίσχυση δηλαδή του υπάρχοντος προγράμματος με οδηγίες προς τον μεταγλωττιστή (pragmas).

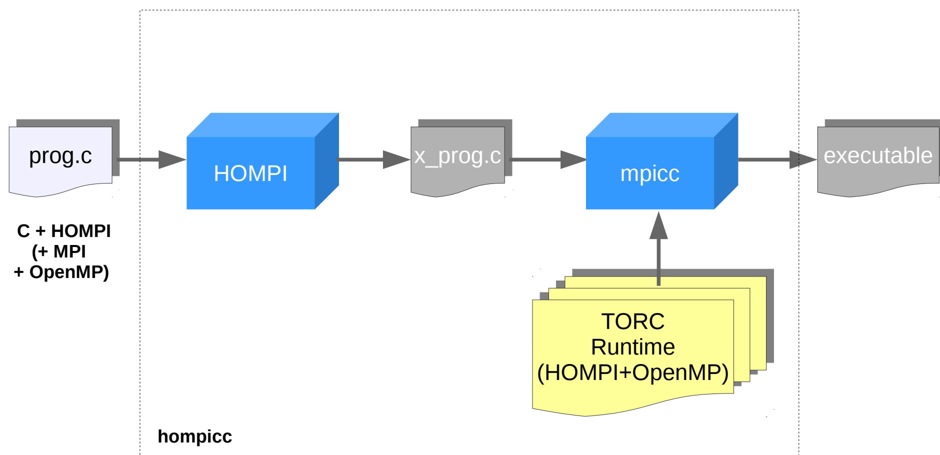
Ο χρήστης είναι υπεύθυνος να δηλώσει ποιες συναρτήσεις επιθυμεί να είναι δυνατό να εκτελεσθούν ως tasks καθώς και να ορίσει τη μέθοδο περάσματος της κάθε παραμέτρου τους. Αυτό γίνεται με το pragma ''taskdef'' και το pragma in/inout, πριν τον ορισμό μίας συνάρτησης της γλώσσας.

Υπάρχουν τρεις πιθανοί τρόποι περάσματος παραμέτρων. In, inout και out. Στον πρώτο, η τιμή της παραμέτρου αποστέλλεται στον κόμβο όπου θα εκτελέσει την εργασία, αλλά παραμένει ως τοπική μεταβλητή, δηλαδή η τιμή της στο πέρας της εργασίας δεν επιστρέφεται στον δημιουργό της. Στον δεύτερο τρόπο, περνάτε η τιμή της παραμέτρου σαν είσοδος και κατά το τέλος της εργασίες επιστρέφεται πίσω στον κόμβο που δημιούργησε την εργασία, ως αποτέλεσμα. Στον τελευταίο τρόπο περάσματος, δεν περνάτε η τιμή της μεταβλητής ως είσοδο, αλλά επιστρέφεται το αποτέλεσμα κατά το πέρας της εκτέλεσης της εργασίας.

Για να εκτελεσθεί μία συνάρτηση ως εργασία, πρέπει να εισαχθεί ένα pragma τύπου task, πριν την κλήση μίας συνάρτησης. Ο μεταγλωττιστής αναλαμβάνει πλήρως να περάσει τις απαραίτητες πληροφορίες για τις μεταβλητές στη βιβλιοθήκη εκτέλεσης, μέσω της διεπαφής.

2.1.2 Ο παραλληλοποιητικός μεταφραστής OMPi

Ο OMPi είναι ένας μεταφραστής από πηγαίο κώδικα σε πηγαίο κώδικα, ο οποίος παρέχει πλήρη υποστήριξη του προτύπου OpenMP. Πιο συγκεκριμένα, ως είσοδος του μεταγλωττιστή είναι ένα πρόγραμμα στη γλώσσα προγραμματισμού C, το



Σχήμα 2.1: Η διαδικασία μεταγλώττισης του HOMPI.

οποίο μπορεί να περιέχει επιπλέον οδηγίες του προτύπου OpenMP, ή επίσης όπως στην περίπτωση του HOMPI, άλλες επιπλέον οδηγίες προς τον μεταγλωττιστή, για την χρήση σε λειτουργίες του HOMPI. Και στις δύο αυτές περιπτώσεις, οι οδηγίες αυτές εκφράζονται υπό την μορφή pragmas, τα οποία δίνουν την επιπλέον πληροφορία στον OMPi για τη χρήση αυτών των επιπρόσθετων λειτουργιών, δίχως να επηρεάζουν τον τρόπο λειτουργίας του αρχικού προγράμματος. Καθώς ο OMPi (και κατ'επέκταση, ο HOMPI) είναι μεταφραστές από πηγαίο σε πηγαίο κώδικα της γλώσσας C, απαιτείται η ύπαρξη ενός system compiler της γλώσσας.

Το λογισμικό του μεταφραστή είναι οργανωμένο σε δύο τμήματα, τα οποία έχουν ελάχιστη συσχέτιση μεταξύ τους. Παρόλο που όλα τα πλεονεκτήματα αυτής της οργάνωσης δεν μπορούν να περιγραφθούν σε αυτή την εργασία καθώς δεν αφορούν το κύριο αντικείμενό της, πολλά σημαντικά από αυτά θα φανούν στη συνέχεια αυτής της ενότητας.

Ο μεταγλωττιστής

Το πρώτο τμήμα του OMPi είναι το τμήμα του μεταφραστή. Το κομμάτι αυτό είναι υπεύθυνο για τη λεκτική και συντακτική ανάλυση του προγράμματος εισόδου, καθώς και τη μετατροπή του στο ενδιάμεσο πρόγραμμα. Ως είσοδος, όπως αναφέρθηκε παραπάνω, ο μεταγλωττιστής δέχεται ένα πρόγραμμα στη γλώσσα προγραμματισμού C, αλλά καθώς πραγματοποιείται πλήρης λεκτική και συντακτική ανάλυση, υπάρχει η δυνατότητα επέκτασης της αρχικής γλώσσας, έτσι ώστε να μπορούν να υποστηριχθούν ιδέες όπως αυτή του HOMPI, δίχως να επηρεάζονται διόλου οι αρχικοί σκοποί του μεταγλωττιστή. Καθώς το αρχικό πρόγραμμα πρέπει να υποστεί μετασχηματισμό, ο συντακτικός αναλυτής παράγει ένα πλήρες αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree), το οποίο χρησιμοποιείται στη συνέχεια για την παραγωγή του τροποποιημένου κώδικα. Ο τροποποιημένος αυτός κώδικας δεν περιέχει πια τα pragmas που αφορούν το πρότυπο OpenMP, ή τον HOMPI. Σύμφωνα όμως με το περιεχόμενό τους, έχουν προστεθεί στο αρχικό πρόγραμμα κλήσεις συναρτήσεων του περιβάλλοντος εκτέλεσης, το οποίο θα περιγραφθεί επόμενο.

Στην περίπτωση του HOMPI, το τροποποιημένο αυτό πρόγραμμα μεταγλωττίζεται από το εργαλείο mpicc της υλοποίησης του MPI, το οποίο διασυνδέει τη βιβλιοθήκη χρόνου εκτέλεσης και στο τέλος παράγει το τελικό, εκτελέσιμο αρχείο (βλέπε σχήμα 2.1). Το εργαλείο mpicc είναι ένας wrapper ο οποίος προσθέτει αυτομάτως τις επιπλέον οδηγίες στον system compiler κατά την μεταγλώττιση ώστε αυτός να μπορεί να κάνει χρήστης της εγκατεστημένης υλοποίησης του MPI.

Το περιβάλλον εκτέλεσης

Το δεύτερο τμήμα του OMPi είναι οι βιβλιοθήκες περιβάλλοντος εκτέλεσης. Οι βιβλιοθήκες αυτές περιέχουν κλήσεις οι οποίες στη γενική περίπτωση του OMPi εκτελούν τις λειτουργίες του προτύπου OpenMP, αλλά στην περίπτωση του HOMPI εκτελούν τις λειτουργίες του framework. Χάρη σε αυτό τον σχεδιασμό, οποιοσδήποτε προγραμματιστής του OMPi είναι ελεύθερος να υλοποιήσει τις συναρτήσεις αυτές με όποιον τρόπο επιθυμεί, παρέχοντας ευελιξία στον σχεδιασμό και την υλοποίηση. Μία από αυτές τις υλοποιήσεις, είναι η βιβλιοθήκη TORC, η οποία μπορεί να χρησιμοποιηθεί για τη δημιουργία και εκτέλεση των λεγόμενων εργασιών (tasks).

2.1.3 Η βιβλιοθήκη TORC

Η βιβλιοθήκη παραλληλισμού βασισμένου σε εργασίες TORC [6], είναι μία βιβλιοθήκη σχεδιασμένη για υπολογισμούς υψηλών επιδόσεων σε συστήματα συστάδων πολυπύρηνων κόμβων. Ο στόχος της είναι η εκμετάλλευση πολλαπλών κόμβων σε επίπεδο συστάδας καθώς και των πολλαπλών SMP (Symmetrical Multi-Processors, Συμμετρικοί Πολυ-επεξεργαστές) κάθε κόμβου, σε επίπεδο κοινόχρηστης μνήμης.

Το μοντέλο προγραμματισμού της βιβλιοθήκης ακολουθεί το μοντέλο προγραμματισμού με εργασίες, όπου κάθε οντότητα εκτέλεσης (εργάτης) να συμμετάσχει στην εκτέλεση των tasks, καθώς και να δημιουργήσει νέες. Το μοντέλο αυτό έχει ευρεία εφαρμογή, ξεκινώντας από συστήματα κοινόχρηστης μνήμης με SMP, μέχρι ασυμετρικά συστήματα συστάδων, παραδείγματος χάριν συστήματα με κόμβους που εκμεταλλεύονται επιταχυντές, ή και γενικού σκοπού επεξεργαστικές μονάδες γραφικών.

Όπως συμπεραίνουμε, τα κύρια προβλήματα τόσο από την προγραμματιστική άποψη και από την άποψη των επιδόσεων στον προγραμματισμό των κατανεμημένων συστημάτων πηγάζουν από το περιορισμένο μοντέλο προγραμματισμού με μεταβίβαση μηνυμάτων στην αρχική του μορφή. Η διαφάνεια που προσφέρει η βιβλιοθήκη για την έκφραση και ανάθεση παράλληλων tasks έχει στόχο την ελάφρωση των δυσκολιών και προβλημάτων και των δύο αυτών πλευρών. Ταυτόχρονα, η βιβλιοθήκη TORC επεκτείνει το μοντέλο αυτό διατηρώντας υπ όψιν τις επιδόσεις και την επεκτασιμότητα.

Ωστόσο, αυτά τα πλεονεκτήματα δεν έρχονται δίχως κάποιο τίμημα. Το τίμημα αυτό είναι η σχετική δυσχρηστία της βιβλιοθήκης, εάν χρησιμοποιηθεί ως ανεξάρτητο προγραμματιστικό εργαλείο. Χρησιμοποιούμε τον όρο "σχετική", για δύο λόγους. Ο πρώτος είναι ότι η βιβλιοθήκη αυτή έχει σχεδιασθεί με κύριο στόχο τη χρήση σε clusters υψηλών επιδόσεων από χρήστες με εμπειρία στον προγραμματισμό τέτοιων συστημάτων. Ο άλλος λόγος είναι ότι ως αναφορά χρησιμοποιείται το επίπεδο ευχρηστίας της βιβλιοθήκης σε συνδυασμό με τον μεταγλωττιστή OMPi, ως μέρος του HOMPI. Καθώς η βιβλιοθήκη δεν διαθέτει κάποια υψηλότερου επιπέδου διεπαφή όπως pragmas, ο χρήστης πρέπει να χρησιμοποιήσει το API της βιβλιοθήκης, στο οποίο ο χρήστης πρέπει ακόμα να δηλώσει ρητά τους τύπους και το πλήθος κάθε παραμέτρου της task που επιθυμεί να δημιουργήσει.

Η υλοποίηση του framework επιτυγχάνει πλήρη διαφάνεια κατά τη χρήση της

βιβλιοθήκης. Υπό την ύπαρξη ενός προγράμματος του οποίου τα δεδομένα και η λογική είναι προσαρμοσμένα για εκτέλεση με το μοντέλο μεταβίβασης μηνυμάτων, οι μοναδικές προσθήκες που απαιτούνται για τη μεταφορά του εν λόγω προγράμματος στο framework είναι η προσθήκη των κατάλληλων pragmas, η οποία μάλιστα δεν θα επηρεάσει καθόλου τη λογική και την ορθότητα του προγράμματος.

Στη συνέχεια θα δώσουμε μία επισκόπηση των λεπτομερειών υλοποίησης της βιβλιοθήκης, περιγράφοντας δύο κύρια συστατικά του. Το πρώτο είναι το μοντέλο εκτέλεσης νημάτων δύο επιπέδων όπου θα περιγραφθεί επίσης η υλοποίηση των νημάτων αυτών, ενώ το δεύτερο αφορά τη δημιουργία και δρομολόγηση των tasks.

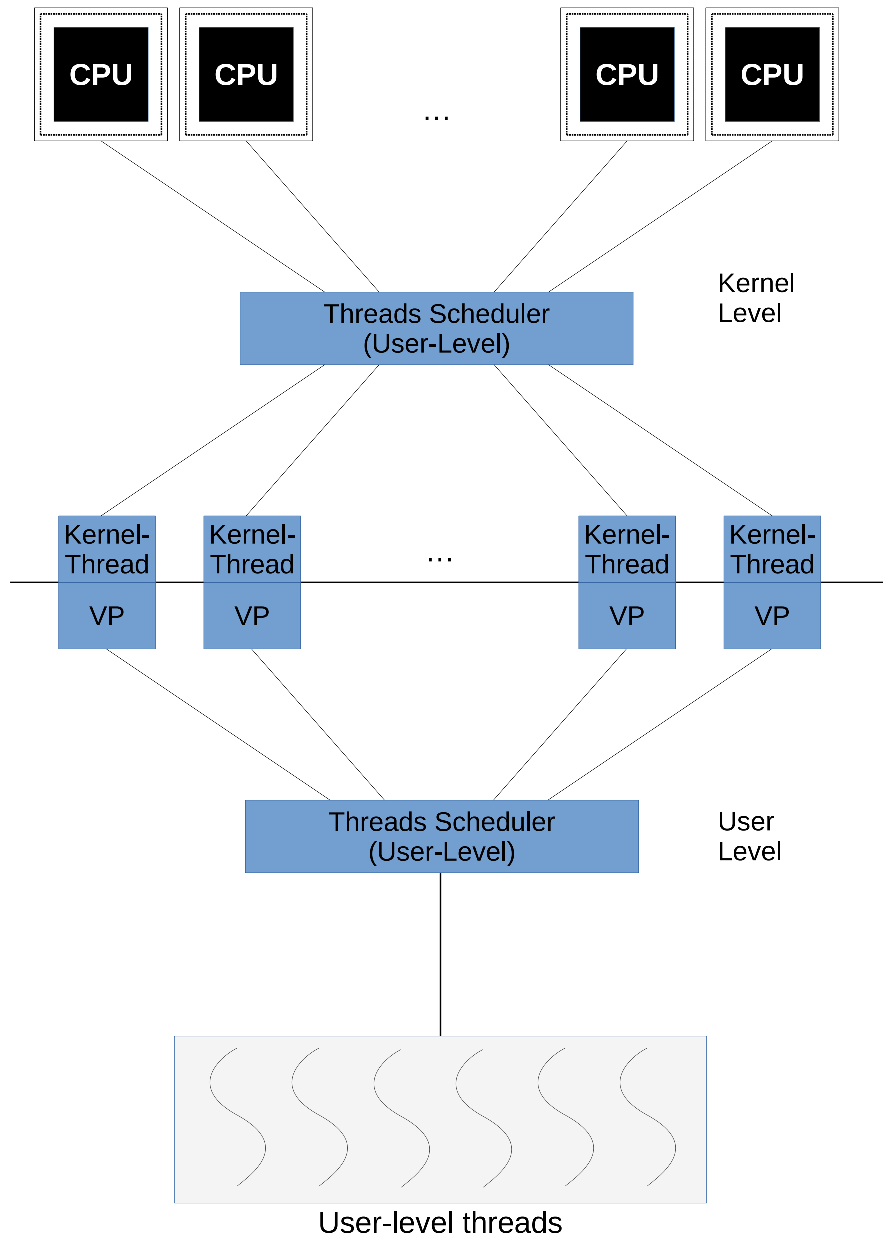
2.1.4 Το μοντέλο εκτέλεσης νημάτων δύο επιπέδων

Για την εκμετάλλευση των πολλαπλών επεξεργαστικών μονάδων σε επίπεδο κόμβου, η βιβλιοθήκη TORC χρησιμοποιεί ένα μοντέλο νημάτων δύο επιπέδων, το οποίο αποτελείται από τις εικονικές εργασίες και τα νήματα εργάτες, όπως παρουσιάζεται στο σχήμα 2.2.

Νήματα επιπέδου χρήστη

Τα νήματα επιπέδου χρήστη (user level threads) είναι οντότητες των οποίων η ύπαρξη δεν είναι γνωστή στο λειτουργικό σύστημα. Ένα νήμα επιπέδου χρήστη περιγράφει την τρέχουσα κατάσταση εκτέλεσης του προγράμματος και υλοποιούνται στο επίπεδο χρήστη, κατί που καθιστά την υλοποίηση τους αρκετά πιο εύκολη από αυτήν των νημάτων επιπέδου πυρήνα (kernel level threads). Η εναλλαγή του περιεχομένου νημάτων επιπέδου χρήστη είναι σε κάθε περίπτωση πιο γρήγορη από την εναλλαγή νημάτων επιπέδου πυρήνα, καθώς αυτή η διαδικασία απαιτεί μονάχα την κλήση απλών συναρτήσεων χρήστη, δίχως να γίνει εναλλαγή σε λειτουργία εκτέλεσης πυρήνα. Στην περίπτωσή της TORC, η περιγραφή της τρέχουσας κατάστασης εκτέλεσης μίας εικονικής εργασίας, συμπεριλαμβανομένων των δεδομένων της (παραμέτρους της συνάρτησης), γίνεται υπό τη μορφή ενός τέτοιου νήματος επιπέδου χρήστη, το οποίο αναφέρεται ως task descriptor. Οι οντότητες αυτές βρίσκονται στο user space, και ως συνέπεια διαχειρίζονται και δρομολογούνται αποκλειστικά από τη βιβλιοθήκη, δίχως να απαιτείται η κλήση κοστοβόρων κλήσεων συναρτήσεων συστήματος.

Φυσικά, ως αντίτιμο για τις ευκολίες αυτής της υλοποίησης, υπάρχουν και τα



Σχήμα 2.2: Το μοντέλο εκτέλεσης δύο επιπέδων νημάτων

αντίστοιχα μειονεκτήματά της. Δύο από αυτά τα μειονεκτήματα που πρέπει να ξεπεράσουμε ώστε να επιτύχουμε την παράλληλη εκτέλεση με αυτή την υλοποίηση, είναι τα εξής. Το λειτουργικό σύστημα δεν μπορεί να δρομολογήσει την εκτέλεση αυτών των νημάτων επιπέδου χρήστη σε διαφορετικές επεξεργαστικές μονάδες, άρα ως έχει, μιλάμε ακόμα για σειριακή εκτέλεση των εργασιών μας. Επιπλέον, αυτό συνεπάγεται ότι στην περίπτωση όπου η εκτέλεση ενός τέτοιου νήματος εκτελέσει μία εμποδιστική λειτουργία (blocking operation), η εκτέλεση όλου του προγράμμα-

τος θα μπλοκάρει. Υπενθυμίζουμε ότι τα νήματα επιπέδου χρήστη είναι οντότητες μη αναγνωρίσιμες από το λειτουργικό σύστημα.

Η λύση για τα παραπάνω προβλήματα δίνεται με το μοντέλο εκτέλεσης δύο επιπέδων νημάτων. Για την εκμετάλλευση των πολλαπλών πυρήνων, δημιουργούνται τα νήματα εργάτες. Οι οντότητες αυτές πρόκειται πια για οντότητες που χρησιμοποιούν βιβλιοθήκες στο επίπεδο του λειτουργικού συστήματος, υλοποιημένες με τη χρήση των Posix threads, οι οποίες δρομολογούνται από το λειτουργικό σύστημα σε διαφορετικούς φυσικούς πυρήνες, κάνοντας έτσι χρήση των πολλαπλών πυρήνων του επεξεργαστή. Οι εργάτες αναλαμβάνουν την εκτέλεση των εργασιών οι οποίες περιγράφονται με τον μηχανισμό που αναλύθηκε προηγουμένως. Καθώς τα νήματα εργάτες τρέχουν παράλληλα σε διαφορετικές επεξεργαστικές μονάδες, επιτυγχάνεται παράλληλη εκτέλεση των εργασιών που δημιουργούνται στο πρόγραμμα χρήστη. Κατά το πέρας μία εργασίας, ο descriptor αυτός αποστέλλεται πίσω στον δημιουργό της εργασίας, ώστε αυτός να ειδοποιηθεί για το πέρας εκτέλεσής της, καθώς και να ανακτήσει αποτελέσματα τα οποία περάσθηκαν με αναφορά μέσω των παραμέτρων.

Διάφορες βελτιστοποιήσεις μπορούν να συναντηθούν στην υλοποίηση της βιβλιοθήκης, όπως την εξής. Κατά την αρχικοποίηση δημιουργείται ένα πλήθος descriptors και αντί για να καταστρέφονται αυτοί όταν τελειώσει η εκτέλεση της εργασίας που αφορούν και να δημιουργηθούν νέοι στη συνέχεια, αυτοί επαναχρησιμοποιούνται για την περιγραφή μίας νέας εργασίας, με αποτέλεσμα να ελαχιστοποιείται η ανάγκη για δημιουργία νέων descriptors. Στην περίπτωση αυτή, η δημιουργία εργασιών να απαιτεί μονάχα την αρχικοποίηση της συνάρτησης και των δεδομένων (παραμέτρων) σε έναν υπάρχων descriptor. Ακόμα και στην περίπτωση όπου δεν βρεθούν έτοιμοι descriptors, χάρη στο γεγονός ότι μιλάμε για απλές δομές της γλώσσας υλοποίησης (στο επίπεδο του χρήστη), το κόστος υλοποίησης των νημάτων επιπέδου χρήστη είναι σημαντικά μικρότερο από αυτό άλλων υλοποιήσεων του μοντέλου εργασιών που χρησιμοποιούν κλήσεις του λειτουργικού συστήματος.

Δημιουργία και δρομολόγηση εργασιών

Για τη δρομολόγηση των εργασιών (δηλαδή των descriptors που τις περιγράφουν), χρησιμοποιούνται πολλαπλές ουρές σε κάθε διεργασία. Κάθε διεργασία υλοποιεί τις δικές τους ουρές, έχοντας έτσι κατανεμημένη λειτουργία των ουρών, αποφεύγοντας τον κεντρικοποιημένο σχεδιασμό και τους περιορισμούς που θέτει αυτός. Κάθε

νήμα-εργάτης κατέχει μία δημόσια ουρά. Επίσης, κάθε διεργασία κατέχει μία ιδιωτική και μία δημόσια ουρά, οι οποίες αφορούν τον κόμβο στον οποίο βρίσκεται. Όταν κάποιο νήμα-εργάτης τερματίσει την εκτέλεση της εργασίας που έχει αναλάβει, η ανασταλεί η εκτέλεση αυτής, αναζητά για διαθέσιμους descriptors στην ιδιωτική του ουρά και εάν δεν βρεθεί κάποιος, συνεχίζουν αναζητώντας κάποια διαθέσιμη εργασία στις ουρές (ιδιωτική και δημόσια) του κόμβου. Εάν είναι ενεργοποιημένος ο μηχανισμός κλοπής, τότε το νήμα-server που θα περιγραφθεί παρακάτω, μπορεί να ελέγξει για διαθέσιμες εργασίες στη δημόσια ουρά κάποιου απομακρυσμένου κόμβου. Κάθε διεργασία κατέχει ένα ειδικό νήμα-server στην περίπτωση όπου υπάρχει παραπάνω από μία διεργασία. Οι προσβάσεις μνήμης από τα νήματα-εργάτες αφορούν κοινόχρηστη μνήμη, στα πλαίσια της ίδιας διεργασίας. Το νήμα αυτό είναι υπεύθυνο για τις επικοινωνίες με απομακρυσμένες διεργασίες, οι οποίες πραγματοποιούνται με τη μεταφορά descriptors από και προς απομακρυσμένες ουρές.

2.1.5 Η διεπαφή OMPi-TORC

Η μέθοδος επικοινωνίας του τμήματος του μεταγλωττιστή με τη βιβλιοθήκη εκτέλεσης TORC είναι με την κλήση συναρτήσεων της βιβλιοθήκης, ακριβώς όπως και σε κάθε άλλη βιβλιοθήκη χρόνου εκτέλεσης του OMPi. Οι κύριες συναρτήσεις που μας ενδιαφέρουν είναι η `torc_create_ox` και ή `torc_tasksync`, οι οποίες παράγονται από τον μεταγλωττιστή, βάσιν των pragmas και clauses του αρχείου εισόδου.

Η συνάρτηση δημιουργίας εργασιών

Η συνάρτηση `torc_create_ox` υλοποιεί τη δημιουργία μίας εργασίας, συμπεριλαμβανόμενης της δρομολόγησής της στη συστάδα. Ως παραμέτρους δέχεται ορισμένες οδηγίες που αφορούν τον κόμβο στον οποίο επιθυμεί ο χρήστης να εκτελεσθεί η εργασία, εάν θα γίνει εκτέλεση σε λειτουργία SPMD, οδηγίες που αφορούν τις λειτουργίες του νέου δρομολογητή FTeQ (υποενότητα 3.2.1) και δείκτες προς τη συνάρτηση εργασίας και τη συνάρτηση callback.

Τέλος, δέχεται όλες τις παραμέτρους της συνάρτησης εργασίας. Αυτές οι παράμετροι μπορεί να αφορούν τοπικά αντίγραφα των πραγματικών παραμέτρων τα οποία παράγονται από τον μεταγλωττιστή, στις περιπτώσεις που είναι απαραίτητο. Η βιβλιοθήκη χειρίζεται αυτές τις παραμέτρους του προγράμματος, ανάλογα με το εάν η εργασία θα δρομολογηθεί στον τρέχων κόμβο, ή σε απομακρυσμένο, όπου

όπως αναφέραμε θα χειριστεί διαφανώς τη μεταφορά αυτών των παραμέτρων με χρήση της υλοποίησης του προτύπου MPI που παρέχεται.

Η συνάρτηση συγχρονισμού

Οι εργασίες έχουν μεταξύ τους σχέση γονέα-παιδιού. Ως συνέπεια, όλες οι εργασίες γνωρίζουν όλα τα παιδιά τους. Χάρη σε αυτή την ιδιότητα δίνεται η δυνατότητα για μία εργασία να περιμένει όλους τις εργασίες παιδιά που έχει δημιουργήσει, σε οποιοδήποτε σημείο της εκτέλεσης. Αυτός ο μηχανισμός είναι κρίσιμος και απολύτως απαραίτητος σε ένα περιβάλλον παράλληλης εκτέλεσης. Συγκεκριμένα, στις εφαρμογές μας δεν υπήρχε περίπτωση όπου δεν χρησιμοποιήθηκε αυτή η λειτουργία, είτε ως μηχανισμός συγχρονισμού, είτε ως απαραίτητο εργαλείο για την υλοποίηση της λογικής του προγράμματος.

Κεφαλαιο 3

Επεκτασεισ του προγραμματιστικου μοντελου

3.1 Επεκτάσεις του μεταγλωττιστή

3.2 Επεκτάσεις της βιβλιοθήκης εκτέλεσης TORC

3.3 Σύνταξη προσθήκων των HOMPI και EFTeQ

3.1 Επεκτάσεις του μεταγλωττιστή

3.1.1 Εκτέλεση task ως απλή συνάρτηση

Η πρώτη βελτίωση που πραγματοποιήσαμε στο προγραμματιστικό μοντέλο HOMPI ήταν η δυνατότητα εκτέλεσης κάποιας συνάρτησης εργασίας ως απλή συνάρτηση της γλώσσας C. Για να πραγματοποιηθεί αυτό, κατά την παραγωγή του τροποποιημένου κώδικα από το συντακτικό δένδρο δημιουργούνται δύο αντίγραφα της συνάρτησης η οποία ορίσθηκε ως συνάρτηση εργασίας.

Το πρώτο αντίγραφο αποτελεί ένα αμετάβλητο αντίγραφο της αρχικής συνάρτησης, το οποίο διατηρεί το αρχικό όνομα της συνάρτησης. Το δεύτερο αντίγραφο είναι ο τροποποιημένος κατάλληλα κώδικας ώστε να είναι συμβατός με την εκτέλεση από τη βιβλιοθήκη TORC. Το αντίγραφο αυτό λαμβάνει μία επέκταση στο όνομα συνάρτησης. Κατά την παραγωγή κώδικα δημιουργίας εργασίας για τη βιβλιοθήκη TORC, χρησιμοποιείται η δεύτερη, τροποποιημένη συνάρτηση.

Η προσθήκη αυτή απλοποιεί σημαντικά τον κώδικα, παρέχοντας τη δυνατότητα στον χρήστη να μπορεί να εκτελέσει μία συνάρτηση είτε ως εργασία, είτε ως απλής συνάρτησης. Έτσι μπορούμε να θεωρήσουμε ότι έχουμε πια δύο εκδόσεις μίας συνάρτησης, τη σειριακή και την παράλληλη. Εάν να κληθεί η συνάρτηση εργασία, τότε επιτυγχάνεται παραλληλισμός. Όμως, ο χρήστης δεν χρειάζεται να τροποποιεί δύο συναρτήσεις σε κάθε περίπτωση όπου επιθυμεί να τροποποιήσει τον αλγόριθμο. Ένα χρήσιμο σενάριο εφαρμογής αυτής της λειτουργίας είναι η επιλογή από τον χρήστη να εκτελέσει τον κώδικα σειριακά, όπως σε σενάριο που θα δείξουμε στην επόμενη ενότητα.

Σκοπός τέτοιων προσθηκών είναι η απλοποίηση και παραγωγή πιο ευανάγνωστου κώδικα και σύντομου κώδικα, ενώ αυτή η λειτουργία μπορεί πλέον να εκφραστεί ακόμα πιο εύκολα και σύντομα με την προσθήκη της οδηγίας ''If'' που θα περιγράψουμε επόμενη.

3.1.2 Υλοποίηση της οδηγίας ''If'' του προτύπου OpenMP στον HOMPI

Η οδηγία ''If'' ορίζεται από το πρότυπο OpenMP ως εξής: εάν η συνθήκη αληθεύει, τότε θα δημιουργηθεί ένα task, διαφορετικά ο κώδικας θα εκτελεσθεί σειριακά, δίχως κάποια τροποποίηση. Στην περίπτωσή μας, μπορεί να μεταφρασθεί ως την εκτέλεση της συνάρτησης εργασίας εάν ισχύει η δοθείσα συνθήκη και ή την εκτέλεση της μη τροποποιημένης συνάρτησης, διαφορετικά.

Για να υλοποιηθεί αυτή η οδηγία, έπρεπε να τροποποιήσουμε το κομμάτι του μεταγλωττιστή. Για να υποστηριχθεί κάποια καινούργια οδηγία στον μεταφραστή, απαιτείται η δημιουργία και διαχείριση ενός νέου κόμβου στο συντακτικό δέντρο. Ο μεταγλωττιστής πρέπει αρχικά να παράγει αυτό τον κόμβο όταν συναντά την συγκεκριμένη οδηγία. Ο κόμβος αυτός περιλαμβάνει επίσης τη συνθήκη της οδηγίας, την οποία πρέπει να διαχειρισθεί ο μεταγλωττιστής.

Στη συνέχεια, πρέπει κατά την παραγωγή του κώδικα της δημιουργίας εργασίας, να ελέγχει για την ύπαρξη της συγκεκριμένης οδηγίας στον γονικό κόμβο της δημιουργίας εργασίας, και να πράττει ανάλογα. Εάν υπάρχει αυτή η οδηγία, παράγεται ο κώδικας σύμφωνα με τον αλγόριθμο 3.1

Αλγόριθμος 3.1 Παραγωγή κώδικα υπό την ύπαρξη If clause

```
1: if ifClauseCondition then
2:     function_torctask()
3: else
4:     function()
5: end if
```

3.1.3 Οι ειδικές περιπτώσεις "here" και "remote" της φράσης atnode

Για τη διευκόλυνση του χρήστη και τη βελτίωση του προγραμματιστικού μοντέλου, υλοποιήθηκαν οι εξής ειδικές παράμετροι για την λειτουργία atnode, η οποία αφορά την εκτέλεση σε κάποιον συγκεκριμένο κόμβο. Οι επιπλέον αυτές φράσεις έχουν σκοπό την απλοποίηση του τελικού προγράμματος, μέσω της αποφυγής κλήσεων συναρτήσεων. Η υλοποίησή τους ακολουθεί όμοια μέθοδο με αυτήν που έχουμε δει έως τώρα, δηλαδή την δημιουργία ενός νέου κόμβου του συντακτικού δένδρου. Η ύπαρξη αυτού ελέγχεται κατά την παραγωγή της εντολής δημιουργίας εργασίας, ώστε να παραχθεί ο κατάλληλος κώδικας. Για τη συγκεκριμένη περίπτωση, τροποποιήθηκαν οι συναρτήσεις διεπαφής του μεταγλωττιστή με τη βιβλιοθήκη TORC, ώστε να περνάμε ειδικά flags στη βιβλιοθήκη σε χρόνο εκτέλεσης, τα οποία σηματοδοτούν ειδικές περιπτώσεις όπως τις δύο αυτές που θα αναφέρουμε. Καθώς διαχωρίστηκαν πια τα flags και η παράμετρος της οδηγίας atnode της διεπαφής, η παράμετρος αυτή μπορεί να ελεγχθεί κατά τον χρόνο εκτέλεσης για τυχών λανθασμένες τιμές.

Η παράμετρος "here" δίνει την εντολή στη βιβλιοθήκη TORC κατά τον χρόνο εκτέλεσης να εκτελέσει την νέα εργασία στον ίδιο κόμβο με αυτόν που τη δημιουργεί, ενώ η παράμετρος "remote" την εντολή να εκτελεσθεί η εργασία σε οποιονδήποτε κόμβο εκτός αυτού. Και οι δύο αυτές λειτουργίες υλοποιούνται στο επίπεδο της βιβλιοθήκης, με την κλήση της αντίστοιχης συνάρτησης. Για την εκτέλεση στον τοπικό κόμβο, η εργασία εισάγεται στην ιδιωτική ουρά του τρέχων κόμβου, ώστε να μην μπορεί να κλαπεί από άλλες διεργασίες. Για την απομακρυσμένη εκτέλεση, υλοποιήθηκε μία νέα συνάρτηση, η οποία λειτουργεί όμοια με την προκαθορισμένη συνάρτηση δρομολόγησης της βιβλιοθήκης. Η προκαθορισμένη συνάρτηση δρομολογεί της εργασίες κυκλικά στους κόμβους, ενώ η νέα συνάρτηση εκτελεί τον ίδιο

αλγόριθμο, αποφεύγοντας όμως την τρέχουσα.

Μπορούν να υπάρξουν διάφορα σενάρια χρήστης της κάθε περίπτωσης. Παραδείγματος χάρην, η πρώτη περίπτωση μπορεί να χρησιμοποιηθεί όταν ο χρήστης επιθυμεί να εκτελέσει μία εργασία εσκεμμένα στον τρέχων κόμβο, είτε για χρήση σε παραλληλισμό κοινόχρηστης μνήμης, είτε για την αποφυγή της κλοπής της εργασίας από κάποιαν διεργασία σε απομακρυσμένο κόμβο, πράγμα το οποίο μπορεί να φέρει ανεπιθύμητες μεταφορές δεδομένων, ή η εργασία να μην είναι κατάλληλη για εκτέλεση σε απομακρυσμένο κόμβο. Η εκτέλεση σε απομακρυσμένο κόμβο μπορεί να χρησιμοποιηθεί σε σενάριο όπου κάποιος κύριος κόμβος αναθέτει εργασίες σε απομακρυσμένες συσκευές, ή όταν δεν θέλουμε να επιβαρύνουμε τον τρέχων κόμβο με κάποια συγκεκριμένη εργασία.

3.1.4 Ασύγχρονη εκτέλεση κώδικα αρχικοποίησης

Ένα αρκετά πιθανό σενάριο (το οποίο συναντήθηκε αρκετά συχνά στις εφαρμογές μας), είναι το σενάριο όπου κατά την αρχικοποίηση της εφαρμογής απαιτήθηκε η εκτέλεση του ίδιου task από κάθε απομακρυσμένη διεργασία. Στην περίπτωση αυτήν, η λύση μας απαιτεί τη δημιουργία ενός task για κάθε μία τέτοια διεργασία. Στη χειρότερη περίπτωση, η λύση αυτή περιλαμβάνει μία ανάθεση, αποστολή και παραλαβή (κενών) αποτελεσμάτων για κάθε διεργασία, καθώς και μία διαδικασία συγχρονισμού, όπου ο συντονιστής πρέπει να περιμένει τις εργασίες αυτές να τερματίσουν, ώστε να είναι βέβαιος ότι αυτές είναι έτοιμες για την εκτέλεση tasks.

Στην καλύτερη περίπτωση, μπορούμε να αποφύγουμε τη διαδικασία συγχρονισμού, προσθέτοντας επιπλέον λογική στο πρόγραμμα, όπως θα περιγράψουμε στο κεφάλαιο 7. Δυστυχώς όμως, στην τρέχουσα της μορφή, η βιβλιοθήκη TORC δεν εκμεταλλεύεται τη λειτουργία Broadcast του προτύπου MPI, η οποία θα μπορούσε να ελαχιστοποιήσει τις μεταφορές δεδομένων π.χ. εάν χρησιμοποιούταν κάποιο δίκτυο κοινόχρηστου μέσου.

