Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη υποδομής για ανάπτυξη, βελτιστοποίηση και απεικόνιση εφαρμογών σε παράλληλες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ Ε. ΤΖΑΝΕΤΤΗΣ

Επιβλέπων : Γεώργιος Γκούμας
Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μελέτη υποδομής για ανάπτυξη, βελτιστοποίηση και απεικόνιση εφαρμογών σε παράλληλες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ Ε. ΤΖΑΝΕΤΤΗΣ

Επιβλέπων : Γεώργιος Γκούμας
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Νοεμβρίου 2019.

.....	.....	.....
Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2019

Ιωάννης Ε. Τζανεττής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Πρόσφατες εξελίξεις σε αλγορίθμους με υψηλό υπολογιστικό φόρτο σε διάφορους τομείς όπως η επεξεργασία εικόνας, η δυναμική ελέγχου ροής και η μηχανική μάθηση παρουσιάζουν μαζικά αυξημένη ανάγκη για υπολογιστική ισχύ. Από την άλλη πλευρά, το High Performance Computing βρίσκεται υπό συνεχή ανάπτυξη, προσφέροντας προηγμένες αρχιτεκτονικές υλικού και παράλληλα μοντέλα προγραμματισμού και πρωτόκολλα που αυξάνουν την απόδοση και μειώνουν το κόστος τέτοιων εργασιών. Ωστόσο, η εκμετάλλευση αυτών των τεχνολογιών είναι διαθέσιμη μόνο σε εξειδικευμένους χρήστες, ενώ οι μέσοι προγραμματιστές μπορούν να ασχοληθούν μόνο με ένα περιορισμένο αριθμό τεχνολογιών που έχουν αναπτυχθεί αρκετά ώστε να περιλαμβάνουν μια απλή διεπαφή ικανή να αποκρύπτει την υψηλή πολυπλοκότητα.

Σε αυτή τη διπλωματική εργασία, μελετήσαμε μια σειρά διαφορετικών τεχνολογιών που μπορούν να συνδυαστούν για να διευκολύνουν την ανάπτυξη, βελτιστοποίηση και εκτέλεση μιας παράλληλης εφαρμογής. Εστιάσαμε στο σχεδιασμό της αρχιτεκτονικής μιας υποδομής, η οποία παίρνει τη μορφή μιας εργαλειοθήκης που προσφέρει μια ποικιλία διαφορετικών λειτουργιών. Η χρήση μιας τέτοιας υποδομής στοχεύει να βοηθήσει τον χρήστη να υλοποιήσει εφαρμογές υψηλής απόδοσης, ικανές να αναπτυχθούν σε διάφορες αρχιτεκτονικές και χώρους μνήμης.

Ένα παράλληλο Προγραμματιστικό Μοντέλο έχει σχεδιαστεί για την καθοδήγηση του χρήστη μέσω της διαδικασίας σχεδιασμού εφαρμογής. Αυτό το μοντέλο αποκρύπτει τον προγραμματιστή από την εφαρμογή χαμηλού επιπέδου, φροντίζοντας όλες τις εργασίες επικοινωνίας και εκτέλεσης. Η ολοκλήρωση και η εκτέλεση της εφαρμογής είναι ευθύνη του εργαλείου Διαχείρισης Εκτέλεσης που έχει αναπτυχθεί για την εφαρμογή των οδηγιών χαμηλού επιπέδου και τη χρήση εξωτερικών βιβλιοθηκών. Η Προγραμματιστική Διεπαφή αποτελεί μέρος του μοντέλου προγραμματισμού και αποτελείται κυρίως από ένα σύνολο βιβλιοθηκών που αναπτύχθηκαν για να επιτρέπουν την επικοινωνία μεταξύ των επιμέρους κομματιών της εφαρμογής. Ένα εργαλείο Αυτόματης Παραλληλοποίησης είναι το αποτέλεσμα μιας εκτεταμένης μελέτης των σύγχρονων τεχνικών που χρησιμοποιούνται στον τομέα και εκμεταλλεύεται το σχεδιασμό του Προγραμματιστικού Μοντέλου για την αυτόματη παραλληλοποίηση μερών της εφαρμογής που περιλαμβάνουν μεγάλο υπολογιστικό φόρτο. Ένα εργαλείο λήψης αποφάσεων όσον αφορά τις διαθέσιμες τεχνικές που υποστηρίζονται, εκμεταλλεύεται τις οδηγίες του χρήστη για την ανάθεση τμημάτων της εφαρμογής στα διάφορα στοιχεία υλικού και επιλέγει τις αντίστοιχες τεχνολογίες για την υλοποίηση των λειτουργιών επικοινωνίας και εκτέλεσης χαμηλού επιπέδου που απαιτούνται.

Οι κύριοι στόχοι αυτής της εργασίας μπορούν να συνοψιστούν σε τρία μέρη, συγκεκριμένα στα κεφάλαια 3, 4 και 5. Στο πρώτο μέρος συζητάμε τις τρέχουσες τεχνολογίες παραλληλοποίησης, εστιάζοντας ειδικά στην αξιολόγηση της απόδοσης του παραλληλοποιητή PLuTo και του μεταγλωττιστή ROSE, εργασία για την κάλυψη περισσότερων περιπτώσεων και βελτιστοποίηση της απόδοσης. Στο δεύτερο μέρος, περιγράφουμε λεπτομερώς το παράλληλο προγραμματιστικό μοντέλο που χρησιμοποιείται για την χρήση των εργαλείων. Τέλος, στο τρίτο μέρος, συζητάμε τις λεπτομέρειες ολοκλήρωσης μιας τέτοιας υποδομής (AEOLUS) και πώς η αλληλεπίδρασή των μερών της μπορεί να προσφέρει ως αποτέλεσμα μια ολοκληρωμένη και βελτιστοποιημένη παράλληλη εφαρμογή.

Λέξεις κλειδιά

Παραλληλοποίηση, Αυτόματη Παραλληλοποίηση, Παράλληλα Συστήματα, Προγραμματιστικό Μοντέλο

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή μου, Γεώργιο Γκούμα, για όλη την πολύτιμη βοήθειά του κατά τη διάρκεια αυτής της περιόδου. Όχι μόνο μου έδωσε κατευθυντήριες γραμμές και υποστήριξη για το θέμα αυτής της διατριβής, αλλά και με εισήγαγε στο πεδίο των λειτουργικών και παράλληλων συστημάτων μέσω των πολλαπλών μαθημάτων που μελετούσα υπό την επίβλεψή του. Το ενδιαφέρον και ο ενθουσιασμός του με οδήγησαν να αντιμετωπίσω το έργο μου με τον ίδιο τρόπο.

Επιπλέον, θα ήθελα να ευχαριστήσω την εταιρεία WINGS ICT Solutions και το πρόγραμμα PHANTOM Horizon 2020, στο οποίο εργάστηκα, παρέχοντας παράλληλα ένα μεγάλο μέρος αυτής της εργασίας για την ανάπτυξή του. Συγκεκριμένα, θα ήθελα να ευχαριστήσω τους προϊστάμενούς μου Παναγιώτη Βλαχέα και Παναγιώτη Δεμέστιχα, που με υποστήριξαν με τις συμβουλές τους, την κατανόηση και την εμπιστοσύνη τους καθ ’όλη αυτή την περίοδο.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω όλους τους φίλους μου που με υποστήριξαν και μου έδωσαν κίνητρο όλο αυτό το διάστημα και ιδιαίτερα τους Κωνσταντίνο, Χάρη και Στάθη με τους οποίους περάσαμε μαζί τα τελευταία χρόνια της φοίτησής μας. Θα ήθελα επίσης να εκφράσω τη βαθύτατη ευγνωμοσύνη μου για τη μητέρα μου, τον πατέρα μου και τα αδέλφια μου, που έμειναν στο πλευρό μου, υποστηρίζοντας και συμβουλεύοντάς με. Ευχαριστώ ιδιαιτέρως την Ελένη, που ήταν πάντα εκεί υπομονετικά για να με βοηθήσει και να με στηρίξει με οποιονδήποτε τρόπο.

Μέσα από αυτές τις γραμμές, θα ήθελα να αναγνωρίσω όλους όσους με βοήθησαν να ολοκληρώσω αυτό το έργο κατά τη διάρκεια αυτής της έντονης περιόδου της ζωής μου με την ηθική τους υποστήριξη και συμβουλές.

Ιωάννης Ε. Τζανεττής,

Αθήνα, 11η Νοεμβρίου 2019

Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Υπολογιστικών Συστημάτων, Νοέμβριος 2019.

URL: `http://artemis.cslab.ece.ntua.gr/`

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Σήμερα, οι υπολογιστικές εργασίες τείνουν να γίνονται ολοένα και πιο απαιτητικές. Η άνοδος της μηχανικής μάθησης, της επιστήμης των δεδομένων και της επεξεργασίας εικόνας, καθώς και η ζήτηση σε εφαρμογές σε πραγματικό χρόνο, δείχνουν σαφώς την ανάγκη για περισσότερη υπολογιστική ισχύ και μειωμένη κατανάλωση ενέργειας. Εκεί γίνεται αντιληπτή η αξία των ετερογενών συστημάτων. Η τρέχουσα και μελλοντική έρευνα εστιάζεται εκτενώς στην ικανότητα συνδυασμού της ισχύος υπολογισμού των συστημάτων υψηλής απόδοσης (π.χ. συστάδες μηχανών) με χαμηλή κατανάλωση μικρότερων, ενεργειακά αποδοτικών συσκευών (π.χ. ενσωματωμένες συσκευές, FPGAs κ.λ.π.) και στην αξιοποίησή της για την προώθηση της ανάπτυξης συστημάτων υψηλής απόδοσης ικανών να εκτελούν βέλτιστα πολλών ειδών εφαρμογές.

Από την άλλη πλευρά, η ανάπτυξη παράλληλων εφαρμογών για διαφορετικές συσκευές είναι μια χρονοβόρα και δύσκολη διαδικασία, που δεν προορίζεται για τον μέσο προγραμματιστή που γνωρίζει μόνο από σειριακό προγραμματισμό. Πολλές βιβλιοθήκες και υποδομές έχουν αναπτυχθεί για να υποστηρίξουν τον παράλληλο προγραμματισμό χρησιμοποιώντας πολλαπλά νήματα, διεργασίες, πυρήνες ή διαφορετικούς επιταχυντές. Ωστόσο, η ανάπτυξη μιας πολυδιάστατης εφαρμογής που μπορεί να εκμεταλλευτεί διαφορετικά επίπεδα μιας παράλληλης αρχιτεκτονικής, συμπεριλαμβανομένων των εξωτερικών επιταχυντών, απαιτεί εξοικείωση με πολλαπλές υποδομές και μπορεί να οδηγήσει σε υπερβολική προσπάθεια για τον προγραμματιστή, απαιτώντας εκτεταμένη αποσφαλμάτωση και αναδιαμόρφωση της εφαρμογής για προσαρμογή στην υπάρχουσα αρχιτεκτονική.

Παρόλο που ο σχεδιασμός ετερογενών παράλληλων εφαρμογών είναι ένα περίπλοκο και πολυδιάστατο έργο, ένα μεγάλο μέρος της διαδικασίας του μπορεί να αυτοματοποιηθεί, με την προϋπόθεση ότι υπάρχει μερική καθοδήγηση από τον χρήστη. Μια υποδομή που μπορεί να διευκολύνει το σχεδιασμό παράλληλων εφαρμογών θα πρέπει να υποστηρίζει διαφορετικές γλώσσες και μοντέλα προγραμματισμού, να μπορεί να εκμεταλλεύεται διαφορετικές αρχιτεκτονικές, παρέχοντας παράλληλα ένα μοντέλο προγραμματισμού υψηλού επιπέδου που απομακρύνει τον χρήστη από τις υλοποιήσεις χαμηλού επιπέδου.

1.2 Στόχοι

Η ιδέα πίσω από μια υποδομή που μπορεί να διευκολύνει και να ενισχύσει τη διαδικασία της παράλληλης ανάπτυξης εφαρμογών αποσκοπεί στην αφαίρεση των υψηλών επιπέδων δυσκολίας που υπάρχουν όταν έχουμε να κάνουμε με σύνθετες αρχιτεκτονικές συστημάτων. Διαφορετικά περιβάλλοντα σημαίνουν διαφορετικά στυλ προγραμματισμού, διεπαφές και ακόμη και γλώσσες, οι οποίες σε πολλές περιπτώσεις καθιστούν την ετερογενή ανάπτυξη πολύ περίπλοκη για τον μέσο προγραμματιστή. Ένας ενδιάμεσος "πράκτορας" μεταξύ του προγραμματιστή και του διαθέσιμου στοχευόμενου συστήματος θα πρέπει να αναλάβει τις υλοποιήσεις χαμηλού επιπέδου της εφαρμογής απαλλάσσοντάς τον από την ευθύνη της αντιμετώπισης της πολυπλοκότητας που δημιουργείται διαθέτοντας με αυτό τον τρόπο λειτουργίες παραλληλοποίησης σε ένα λιγότερο εξειδικευμένο κοινό.

Το πρόβλημα που θα αντιμετωπίσουμε βασίζεται στο σχεδιασμό και την αρχιτεκτονική μιας σει-

ράς εργαλείων που θα επιτρέψει στον χρήστη να σχεδιάσει μια παράλληλη εφαρμογή ακολουθώντας συγκεκριμένες κατευθυντήριες γραμμές καθώς και την υλοποίηση των βασικών εργαλείων που αποτελούν τον πυρήνα της υποδομής, στα οποία στη συνέχεια μπορούν να προστεθούν κι άλλα ολοκληρώνοντας τη λειτουργικότητά της. Σκοπός της παρούσας εργασίας είναι να παράγει μια υποδομή που να υποστηρίζει το σχεδιασμό παράλληλων εφαρμογών που αποτελούνται από πολλά ανεξάρτητα μέρη που μπορούν να εκτελεστούν παράλληλα, καθώς και την ατομική παραλληλοποίησή τους και τελικά την ανάπτυξή τους σε μια κατανεμημένη και παράλληλη αρχιτεκτονική αποτελούμενη από πολλαπλούς υπολογιστικούς κόμβους με τις δικές τους δυνατότητες. Ο σχεδιασμός του μοντέλου προγραμματισμού θα γίνει με τέτοιο τρόπο ώστε να μπορεί να εξεταστεί και η ετερογενής αρχιτεκτονική, συμπεριλαμβανομένων των επιταχυντών (π.χ. GPUs) ή των FPGAs, εάν υπάρχουν τέτοιες συσκευές.

Κεφάλαιο 2

Γενικές πληροφορίες

2.1 Παράλληλες Αρχιτεκτονικές

Για να ξεπεραστεί η επίπτωση της παρακμής του Νόμου του Moore, η επιστημονική κοινότητα εργάζεται όλο και περισσότερο για την εκμετάλλευση πολλαπλών πυρήνων / επεξεργαστών / μηχανών για την εκτέλεση εργασιών παράλληλα. Η ταχύτητα των μεμονωμένων επεξεργαστών δεν μπορεί να αυξηθεί τόσο γρήγορα όσο πριν, αλλά η συλλογική εργασία των υπολογιστικών συσκευών στοχεύει να καλύψει αυτό το κενό και να αναλάβει την υποστήριξη των όλο και πιο απαιτητικών υπολογιστικών εργασιών που περιλαμβάνουν οι σύγχρονοι αλγόριθμοι. Οι παράλληλοι υπολογιστές μπορούν να ταξινομηθούν κατά προσέγγιση σύμφωνα με το επίπεδο στο οποίο το υλικό υποστηρίζει παραλληλισμό. Αυτή η ταξινόμηση είναι σε γενικές γραμμές ανάλογη με την απόσταση μεταξύ των βασικών κόμβων υπολογιστών. Οι τύποι αυτοί δεν είναι αμοιβαίως αποκλεισμένοι, για παράδειγμα, συστάδες συμμετρικών πολυεπεξεργαστών είναι σχετικά κοινές.

2.1.1 Πολυπύρηνα Συστήματα

Ένας επεξεργαστής πολλαπλών πυρήνων είναι ένας επεξεργαστής που περιλαμβάνει πολλαπλές μονάδες επεξεργασίας (που ονομάζονται ”πυρήνες”) στο ίδιο τσιπ. Αυτός ο επεξεργαστής διαφέρει από έναν υπεργολάβο επεξεργαστή, ο οποίος περιλαμβάνει πολλαπλές μονάδες εκτέλεσης και μπορεί να εκδώσει πολλαπλές οδηγίες ανά κύκλο ρολογιού από μία ροή εντολών (thread). Αντιθέτως, ένας επεξεργαστής πολλαπλών πυρήνων μπορεί να εκδώσει πολλαπλές οδηγίες ανά κύκλο ρολογιού από πολλαπλές ροές εντολών. Κάθε πυρήνας σε έναν επεξεργαστή πολλαπλών πυρήνων μπορεί ενδεχομένως να είναι και υπερσκληρικός - δηλαδή, σε κάθε κύκλο ρολογιού, κάθε πυρήνας μπορεί να εκδώσει πολλαπλές οδηγίες από ένα νήμα.

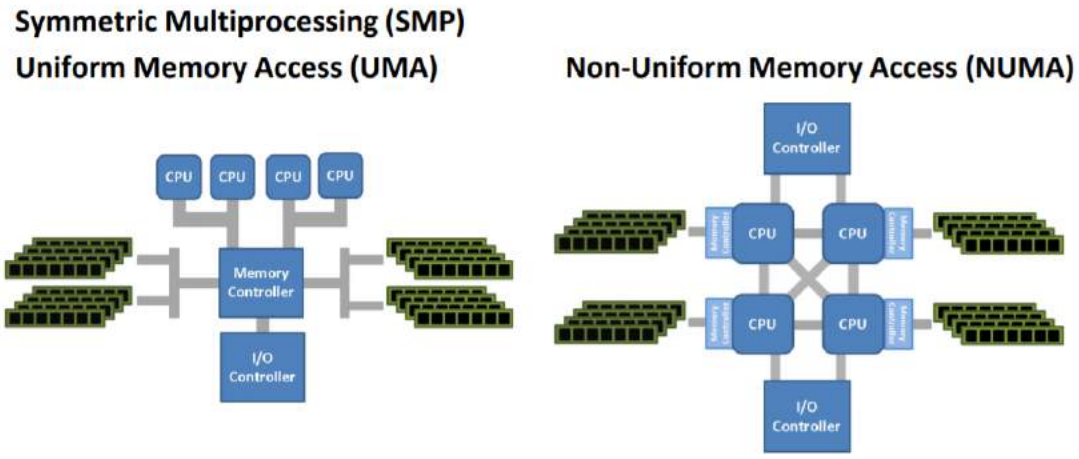
Ο ταυτόχρονος πολυ-νηματισμός (π.χ. Hyper-Threading της Intel) ήταν μια πρώιμη μορφή ψευδο-πολυ-πυρηνισμού. Ένας επεξεργαστής ικανός για ταυτόχρονο πολυ-νηματισμό περιλαμβάνει πολλαπλές μονάδες εκτέλεσης στην ίδια μονάδα επεξεργασίας (δηλ. Έχει μια υπερσύγχρονη αρχιτεκτονική) και μπορεί να εκδίδει πολλαπλές οδηγίες ανά κύκλο ρολογιού από πολλαπλά νήματα. Η χρονική πολλαπλή επεξεργασία περιλαμβάνει μια ενιαία μονάδα εκτέλεσης στην ίδια μονάδα επεξεργασίας και μπορεί να εκδώσει μια εντολή κάθε φορά από πολλαπλά νήματα.

2.1.2 Αρχιτεκτονικές Κοινής Μνήμης

Ένας συμμετρικός πολυεπεξεργαστής (SMP) είναι ένα σύστημα υπολογιστή με πολλαπλούς ταυτόσημους επεξεργαστές που μοιράζονται τη μνήμη και συνδέονται μέσω ενός διαύλου. Το πρόβλημα του διαύλου αποτρέπει την κλιμάκωση των αρχιτεκτονικών διαύλων. Ως αποτέλεσμα, τα SMP γενικά δεν περιλαμβάνουν περισσότερους από 32 επεξεργαστές. Λόγω του μικρού μεγέθους των επεξεργαστών και της σημαντικής μείωσης των απαιτήσεων για το εύρος ζώνης του διαύλου που επιτυγχάνονται με μεγάλες κρυφές μνήμες, αυτοί οι συμμετρικοί πολυεπεξεργαστές είναι εξαιρετικά οικονομικοί, με την προϋπόθεση ότι υπάρχει επαρκές εύρος ζώνης μνήμης.