Λόγω των παραπάνω, ως χρήσιμη προσθήκη για ένα σύστημα της μορφής του HOMPI, κρίναμε την εξής λειτουργία, στις περιπτώσεις όπου η αρχικοποίηση αυτή δεν απαιτεί κάποια μεταφορά δεδομένων από ή/και προς τον συντονιστή, πληροφορίας που γνωρίζει μονάχα ο συντονιστής.

Ο χρήστης μπορεί να ορίσει μία συνάρτηση με το όνομα `hompi_init_func`, η οποία θα εκτελεσθεί πριν την αρχικοποίηση των διεργασιών. Αυτή η συνάρτηση θα εκτε-

λεσθεί σε κάθε διεργασία με κατανεμημένο τρόπο.

Με αυτό τον τρόπο, αποφεύγεται η μεταφορά δεδομένων ανάμεσα στις διεργασίες της συστάδας. Η μεταφορά αυτή είχε το επιπλέον μειονέκτημα ότι ήταν μία κεντρικοποιημένη διαδικασία, όπου ο συντονιστής έπρεπε να αποστείλει τα ίδια δεδομένα σε όλες τις άλλες διεργασίες, δίχως τη χρήση Broadcast, το οποίο σε ορισμένες περιπτώσεις μπορεί να μείωνε κάποιον από τον φόρτο του συντονιστή. Η υλοποίηση αυτή, όπως και κάθε άλλη κεντρικοποιημένη υλοποίηση, δεν είναι κλιμακώσιμη. Ακόμα και όταν ο αριθμός των απομακρυσμένων διεργασιών δεν ξεπεράσει τον αριθμό των νημάτων του συντονιστή, ενδέχεται να υποφέρουμε από άλλα προβλήματα, όπως race conditions για (περιορισμένους) πόρους μνήμης είτε του πυρήνα, είτε υλικού παρακάτω επιπέδων που υλοποιεί τις μεταφορές αυτές.

Μία εκτίμηση της λειτουργίας αυτή θα γίνει στα επόμενα κεφάλαια.

3.2 Επεκτάσεις της βιβλιοθήκης εκτέλεσης TORC

3.2.1 Ο δρομολογητής εργασιών EFTeQ (Expandable/Flexible TORC enQueuer)

Ως επιπλέον επέκταση υλοποιήθηκε ένας νέος δρομολογητής, με το όνομα EFTeQ, για το περιβάλλον εκτέλεσης TORC. Ο δρομολογητής αυτός είναι ένα ανεξάρτητο σύστημα, το οποίο μπορεί να χρησιμοποιηθεί όταν επιθυμείται από τον χρήστη.

Ο δρομολογητής TeQ έχει σκοπό την χρήση σε μη ομοιογενή κατανεμημένα συστήματα όπου ο κάθε κόμβος έχει όχι μόνο διαφορετική υπολογιστική ισχύ, αλλά και διαφορετικές δυνατότητες επεξεργασίας δεδομένων. Ο δρομολογητής έχει δύο κύριους στόχους. Ο πρώτος από αυτούς είναι η ίση κατανομή του συνολικού φόρτου εργασίας στη συστάδα. Για να πραγματοποιηθεί δίκαια αυτή η κατανομή, πρέπει να ληφθούν υπ όψιν οι επεξεργαστικές ικανότητες και δυνατότητες κάθε κόμβου, όπως αυτές δόθηκαν από τον χρήση. Έτσι, αποφεύγεται κάποια πιθανή δρομολόγηση κατά την οποία μεγάλος φόρτος εργασίας ανατίθεται σε "αδύναμους" κόμβους, το πέρας των οποίων διεργασιών μπορεί να ήταν απαραίτητο για την συνέχεια της εκτέλεσης, θέτοντας έτσι τους άλλους πιο "δυνατούς" κόμβους σε μία κατάσταση αναμονής, η οποία θα μείωνε τα ποσοστά αξιοποίησης των πόρων μας. Εναλλακτικά, αυτοί οι αδρανείς κόμβοι θα μπορούσαν να κλέψουν την εργασία αυτήν, κάτι

το οποίο όμως θα επέφερε επιπλέον κόστη μεταφοράς δεδομένων.

Ο δεύτερος στόχος του TeQ είναι η αποφυγή ανάθεσης εργασιών συγκεκριμένης φύσης, σε κόμβους τους οποίους ο προγραμματιστής δεν έχει κρίνει κατάλληλους για αυτές. Αυτή η κρίση μπορεί είτε να απευθύνεται σε αίτια όπως το γεγονός ότι ο χρόνος εκτέλεσης μερικών πολύ απλών πράξεων μπορεί να εξαρτάται μονάχα από των ρυθμό ρολογιού μίας επεξεργαστικής μονάδας, είτε για παράδειγμα το γεγονός ότι οι εμπορικές εκδόσεις καρτών γραφικών γενικού σκοπού δεν υποστηρίζουν αριθμητική κινητής υποδιαστολής διπλής ακρίβειας, σε αντίθεση με τις αντίστοιχες επαγγελματικές εκδόσεις τους. Ο υπολογισμός σε μεταβλητές διπλής ακρίβειας στο πρώτο σύστημα, θα επέφερε λάθος αποτελέσματα, άρα ο προγραμματιστής έχει την δυνατότητα να αποφύγει αυτό το λάθος αυτόματα.

Το σύστημα διαβάζει κατά την αρχικοποίηση του περιβάλλοντος εκτέλεσης πληροφορίες σχετικά με τους κόμβους, όπως αυτές δίνονται από τον προγραμματιστή του συστήματος, μέσω ενός απλού αρχείου κειμένου, του οποίου η μορφή δόθηκε παραπάνω. Οι πληροφορίες αυτές αφορούν την επεξεργαστική ισχύ κάθε κόμβου, καθώς και την δυνατότητα, ή αλλιώς την επιθυμία του προγραμματιστή, να ανατεθούν σε αυτούς τους κόμβους εργασίες οι οποίες περιλαμβάνουν ή όχι αριθμητικές πράξεις με αριθμούς κινητής υποδιαστολής ή αριθμούς με κινητή υποδιαστολή διπλής ακρίβειας.

Για να περαστεί αυτή η πληροφορία στο σύστημα, χρειάσθηκαν ορισμένες προσθήκες στον μεταφραστή OMPi. Η απαραίτητη τροποποίηση ώστε να μπορεί ο προγραμματιστής να περάσει τις πληροφορίες αυτές στο περιβάλλον εκτέλεσης σε χρόνο εκτέλεσης ήταν η προσθήκη ενός νέου clause, το λεγόμενο "hint", το οποίο είναι μέρος του task directive, και δέχεται ως παραμέτρους το είδος των πράξεων που περιλαμβάνει η εργασία, καθώς και το βάρος των πράξεων ακέραιας αριθμητικής και αριθμητικής κινητής υποδιαστολής της εργασίας αυτής. Επίσης αναφέρουμε σε αυτό το σημείο ότι αντικαθιστούμε τον όρο του κόμβου με αυτόν της διεργασίας, καθώς θα λάβουμε υπ όψιν την περίπτωση όπου κάθε κόμβος εκτελεί παραπάνω από μία διεργασία. Ένα σενάριο όπου θεωρούμε χρήσιμη αυτή την περίπτωση είναι όταν ο χρήστης επιθυμεί να εκτελεί σε έναν κόμβο μία διεργασία για εκτέλεση tasks στην κύρια επεξεργαστική μονάδα (CPU) και μία άλλη διεργασία για εκτέλεση tasks σε μία κάρτα γραφικών γενικού σκοπού.

Ο αλγόριθμος δρομολόγησης έχει δύο τρόπους λειτουργίας. Ο πρώτος είναι η στατική έκδοση, όπου κάθε εργασία θεωρείται ότι έχει ίδιο υπολογιστικό κόστος με

τις υπόλοιπες, ενώ ο δεύτερος δίνει την δυνατότητα στον προγραμματιστή να θέτει αυτός το βάρος της κάθε εργασίας, καθώς αυτή δημιουργείται.

Το περιβάλλον της TORC πρέπει να διατηρεί κάποιες επιπλέον πληροφορίες σε χρόνο εκτέλεσης. Όπως προαναφέρθηκε, δημιουργούνται κατά την αρχικοποίηση διανύσματα τα οποία αποθηκεύουν πληροφορίες σχετικά με την απόλυτη τιμή της επεξεργαστικής ισχύς κάθε κόμβου για κάθε είδος μαθηματικών λειτουργιών, όπως αυτές έχουν ορισθεί από τον προγραμματιστή από το αρχείο ρυθμίσεων, καθώς και τα αντίστοιχα flags που δηλώνουν τις επιτρεπτές μαθηματικές λειτουργίες ανά κόμβο. Επίσης πρέπει να διατηρηθούν πληροφορίες σχετικά με το άθροισμα του υπολογιστικού φόρτου που έχουν ανατεθεί έως τώρα, καθώς και τον φόρτο που έχει εκτελεσθεί από κάθε διεργασία. Ο φόρτος αυτός θα είναι είτε για το πλήθος των εργασιών που έχουν ανατεθεί, είτε για το άθροισμα των βαρών των εργασιών, όπως αυτά ορίσθηκαν κατά τον χρόνο εκτέλεσης από τον ίδιο τον προγραμματιστή και οι αντίστοιχες τιμές θα διατηρηθούν για την κάθε διεργασία της συστάδας. Οι δύο αυτές περιπτώσεις είναι διαφορετικές παραλλαγές του αλγορίθμου, οι οποίες θα περιγραφθούν παρακάτω.

Ο δρομολογητής είναι χωρισμένος σε δύο ξεχωριστές, πλήρως ανεξάρτητες μεταξύ τους υπο-οντότητες. Τον δρομολογητή ακέραιας αριθμητικής και των δρομολογητή αριθμητικής κινητής υποδιαστολής. Ο κάθε δρομολογητής διατηρεί τις ίδιες πληροφορίες και εκτελεί τον ίδιο αλγόριθμο, με την διαφορά ότι η δρομολόγηση μίας εργασίας αναλαμβάνεται από τον κατάλληλο δρομολογητή, ανάλογα με τα hints που έχουν δοθεί από το πρόγραμμα κατά τον χρόνο εκτέλεσης.

Τελευταία κρίσιμη πληροφορία για τον αλγόριθμο είναι το ποσοστό φόρτου που πρέπει να αναλάβει κάθε διεργασία. Το ποσοστό αυτό, το οποίο θα αναφέρεται από εδώ και στην συνέχεια ως "threshold" είναι κανονικοποιημένο ως προς την ολική πληροφορία που είναι γνωστή για τη συστάδα, και ορίζεται ως εξής. Για τον κόμβο i, στον δρομολογητή ακέραιας αριθμητικής:

$$IT_i = \frac{IP_i}{\sum_{j=1}^{N} IP_i} \tag{3.1}$$

Αντίστοιχα, για τον δρομολογητή αριθμητικής κινητής υποδιαστολής:

$$FT_i = \frac{FP_i}{\sum_{j=1}^{N} FP_i} \tag{3.2}$$

Στον παραπάνω τύπο, οι δείκτες i και j αφορούν μόνο διεργασίες οι οποίες

μπορούν να επεξεργασθούν αριθμούς υποκινητής υποδιαστολής, καθώς μόνο αυτές λαμβάνουν μέρος στον αντίστοιχο αλγόριθμο δρομολόγησης. Και στις δύο παραπάνω περιπτώσεις, η μεταβλητή δηλώνει το πλήθος των διεργασιών. Για λόγους ευκολίας, θα αναφερόμαστε στο threshold της διεργασίας i ως T_i, το οποίο κατά περίπτωση αντιπροσωπεύει το IT_i, το FT_i και το FP_i. Οι μεταβλητές αυτές δηλώνουν την υπολογιστική ισχύ του κόμβου για ακέραια αριθμητική, αριθμητική κινητής υποδιαστολής απλής και διπλής ακρίβειας, αντίστοιχα. Σε μια δεδομένη χρονική στιγμή, ας συμβολίσουμε με W_i το άθροισμα των βαρών των εργασιών που έχει πάρει ο κόμβος i μέχρι εκείνη τη στιγμή. Το "quota" είναι ουσιαστικά το ποσοστό του σε σχέση με το συνολικό βάρος των εργασιών που έχουν δρομολογηθεί στο σύστημα, δηλαδή:

$$Q_i = \frac{W_i}{\sum_{j=1}^{N} W_j} \tag{3.3}$$

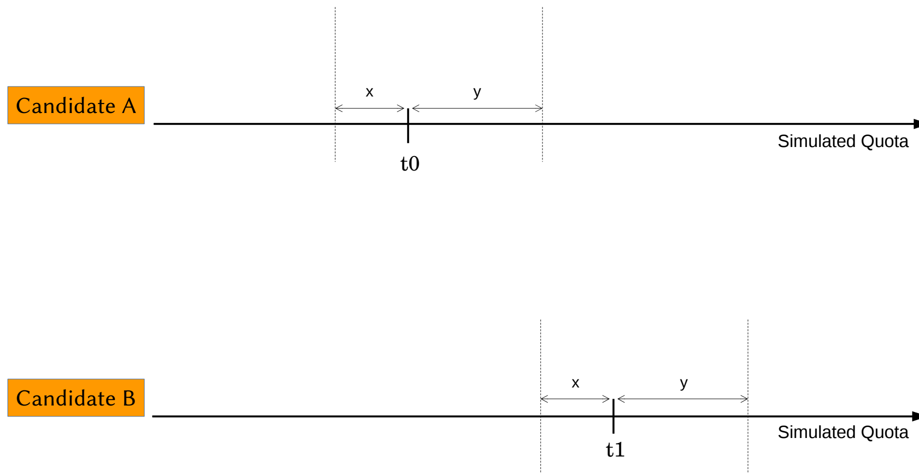
Σε οποιαδήποτε στιγμή κατά την εκτέλεση του προγράμματος, ο χρήστης μπορεί με την κλήση της συνάρτησης `torc_enable_teq` να διαλέξει ανάμεσα στις δύο παραλλαγές του αλγορίθμου, καθώς και τον δίκαιο ή απλό τρόπο λειτουργίας. Οι διάφορες αυτές εκδοχές του αλγορίθμου αναλύονται λεπτομερώς παρακάτω. Στην περίπτωση εναλλαγής μεταξύ των δύο τρόπων λειτουργίας αυτών το σύστημα είναι υπεύθυνο να επαναφέρει τις πληροφορίες του δρομολογητή στην αρχική τους κατάσταση.

3.2.2 Ο αλγόριθμος δρομολόγησης του EFTeQ

Ο αλγόριθμος λειτουργεί στοχεύοντας να διατηρεί αληθής μετά από κάθε δρομολόγηση νέας εργασίας την εξής συνθήκη: κάθε διεργασία πρέπει να έχει εκτελέσει ένα ποσοστό των συνολικών εργασιών που έχουν δημιουργηθεί, μικρότερο ή ίσο του ποσοστού φόρτου (threshold) που της αντιστοιχεί. Ο αλγόριθμος έχει δύο τρόπους λειτουργίας. Τον απλό τρόπο λειτουργίας και τον δίκαιο τρόπο λειτουργίας. Οι δύο τρόποι λειτουργίας θα περιγραφθούν ξεχωριστά.

Ο απλός τρόπος λειτουργίας

Η ιδέα του αλγορίθμου είναι σχετικά απλή, και στοχεύει στο να διατηρεί έπειτα από κάθε βήμα την παραπάνω συνθήκη αληθή. Μέσα σε έναν βρόγχο, αναζητεί ταυτόχρονα για κάποιαν διεργασία η οποία έχει εκτελέσει μικρότερο quota απ' ό,τι το



Σχήμα 3.1: Προσομοιωμένη κατάσταση συστήματος

threshold της και ταυτόχρονα τη διεργασία με το ελάχιστο quota. Εάν καμία διεργασία δεν είναι κάτω από το threshold της, τότε επιλέγεται ως νικητής αυτή με τη μεγαλύτερη σχετική απόσταση από το threshold της. Η νέα εργασία δρομολογείται στον νικητή αυτόν. Η σχετική απόσταση ορίζεται ως το ποσοστό της απόστασης από το threshold ως προς το threshold της αντίστοιχης εργασίας.

Ο δίκαιος τρόπος λειτουργίας

Στον δίκαιο τρόπο λειτουργίας, ο αλγόριθμος ξεκινά εκτελώντας μία προσπέλαση του διανύσματος των quota της τρέχουσας χρονικής στιγμής. Κατά αυτό το πέρασμα εισάγονται σε μία λίστα όλες οι διεργασίες οι οποίες έχουν quota μικρότερο από το αντίστοιχο threshold τους, στις οποίες θα αναφερόμαστε ως υποψήφιους. Ταυτόχρονα, βρίσκουμε τη διεργασία με τη μικρότερη σχετική απόσταση από το threshold, με τον ίδιο τρόπο που εκτελείται αυτή η διαδικασία στην απλή εκδοχή του αλγορίθμους.

Χαρακτηρίζουμε αυτόν τον τρόπο λειτουργίας ως δίκαιο σε σχέση με τον απλό τρόπο, καθώς ο αλγόριθμος δρομολόγησης που υλοποιεί αποφεύγει τη δημιουργία

σεναρίων όπου κάποιος υποψήφιος ξεπερνά κατά μεγάλο βαθμό το threshold του σε σχέση με τους άλλους. Επιπλέον, ο αρχικός αλγόριθμος επιλέγει ευρεστικά την πρώτη διεργασία που δεν έχει φτάσει το threshold της. Ως αποτέλεσμα, όχι μόνον ενδέχεται να εμφανισθεί η περίπτωση που μόλις περιγράψαμε, αλλά με αυτόν τον τρόπο επίσης ευνοούνται οι διεργασίες με χαμηλότερο ταυτοποιητή.

Εάν βρισκόμαστε στην απλή περίπτωση όπου έχουμε έναν μόνο υποψήφιο, αναθέτουμε την εργασία στη διεργασία αυτή και ο αλγόριθμος τερματίζει. Στην περίπτωση όπου έχουμε παραπάνω από έναν υποψήφιο, εκτελούμε τα παρακάτω βήματα. Η ιδέα του αλγορίθμου είναι ότι "προσομοιώνουμε" το σενάριο όπου επιλέχθηκε ο κάθε υποψήφιος και κοιτάμε την εικόνα του συστήματος έπειτα από την ανάθεση της εργασίας σε αυτόν. Αυτό που μας ενδιαφέρει είναι εάν αφού του αναθέσουμε μία εργασία, εάν θα ξεπεράσει ή εάν θα βρίσκεται κάτω από το threshold του. Για να προσομοιώσουμε την εικόνα του quota του υποψηφίου έπειτα από την ανάθεση της νέας εργασίας στον κόμβο i αρκεί να υπολογισθεί το εξής:

$$Q_i = \frac{W_i + tw}{(\sum_{j=1}^{N} W_j) + tw} \tag{3.4}$$

όπου tw το βάρος της νέας εργασίας.

Ο άξονας x στο σχήμα 3.1 δηλώνει τις τιμές των quota για κάθε υποψήφιο, με τα αντίστοιχα threshold τους να έχουν σημειωθεί ως $t0$ και $t1$. Η θέση στον άξονα δηλώνει το quota του υποψηφίου στο σύστημα αφού πραγματοποιηθεί η προσομοίωση της δρομολόγησης του νέου task σε αυτόν. Η τιμή dist για κάθε διεργασία i ορίζεται ως

$$D_i = \frac{x}{T_i} \tag{3.5}$$

άρα είναι αντιστρόφως ανάλογη του threshold κάθε υποψηφίου. Η τιμή αυτή εκφράζει έναν λόγο της απόστασης ως προς το threshold και είναι το κριτήριο με βάση το οποίο αποφασίζεται ο επικρατέστερος υποψήφιος.

Εάν η τιμή αυτή είναι μικρότερη του μηδενός, σημαίνει ότι εάν αναθέσουμε την εργασία σε αυτό τον υποψήφιο, το quota του δεν θα ξεπεράσει το threshold του. Εάν είναι θετική, σημαίνει ότι το threshold θα ξεπερασθεί. Η ποσοστιαία τιμή αυτή παίζει σημαντικό ρόλο και διαφέρει στις δύο παρακάτω περιπτώσεις, όπου θα χρησιμοποιηθεί.

Θα δώσουμε προτεραιότητα σε υποψήφιους των οποίων το quota θα μείνει κάτω

από το threshold εάν νικήσουν, έναντι αυτών όπου θα το ξεπεράσει. Για αυτό τον λόγο, ξεκινάμε εξετάζοντας έναν-έναν αρχικά τους υποψήφιους της πρώτης περίπτωσης. Η σύγκριση δύο υποψηφίων γίνεται ως εξής. Εάν δύο υποψήφιοι A και B απέχουν από το threshold τους κατά απόσταση x, με thresholds $t0$ και $t1$, αντίστοιχα, εάν ισχύει ότι $t0 < t1$ όπως φαίνεται στο σχήμα, τότε ο υποψήφιος B απέχει σχετικά λιγότερο από τον στόχο (threshold) του. Κατά συνέπεια ο νικητής θα είναι ο υποψήφιος A. Γενικεύοντας, νικητής είναι αυτός ο οποίος έχει τη μεγαλύτερη τιμή distance, δηλαδή τη μεγαλύτερη σχετική απόσταση από τον στόχο του, ανάμεσα στους υποψηφίους που εξετάζονται σε αυτή την φάση του αλγορίθμου.

Εάν κατά την προηγούμενη φάση του αλγορίθμου δεν βρέθηκε κανένας τέτοιος υποψήφιος, τότε ο αλγόριθμος συνεχίζει εξετάζοντας τους υποψηφίους της άλλης περίπτωσης, δηλαδή των οποίων το quota θα ξεπεράσει το threshold τους εάν αναλάβουν το task. Σε αυτή την περίπτωση δουλεύουμε με συμπληρωματική λογική, δηλαδή επιλέγουμε τον υποψήφιο που θα βρίσκεται πιο κοντά στον στόχο του, έτσι ώστε ο νικητής να ξεπεράσει όσο δυνατόν λιγότερο το threshold του.

3.3 Σύνταξη προσθήκων των HOMPI και EFTeQ

Στην συνέχεια δίνονται παραδείγματα χρήσης για κάθε οδηγία.

3.3.1 Οι φράσεις οδηγιών atnode(here) και atnode(remote)

```
1  #pragma ompix taskdef
2  void func(){
3
4  }
5
6
7  // Execute this task explicitly on this node
8  #pragma ompix task atnode(here)
9  func();
10
11
12 // Execute this task on any node except the current one
13 #pragma ompix task atnode(remote)
14 func();
```

Οι παραπάνω φράσεις επιτρέπεται να χρησιμοποιηθούν με την οδηγία `task`. Επιτρέπεται μία φράση τύπου `atnode()` ανά οδηγία, είτε αυτή περιέχει κάποια έκφραση, είτε είναι της μορφής `atnode(here)` ή `atnode(remote)`.

3.3.2 Η φράση οδηγιών if

```
#pragma ompix taskdef
void func(){

}


/* Execute this function as a task
if and only if the condition is true.

If the condition is false, execute
as a normal function */
#pragma ompix task if (expression)
func();
```

Η φράση αυτή επιτρέπεται να χρησιμοποιηθεί με την οδηγία task, ενώ επιτρέπεται μονάχα μία τέτοια φράση ανά οδηγία. Η έκφραση που περνάτε ως παράμετρος ακολουθεί τους κανόνες σύνταξης εκφράσεων της γλώσσας C.

3.3.3 Σύνταξη αρχείου ρυθμίσεων συστάδας του EFTeQ

$$
\begin{aligned}
\langle \text{configuration} \rangle &\models \langle \text{num-of-nodes} \rangle \langle \text{node-entry} \rangle \\
\langle \text{node-entry} \rangle &\models \langle \text{I-Vec} \rangle \langle \text{F-Vec} \rangle \langle \text{D-Vec} \rangle \langle \text{P-Cap} \rangle \;\mid\; \langle \text{node-entry} \rangle \\
\langle \text{node-entry} \rangle &\models \lambda \\
\langle \text{I-Vec} \rangle &\models \langle \text{unsigned-short-int} \rangle \\
\langle \text{F-Vec} \rangle &\models \langle \text{unsigned-short-int} \rangle \\
\langle \text{D-Vec} \rangle &\models \langle \text{unsigned-short-int} \rangle \\
\langle \text{unsigned-short-int} \rangle &\models \textit{integer in range 1-255} \\
\langle \text{P-Cap} \rangle &\models \textit{integer in range 0-7}
\end{aligned}
$$

Ο ακέραιος `P-Cap` δηλώνει τις υπολογιστικές δυνατότητες και ικανότητες του κόμβου. Κάθε bit δηλώνει την δυνατότητα επεξεργασίας αριθμητικής συγκεκριμένων τύπων δεδομένων.

Το λιγότερο σημαντικό ψηφίο αντιπροσωπεύει τους ακέραιους αριθμούς και πρέπει να είναι πάντα ίσο με ένα. Το δεύτερο λιγότερο σημαντικό ψηφίο αντιπροσωπεύει τους αριθμούς κινητής υποδιαστολής απλής ακρίβειας, ενώ το τρίτο λιγότερο σημαντικό τον αντίστοιχο τύπο δεδομένων διπλής ακρίβειας.

3.3.4 Οι φράσεις hint του δρομολογητή EFTeQ

```

#pragma ompix taskdef
void func(){

}


```

0	Η εργασία αυτή δεν θα δρομολογηθεί από τον EFTeQ
1	Εμπεριέχονται υπολογισμοί κινητής υποδιαστολής απλής ακρίβειας
2	Εμπεριέχονται υπολογισμοί κινητής υποδιαστολής διπλής ακρίβειας
3	Δεν χρησιμοποιείται
4	Εμπεριέχονται μόνο υπολογισμοί ακέραιας αριθμητικής

```
8 #pragma ompix task hint (flags, weight)
9 func();
```

Η φράση αυτή επιτρέπεται να χρησιμοποιηθεί με την οδηγία `task`, ενώ επιτρέπεται μονάχα μία τέτοια φράση ανά οδηγία. Η τιμή flag δηλώνει πληροφορίες σχετικά με τις πράξεις που περιλαμβάνει η εργασία, όπως αυτές οριστούν από τον χρήστη, ως εξής.

Ως `weight` δίνεται μία έκφραση σύμφωνα με τους κανόνες της γλώσσας C.

Κεφαλαιο 4

Εφαρμογες

4.1 Εφαρμογές ταξινόμησης μονοδιάστατων πινάκων

4.2 Κατανεμημένος πολλαπλασιασμός δισδιάστατων πινάκων

4.3 Αναγνώριση προσώπων

4.1 Εφαρμογές ταξινόμησης μονοδιάστατων πινάκων

4.1.1 Merge Sort

Η εφαρμογή αυτή είναι μία υλοποίηση του γνωστού αλγορίθμου merge sort. Το πρόγραμμα αποτελεί μία τροποποιημένη έκδοση των αλγορίθμων της εργασίας [7], συμβατή με το προγραμματιστικό μοντέλο HOMPI. Η εφαρμογή 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, τα βέλη υποδηλώνουν επικοινωνίες μεταξύ των αντίστοιχων κόμβων (οι οποίες περιλαμβάνουν επικοινωνίας), ενώ οι διακεκομμένες γραμμές δηλώνουν τη δημιουργία μίας εργασίας στον συγκεκριμένο κόμβο. Η λειτουργία του αλγορίθμου λειτουργεί ως γνωστόν με την τεχνική Διαίρει-και-Βασίλευε. Στην πρώτη φάση, αυτήν της διαίρεσης, ο πίνακας διασπάται και ταξινομείται αναδρομικά σε δύο ίσα τμήματα, μέχρις ότου αυτοί οι υποπίνακες να περιέχουν μονάχα ένα στοιχείο. Στη δεύτερη φάση, αυτά τα δύο (ταξινομημένα πια) τμήματα κάθε διάσπασης συγχωνεύονται, δημιουργώντας έναν ενιαίο, ταξινο-

μημένο πίνακα. Η υλοποίησή μας διαφέρει από τον απλό αλγόριθμο σε δύο κύρια σημεία.

Πρώτον, ο διαχωρισμός κάθε πίνακα σε δύο τμήματα συμβαίνει με τον εξής τρόπο: το πρώτο ήμισυ του πίνακα ταξινομείται με τη δημιουργία μίας εργασίας στον τρέχοντα κόμβο, ενώ το άλλο ήμισυ αποστέλλεται σε έναν απομακρυσμένο κόμβο, με ταυτότητα $rank + 2^{depth}$, όπου $rank$ ο βαθμός του κόμβου που δημιουργεί την απομακρυσμένη πια αυτήν εργασία. Αυτός ο διαχωρισμός συνεχίζεται με τη δημιουργία εμφωλευμένων εργασιών μέχρις ότου να έχουμε μοιράσει τον αρχικό πίνακα σε ίσα τμήματα, ένα για κάθε κόμβο του δικτύου μας. Κατά την επιτυχή ταξινόμηση των δύο υποπινάκων, ο κόμβος που δημιούργησε τις δύο αυτές εργασίες είναι υπεύθυνος για τη συγχώνευση των δύο υποπινάκων. Κατά τη δημιουργία μία απομακρυσμένης εργασίας ο δημιουργός περνά στον όπου αυτή θα εκτελεσθεί το τμήμα που επιθυμεί να ταξινομήσει, το οποίο επιστρέφεται ταξινομημένο κατά το πέρας της εργασίας.

Η άλλη κύρια τροποποίηση του αλγορίθμου είναι ο τρόπος ταξινόμησης σε τοπικό επίπεδο. Ο κάθε κόμβος αναλαμβάνει την ταξινόμηση ενός ενιαίου τμήματος του πίνακα, όπως αυτό του στάλθηκε από κάποια διεργασία. Ο κόμβος έχει την επιλογή να ταξινομήσει το κάθε τμήμα αυτό με διάφορες μεθόδους. Μία από αυτή είναι με χρήση είτε του αλγορίθμου quicksort, αφού διασπάσει τον αρχικό πίνακα σε τμήματα μεγέθους $cutoff$ στοιχείων, όπου το $cutoff$ ορίζεται ανάλογα με το πλήθος των επεξεργαστών του κόμβου. Αυτή η τιμή θα χρησιμοποιείται από εδώ και στο εξής και θα αναφέρεται ως σημείο αποκοπής.

Εναλλακτικά, ο κόμβος μπορεί να ταξινομήσει το τμήμα αυτό χρησιμοποιώντας μία παράλληλη έκδοση του αλγορίθμου merge sort, παραλληλοποιημένου με τη χρήση των εργαλείων του προτύπου OpenMP, ή ακόμα και με χρήση εργασιών.

Επίσης υλοποιήθηκαν και δοκιμάσθηκαν μικρές τροποποιήσεις του αλγορίθμου όσον αφορά στη μεταφορά δεδομένων, ώστε να μελετηθούν οι επιπτώσεις της στις επιδόσεις του αλγορίθμου. Η μία τροποποίηση αφαιρεί τη μεταφορά του αρχικού (μη ταξινομημένου) υποπίνακα δεδομένων, υποθέτοντας ένα κοινόχρηστο σύστημα αρχείων, από το οποίο οι απομακρυσμένοι κόμβοι ανακτούν τον πίνακα που επιθυμεί η εφαρμογή να ταξινομήσει. Παραμένει όμως η επιστροφή των αποτελεσμάτων στον δημιουργό της εργασίας κατά το πέρας της εργασίας που εκτέλεσαν. Η δεύτερη τροποποίηση αναλύεται λεπτομερέστερα παρακάτω.

Ελαχιστοποιώντας τις επικοινωνίες

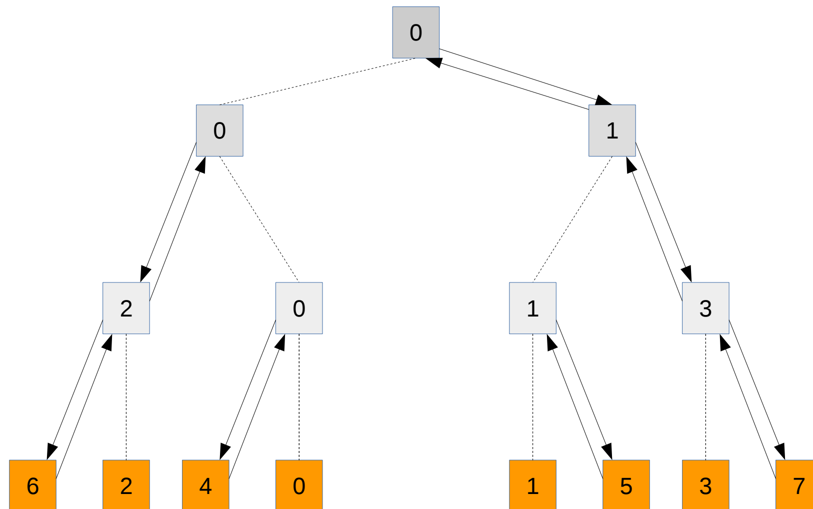
Η άλλη τροποποίηση αφορά τη βελτίωση του αλγορίθμου αποφεύγοντας μη απαραίτητες μεταφορές. Όπως μπορεί κανείς να παρατηρήσει στο δίκτυο ταξινόμησης του αρχικού αλγορίθμου, οποιαδήποτε μεταφορά σε βάθος μεγαλύτερου του ενός στο δένδρο αποτελεί επανα-αποστολή δεδομένων που έχουν ήδη μεταφερθεί. Για παράδειγμα, η μεταφορά από τον κόμβο 1 στον κόμβο 3, πρόκειται για μία επανα-αποστολή δεδομένων που έλαβε ο κόμβος 1 από τον κόμβο 0. Το ίδιο συμβαίνει και κατά την επιστροφή αυτών των δεδομένων. Με άλλα λόγια, ο κόμβος 1 λειτουργεί ως ενδιάμεσος κόμβος για την αποστολή των δεδομένων, μεταξύ του κόμβου 0 και του κόμβου 3. Καθώς βρισκόμαστε σε περιβάλλον κατανεμένης μνήμης, αυτές οι μεταφορές εισάγουν μη απαραίτητα χρονικά overheads. Στη νέα μας υλοποίηση, όλος ο λογικός διαχωρισμός των δεδομένων σε υποτμήματα συμβαίνει τοπικά, με τη χρήση εικονικών ταυτοποιητών κόμβων για κάθε εργασία. Αυτοί οι εικονικοί ταυτοποιητές ταυτίζονται με την τάξη των κόμβων στο αρχικό δέντρο ταξινόμησης και χρησιμοποιούνται ώστε να δημιουργήσουμε ένα τέτοιο δένδρο στον αρχικό κόμβο. Η ανάθεση της ταξινόμησης των τελικών υποτμημάτων στους απομακρυσμένους κόμβους συμβαίνει όταν έχουμε φτάσει σε φύλλα του εικονικού δένδρου αυτού (πορτοκαλί κόμβοι), αποφεύγοντας έτσι όλες τις μη απαραίτητες που περιγράψαμε παραπάνω. Η τάξη του κόμβου στον οποίο θα αποσταλεί κάθε εργασία ταυτίζεται με τον εικονικό ταυτοποιητή στο δένδρο αυτό. Τώρα πια κάθε τμήμα της εισόδου μεταφέρεται μία φορά, μεταξύ του κύριου κόμβου και του κόμβου φύλλο ο οποίος θα ταξινομούσε το αντίστοιχο υποπρόβλημα στο αρχικό δίκτυο.

Όταν το βάθος του δένδρου ταξινόμησης (όπως αυτό φαίνεται στο σχήμα 4.1) είναι ίσο με 1 αποστέλνονται $N/2$ στοιχεία σε απομακρυσμένους κόμβους. Σε βάθος 2 αποστέλλονται $N/4$ στοιχεία, κ.ο.κ. Συνεπώς, η υλοποίηση αυτή αποφεύγει την επανα-αποστολή

$$\sum_{i=2}^{d} \frac{N}{2^i} \tag{4.1}$$

στοιχείων, όπου 2^d το πλήθος των κόμβων του δικτύου, έναντι της αρχικής υλοποίησης.

Αυτή η υλοποίηση έχει το μοναδικό πρόβλημα 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



Σχήμα 4.1: Το δίκτυο ταξινόμησης του αλγορίθμου σε cluster 8 κόμβων.

του αλγορίθμου, όπως η συγχώνευση των ταξινομημένων υποπινάκων, δεν εκτελούνται πια κατανεμημένα, αλλά εκτελούνται σε έναν μόνο κόμβο. Ως λύση για αυτό το πρόβλημα υλοποιήσαμε μία παράλληλη εκδοχή του αλγόριθμου συγχώνευσης (merge) [8], όπως αυτή προτάθηκε στην [7], την οποία θα παρουσιάσουμε στην επόμενη ενότητα.

Το πρόβλημα που μόλις αναφέραμε είναι αυτό της συγχώνευσης δύο ταξινομημένων πινάκων, το οποίο έχει χρονική πολυπλοκότητα $O(N)$. Ακόμα και για πίνακες εκατομμυρίων στοιχείων, οι μετρήσεις μας έδειξαν ότι οι χρόνοι αυτοί είναι αμελητέοι μπροστά στους χρόνους ταξινόμησης των δύο επιμέρους πινάκων. Παρόλο που η νέα αυτή υλοποίηση έχει το πρόβλημα του κεντρικού κόμβου, στην περίπτωση ενός δικτύου κοινόχρηστου μέσου οι επικοινωνίες που γλιτώσαμε ευνόησαν σημαντικά τις επιδόσεις.

4.1.2 Παραλληλοποιημένη συγχώνευση (Merge)

Η ιδέα του αλγορίθμου βασίζεται στην τεχνική Διαίρει-και-Βασίλευε, που λειτουργεί ως εξής. Χωρίζουμε τη διαδικασία της συγχώνευσης σε αμοιβαία αποκλειόμενα

υποπροβλήματα, τα οποία μπορούν να εκτελεσθούν ταυτόχρονα. Η είσοδος του προβλήματος είναι δύο ταξινομημένοι πίνακες, έστω A και B, με την επιθυμητή έξοδος να είναι ένας ενιαίος ταξινομημένος πίνακας, ο οποίος περιλαμβάνει τα στοιχεία και των δύο πινάκων εισόδου.

Αρχικά, βρίσκουμε τη διάμεσο του πίνακα A, ή το μεσαίο σε δείκτη στοιχείο του πίνακα. Εάν υπάρχουν διπλότυπες εμφανίσεις στοιχείων αυτές οι δύο τιμές θα διαφέρουν, αλλά με την εξαίρεση ακραίων περιπτώσεων, το μεσαίο στοιχείο θα αρκεί, καθώς θα διαχωρίζει στον αρχικό πίνακα σε υποπίνακες αρκετά κοντινού μεγέθους για τους σκοπούς μας. Παρόλα αυτά, η εύρεση της διαμέσου μπορεί να βρεθεί σε γραμμικό χρόνο όπως είναι γνωστό, η οποία εύρεση μπορεί επίσης να παραλληλοποιηθεί, οπότε η επιλογή αυτή αφορά θέμα βελτιστοποίησης και αφήνεται στην ευθύνη του προγραμματιστή, ανάλογα με το σενάριο χρήσης. Έπειτα, βρίσκουμε τη σωστή θέση αυτού του στοιχείου στον πίνακα B, δηλαδή τη θέση στην οποία θα εισάγαμε το στοιχείο αυτό ώστε να διατηρηθεί ταξινομημένος ο πίνακας B.

Αρκεί τώρα να παρατηρήσουμε ότι τα (ταξινομημένα) στοιχεία του πίνακα A μικρότερων της διαμέσου, μπορούν να συγχωνευθούν με αυτά του πίνακα B τα οποία είναι μικρότερα της διαμέσου. Αυτό μπορεί να εκτελεσθεί ως μία εργασία. Το ίδιο ισχύει και για τα υπόλοιπα στοιχεία, δηλαδή αυτά των πινάκων A και B που είναι μεγαλύτερα της διαμέσου. Αυτές οι δύο εργασίες είναι ανεξάρτητες μεταξύ τους, κατά συνέπεια μπορούν να εκτελεσθούν παράλληλα σε ένα περιβάλλουν κοινόχρηστης μνήμης.

Η ίδια διαδικασία διαίρεσης μπορεί να συνεχισθεί αναδρομικά μέχρις ότου να φτάσουμε σε ένα επιθυμητό μέγεθος υποπινάκων, όπου θα εκτελέσουμε σειριακή συγχώνευση. Αυτή η ιδέα ήταν εύκολο να υλοποιηθεί χάριν στο μοντέλο προγραμματισμού με εργασίες του HOMPI, προσθέτοντας τα κατάλληλα pragma στο αρχικό πρόγραμμα.

Κατά την εκτέλεση αυτών των πειραμάτων παρουσιάστηκαν γνωστά προβλήματα στον κόσμο του παράλληλου προγραμματισμού, όπως αυτό της ανισοκατανομής φόρτου και του φράγματος επιδόσεων που συναντάμε εξαιτίας σειριακά εκτελέσιμων υποπροβλημάτων-εργασιών. Το μοντέλο παράλληλου προγραμματισμού με εργασίες είναι ιδανικό σε τέτοιες περιπτώσεις, όπου προβλήματα ανισοκατανομής φόρτου εξαλείφονται, δεδομένου ότι υπάρχουν οι απαραίτητες γνώσεις και εμπειριά από την πλευρά του προγραμματιστή, όπως φάνηκε στα πειραματικά αποτελέσματα που θα παρουσιασθούν αργότερα, στα οποία η αποδοτικότητα ξεπερνά το

90%.

4.1.3 Bitonic/Quick Sort

Η εφαρμογή αυτή αποτελεί μία υλοποίηση του γνωστού αλγορίθμου bitonic sort, σε μία κατανεμημένη εκδοχή του, προσαρμοσμένη στο προγραμματιστικό μοντέλο HOMPI. Η διαφορά είναι ότι αφού φτάσουμε σε ένα αρκετά μικρό μέγεθος υποπίνακα, μπορούμε να επιλέξουμε ξανά ανάμεσα στην ταξινόμηση με τη χρήση του ίδιου (αλλά σειριακού) αλγορίθμου, ή τη χρήση του αλγόριθμου quicksort. Ο αλγόριθμος αποτελείται από δύο στάδια.

Στο πρώτο στάδιο, χωρίζουμε την είσοδο σε τμήματα μεγέθους που επιλέγουμε οι ίδιοι ως παράμετρο, την οποία θα αποκαλούμε βήμα. Ταξινομούμε με εναλλάξ σειρά ταξινόμησης αυτά τα τμήματα, χρησιμοποιώντας είτε τον αλγόριθμο αναδρομικά, είτε υποδιαιρούμε αναδρομικά την είσοδο μέχρι κάποιο μέγεθος και στη συνέχεια εφαρμόζοντας κάποιον αλγόριθμο της επιλογής μας (όπως τον αλγόριθμο quick sort) σε αυτό το τμήμα. Το πρώτο υποτμήμα θα είναι ταξινομημένο σε αύξουσα σειρά, το δεύτερο σε φθίνουσα, κ.ο.κ. Κατά το δεύτερο στάδιο του αλγορίθμου, το βήμα διπλασιάζεται επαναληπτικά, και τα τμήματα υπόκεινται bitonic συγχώνευση, από την οποία προκύπτει ένα ενιαίο, ταξινομημένο, τμήμα. Η διαδικασία αυτή έχει παραλληλοποιηθεί επίσης, και μπορεί να εκτελεσθεί είτε τοπικά, είτε απομακρυσμένα. Ο αλγόριθμος τερματίζει όταν το βήμα γίνει ίσο με το μέγεθος της εισόδου.

Το πρώτο στάδιο του αλγορίθμου εκτελείται κατανεμημένα, ενώ το δεύτερο επιλέγεται εάν θα εκτελεσθεί απομακρυσμένα ή τοπικά. Στο δεύτερο στάδιο, οι εναλλάξ ταξινομημένοι πίνακες συγχωνεύονται παράλληλα ανά δύο, με εναλλάξ σειρά ταξινόμησης, μέχρις ότου να καταλήξουμε σε δύο τελικούς πίνακες με διαφορετική σειρά ταξινόμησης, οι οποίοι θα συγχωνευθούν, έχοντας ως αποτέλεσμα τον τελικό, ταξινομημένο πίνακα.

4.1.4 Γρήγορη Ταξινόμηση (Quicksort)

Η εφαρμογή είναι μία τροποποιημένη παράλληλη εκδοχή του γνωστού αλγορίθμου γρήγορης ταξινόμησης, κατάλληλη για χρήση στο περιβάλλον της εργασίας αυτής. Η υλοποίησή μας δεν διαφέρει σημαντικά από την απλή, σειριακή εκδοχή του αλγορίθμου. Καθώς αυτός ανήκει στην οικογένεια των αλγορίθμων Διαίρει-και-Βασίλευε,

ταιριάζει πλήρως με το προγραμματιστικό μοντέλο των εργασιών.

Ο αρχικός αλγόριθμος κατά κάθε διαίρεση του υποπροβλήματος εκτελεί δύο κλήσεις συναρτήσεων, μία σε όλα τα στοιχεία του πίνακα που είναι μικρότερα του άξονα ταξινόμησης (pivot) και μία σε όλα αυτά μεγαλύτερα αυτού του στοιχείου. Οι δύο κλήσεις αυτές αποτελούν ανεξάρτητες μεταξύ τους λειτουργίες και ως συνέπεια μπορούν να εκτελεσθούν ως εργασίες, ταυτόχρονα. Η μόνη τροποποίηση που έπρεπε να γίνει στο αρχικό πρόγραμμα ήταν η προσθήκη των κατάλληλων pragmas στην αναδρομική συνάρτηση, έτσι ώστε αυτές να εκτελούνται ως εργασίες σε διαφορετικά νήματα εκτέλεσης, παράλληλα.

Όπως και στην εφαρμογή της παραλληλοποιημένης συγχώνευσης που παρουσιάσαμε, ισχύουν οι ίδιες παρατηρήσεις και πρακτικές για το σημείο αποκοπής, το μέγεθος δηλαδή των υποπροβλημάτων στο οποίο σταματάμε να δημιουργούμε παράλληλες εργασίες, και ταξινομούμε πια τον υποπίνακα σειριακά.

Το πρόγραμμα αυτό μπορεί να λειτουργήσει σε περιβάλλοντα κοινόχρηστης μνήμης, ή και σε συστάδες, δίχως καμία τροποποίηση.

4.2 Κατανεμημένος πολλαπλασιασμός δισδιάστατων πινάκων

Η εφαρμογή αυτή αποτελεί μία τροποποίηση υπάρχων προγραμμάτων για τον πολλαπλασιασμό πινάκων με τη χρήση του προτύπου OpenMP, η οποία εμφανίζει χρήσιμες παρατηρήσεις που αφορούν την παράλληλη επεξεργασία και πιο συγκεκριμένα τη μετατροπή παράλληλων προγραμμάτων κοινόχρηστης μνήμης σε προγράμματα κατανεμημένης μνήμης. Η εφαρμογή πρόκειται για την υλοποίηση του πολλαπλασιασμού δύο τετράγωνων πινάκων ίσων διαστάσεων.

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, το οποίο πρόκειται για παράμετρο του χρόνου εκτέλεσης. Κάθε πολλαπλασιασμός ενός τέτοιου (επίσης τετραγωνικού) υποπίνακα θεωρείται ως μία εργασία. Κάθε τέτοια εργασία χαρακτηρίζεται από έναν ταυτοποιητή, ο οποίος χρησιμοποιείται από τον εργάτη που θα αναλάβει την εργασία έτσι ώστε να γνωρίζει ποιο υποπρόβλημα πρόκειται να λύσει.

Το υπάρχων πρόγραμμμα χρησιμοποιούσε παραλληλισμό με τη χρήση των εργασιών του προτύπου OpenMP. Η μετατροπή αυτού του προγράμματος ώστε να είναι

συμβατό με το προγραμματιστικό μοντέλο που χρησιμοποιήσαμε ήταν τετριμμένη, στην περίπτωση ενός περιβάλλοντος κοινόχρηστης μνήμης. Η μόνη προσθήκη που απαιτήθηκε ήταν τα κατάλληλα pragmas στο αρχικό πρόγραμμα. Στην εκτέλεση με το πρότυπο OpenMP τα tasks αυτά αναλαμβανόταν από τα νήματα-εργάτες του OpenMP, ενώ τώρα αναλαμβάνονται από τους εργάτες της βιβλιοθήκης TORC.

Η μεταφορά όμως του προγράμματος στην εκδοχή κατανεμημένης μνήμης, δεν ήταν τόσο απλή όσο η προηγούμενη. Αρχικά, πρέπει να αποστείλουμε τους δύο πίνακες εισόδου σε κάθε απομακρυσμένο κόμβο κατά τη φάση αρχικοποίησης, ο οποίος είναι υπεύθυνος για την δέσμευση της μνήμης που χρειάζεται για να αποθηκευθούν αυτοί οι πίνακες. Σε αυτό το σημείο φαίνονται μικρά μειονεκτήματα του μοντέλου προγραμματισμού που χρησιμοποιούμε, όπως το πως μπορούμε να αποστείλουμε α) τους δισδιάστατους πίνακες εισόδου και β) τους διδιάστατους πίνακες αποτελεσμάτων, ενώ τα εργαλεία μας επιτρέπουν μονάχα την αποστολή μονοδιάστατων πινάκων.

Η απλή λύση είναι η δέσμευση συνεχόμενου χώρου μνήμης για την αποθήκευση των πινάκων εισόδου καθώς και εξόδου. Στην τυπική υλοποίηση δέσμευσης μνήμης δυναμικά για την αποθήκευση ενός δισδιάστατου πίνακα, συνήθως ακολουθούμε την εξής λογική. Κάθε γραμμή του πίνακα αναπαριστάται από έναν μονοδιάστατο πίνακα στοιχείων και ο κύριος πίνακας διατηρεί N αναφορές, μία για καθέναν καθέναν από αυτούς του πίνακες. Ένας δισδιάστατος πίνακας N γραμμών και N στηλών μπορεί να αναπαρασταθεί ως ένας μονοδιάστατος πίνακας μήκους $N \times N$, καθώς μας ενδιαφέρει μονάχα να αποθηκεύσουμε αυτά τα $N \times N$ στοιχεία. Σημειώνουμε ότι αυτή η τεχνική αποφεύγεται σκόπιμα στον προγραμματισμό συστημάτων για διάφορους λόγους επιδόσεων και καλών πρακτικών, όμως στην περίπτωσή μας είναι μία από τις θυσίες που πρέπει να γίνουν για την επίλυση του προβλήματος και δείχνει ένα από τα προβλήματα που μπορεί να συναντήσει κάποιος σε τέτοιες περιπτώσεις. Ο μονοδιάστατος πίνακας αυτός μπορεί να σταλθεί επιτυχώς στην περίπτωσή μας, για τους πίνακες εισόδου.

Στην περίπτωση των αποτελεσμάτων χρησιμοποιούμε την ίδια τεχνική. Όμως αυτή τη φορά ο κόμβος-συντονιστής ο οποίος είναι και υπεύθυνος για τη συλλογή αυτών των αποτελεσμάτων πρέπει να οργανώσει τα αποτελέσματα που λαμβάνει. Τα αποτελέσματα αυτά είναι ένας μονοδιάστατος πίνακας για κάθε υποπρόβλημα, ο οποίος αναπαριστά μη συνεχείς θέσεις του πίνακα αποτελεσμάτων. Ως συνέπεια αυτού, ο συντονιστής είναι υπεύθυνος να χαρτογραφήσει σωστά τα αποτελέσματα

αυτά κατά το πέρας κάθε εργασίας. Αυτό συμβαίνει με τη χρήση μίας κατάλληλης συνάρτησης callback, στην οποία ο συντονιστής εκτελεί αυτή την διαδικασία.

Μία άλλη βελτίωση που εφαρμόσθηκε στην υλοποίησή μας είναι η αφαίρεση λειτουργιών συγχρονισμού κατά την αρχικοποίηση του αλγορίθμου. Η τεχνική αυτή αναλύεται στο Κεφάλαιο 7. Με σκοπό τη μεθοδοποίηση της τεχνικής αυτής, υλοποιήσαμε την επιπλέον λειτουργία HOMPI Init, την οποία περιγράψαμε στο κεφάλαιο 2.

4.3 Αναγνώριση προσώπων

Η εξής εφαρμογή αφορά την υλοποίηση της μεθόδου [9] στο μοντέλο του HOMPI.

Σε αυτή την εφαρμογή θα δείξουμε μία περίπτωση όπου η μετατροπή ενός υπάρχοντος πηγαίου κώδικα, σε κώδικα του HOMPI έγινε με μεγάλη ευκολία και επέφερε σημαντική βελτίωση του χρόνου εκτέλεσης.

Η περιγραφή μας θα εστιάσει στον τρόπο λειτουργίας της εφαρμογής ώστε να περιγραφεί στη συνέχεια η παραλληλοποίηση αυτού, έναντι στο να εστιάσει στην ανάλυση του αλγορίθμου που υλοποιεί. Η εφαρμογή ξεκινά διαβάζοντας μία λίστα αρχείων εισόδου και δημιουργώντας μία λίστα με τα αρχεία που επιθυμεί ο χρήστης να αναλυθούν. Στη συνέχεια, επεξεργάζεται μία εικόνα κάθε φορά, σε πέντε φάσεις. Οι φάσεις που καταλαμβάνουν το μεγαλύτερο ποσοστό του υπολογισμού είναι η πρώτη και η τρίτη φάση, οι οποίες θα περιγραφθούν στη συνέχεια. Η δεύτερη, τέταρτη και πέμπτη φάση, καταλαμβάνουν πολύ μικρό ποσοστό του υπολογιστικού χρόνο, τόσο μικρό ώστε η παραλληλοποίησή τους θα επέφερε μη παρατηρήσιμο κέρδος, έως και πτώση στις επιδόσεις.

Η πρώτη φάση του αλγορίθμου αποτελείται από έναν βρόγχο με ένα σταθερό πλήθος επαναλήψεων, ενώ η τρίτη φάση από ένα άγνωστο αλλά μικρό πλήθος επαναλήψεων. Και οι δύο αυτές φάσεις περιλαμβάνουν περαιτέρω εμφωλευμένους βρόγχους, των οποίων την παραλληλοποίηση τη συζητούμε κατά την παρουσίαση των αποτελεσμάτων.

Αρχικά, κατά την εκκίνηση τόσο του τοπικού όσο και των απομακρυσμένων κόμβων, πρέπει η κάθε διεργασία να αρχικοποιηθεί κατάλληλα. Στο μοντέλο εκτέλεσης του HOMPI, κάθε κόμβος εκτελεί μία η παραπάνω διεργασίες, καθεμιά με τη δική της στοίβα. Χάρη σε αυτή τη δομή, αυτή η αρχικοποίηση δεν διαφέρει με τις άλλες

περιπτώσεις που έχουμε συναντήσει μέχρι τώρα, ανεξαρτήτως της πολυπλοκότητας των διαδικασιών που περιλαμβάνει η αρχικοποίηση σε κάθε κόμβο. Η εκτέλεση μίας κατάλληλης εργασίας σε κάθε διεργασία είναι αρκετή. Στο πέρας αυτής της διαδικασίας, όλες οι διεργασίες είναι έτοιμες να επεξεργασθούν εικόνες που θα τους αποσταλούν.

Η υλοποίησή μας αποτελείται από δύο επίπεδα παραλληλισμού. Το πρώτο επίπεδο θεωρεί ως task την ανάλυση μίας εικόνας. Μία τέτοια εργασία μπορεί να εκτελεσθεί σε οποιαδήποτε διεργασία του συστήματός μας, είτε τοπική, είτε απομακρυσμένη.

Το δεύτερο επίπεδο αποτελεί εμφωλευμένη παραλληλοποίηση της παραπάνω εργασίας. Σε καθεμιά από τις παραπάνω εργασίες, η εκτέλεση των πρώτων και δεύτερων φάσεων παραλληλοποιείται, διασπώντας τις επαναλήψεις των αντίστοιχων βρόγχων σε επιμέρους τμήματα, με καθένα από αυτά να εκτελείται υπό την μορφή ενός task.

Συνοψίζοντας λοιπόν, ο αρχικός κόμβος δημιουργεί μία εργασία για κάθε εικόνα της λίστας του, υπό τη μορφή ενός on-demand συστήματος το οποίο εκμεταλλεύεται πολλαπλούς κόμβους. Τα δεδομένα εισόδου κάθε task είναι μονάχα η διαδρομή του αρχείου στο κοινόχρηστο σύστημα αρχείων το οποίο αξιοποιεί η συστάδα μας. Σε καθεμιά από αυτές τις εργασίες, τα πιο χρονοβόρα τμήματα της εκτέλεσης παραλληλοποιούνται επίσης, με τη χρήση τοπικών πια tasks. Ο εμφωλευμένος παραλληλισμός αυτός εφαρμόστηκε ώστε να ευνοήσει την επεξεργασία μεγάλων εικόνων, καθώς και για να ελαττώσουμε τους χρόνους μη αξιοποίησης εργατών, στην εξής περίπτωση. Εάν ο κόμβος δεν έχει μία διαθέσιμη εργασία για κάθε εργάτη του, σημαίνει ότι κάποιος εργάτης θα παραμείνει αδρανής. Με τη χρήση εμφωλευμένου παραλληλισμού, τα νήματα-εργάτες που είναι αδρανή θα αναλάβουν φόρτο αυτής της παραλληλίας, ελαχιστοποιώντας έτσι τον χρόνο αδράνειάς τους. Κατά το πέρας κάθε task επεξεργασίας εικόνας, τα αποτελέσματα επιστρέφονται ασύγχρονα στον αρχικό κόμβο.

Κεφαλαιο 5

Πειραματικες εκτελεσεις και συμπερασματα

5.1 Πολλαπλασιασμός τετραγωνικών πινάκων

5.2 Τροποποιημένη Merge Sort

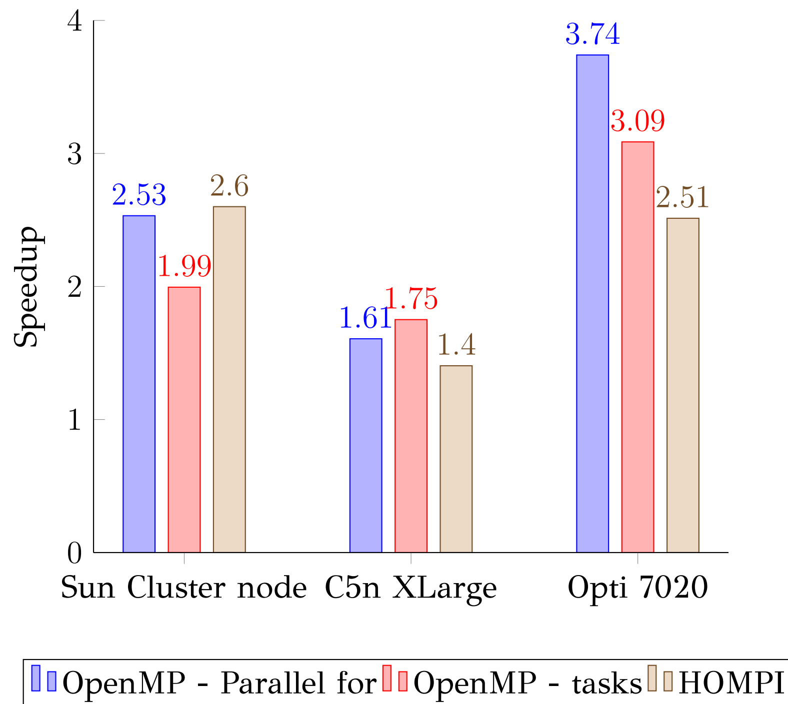
5.3 Bitonic Sort

5.4 Αναγνώριση προσώπων

5.5 Θεωρητική Ανάλυση κατανεμημένης εκτέλεσης

Στο κεφάλαιο αυτό θα παρουσιάσουμε τις επιδόσεις των εφαρμογών μας κατά τις πειραματικές εκτελέσεις σε διάφορα συστήματα. Στα πειράματα αυτά συμπεριλαμβάνουμε τις επιδόσεις των εφαρμογών μας σε περιβάλλον κοινόχρηστης μνήμης καθώς και κατανεμημένης μνήμης. Ως κατανεμημένα συστήματα χρησιμοποιήσαμε έναν cluster αποτελούμενο από 12 κόμβους τύπου Sun Fire X4100 και έναν cluster των AWS με 4 κόμβους τύπου C5n XLarge. Ως περιβάλλοντα κοινόχρηστης μνήμης χρησιμοποιήθηκαν οι κόμβοι αυτών των συστάδων, καθώς και άλλα τερματικά συμπεριλαμβανομένων τύπου Dell Opti7020 και ενός Raspberry Pi 4, ώστε να ελέγξουμε τις επιδόσεις σε μία αρχιτεκτονική διαφορετική της x64. Οι κόμβοι που χρησιμοποιήθηκαν καλύπτουν μεγάλο χρονικό διάστημα στα έτη παραγωγής τους (με άμεση συνέπεια στις επιδόσεις τους), καθώς και στη φύση τους, συμπεριλαμβάνοντας μία διαφορετική αρχιτεκτονική αλλά και εικονικούς κόμβους των υπηρεσιών

Σχήμα 5.1: Επιτάχυνση πολλαπλασιασμού πινάκων σε διαφορετικά συστήματα



EC2 των AWS.

5.1 Πολλαπλασιασμός τετραγωνικών πινάκων

Στο πρόβλημα αυτό ζητείται να εκτελεσθεί ο πολλαπλασιασμός δύο αλγεβρικών πινάκων. Ως είσοδος δίνονται δύο πίνακες ακεραίων αριθμών διαστάσεων $N \times N$ και η εφαρμογή υπολογίζει τον πίνακα που προκύπτει από τον πολλαπλασιασμό των δύο πινάκων εισόδου.

5.1.1 Συμπεριφορά σε περιβάλλον κοινόχρηστης μνήμης

Προτού γίνει μελέτη των αποτελεσμάτων στην κατανεμημένη εκτέλεση του προγράμματος, πρέπει να μελετήσουμε τη συμπεριφορά του παράλληλου προγράμματος σε περιβάλλον κοινόχρηστης μνήμης, διαφορετικά δεν θα μπορούμε να ερμηνεύσουμε ορθά τα αποτελέσματα της κατανεμημένης εκτέλεσης. Θα συγκρίνουμε την παραλληλοποίηση του προγράμματος με τη χρήση της υλοποίησης του προτύπου OpenMP του μεταγλωττιστή GCC, με την υβριδική έκδοση βασισμένη στο HOMPI. Όλα τα υπολογιστικά συστήματα διαθέτουν συνολικά τέσσερις πυρήνες ανά κόμβο (βλέπε Παράρτημα Β).

Από το σχήμα 5.1 μπορούμε να εξάγουμε πολλαπλά συμπεράσματα, τόσο ως προς τις τεχνικές που χρησιμοποιήθηκαν, όσο και για τα συστήματα στα οποία απευθυνόμαστε.

Στην περίπτωση του συστήματος τύπου Opti7020, η παραλληλοποίηση βρόγχων με χρήση του προτύπου OpenMP, εμφάνισε τις καλύτερες επιδόσεις ανάμεσα στα πειράματά μας, με επιτάχυνση 3.74, η οποία συνεπάγεται σχεδόν πλήρη αξιοποίηση όλων των υπολογιστικών μονάδων. Η πτώση των επιδόσεων σε σχέση με αυτόν τον σχεδόν ιδανικό αριθμό σε όλες τις άλλες περιπτώσεις μπορεί να αποδοθεί εν μέρη, εάν όχι πλήρως, στην τροποποίηση του προγράμματος.

Στη σειριακή έκδοση του προγράμματος, η συμπεριφορά του αλγορίθμου έχει ως μεγάλο πλεονέκτημα την πρόσβαση σε γειτονικές θέσεις μνήμης, καθώς επεξεργαζόμαστε τρεις πίνακες, γραμμή ανά γραμμή. Ένας από τους λόγους που επιλέχθηκε το σχετικά μικρό μέγεθος εισόδου, είναι ώστε ένας επεξεργαστής με αρκετά μεγάλη μνήμη cache να μπορεί ενδεχομένως να χωρέσει μεγάλα τμήματα των πινάκων αυτών, επιτυγχάνοντας έτσι αρκετές επιτυχημένες αναζητήσεις στη μνήμη αυτήν, τόσο στη σειριακή, όσο και στην παράλληλη εκδοχή του προγράμματος. Στην περίπτωση των παράλληλων όμως προγραμμάτων, το αν θα μπορεί να γίνει χρήση της μνήμης αυτής, θα εξαρτάται από δύο παράγοντες. Ο ένας είναι, όπως αναφέραμε, το μέγεθος της μνήμης αυτής.

Το μοντέλο επεξεργαστή i5-4590 στο σύστημα Opti7020 περιλαμβάνει 6MB μνήμης κοινόχρηστης κρυφής μνήμης. Εφόσον κάθε πίνακας μας έχει μέγεθος $1024 \times 1024 \times s_{int}$, με s_int ίσο με το πλήθος των bytes ανά ακέραιο αριθμό, η κρυφή μνήμη μπορεί να αποθηκεύσει σημαντικό ποσοστό των πινάκων εισόδου και εξόδου. Εφό-

σον επίσης το επίπεδο κρυφής μνήμης που περιγράψαμε είναι κοινόχρηστο μεταξύ των επεξεργαστικών μονάδων, είναι αρκετά πιθανό το σενάριο όπου κάποιο νήμα θα ζητήσει μία διεύθυνση μνήμης η οποία βρίσκεται ήδη στην κρυφή μνήμη. Η πιθανότητα αυτή είναι αυξημένη όταν εκτελούμε παραλληλοποίηση βρόγχων, όπου κάθε νήμα αναλαμβάνει ένα ίσο πλήθος επαναλήψεων, άρα τα τμήματα που αναλαμβάνονται αφορούν συνεχείς θέσεις των πινάκων εισόδου.

Ο άλλος παράγοντας είναι η υλοποίηση του παράλληλου αλγορίθμου. Η τροποποιημένη έκδοση του αλγορίθμου, έπειτα από κάθε επανάληψη του εσωτερικού βρόχου γράφει το αποτέλεσμα αυτό σε έναν διαφορετικό πίνακα. Ο πίνακας αυτός είναι ένας πίνακας αποτελεσμάτων, τον οποίο διαχειρίζεται αυτόματα η βιβλιοθήκη TORC, κατά τον χρόνο εκτέλεσης. Είναι πιθανό αυτός ο δεσμευμένος πίνακας να βρίσκεται σε χώρο μνήμης σε αρκετά απομακρυσμένες θέσεις από αυτές των πινάκων εισόδου που επεξεργαζόμαστε. Αλλά πιο σημαντικό είναι το γεγονός ότι κάθε τέτοια εργασία η οποία εκτελείται σε διαφορετικό νήμα-εργάτη, χρησιμοποιεί τον δικό της πίνακα αποτελεσμάτων. Με άλλα λόγια, αντί για έναν κοινό πίνακα αποτελεσμάτων, σε κάθε δεδομένη στιγμή εκτελούνται προσβάσεις μνήμης σε τόσους πίνακες όσους είναι οι οντότητες εκτέλεσης που είναι ενεργές τη χρονική στιγμή αυτή. Και όπως αναφέραμε, αυτοί οι πίνακες μπορεί να βρίσκονται σε αυθαίρετα μακρινές μεταξύ τους θέσεις μνήμης. Ως συνέπεια αυτού, κάθε φορά που επιθυμούμε να προσπελάσουμε τον πίνακα αποτελεσμάτων, ένα τμήμα των πινάκων εισόδου και εξόδου που βρίσκεται στην κρυφή μνήμη θα αντικατασταθεί από αυτόν τον νέο πίνακα, εάν αυτός δεν βρεθεί στην κρυφή μνήμη. Επίσης, τα νήματα εργάτες καθώς και άλλοι μηχανισμοί ελέγχου της βιβλιοθήκης εκτελούν διάφορες προσβάσεις μνήμης, οι οποίες είναι πιθανό να εκτελούν προσβάσεις μνήμης, σε αντίστοιχα απομακρυσμένες διευθύνσεις. Τελευταίο αλλά όχι λιγότερο σημαντικό είναι το γεγονός ότι κατά το πέρας μίας εργασίας, είτε αυτή εκτελέσθηκε σε απομακρυσμένο κόμβο είτε στον αρχικό κόμβο, εκτελείται ένα callback, το οποίο όχι μόνο προσθέτει ένα επιπλέον υπολογιστικό κόστος ίσο με (N^2), αλλά ενισχύει περαιτέρω τα προβλήματα που δημιουργούμε στην κρυφή μνήμη.

Περιγράψαμε λοιπόν πως ένα σειριακό πρόγραμμα, αλλά και ένα παράλληλο πρόγραμμα που τρέχει σε έναν επεξεργαστή με μεγάλο μέγεθος κρυφής μνήμης μπορεί να έχει το προτέρημα της κρυφής μνήμης. Δυστυχώς αυτό το προτέρημα δεν εμφανίζεται στα άλλα συστήματα που δοκιμάσθηκαν.

Ο επεξεργαστής Opteron 275 των κόμβων της συστάδας Sun, κατέχει μονάχα

1MB κοινόχρηστης κρυφής μνήμης δευτέρου επιπέδου. Αυτό σημαίνει ότι η σειριακή εκδοχή του προγράμματος θα έχει κάποιο αρκετά μικρότερο κέρδος από αυτό του i5-4590 χάρη στην κρυφή μνήμη του, αλλά πιο σημαντικά, οποιαδήποτε από τις παράλληλες εκδοχές του προγράμματος θα μεγεθύνουν τα προβλήματα που περιγράψαμε παραπάνω. Αυτό θα οδηγήσει στις αστοχίες κρυφής μνήμης που περιγράψαμε, θεωρία η οποία επαληθεύεται στο σχήμα 5.1.

Η θεωρία αυτή φαίνεται επίσης στην περίπτωση του εικονικού κόμβου στην περίπτωση των υπηρεσιών EC2, όπου η μέγιστη επιτάχυνση που επιτεύχθηκε ήταν του μεγέθους 1.75. Στην περίπτωση αυτή η ελάχιστη επιτάχυνση πιθανώς ευθύνεται για το γεγονός ότι μιλάμε πια για εικονικά συστήματα, όπου οι κόμβοι αυτοί προσομοιώνονται υπό τη μορφή Εικονικών Μηχανών (Virtual Machines), όπου δεν έχουμε άμεση πρόσβαση στο υλικό στο οποίο εκτελούμε το πρόγραμμά μας, ενώ δεν έχουμε επίσης κέρδος από κοινόχρηστη cache μεταξύ δύο ή παραπάνω πυρήνων στους οποίους εκτελούνται τα νήματά μας.

Επιπλέον παρατηρήσεις

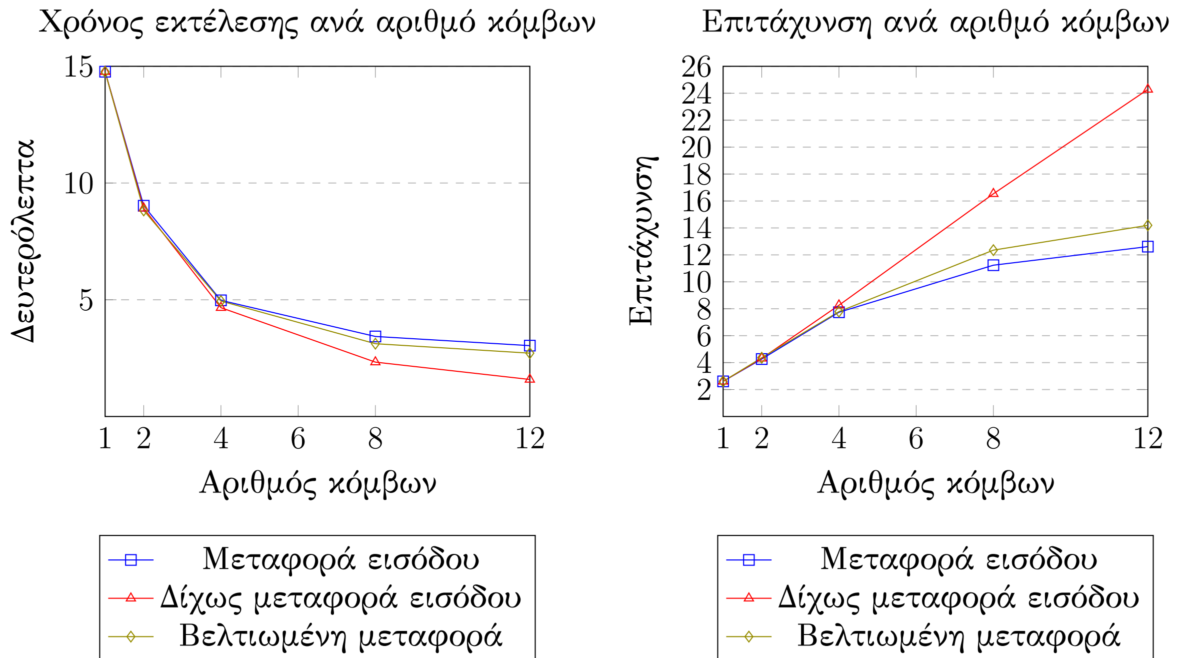
Στη συνέχεια θα παρουσιάσουμε δύο παρατηρήσεις, τις οποίες θεωρούμε άκρως σημαντικές για τη μελέτη μας και πιστεύουμε ότι δίχως αυτές η παραπάνω μελέτη και τα συμπεράσματά της είναι ελλειπή, έως και λανθασμένα. Αρχικά, πρέπει να τονίσουμε ότι οι πτώσεις των επιδόσεων, ενώ ευθύνονται στην παράλληλη έκδοση του προγράμματος, δεν καθιστούν από μόνες τους μειονέκτημα. Αντίθετα, είναι ένα μειονέκτημα σε σχέση με τη σειριακή έκδοση του προγράμματος, από το οποίο έχουμε πια αφαιρέσει το προτέρημα που του παρείχαν οι επιτυχημένες αναζητήσεις στην κρυφή μνήμη. Κατά συνέπεια, εάν η σειριακή έκδοση του προγράμματος δεν είχε ούτε αυτή το πλεονέκτημα αυτό, πχ. εάν είχαμε πολύ μεγαλύτερα μεγέθη εισόδου, το σειριακό πρόγραμμα θα υπέφερε από το ίδιο πρόβλημα με την παράλληλη έκδοση. Εάν οι διαστάσεις των πινάκων μας μεγάλωναν σημαντικά στην τάξη των δεκάδων χιλιάδων στοιχείων ανά γραμμή, κάτι το οποίο υποστηρίζεται πλήρως από τον αλγόριθμό μας, οποιεσδήποτε γειτονικές προσβάσεις στη μνήμη θα προκαλούσαν αποτυχημένη αναζήτηση στην κρυφή μνήμη.

Η επόμενη παρατήρηση αφορά την εκτέλεση πολλαπλών διεργασιών (batch processing), η οποία θα περιγραφεί στη συνέχεια του κεφαλαίου. Ας θεωρήσουμε ότι θέλουμε να εκτελέσουμε τον αλγόριθμό μας για τέσσερις διαφορετικές εισόδους. Για λόγους απλότητας, στο παράδειγμά μας θα θεωρήσουμε ένα περιβάλλον κοινό-

χρηστης μνήμης, δηλαδή έναν μονάχα διαθέσιμο επεξεργαστικό κόμβο. Έχουμε δύο επιλογές: η πρώτη είναι να εκτελέσουμε τέσσερις φορές την παράλληλη έκδοση του προγράμματος, ενώ η άλλη είναι να εκτελέσουμε ταυτόχρονα τέσσερις διεργασίες της σειριακής εκδοχής του αλγορίθμου, καθεμιά σε διαφορετική είσοδο.