Τα SMP χρησιμοποιούν Αρχιτεκτονικές Ομοιόμορφα Κατανεμημένης Μνήμης (UMA) που σημαίνει ότι όλοι οι επεξεργαστές έχουν πρόσβαση στη μνήμη με το ίδιο εύρος ζώνης και την ίδια καθυ-

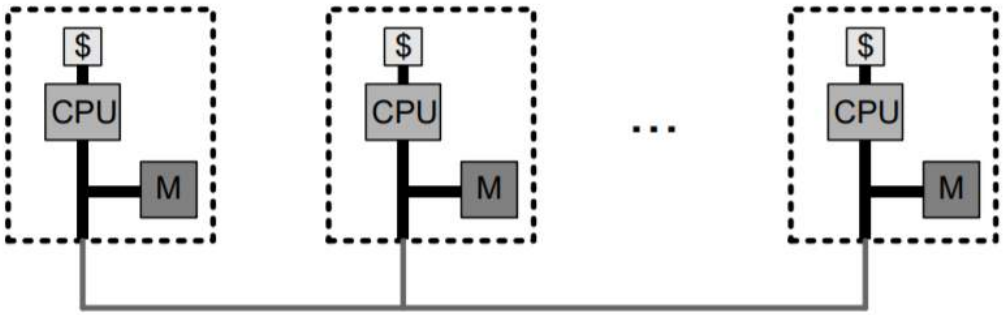
στέρηση. Υπάρχουν μηχανές που δεν ακολουθούν την ίδια προσέγγιση, όπου η πρόσβαση στη μνήμη γίνεται ανομοιόμορφα. Τα μηχανήματα που χρησιμοποιούν αρχιτεκτονική μη ομοιόμορφης μνήμης (NUMA) εμφανίζουν διαφορετική καθυστέρηση όταν οι διαφορετικοί επεξεργαστές έχουν πρόσβαση στη μνήμη.



Σχήμα 2.1: SMP και NUMA αρχιτεκτονικές

2.1.3 Αρχιτεκτονικές Κατανεμημένης Μνήμης

Ένας κατανεμημένος υπολογιστής (επίσης γνωστός ως πολυεπεξεργαστής κατανεμημένης μνήμης) είναι ένα σύστημα υπολογιστή κατανεμημένης μνήμης στο οποίο τα στοιχεία επεξεργασίας συνδέονται μέσω ενός δικτύου. Οι κατανεμημένοι υπολογιστές είναι εξαιρετικά επεκτάσιμοι. Οι όροι ”παράλληλος υπολογισμός”, ”ταυτόχρονος υπολογισμός” και ”κατανεμημένος υπολογισμός” έχουν πολλές αλληλεπικαλύψεις και δεν υπάρχει σαφής διάκριση μεταξύ τους. Το ίδιο σύστημα μπορεί να χαρακτηριστεί και ως ”παράλληλο” και ”κατανεμημένο”, οι επεξεργαστές σε ένα τυπικό κατανεμημένο σύστημα εκτελούνται ταυτόχρονα παράλληλα.

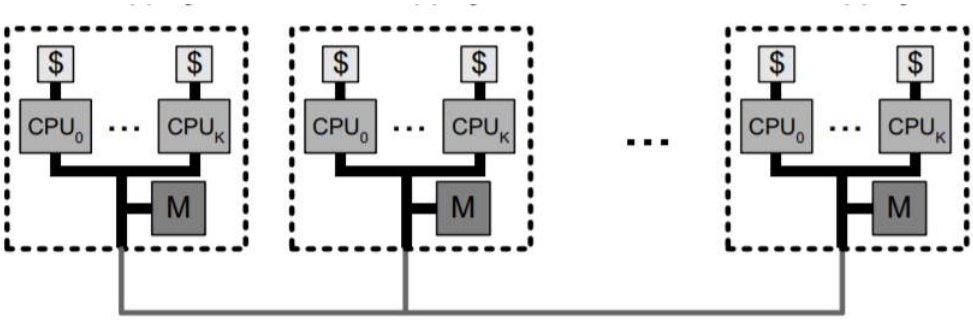


Σχήμα 2.2: Αρχιτεκτονικές κατανεμημένης μνήμης

Οι κατανεμημένες αρχιτεκτονικές περιλαμβάνουν επίσης υβριδικές λύσεις που χρησιμοποιούν πολλαπλά SMPs συνδεδεμένα μεταξύ τους μέσω δικτύου.

2.1.4 Υπολογιστικά συστήματα συμπλέγματος

Ένα σύμπλεγμα (cluster) είναι μια ομάδα από χαλαρά συζευγμένους υπολογιστές που συνεργάζονται στενά, έτσι ώστε από κάποιες απόψεις να μπορούν να θεωρηθούν ως ένας μόνο υπολογιστής. Τα



Σχήμα 2.3: Υβριδικές αρχιτεκτονικές

clusters αποτελούνται από πολλαπλά ανεξάρτητα μηχανήματα που συνδέονται με ένα δίκτυο. Ενώ τα μηχανήματα σε ένα σύμπλεγμα δεν χρειάζεται να είναι συμμετρικά, η εξισορρόπηση φορτίου είναι πιο δύσκολη αν δεν είναι. Ο συνηθέστερος τύπος cluster είναι το σύμπλεγμα Beowulf, το οποίο είναι ένα σύμπλεγμα που εφαρμόζεται σε πολλούς ίδιους εμπορικούς υπολογιστές εκτός του ράφι που συνδέονται με ένα τοπικό δίκτυο TCP / IP Ethernet. Η τεχνολογία Beowulf αναπτύχθηκε αρχικά από τον Thomas Sterling και τον Donald Becker. Το 87% όλων των υπερυπολογιστών Top500 είναι clusters.

Επειδή τα συστήματα υπολογιστών πλέγματος (grid) μπορούν εύκολα να χειριστούν υπερβολικά παράλληλα προβλήματα, τα σύγχρονα συμπλέγματα είναι συνήθως σχεδιασμένα για να χειρίζονται πιο δύσκολα προβλήματα - προβλήματα που απαιτούν κόμβους να μοιράζονται συχνότερα τα ενδιάμεσα αποτελέσματα. Αυτό απαιτεί υψηλό εύρος ζώνης και, το σημαντικότερο, δίκτυο διασύνδεσης χαμηλής καθυστέρησης. Πολλοί ιστορικοί και τρέχοντες υπερυπολογιστές χρησιμοποιούν προσαρμοσμένο υλικό δικτύου υψηλής απόδοσης ειδικά σχεδιασμένο για υπολογιστές συμπλέγματος, όπως το δίκτυο Cray Gemini. Από το 2014, οι περισσότεροι σύγχρονοι υπερυπολογιστές χρησιμοποιούν ειδικό τυποποιημένο υλικό δικτύου, συχνά Myrinet, InfiniBand ή Gigabit Ethernet.

2.1.5 Μονάδες επεξεργασίας γραφικών

Μια μονάδα επεξεργασίας γραφικών (GPU) είναι ένα εξειδικευμένο ηλεκτρονικό κύκλωμα που έχει σχεδιαστεί για να χειρίζεται γρήγορα και να τροποποιεί τη μνήμη για να επιταχύνει τη δημιουργία εικόνων σε ένα προσωρινό buffer που προορίζεται για έξοδο σε μια συσκευή απεικόνισης. Οι μονάδες GPU χρησιμοποιούνται σε ενσωματωμένα συστήματα, κινητά τηλέφωνα, προσωπικούς υπολογιστές, σταθμούς εργασίας και κονσόλες παιχνιδιών. Οι σύγχρονες μονάδες GPU είναι πολύ αποδοτικές στο χειρισμό γραφικών υπολογιστών και επεξεργασίας εικόνας. Η εξαιρετικά παράλληλη δομή τους καθιστά πιο αποδοτικές από τις κεντρικές μονάδες επεξεργασίας γενικής χρήσης (CPU) για αλγόριθμους που επεξεργάζονται παράλληλα μεγάλα τμήματα δεδομένων. Σε έναν προσωπικό υπολογιστή, μια GPU μπορεί να είναι παρούσα σε μια κάρτα γραφικών ή ενσωματωμένη στη μητρική πλακέτα. Σε ορισμένες CPU, είναι ενσωματωμένες στο CPU chip.

2.1.6 Συσκευές προγραμματιζόμενες στο πεδίο

FPGA είναι ένα ολοκληρωμένο κύκλωμα σχεδιασμένο για να διαμορφώνεται από έναν πελάτη ή έναν σχεδιαστή μετά την κατασκευή - εξ ου και ο όρος ”field-programmable”. Η διαμόρφωση FPGA καθορίζεται γενικά χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL), παρόμοια με εκείνη που χρησιμοποιείται για ένα ολοκληρωμένο κύκλωμα ειδικής εφαρμογής (ASIC). Τα διαγράμματα κυκλωμάτων χρησιμοποιήθηκαν στο παρελθόν για να καθορίσουν τη διαμόρφωση, αλλά αυτό καθίσταται όλο και πιο σπάνιο λόγω της εμφάνισης εργαλείων αυτοματισμού ηλεκτρονικού σχεδιασμού.

Τα FPGA περιέχουν μια σειρά προγραμματιζόμενων λογικών μπλοκ και μια ιεραρχία ”επαναπροσδιοριζόμενων διασυνδέσεων” που επιτρέπουν την ”συρμάτωση” των μπλοκ, όπως πολλές πύλες λογικής που μπορούν να διασυνδεθούν σε διαφορετικές διαμορφώσεις. Τα λογικά μπλοκ μπορούν

να διαμορφωθούν έτσι ώστε να εκτελούν πολύπλοκες συνδυαστικές λειτουργίες ή απλά απλές λογικές πύλες όπως οι AND και XOR. Στις περισσότερες FPGAs, τα λογικά μπλοκ περιλαμβάνουν επίσης στοιχεία μνήμης, τα οποία μπορεί να είναι απλά flip-flops ή πιο ολοκληρωμένα μπλοκ μνήμης. Πολλά FPGAs μπορούν να επαναπρογραμματιστούν για να εφαρμόσουν διαφορετικές λογικές λειτουργίες, επιτρέποντας ευέλικτους υπολογισμούς που εκτελούνται σε λογισμικό υπολογιστή.