Στην πρώτη λύση, η παραδοχή ότι θα έχουμε όμοιους χρόνους εκτέλεσης και στις τέσσερις επαναλήψεις του αλγορίθμου, είναι σωστή. Όμοια, κάποιος θα μπορούσε να πει ότι καθεμιά από τις τέσσερις διεργασίες της δεύτερης λύσης θα χρειαζόταν όσο χρόνο παρουσιάσαμε στην εκτέλεση της παράλληλης εκδοχής του αλγορίθμου. Αυτή η παραδοχή είναι λανθασμένη, στα συστήματα τα οποία δείξαμε ότι η παράλληλη εκδοχή του αλγορίθμου προκάλεσε αποτυχίες στην κρυφή μνήμη. Σε αυτά τα συστήματα, οι διεργασίες υποφέρουν από το ίδιο πρόβλημα, καθώς οι φυσικοί επεξεργαστές μοιράζονται την κρυφή μνήμη δευτέρου επιπέδου, η οποία παρείχε το πλεονέκτημα στον σειριακό αλγόριθμο. Είτε λοιπόν η κρυφή μνήμη στην οποία αναζητεί τα δεδομένα του ένα νήμα χρησιμοποιείται από ένα διαφορετικό νήμα της ίδιας διεργασίας, είτε από ένα νήμα μίας διαφορετικής διεργασίας, τα αποτελέσματα είναι παρόμοια. Η επιβράδυνση του σειριακού προγράμματος παρατηρήθηκε στα συστήματα με μικρότερα μεγέθη κρυφής μνήμης και όχι σε αυτά που παρατηρούσαμε υψηλή αξιοποίηση των πολλαπλών επεξεργαστών. Στην πρώτη περίπτωση, η επιβράδυνση δεν ήταν όση αυτήν που παρουσιάσαμε στην παράλληλη εκδοχή του αλγορίθμου, με τον έναν λόγο να είναι η χρήση ενός πίνακα αποτελεσμάτων του χρήστη στη σειριακή εκδοχή, έναντι του επιπλέον πίνακα της βιβλιοθήκης χρόνου εκτέλεσης και της διαδικασίας callback την οποία εκτελεί η κατανεμημένη εκδοχή του αλγορίθμου.

Σχήμα 5.2: Επιδόσεις πολλαπλασιασμού πινάκων στον Sun Cluster



5.1.2 Sun Cluster

Κατά τον κατανεμημένο πολλαπλασιασμό πινάκων, επιλέχθηκαν ως είσοδος σχετικά μικροί πίνακες, διαστάσεων 1024×1024. Αυτή η επιλογή έγινε ώστε να μπορούμε να παρατηρήσουμε τις επιπτώσεις των μεταφορών δεδομένων στις επιδόσεις, συγκεκριμένα σε ένα σενάριο όπου ο χρόνος υπολογισμών δεν είμαι μεγάλος, άρα οι μεταφορές δεδομένων θα αποτελέσουν σημαντικό ποσοστό του χρόνου εκτέλεσης. Επίσης, χάρη σε αυτό το μικρό σχετικά μέγεθος θα φανούν κάποια μειονεκτήματα τα οποία πρέπει να έχουμε υπόψιν κατά την ανάπτυξη παράλληλων και υβριδικών προγράμματος, καθώς και κατά την τροποποίηση υπαρχόντων προγραμμάτων.

Αρχικά, θα μελετήσουμε τις επιπτώσεις των μεταφορών δεδομένων στις επιδόσεις. Στο σχήμα 5.2, στην περίπτωση της μεταφοράς εισόδου έχουμε εφαρμόσει την αρχική διαδικασία αρχικοποίησης, κατά την οποία ο συντονιστής περιμένει να τερματίσουν οι εργασίες αρχικοποίησης σε όλους τους απομακρυσμένους κόμβους. Στην περίπτωση της βελτιωμένης μεταφοράς δεδομένων εισόδου, χρησιμοποιήθηκε η τεχνική που περιγράφθηκε στην ενότητα 4.2. Στην περίπτωση της μη ύπαρξης μεταφοράς δεδομένων εισόδου, η διαδικασία της αρχικοποίησης δεν έχει συμπεριληφθεί στη χρονομέτρηση των πειραμάτων. Αξίζει να σημειωθεί ότι αυτό το σενάριο δεν είναι ρεαλιστικό, καθώς στις τρέχουσες υλοποιήσεις η αρχικοποίηση των απομακρυσμένων διεργασιών γίνεται μέσω δημιουργίας εργασιών και οι οποίες δημιουργίες

συνεπάγονται μεταφορές δεδομένων, ακόμα και στην περίπτωση όπου δεν υπάρχει μεταφορά της εισόδου. Συνεπώς, αυτό το σενάριο χρησιμοποιήθηκε για σκοπούς ανάλυσης.

Βάση του σχήματος 5.2, φαίνεται να ισχύει ότι στην περίπτωση όπου δεν υπάρχει μεταφορά των δεδομένων εισόδου, έχουμε πρακτικά γραμμική επιτάχυνση ως προς τον αριθμό των κόμβων. Αυτή η γραμμικότητα σημαίνει ότι η αξιοποίηση των νέων επεξεργαστικών μονάδων παραμένει στο ίδιο περίπου ποσοστό, ανεξαρτήτως του αριθμού των κόμβων που χρησιμοποιούμε. Στις πιο ρεαλιστικές περιπτώσεις όπου συμπεριλαμβάνουμε τη μεταφορά δεδομένων, παρατηρούμε ελάττωση της αποδοτικότητας, όσο αυξάνεται ο αριθμός των κόμβων, με μία επίσης αναμενόμενη βελτίωση χάρη στη νέα τεχνική αρχικοποίησης.

Η μείωση της αποδοτικότητας είναι ένα αναμενόμενο φαινόμενο και ευθύνεται σε αρκετούς παράγοντες. Όσο αυξάνεται ο αριθμός των απομακρυσμένων διεργασιών, αυξάνεται ανάλογα το πλήθος των απομακρυσμένων συνδέσεων που πρέπει να διαχειριστεί ο συντονιστής, ο οποίος κατέχει μονάχα ένα νήμα εκτέλεσης το οποίο είναι υπεύθυνο για τη διαχείριση της μεταφοράς των δεδομένων.

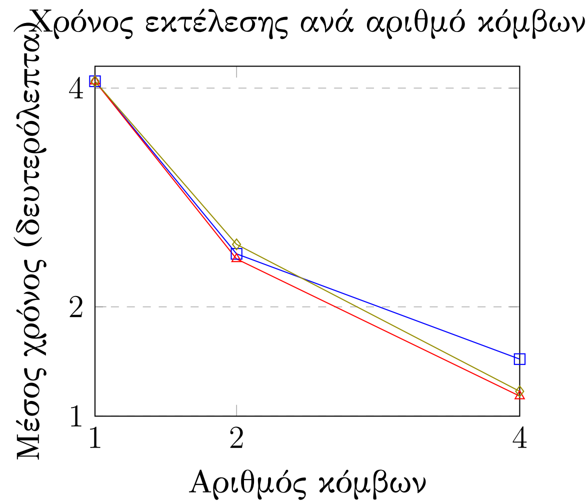
Οι μεταφορές δεδομένων αυτές περιλαμβάνουν την αποστολή των εργασιών και τη λήψη των αποτελεσμάτων κατά το πέρας μίας εργασίας. Επίσης, όσο αυξάνουμε το πλήθος των κόμβων, πρέπει να εκτελέσουμε όλο και πιο λεπτόκοκκο παραλληλισμό, έτσι ώστε να είμαστε βέβαιοι ότι οι απομακρυσμένοι κόμβοι θα έχουν σε κάθε δεδομένη χρονική στιγμή φόρτο εργασίας να εκτελέσουν. Δηλαδή, πρέπει να διαιρέσουμε το αρχικό μας πρόβλημα σε όλο και περισσότερα και μικρότερα υποπροβλήματα, άρα εργασίες. Αυτές οι επιπλέον εργασίες συνεπάγονται επιπλέον υπολογιστικό κόστος. Παρόλο που ο συνολικός όγκος των αποτελεσμάτων που θα μεταφερθεί παραμένει ίδιος χάριν στη φύση του προβλήματος, αυτό δεν σημαίνει ότι το συνολικό κόστος της μεταφοράς των δεδομένων αυτών θα παραμείνει ίδιο. Η εκτέλεση μίας μεταφοράς μεγάλου όγκου δεδομένων μέσω του δικτύου είναι πιο αποδοτική από τη διάσπαση του όγκου αυτού σε μικρότερα τμήματα και την αποστολή αυτών υπό τη μορφή πολλών μεταφορών. Η αποστολή αιτημάτων δικτύου σημαίνει επιπλέον δεδομένα προς αποστολή και όπως είναι γνωστό λόγω του χρόνου διάδοσης στα δίκτυα επικοινωνιών, δεν είναι προς όφελος μας η διάσπαση μεγάλου όγκου δεδομένων σε επιμέρους μικρότερα τμήματα. Αυτό το επιπλέον υπολογιστικό κόστος καταλαμβάνει χρόνο του επεξεργαστή του συντονιστή, ο οποίος θα χρησιμοποιούταν διαφορετικά για την εκτέλεση εργασιών.

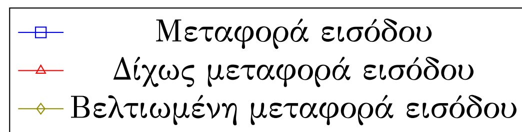
Επιπλέον, είναι πιθανό όταν φτάνουμε σε μεγάλο αριθμό κόμβων, να εμφανισθούν άλλα επιπλέον προβλήματα στο νήμα του διακομιστή. Μερικά από αυτά είναι η έλλειψη μνήμης για την αποθήκευση των δεδομένων κατά τις μεταφορές μέσω του δικτύου, ή η έλλειψη επεξεργαστικού χρόνου για τη διαχείριση όλων αυτών των διαδικασιών μεταφορών. Τα προβλήματα αυτά μπορεί να έχουν ως αποτέλεσμα την αναμονή απάντησης από τον συντονιστή σε κάθε απομακρυσμένο κόμβο, σπαταλώντας έτσι πολύτιμο υπολογιστικό χρόνο.

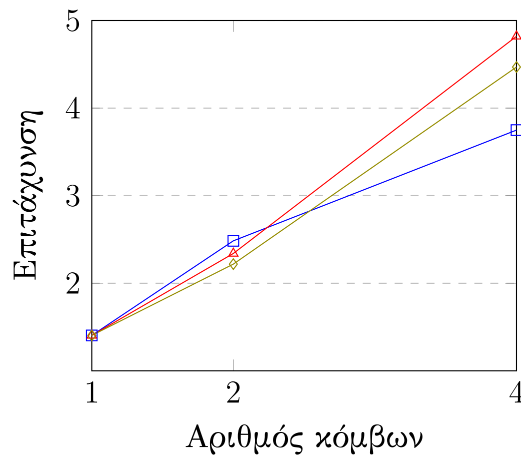
Συνοψίζοντας, για να πετύχουμε την πλήρη υβριδική παραλληλοποίηση του προγράμματός μας με τα υπάρχοντα εργαλεία, μειώσαμε την αποδοτικότητα σε κάθε κόμβο. Άρα γνωρίζουμε πια την μέγιστη επιτάχυνση που μπορούμε να πετύχουμε ανάλογα με το πλήθος τον κόμβων στην κατανεμημένη εκδοχή του προγράμματος. Εάν η επιτάχυνση που μπορούμε να επιτύχουμε σε κάθε κόμβο είναι ίση με $speedup$, η μέγιστη επιτάχυνση για N κόμβους θα είναι ίση με $speeup \times N$. Αυτή η χαμηλή επιτάχυνση ανά κόμβο σε σχέση με τις εναλλακτικές υλοποιήσεις κοινόχρηστης μνήμης του αλγορίθμου είναι το απαραίτητο τίμημα για την αξιοποίηση πολλαπλών απομακρυσμένων κόμβων.

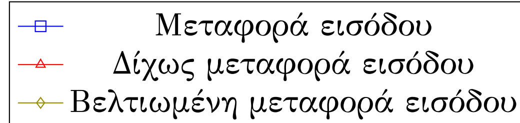
5.1.3 EC2 Cluster

Σχήμα 5.3: Επιδόσεις πολλαπλασιασμού πινάκων στον EC2 Cluster





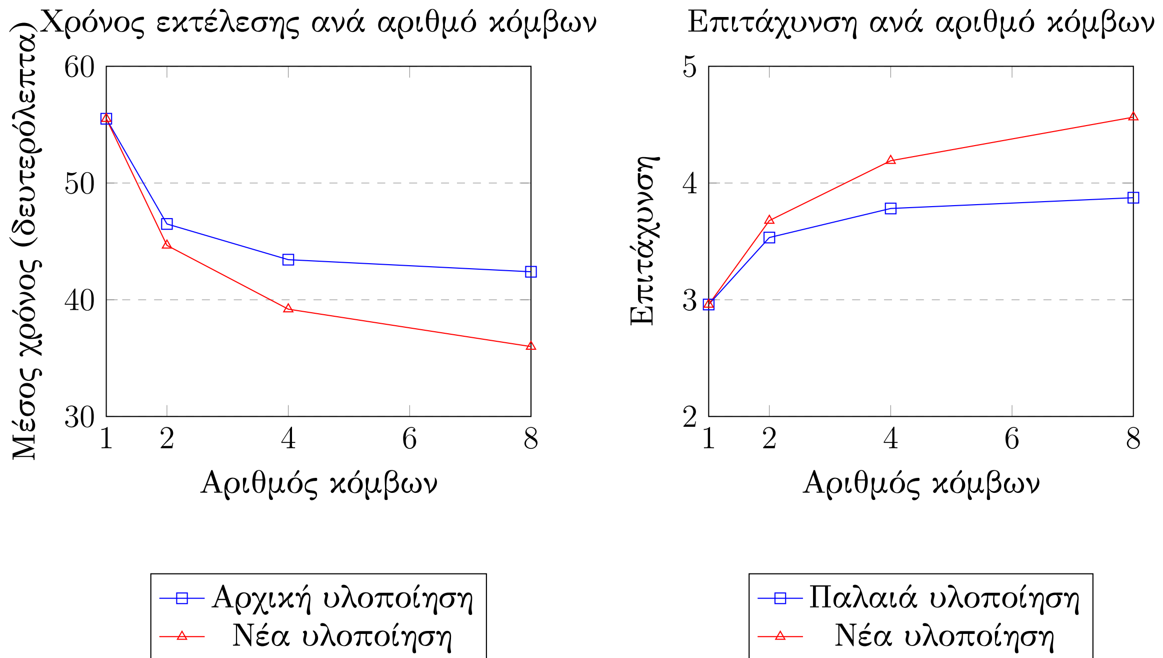




Μελετώντας τον χρόνο εκτέλεσης καθώς και την επιτάχυνση ανά μέθοδο αρχικοποίησης, μπορούμε να παρατηρήσουμε τα εξής.

Οι περιπτώσεις δίχως χρονομέτρηση της μεταφοράς δεδομένων και με βελτιωμένη μεταφορά έχουν πλεονέκτημα έναντι της άλλης περίπτωσης, ενώ η βελτιωμένη μεταφορά παρουσιάζει χρόνο εκτέλεσης αρκετά κοντά σε αυτόν δίχως μεταφορά. Από αυτό το φαινόμενο μπορούμε να συμπεράνουμε ότι η πιο χρονοβόρα διαδικασία κατά τη φάση της αρχικοποίησης είναι η πράξη του συγχρονισμού, την οποία

Σχήμα 5.4: Επιδόσεις Merge Sort στον Sun Cluster



έχουμε αφαιρέσει στη βελτιωμένη μας λύση. Ως αποτέλεσμα, το επιπλέον χρονικό κόστος της αρχικοποίησης έχει γίνει αμελητέο χάριν στην ιδέα αυτήν.

Η εφαρμογή φαίνεται να παρουσιάζει γραμμική επιτάχυνση ως προς το πλήθος των κόμβων που χρησιμοποιούνται. Η μέγιστη επιτάχυνση που επιτεύχθηκε (συμπεριλαμβάνοντας τις μεταφορές δεδομένων στους χρόνους εκτέλεσης) ήταν ίση με 4.65. Δεδομένου του γεγονότος ότι η μέγιστη επιτάχυνση αυτού του προγράμματος σε περιβάλλον κοινόχρηστης μνήμης ήταν ίση με 1.40, αυτό σημαίνει ότι η μέγιστη δυνατή επιτάχυνση με 4 κόμβους θα ήταν ίση με $1.40 \times 4 = 5.6$. Δεδομένων των μεταφορών που περιλαμβάνονται στον χρόνο εκτέλεσης, η επιτάχυνση που επιτεύχθηκε είναι ικανοποιητική.

Τέλος, θα σημειώσουμε ότι στην περίπτωση των τεσσάρων κόμβων, η εφαρμογή είχε βελτιωμένες επιδόσεις όταν το μέγεθος του υποπίνακα ήταν ίσο με 64×64, έναντι 32×32.

5.2 Τροποποιημένη Merge Sort

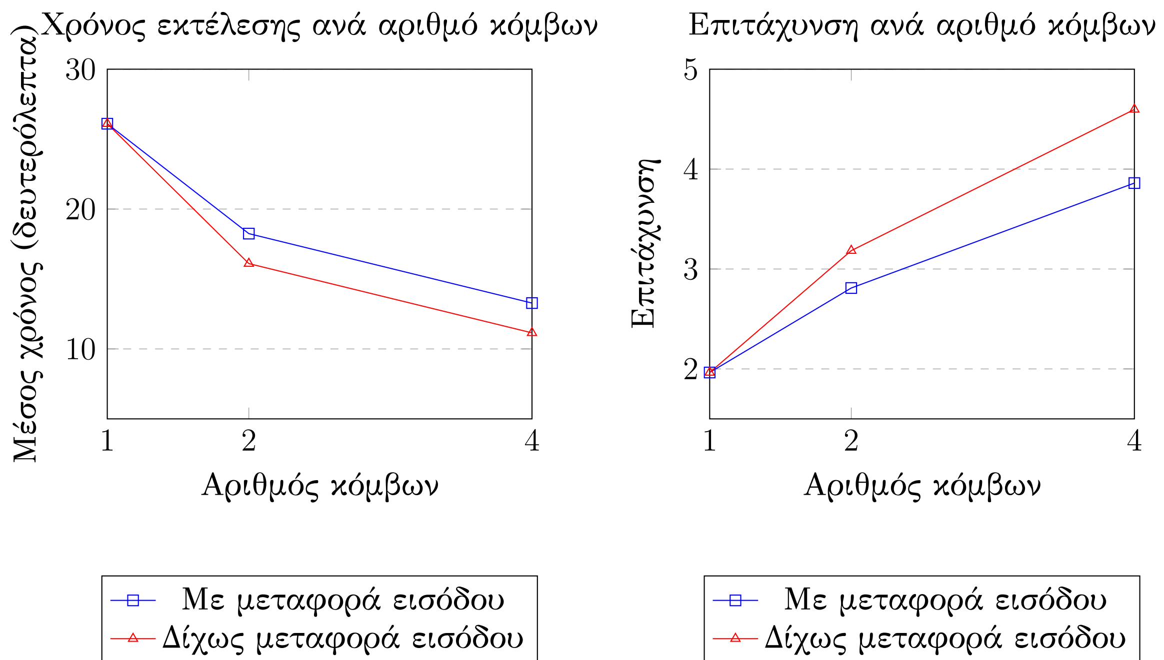
5.2.1 Sun Cluster

Για τα πειράματα αυτής της εφαρμογής επιλέχθηκε ως είσοδος πίνακας ακεραίων με 2^{28} στοιχεία. Αυτή η επιλογή έγινε ώστε να μελετήσουμε τις επιδόσεις στην περίπτωση όπου αποστέλλουμε αρκετά μεγάλους όγκους δεδομένων για επεξεργασία σε απομακρυσμένους κόμβους.

Στην εκτέλεση σε περιβάλλον κοινόχρηστης μνήμης, η επιτάχυνση που επιτεύχθηκε είχε τιμή 2.95. Και στις δύο υλοποιήσεις μας, όσο αυξάνεται το πλήθος των κόμβων, η αποδοτικότητα μειώνεται. Η εκτέλεση αυτή δεν επιφέρει τόσο καλά αποτελέσματα όσο θα θέλαμε, αλλά οι επιδόσεις αυτές είναι αναμενόμενες δεδομένου του συμβατικού δικτύου που χρησιμοποιήθηκε.

Στην εφαρμογή αυτή όμως δείχνουμε ότι το πρόβλημά μας δημιουργείται από τη μεταφορά του μεγάλου όγκου δεδομένων. Αυτό φαίνεται στη βελτιωμένη μας υλοποίηση, η οποία επέφερε κέρδος μέχρι 15% έναντι της αρχικής, λόγω της αφαίρεσης των μη απαραίτητων μεταφορών στο δίκτυο.

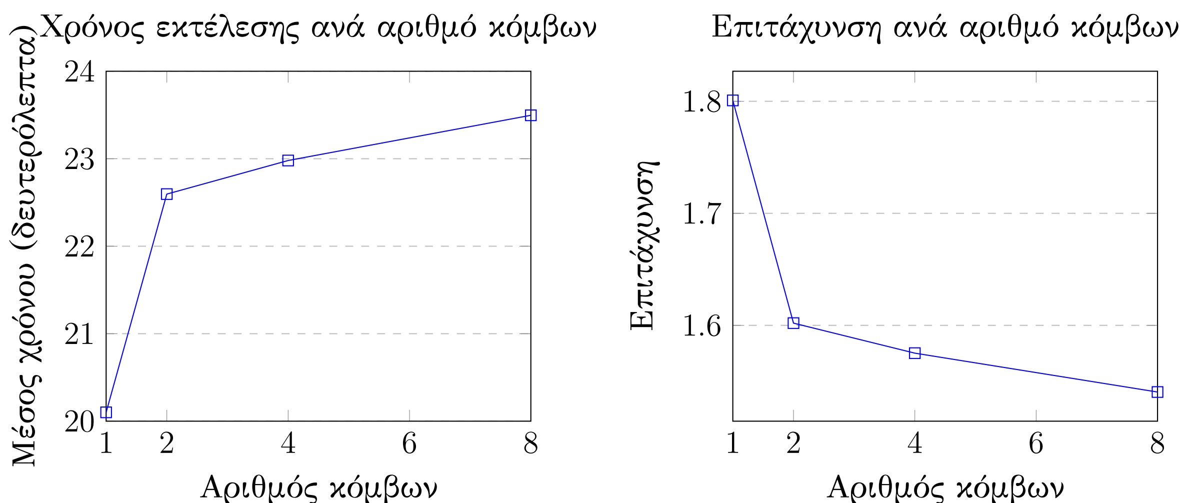
Σχήμα 5.5: Επιδόσεις Merge Sort στον EC2 Cluster



5.2.2 EC2 Cluster

Στην περίπτωση του εικονικού μας cluster, η εκτέλεση κοινόχρηστης μνήμης έφερε επιτάχυνση 1.94. Παρά τη μειωμένη αποδοτικότητα σε σχέση με τον φυσικό μας cluster, το πείραμα αυτό εμφανίζει μικρότερο λόγο μείωσης της αποδοτικότητας όσο αυξάνουμε το πλήθος των κόμβων. Παρά τη σημαντική βελτίωση στο δίκτυο σε αυτή την περίπτωση, παρατηρούμε ξανά ότι οι μεταφορές έχουν σημαντική επίπτωση στις επιδόσεις. Επίσης πρέπει να σημειώσουμε ότι σημαντικός παράγοντα για τη χαμηλή αποδοτικότητα του συστήματος αποτελεί η χαμηλή επιτάχυνση που επιτυγχάνεται σε επίπεδο κόμβου, για κάθε task που ανατίθεται σε αυτόν, όπως παρατηρήθηκε σε όλες τις πειραματικές μας εκτελέσεις στους εικονικούς κόμβους αυτούς.

Σχήμα 5.6: Επιδόσεις Bitonic Sort στον Sun Cluster

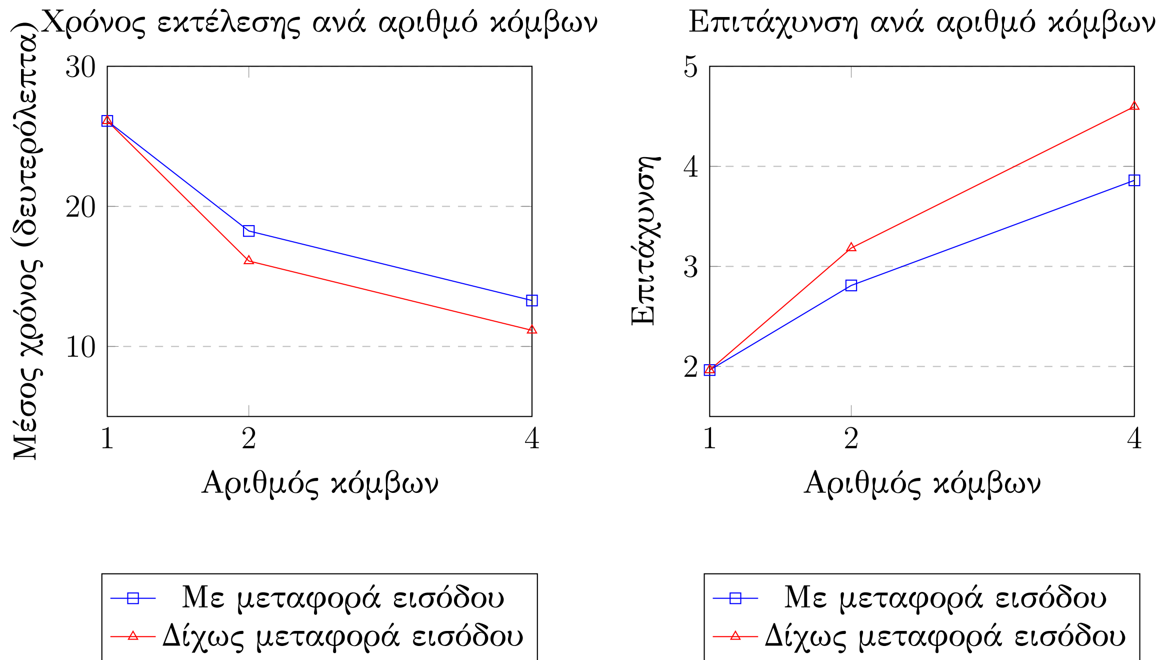


5.3 Bitonic Sort

Στην εφαρμογή αυτήν χρησιμοποιούμε μικρότερο μέγεθος εισόδου (2^{25} στοιχεία), έτσι ώστε να μελετήσουμε τις επιδόσεις σε δύο διαφορετικές υλοποιήσεις και μεγέθους εισόδου του ίδιου προβλήματος. Επιπλέον, λόγω της απλότητας της δεύτερης φάσης, η εκτέλεσή της γίνεται στον τοπικό κόμβο. Η εφαρμογή αυτή εμφανίζει χαμηλή αξιοποίηση των πολλαπλών πυρήνων σε επίπεδο κόμβου. Για αυτόν τον λόγο, ο χρόνος εκτέλεσης σε απομακρυσμένους κόμβους δεν υπερνικά σημαντικά τους χρόνους μεταφορών καθώς και τα επιπλέον κόστη της διαδικασίας αυτής, με αποτέλεσμα τη μείωση των επιδόσεων στην περίπτωση του Sun Cluster και τη χαμηλή αποδοτικότητα στον EC2 Cluster. Η ανάλυση αυτών των αιτίων θα συνεχισθεί στο τέλος του κεφαλαίου.

Στην περίπτωση του εικονικού μας cluster, η εκτέλεση κοινόχρηστης μνήμης έφερε επιτάχυνση 1.94. Παρά τη μειωμένη αποδοτικότητα σε σχέση με τον φυσικό μας cluster, το πείραμα αυτό εμφανίζει μικρότερο λόγο μείωσης της αποδοτικότητας όσο αυξάνουμε το πλήθος των κόμβων. Παρά την σημαντική βελτίωση στο δίκτυο σε αυτή την περίπτωση, παρατηρούμε ξανά ότι οι μεταφορές έχουν σημαντική επίπτωση στις επιδόσεις. Επίσης πρέπει να σημειώσουμε ότι σημαντικός παράγοντα για τη χαμηλή αποδοτικότητα του συστήματος αποτελεί η χαμηλή επιτάχυνση που επιτυγχάνεται σε επίπεδο κόμβου, για κάθε task που ανατίθεται σε αυτόν, όπως παρατηρήθηκε σε όλες τις πειραματικές μας εκτελέσεις στους εικονικούς κόμβους αυτούς.

Σχήμα 5.7: Επιδόσεις Bitonic Sort στον EC2 Cluster

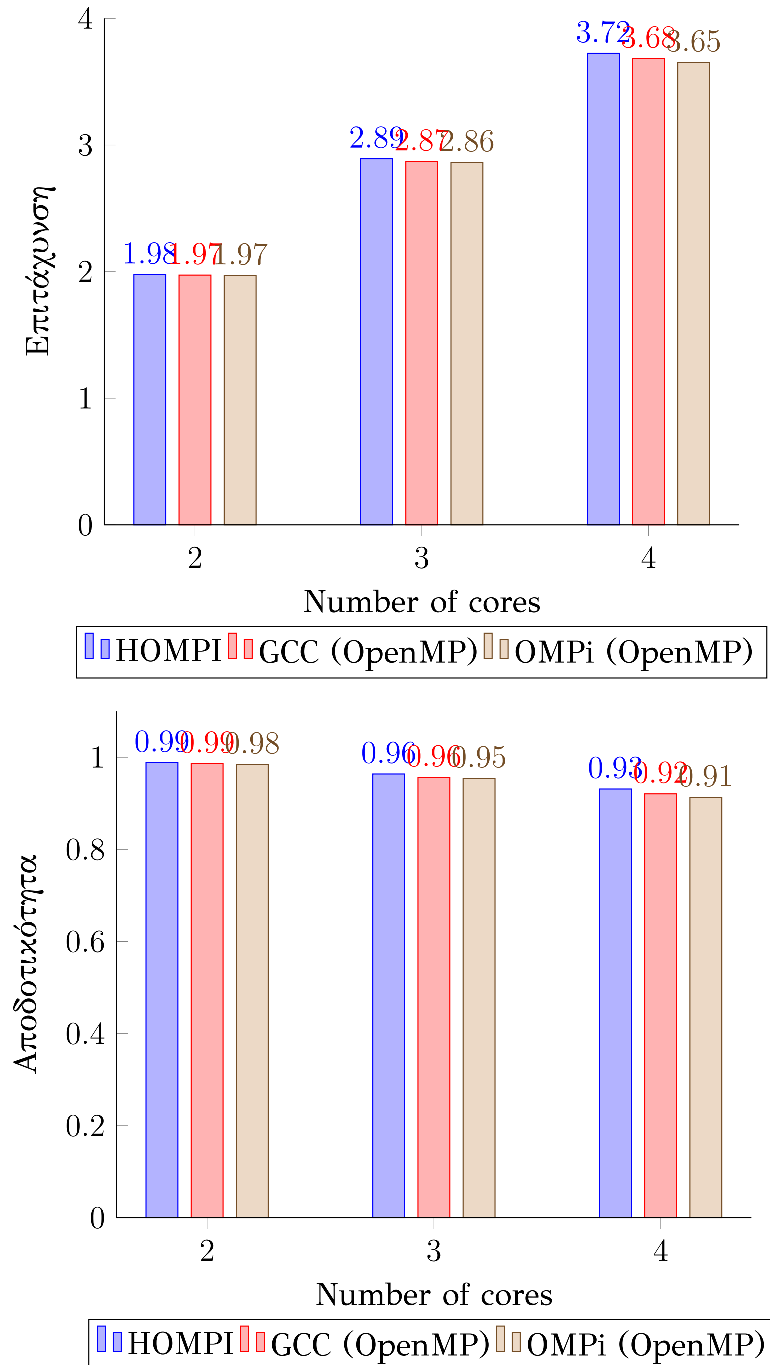


5.4 Αναγνώριση προσώπων

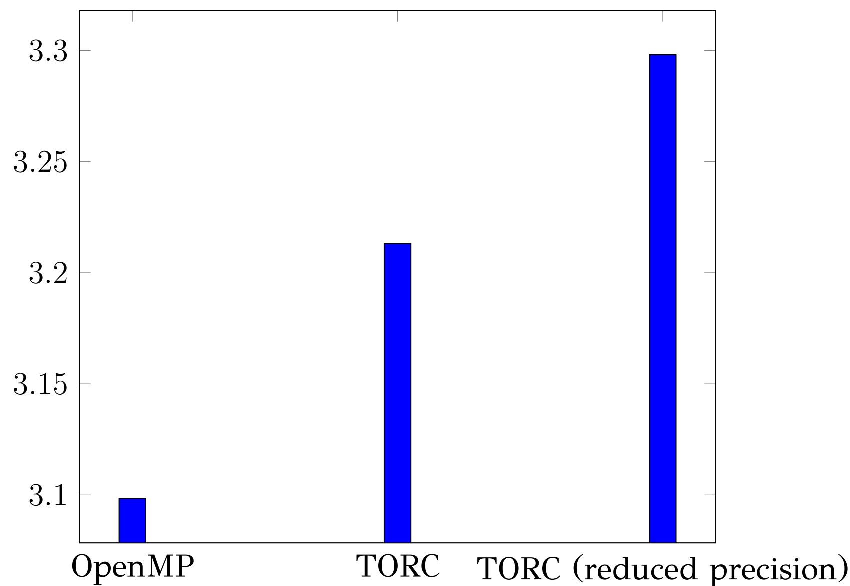
5.4.1 Επιδόσεις κοινόχρηστης μνήμης

Raspberry Pi 4

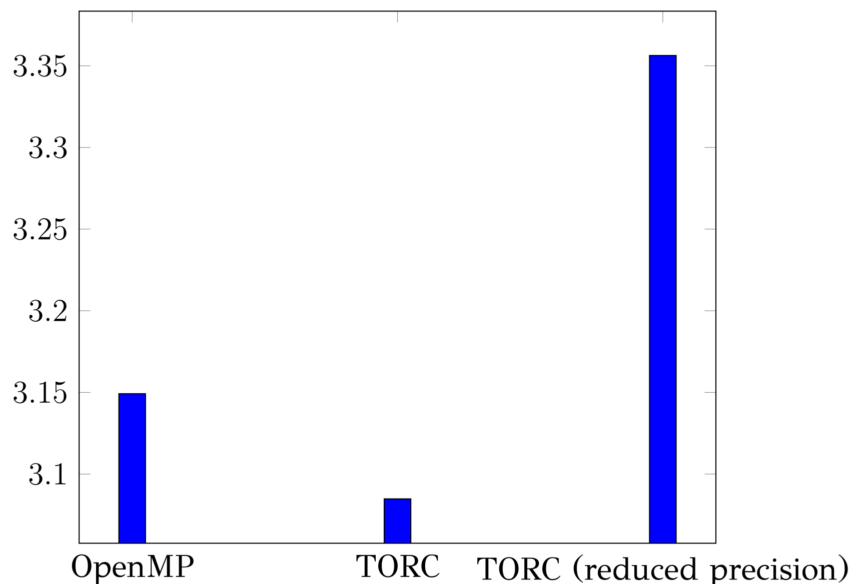
Σχήμα 5.8: Επιδόσεις σε Batch processing 160 εικόνων



Σχήμα 5.9: Επιτάχυνση στην εικόνα ”Class57”



Σχήμα 5.10: Επιτάχυνση στην εικόνα ”The many faces of Anthony Hopkins”



Ως σύστημα για τη σύγκριση μεθόδων παραλληλοποίησης επιλέξαμε το Raspberry Pi 4, ώστε να μελετήσουμε τον παραλληλισμό σε μία διαφορετική αρχιτεκτονική, καθώς και τη φορητότητα του HOMPI framework. Τα αποτελέσματα που παρουσιάζουμε είναι όμοια και στα άλλα συστήματα που δοκιμάσαμε. Μελετώντας το σχήμα 5.8 παρατηρούμε ότι για την επεξεργασία πολλαπλών εικόνων η σωστή μέθοδος είναι η επεξεργασία κάθε εικόνας ως ένα ανεξάρτητο task, το οποίο μπορεί να δρομολογηθεί σε οποιονδήποτε κόμβο, χάρη στον σχεδιασμό του προγράμματος.

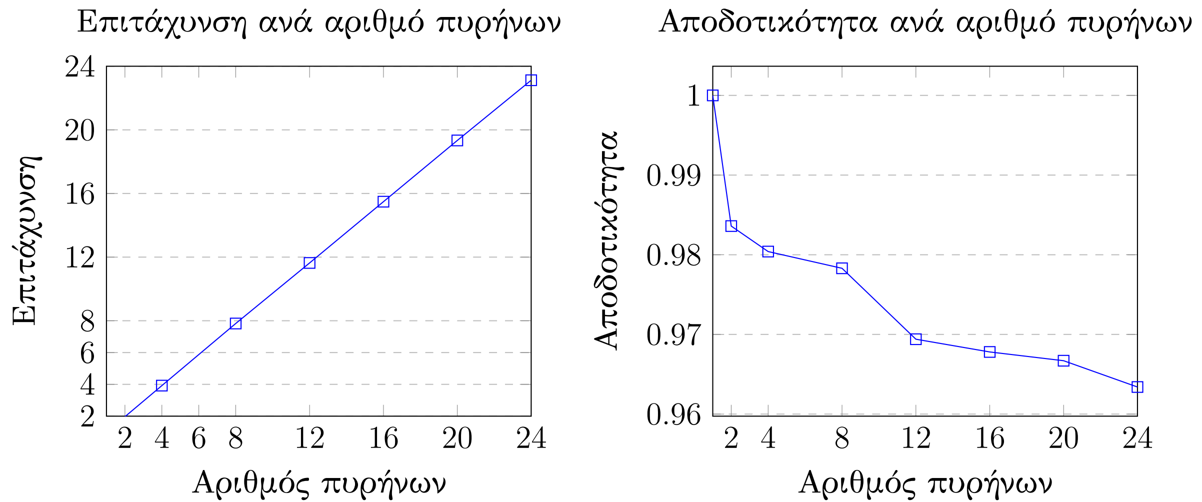
Τόσο σε αυτό το σύστημα, όσο και στο επόμενο που θα μελετήσουμε, η αποδοτικότητα παραμένει άνω του 90% ακόμα και όταν χρησιμοποιούνται όλοι οι επεξεργαστικοί πόροι του συστήματος. Αυτό το αποτέλεσμα είναι αναμενόμενο, καθώς τα tasks είναι πλήρως ανεξάρτητα μεταξύ τους, δίχως καμία εξάρτηση ή διαδικασίες συγχρονισμού. Εάν δούμε την εκτέλεση στους πυρήνες μας ως ένα pipeline, αυτό το pipeline δεν θα βρίσκεται ποτέ άδειο (πέρα από τις διαδικασίες του fill και drain) με αποτέλεσμα κανένας επεξεργαστής να μην παραμένει αδρανής σε οποιαδήποτε χρονική στιγμή. Με αυτόν τον τρόπο επιτυγχάνουμε εξ ορισμού υψηλή αποδοτικότητα.

Στη συνέχεια, ελέγξαμε τις επιδόσεις της παραλληλοποίησης σε επίπεδο μίας μοναδικής εικόνας. Ως δείγματα επιλέξαμε δύο εικόνες με σημαντικά μεγάλη ανάλυση, καθώς και πλήθος προσώπων που εμφανίζονται. Ως μέθοδο δοκιμής του προτύπου OpenMP επιλέξαμε τις παραμέτρους που επέφεραν τα καλύτερα αποτελέσματα, ενώ στη νέα μας υλοποίηση δοκιμάσαμε το εξής. Στα σχήματα 5.9 και 5.10 σημειώνουμε τις τιμές για την υλοποίηση "TORC" και "TORC (reduced precision)". Στην πρώτη περίπτωση έχουν επιλεχθεί οι βέλτιστες παράμετροι για τις επιδόσεις, στην κανονική μας υλοποίηση. Στη δεύτερη περίπτωση έχουμε αφαιρέσει μερικούς μηχανισμούς αμοιβαίου αποκλεισμού στο πρόγραμμα, με στόχο την αύξηση των επιδόσεων. Ως αποτέλεσμα η εφαρμογή αυτή επιστρέφει ένα ορθογώνιο στο οποίο εμπεριέχονται τα πρόσωπα που βρέθηκαν στη δοθείσα εικόνα. Αυτή η υλοποίηση επιφέρει μικρή πτώση στην ακρίβεια του αποτελέσματος, της τάξης των 2-6 pixels απόκλισης έναντι της αρχικής υλοποίησης.

Paragon

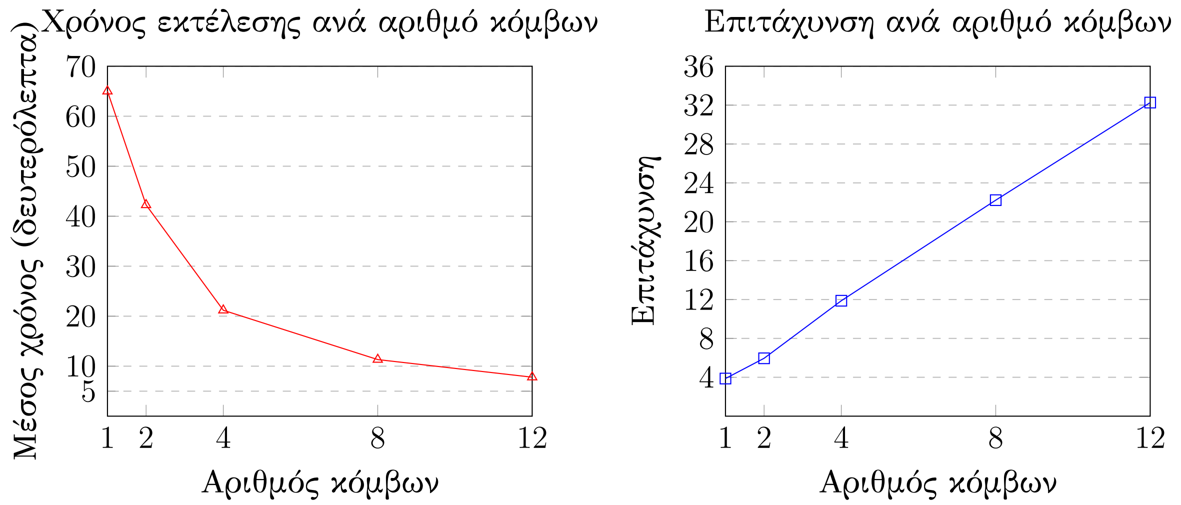
Οι επιδόσεις που φαίνονται στο σχήμα 5.11 επαληθεύουν άμεσα το θεώρημα του Gustafson [10]. Στην περίπτωση του Batch Processing, αντί για να επιλέξουμε να παραλληλοποιήσουμε κάθε πρόβλημά ανεξάρτητα, επιλέξαμε να λύσουμε ταυτόχρονα πολλαπλά προβλήματα, να αυξήσουμε δηλαδή το μέγεθος της εισόδου. Η αποδοτικότητα ξεπερνά το 96%, ακόμα και σε ένα σύστημα με 24 πυρήνες. Εάν συγκρίνουμε το μοντέλο εκτέλεσης αυτό, έναντι της παραλληλοποίησης της επεξεργασίας κάθε εικόνας και να τρέξουμε διαδοχικά τις εργασίες μας, μπορούμε εύκολα να ξέρουμε τι να περιμένουμε, καθώς η παραλληλοποίηση κάθε εικόνας επέφερε μέγιστη αποδοτικότητα ίση με 80%. Κατά συνέπεια, ο χρόνος αδράνειας 20% αυτής της μεθόδου θα ίσχυε σε κάθε εικόνα, με αυτούς τους χρόνους να αθροίζονται.

Σχήμα 5.11: Επιτάχυνση ανά αριθμό πυρήνων



Εξ ορισμού, η έννοια της αποδοτικότητας εκφράζει το ποσοστό αξιοποίησης των επεξεργαστών που χρησιμοποιούνται. Κατά συνέπεια, εάν σε κάθε λύση του προβλήματός μας εισάγουμε χρόνο μη αξιοποίησης των πόρων αυτών, η επιτάχυνση που θα πετύχουμε είναι φραγμένη, όπως αυτό εκφράζεται από τον νόμο του Amdahl [11].

Σχήμα 5.12: Επιδόσεις Face Detection στον Sun Cluster

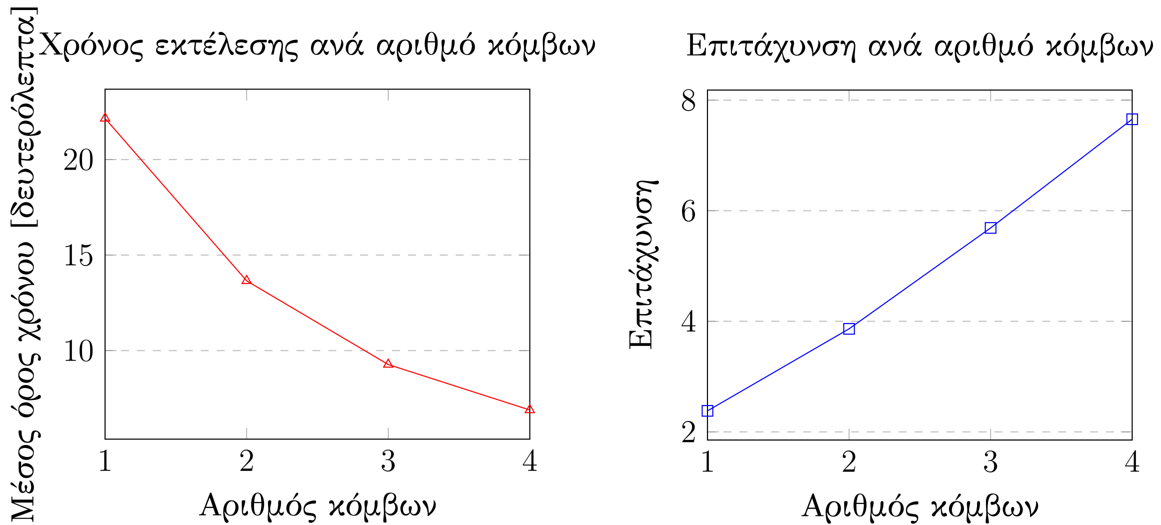


Πλήθος κόμβων	1	2	4	8	12
Αποδοτικότητα	0.9678	0.7443	0.7422	0.6945	0.6721

Σχήμα 5.13: Αποδοτικότητα ανά αριθμό κόμβων

5.4.2 Επιδόσεις κατανεμημένης εκτέλεσης

Σχήμα 5.14: Επιδόσεις Face Detection στον EC2 Cluster

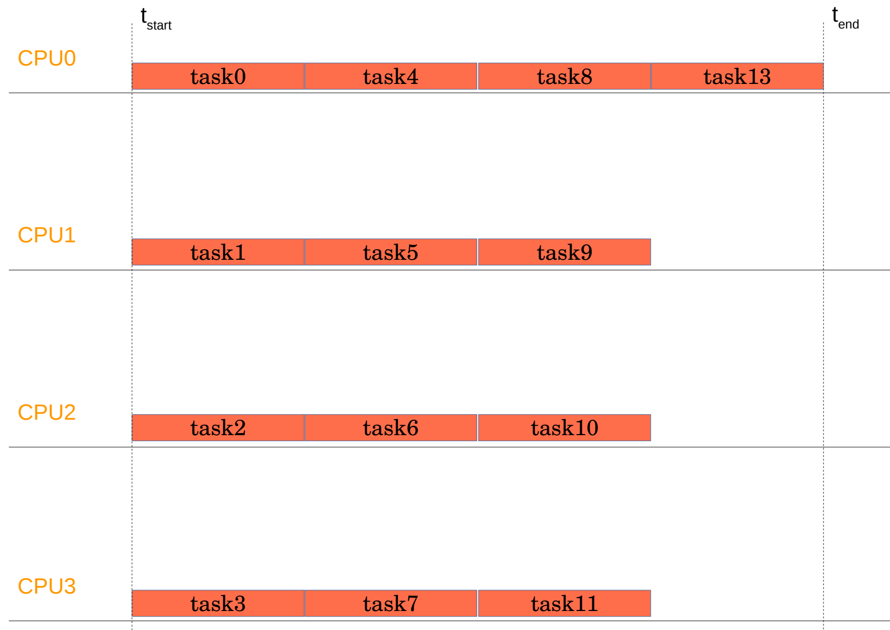


Πλήθος κόμβων	1	2	3	4
Αποδοτικότητα	0.5951	0.4829	0.474	0.4784

Σχήμα 5.15: Αποδοτικότητα ανά αριθμό κόμβων

Στην εφαρμογή αυτή, λόγω της χαμηλότερης κίνησης δεδομένων στο δίκτυο, παρατηρούμε ικανοποιητική αξιοποίηση των κόμβων-εργατών. Η αποδοτικότητα φαίνεται να μην πέφτει σημαντικά όσο αξιοποιούμε περισσότερους κόμβους. Παρόλα αυτά, όπως και στις προηγούμενες εφαρμογές που παρουσιάσαμε, ο κάθε κόμβος φαίνεται να μην επιτυγχάνει υψηλή απόδοση σε εκτέλεση κοινόχρηστης μνήμης. Ένας από τους λόγους στους οποίους θα προσπαθήσουμε να αποδώσουμε αυτό το φαινόμενο, είναι η φύση του υπολογιστικού συστήματος. Η θεωρία μας είναι η εξής. Καθώς κάθε timeshare του συνολικού επεξεργαστικού χρόνου που μας δίνεται δεν εκτελείται απαραίτητα στην ίδια φυσική μονάδα, θα υπάρχουν επιπλέον καθυστερήσεις στις κοινόχρηστες προσβάσεις μνήμης από πολλαπλά νήματα, ή τουλάχιστον αυτό που εμείς θεωρούμε νήμα, με το τυπικό μοντέλο επεξεργασίας μας. Εάν δύο ή παραπάνω νήματα προσπελαύνουν τις ίδιες θέσεις μνήμης, οι επεξεργαστικές οντότητες αυτές ενδέχεται να εκτελούνται σε διαφορετικές φυσικές μονάδες. Συνεπώς, θα υπάρχουν κάποια επιπλέον κόστη για αυτές τις προσβάσεις μνήμης, έναντι της περίπτωσης π.χ. ενός διαύλου. Επίσης, καθώς ο υπολογιστικός μας χρόνος δεν είναι παρά ένα timeshare του ολικού επεξεργαστικού χρόνου των φυσικών μονάδων. Κατά συνέπεια, δεν μπορούμε να περιμένουμε πια να υπάρχει η έννοια των

ευστοχιών της κρυφής μνήμης, η οποία ευνοεί σημαντικά εφαρμογές με συχνές γειτονικές προσβάσεις μνήμης, όπως τις εφαρμογές ταξινόμησης που επιλέξαμε, όπου εκ κατασκευής το πρόγραμμά μας ταξινομεί συνεχή τμήματα μνήμης με την τεχνική Διαίρει-και-Βασίλευε.

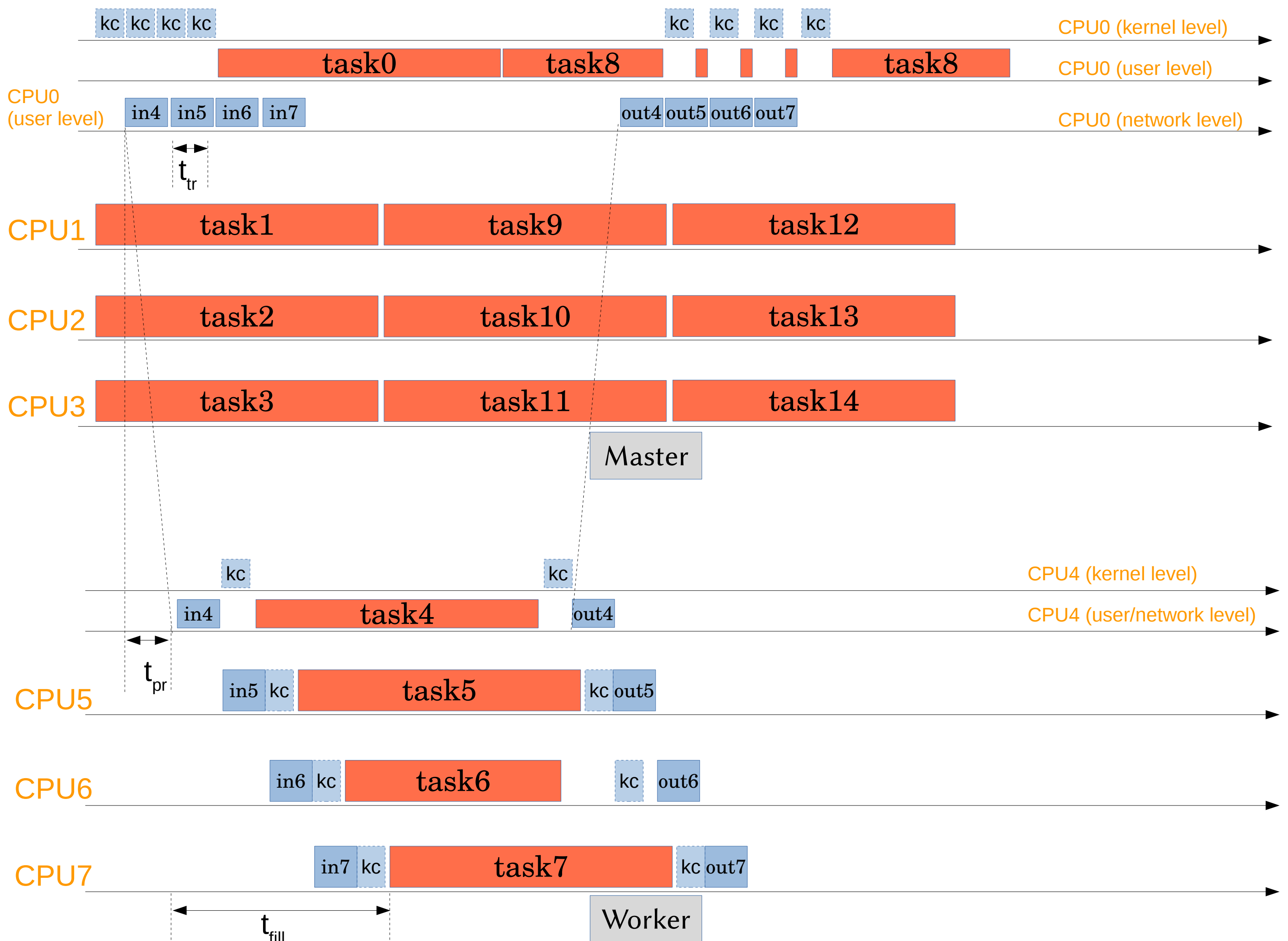


Σχήμα 5.16: Σχηματική ανάλυση παράλληλης εκτέλεσης σε κοινόχρηστη μνήμη

5.5 Θεωρητική Ανάλυση κατανεμημένης εκτέλεσης

Στη συνέχεια θα δείξουμε μία ανάλυση της κατανεμημένης εκτέλεσης ενός προγράμματος, παρουσιάζοντας τα κύρια μειονεκτήματα και περιορισμούς του μοντέλου, καθώς και της σχεδίασης της εφαρμογής. Το σχήμα 5.17 παρουσιάζει τα γεγονότα που λαμβάνουν χώρο κατά την κατανεμημένη εκτέλεση του προγράμματός μας σε ένα σύστημα δύο κόμβων. Καθένας από τους οριζόντιους άξονες x αναπαριστούν τα γεγονότα σε κάθε πυρήνα καθενός από τους δύο αυτούς κόμβους. Ο άξονας γεγονότων του πρώτου πυρήνα έχει χωρισθεί σε τρία τμήματα. Οι άξονες αυτοί αναπαριστούν το επίπεδο χρήστη, το επίπεδο πυρήνα και το επίπεδο δικτύου, ώστε να παρατηρήσουμε τις εξαρτήσεις των γεγονότων ανάμεσα στα επίπεδα αυτά. Όμοια, ο πρώτος πυρήνας του κόμβου-εργάτη έχει χωρισθεί σε έναν άξονα για το επίπεδο χρήστη και ένα για το επίπεδο πυρήνα. Στο σενάριο αυτό υποθέτουμε για ευκολία μας ότι το server-thread του master κόμβου δρομολογείται πάντα στον πρώτο επεξεργαστή, ενώ στον κόμβο-εργάτη δρομολογείται σε οποιονδήποτε επεξεργαστή είναι διαθέσιμος.

Η ανάλυση που ακολουθεί υποθέτει ότι συνολικά εκτελούνται 15 tasks, 4 εκ των οποίων δρομολογούνται σε έναν κόμβο-εργάτη. Η ανάλυσή μας στην συνέχεια γενι-



Σχήμα 5.17: Σχηματική ανάλυση κατανεμημένης εκτέλεσης με δύο κόμβους

κεύεται για οποιοδήποτε πλήθος κόμβων-εργατών, όπου κάθε κόμβος αναλαμβάνει μία τέτοια ομάδα tasks.

Ας αναλύσουμε τα επιμέρους βήματα της διαδικασίας μεταφοράς δεδομένων μεταξύ δύο κόμβων. Η διαδικασία αυτή χωρίζεται σε τρία βήματα.

- Καθυστέρηση διάδοσης, t_{pr}
- Καθυστέρηση μετάδοσης, t_{tr}
- Ο χρόνος επεξεργασίας από τη βιβλιοθήκη, t_{kc}

Τα πρώτα δύο βήματα αυτά αφορούν τα επίπεδα του δικτύου και πραγματοποιούνται σε υλικό διαφορετικό των πυρήνων όπου εκτελούνται τα tasks του χρήστη.

Ως συνέπεια, κατά την εκτέλεση αυτών των βημάτων οι πυρήνες μας μπορούν να εκτελούν tasks του χρήστη, όσο υπάρχουν τέτοια tasks διαθέσιμα. Στην περίπτωση του pipeline fill, δεν υπάρχουν διαθέσιμα tasks, άρα οι χρόνοι αυτοί συμπεριλαμβάνονται στην ανάλυση του χρόνου αδράνειας του συστήματος.

Το τρίτο βήμα αφορά την εκτέλεση συναρτήσεων της βιβλιοθήκης, καθώς και κώδικα πυρήνα. Κατά τη διαδικασία αυτήν εκτελούνται αντιγραφές μνήμης από την κάρτα δικτύου από και προς τον πυρήνα του συστήματος και καθώς δεν μπορούν να συναντηθούν υλοποιήσεις zero-copy στις πιο συνήθη υλοποιήσεις του προτύπου MPI, πραγματοποιούνται επίσης κάποιες αντιγραφές των μεταφερόμενων δεδομένων μεταξύ επιπέδου πυρήνα και user space. Με απλά λόγια, όταν μεταφέρουμε κάποια δεδομένα μέσω δικτύου, αυτά αντιγράφονται πρώτα τουλάχιστον μία φορά σε κάποια ενδιάμεσα επίπεδα μνήμης, προτού αποσταλούν μέσω του ελεγκτή δικτύου στο φυσικό μέσο. Αυτές οι μεταφορές περιλαμβάνουν και άλλες διαδικασίες οι οποίες εκτελούνται στους πυρήνες του συστήματος, όπως ενδεχόμενους ελέγχους ακεραιότητας και ορθότητας δεδομένων, δημιουργία και επεξεργασία επικεφαλίδων πακέτων δεδομένων, κ.α.

Η πρώτη μας παρατήρηση αφορά τον χρόνο επεξεργασίας που καταναλώνεται από λειτουργίες χαμηλότερου επιπέδου από το επίπεδο χρήστη, χρόνο τον οποίο διαφορετικά θα χρησιμοποιούσαμε για την εκτέλεση tasks. Η παραπάνω διαδικασία αντιγραφής και μετακίνησης δεδομένων της μνήμης πραγματοποιείται τόσο σε αποστολές, όσο και σε λήψεις δεδομένων μέσω του δικτύου. Αυτό έχει ως αποτέλεσμα την κατανάλωση $2t_{kc}$ υπολογιστικού χρόνου ανά επικοινωνία, καθώς η διαδικασία αυτή θα εκτελεσθεί τόσο από τον αποστολέα, τόσο και από τον παραλήπτη του μηνύματος. Αυτός ο χρόνος δεν είναι σε καμία περίπτωση αμελητέος, όπως παρατηρήθηκε και στις μελέτες μας. Οφείλουμε επίσης να αναφέρουμε ότι τέτοιες προσβάσεις μνήμης, μπορούν επίσης να ενισχύσουν φαινόμενα όπως η απόρριψη γραμμών κοινόχρηστης κρυφής μνήμης L2, η οποία μπορεί να περιέχει γραμμές της μνήμης που αξιοποιούνται και ευνοούν την εκτέλεση του προγράμματος χρήστη, όπως στην περίπτωση του πολλαπλασιασμού πινάκων που παρουσιάσαμε.

Η επόμενη φάση της ανάλυσής μας αφορά τη διοχέτευση των εργασιών στον απομακρυσμένο κόμβο. Στην περίπτωση της εφαρμογής μας, κάθε απομακρυσμένο task δέχεται ως αποτέλεσμα κάποια δεδομένα και κατά το πέρας της τα επιστρέφει στη διεργασία που δημιούργησε αυτό το task. Όταν οι διεργασίες αυτές βρίσκονται σε διαφορετικούς κόμβους του δικτύου, οι μεταφορές αυτές πραγματοποιούνται

μέσω του δικτύου, απαιτώντας κάποιον χρόνο. Στην έναρξη της εφαρμογής, η διοχέτευση θα είναι άδεια, δηλαδή δεν θα υπάρχουν διαθέσιμα tasks προς εκτέλεση στους απομακρυσμένους κόμβους. Η έναρξη εκτέλεσης ενός task σε κάποιο απομακρυσμένο νήμα-εργάτη είναι γεγονός που εξαρτάται άμεσα από τη μεταφορά κάθε εργασίας.

Κατά συνέπεια, για να γεμίσει η διοχέτευση των εργασιών των w νημάτων-εργατών αυτών, θα πρέπει να καταφτάσουν στον κόμβο αυτόν w μηνύματα, όπως μπορεί να παρατηρηθεί επίσης από τον άξονα του 4ου πυρήνα του κόμβου-εργάτη. Μέχρι να ολοκληρωθεί η διαδικασία των μεταφορών των οποίων οι μηχανισμοί θα αναλυθούν στη συνέχεια, κάθε νήμα-εργάτης παραμένει αδρανής. Στη συνέχεια θα γίνει ανάλυση αυτού του χρόνου, καθώς και γενίκευσή του για οποιονδήποτε αριθμό κόμβων, σε συνδυασμό με ανάλυση του κόστους μεταφορών.

Ένα σημαντικό μειονέκτημα της τοπολογίας Ethernet, ή οποιουδήποτε δικτύου με κοινόχρηστο φυσικό μέσο, είναι ότι η διαδικασία της αποστολής όλων των δεδομένων προς τους απομακρυσμένους κόμβους θα πραγματοποιηθεί σειριακά, καθώς ανά κάθε δεδομένη χρονική στιγμή μπορούμε να αποστέλλουμε μονάχα έναν task descriptor. Η καθυστέρηση διάδοσης του δικτύου συμβαίνει μόνο μία φορά κατά τη διαδικασία αυτήν εάν υποθέσουμε ότι το server thread αποστέλλει δίχως διακοπή τα δεδομένα του, άρα δεν αδειάζει το pipeline του δικτύου. Το ίδιο ισχύει και για την πρώτη διαδικασία kc, καθώς ο συντονιστής και ο κόμβος-εργάτης εκτελούν αυτή τη λειτουργία ταυτόχρονα, πέραν της πρώτης εκτέλεσης η οποία είναι απαραίτητη ώστε να αρχίσουμε να γεμίζουμε το δίκτυο με χρήσιμα δεδομένα. Καθώς αυτές οι μεταφορές πραγματοποιούνται σειριακά, τα νήματα-εργάτες θα λάβουν τα απαραίτητα δεδομένα τους διαδοχικά. Η κάθε μεταφορά απαιτεί χρόνο $t_{tr} + 2t_{kc}$. Συνεπώς, το νήμα-εργάτης θα μείνει αδρανές για χρόνο ίσο με

$$itimes(t_{tr} + t_{kc}) \tag{5.1}$$

ενώ χρειάζεται χρόνος

$$wt(t_{tr} + t_{kc}) \tag{5.2}$$

ώστε να γίνει fill του pipelining, όπου wt το πλήθος των νημάτων-εργατών. Η παραπάνω τιμή πολλαπλασιασμένη με το πλήθος των νημάτων-εργατών κάθε κόμβου θα συμβολίζεται με t_{fill}. Κατά το fill, ο συνολικός χρόνος αδράνειας του απομακρυ-

σμένου κόμβου θα είναι ίσος με

$$\sum_{i=1}^{wt} i(t_{tr} + t_{kc}) \tag{5.3}$$

Στον έναν απομακρυσμένο κόμβο του παραδείγματος, ο συνολικός υπολογιστικός χρόνος κατά τον οποίο τα νήματα-εργάτης του κόμβου αυτού παραμένουν αδρανή είναι ίσος με

$$\sum_{i=1}^{w} ti(t_{tr} + tkc) \tag{5.4}$$

Αυτή η ανάλυση μπορεί να γενικευθεί για N κόμβους-εργάτες. Η τιμή του παραπάνω αθροίσματος θα συμβολίζεται με $t_{node-idle}$. Για να αρχίσει ο i-οστός κόμβος να γεμίζει το pipeline του, πρέπει να έχει ολοκληρωθεί αυτή η διαδικασία στους προηγούμενους i–1 κόμβους. Έπειτα από αυτή την αναμονή, η διαδικασία που θα ακολουθήσει δεν διαφέρει από αυτήν που συνέβη στο παράδειγμα με τον μοναδικό κόμβο-εργάτη. Στην περίπτωση αυτή δείξαμε ότι ο επιπλέον χρόνος μη αξιοποίησης των πυρήνων αυτού του κόμβου ισούται με t_{node_idle}. Συνολικά, ο χρόνος που θα βρίσκονται αδρανή νήματα-εργάτες του i-οστού κόμβου θα είναι ίσος με

$$(i–1)t_{fill} + t_{node-idle} \tag{5.5}$$

Εάν αθροίσουμε αυτό τον χρόνο για κάθε απομακρυσμένο κόμβο, μπορούμε να υπολογίσουμε τον συνολικό χρόνο κατά τον οποίο δεν αξιοποιούνται πυρήνες των κόμβων-εργατών της συστάδας μας. Ο χρόνος αυτός ισούται με

$$\sum_{k=1}^{N} (i–1)t_{fill} + t_{node_idle} + t_{kc} + t_{pr} \tag{5.6}$$

Συνολικά απαιτείται $(N–1)t_{fill}$ χρόνος ώστε να γίνει fill του pipeline, όπου N το πλήθος των κόμβων-εργατών.

Κατά το πέρας της εκτέλεσης, τα αποτελέσματα επιστρέφονται στον master κόμβο. Κατά τη μεταφορά των αποτελεσμάτων ισχύουν τα φαινόμενα που περιγράφαμε παραπάνω. Κατά τη λήψη των μηνυμάτων, υπάρχουν αρκετά διαφορετικά σενάρια που μπορεί να συμβούν. Ένα από αυτό φαίνεται στο παράδειγμά μας. Στην περίπτωση αυτήν, όταν ληφθούν επιτυχώς τα αποτελέσματα, όλοι οι πυρήνες του master κόμβου είναι κατειλημμένοι. Σε αυτή την περίπτωση μπορεί να συμβούν δύο σενάρια, τα οποία εξαρτώνται από την υλοποίηση της βιβλιοθήκης TORC.

Στην άλλη περίπτωση, μπορούμε να υποθέσουμε ότι δεν υπάρχουν τα tasks με ταυτοποιητή από 8 έως 14. Το νήμα-server θα αναλάβει την παραλαβή των δεδομένων όπως αυτά κατέφθασαν επιτυχώς στον κόμβο και η εκτέλεση θα τερματίσει. Αυτή η περίπτωση είναι το γνωστό pipeline drain η οποία είναι πλήρως συμμετρική με αυτήν του fill από την πλευρά της ανάλυσης συνολικού χρόνου που απαιτείται, καθώς και από την ανάλυση χρόνου αδράνειας των διαθέσιμων πυρήνων του κόμβου-εργάτη. Διαφορετικά όμως με την περίπτωση του fill, σε αυτό το σενάριο θα βρίσκονται αδρανείς όχι μόνο οι πυρήνες του κόμβου-εργάτη, αλλά και αυτοί του master κόμβου, καθώς οι διαθέσιμες εργασίες τους έχουν τελειώσει.

Δυστυχώς για τις επιδόσεις, τα προβλήματά μας δεν τελειώνουν εδώ. Σε αυτό το σενάριο δείχνουμε μία περίπτωση με μονάχα έναν κόμβο-εργάτη, ενώ γενικεύσαμε την ανάλυση των χρόνων αδράνειας του pipeline fill για οσουσδήποτε κόμβους-εργάτες. Στην ανάλυση αυτήν παρατηρούμε ότι όσο περισσότερους κόμβους-εργάτες έχουμε διαθέσιμους, τόση περισσότερη ώρα αυτοί μένουν αδρανείς, μέχρις ότου να αρχίσουμε να αποστέλλουμε σε αυτούς τις εργασίες ώστε να ολοκληρωθεί το pipeline fill. Επίσης, επεξεργαστικός χρόνος σπαταλάται και κατά το pipeline drain, για τους ίδιους λόγους. Φυσικά, αυτοί οι χρόνοι θα μπορούσαν να αξιοποιηθούν από άλλες εφαρμογές που τρέχουν στους κόμβους των συστάδων μας, μέσω context-switching των νημάτων όπου τρέχουν τα νήματα-εργάτες μας. Αυτή η διαδικασία προϋποθέτει όμως την ύπαρξη τέτοιων εφαρμογών και πρέπει να σημειώσουμε ότι τα context-switch αυτά δεν έρχονται δίχως κόστη, κόστη τα οποία πρέπει να λάβουμε υπ όψιν εάν αναφερόμαστε σε συστήματα υψηλών επιδόσεων.

Επίσης, ο υπολογιστικός χρόνος που απαιτείται από το λειτουργικό σύστημα για τις αντιγραφές μνημών αυξάνεται με ρυθμό ανάλογο του πλήθους των κόμβων-εργατών, επί έναν σταθερό παράγοντα 2. Το δεύτερο πρόβλημά μας πηγάζει από την υλοποίηση της βιβλιοθήκης που εκτελεί τις μεταφορές δεδομένων και από την αρχιτεκτονική των λειτουργικών συστημάτων, παράγοντες τους οποίους δυστυχώς θα συναντάμε στα πιο πολλά υπολογιστικά συστήματα, με εξαιρέσεις συστήματα ειδικού σκοπού με εξειδικευμένο υλικό. Η μόνη "λύση" που μπορούμε να εφαρμόσουμε ώστε να επιτύχουμε βελτιωμένες επιδόσεις, είναι να αυξήσουμε τον λόγο του υπολογιστικού χρόνου προς τον όγκο δεδομένων, έτσι ώστε το overhead αυτό να γίνεται αμελητέο για κάθε task που αποστέλλουμε, και να υπερισχύει το υπολογιστικό κόστος έναντι αυτού. Η πηγή του πρώτου προβλήματος όμως, το οποίο οδηγεί σε ελάττωση της αξιοποίησης των πόρων υπολογιστικών μας πόρων λόγω,

αφορά την τοπολογία του δικτύου. Σε τοπολογίες διαύλου, όπως αναφέραμε μόνο ένας κόμβος του δικτύου μπορεί να αποστέλει δεδομένα ανά κάθε δεδομένη χρονική στιγμή. Το πρόβλημα αυτό θα αναλυθεί περαιτέρω στην επόμενη ενότητα, ενώ θα προταθούν διάφορες λύσεις ώστε να ελαττώσουμε τις επιπτώσεις του.

Στη συνέχεια, θέλουμε να παρουσιάσουμε μία ακόμα θεωρητική ανάλυση του παραπάνω παραδείγματος, η οποία μπορεί να εφαρμοσθεί σε κάθε κατανεμημένη εκτέλεση. Κατά την ανάλυση αυτήν μας ενδιαφέρουν δύο βήματα του αλγορίθμου. Το πρώτο είναι η μεταφορά των δεδομένων μέσω του δικτύου, ενώ το δεύτερο είναι η αντιγραφή των δεδομένων από το user space προς το kernel space, από το server thread. Έχουμε εξηγήσει γιατί η μεταφορά των δεδομένων μέσω του δικτύου πραγματοποιείται σειριακά. Η διαδικασία της αντιγραφής των δεδομένων είναι μία διαδικασία που εκτελείται επίσης σειριακά ανά διεργασία, καθώςς καθεμιά έχει μόνο ένα server thread.

Και οι δύο αυτές διαδικασίες, ενώ συναντώνται μόνο στην κατανεμημένη εκτέλεση, είναι μέρη του αλγορίθμου που έχουμε υλοποιήσει. Αρκεί μονάχα να παρατηρήσουμε ότι έχουμε μειώσει το ποσοστό των παραλληλοποιήσιμων τμημάτων του αλγορίθμου, καθώς έχουμε αυξήσει το πλήθος των σειριακών βημάτων του αλγόριθμου μας. Σύμφωνα με την ανάλυση του νόμου του Amdahl, όσο αυτός ο λόγος μικραίνει, τόσο μικρότερη η επιτάχυνση που μπορεί να επιτευχθεί σε ένα περιβάλλον παράλληλης εκτέλεσης.

Επίσης, θα θεωρήσουμε ότι όλος ο φόρτος εργασίας που αποστέλλουμε σε κόμβους-εργάτες είναι πλήρως παραλληλοποιήσιμος. Εάν αυτός ο φόρτος εκτελεσθεί τοπικά, τότε δεν θα εκτελεσθούν τα σειριακά αυτά τμήματα που αναφέραμε. Σε όσο περισσότερα απομακρυσμένα tasks διασπούμε τον φόρτο αυτόν, τόσο περισσότερες σειριακές διαδικασίες θα εκτελεσθούν. Κατά συνέπεια, το ποσοστό των παραλληλοποιήσιμων υπολογισμών ως προς το πλήθος των συνολικών υπολογισμών που πραγματοποιούνται πια κατά την εκτέλεση αυτού του φόρτου εργασίας μειώνεται όσο αυξάνεται το πλήθος των τμημάτων στα οποία διαιρούμε αυτό τον φόρτο εργασίας, ενισχύοντας έτσι τις επιπτώσεις του φαινομένου που περιγράψαμε προηγουμένως. Ένα παράδειγμα όπου παρατηρήσαμε αυτό το φαινόμενο στην πράξη, ήταν στην εφαρμογή του πολλαπλασιασμού πινάκων, στον Cluster των AWS, όπου στην περίπτωση των τεσσάρων κόμβων, τα μικρότερα μεγέθη υποπινάκων επέφεραν χειρότερες επιδόσεις (βλέπε Παράρτημα Α).