2.2 Μοντελοποίηση ανάλυσης

Βέλτιστα, η επιτάχυνση από παραλληλισμό θα ήταν γραμμική-διπλασιασμός του αριθμού των στοιχείων επεξεργασίας θα πρέπει να μειώσει κατά το ήμισυ τον χρόνο εκτέλεσης και διπλασιασμός για δεύτερη φορά θα πρέπει να μειώσει και πάλι κατά το ήμισυ το χρόνο εκτέλεσης. Ωστόσο, πολύ λίγοι παράλληλοι αλγόριθμοι επιτυγχάνουν τη βέλτιστη επιτάχυνση. Οι περισσότεροι από αυτούς έχουν μια σχεδόν γραμμική ταχύτητα για μικρό αριθμό στοιχείων επεξεργασίας, η οποία συγκλίνει σε μια σταθερή τιμή για μεγάλο αριθμό στοιχείων επεξεργασίας.

Η πιθανή επιτάχυνση ενός αλγορίθμου σε μια παράλληλη υπολογιστική πλατφόρμα δίνεται από τον νόμο Amdahl:

$$S_{latency} = \frac{1}{1 - p + p/s} \tag{2.1}$$

όπου $S_{latency}$ είναι η πιθανή επιτάχυνση της καθυστέρησης της εκτέλεσης ολόκληρου του έργου. s είναι η επιτάχυνση της καθυστέρησης της εκτέλεσης του παράλληλου μέρους του έργου. p είναι το ποσοστό του χρόνου εκτέλεσης ολόκληρης της εργασίας που αφορά το παράλληλο μέρος της εργασίας πριν από την παραλληλισμό.

Το γεγονός ότι $S_{latency}$ < 1/(1 - p), δείχνει ότι ένα μικρό μέρος του προγράμματος που δεν μπορεί να παραλληλιστεί θα περιορίσει τη συνολική ταχύτητα που υπάρχει από παραλληλισμό. Ένα πρόγραμμα επίλυσης ενός μεγάλου μαθηματικού ή μηχανικού προβλήματος θα αποτελείται τυπικά από πολλά παράλληλα τμήματα και από πολλά μη παράλληλα (σειριακά) μέρη. Εάν το μη παράλληλο μέρος ενός προγράμματος αντιστοιχεί σε 10% του χρόνου εκτέλεσης (p = 0.9), δεν μπορούμε να πετύχουμε περισσότερες από 10 φορές ταχύτητα, ανεξάρτητα από τον αριθμό των επεξεργαστών που προστίθενται. Αυτό θέτει ένα ανώτερο όριο στη χρησιμότητα της προσθήκης περισσότερων μονάδων παράλληλης εκτέλεσης. Όταν μια εργασία δεν μπορεί να χωριστεί λόγω διαδοχικών περιορισμών, η εφαρμογή περισσότερης προσπάθειας δεν έχει καμία επίδραση στο χρονοδιάγραμμα.

Ο νόμος του Amdahl ισχύει μόνο για περιπτώσεις όπου το μέγεθος του προβλήματος είναι σταθερό. Στην πράξη, καθώς διατίθενται περισσότεροι υπολογιστικοί πόροι, τείνουν να συνηθίσουν σε μεγαλύτερα προβλήματα (μεγαλύτερα σύνολα δεδομένων) και ο χρόνος που αφιερώνεται στο παράλληλο μέρος αυξάνεται πολύ ταχύτερα από την εγγενώς σειριακή εργασία. Στην περίπτωση αυτή, ο νόμος του Gustafson δίνει μια λιγότερο απαισιόδοξη και πιο ρεαλιστική εκτίμηση της παράλληλης απόδοσης:

$$S = 1 - p + sp \tag{2.2}$$

Ο νόμος του Amdahl και ο νόμος του Gustafson υποθέτουν ότι ο χρόνος εκτέλεσης του σειριακού μέρους του προγράμματος είναι ανεξάρτητος από τον αριθμό των επεξεργαστών. Ο νόμος του Amdahl υποθέτει ότι ολόκληρο το πρόβλημα είναι σταθερού μεγέθους έτσι ώστε το συνολικό ποσό εργασίας που πρέπει να γίνει παράλληλα να είναι επίσης ανεξάρτητο από τον αριθμό των επεξεργαστών, ενώ ο νόμος του Gustafson υποθέτει ότι το συνολικό ποσό εργασίας που πρέπει να γίνει παράλληλα ποικίλλει γραμμικά με τον αριθμό των επεξεργαστών.

2.3 Τεχνολογίες παραλληλοποίησης

Κατά την ανάπτυξη μιας παράλληλης εφαρμογής, ο προγραμματιστής πρέπει να λάβει υπόψη μια σειρά παραμέτρων που πρόκειται να καθορίσουν τα εργαλεία και τις τεχνικές που θα πρέπει να χρη-

σιμοποιήσουν για την υλοποίηση. Οι υπολογιστικοί πόροι και οι αρχιτεκτονικές που είναι διαθέσιμες περιορίζουν τις επιλογές που μπορούν να χρησιμοποιηθούν όσον αφορά τα μοντέλα προγραμματισμού που μπορούν να αξιοποιηθούν. Από την άλλη πλευρά, οι ανάγκες της εφαρμογής καθώς και η φύση της καθοδηγούν τον προγραμματιστή σε συγκεκριμένα μοντέλα που μπορεί να είναι πιο επωφελής για κάθε συγκεκριμένη περίπτωση. Στον πίνακα 2.1, κατηγοριοποιούμε τις διαφορετικές προσεγγίσεις που μπορούν να ακολουθηθούν και τα διάφορα πλεονεκτήματα / μειονεκτήματα που υπάρχουν για κάθε μία από αυτές.

		Architecture	
		Shared Memory	**Distributed Memory**
Programming Model	**Shared address space**	(1) + Easy implementation + Easy programming + High performance	(3) + Easy programming - Difficult implementation - Low performance
	Distributed address space	(2) + Easy implementation + High performance - Difficult programming	(4) + Easy implementation + High performance - Difficult programming

Πίνακας 2.1: Αρχιτεκτονικές και προγραμματιστικά μοντέλα

1. Τα στοιχεία επεξεργασίας που εκτελούν τη ροή έχουν άμεση πρόσβαση στην ίδια μνήμη και μπορούν επίσης να αναφέρονται σε διευθύνσεις μνήμης χρησιμοποιώντας τους ίδιους δείκτες. Το POSIX multi-threading είναι ένα τέτοιο παράδειγμα, όπου τα διαφορετικά νήματα χρησιμοποιούν την ίδια μνήμη και χώρο μνήμης.

2. Τα στοιχεία επεξεργασίας έχουν πρόσβαση στην ίδια μνήμη, αλλά δεν μπορούν να χρησιμοποιήσουν τις ίδιες αναφορές μνήμης (δείκτες), επειδή χρησιμοποιούν διαφορετικό χώρο μνήμης. Επομένως, πρέπει να χρησιμοποιούν διαφορετικούς μηχανισμούς για να μεταφέρουν δεδομένα διαφορετικά από τους δείκτες, εξ ου και η δυσκολία προγραμματισμού. Οι διαδικασίες UNIX χρησιμοποιούν την ίδια μνήμη, αλλά αναφέρονται σε διαφορετικό χώρο μνήμης.

3. Τα στοιχεία επεξεργασίας δεν έχουν πρόσβαση στην ίδια φυσική μνήμη, αλλά μπορούν να αναφέρονται στον ίδιο χώρο μνήμης. Δεδομένου ότι η πρόσβαση στη μνήμη είναι έμμεση, υπάρχει κόστος για την απόδοση, αλλά και για την υλοποίηση δεδομένου ότι θα πρέπει να υπάρχει ένα επίπεδο middleware που να χειρίζεται όλες τις λειτουργίες μνήμης.

4. Τα στοιχεία επεξεργασίας δεν έχουν πρόσβαση ούτε στην ίδια φυσική μνήμη ούτε στον ίδιο χώρο μνήμης. Έτσι, ο χρήστης πρέπει να εφαρμόσει κάθε επικοινωνία που είναι απαραίτητη για την εφαρμογή, κάνοντας τον προγραμματισμό ένα δύσκολο έργο όπως με τη χρήση διαδικασιών στον ίδιο χώρο μνήμης. Ένα παράδειγμα αυτής της προσέγγισης είναι η διεπαφή μετάδοσης μηνυμάτων (MPI) και οι υλοποιήσεις της (OpenMPI, MPICH).

2.3.1 Κοινώς χρησιμοποιούμενες τεχνολογίες

OpenMP

Το OpenMP (Open Multi-Processing) είναι μια διεπαφή προγραμματισμού εφαρμογών (API) που υποστηρίζει προγραμματισμό πολλαπλών επεξεργαστών μνήμης πολλαπλών πλατφορμών σε C, C ++ και Fortran στις περισσότερες πλατφόρμες, στις αρχιτεκτονικές συνόλων εντολών και στα λειτουργικά συστήματα, συμπεριλαμβανομένου του Linux και των Windows. Αποτελείται από ένα σύνολο οδηγιών μεταγλωττιστή, ρουτίνες βιβλιοθηκών και μεταβλητές περιβάλλοντος που επηρεάζουν

τη συμπεριφορά χρόνου εκτέλεσης. Χρησιμοποιεί ένα φορητό, κλιμακωτό μοντέλο που δίνει στους προγραμματιστές μια απλή και ευέλικτη διεπαφή για την ανάπτυξη παράλληλων εφαρμογών για πλατφόρμες που κυμαίνονται από τον τυπικό επιτραπέζιο υπολογιστή έως τον υπερυπολογιστή.