Θα ολοκληρώσουμε αυτή την ανάλυση επαληθεύοντας την πρότασή μας για την

αύξηση επιδόσεων σε περιβάλλοντα κατανεμημένης εκτέλεσης η οποία συμφωνεί στον πυρήνα της με τον νόμο του Amdahl. Η ιδέα μας πρότεινε την αύξηση του λόγου του χρόνου υπολογισμού ως προς τον χρόνο μεταφοράς των δεδομένων ενός task, στην περίπτωση εκτέλεσής του σε απομακρυσμένο κόμβο. Η ιδέα αυτή δεν αποτελεί παρά μία διαφορετική μετάφραση του νόμου του Amdahl, η οποία προτείνει την αύξηση του ποσοστού του παραλληλοποιήσιμου τμήματος της εκτέλεσης ως προς το σειριακό. Στην περίπτωσή μας το παραλληλοποιήσιμο μέρος της εκτέλεσης είναι η εκτέλεση του task, ενώ το σειριακό είναι η σειριακή μεταφορά και αντιγραφή των δεδομένων του.

Κεφαλαιο 6

Αξιολογηση του HOMPI Framework

6.1 Τα εργαλεία προγραμματισμού από την σκοπιά της Τεχνολογίας Λογισμικού

6.1 Τα εργαλεία προγραμματισμού από την σκοπιά της Τεχνολογίας Λογισμικού

Στο κεφάλαιο αυτό θα γίνει μία σύντομη αξιολόγηση του HOMPI Framework, βάση της εμπειρίας εκμάθησης και επέκτασής του.

6.1.1 Συντηρησιμότητα και Επεκτασιμότητα

Χάρη στο μοντέλο οργάνωσης που παρουσιάσθηκε κατά την ανάλυση της συλλογής εργαλείων, ο μεταγλωττιστής OMPi παρουσιάζει μία οργάνωση κατάλληλη για έναν μεταγλωττιστή με επιστημονικούς στόχους. Η σχεδίαση του συστήματος είναι ιδανική για σκοπούς εκμάθησης και εμβάθυνσης γνώσεων στον τομέα των μεταγλωττιστών, καθώς και των βιβλιοθηκών χρόνου εκτέλεσης παράλληλων συστημάτων.

Οι δύο σχεδόν πλήρως ανεξάρτητες μεταξύ τους οντότητες, αυτή του μεταγλωττιστή και αυτή του περιβάλλοντος εκτέλεσης, εμφανίζουν μεταξύ τους τον ελάχιστο δυνατό δείκτη συσχέτισης (coupling), με αποτέλεσμα οι προγραμματιστές να μπορούν να μπορούν να εργασθούν σε μία από αυτές ανεξάρτητα, δίχως να απαιτούνται εκτεταμένες γνώσεις και εξοικείωση περί του τρόπου λειτουργίας της άλλης οντότητας. Για τις ελάχιστες γνώσεις που απαιτούνται για την εξοικείωση με οποιαδήποτε

από τις δύο οντότητες, τα έγγραφα χρήστη που διατίθενται παρέχουν όλες τις απαραίτητες πληροφορίες.

Η οργάνωση αυτή έχει κάνει δυνατή την ανάπτυξη πολλαπλών βιβλιοθηκών περιβάλλοντος εκτέλεσης, όλες εκ των οποίων είναι επίσης πλήρως ανεξάρτητες μεταξύ τους οντότητες. Η τεχνική που χρησιμοποιείται για τη δημιουργία βιβλιοθηκών χρόνου εκτέλεσης ακολουθεί την τεχνική του Strategy Pattern. Οι βιβλιοθήκες αυτές έχουν συνεχίσει μέχρι σήμερα να επεκτείνονται ώστε να υποστηρίζουν πλήρως το πρότυπο OpenMP. Όλα τα παραπάνω ισχύουν και για το σενάριο του HOMPI, όπου η βιβλιοθήκη εκτέλεσης είναι πια η βιβλιοθήκη TORC. Πέραν της διεπαφής μεταξύ του μεταγλωττιστή και της βιβλιοθήκης, δεν συναντάμε κάπου αλλού συσχέτιση μεταξύ των δύο παρά σε σενάρια που αφορούν την υποστήριξη πολύπλοκων λειτουργιών, όπως για παράδειγμα την υποστήριξη εντολών OpenMP μέσα σε ένα task της βιβλιοθήκης. Ο μεταγλωττιστής λειτουργεί ως ένα εργαλείο για τη διευκόλυνση του χρήστη στη χρήση της βιβλιοθήκης, μέσω της χρήσης pragmas έναντι της κλήσης πολύπλοκων συναρτήσεων και διαχείρισης δεδομένων από τον χρήστη. Ως συνέπεια, πέραν της προσθήκης επιπλέον χαρακτηριστικών στον HOMPI ως σύνολο, όπου απαιτείται η τροποποίηση και των δύο οντοτήτων, οι δύο αυτές οντότητες μπορούν να συντηρηθούν και να τροποποιηθούν ανεξάρτητα, δίχως επιπλοκές.

Παραδείγματος χάριν, για την υλοποίηση του δρομολογητή FTeQ, η μόνη τροποποίηση που υπέστη ο μεταγλωττιστής ήταν η προσθήκη ενός επιπλέον pragma και των αντίστοιχων παραμέτρων του, ώστε να μπορεί να περνά πληροφορίες κατά τον χρόνο εκτέλεσης στη βιβλιοθήκη, οι οποίες αφορούν το task το οποίο πρόκειται να δημιουργηθεί.

Οι προσθήκες τέτοιου τύπου ακολουθούν κυρίως κοινά βήματα μεταξύ τους, βήματα τα οποία πραγματοποιήθηκαν για την εμπλούτιση του framework σε όλες τις υπόλοιπες λειτουργίες που αναφέρθηκαν στα προηγούμενα κεφάλαια. Η διαδικασία ξεκινά με την τροποποίηση των λεκτικών και συντακτικών κανόνων, με σκοπό την υποστήριξη των νέων pragmas, σύμφωνα με τη γραμματική που περιγράφει τις προσθήκες αυτές. Επόμενο βήμα είναι η δήλωση των νέων τύπων κόμβων του συντακτικού δένδρου. Τέλος, ο μεταγλωττιστής χρησιμοποιεί τις πληροφορίες των νέων κόμβων αυτών για να παράξει τον κατάλληλο τροποποιημένο κώδικα. Παραδείγματος χάριν, στην περίπτωση της υποστήριξης του "if" pragma, ο μεταγλωττιστής εντοπίζει την ύπαρξη του αντίστοιχου κόμβου και παράγει ένα αντίστοιχο "if" statement του οποίου η συνθήκη εμπεριέχεται στον κόμβο αυτόν, παράγοντας τον

κατάλληλο τροποποιημένο κώδικα για κάθε περίπτωση του statement.

Όπως φαίνεται από την απλότητα της παραπάνω διαδικασίας, οι επεκτάσεις τέτοιας μορφής απαιτούν ελάχιστη προσπάθεια και όχι περισσότερες γνώσεις πέραν μίας βασικής εξοικείωσης του προγραμματιστή με τα αντίστοιχα εργαλεία λεκτικής και συντακτικής ανάλυσης του μεταγλωττιστή και των βασικών αρχών των εννοιών αυτών.

Φορητότητα

Το σύνολο του πηγαίου κώδικα του HOMPI είναι γραμμένο διατηρώντας κατά νου τις επιδόσεις, αλλά και την φορητότητα. Δεν γίνεται χρήση εξωτερικών βιβλιοθηκών, ενώ έχουν ληφθεί υπ όψιν οι πιο διαδεδομένες οικογένειες λειτουργικών συστημάτων. Ως αποτέλεσμα αυτών, οι εφαρμογές που παρουσιάσαμε στα προηγούμενα κεφάλαια δοκιμάσθηκαν σε μία πληθώρα συστημάτων, συμπεριλαμβανόμενων διαφορετικών αρχιτεκτονικών όπως την αρχιτεκτονική ARM, καθώς και υπηρεσίες cloud computing. Κατά την εργασία αυτήν χρησιμοποιήσαμε μόνο λειτουργικά συστήματα της οικογένειας Linux, όπου η μόνη απαίτηση για τον μεταγλωττιστή OMPi είναι η ύπαρξη ενός μεταγλωττιστή για τη γλώσσα προγραμματισμού C, ενώ για την υποστήριξη του HOMPI απαιτείται η ύπαρξη μίας υλοποίησης του προτύπου MPI, με τη βιβλιοθήκη TORC να υποστηρίζει και τις δύο πιο διαδεδομένες υλοποιήσεις του προτύπου, Open MPI και MPICH.

Ένα άλλο σημαντικό κατά τη γνώμη μας χαρακτηριστικό του συνόλου εργαλείων είναι η φορητότητα της εγκατάστασής του. Το framework δεν απαιτεί ειδικά δικαιώματα χρήστη για την εγκατάστασή του, κάτι το οποίο επιτρέπει την εγκατάστασή του από χρήστες οποιουδήποτε επιπέδου. Αυτό το χαρακτηριστικό σε συνδυασμό με τη δυνατότητα δυναμικού linking των βιβλιοθηκών χρόνου εκτέλεσης ενισχύει σημαντικά τη φορητότητα των εργαλείων αυτών.

Η βιβλιοθήκη TORC

Η βιβλιοθήκη TORC ακολουθεί, όπως και ο μεταγλωττιστής OMPi, 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 και τη συντήρηση της βιβλιοθήκης και του framework ως συνόλου. Κάθε τροποποίηση και προσθήκη πραγματοποιήθηκε τροποποιώντας κατάλληλα τα αντίστοιχα συστατικά του λογισμικού, δίχως επιπλοκές ή ανάγκη για τροποποίηση συστατικών 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. Αρχικά, η μόνες πληροφορίες που απαιτούνται από τη διεπαφή με τον μεταγλωττιστή OMPi είναι η μεταφορά των κατάλληλων διευθύνσεων μεταβλητών μαζί με τη μέθοδο περάσματός τους και το όνομα της συνάρτησης που επιθυμούμε να εκτελεσθεί. Η βιβλιοθήκη είναι υπεύθυνη να διαχειρισθεί τα δεδομένα αυτά τόσο σε επίπεδο του εκτελέσιμου προγράμματος, τόσο και από την πλευρά των προγραμματιστών που επιθυμούν να υλοποιήσουν νέα API για τη χρήση αυτής της βιβλιοθήκης, όπως τη διεπαφή με τον μεταγλωττιστή OMPi, ή τον νέο δρομολογητή που υλοποιήσαμε.

Στην περίπτωση του δρομολογητή μας, αξιοποιήθηκαν η οντότητα του server thread και οι έτοιμες συναρτήσεις δρομολόγησης που χρησιμοποιεί η βιβλιοθήκη. Η ευθύνη μας ήταν η διαχείριση και συντήρηση των απαραίτητων δεδομένων. Πέραν από αυτές τις λειτουργίες, αφού πραγματοποιηθούν οι έλεγχοι ορθότητας των νέων παραμέτρων, χρησιμοποιείται η κατάλληλη συνάρτηση δρομολόγησης του task που μόλις δημιουργήθηκε. Οι συναρτήσεις αυτές είναι ανεξάρτητες για τη δρομολόγηση σε οποιαδήποτε συγκεκριμένη ουρά επιθυμούμε, είτε αυτή είναι ουρά κόμβου/εργάτη, είτε είναι ιδιωτική/δημόσια.

Επιπλέον, παρέχεται πλήρης διαφάνεια στις υποκείμενες βιβλιοθήκες που χρησιμοποιούνται από τη βιβλιοθήκη TORC, με αποτέλεσμα να μην είναι απαραίτητη από τον χρήστη γνώση των εργαλείων αυτών, όπως τη βιβλιοθήκη η οποία υλοποιεί τα user threads.

Κεφάλαιο 7

Προγραμματισμός με Tasks για Παράλληλη και Κατανεμημένη επεξεργασία

7.1 Δυσκολίες του task based μοντέλου προγραμματισμού σε αλγορίθμους ανάλυσης γραφημάτων

7.2 Λύσεις στα προβλήματα της κατανεμημένης εκτέλεσης

7.3 Θεωρητικά εναντίον αληθινών σεναρίων

Τόσο ο προγραμματισμός σε περιβάλλοντα κοινόχρηστης μνήμης, όσο και σε περιβάλλοντα συστάδων κρύβει προβλήματα άλλες φορές φανερά ενώ άλλες όχι, τα οποία μπορεί να συναντήσουν αλλά να μην λάβουν υπ όψιν οι προγραμματιστές, ανεξαρτήτως της εμπειρίας τους.

Ένα άκρως σημαντικό πλεονέκτημα του μοντέλου προγραμματισμού με tasks είναι η δυνατότητα διαφανούς λύσης του προβλήματος της ανισοκατανομής φόρτου, η οποία αποτελεί όχι μόνο πρόβλημα του παράλληλου προγραμματισμού με tasks, αλλά του παράλληλου προγραμματισμού γενικότερα. Περισσότερα για αυτή την ιδέα θα ειπωθούν σε μεταγενέστερα σημεία της παρούσας εργασίας.

Η πιο σημαντική πλευρά της διαδικασίας ανάπτυξης μίας παράλληλης εφαρμογής είναι κατά τη γνώμη μας η σχεδίαση. Πριν κάποιος προχωρήσει σε θέματα

πιο άμεσα συσχετισμένα με την υλοποίηση πρέπει να προβεί σε εκτενής μελέτη του προβλήματος, κατά την οποία πρέπει να ληφθούν υπόψιν όσο δυνατόν περισσότεροι παράγοντες του προβλήματος που μοντελοποιείται. Χρησιμοποιούμε τη μετρική "κάποιοι", καθώς μέρος αυτών των παραγόντων μπορούν να είναι εκ των προτέρων γνωστοί σε κάποιον έμπειρο προγραμματιστή, αλλά δυστυχώς στον πραγματικό κόσμο πολλοί από αυτούς μπορεί να είναι είτε άγνωστοι, είτε κρυμμένοι μέσα στο πρόβλημα του οποίου αναζητάται η επίλυση.

7.1 Δυσκολίες του task based μοντέλου προγραμματισμού σε αλγορίθμους ανάλυσης γραφημάτων

Η πλειοψηφία των δυσκολιών που θα περιγραφεί παρακάτω δεν αποτελούν αφ'εαυτού μειονέκτημα του μοντέλου παραλληλισμού με tasks, αλλά τις κρίνουμε ως σενάρια στα οποία η χρήση του μοντέλου αυτού δεν επιφέρει κάποιο κέρδος. Αναφερόμαστε στο κέρδος τόσο από την άποψη της παραγωγικότητας κατά την υλοποίηση, όσο και των επιδόσεων.

7.1.1 Μη βέλτιστες περιπτώσεις κατανεμημένης εκτέλεσης

Κατανεμημένη εκδοχή του αλγορίθμου του Prim

Θα παρουσιάσουμε μία κατανεμημένη εκδοχή του αλγορίθμου του Prim, η οποία θα χρησιμοποιηθεί ως παράδειγμα για παρουσίαση των παραπάνω απόψεών μας. Η εκδοχή αυτή περιγράφεται λεπτομερώς στο σύγγραμα [12]. Ο αλγόριθμος του Prim είναι ένας επαναληπτικός αλγόριθμος εύρεσης Ελάχιστου Σκελετικού Δένδρου (Minimum Spanning Tree, για συντομία MST). Η σειριακή εκδοχή του αλγορίθμου του Prim περιγράφεται από τον αλγόριθμο 7.1.

Ο αλγόριθμος διατηρεί δύο σύνολα και έναν πίνακα αποστάσεων. Το σύνολο F περιλαμβάνει τις ακμές που έχουν προστεθεί στο ελάχιστο σκελετικό δένδρο, ενώ το σύνολο S περιγράφει τις προσβάσιμες κορυφές. Ο πίνακας ελαχίστων αποστάσεων διατηρεί τις αποστάσεις από τους μέχρι τώρα προσβάσιμους κόμβους του MST προς όλους τους μη προσβάσιμους μέσω της τομής των συνόλων S και $\{V - S\}$. Κατά την αρχικοποίηση το σύνολο F είναι το κενό σύνολο, ενώ το σύνολο S περιλαμβάνει έναν αυθαίρετο εναρκτήριο κόμβο. Ο πίνακας αποστάσεων αρχικοποείται με το βάρος

Αλγόριθμος 7.1 Αλγόριθμος του Prim

```
1: F = {}, S = {s};
2: while S ≠ V do
3:    Έστω ακμή (u, v) η ακμή της τομής με το μικρότερο κόστος ώστε u να ανήκει
      στο U και v να ανήκει στο V - U;
4:    F = F ∪ (u, v)
5:    U = U ∪ v
6: end while
```

των ακμών από τον εναρκτήριο κόμβο προς όλες τις γειτονικές κορυφές του και άπειρο για όλες τις άλλες.

Ο αλγόριθμος λειτουργεί επαναληπτικά, προσθέτοντας σε κάθε επανάληψη την ακμή (u, v) της τομής με το ελάχιστο κόστος στο MST και τον κόμβο v στους κόμβους που έχει πρόσβαση το MST στην τρέχουσα επανάληψη. Αυτή η διαδικασία συνεχίζεται μέχρις ότου όλοι οι κόμβοι του γραφήματος να είναι προσβάσιμοι μέσω του MST.

Ο αλγόριθμος μπορεί να τροποποιηθεί ως εξής ώστε να μπορεί να εκτελεστεί σε περιβάλλον κατανεμημένης μνήμης. Κάθε μία από τις P διεργασίες διατηρεί ανεξάρτητα πληροφορίες για τα σύνολα F και S. Ο πίνακας αποστάσεων χωρίζεται σε ίσα τμήματα μεγέθους $|V|/P$, ενώ μία από τις διεργασίες εκτελεί τον ρόλο του συντονιστή. Κάθε διεργασία διατηρεί ένα τέτοιο υποσύνολο. Η επιλογή της ακμής ελαχίστου κόστους πραγματοποιείται πια κατανεμημένα. Σε κάθε επανάληψη η κάθε διεργασία βρίσκει την ακμή ελάχιστου κόστους ανάμεσα στο υποσύνολο που της αντιστοιχεί και το αποστέλει στον συντονιστή. Η κεντρική διεργασία συλλέγει αυτά τα αποτελέσματα και επιλέγει την ακμή με το ελάχιστο κόστος.

Στη συνέχεια, η κεντρική διεργασία αποστέλλει το αποτέλεσμα σε όλες τις άλλες διεργασίες, δηλαδή τον ταυτοποιητή της ακμής που επιλέχθηκε. Η κάθε διεργασία είναι υπεύθυνη να ενημερώσει τα σύνολα F και S καθώς και τον τμήμα του πίνακα αποστάσεών της, αντίστοιχα, ώστε να απεικονίζουν πια τη νέα κατάσταση του αλγορίθμου, σύμφωνα με τα νέα σύνολα.

Με τη χρήση του γνωστού προτύπου MPI, η διαδικασία συλλογής μπορεί να πραγματοποιηθεί με τη λειτουργία "Reduce", ενώ η διαδικασία ενημέρωσης με τη λειτουργία "Broadcast". Στη συνέχεια θα δείξουμε τις θεωρητικές συνθήκες που απαιτούνται ώστε αυτή η εκδοχή του αλγορίθμου να επιφέρει επιτάχυνση.

Θα υποθέσουμε ότι το δίκτυο που χρησιμοποιείται είναι ένα δίκτυο κοινόχρηστου μέσου, όπως ένα δίκτυο Ethernet `802.3`. Επίσης, δεν θα συμπεριλάβουμε στην ανάλυση μας τη λειτουργία Broadcast, καθώς πρόκειται για μία σταθερά και δεν μπορούμε να είμαστε βέβαιοι εάν θα υλοποιηθεί από τη βιβλιοθήκη MPI με τρόπο ο οποίος εκμεταλλεύεται σωστά το κοινόχρηστο μέσο του δικτύου, ώστε να ελαχιστοποιήσει τις επικοινωνίες. Επίσης, καθώς η ανάλυση του επιπλέον κόστους του δικτύου θα φανεί ότι είναι αρκετή, θα παραλείψουμε τις διαδικασίες ενημέρωσης από τις εργασίες πέραν του συντονιστή.

Η διαδικασία εύρεσης ελαχίστου έχει χρονική πολυπλοκότητα $O(V)$, όπου V το πλήθος των κόμβων. Η διαδικασία πραγματοποιείται προσπελαύνοντας των στοιχείων του πίνακα, στην περίπτωσή μας του πίνακα αποστάσεων. Στην κατανεμημένη εκδοχή του αλγορίθμου, η κάθε διεργασία πραγματοποιεί αναζήτηση του ελαχίστου στο δικό της τμήμα του πίνακα αποστάσεων. Άρα έχουμε P πίνακες, ή αλλιώς P εργασίες, καθεμιά με χρονικό κόστος c. Αντιστοιχούμε μία εργασία σε κάθε διεργασία. Ο συντονιστής εκτελεί μία εργασία, ενώ $P-1$ εργασίες αναλαμβάνονται από άλλες διεργασίες. Το κόστος για να εκτελεσθούν αυτές οι εργασίες σε απομακρυσμένους κόμβους είναι τουλάχιστον όσο το κόστος της συλλογής των αποτελεσμάτων.

Σειριακά, το κόστος εύρεσης ελαχίστου είναι ίσο με $P \times c$. Η παρακάτω ανίσωση περιγράφει τις συνθήκες υπό τις οποίες η απομακρυσμένη εκτέλεση της διαδικασίας αυτής επιφέρει κέρδος.

$$c(P-1) > (P-1)(t_{pr} + t_{tr}) \tag{7.1}$$

Από την οποία συνεπάγεται ότι

$$c > t_{pr} + t_{tr} \tag{7.2}$$

όπου t_{pr} η καθυστέρηση διάδοσης του δικτύου, ενώ t_{tr} η καθυστέρηση μετάδοσης. Στο αριστερό μέλος της ανίσωσης βρίσκεται το κόστος της σειριακής εκτέλεσης των $P-1$ εργασιών στον αρχικό κόμβο, ενώ στο δεξί το επιπλέον κόστος που απαιτείται για την αποστολή αυτών των εργασιών για εκτέλεση σε απομακρυσμένους κόμβους. Όταν αυτή η ανίσωση αληθεύει, τότε ο χρόνος κατανεμημένης εκτέλεσης είναι μικρότερος αυτού της σειριακής.

Με άλλα λόγια, πρέπει το χρονικό κόστος του υπολογισμού που αποστέλλουμε σε απομακρυσμένες διεργασίες να είναι (σημαντικά) μεγαλύτερο από αυτό του κόστους της μεταφοράς. Στην περίπτωσή μας, δεν ισχύει αυτή η συνθήκη. Το υπο-

λογιστικό κόστος αυτής της εργασίες είναι πολύ μικρό λόγω της απλότητάς του υποπροβλήματος, ενώ το δίκτυο είναι τάξεις μεγέθους πιο αργό από το υλικό που πραγματοποιεί τους υπολογισμούς.

Παραδείγματος χάριν, ας υποθέσουμε ότι η είσοδος περιέχει 2^{33} κορυφές και οι αποστάσεις αναπαριστάνονται με αριθμούς κινητής υποδιαστολής διπλής ακρίβειας (8 byte ανά μεταβλητή). Εάν έχουμε 8 διεργασίες, η εύρεση ελαχίστου στο 1/8 του πίνακα αυτού με τη χρήση της λειτουργίας reduction του OpenMP σε ένα σύστημα 4 πυρήνων (i5-4460 με βασικό ρυθμό ρολογιού 3.2GHz) πραγματοποιείται κατά μέσο όρο σε $0.7\mu sec$. Ο χρόνος που μετρήθηκε σε πειραματικές εκτελέσεις της διαδικασίας ping-pong (η οποία απαιτεί RTT - Round Trip Time) για το δίκτυο Ethernet του Sun Cluster απαιτεί $32\mu sec$, άρα ο χρόνος t_{pr} θα κυμαίνεται στο μισό αυτής της τιμής. Βλέπουμε ότι οι δύο αυτές τιμές έχουν διαφορά τάξεων μεγέθους μεταξύ τους.

Παρά λοιπόν την αισιόδοξη ανάλυσή μας, μπορούμε να συμπεράνουμε ότι σε σενάρια πραγματικού κόσμου η υλοποίηση αυτή δεν είναι πρακτική, με εξαίρεση ενδεχομένως περιπτώσεις όπου χρησιμοποιούνται εξειδικευμένα δίκτυα υψηλών επιδόσεων, ή στη γενικότερη περίπτωση όπου το κόστος υπολογισμού κάθε υποπροβλήματος είναι σημαντικά μεγαλύτερο από τις χρονικές καθυστερήσεις του δικτύου. Όταν υποσύνολο προσβάσιμο, κόμβος κάθεται αδρανής.

Ο αλγόριθμος CURE

Για αυτό το παράδειγμα θα χρησιμοποιηθεί ο αλγόριθμος CURE (Clustering Using REpresentatives) και η μεταφορά του στον HOMPI. Όπως και στο πρόβλημα της ανίχνευσης προσώπων, θα περιγράψουμε μονάχα την αλγοριθμική πλευρά της λύσης που τροποποιήθηκε, ώστε να εστιάσουμε στα προβλήματα που προέκυψαν κατά τη μεταφορά αυτήν.

Το κύριο τμήμα του αλγορίθμου αποτελείται από δύο βρόγχους. Και οι δύο βρόγχοι εκτελούνται τόσες φορές όσες το πλήθος των κόμβων του γραφήματος εισόδου. Για την παραλληλοποίηση της εφαρμογής σε περιβάλλον κοινόχρηστης μνήμης οι επαναλήψεις του εσωτερικού βρόγχου κατανέμονται στα διαθέσιμα νήματα.

Στη μεταφορά αυτού του φόρτου εργασίας σε απομακρυσμένες διεργασίες, προκύπτουν αρκετά προβλήματα. Αρχικά, στη συγκεκριμένη εφαρμογή ο εσωτερικός βρόγχος απαιτεί πρόσβαση σε δομές που αφορούν τα σημεία της εισόδου. Τα σημεία αυτά ενδέχεται να τροποποιηθούν σε κάθε επανάληψη. Ο βρόγχος αυτός αποτελείται από πολλαπλές διαφορετικές περιπτώσεις. Κάποιες από αυτές απαιτούν

πρόσβαση σε δεδομένα μονάχα των κορυφών που αντιστοιχούν σε αυτές τις επαναλήψεις, ενώ κάποιες άλλες απαιτούν πρόσβαση στα δεδομένα όλων των κορυφών.

Σε οποιαδήποτε από τις παραπάνω περιπτώσεις, τα δεδομένα αυτά πρέπει να αποσταλούν μαζί με την εργασία σε κάθε απομακρυσμένη διεργασία όπου δρομολογείται. Επίσης, καθώς αυτά τα δεδομένα τροποποιούνται κατά την επεξεργασία, ο συντονιστής είναι υπεύθυνος να συλλέξει αυτές αλλαγές.

Για να αποφύγουμε την αποστολή όλων των δεδομένων στις περιπτώσεις που αναφέρθηκαν παραπάνω, μπορούμε να εφαρμόσουμε την εξής τεχνική. Κάθε απομακρυσμένη διεργασία θα κωδικοποιήσει τον αριθμό των επαναλήψεων στις οποίες συναντήθηκαν αυτές οι περιπτώσεις, θα επιστρέψει αυτή την κωδικοποιημένη τιμή στον δημιουργό της εργασίας, ο οποίος με τη χρήση callback θα εκτελέσει τις αντίστοιχες περιπτώσεις στις κατάλληλες επαναλήψεις.

Ακόμα και με αυτήν τη βελτίωση, τα επιπλέον κόστη της διαδικασίας φαίνονται στους αλγόριθμους 7.2, 7.3 και 7.4. Αναφερόμαστε ως codeSequence στην κωδικοποιημένη ακολουθία η οποία περιγράφει τις επαναλήψεις στις οποίες συναντήθηκαν ειδικές περιπτώσεις. Η ακολουθία αυτή αντιμετωπίζεται ως μία ακολουθία από bits, στην οποία το i-οστό bit συμβολίζει την ύπαρξη ειδικής περίπτωσης στην i-οστή επανάληψη. Αυτή η ακολουθία έχει μέγεθος $log_2 N$.

Αλγόριθμος 7.2 Κατανεμημένος εσωτερικός βρόγχος

```
1: for i = 2 to P do
2:     serializeItems();
3:     serializeNNBs();
4: end for
5: for i = 2 to P  do
6:     createTask(Items[i], NNB[i])
7: end for
8: taskSync()
9: for i = 2 to P do
10:    deserializeItems();
11:    deserializeNNBs();
12: end for
```

Καθώς το framework μας δεν υποστηρίζει την αποστολή πολύπλοκων δομών όπως τα structs με τα οποία αποθηκεύονται οι πληροφορίες της εφαρμογής. Κατά συνέπεια, πρέπει να σειριοποιήσουμε τα δεδομένα αυτά, ώστε να αποσταλούν υπό τη μορφή μία πρωτόγονης μορφής δεδομένων, όπως έναν πίνακα χαρακτήρων.

Η ανάλυση που θα εφαρμοστεί για να δείξουμε ότι είναι πρακτικά αδύνατη η επίτευξη κέρδους σε αυτό τον απλό βρόγχο γίνεται με όμοιο τρόπο με την ανάλυση της κατανεμημένης εκδοχής του αλγορίθμου του Prim. Τα επιπλέον υπολογιστικά κόστη αθροίζονται σε

$$n[\,2(P-1)(t_{NS}+t_{IS})+2(P-1)(t_{NDS}+t_{IDS})+2t_{pr}+2t_{tr}] \tag{7.3}$$

ενώ ο φόρτος που γλιτώνουμε στον συντονιστή είναι ίσος με $(P-1)\times c$, όπου το c έχει χρονική πολυπλοκότητα $O(V)$, όπου V το πλήθος των σημείων της εισόδου. Οι τιμές t_{NS} και t_{NS} συμβολίζουν το κόστος σειριοποίησης των δομών NNB και Items, αντίστοιχα. Όμοια, οι τιμές t_{NDS} και t_{NS} συμβολίζουν τα αντίστοιχα κόστους αποσειριοποίησης. Όπως και στην προηγούμενη περίπτωση που αναλύσαμε, τα επιπλέον κόστη της διαδικασίας υπερτερούν σημαντικά του κέρδους, στην περίπτωσή όμως αυτή σε τέτοιο βαθμό, όπου ακόμα και στην περίπτωση δικτύων υψηλών επιδόσεων δεν θα μπορούσαμε να δούμε κέρδη.

Πέραν αυτών, πρέπει να σημειωθεί ότι η αποστολή δεδομένων με τύπο διαφορετικό των υποστηριζόμενων του framework έγινε με τη διαδικασία σειριοποίησης που θα περιγράψουμε στη συνέχεια.

Αλγόριθμος 7.3 Ο αλγόριθμος κάθε εργασίας

```
1:  instatCodeSequence()
2:  deserializeItems()
3:  deserializeNNBs()
4:  {chunkSize = N / P}
5:  for i = P_id × chunkSize to (P_id + 1) × chunkSize do
6:     if specialCase then
7:        encode(i);
8:     else
9:        cure(i);
10:    end if
11: end for
12: serializeItems()
13: serializeNNBs()
14: sendCodeSequence()
```

Αλγόριθμος 7.4 Ο αλγόριθμος των callbacks

```
1: decodeSequence()
2: deserializeItems()
3: deserializeNNBs()
4: for iterationNo in sequence do
5:    executeSpecialCase(iterationNo)
6: end for
```

Τα παραπάνω προβλήματα πηγάζουν από το γεγονός των εξαρτήσεων και τον επικοινωνιών. Η παραλληλοποίηση του εξωτερικού βρόγχου είναι αδύνατη, καθώς κάθε επανάληψή του εξαρτάται από το αποτέλεσμα της προηγούμενης. Το κόστος των επικοινωνιών καθιστούν μη πρακτική από μόνα τους μία τέτοια υλοποίηση. καθώς και τα κόστη της σειριοποίησης καθιστούν μία τέτοια υλοποίηση μη πρακτική.

7.1.2 Σειριοποίηση πολύπλοκων δεδομένων

Όπως αναφέρθηκε, το μοντέλο προγραμματισμού υποστηρίζει ως παραμέτρους μονάχα τους βασικούς τύπους της γλώσσας προγραμματισμού C, καθώς και πίνακες αυτών. Εάν υπάρχει κάποια διαφορετική δομή δεδομένων η οποία περιγράφεται με

struct, τότε ο χρήστης είναι υπεύθυνος να επινοήσει κάποια λύση ώστε να καταφέρει να αποστείλει αυτά τα δεδομένα χρησιμοποιώντας τη βιβλιοθήκη. Η λύση που προτείνουμε είναι η εξής.

Όπως γνωρίζουμε, η βιβλιοθήκη υποστηρίζει την αποστολή δεδομένων βασικών τύπων. Θα μεταφέρουμε τα δεδομένα του struct υπό την μορφή ενός τέτοιου τύπου δεδομένων, πχ. μέσω ενός πίνακα χαρακτήρων. Θα δεσμεύσουμε δυναμικά έναν χώρο μνήμης ίσο με αυτόν των αρχικών δεδομένων που επιθυμούμε να αποστείλουμε.

Στη συνέχεια τα δεδομένα των δομών αυτών αντιγράφονται, σειριοποιώντας κάθε πεδίο του κάθε στοιχείου σε συνεχείς θέσεις μνήμης. Τέλος, αυτός ο συνεχής χώρος μνήμης μπορεί να αποσταλεί υπό τη μορφή του βασικού τύπου που επιλέχθηκε.

Ενώ η παραπάνω διαδικασία μπορεί να φαίνεται ακίνδυνη σε πρώτη όψη, συναντώνται πολλά προβλήματα κατά τη διάρκεια υλοποίησης και ενσωμάτωσής της σε μία υπάρχουσα εφαρμογή. Η υλοποίηση της υποδομής της σειριοποίησης αυτής ακολουθεί μία ευέλικτη σχεδίαση, η οποία μπορεί να χρησιμοποιηθεί σε οποιαδήποτε δομή δεδομένων. Ακόμα και με τη σχεδίαση αυτή, η υλοποίηση αυτή έχει χωρική επιβάρυνση τουλάχιστον ίση με τον όγκο των δεδομένων που θα αποσταλούν, τόσο στον αποστολέα, όσο και στον παραλήπτη. Αυτό συμβαίνει λόγω του γεγονότος ότι οι buffers που περιέχουν τα σειριοποιημένα δεδομένα απαιτούν ξεχωριστό χώρο μνήμης στον οποίο αντιγράφονται τα αρχικά δεδομένα. Ο αποστολέας είναι υπεύθυνος να δεσμεύσει χώρο για έναν τέτοιο buffer από τον οποίο θα αναγνωστούν τα δεδομένα κατά την αποστολή, ενώ ο παραλήπτης είναι πρέπει να δεσμεύσει έναν αντίστοιχο buffer, στον οποίο θα αποθηκεύσει τα δεδομένα που παραλαμβάνει.

Το παραπάνω κόστος είναι χωρικό, ενώ πρέπει να αναφερθεί ότι η παραπάνω διαδικασία έχει το επιπλέον γραμμικό υπολογιστικό κόστος της αντιγραφής των αρχικών δεδομένων στους αντίστοιχους buffers.

Τα παραπάνω μειονεκτήματα συνοδεύονται από τα εξής προβλήματα, στην περίπτωση της χρήσης αυτής της λειτουργίας σε μία πραγματική εφαρμογή. Παρόλο που οι βασικές λειτουργίες της σειριοποίησης είναι γενικές και μπορούν να χρησιμοποιηθούν στο πνεύμα ενός API, η χρήση αυτή είναι πλήρης ευθύνη του χρήστη. Για παράδειγμα, το μικρό αυτό API παρέχει τη δυνατότητα προσάρτησης δεδομένων στον buffer, όπου ο χρήστης πρέπει να παρέχει το μέγεθος των δεδομένων αυτών. Τα δεδομένα αυτά δεν αποτελούν στη συνέχεια τίποτα πέρα από συνεχόμενα δεδομένα

σε έναν χώρο μνήμης, δίχως καμία ευρετηρίαση. Ο χρήστης είναι τόσο υπεύθυνος για τη σωστή αποθήκευση των δεδομένων του, όσο και για την ανάκτησή τους. Και οι δύο διαδικασίες αυτές είναι επιρρεπείς σε σφάλματα του χρήστη, όπου στις πιο πολλές περιπτώσεις δεν μπορούν να εκσφαλματωθούν εύκολα, καθώς στα σφάλματα αυτά δεν θα πραγματοποιείται καμία παράνομη λειτουργία, όπως παράνομες προσβάσεις μνήμης. Η ορθή λειτουργία της διαδικασίας αυτής βασίζεται καθαρά στη σημασιολογία που παρέχει ο χρήστης.

Τα προβλήματά μας δυστυχώς δεν τελειώνουν εδώ. Στο παρακάτω παράδειγμα θα δούμε μία περίπτωση όπου τα δεδομένα που αποστέλλονται πρέπει να επιστραφούν στον αρχικό αποστολέα, εκτελώντας μία χρονοβόρα διαδικασία ping-pong.

7.1.3 Αφαίρεση μη απαραίτητων πράξεων συγχρονισμού

Ένα ακόμα σημαντικό εμπόδιο για την επίτευξη υψηλών επιδόσεων στην Παράλληλη και Κατανεμημένη επεξεργασία είναι οι πράξεις συγχρονισμού. Όταν δύο ή παραπάνω διεργασίες επιθυμούν να συγχρονιστούν, υπάρχουν διάφορες υλοποιήσεις, ανάλογα με το περιβάλλον προγραμματισμού. Κατά τον συγχρονισμό, δύο η παραπάνω οντότητες εκτέλεσης επιθυμούν να συγχρονιστούν σε κάποιο σημείο εκτέλεσης που ορίζεται από το πρόγραμμα. Αυτό το σημείο μπορεί να είναι κάποιο σημείο του προγράμματος, ή όπως στην περίπτωση των εργασιών του HOMPI, κάποια εργασία να περιμένει το πέρας όλων των εργασιών που έχει δημιουργήσει. Αυτό εκτελείται από τη διαδικασία tasksync, στο μοντέλο προγραμματισμού μας.

Εξ ορισμού, το να περιμένει κάποια οντότητα εκτέλεσης κάποιαν άλλην συνεπάγεται ότι για κάποιο χρονικό διάστημα, η εκτέλεση του προγράμματος αναστέλλεται, μέχρις ότου να ισχύει η συνθήκη που επιθυμούμε να ικανοποιηθεί. Η αναστολή της εκτέλεσης σημαίνει ότι για κάποιο χρονικό διάστημα, η μονάδα επεξεργασίας παραμένει αδρανής, άρα μειώνουμε την αποδοτικότητα του προγράμματος. Υπάρχει βέβαια και η περίπτωση όπου η οντότητα εκτέλεσης αναλαμβάνει άμεσα την εκτέλεση κάποιας άλλης εργασίας, αλλά και σε αυτή την περίπτωση θα υπαρχει το εξής μειονέκτημα. Στο καλύτερο πιθανό σενάριο, βρισκόμαστε σε περιβάλλον κοινόχρηστης μνήμης και οι εργασίες τις οποίες περιμένουμε έχουν τερματίσει πρωτού κληθεί η λειτουργία tasksync, άρα η εκτέλεση του προγράμματος συνεχίζει άμεσα. Ακόμα και σε αυτή την περίπτωση εκτελούνται επιπλέον εντολές, μεταξύ των οποιών κάποιες είναι κλήσης συναρτήσεων, άρα έχουμε ένα υπολογιστικό κόστος κατά την

εκτέλεση της διαδικασίας αυτής, σε κάθε περίπτωση.

Σε ένα περιβάλλον συστάδας, ο μηχανισμός συγχρονισμού συνεπάγεται την εκτέλεση επικοινωνιών μέσω δικτύου, οι οποίες ακόμα και σε ειδικά δίκτυα υψηλών επιδόσεων, συνεπάγονται σημαντικά χρονικά και υπολογιστικά κόστη.

Στην πλειοψηφία των εφαρμογών μας ήταν απαραίτητη η αρχικοποίηση των απομακρυσμένων διεργασιών. Σε κάποιες από αυτές μάλιστα, κατά την αρχικοποίηση έπρεπε να αποσταλούν και τα δεδομένα της εισόδου του προγράμματος. Θα χρησιμοποιήσουμε ως παράδειγμα την εφαρμογή του κατανεμημένου πολλαπλασιασμού πινάκων. Σε αυτή την εφαρμογή, κατά την αρχικοποίηση αποστέλλονται οι δύο πίνακες της εισόδου. Σημειώνεται ότι θα μπορούσαμε ιδανικά να αναθέταμε συγκεκριμένα υποπροβλήματα-εργασίες σε κάθε κόμβο και κατά συνέπεια να ήταν δυνατή η αποστολή μονάχα των τμημάτων της εισόδου που αντιστοιχούν σε αυτά. Αυτό αποφεύχθηκε για δύο λόγους. Ο πρώτος είναι ότι θα εισάγαμε επιπλέον πολυπλοκότητα στο πρόγραμμα, με αποτέλεσμα να αναιρέσαμε την ευκολία της παραλληλοποίησης της εφαρμογής αυτής στο προγραμματιστικό μοντέλο μας, καθώς και να επιφέραμε επιπλέον υπολογιστικά κόστη κατά την αρχικοποίηση. Ο δεύτερος λόγος είναι ότι αυτή η τεχνική θα αναιρούσε την ιδιότητα της κλιμακοσιμότητας της εφαρμογής, καθώς θα έπρεπε η επιπλέον αυτή λογική να προσαρμόζεται ανάλογα με το πλήθος των κόμβων.

Η πρώτη λύση για αυτή την αρχικοποίηση, η οποία ισχύει για κάθε περίπτωση όπου επιθυμούμε να αποστείλουμε τα δεδομένα εισόδου περιγράφεται από τον αλγόριθμο 7.5. Όταν το κύριο πρόγραμμα φτάσει την εντολή tasksync, θα περιμένει πρωτού συνεχίσει την εκτέλεσή του, ώστε να γνωρίζει ότι οι απομακρυσμένοι κόμβοι έχουν αρχικοποιηθεί κατάλληλα, έχοντας αρχικοποιήσει τα δεδομένα που είναι απαραίτητα για την εκτέλεση εργασιών. Σε αυτή τη λύση, ο συντονιστής είναι υπεύθυνος για το τμήμα της λογικής αυτής, κάτι το οποίο όχι μόνο συνεπάγεται έναν κεντρικοποιημένο σχεδιασμό τον οποίο προτιμάμε να αποφεύγουμε, αλλά επιφέρει χρόνους αναμονής υπό τη μορφή συγχρονισμών.

Η βελτίωση που μπορούμε να εφαρμόσουμε είναι η εξής. Ο συντονιστής δημιουργεί τις εργασίες αρχικοποίησης για κάθε απομακρυσμένο κόμβο, αλλά δεν περιμένει να τερματίσουν. Αντιθέτως, η λογική του ελέγχου αρχικοποίησης κατά την εκτέλεση εργασιών αναλαμβάνεται από τον κάθε απομακρυσμένο κόμβο. Η κάθε διεργασία διατηρεί μία μεταβλητή που δηλώνει εάν η διεργασία έχει αρχικοποιηθεί. Κατά την έναρξη εκτέλεσης μίας εργασίας, ο κάθε κόμβος ελέγχει αυτή τη μεταβλητή, προτού

Αλγόριθμος 7.5 Διαδικασία προετοιμασίας προγράμματος

```
1: for all remote processes do
2:   create_init_task(input)
3: end for
4: tasksync
```

Αλγόριθμος 7.6 Κώδικας στην έναρξη εργασίας

```
1: while process_not_instantiated do
2:   wait
3: end while
4: execute_task
```

αρχίσει να εκτελεί την εργασία 7.6. Σε περιβάλλοντα πολλαπλών νημάτων-εργατών αυτός ο έλεγχος είναι απαραίτητος, καθώς είναι πιθανό ο εργάτης Β να έχει αναλάβει μία εργασία φόρτου, όσο ο εργάτης Α έχει αναλάβει αλλά δεν έχει τερματίσει την εκτέλεση της εργασίας αρχικοποίησης.

Αυτή η αναμονή στη συνθήκη μπορεί να υλοποιηθεί με διάφορους τρόπους, ανάλογους με το σύνολο εργαλείων στο οποίο βασίζεται η εφαρμογή. Στην υλοποίησή μας, εκτελείται ενεργός αναμονή, για λόγους επιδόσεων. Η μέθοδος αυτή επέφερε αύξηση της επιτάχυνσης των εφαρμογών, όπως μπορεί να φανεί στα αποτελέσματα των πειραμάτων.

7.1.4 Ασύγχρονη εκτέλεση κώδικα αρχικοποίησης - εκτίμηση

Παραπάνω δείξαμε μία από τις πιθανές λύσεις για τη λύση ενός σεναρίου αρχικοποίησης κάποιων προγραμμάτων. Στο δεύτερο κεφάλαιο παρουσιάσαμε μία εναλλακτική λύση για κάποιες περιπτώσεις αυτού του προβλήματος, η οποία ενσωματώθηκε στο framework. Η χρήση αυτής της λειτουργίας όμως απαιτεί προσοχή. Παραδείγματος χάρην, έστω το σενάριο όπου το πρόγραμμα αρχικοποιεί κάθε διεργασία με τον ίδιο, μεγάλο όγκο δεδομένων, τον οποίο φορτώνει από τον (συμβατικό) δίσκο του. Στην αρχική υλοποίηση, ο συντονιστής στέλνει το ίδιο μεγάλο αυτό κομμάτι δεδομένων σε κάθε διεργασία, ξεχωριστά, κάτι το οποίο έχει προβλήματα, όπως περιγράψαμε παραπάνω.

Εάν χρησιμοποιηθεί η λειτουργία που παρέχουμε, η κάθε διεργασία θα φορτώσει αυτά τα δεδομένα ανεξάρτητα, έχοντας πια μία κατανεμημένη εκτέλεση αυτής

της λειτουργίας. Αυτό μπορεί να ακούγεται τόσο ιδανικό, όσο και κλιμακώσιμο. Δυστυχώς όμως αυτό δεν θα ισχύει σε όλες τις περιπτώσεις.

Στην περίπτωση όπου οι πόροι από τους οποίους φορτώνουν τα δεδομένα τους είναι ανεξάρτητοι δια των διεργασιών, η προσδοκία μας αυτή θα πραγματοποιηθεί. Εάν όμως οι πόροι αυτοί είναι κοινόχρηστοι με οποιονδήποτε τρόπο, το κέρδος αυτής της λειτουργίας δεν θα είναι βέλτιστο και είναι πιθανόν να επιφέρει έως και επιβράδυνση. Παραδείγματος χάρην, εάν δύο ή παραπάνω διεργασίες βρίσκονται στον ίδιο κόμβο, ακόμα και εάν με κάποιον τρόπο φροντίσουμε καθεμιά άπό αυτές να φορτώσει τα δεδομένα από διαφορετικό φυσικό δίσκο, ενδέχεται να μοιράζονται κάποιους πόρους που εκτελούν τη μεταφορά από τον δίσκο, όπως διαύλους ή ελεγκτές υλικού. Εάν οι διεργασίες μοιράζονται τον ίδιο δίσκο, ο ανταγωνισμός των διεργασιών αυτών για τον ίδιο δίσκο θα αποτελέσει επιπλέον πρόβλημα.

Επίσης, σε συστήματα συστάδων κυριαρχεί η χρήση ενός κοινόχρηστου συστήματος αρχείου. Ως συνέπεια, κατά τη χρήση της λειτουργίας που παρέχουμε όλοι οι κόμβοι στους οποίους βρίσκονται διεργασίες θα προσπελάσουν αυτό το σύστημα αρχείων ταυτόχρονα. Καθώς αναφερόμαστε επίσης σε κοινόχρηστους πόρους, αυτό θα επιφέρει διάφορα κόστη, τα οποία θα επιβραδύνουν τον αναμενόμενο χρόνο εκτέλεσής μας.

Ένα άλλο πρόβλημα είναι ότι εάν αναφερόμαστε σε πρόσβαση σε συμβατικούς δίσκους, πρέπει να λάβουμε υπ' όψιν ότι οι δίσκοι αυτοί έχουν τόσο αργούς χρόνους μεταφοράς, όσο και χρόνους αναζήτησης. Οι ταχύτητες αυτές σε αρκετές περιπτώσεις είναι συγκρίσιμες με αυτές των δικτύων, άρα πρέπει να αποφευχθούν.

7.2 Λύσεις στα προβλήματα της κατανεμημένης εκτέλεσης

Σχεδόν όλα τα προβλήματα που αναφέραμε μέχρι τώρα αποτελούν γενικότερα προβλήματα των κατανεμημένων συστημάτων και του παράλληλου προγραμματισμού. Οι παράμετροι καθώς και οι πιθανές λύσεις των προβλημάτων αυτών μεταβάλλονται ανάλογα με τις συνθήκες στις οποίες αυτά προκύπτουν. Σε σχεδόν κανένα από τα προβλήματα αυτά δεν μπορεί να δοθεί μία καθολική λύση. Είμαστε όμως υπεύθυνοι ως σχεδιαστές για δύο κύρια πράγματα.

Το πρώτο είναι να μπορούμε να προβλέψουμε την ύπαρξη τέτοιων προβλημάτων σε όσο πιο δυνατόν πρώιμα στάδια της σχεδίασής μας. Τέτοιες προβλέψεις είναι

όχι μόνο κερδοφόρες, αλλά απαραίτητες για διαδικασία της ανάπτυξης παράλληλων εφαρμογών. Δίχως αυτές η συμπεριφορά των εφαρμογών μπορεί να διαφέρει δραματικά από τις αρχικές προβλέψεις μας, σε βαθμό όπου όπως δείξαμε μπορεί να καθιστά κάποια υλοποίηση πρακτικά άσκοπη.

Η πρόβλεψη είναι το πρώτο μέρος της επίλυσης (όπου είναι δυνατό) των προβλημάτων που συναντάμε. Το επόμενο μέρος αφορά τη δυναμική προσαρμογή λύσεων οι οποίες λαμβάνουν υπ' όψιν όσες παραμέτρους επηρεάζουν το πρόβλημά μας και την ταυτόχρονη εκμετάλλευση των τέτοιων παραμέτρων από τις οποίες μπορούμε να ευνοηθούμε. Οι παράμετροι αυτοί μεταβάλλονται συνεχώς, με την εμφάνιση νέων τεχνολογιών και την πρόοδο της επιστήμης μας και την πάροδο του χρόνου. Μερικές από αυτές αφορούν την επεξεργαστική ισχύ, τις τοπολογίες διασύνδεσης των δικτύων που χρησιμοποιούνται, εμφάνιση νέου υλικού ειδικού σκοπού χάρη στην εξέλιξη των ηλεκτρονικών κυκλωμάτων, κ.α. Παρακάτω θα δώσουμε ένα τέτοιο παράδειγμα, όπου η εφαρμογή τροποποιείται ώστε να εκμεταλλευτεί την τοπολογία του δικτύου.

7.2.1 Εκμετάλλευση της τοπολογίας του δικτύου

Μερικά από τα προβλήματα που συναντήσαμε μέχρι τώρα πηγάζουν από την κοινή χρήση πόρων από δύο ή παραπάνω οντότητες, ταυτόχρονα. Το πρόβλημα αυτό αποτελεί γενικότερο πρόβλημα των κατανεμημένων συστημάτων και δυστυχώς δεν έχει κάποια καθολική λύση.

Βρισκόμαστε σε ένα πρόβλημα του οποίου οι λύσεις εξαρτώνται από την τοπολογία του συστήματος στο οποίο συναντώνται, άρα θα πρέπει να προσαρμόσουμε τις λύσεις μας αντίστοιχα, ώστε να εκμεταλλευτούμε τις συνθήκες αυτές.

Ας χρησιμοποιήσουμε ως παράδειγμα την περίπτωση της αρχικοποίησης όμοιων δεδομένων δια πολλαπλών διεργασιών. Αυτή η περίπτωση μπορεί να γενικευθεί σε αυτήν της πολυεκπομπής.

Στα συστήματα που εφαρμόσαμε την ανάλυση μας και τις παρατηρήσεις μας, χρησιμοποιούταν δίκτυα Ethernet. Τα δίκτυα αυτά είναι εμποδιστικά δίκτυα κοινόχρηστου μέσου, στα οποία ανά κάθε χρονική στιγμή είναι δυνατή η αποστολή ενός μοναδικού μηνύματος από έναν κόμβο. Ας υποθέσουμε πια διαφορετικές τοπολογίες. Έστω ότι βρισκόμαστε σε ένα δίκτυο όπου το επίπεδο μεταφοράς δεν είναι πια κοινόχρηστο. Στη συνέχεια, ας υποθέσουμε ότι το δίκτυο αυτό έχει αντιστοιχιστεί

σε έναν υπερκύβο.

Μία λύση για την εκπομπή ενός μηνύματος σε όλους τους κόμβους του δικτύου είναι ο αλγόριθμος της πλημμύρας. Όπως είναι γνωστό, στην τοπολογία υπερκύβου με 2^d κόμβους η μέγιστη απόσταση μεταξύ οποιονδήποτε δύο κόμβων είναι ίση με d. Κατά συνέπεια, η κατανεμημένη διαδικασία αυτή θα απαιτήσει d βήματα και τη συνολική αποστολή $P-1$ μηνυμάτων, όπου P το πλήθος των κόμβων. Στη διάρκεια αυτής της διαδικασίας, ο κάθε κόμβος θα εκτελεί το πολύ όσες αποστολές όσες τον βαθμό του, δηλαδή d, έναντι της περίπτωσης κοινόχρηστου μέσου όπου ο συντονιστής εμπλέκεται ταυτόχρονα σε $P-1$ αποστολές.

7.3 Θεωρητικά εναντίον αληθινών σεναρίων

Μία ερώτηση προς τον αναγνώστη, έχοντας διαβάσει τα τελευταία κεφάλαια, θα ήταν η εξής. Έστω ότι είμαστε οι διαχειριστές ενός μικρού cluster, κλίμακας όμοιας με του εικονικού cluster που χρησιμοποιήθηκε στα πειράματα στα Amazon Web Services. Για τις ανάγκες του οργανισμού στον οποίο ανήκουμε, μας ανατίθεται ως εργασία η χρήση του cluster μας για την επίλυση πολλών προβλημάτων πολλαπλασιασμού πινάκων. Η περίπτωση αυτή δεν είναι καθόλου φανταστική, καθώς ο πολλαπλασιασμός πινάκων είναι ένα πρόβλημα με εφαρμογές σε πολλούς κλάδους της Επιστήμης μας, όπως τα Γραφικά, την Ρομποτική, την Γραμμική Άλγεβρα, κτλ. Με άλλα λόγια, ως είσοδο έχουμε τους πίνακες A και B και ζητάτε από εμάς να δώσουμε τα αποτελέσματα του πολλαπλασιασμού των πινάκων αυτών, δηλαδή τον πίνακα C. Το πλήθος τέτοιων προβλημάτων θα είναι σημαντικά μεγάλο. Ο στόχος μας είναι η επίλυση όλων αυτών των προβλημάτων στο μικρότερο πιθανό χρονικό διάστημα. Κάποιος ερευνητής μας προτείνει δύο επιλογές. Η πρώτη είναι η χρήση του κατανεμημένου προγράμματος που παρουσιάσθηκε, ώστε να επιταχύνουμε την επίλυση κάθε προβλήματος χρησιμοποιώντας όλους μας τους πόρους σε κάθε πρόβλημα ταυτόχρονα, ή το παράλληλο πρόγραμμα κοινόχρηστης μνήμης, τρέχοντας ανεξάρτητα σε κάθε κόμβο ένα στιγμιότυπο του προγράμματος αυτού ανά χρονική στιγμή, για να λύσουμε τα προβλήματα που μας ανατέθηκαν με κάθε κόμβο ως ανεξάρτητη οντότητα. Η ερώτηση είναι, ποια από τις δύο παραπάνω τεχνικές αυτές να χρησιμοποιήσουμε.

Η απάντηση είναι ότι στις πιο πολλές περιπτώσεις, καμία από αυτές τις δύο

λύσεις δεν είναι η βέλτιστη από αυτές που έχουμε εξετάσει σε αυτή την εργασία. Κανείς πρέπει μόνο να κοιτάξει τα efficiency σε καθένα από αυτά τα προγράμματα. Για να λύσουμε πολλά προβλήματα ανά μονάδα χρόνου, θέλουμε το υψηλότερο δυνατό efficiency. Ακόμα και εάν διαθέτουμε μονάχα έναν κόμβο, για να είναι το ίδιο αποδοτική η χρήση του παράλληλου προγράμματος με αυτήν του σειριακού, θα πρέπει να έχουμε efficiency κοντά στο 100%, ενώ είδαμε ότι σύμφωνα με τη θεωρία αλλά και την πράξη, αυτός ο αριθμός είναι πιο χαμηλός, ή και πολύ χαμηλότερος από αυτό το ποσοστό στον πραγματικό κόσμο. Η βέλτιστη λύση με βάση τις επιλογές που έχουν παρουσιασθεί μέχρι τώρα, είναι η εκτέλεση ενός στιγμιότυπου του σειριακού προγράμματος πολλαπλασιασμού πινάκων σε κάθε πυρήνα του κάθε κόμβου, λόγω του μη ιδανικού efficiency του παράλληλου προγράμματος. Στην περίπτωση συστάδας τέτοιων κόμβων, αρκεί η εφαρμογή της ίδιας τεχνικής σε κάθε κόμβο. Με άλλα λόγια, η επεξεργασία παρτίδας.

Φυσικά, η παραπάνω απάντηση δεν είναι καθολική. Μπορούμε να συναντήσουμε περιπτώσεις, όπου οι παράμετροι συνωμοτούν κατά κάποιον τρόπο εναντίον μας. Παραδείγματος χάριν, στη συγκεκριμένη εφαρμογή, ελέγξαμε την παραπάνω μας θεωρία σε έναν επεξεργαστή παλαιότερης γενιάς, Core 2 Quad Q6600. Στο σύστημα αυτό, η concurrent εκτέλεση τεσσάρων διεργασιών επίλυσης του προβλήματος πολλαπλασιασμού πινάκων είχαν μικρή βελτίωση έναντι του χρόνου εκτέλεσης τεσσάρων παράλληλων διεργασιών, στη σειρά. Αυτό μας δείχνει ότι τα σενάρια του πραγματικού κόσμου δεν συμφωνούν πάντα με τις θεωρίες μας. Ευτυχώς για εμάς, έχουμε αναλύσει την εφαρμογή αυτήν αρκετά ώστε να μπορούμε να εξηγήσουμε αυτό το συμβάν. Η απάντηση είναι απλή. Η καθεμιά σειριακή διεργασία επιβραδύνεται για τους ίδιους λόγους που επιβραδύνονται τα νήματα της παράλληλης διεργασίας. Καθώς στο σύστημα μας εκτελούνται ταυτόχρονα πολλαπλές διεργασίες, αυτές συναγωνίζονται για πρόσβαση στους πόρους, μεταξύ των οποίων όπως αναφέραμε ο πιο σημαντικός είναι η κοινόχρηστη κρυφή μνήμη και οι αποτυχίες που προκαλεί η ταυτόχρονη πρόσβαση από πολλά νήματα σε αυτήν. Επιπλέον, στο τρέχων σύστημα παλαιότερης γενιάς κάθε παράγοντας του συστήματος υστερεί σε σχέση με ένα μοντέρνο υπολογιστικό σύστημα, παράγοντες όπως την ταχύτητα της μνήμης και τις ταχύτητες των διαύλων. Ο συγκεκριμένος επεξεργαστής εμφανίσθηκε στην αγορά το 2007, λίγα χρόνια αφότου τα πολυπύρηνα συστήματα έγιναν διαδεδομένα σε καταναλωτικό επίπεδο.

Αντίθετα, τα αρνητικά αποτελέσματα αυτά δεν εμφανίστηκαν σε μοντέρνα συ-

στήματα, ούτε στους εικονικούς κόμβους των AWS. Σε αυτές τις περιπτώσεις η πτώση των επιδόσεων λόγω της ταυτόχρονης εκτέλεσης πολλαπλών διεργασιών ήταν αρκετά μικρή ώστε η προτεινόμενή μας λύση να νικά τις εναλλακτικές της.

Κεφάλαιο 8

Επίλογος

8.1 Σύνοψη διπλωματικής εργασίας

8.2 Προτάσεις για μελλοντική εργασία

8.1 Σύνοψη διπλωματικής εργασίας

Η διπλωματική εργασία αυτή είχε σκοπό τη μελέτη ενός μοντέρνου συνόλου εργαλείων για την ανάπτυξη παράλληλων εφαρμογών σε συστήματα συστάδων, καθώς και την εκτέλεση εφαρμογών σε τέτοια περιβάλλοντα. Στο πρώτο τμήμα, επεκτείναμε τις τρέχων δυνατότητες του συνόλου εργαλείων ενώ ταυτόχρονα μελετούμε την αρχιτεκτονική και εσωτερική δομή τόσο του μεταγλωττιστή, όσο και της βιβλιοθήκης περιβάλλοντος εκτέλεσης.

Στο δεύτερο τμήμα αναπτύξαμε και αναλύσαμε την εκτέλεση εφαρμογών στο εν λόγω framework. Κατά τη μελέτη αυτήν ανάγαμε την ανάλυσή μας σε δεδομένες αρχές της ανάλυσης των παράλληλων συστημάτων, ώστε να μελετήσουμε τις επιπτώσεις των μεταφορών και γενικότερα των επιπλέον κοστών που επιφέρει η κατανεμημένη εκτέλεση. Επίσης, μελετήσαμε και δείξαμε τις επιπτώσεις που μπορεί να επιφέρει η μετατροπή μίας εφαρμογής στην αντίστοιχη κατανεμημένη εκδοχή της, καθώς και τις δυσκολίες που κρύβει η διαδικασία αυτή για τους προγραμματιστές.

8.2 Προτάσεις για μελλοντική εργασία

Η πρότασή μας για επέκταση του συνόλου εργαλείων εστιάζεται γύρω από την κύρια λειτουργία που λείπει από το εν λόγω framework, την μη δυνατότητα αποστολής structs της γλώσσας C. Η μέθοδος για να αποστείλει ο χρήστης τέτοια δεδομένα επιφέρει τόσο υπολογιστικό, όσο και επεξεργαστικό κόστος. Καθώς το πρότυπο MPI επιτρέπει την αποστολή τέτοιων δεδομένων, η αυτοματοποίηση της διαδικασίας αυτής με τη βοήθεια του OMPi είναι εφικτή, και προτείνεται ως μελλοντική εργασία.

Ο νέος δρομολογητής EFTeQ έχει σχεδιαστεί με την επεκτασιμότητα υπ όψιν και οι περιορισμοί του φράσσονται κατά τη γνώμη μας μονάχα από τους περιορισμούς της εφαρμογής. Ο προγραμματιστής μπορεί να εισάγει εάν επιθυμεί νέα πληροφορία στο σύστημα και στη συνέχεια να τροποποιήσει τον αλγόριθμο δρομολόγησης ή να δημιουργήσει άλλα πλαίσια λειτουργίας του, εκ νέου. Παραδείγματος χάριν, καθώς ο δρομολογητής γνωρίζει ήδη τις απαραίτητες πληροφορίες επεξεργαστικών δυνατοτήτων για τους κόμβους, θα μπορούσε να υλοποιήσει έναν στατικό αλγόριθμο δρομολόγησης, για τις περιπτώσεις όπου γνωρίζει a priori το πλήθος και τη φύση των εργασιών. Αυτό θα μπορούσε να επιτευχθεί υλοποιώντας έναν αλγόριθμο μέγιστου ταιριάσματος ελάχιστης ροής. Στο ένα μέρος του διμερούς γραφήματος κόμβων-εργασιών θα βρίσκονται οι κόμβοι, ενώ στο άλλο οι εργασίες, ενώ τα βάρη και η ύπαρξη των ακμών αυτών θα είναι συνάρτηση των δυνατοτήτων του κόμβου και τη φύση των εργασιών.

Το θέμα της μελέτης των κατανεμημένων εφαρμογών αποτελεί έναν τομέα που προσελκύει ενδιαφέρον τόσο από τεχνικής άποψης, τόσο και από θεωρητικής, εδώ και αρκετά χρόνια. Η μελέτη αυτή θα συνεχίσει να απασχολεί τον τομέα μας, για αυτό και πιστεύουμε ότι η μελέτη που εφαρμόσαμε στα πλαίσια αυτά μπορεί να συνεχιστεί, σε όσο βάθος και πλάτος επιθυμεί κανείς. Η πρότασή μας θα ήταν η εκτενής μελέτη των χρόνων αδράνειας, μεταφοράς και τον υπολογισμών από χαμηλότερα επίπεδα του λειτουργικού κατά την εκτέλεση εφαρμογών, καθώς αυτή η πτώση στις επιδόσεις σχετίζεται άμεσα με αυτούς.

Βιβλιογραφια

[1] V. V. Dimakopoulos, *Parallel Systems and Programming*, pp. 27–36. Athens, Greece: Hellenic Academic Libraries Link, 2015.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," November 2015.

[3] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.1," June 4th 2015. available at: `http://www.mpi-forum.org` (Jun. 2015).

[4] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Task-based execution of nested openmp loops," Tech. Rep. Proc IWOMP 2012, University of Ioannina, 2012.

[5] P. E. Hadjidoukas and V. V. Dimakopoulos, "Hompi: A hybrid programming framework for expressing and deploying task-based parallelism," *Proc. Euro-Par 2011, 17th Int'l Euro-Par Conference on Parallel Processing, Bordeaux, France*, vol. 21, no. 15, pp. 14–26, 2011.

[6] P. Hadjidoukas, E. Lappas, and V. Dimakopoulos, "A runtime library for platform-independent task parallelism," pp. 229–236, 02 2012.

[7] A. Radenski, "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps," Tech. Rep. Proc PDPTA'11, Chapman University, 2011.

[8] D. Huba, "Parallel merge sort." `http://dzmitryhuba.blogspot.com/2010/10/parallel-merge-sort.html`, Oct 2010. Accessed on 2018-10-01.

[9] P. E. Hadjidoukas, V. V. Dimakopoulos, M. Delakis, and C. Garcia, "A high-performance face detection system using openmp," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 15, pp. 1819–1837, 2009.

[10] J. L. Gustafson, *Gustafson's Law*, pp. 819–825. Boston, MA: Springer US, 2011.

[11] J. L. Gustafson, *Amdahl's Law*, pp. 53–60. Boston, MA: Springer US, 2011.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Addison-Wesley, second ed., 2004.

Παραρτημα A

Πηγαιος κωδικας εφαρμογων

A.1 Merge sort

A.2 Παραλληλοποιημένη συγχώνευση (Merge)

A.3 Bitonic/Quick Sort

A.4 Γρήγορη ταξινόμηση (Quicksort)

A.5 Κατανεμημένος πολλαπλασιασμός πινάκων

A.6 Αναγνώριση προσώπων

A.1 Merge sort

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5
6 #include <torc.h>
7
8 #define N 1024 * 1024 * 32
9
10 #define NUM_NODES 8
11 #define NUM_CORES 4      // Used in Hybrid Parallelism
12 #define SPLT_THRESHOLD 3// Log2(NUM_NODES), provided N % NUM_NODES = 0
13
14
15 #define LOCAL_CUTOFF (N / (NUM_NODES * NUM_CORES * 2) ) // Split a bit further
16
17 void mergesort_tree_task (int *a,  int size, int depth);
18 void mergesort_local_task(int *a,  int size);
19 void merge_sort_serial   (int a[], int l,    int r);
20 void merge               (int a[], int l,    int m, int r);
21 void print_array         (int a[], int size);
22 int  check_sort          (int a[], int size);
23
24
25 int n_gl = N, qsort_enable = 0, omp_enable = 0;
26
```

```
27
28 int cmp_func(const void * a, const void * b) {
29    return ( *(int *) a - * (int *) b );
30 }
31
32
33
34 void merge(int arr[], int l, int m, int r){
35
36     int i, j, k; // Indexes for A, B, result array, respectively
37     int n1, n2;
38
39     int *a, *b;
40
41     n1 = m - l + 1;
42     n2 = r - m;
43
44     a = (int *) malloc (n1 * sizeof(int));
45     b = (int *) malloc (n2 * sizeof(int));
46
47
48     // Copy arr's subarray's contents to a and b arrays
49     for (i = 0; i < n1; i++)
50         a[i] = arr[l + i];
51     for (j = 0; j < n2; j++)
52         b[j] = arr[m + 1 + j];
53
54     i = 0, j = 0, k = l;
55
56
57     while (i < n1 && j < n2){
58
59         if (a[i] <= b[j])
60             arr[k] = a[i++];
61         else
62             arr[k] = b[j++];
63         k++;
64     }
65
66     // What's left of A
67     while (i < n1)
68         arr[k++] = a[i++];
69
70     // What's left of B
71     while (j < n2)
72         arr[k++] = b[j++];
73
74
75     free(a);
76     free(b);
77
78 }
79
80
81
82 void omp_mergesort(int a[], int size, int threads){
83
84     printf("omp_merge of size %d\n", size);
85
86     if (threads == 1)
87         qsort(a, size, sizeof(a[0]), cmp_func);
88     else if (threads > 1){
89
90         #pragma omp task
91         omp_mergesort(a, (size >> 1), (threads >> 1));
92
93         #pragma omp task
94         omp_mergesort(a + (size >> 1), size - (size >> 1),
95                                 threads - (threads >> 1));
96
97         #pragma omp taskwait
98
99         merge(a, 0, size / 2, size - 1); // Parallel merge can be used
100
101     }
102
103 }
104
105
106
```

```
107 void serial_sort(int *a, int size){
108 
109     if (qsort_enable)
110         qsort(a, size, sizeof(a[0]), cmp_func);
111     else if (omp_enable)
112         omp_mergesort(a, size, NUM_CORES);
113     else
114         merge_sort_serial(a, 0, size - 1);
115 }
116 
117 
118 
119 /* Split inside node to take advantage of multiple
120 CPU's */
121 #pragma ompix taskdef in(size) inout(a[size])
122 void mergesort_local_task(int *a, int size){
123 
124     int c, my_rank;
125 
126     // If we reached the task leaf (single core from now on)
127     if (size <= LOCAL_CUTOFF){
128 
129         serial_sort(a, size);
130 
131     }else{ // Must move down local tasktree
132 
133         MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
134 
135         #pragma ompix task atnode(my_rank)
136         mergesort_local_task(a,  (size >> 1));
137 
138         #pragma ompix task atnode(my_rank)
139         mergesort_local_task(a + (size >> 1), size - (size >> 1));
140 
141         #pragma ompix tasksync // Wait for above tasks to finish
142 
143         merge(a, 0, (size - 1)/ 2, size - 1);
144 
145 
146         if (c = check_sort(a, size)){
147             printf("Merge error\n");
148             exit(-1);
149         }
150 
151 
152     }
153 
154 
155     return;
156 }
157 
158 
159 
160 /* Create a merge sort tree across nodes */
161 #pragma ompix taskdef in(size, depth) inout(a[size])
162 void mergesort_tree_task(int *a, int size, int depth){
163 
164     int c, n0, n1, my_rank;
165 
166     // If we reached leaf
167     if (depth == SPLT_THRESHOLD){
168 
169         //printf("Depth = %d\n", depth);
170         MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
171 
172         mergesort_local_task(a, size);
173 
174     }else{ // Must move down task tree
175 
176         MPI_Comm_rank(MPI_COMM_WORLD, &n0); // Self
177         n1 = n0 + (1 << (depth));
178 
179         printf("About to split to nodes %d and %d\n", n0, n1);
180 
181 
182         #pragma ompix task atnode(here)
183         mergesort_tree_task(a,  (size >> 1), depth + 1);
184 
185 
186         #pragma ompix task atnode(n1)
```

```
187         mergesort_tree_task(a + (size >> 1), size - (size >> 1), depth + 1);
188
189
190         #pragma ompix tasksync // Wait for above tasks to finish
191
192         merge(a, 0, (size - 1)/ 2, size - 1);
193
194         if (c = check_sort(a, size)){
195             printf("Merge error\n");
196             exit(-1);
197         }
198
199
200     }
201
202
203     return;
204 }
205
206
207
208 // Reach this (serial) program when we're done splitting
209 void merge_sort_serial(int a[], int l, int r){
210
211     int m, my_rank;
212
213     if (l < r)
214     {
215         // Avoids overflow for large values
216         m = l + (r - l) / 2;
217
218         merge_sort_serial(a,     l, m);
219         merge_sort_serial(a, m + 1, r);
220
221         merge(a, l, m, r);
222     }
223
224
225 }
226
227
228
229 int main(int argc, char *argv[]){
230
231     int i, err, *A;
232     double start;
233
234     if (argc > 1)
235         if      (! strcmp("qsort", argv[1]) )
236             qsort_enable = 1;
237         else if (! strcmp("omp", argv[1]))
238             omp_enable    = 1;
239
240
241     A = (int *) malloc (N * sizeof (int));
242     for (i = 0; i < N; i++)
243         A[i] = rand() % 150000;
244
245
246     start = torc_gettime();
247
248     mergesort_tree_task(A, N, 0);
249
250     printf("Merge sort CPU Time = %.4lf sec\n", torc_gettime() - start);
251
252     return (0);
253 }
```

A.2 Παραλληλοποιημένη συγχώνευση (Merge)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 #include <torc.h>
6
7 #define N 1024 * 1024 * 1024
```

```
8
9  #define NUM_CORES 4
10 #define CUTOFF N / (NUM_CORES * 8)
11 int n_gl = N;
12
13
14
15 int binary_search(int *arr, int low, int high, int elem)
16 {
17     int mid;
18
19     if (high + 1 > low) high = high + 1;
20     else high = low;
21
22     while (low < high){
23
24         mid = (low + high) >> 1;
25         if (arr[mid] < elem)
26             low = mid + 1;
27         else
28             high = mid;
29     }
30
31     return low;
32 }
33
34
35
36 void sequential_merge(int *to, int *temp, int lowX, int highX,
37                                           int lowY, int highY,
38                                           int lowTo)
39 {
40
41     int highTo;
42     highTo = lowTo + highX - lowX + highY - lowY + 1;
43
44     for (; lowTo <= highTo; lowTo++){
45
46         if      (lowX > highX)
47             to[lowTo] = temp[lowY++];
48         else if (lowY > highY)
49             to[lowTo] = temp[lowX++];
50         else
51         {
52             if (temp[lowX] < temp[lowY])
53                 to[lowTo] = temp[lowX++];
54             else
55                 to[lowTo] = temp[lowY++];
56         }
57
58     }
59
60     return;
61 }
62
63
64
65 #pragma ompix taskdef in(lowX, highX, lowY, highY) inout(to[n_gl], temp[n_gl])
66 void parallel_merge(int *to, int *temp, int lowX, int highX,
67                                        int lowY, int highY,
68                     int lowTo)
69 {
70     int midX, midY, midTo, lengthX, lengthY;
71
72
73     lengthX = highX - lowX + 1;
74     lengthY = highY - lowY + 1;
75
76
77     if (lengthX + lengthY <= CUTOFF)
78     {
79         sequential_merge(to, temp, lowX, highX, lowY, highY, lowTo);
80         return;
81     }
82
83     if (lengthX < lengthY)
84     {
85         parallel_merge(to, temp, lowY, highY, lowX, highX, lowTo);
86         return;
87     }
```

```
88
89     // Median if all elements are unique
90     midX = (lowX + highX) >> 1;
91
92     midY = binary_search(temp, lowY, highY, temp[midX]);
93
94     midTo      = lowTo + midX - lowX + midY - lowY;
95     to[midTo] = temp[midX];
96
97     #pragma ompix task
98     parallel_merge(to, temp, lowX, midX - 1, lowY, midY - 1, lowTo);
99
100    #pragma ompix task
101    parallel_merge(to, temp, midX + 1, highX, midY, highY, midTo + 1);
102
103    #pragma ompix tasksync
104
105    return;
106 }
107
108
109
110 int main(){
111
112     int i, *A, *R, *temp, status;
113     double start, stop;
114
115     A = (int *) malloc (N * sizeof(int));
116     R = (int *) malloc (N * sizeof(int));
117
118     temp = (int *) malloc (N * sizeof(int));
119
120     for (i = 0; i < (N >> 1); i++){
121         A[i]       = (i + 1) * 2;
122         A[i + N/2] = (i + 1) * 3;
123     }
124
125     start = torc_gettime();
126
127     parallel_merge(R, A, 0, (N >> 1) - 1, (N >> 1), N - 1, 0);
128
129     stop = torc_gettime();
130
131
132     printf("CPU Time = %.4lf sec\n", stop - start);
133
134     free(A);
135     free(R);
136     return (0);
137 }
```