CUDA

Η CUDA (Compute Unified Device Architecture) είναι μια παράλληλη πλατφόρμα υπολογιστών και μοντέλο διεπαφής προγραμματισμού εφαρμογών (API) που δημιουργήθηκε από τη Nvidia. Επιτρέπει στους προγραμματιστές λογισμικού και στους μηχανικούς λογισμικού να χρησιμοποιούν μια μονάδα επεξεργασίας γραφικών με δυνατότητα CUDA (GPU) για γενική επεξεργασία - μια προσέγγιση που ονομάζεται GPGPU (υπολογισμός γενικής χρήσης σε μονάδες επεξεργασίας γραφικών). Η πλατφόρμα CUDA είναι ένα στρώμα λογισμικού που παρέχει άμεση πρόσβαση στο εικονικό σύνολο εντολών της GPU και σε παράλληλα υπολογιστικά στοιχεία για την εκτέλεση υπολογιστικών πυρήνων.

Η πλατφόρμα CUDA έχει σχεδιαστεί για να λειτουργεί με γλώσσες προγραμματισμού όπως C, C ++ και Fortran. Αυτή η προσβασιμότητα διευκολύνει τους ειδικούς στον παράλληλο προγραμματισμό να χρησιμοποιούν τους πόρους GPU, σε αντίθεση με τα προηγούμενα API όπως το Direct3D και το OpenGL, τα οποία απαιτούσαν προηγμένες δεξιότητες στον προγραμματισμό γραφικών. Επίσης, η CUDA υποστηρίζει υποδομές προγραμματισμού όπως το OpenACC και το OpenCL. Όταν παρουσιάστηκε για πρώτη φορά από τη Nvidia, το όνομα CUDA ήταν ένα αρκτικόλεξο για την Compute Unified Device Architecture, αλλά η Nvidia έπεσε στη συνέχεια την κοινή χρήση του ακρωνυμίου.

MPI

Το MPI είναι ένα τυποποιημένο και φορητό πρότυπο μετάδοσης μηνυμάτων που σχεδιάστηκε από μια ομάδα ερευνητών από τον ακαδημαϊκό κόσμο και τη βιομηχανία για να λειτουργήσει σε μια μεγάλη ποικιλία παράλληλων υπολογιστικών αρχιτεκτονικών. Το πρότυπο καθορίζει τη σύνταξη και τη σημασιολογία ενός πυρήνα ρουτίνες βιβλιοθηκών που είναι χρήσιμες σε ένα ευρύ φάσμα χρηστών που γράφουν φορητά προγράμματα πέρασμα μηνυμάτων σε C, C ++ και Fortran. Υπάρχουν αρκετές δοκιμασμένες και αποδοτικές εφαρμογές του MPI, πολλές από τις οποίες είναι ανοιχτού κώδικα ή στο δημόσιο τομέα. Αυτές ενθάρρυναν την ανάπτυξη παράλληλης βιομηχανίας λογισμικού και ενθάρρυναν την ανάπτυξη φορητών και κλιμακούμενων μεγάλης κλίμακας παράλληλων εφαρμογών.

Επίσημες υλοποιήσεις:

1. Η αρχική εφαρμογή του προτύπου MPI 1.x ήταν η MPICH, από το εθνικό εργαστήριο Argonne (ANL) και το κρατικό πανεπιστήμιο του Mississippi. Η IBM ήταν επίσης πρώιμη εταιρεία υλοποίησης και οι περισσότερες εταιρείες υπερυπολογιστών της δεκαετίας του 90 είτε εμπορευόταν την MPICH είτε δημιούργησαν τη δική τους εφαρμογή. Το LAM / MPI από το Κέντρο υπερυπολογιστών του Οχάιο ήταν μια άλλη έγκαιρη ανοικτή εφαρμογή. Η ANL συνέχισε την ανάπτυξη της MPICH για πάνω από μια δεκαετία και τώρα προσφέρει MPICH-3.2, εφαρμόζοντας το πρότυπο MPI-3.1.

2. Το Open MPI (που δεν πρέπει να συγχέεται με το OpenMP) δημιουργήθηκε από τα συγχωνευμένα FT-MPI, LA-MPI, LAM / MPI και PACX-MPI και βρίσκεται σε πολλούς υπερυπολογιστές TOP-500.

2.3.2 Γλώσσες περιγραφής υλικού (HDL)

Verilog

Η Verilog, η οποία είναι τυποποιημένη ως IEEE 1364, είναι μια γλώσσα περιγραφής υλικού (HDL) που χρησιμοποιείται για τη μοντελοποίηση ηλεκτρονικών συστημάτων. Χρησιμοποιείται συνηθέστερα στο σχεδιασμό και την επαλήθευση ψηφιακών κυκλωμάτων στο επίπεδο αφαίρεσης καταχώρησης-

μεταφοράς. Χρησιμοποιείται επίσης στην επαλήθευση των αναλογικών κυκλωμάτων και των κυκλωμάτων μικτού σήματος, καθώς και στο σχεδιασμό γενετικών κυκλωμάτων.

Ένα υποσύνολο των δηλώσεων στη γλώσσα Verilog είναι συνθεσιμότητας. Μονάδες Verilog που συμμορφώνονται με ένα σύνθετο στυλ κωδικοποίησης, γνωστό ως RTL (επίπεδο καταχώρησης-μεταφοράς), μπορούν να υλοποιηθούν φυσικά με λογισμικό σύνθεσης. Το λογισμικό σύνθεσης αλγοριθμικά μετατρέπει την (αφηρημένη) πηγή Verilog σε μια netlist, μια λογικά ισοδύναμη περιγραφή που αποτελείται μόνο από στοιχειώδη λογικά αρχέτυπα (AND, OR, NOT, flip-flops κ.λπ.) που είναι διαθέσιμα σε μια συγκεκριμένη τεχνολογία FPGA ή VLSI. Περαιτέρω χειρισμοί στο netlist οδηγούν τελικά σε ένα σχέδιο κατασκευής κυκλώματος (όπως μια φωτογραφική μάσκα που έχει οριστεί για ένα ASIC ή ένα bitstream αρχείο για ένα FPGA).

VHDL

Η VHDL (VHSIC-HDL) είναι μια γλώσσα περιγραφής υλικού που χρησιμοποιείται στην ηλεκτρονική αυτοματοποίηση σχεδιασμού για την περιγραφή ψηφιακών συστημάτων και συστημάτων μικτού σήματος, όπως οι προγραμματιζόμενες σειρές πύλης και τα ολοκληρωμένα κυκλώματα. Το VHDL μπορεί επίσης να χρησιμοποιηθεί ως παράλληλη γενική γλώσσα προγραμματισμού.

Το VHDL χρησιμοποιείται συνήθως για την εγγραφή μοντέλων κειμένου που περιγράφουν ένα λογικό κύκλωμα. Ένα τέτοιο μοντέλο επεξεργάζεται από ένα πρόγραμμα σύνθεσης, μόνο αν είναι μέρος του λογικού σχεδιασμού. Ένα πρόγραμμα προσομοίωσης χρησιμοποιείται για τη δοκιμή του λογικού σχεδιασμού χρησιμοποιώντας μοντέλα προσομοίωσης για να αντιπροσωπεύει τα λογικά κυκλώματα που διασυνδέονται με το σχέδιο.

2.3.3 Ετερογενή υπολογιστικά συστήματα

Ο ετερογενής υπολογισμός μπορεί να οριστεί ως η συντονισμένη χρήση διαφόρων τύπων μηχανών, δικτύων και διεπαφών για τη μεγιστοποίηση της συνδυασμένης απόδοσής τους ή / και της σχέσης κόστους-αποτελεσματικότητας. Αυτά τα συστήματα επωφελούνται από την ποικιλία δυνατοτήτων παραλληλισμού που μπορούν να προσφέρουν σε διαφορετικά μέρη του φορτίου υπολογισμού διαφορετικά είδη επεξεργαστών, συνεπεξεργαστών και άλλων επιταχυντών υλικού, όπως μονάδες επεξεργασίας γραφικών (GPU) ή FPGAs. Έτσι, συνδυάζονται και χρησιμοποιούνται διαφορετικά πλεονεκτήματα που μπορούν να προσφέρουν διάφορα μέρη του υλικού, καθένα για το μέρος του υπολογισμού όπου είναι πιο πολύτιμα, βελτιστοποιώντας τη συνολική απόδοση του συστήματος.

OpenCL

Το OpenCL (Open Computing Language) είναι μια υποδομή για την εγγραφή προγραμμάτων που εκτελούνται σε ετερογενείς πλατφόρμες αποτελούμενες από κεντρικές μονάδες επεξεργασίας (CPU), μονάδες επεξεργασίας γραφικών (GPU), ψηφιακούς επεξεργαστές σημάτων (DSPs), προγραμματισμένες σειρές πύλης (FPGAs) επεξεργαστές ή επιταχυντές υλικού. Το OpenCL καθορίζει γλώσσες προγραμματισμού (βασισμένες σε C99 και C ++ 11) για τον προγραμματισμό αυτών των συσκευών και διεπαφών προγραμματισμού εφαρμογών (API) για τον έλεγχο της πλατφόρμας και την εκτέλεση προγραμμάτων στις υπολογιστικές συσκευές. Το OpenCL παρέχει μια τυποποιημένη διεπαφή για παράλληλο υπολογισμό χρησιμοποιώντας παραλληλισμό βάσει εργασίας και δεδομένων.

OpenACC

Το OpenACC (για ανοικτούς επιταχυντές) είναι ένα πρότυπο προγραμματισμού για παράλληλους υπολογιστές που αναπτύχθηκε από Cray, CAPS, Nvidia και PGI. Το πρότυπο έχει σχεδιαστεί για να απλοποιεί τον παράλληλο προγραμματισμό ετερογενών συστημάτων CPU / GPU.

Όπως και στο OpenMP, ο προγραμματιστής μπορεί να σχολιάσει τον πηγαίο κώδικα C, C ++ και Fortran για να προσδιορίσει τις περιοχές που πρέπει να επιταχυνθούν χρησιμοποιώντας οδηγίες μεταγλωττιστή και πρόσθετες λειτουργίες. Όπως το OpenMP 4.0 και το νεότερο, το OpenACC μπορεί

να στοχεύσει τόσο στην αρχιτεκτονική της CPU όσο και στην GPU και να ξεκινήσει τον υπολογιστικό κώδικα σε αυτά.

Κεφάλαιο 3

Αυτόματη Παραλληλοποίηση

Η άνοδος της υπολογιστικής επεξεργασίας τις τελευταίες δεκαετίες προκάλεσε την εμφάνιση του πεδίου αυτόματης παραλληλοποίησης με στόχο την πλήρη παράδοση οποιωνδήποτε εργασιών βελτιστοποίησης ή παραλληλισμού που διαχειρίζεται ο προγραμματιστής με υπερσύγχρονα εργαλεία ανάπτυξης. Αυτά τα εργαλεία είναι υπεύθυνα όχι μόνο για την ανάλυση και τη σύνταξη του κώδικα αλλά και για τη μετατροπή του σε μορφές που επιτρέπουν την εκμετάλλευση των προδιαγραφών της μνήμης του συστήματος καθώς και τους διαθέσιμους πόρους που μπορούν να χρησιμοποιηθούν για την κατανομή εργασιών και την επιτάχυνση. Πολλές έρευνες έχουν επικεντρωθεί στην ανάπτυξη τέτοιων τεχνικών, που απευθύνονται σε διαφορετικά περιβάλλοντα υπολογιστών, αρχιτεκτονικές και γλώσσες προγραμματισμού.

Παρόλο που η παραλληλοποίηση μπορεί να θεωρηθεί ως η κατανομή πλήρως ανεξάρτητων εργασιών σε διαφορετικούς πόρους επεξεργασίας, το πιο δημοφιλές τμήμα της αυτόματης παραλληλοποίησης αφορά κυρίως την ανάλυση και παραλληλισμό των δηλώσεων βρόχου, οι οποίες περιλαμβάνουν παρόμοια καθήκοντα υπολογισμών που εκτελούνται σε διαφορετικά δεδομένα. Αυτό προκαλείται κυρίως λόγω της πολυπλοκότητας που προκύπτει όταν αναλύονται οι μη ομοιόμορφες εργασίες, αλλά και επειδή ένα μεγάλο μέρος του υπολογιστικού φορτίου επικεντρώνεται σε αυτά τα τμήματα του κώδικα. Από την άλλη πλευρά, η ανάλυση εργασιών που αφορούν τις ίδιες ενέργειες αλλά σε διαφορετικές περιοχές μνήμης μπορεί να γίνει πολύ πιο εύκολη με τα υπάρχοντα εργαλεία σύνταξης. Φυσικά, διάφοροι ερευνητές στον τομέα διερευνούν όλες τις πιθανές κατευθύνσεις που μπορούν να δώσουν καρποφόρα αποτελέσματα, αλλά στο πλαίσιο αυτής της εργασίας δεν θα επικεντρωθούμε σε προσεγγίσεις που αποκλίνουν από την βροχο-κεντρική παραλληλοποίηση.

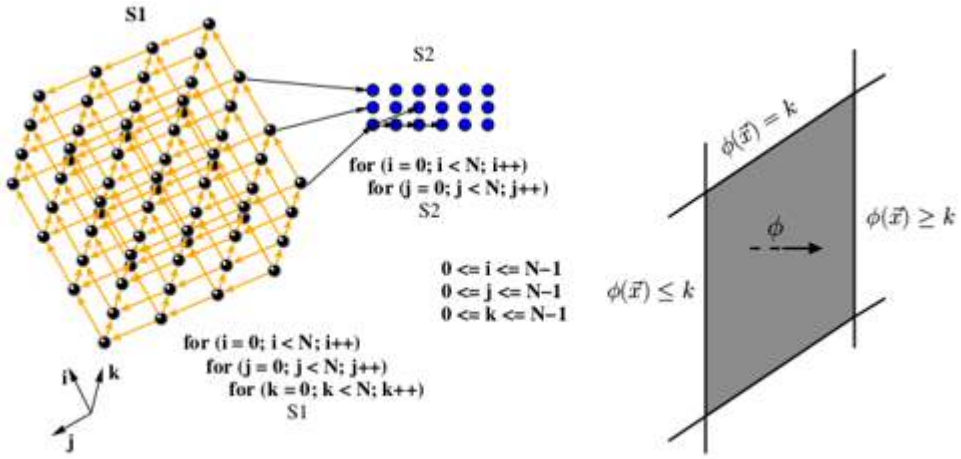
Αυτή η ενότητα αρχίζει με μια συζήτηση για μερικές από τις πιο δημοφιλείς προσεγγίσεις που εξετάζονται στον τομέα και συνεχίζει με την εισαγωγή και τη σύγκριση εργαλείων που έχουν αναπτυχθεί χρησιμοποιώντας αυτές τις τεχνικές για τη βελτιστοποίηση και παραλληλισμό των χειρόγραφων προγραμμάτων. Τέλος, θα παρουσιαστεί και θα χρησιμοποιηθεί ένα εκτεταμένο μοντέλο προγραμματισμού για την επιτυχή ενσωμάτωση των εργαλείων με μια πραγματική εφαρμογή χρήσης που θα χρησιμοποιηθεί για την αξιολόγησή τους.

3.1 Το Πολυεδρικό (Polyhedral) Μοντέλο

Το πολυεδρικό μοντέλο για τη βελτιστοποίηση του μεταγλωττιστή είναι ένα ισχυρό μαθηματικό πλαίσιο που βασίζεται στην παραμετρική γραμμική άλγεβρα και τον ακέραιο γραμμικό προγραμματισμό. Παρέχει μια αφαίρεση για να αντιπροσωπεύει τον υπολογισμό ένθετου βρόχου και τις εξαρτήσεις των δεδομένων του χρησιμοποιώντας ακέραια σημεία σε πολύεδρα.

Η σύνθετη αναδιοργάνωση εκτέλεσης, η οποία μπορεί να βελτιώσει την απόδοση με παραλληλισμό καθώς και με την ενίσχυση της τοποθεσίας, συλλαμβάνεται από μετασχηματισμούς συναρμογής στο πολυεδρικό μοντέλο. Το μοντέλο έχει φτάσει σε επίπεδο ωριμότητας σε διάφορες πτυχές - συγκεκριμένα, ως μια ισχυρή ενδιάμεση αναπαράσταση για την εκτέλεση μετασχηματισμών και την παραγωγή κώδικα μετά από μετασχηματισμούς.

Κάποιοι ορισμοί εμφανίζονται παρακάτω για να διευκολυνθεί η παρουσίαση των τεχνικών που βασίζονται στο μοντέλο:



Σχήμα 3.1: Το Πολυεδρικό (Polyhedral) μοντέλο

Αφινικό υπερεπίπεδο Το σύνολο X των διανυσμάτων $\vec{x} \in Z^n$ τέτοιων ώστε $h.\vec{x} = k$, για $k \in Z$, είναι ένα αφινικό υπερεπίπεδο.

Πολύεδρο Το σύνολο των διανυσμάτων $\vec{x} \in Z^n$ τέτοιων ώστε $A\vec{x}+b \geq 0$, όπου A, ένας πίνακας ακεραίων, ορίζει ένα (κυρτό) ακέραιο πολύεδρο. Ένα πολύτοπο ορίζεται ως ένα φραγμένο πολύεδρο.

Πολυεδρική αναπαράσταση προγραμμάτων

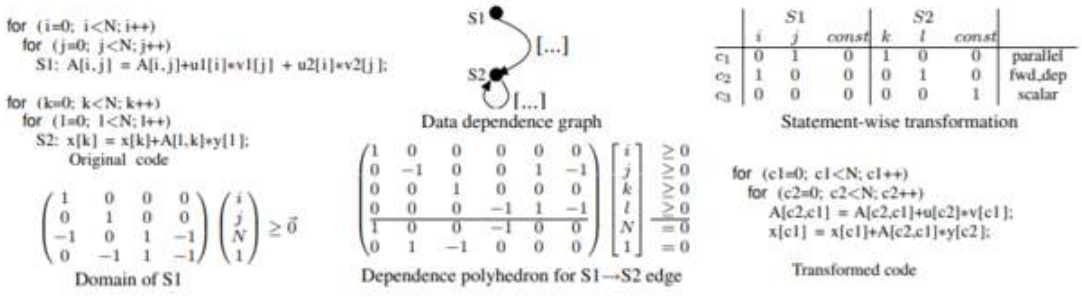
Με δεδομένο ένα πρόγραμμα, κάθε δυναμική εμφάνιση μιας εντολής, S, ορίζεται από το διάνυσμα επανάληψής της i, το οποίο περιέχει τιμές για τους δείκτες των βρόχων που περιβάλλουν το S, από το εξωτερικό έως το εσωτερικό. Κάθε φορά που τα όρια βρόχου είναι γραμμικοί συνδυασμοί δεικτών εξωτερικού βρόχου και παραμέτρων προγράμματος (τυπικά, συμβολικές σταθερές που αντιπροσωπεύουν τα μεγέθη προβλημάτων), το σύνολο διανυσμάτων επανάληψης που ανήκει σε μια δήλωση ορίζει ένα πολύτοπο. Έστω D_S να αντιπροσωπεύει το πολύτοπο και η διάστασή του να είναι m_S. Έστω $\vec{p}$ το διάνυσμα των παραμέτρων του προγράμματος.

3.1.1 Πολυεδρικές εξαρτήσεις

Το μοντέλο εξάρτησης που λαμβάνεται υπόψη αναφέρεται σε εξαρτήσεις που προσδιορίζονται με ακρίβεια μέσω της ανάλυσης ροής δεδομένων, αλλά θεωρούμε όλες τις εξαρτήσεις, συμπεριλαμβανομένων των εξαρτήσεων WAR (εγγραφή μετά την ανάγνωση), WAR (εγγραφή μετά την εγγραφή) και RAR (ανάγνωση μετά την ανάγνωση) δηλαδή ο κώδικας εισόδου δεν απαιτεί μετατροπή σε μορφή απλής εκχώρησης.