A.3 Bitonic/Quick Sort

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #include <torc.h>
6
7  #define N 1024 * 1024 * 8
8  #define NUM_NODES 8
9  #define NUM_CORES 4
10
11 #define PLUS 0
12 #define MINUS 1
13
14
15 int qsort_enable = 1;
16
17
18 int cmp_func_plus  (const void * a, const void * b) {
19    return ( *(int *) a - * (int *) b );
20 }
21
22 int cmp_func_minus (const void * a, const void * b) {
23    return ( *(int * ) a - *(int * ) b );
24 }
```

```
#pragma ompix taskdef in(cnt, flag) inout(A[cnt])
void bitonic_merge(int *A, int cnt, int flag)
{

    int i, tmp;
    int y = cnt / 2;

    if (cnt == 1) return;

    for (i = 0; i < y; i++){
        if (flag == PLUS && A[i] > A[i + y]){
            tmp      = A[i];
            A[i]     = A[i + y];
            A[i + y] = tmp;
        }
        else if (flag == MINUS && A[i] < A[i + y]){
            /*swap*/
            tmp = A[i];
            A[i]     = A[i + y];
            A[i + y] = tmp;
        }
    }

    bitonic_merge(A,     y, flag);
    bitonic_merge(A + y, y, flag);
}






#pragma ompix taskdef  in(cnt) inout (A[cnt]) in(flag)
void bitonic_sort(int *A, int cnt, int flag)
{

    int i, y, num_workers;

    if (cnt == 1) return;

    y =  cnt / 2;

    if (y >= N / ((NUM_NODES * NUM_CORES )) ){

        num_workers = torc_num_workers();

        #pragma ompix task atnode(here)
        bitonic_sort(A,     y, PLUS);

        #pragma ompix task atnode(here)
        bitonic_sort(A + y, y, MINUS);

        #pragma ompix tasksync

    }else{

        // Split completed, sort serially
        if (qsort_enable)
        {
            if (flag)
                qsort(A, cnt, sizeof(A[0]), cmp_func_plus);
            else
                qsort(A, cnt, sizeof(A[0]), cmp_func_minus);

        }else{
            bitonic_sort(A,     y, PLUS);
            bitonic_sort(A + y, y, MINUS);
        }

    }

    bitonic_merge(A, cnt, flag);

    return;
}


```

```
105 int main(int argc, char *argv[])
106 {
107
108     int i, node;
109     int *A;
110     int status, flag, step;
111
112     double start, stop;
113
114
115     A = (int *) malloc (N * sizeof(int));
116
117     for(i = 0; i < N; i++)
118         A[i] = rand() % 1000000;
119
120     step = N / (NUM_NODES);
121
122     /* First phase */
123     start = torc_gettime();
124
125     for(i = 0, node = 0, flag = 0; i < N; i += step,
126                                           node++,
127                                           flag ^= 1){
128
129         #pragma ompix task atnode(node)
130         bitonic_sort(A + i, step, flag);
131
132     }
133
134     #pragma ompix tasksync
135
136     // Second phase
137     step *= 2;
138
139     while (step <= N){
140         flag = PLUS;
141
142         for (i = 0; i < N; i += step, flag ^= 1){
143
144             #pragma ompix task
145             bitonic_merge(A + i, step, flag);
146
147         }
148
149         #pragma ompix tasksync
150         step *= 2;
151     }
152
153     stop = torc_gettime();
154
155     status = 0;
156     for (i = 1; i < N; i++){
157
158         if (A[i] < A[i - 1]){
159             status = i;
160             break;
161         }
162
163     }
164
165     printf("CPU Time = %.4lf sec\n", stop - start);
166
167     free(A);
168
169     return (0);
170 }
```

A.4 Γρήγορη ταξινόμηση (Quicksort)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <torc.h>
5
6
7 #define N 1024 * 1024 * 32
8
```

```c
#define NUM_CORES 4
#define LOCAL_CUTOFF (N / (NUM_CORES * 8) )

void quicksort(int *a, int size);


#pragma ompix taskdef in(len) inout(A[len])
void quicksort(int *A, int len) {

    int pivot, i, j, temp;

    if (len < 2) return;

    pivot = A[len >> 1];


    for (i = 0, j = len - 1; ; i++, j--){

        while (A[i] < pivot) i++;
        while (A[j] > pivot) j--;

        if (i >= j) break;

        temp = A[i];
        A[i]     = A[j];
        A[j]     = temp;
    }

    #pragma ompix task if (len >= LOCAL_CUTOFF)
    quicksort(A, i);

    #pragma ompix task if (len >= LOCAL_CUTOFF)
    quicksort(A + i, len - i);

    #pragma ompix tasksync

}



int main(){

    int i, status;
    int *A;
    double start, stop;

    A = (int *) malloc (N * sizeof(int));
    for (i = 0; i < N; i++)
        A[i] = rand() % 1500000;

    start = torc_gettime();

    quicksort(A, N);

    stop = torc_gettime();

    printf("CPU Time = %.4lf sec\n", stop - start);


    free(A);
    return 0;
}
```

A.5 Κατανεμημένος πολλαπλασιασμός πινάκων

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.h>

#include <torc.h>

#define N 1024       // Matrix size
#define NTASK M*M    // Number of submatrices
#define L 4          // Number of worker threads (optimally equal to number of physical cores,
                     // or physical threads if multithreading technology is present)
#define NUM_OF_TESTS 1
```

```
13
14
15 //Function & data declarations
16 int **A, *A_data, **B, *B_data, **C, *C_data;
17 int *res_data, **res; // Reusable across tasks
18
19
20 int readmat(char *fname, int *mat, int n),
21     writemat(char *fname, int *mat, int n);
22
23 int M, S; // Must inform every node of these dimensions
24
25 void taskexecute(int wid, int *results);
26 void *threadworker(void *arg);
27
28 //Global variable declaration
29 int taskid = 0;      //Next task's id
30 int ntask;
31
32 #pragma ompix taskdef in(m,s) inout(A_input[1024 * 1024], B_input[1024 * 1024])
33 void node_instat(int m, int s, int *A_input, int *B_input)
34 {
35     int i;
36     M = m;
37     S = s;
38
39     A_data = (int *) malloc(N * N * sizeof(int));
40     B_data = (int *) malloc(N * N * sizeof(int));
41
42
43     A = (int **) malloc (N * sizeof(int *));
44     B = (int **) malloc (N * sizeof(int *));
45
46     for (i = 0; i < N; i++)
47     {
48         A[i] = (int *) &(A_data[N * i]);
49         B[i] = (int *) &(B_data[N * i]);
50     }
51
52     for (i = 0; i < N * N; i++)
53     {
54         A_data[i] = A_input[i];
55         B_data[i] = B_input[i];
56     }
57
58     res_data = (int *) malloc (S * S * sizeof(int));
59     res = (int **) malloc (S * sizeof (int *));
60
61     return;
62 }
63
64 int **A, *A_data, **B, *B_data, **C, *C_data;
65 int *res_data, **res; // Reusable across tasks
66
67
68
69 #pragma ompix taskdef
70 void node_free_mem()
71 {
72     free(A[0]);
73     free(A);
74     free(B[0]);
75     free(B);
76
77     free(res[0]);
78     free(res);
79 }
80
81
82 #pragma ompix taskdef in (wid) out(results[S * S])
83 void taskexecute(int wid, int *results){
84
85     int i, j, k, x, y, outer_limit, inner_limit, r_i = 0;
86     int sum = 0;
87
88
89     x           = wid / M;
90     y           = wid % M;
91     outer_limit = (x + 1) * S;  //Computed here once, so 'for' loops do not
92     inner_limit = (y + 1) * S;  //have to perform these multiplications
```

```
93                                  //on each iteration.
94
95
96      for (i = x * S; i < outer_limit; i++){
97          for (j = y * S; j < inner_limit; j++){
98
99              for (k = 0, sum = 0; k < N; k++)
100                 sum += A[i][k] * B[k][j];
101
102
103             results[r_i++] = sum;
104         }
105     }
106
107
108     return;
109
110 }
111 {
112     // Callback function: Copy retrieved results to final array
113     int i, j, x, y, outer_limit, inner_limit, r_i = 0;
114
115     x           = wid / M;
116     y           = wid % M;
117     outer_limit = (x + 1) * S;
118     inner_limit = (y + 1) * S;
119
120     for (i = x * S; i < outer_limit; i++)
121
122         for (j = y * S; j < inner_limit; j++)
123
124             C[i][j] = results[r_i++];
125
126 }
127
128
129
130 int main(int argc, char *argv[]){
131
132     int i, task_num, **master_res;
133     double start_time, end_time, test_times[NUM_OF_TESTS], avg_time = 0;
134
135
136     A_data = (int *) malloc (N * N * sizeof(int));
137     B_data = (int *) malloc (N * N * sizeof(int));
138     C_data = (int *) malloc (N * N * sizeof(int));
139
140     A = (int **) malloc (N * sizeof(int *));
141     B = (int **) malloc (N * sizeof(int *));
142     C = (int **) malloc (N * sizeof(int *));
143
144     for (i = 0; i < N; i++)
145     {
146         A[i] = (int *) &(A_data[N * i]);
147         B[i] = (int *) &(B_data[N * i]);
148         C[i] = (int *) &(C_data[N * i]);
149     }
150
151     if (argc == 1){
152         printf("Submatrix size must be entered in command "
153         "line arguments (32, 64, or 256)\n");
154         return (1);
155     }
156
157     S = atoi(argv[1]);
158
159     // M must be               4, 16, 32
160     // Therefore, S must be 256, 64, 32, respectively.
161
162     if (S != 256 && S != 64 && S != 32){
163         printf("Acceptable submatrix sizes: 32, 64, 256\n");
164         return (1);
165     }
166
167     M = N / S;
168
169     ntask  = NTASK;
170     // Need this many result arrays
171     master_res = (int **) malloc (ntask * sizeof(int *));
172     for (i = 0; i < ntask; i++)
```

```
173         master_res[i] = (int *) malloc (S * S * sizeof(int));
174 
175     /* Read matrices from files: "A_file", "B_file"
176      */
177     if (readmat("Amat1024", (int *) A_data, N) < 0)
178         exit( 1 + printf("file problem\n") );
179     if (readmat("Bmat1024", (int *) B_data, N) < 0)
180         exit( 1 + printf("file problem\n") );
181 
182 
183 
184     // Inform all worker nodes of the decided upon task dimensions
185     for (i = 1; i < torc_num_nodes(); i++)
186     {
187         #pragma ompix task atnode(i)
188         node_instat(M, S, A_data, B_data);
189     }
190     #pragma ompix tasksync
191 
192     start_time = torc_gettime();
193 
194 
195     while(1){
196 
197         task_num = taskid++;
198 
199         if (task_num >= ntask)
200             break;          //All tasks created
201 
202 
203         #pragma ompix task
204         taskexecute(task_num, master_res[task_num]);
205 
206     }
207     #pragma ompix tasksync
208 
209 
210     end_time      = torc_gettime();
211     test_times[i] = end_time - start_time;
212     avg_time     += test_times[i];
213 
214 
215     // No need to wait for them at this time, we can wait right before we finish
216     for (i = 1; i < torc_num_nodes(); i++)
217     {
218         #pragma ompix task atnode(i)
219         node_free_mem();
220     }
221 
222     for (i = 0; i < NUM_OF_TESTS; ++i)
223         printf("Test #%d: %lf seconds\n", i + 1, test_times[i]);
224 
225     printf("[S = %d] Average time: %f\n", S, avg_time / (double) NUM_OF_TESTS);
226 
227 
228 
229     /* Save result in "Cmat1024Results"
230      */
231     writemat("Cmat1024Results", (int *) C_data, N);
232 
233     for (i = 0; i < ntask; i++)
234         free(master_res[i]);
235     free(master_res);
236 
237     // First line frees the data, second one the int * array we used.
238     free(A[0]);
239     free(A);
240     free(B[0]);
241     free(B);
242     free(C[0]);
243     free(C);
244 
245 
246     #pragma ompix tasksync
247     return (0);
248 }
249 
250 
251 
252 #define _mat(i,j) (mat[(i)*n + (j)])
```

```
253
254
255 int readmat(char *fname, int *mat, int n){
256     FILE *fp;
257     int  i, j;
258
259     if ((fp = fopen(fname, "r")) == NULL)
260         return (-1);
261     for (i = 0; i < n; i++){
262         for (j = 0; j < n; j++){
263             if (fscanf(fp, "%d", &_mat(i,j)) == EOF){
264                 fclose(fp);
265                 return (-1);
266             };
267         }
268     }
269     fclose(fp);
270     return (0);
271 }
272
273
274
275 int writemat(char *fname, int *mat, int n){
276     FILE *fp;
277     int  i, j;
278
279     if ((fp = fopen(fname, "w")) == NULL)
280         return (-1);
281     for (i = 0; i < n; i++, fprintf(fp, "\n")){
282         for (j = 0; j < n; j++)
283             fprintf(fp, " %d", _mat(i, j));
284     }
285     fclose(fp);
286     return (0);
287 }
```

A.6 Αναγνώριση προσώπων

Λόγω της μεγάλης έκτασης του πηγαίου κώδικα της εφαρμογής, θα συμπεριλάβουμε μονάχα κάποια τμήματα του πηγαίου μας κώδικα, τον οποίο θα συγκρίνουμε με τον αρχικό της [9].

A.6.1 main.c

Αρχικός κώδικας με OpenMP

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <stdlib.h>
5
6 #if defined(_OPENMP)
7 #include <omp.h>
8 #endif
9
10 #include "LpiImage.h"
11
12 extern int callbackStillImage(struct lpiImage* image, int *X, int *Y, int *H, int *W, float *O);
13 extern int callbackFrameImage(struct lpiImage *image, int *X, int *Y, int *H, int *W, float *O);
14 extern void InitCFF();
15
16 extern double my_gettime(void);
17
18 #define MAX_IMAGES  256
19 #define MAX_FACES   64
20
21 int main(int argc, char *argv[])
22 {
```

```
23      FILE *fp;
24      char buf[512], name[512];
25      int count = 0;
26      struct lpiImage *src[MAX_IMAGES] = {0};
27      int num_threads = 1;
28      int nested = 0;
29      double t1, t2, t3;
30      int i, j;
31
32      if (argc == 2)
33          num_threads = atoi(argv[1]);
34
35      if (argc == 3) {
36          num_threads = atoi(argv[1]);
37          nested = atoi(argv[2]);
38      }
39
40      InitCFF();
41      fp = fopen("list.txt", "r");
42
43
44      t1 = my_gettime();
45      while( fgets(buf, sizeof(buf), fp) != NULL )
46      {
47          strcpy(name,"");
48          sscanf(buf, "%s", name);
49
50          src[count] = lpiImage_loadPGM(name);
51          count++;
52          if (count == MAX_IMAGES) {
53              printf("too many images...\n");
54              exit(1);
55          }
56      }
57
58      t2 = my_gettime();
59      #if defined(_OPENMP)
60      printf("Running with %d threads and nested %s\n", num_threads, (nested)?"ENABLED":"DISABLED"
        );
61      omp_set_nested(nested);
62      omp_set_num_threads(num_threads);
63      #endif
64
65      #if defined(PARBATCH)
66      #pragma omp parallel for schedule(dynamic,1)
67      #endif
68      for (i = 0; i < count; i++) {
69          int X[MAX_FACES], Y[MAX_FACES], H[MAX_FACES], W[MAX_FACES];
70          float O[MAX_FACES];
71
72          int nofFaces=callbackStillImage(src[i], X, Y, H, W, O);
73              for(j = 0; j < nofFaces; j++)
74          {
75              printf("FACE[%d]: %d\t%d\t%d\t%d\t%f\n", j, X[j], Y[j], H[j], W[j], O[j]);
76          }
77
78          printf("%d : Faces = %d\n", i, nofFaces);
79          lpiImage_lpiReleaseImage(&src[i]);
80
81      }
82      t3 = my_gettime();
83
84      printf("TIME FOR LOADING IMAGES = %f\n", t2-t1);
85      printf("TIME FOR FACE DETECTION = %f\n", t3-t2);
86      printf("TOTAL EXECUTION TIME    = %f\n", t3-t1);
87
88      exit(0);
89  }
```

Αντίστοιχος κώδικας με HOMPI

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <stdlib.h>
5
6 #include "LpiImage.h"
```

```
7
8  extern int callbackStillImage(struct lpiImage* image, int *X, int *Y, int *H, int *W, float *O);
9  extern int callbackFrameImage(struct lpiImage *image, int *X, int *Y, int *H, int *W, float *O);
10 extern void InitCFF();
11
12 extern double my_gettime(void);
13
14 #define MAX_IMAGES  256
15 #define MAX_FACES   64
16
17
18
19 int max_faces = MAX_FACES;
20 #include <torc.h>
21
22 #pragma ompix taskdef
23 void torc_instat(){
24     printf("Node #%d instantiating CNN...\n", torc_node_id());
25     InitCFF();
26 }
27
28
29
30 #pragma ompix taskdef inout(par_name[512]) in(seqno)
31 void torc_remote_imagetask(char *par_name, int seqno){
32
33     int i;
34
35     struct lpiImage *src;
36     char name[512];
37
38     int X[MAX_FACES], Y[MAX_FACES], H[MAX_FACES], W[MAX_FACES];
39     float O[MAX_FACES];
40     int nofFaces;
41
42     printf("\n\nNode #%d\n", torc_node_id());
43
44     name[0] = '\0';
45     sscanf(par_name, "%s", name);
46
47
48     src = lpiImage_loadPGM(par_name);
49
50     nofFaces = callbackStillImage(src, X, Y, H, W, O);
51
52
53     for (i = 0; i < nofFaces; i++){
54         printf("FACE[%d]: %d\t%d\t%d\t%d\t%f\n",
55                i, X[i], Y[i], H[i], W[i], O[i]);
56     }
57
58     // Can alternatively print to file
59     printf("#%d : Torc node %d found %d faces\n",
60            seqno, torc_node_id(), nofFaces);
61     lpiImage_lpiReleaseImage(&src);
62
63 }
64
65
66 int main(int argc, char *argv[]){
67
68     FILE *fp;
69     char buf[512], name[512];
70     struct lpiImage *src[MAX_IMAGES] = {0};
71     int num_threads = 1;
72     int nested = 0;
73     double t1, t2, t3;
74     int i, j, num_nodes, my_node;
75     char **name_cpies;
76
77     j = 0;
78
79     if (argc == 2)
80         num_threads = atoi(argv[1]);
81
82     if (argc == 3) {
83         num_threads = atoi(argv[1]);
84         nested = atoi(argv[2]);
85     }
86
```

```
87      InitCFF();
88      fp = fopen("list.txt", "r");
89
90      name_cpies = (char **) malloc(170 * sizeof (char *));
91
92      // Instantiate convolutional network on each process
93      num_nodes = torc_num_nodes();
94      for (i = 0; i < num_nodes; i++){
95          #pragma ompix task atnode(i)
96          torc_instat();
97      }
98
99      #pragma ompix tasksync
100
101
102     t1 = my_gettime();
103
104     while( fgets(buf, sizeof(buf), fp)!=NULL ){
105
106         name[0] = '\0';
107         strcpy(name, buf);
108         name[strlen(name) - 1 ] = '\0';
109
110
111         name_cpies[j] = (char *) malloc (512 * sizeof(char));
112         strcpy(name_cpies[j], name);
113
114         #pragma ompix task
115         torc_remote_imagetask(name_cpies[j], j);
116
117         j++;
118
119     }
120
121     t2 = my_gettime();
122
123
124     #pragma ompix tasksync
125
126     t3 = my_gettime();
127
128     printf("TIME FOR LOADING IMAGES = %f\n", t2-t1);
129     printf("TIME FOR FACE DETECTION = %f\n", t3-t2);
130     printf("TOTAL EXECUTION TIME    = %f\n", t3-t1);
131
132     for (i = 0 ; i < 170; i++)
133         free(name_cpies[i]);
134
135     free(name_cpies);
136
137     return (0);
138 }
```

A.6.2 lolac.c

Αρχικός κώδικας πρώτης φάσης με OpenMP

```
1       #pragma omp parallel for private(S) schedule(dynamic,1)
2       for(iter = 0; iter< itotal; iter++){
3
4           int i, j;
5           int width = img_source->width;
6           int height = img_source->height;
7           int nbS;
8
9           {
10          int h1;
11          int w1;
12
13          struct lpiImage *img_tmp;
14          unsigned char *p1;
15          float *p2;
16
17          int current_width, current_height, total_size;
18
```

```
19          float *pp;
20
21          nbS = iter;
22          S = Sm[iter];
23
24
25          h1 = (int)floor((double)height*((double)SH/(double)S));
26          w1 = (int)floor((double)width*((double)SH/(double)S));
27
28
29          if(h1 < 36 || w1 < 32) {
30              printf("this should not happen!\n");
31              exit(1);
32              continue;
33          }
34
35          data->imgt[nbS] = lpiImage_lpiCreateImage(w1, h1, sizeof(unsigned char));
36
37              //CONVOLVE
38              lpiImage_lpiResize(img_input, data->imgt[nbS]);
39
40
41
42          img_tmp = lpiImage_lpiCreateImage(w1, h1, sizeof(float));
43
44
45          p1 = (unsigned char *) data->imgt[nbS]->imageData;
46          p2 = (float *) img_tmp->imageData;
47          for(i = 0; i < h1; i++)
48              for(j = 0; j < w1; j++)
49                  p2[i * w1 + j] = (float)( ((int)p1[i*data->imgt[nbS]->width+j]) - 128)/((float)
    128.0);
50
51              pp=CConvolver_ConvolveRoughlyStillImage(&data->cconv[nbS], img_tmp, w1, h1, &
    current_width, &current_height, &total_size, S);
52
53
54
55          //CHECK THE RESULTS
56          for(i=0;i<total_size;i++)
57          {
58              float output = pp[i];
59              int y;
60              int x;
61
62              int ycenter;
63              int xcenter;
64
65              if (output <= (float) 0.0)
66                  continue;
67
68              y = i/current_width;
69              x = i%current_width;
70
71              ycenter = (int)floor((double)(4*y*S)/(double)SH) + (int)floor((double)S/2.0);
72              xcenter = (int)floor((double)(4*x*S)/(double)SH) + (int)floor((double)S*SR/2.0);
73
74
75
76              #if defined(_OPENMP)
77              omp_set_lock(&lock);
78              #endif
79
80              data->Xs[facesFound]=xcenter;
81              data->Ys[facesFound]=ycenter;
82              data->heights[facesFound]=S;
83              data->outputs[facesFound]=output;
84              facesFound++;
85
86              #if defined(_OPENMP)
87              omp_unset_lock(&lock);
88              #endif
89
90          }
91
92          //FREE IMAGE
93          lpiImage_lpiReleaseImage(&data->imgt[nbS]);
94          CConvolver_DeallocateOutput(&data->cconv[nbS]);
95
96          lpiImage_lpiReleaseImage(&img_tmp);
```

```


        }
    }
```

Αντίστοιχος κώδικας με HOMPI

Η τεχνική που εφαρμόζουμε για την παραλληλοποίηση κάθε βρόγχου ο οποίος υλοποιεί τις φάσεις της ανάλυσης κάθε εικόνας είναι η εξής. Δημιουργούμε μία συνάρτηση (στο παράδειγμα μας την torc_phase1), στην οποία μεταφέρουμε τον κώδικα του αντίστοιχου βρόχου OpenMP. Η όλη διαδικασία αυτή πραγματοποιείται “με το χέρι“. Ως συνέπεια, είναι δική μας ευθύνη να μελετήσουμε την εφαρμογή και να λύσουμε προβλήματα εξάρτησης δεδομένων. Όπως παρατηρούμε παρακάτω, έπρεπε να προστεθούν οι κατάλληλες μεταβλητές στον κώδικα της νέας συνάρτησης, οι οποίες προσπελαύνονται στον κώδικά που μεταφέραμε.

```c
#pragma ompix taskdef in(iter, itotal) inout(img_source_ptr[1],  img_input_ptr[1], facesFound
      [1], data_ptr[1], Sm[64])
void torc_phase1(int iter, int itotal,
    void *img_source_ptr, void *img_input_ptr,
    void *data_ptr, int *Sm, int *facesFound
)
{
    int i, S, SH = 36;
    float SR = 36.0 / 32.0; // ((float) SW / ((float)SH);
    struct per_image_data *data;
    struct lpiImage       *img_source, *img_input;

    img_source = (struct lpiImage *)    img_source_ptr;
    img_input  = (struct lpiImage *)    img_input_ptr;

    data = (struct per_image_data *) data_ptr;


    // Single iteration (64 in total in our case)
    //for(iter = 0; iter < itotal; iter++)
    {
        int i, j, S; // private(S)
        int width = img_source->width;
        int height = img_source->height;
        int nbS;

        {
        int h1;
        int w1;

        struct lpiImage *img_tmp;
        unsigned char *p1;
        float *p2;

        int current_width, current_height, total_size;

        float *pp;

        nbS = iter;
        S = Sm[iter];


        h1 = (int)floor((double)height*((double)SH/(double)S));
        w1 = (int)floor((double)width*((double)SH/(double)S));


        if(h1 < 36 || w1 < 32) {
            printf("this should not happen!\n");
            printf("iter = %d\n", iter);
            exit(1);
            //continue; used in loop
```

```
51          }
52
53          data->imgt[nbS] = lpiImage_lpiCreateImage(w1, h1, sizeof(unsigned char));
54              //CONVOLVE
55
56              lpiImage_lpiResize(img_input, data->imgt[nbS]);
57
58
59
60          img_tmp = lpiImage_lpiCreateImage(w1, h1, sizeof(float));
61
62
63          p1 = (unsigned char *)data->imgt[nbS]->imageData;
64          p2 = (float *)img_tmp->imageData;
65          for(i=0;i<h1;i++)
66              for(j=0;j<w1;j++)
67                  p2[i*w1+j]=(float)( ((int)p1[i*data->imgt[nbS]->width+j]) - 128)/((float)128.0);
68
69
70              pp=CConvolver_ConvolveRoughlyStillImage(&data->cconv[nbS], img_tmp, w1, h1, &
        current_width, &current_height, &total_size, S);
71
72
73          //CHECK THE RESULTS
74          for(i=0;i<total_size;i++)
75          {
76              float output=pp[i];
77              int y;
78              int x;
79
80              int ycenter;
81              int xcenter;
82
83              if (output<=(float)0.0)
84                  continue;
85
86              y=i/current_width;
87              x=i%current_width;
88
89              ycenter = (int)floor((double)(4*y*S)/(double)SH) + (int)floor((double)S/2.0);
90              xcenter = (int)floor((double)(4*x*S)/(double)SH) + (int)floor((double)S*SR/2.0);
91
92
93              data->Xs[*facesFound]=xcenter;
94              data->Ys[*facesFound]=ycenter;
95              data->heights[*facesFound]=S;
96              data->outputs[*facesFound]=output;
97              (*facesFound)++;
98
99          }
100
101         //FREE IMAGE
102         lpiImage_lpiReleaseImage(&data->imgt[nbS]);
103         CConvolver_DeallocateOutput(&data->cconv[nbS]);
104
105         lpiImage_lpiReleaseImage(&img_tmp);
106         //nbS++;
107
108
109         }
110     }
111
112 }
```

Στην συνέχεια καλούμε αυτή την συνάρτηση κατάλληλα, στην θέση όπου βρισκόταν ο κώδικας που μεταφέρθηκε σε αυτήν, δηλώνοντας με το clause "here" ότι επιθυμούμε αυτό το task να εκτελεσθεί στον τρέχων κόμβο. Ο παρακάτω κώδικας αντικαθιστά τον κώδικα A.6.2

```
1      for (iter = 0; iter < itotal; iter++)
2      {
3          #pragma ompix task atnode(here)
4          torc_phase1( iter, itotal,
5                  (void *) img_source, (void *) img_input,
```

```
6                   (void *) data, &Sm, &facesFound);
7       }
8
9       #pragma ompix tasksync
```

Η μέθοδος μετατροπής αυτή είναι όμοια για την δεύτερη και τρίτη φάση.

A.6.3 Convolver.c

Αρχικός κώδικας πρώτου βρόχου με OpenMP

```
1       #ifdef L2PAR
2       #pragma omp parallel for schedule(static,1)
3       #endif
4       for(i = 0; i < 4; i++)
5       {
6       #ifdef L2PARTASK
7           #pragma omp task firstprivate(i) shared(width, height, CConvolver_v, current_width,
        current_height)
8       #endif
9           {
10          struct lpiImage *fm0_tmp;
11
12          fm0_tmp = CConvolver_CreateImage(width - 4, height - 4);
13
14          CConvolver_v->fm0Sub[i] = CConvolver_CreateImage(current_width, current_height);
15
16          CConvolver_Convolve(img, CConvolver_v->m_cnn->m_kernels0[i].kern, CConvolver_v->m_cnn->
        m_kernels0[i].bias, 5, fm0_tmp);
17
18          CConvolver_SubSample(fm0_tmp, CConvolver_v->m_cnn->m_kernels0[i].coeff, CConvolver_v->
        m_cnn->m_kernels0[i].sbias, 0, CConvolver_v->fm0Sub[i]);
19
20          lpiImage_lpiReleaseImage(&fm0_tmp);
21          }
22
23      }
24      #ifdef L2PARTASK
25      #pragma omp taskwait
26      #endif
```

Αντίστοιχος κώδικας με HOMPI

Για την μετατροπή του κώδικα ακολουθούμε την ίδια μέθοδο που δείξαμε και στην μετατροπή των βρόχων της κάθε φάσης (A.6.2). Και σε αυτή την περίπτωση υπήρχε ανάγκη για τον κατάλληλο χειρισμό των απαραίτητων μεταβλητών της συνάρτησης.

```
1 #pragma ompix taskdef in(i, width, height, current_width, current_height) inout(CConvolver_v_ptr
      [1], img_ptr[1])
2 void torc_conv_task0(int i, int width, int height,
3                      void *CConvolver_v_ptr, void *img_ptr, // CConvolver *, lpiImage *,
      respectively
4                      int current_width, int current_height)
5 {
6     struct lpiImage *fm0_tmp;
7     struct lpiImage *img;
8     struct CConvolver *CConvolver_v;
9
10    img = (struct lpiImage *) img_ptr;
11    CConvolver_v  = (struct CConvolver *) CConvolver_v_ptr;
12
13
14    fm0_tmp=CConvolver_CreateImage(width - 4, height - 4);
15
16
17    CConvolver_v->fm0Sub[i]=CConvolver_CreateImage(current_width, current_height);
18
```

```
19
20     CConvolver_Convolve(img, CConvolver_v->m_cnn->m_kernels0[i].kern, CConvolver_v->m_cnn->
       m_kernels0[i].bias, 5, fm0_tmp);
21
22     CConvolver_SubSample(fm0_tmp, CConvolver_v->m_cnn->m_kernels0[i].coeff, CConvolver_v->m_cnn
       ->m_kernels0[i].sbias, 0, CConvolver_v->fm0Sub[i]);
23
24     lpiImage_lpiReleaseImage(&fm0_tmp);
25 }
```

Ο αρχικός κώδικας αντικαταστάθηκε από τις κλήσεις της παραπάνω συνάρτησης, όπως φαίνεται παρακάτω.

```
1      for(i = 0; i < 4; i++){
2
3      #ifdef L2PARTASK_LO1
4          #pragma ompix task untied
5          torc_conv_task0(i, width, height, (void *) CConvolver_v, (void *) img, current_width,
       current_height);
6      #else
7
8          struct lpiImage *fm0_tmp;
9
10         fm0_tmp=CConvolver_CreateImage(width-4, height-4);
11
12         CConvolver_v->fm0Sub[i]=CConvolver_CreateImage(current_width, current_height);
13
14         CConvolver_Convolve(img, CConvolver_v->m_cnn->m_kernels0[i].kern, CConvolver_v->m_cnn->
       m_kernels0[i].bias, 5, fm0_tmp);
15
16         CConvolver_SubSample(fm0_tmp, CConvolver_v->m_cnn->m_kernels0[i].coeff, CConvolver_v->
       m_cnn->m_kernels0[i].sbias, 0, CConvolver_v->fm0Sub[i]);
17
18         lpiImage_lpiReleaseImage(&fm0_tmp);
19
20     #endif
21
22     }
```

Στην περίπτωση αυτήν δίνεται η δυνατότητα της μεταγλώττισης είτε του αρχικού προγράμματος είτε της δικής μας υλοποίησης, με την χρήση των κατάλληλων οδηγιών κατά την μεταγλώττιση.

Παραρτημα B

Περιβαλλον πειραματων

B.1 Απαιτήσεις λογισμικού

B.2 Sun Cluster

B.3 EC2 Cluster

B.4 Opti7020

B.5 Προσωπικός Η/Υ

B.6 Raspberry Pi 4

B.1 Απαιτήσεις λογισμικού

Για την μεταγλώττιση και χρήση του HOMPI framework, έχουμε τις εξής απαιτήσεις σε λογισμικό:

- automake: automake v1.11
- libtool: libtool v2.4.6
- autoconf
- flex
- bison
- hwloc, libhwloc-dev (προαιρετικό)

- gcc
- Υλοποίηση MPI: Ένα από τα παρακάτω

  openmpi v3.0 ή μεταγενέστερη

  mpich v3.0 ή μεταγενέστερη

B.2 Sun Cluster

O Sun Cluster του τμήματος Μηχανικών Η/Υ & Πληροφορικής του Πανεπιστημίου αποτελείται από 16 κόμβους τύπου Sun Fire X4100. Οι κόμβοι είναι διασυνδεδεμένοι μεταξύ τους με δίκτυο 802.3 με εύρος ζώνης 1 Gigabit.

Κάθε κόμβος έχει 2 επεξεργαστές τύπου AMD Opteron 275 2 πυρήνων, με ρυθμό ρολογιού 2.2GHz και 12GB μνήμης ανά κόμβο.

Η υλοποίηση του προτύπου MPI που χρησιμοποιήθηκε από την βιβλιοθήκη χρόνο εκτέλεσης είναι η OpenMPI, έκδοση 4.0.1.

Ως λειτουργικό σύστημα χρησιμοποιήθηκε το Ubuntu, έκδοση 14.04.

B.3 EC2 Cluster

Ο εικονικός cluster των AWS που χρησιμοποιήθηκε αποτελείται από 4 κόμβους τύπου C5n XLarge.

Κάθε εικονικός επεξεργαστής είναι τύπου Intel Xeon Scalable (Skylake) με ρυθμό ρολογιού μέχρι 3.6GHz και 10.5GiB μνήμης. Ο κάθε κόμβος διαθέτει 4 τέτοιους εικονικούς επεξεργαστές. Χρησιμοποιήθηκε η τεχνολογία ENA (Elastic Network Adapter) για αύξηση των επιδόσεων του δικτύου, του οποίου ο ονομαστικός ρυθμός μετάδοσης ανέρχεται στα 25Gbps. Ο cluster αυτός ρυθμίσθηκε με την χρήση της εφαρμογής AWS ParallelCluster.

Η υλοποίηση του προτύπου MPI που χρησιμοποιήθηκε από την βιβλιοθήκη χρόνο εκτέλεσης είναι η MPICH, έκδοση `3.1.4`.

Η εικόνα λειτουργικού συστήματος (AMI, Amazon Machine Image) βασίζεται στο λειτουργικό σύστημα Ubuntu, έκδοση `16.04`.

B.4 Opti7020

Το σύστημα Opti7020 έχει έναν επεξεργαστή τύπου Intel i5-4590 με 4 πυρήνες, με βασικό ρυθμό ρολογιού 3.3GHz με Turbo Boost έως 3.7Ghz. Ο κόμβος κατέχει 8GB μνήμης. Ως λειτουργικό σύστημα χρησιμοποιήθηκε η έκδοση 14.04 του λειτουργικού Ubuntu.

B.5 Προσωπικός Η/Υ

Το εν λόγω σύστημα έχει έναν επεξεργαστή τύπου Intel i5-4460 με 4 πυρήνες, με βασικό ρυθμό ρολογιού 3.2GHz με Turbo Boost έως 3.4Ghz. Ο κόμβος κατέχει 16GB μνήμης. Ως λειτουργικό σύστημα χρησιμοποιήθηκε η έκδοση 9 του λειτουργικού Debian.

B.6 Raspberry Pi 4

Το σύστημα αυτό είναι ένας single board υπολογιστής, βασισμένος στην αρχιτεκτονική ARM. Ο 64-bit Cortex-A72 επεξεργαστής αρχιτεκτονικής ARM v8 κατέχει 4 πυρήνες με ρυθμό ρολογιού 1.5GHz. Το σύστημα διαθέτει 4GB μνήμης. Ως λειτουργικό σύστημα χρησιμοποιήθηκε η έκδοση του Σεπτεμβρίου 2019 του λειτουργικού Raspbian, βασισμένη στο λειτουργικό Debian.