Γράφος εξάρτησης δεδομένων Ο Γράφος Εξάρτησης Δεδομένων (DDG) είναι ένα κατευθυνόμενο πολλαπλό γράφημα με κάθε κορυφή που αντιπροσωπεύει μια εντολή και μια ακμή, $e \in E$, από τον κόμβο S_i έως S_j που αντιπροσωπεύει μια πολυεδρική εξάρτηση από μια δυναμική εμφάνιση S_i σε ένα από τα S_j: χαρακτηρίζεται από ένα πολύεδρο, P_e, που ονομάζεται πολύεδρο εξάρτησης που συλλαμβάνει τις ακριβείς πληροφορίες εξάρτησης που αντιστοιχούν στο e. Το πολυεδρικό εξάρτησης είναι στο άθροισμα των διαστάσεων του πολυέδρου της πηγής και του προορισμού (με διαστάσεις για τις παραμέτρους του προγράμματος επίσης). Έστω $\vec{s}$ αντιπροσωπεύει την επανάληψη πηγής και $\vec{t}$ είναι η επαναληπτική στόχευση που σχετίζεται με την εξάρτηση εξάρτησης e. Είναι δυνατόν να εκφραστεί η επανάληψη του πηγαίου κώδικα ως αφινική συνάρτηση του στόχου επαναλήψεως 102, δηλ. για να βρεθεί η τελευταία ταυτόχρονη πρόσβαση. Αυτή η αφινική συνάρτηση είναι επίσης γνωστή ως h-μετασχηματισμός, και θα εκπροσωπείται από τον για μια ακμή εξάρτησης e. Ως εκ τούτου, $\vec{s} = h_e(\vec{t})$. Οι ισότητες που αντιστοιχούν στον μετασχηματισμό h είναι ένα μέρος του πολύεδρου εξάρτησης και μπορούν να χρησιμοποιηθούν για να μειώσουν τις διαστάσεις του.

Το σχήμα 3.2 δείχνει την πολυεδρική αναπαράσταση ενός απλού παραδείγματος. Έστω S_1, S_2, . . . , S_n οι εντολές του προγράμματος. Ένας μονοδιάστατος αφινικός μετασχηματισμός για την εντολή



Σχήμα 3.2: Παράδειγμα πολυεδρικής αναπαράστασης προγράμματος

S_k ορίζεται ως:

$$\phi_{s_k}(\vec{i}) = [c_1 \ldots c_{m_{s_k}}](\vec{i}) + c_0, c_i \in Z \tag{3.1}$$

ϕ_{s_k} μπορεί επίσης να ονομάζεται αφινικό υπερεπίπεδο ή μια συνάρτηση σκέδασης όταν ασχολείται με τη γεννήτρια κώδικα. Ένας πολυδιάστατος αφινικός μετασχηματισμός για μια εντολή αντιπροσωπεύεται από μια μήτρα με κάθε σειρά να είναι ένα αφινικό υπερεπίπεδο.

Ικανοποίηση εξάρτησης: Μία αφινική εξάρτηση με το πολύεδρο P_e ικανοποιείται σε ένα επίπεδο l αν και μόνο αν η ακόλουθη συνθήκη ικανοποιείται:

$$\forall k(1 \leq k \leq l-1) : \phi^k_{s_j}(\vec{t}) - \phi^k_{s_i}(\vec{s}) \geq 0, \ <\vec{s},\vec{t}> \in \ P_e \tag{3.2}$$

και

$$\phi^l_{s_j}(\vec{t}) - \phi^l_{s_i}(\vec{s}) \geq 0, \ <\vec{s},\vec{t}> \in \ P_e \tag{3.3}$$

3.1.2 Μετασχηματισμοί βρόχων

Έρευνες σε τεχνολογίες αιχμής στον τομέα υποδεικνύουν τη σημασία του πολυεδρικού μοντέλου στην περιοχή της αυτόματης παραλληλοποίησης, το οποίο χρησιμοποιείται για τον μετασχηματισμό του κώδικα με τέτοιο τρόπο, όπου υφίστανται οι υπάρχουσες εξαρτήσεις και εμφανίζονται νέα παράλληλα τμήματα κώδικα.

Υπάρχουν δύο κοινές προσεγγίσεις για τον μετασχηματισμό του βρόχου. Στην πρώτη προσέγγιση, οι ερευνητές έχουν θεωρήσει σύνθετους μετασχηματισμούς ως μια σειρά μεμονωμένων μετασχηματισμών στη φωλιά βρόχων. Αυτή η προσέγγιση ισχύει για γενικές φωλιές βρόχου, αλλά δεν είναι σαφές πώς να επιλέξουμε τον καλύτερο συνδυασμό μετασχηματισμών που θα εφαρμοστεί σε μια δεδομένη φωλιά βρόχου. Μια δεύτερη προσέγγιση θεωρεί τους μετασχηματισμούς να είναι μετασχηματισμοί μήτρας σε ένα χώρο επανάληψης. Αυτή η προσέγγιση είναι κομψή αλλά εφαρμόζεται μόνο σε περιορισμένες περιπτώσεις, όταν οι εξάρσεις μπορούν να συνοψιστούν ως διανύσματα απόστασης. Το πολυεδρικό μοντέλο συνδυάζει τη μαθηματική αυστηρότητα στο μοντέλο μετασχηματισμού μήτρας με τη μεγαλύτερη γενικότητα της ακολουθίας των μεμονωμένων μετασχηματισμών. Η ανταλλαγή βρόχων (μεταστοιχείωση), η αντιστροφή και η κλίση είναι ενοποιημένες ως μη ομοιώδεις μετασχηματισμοί και οι φορείς εξάρτησης ενσωματώνουν πληροφορίες απόστασης και κατεύθυνσης. Αυτή η ενοποίηση παρέχει μια γενική δοκιμή για να προσδιοριστεί εάν ο κώδικας που λαμβάνεται μέσω ενός μετασχηματισμού ένωσης είναι νόμιμος, σε αντίθεση με μια ειδική δοκιμή νομιμότητας για κάθε μεμονωμένο στοιχειώδη μετασχηματισμό. Έτσι, το πρόβλημα μετασχηματισμού βρόχου μπορεί να διατυπωθεί ως άμεση επίλυση για τον μετασχηματισμό που μεγιστοποιεί κάποια αντικειμενική λειτουργία, ενώ ικανοποιεί ένα σύνολο περιορισμών. Χρησιμοποιώντας αυτή τη θεωρία, έχουν αναπτυχθεί πολλοί αλγόριθμοι για τη βελτίωση του παραλληλισμού και της θέσης μιας φωλιάς βρόχου μέσω μετασχηματισμών βρόχου. Το πλακίδιο, αν και δεν είναι ένας μονόμορφος μετασχηματισμός,

θεωρείται επίσης και περιγράφεται ως συνήθως χρησιμοποιούμενη τεχνική στο πεδίο αυτόματης παραλληλοποίησης.

Η μεταστροφή, η αναστροφή και η κλίση είναι τρεις γνωστοί μετασχηματισμοί που μπορούν να αναπαρασταθούν ως μη μονωτικές μήτρες.

Permutation: A permutation σ on a loop nest transforms iteration $(p_1 \ldots p_n)$ to $(p_{\sigma_1} \ldots p_{\sigma_n})$. This transformation can be expressed in matrix form as I_σ, the n×n identity matrix I with rows permuted by σ. The loop interchange above is an n = 2 example of the general permutation transformation.

Reversal: Reversal of the $i_t h$ loop is represented by the identity matrix, but with the $i_t h$ diagonal element equal to 1 rather than 1. For example, the matrix representing loop reversal of the outermost loop of a two-deep loop nest is:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Skewing: Skewing loop I_j by an integer factor f with respect to loop I_i maps the iteration:

$$(p_1 \ldots p_{i-1}, p_i, p_{i+1} \ldots p_{j-1}, p_j, p_{j+1} \ldots p_n) \tag{3.4}$$

to:

$$(p_1 \ldots p_{i-1}, p_i, p_{i+1} \ldots p_{j-1}, p_j + f p_i, p_{j+1} \ldots p_n) \tag{3.5}$$

Η μήτρα μετασχηματισμού T που παράγει κλίση είναι η μήτρα ταυτότητας, αλλά με το στοιχείο tj, i ίσο με f και όχι μηδέν. Δεδομένου ότι το i <j, το T πρέπει να είναι κάτω τριγωνικό.

Tiling: Ο μετασχηματισμός tiling παρά το γεγονός ότι δεν είναι ένας unimodular μετασχηματισμός, είναι ένας βασικός μετασχηματισμός για τη βελτίωση της θέσης και μπορεί επίσης να χρησιμοποιηθεί για να δημιουργήσει δυνατότητες course-grain παραλληλισμού. Το tile περιλαμβάνει τους γνωστούς μετασχηματισμούς strip-mine-and-interchange και unroll-and-jam. Όπως και με τους unimodular μετασχηματισμούς, ο προγραμματιστής πρέπει να εξετάσει υπό ποιες συνθήκες ο μετασχηματισμός είναι έγκυρος και πώς η εφαρμογή του σε μια φωλιά βρόχου αλλάζει τη φωλιά βρόχου. Οι πιο δημοφιλείς μέθοδοι tiling θεωρούν μόνο ορθογώνια tiles. Εντούτοις, μπορούν να επιτευχθούν μη ορθογώνια tiles συνδυάζοντας (ορθογώνια) tiles με unimodular μετασχηματισμούς βρόχου.

3.2 Εργαλεία Αυτόματης Παραλληλοποίησης

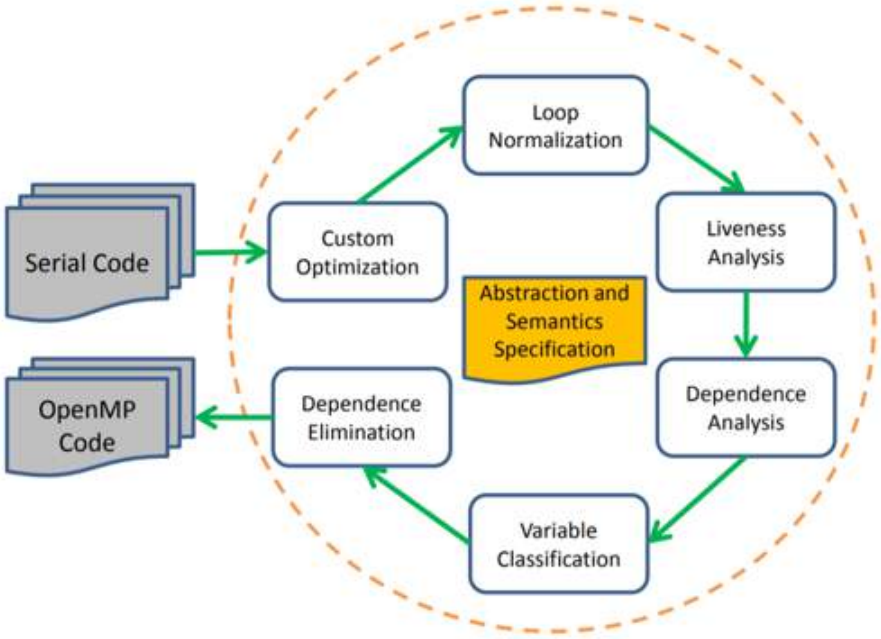
3.2.1 Μεταγλωττιστής ROSE – εργαλείο autoPar

Το ROSE είναι μια υποδομή μεταγλωττιστή ανοιχτού κώδικα για την κατασκευή εργαλείων μετατροπής και ανάλυσης προγραμμάτων πηγής προς πηγή για εφαρμογές C / C ++ και Fortran μεγάλης κλίμακας. Δεδομένου ότι διατηρεί την αναπαράσταση της αφαίρεσης υψηλού επιπέδου, δεν χάνουν τις απαιτούμενες πληροφορίες για να αναγνωρίσουν τέτοιες αφαίρεσεις και η σχετική σημασιολογία μπορεί να συναχθεί αξιόπιστα. Το ROSE επιτρέπει ακόμη και σε μη εξειδικευμένους χρήστες να εκμεταλλευτούν τεχνικές μεταγλωττιστή για την αντιμετώπιση της ανάλυσης και του μετασχηματισμού των αφαίρεσης.

Ένας παραλληλιστής σχεδιάστηκε στο πλαίσιο της ανάπτυξης του έργου ROSE χρησιμοποιώντας τον μεταγλωττιστή για τον αυτόματο παραλληλισμό των βρόχων-στόχων και των λειτουργιών, εισάγοντας είτε omp for είτε omp task και άλλες απαιτούμενες οδηγίες OpenMP και ρήτρες. Για προγράμματα εισόδου με υπάρχουσες οδηγίες OpenMP, το εργαλείο θα ελέγξει διπλά την ορθότητα όταν είναι ενεργοποιημένη η σωστή επιλογή.

Έχει σχεδιαστεί για να χειρίζεται και τους συμβατικούς βρόχους που λειτουργούν σε πρωτόγονες συστοιχίες και σύγχρονες εφαρμογές χρησιμοποιώντας αφαίρεσεις υψηλού επιπέδου.

Η βασική ιδέα του αλγόριθμου είναι να συλλάβει εξαρτήσεις εντός ενός στόχου και να εξαλειφθεί αργότερα όσο το δυνατόν περισσότερο με βάση διάφορους κανόνες. Ο παραλληλισμός είναι ασφαλής αν δεν υπάρχουν υπόλοιπες εξαρτήσεις. Τα semantics της αφαίρεσης χρησιμοποιούνται σχεδόν σε κάθε βήμα για να διευκολύνουν τους μετασχηματισμούς και τις αναλύσεις, συμπεριλαμβανομένης



Σχήμα 3.3: Ροή εκτέλεσης του μεταγλωττιστή ROSE

της αναγνώρισης των κλήσεων λειτουργίας ως μεταβλητών αναφοράς, της αναγνώρισης του τρέχοντος στοιχείου που προσπελαύνεται και της διασφάλισης αν υπάρχουν περιορισμοί για την σειράς προσβάσεων εγγραφής σε κοινές μεταβλητές.

3.2.2 Παραλληλοποιητής PLuTo

Το PLuTo είναι ένα αυτόματο εργαλείο παραλληλισμού που βασίζεται στο πολυεδρικό μοντέλο. Το εργαλείο μετασχηματίζει τα προγράμματα C από πηγή σε πηγή για το course-grain παραλληλισμό και την τοποθεσία δεδομένων ταυτόχρονα. Το πλαίσιο μετασχηματισμού πυρήνα λειτουργεί κυρίως με την εύρεση συναφών μετασχηματισμών για αποδοτικό πλακάκι. Ο παράλληλος κώδικας OpenMP για πολυπύρηνα μπορεί να δημιουργηθεί αυτόματα από διαδοχικές ενότητες προγράμματος C. Εξωτερική (χωρίς επικοινωνία), εσωτερική ή σωληνοειδής παραλληλοποίηση επιτυγχάνεται καθαρά με OpenMP παράλληλο για pragmas, ο κώδικας είναι επίσης βελτιστοποιημένος για την τοποθεσία και είναι κατάλληλος για αυτόματη διανυσμάτωση.

Όπως αναφέρθηκε ήδη, το πολυεδρικό μοντέλο αποτελεί ένα ολοκληρωμένο μοντέλο για την ανάλυση της ροής δεδομένων και την εφαρμογή οποιωνδήποτε μετασχηματισμών που θεωρούνται απαραίτητες για την εκμετάλλευση του παραλληλισμού. Ωστόσο, μια προσέγγιση για την αυτόματη εύρεση καλών μετασχηματισμών για τη βελτιστοποίηση της επικοινωνίας με course-grain παραλληλισμό μαζί με τη βελτιστοποίηση της τοποθεσίας υπήρξε βασικός σύνδεσμος που λείπει. Το PLuTo έχει αναπτυχθεί για να προσφέρει αυτό ακριβώς. ένα νέο πλαίσιο αυτόματου μετασχηματισμού που λύνει το παραπάνω πρόβλημα. Αυτή η προσέγγιση λειτουργεί με την εξεύρεση καλών μετασχηματισμών συνάφειας μέσω μιας ισχυρής και πρακτικής γραμμικής συνάρτησης κόστους που επιτρέπει αποτελεσματικό tiling και σύνθεση αλληλουχιών αυθαίρετα εμφωλευμένων βρόχων. Αυτό με τη σειρά του επιτρέπει ταυτόχρονη βελτιστοποίηση για course-grain παραλληλισμό και τοπολογικές βελτιστοποιήσεις. Ο παραλληλισμός χωρίς συγχρονισμό και ο αγωγός παραλληλισμός σε διάφορα επίπεδα μπορούν να εξαχθούν. Το πλαίσιο μπορεί να στοχεύει σε διαφορετικές παράλληλες αρχιτεκτονικές, όπως οι πολλαπλές γενικές χρήσεις, ο επεξεργαστής κυψέλης, οι μονάδες GPU ή οι ενσωματωμένοι SoCs πολλαπλών επεξεργαστών.

Το περιγραφόμενο πλαίσιο έχει εφαρμοστεί, σε ένα νέο εργαλείο μετασχηματισμού από άκρο σε άκρο, το PLuTo, το οποίο μπορεί να παράγει αυτόματα παράλληλο κώδικα από τα κανονικά τμήματα του προγράμματος C. Τα πειραματικά αποτελέσματα από το εφαρμοσμένο σύστημα δείχνουν σημαντική βελτίωση της απόδοσης για μονόπλευρη και πολυπύρηνη εκτέλεση μέσω υπερσύγχρονων πλαισίων μεταγλωττιστών έρευνας καθώς και των καλύτερων μεταγλωττιστών φυσικής παραγωγής. Για πολλούς πυκνούς πυρήνες γραμμικής άλγεβρας, ο κώδικας που παράγεται από τον Πλούτωνα χτυπά, με σημαντικό περιθώριο, τους ίδιους πυρήνες που εφαρμόζονται με τις ακολουθίες κλήσεων σε βιβλιοθήκες υψηλής ευκρίνειας που παρέχονται από τους προμηθευτές. Το σύστημα επιτρέπει επίσης τη διενέργεια εμπειρικής βελτιστοποίησης σε ένα πολύ ευρύτερο πλαίσιο από ό, τι έχει επιχειρηθεί προηγουμένως. Επιπλέον, το PLuTo μπορεί να χρησιμεύσει ως παράθυρο δημιουργίας παράλληλου κώδικα για αρκετές γλώσσες υψηλού επιπέδου για συγκεκριμένο τομέα.

Το πολυεδρικό πλαίσιο μεταγλωττιστή είναι μια αφαίρεση για ανάλυση και μετασχηματισμό των προγραμμάτων. Καταγράφει την εκτέλεση ενός προγράμματος σε μια στατική ρύθμιση, εκπροσωπώντας τις περιπτώσεις του ως ακέραια σημεία εντός παραμετρικής πολυεδρίας. Τα περισσότερα διαθέσιμα στο κοινό εργαλεία και μεταγλωττιστές που χρησιμοποιούν αυτό το πλαίσιο, εξάγουν μια τέτοια αναπαράσταση από τα προγράμματα C, C ++ και Fortran.

Πολυεδρική αναπαράσταση προγραμμάτων: Έστω S_1, S_2 ... S_n οι δηλώσεις του προγράμματος. Κάθε δυναμική εμφάνιση μιας δήλωσης, S, αναγνωρίζεται από το διάνυσμα επανάληψής της i που περιέχει τιμές για τους δείκτες των βρόχων που περιβάλλουν το S, από το εξωτερικό στο εσωτερικό. Κάθε φορά που τα όρια του βρόχου είναι συναρτήσεις συναρτήσεων των δεικτών του εξωτερικού βρόχου και των παραμέτρων του προγράμματος, το σύνολο των διανυσμάτων επανάληψης που ανήκουν σε μια δήλωση σχηματίζει ένα κυρτό πολυεδρικό που ονομάζεται τομέα ή σύνολο δεικτών του. Έστω ότι I_S είναι το σύνολο δεικτών του S και αφήστε τη διαστατικότητα του να είναι m_S. Έστω p ο φορέας των παραμέτρων του προγράμματος. Οι παράμετροι του προγράμματος δεν τροποποιούνται οπουδήποτε στο τμήμα του κώδικα που προσπαθούμε να μοντελοποιήσουμε.

Μια συνάρτηση f σε ένα πεδίο I_S λέγεται αφινική συνάρτηση αν μπορεί να αναπαρασταθεί από την ακόλουθη μορφή:

$$f(\vec{i}) = [c_1 c_2 \ldots c_{m_S}](\vec{i}) + c_0, i \in I_S \tag{3.6}$$

Οι τακτικές προσβάσεις δεδομένων σε μια δήλωση αντιπροσωπεύονται ως πολυδιάστατες αφινικές συναρτήσεις των δεικτών τομέα. Οι κώδικες που ικανοποιούν αυτούς τους περιορισμούς είναι επίσης γνωστοί ως αφινικοί εμφωλευμένοι βρόχοι. Πολυεδρικές εξάρσεις: Το γράφημα εξάρτησης δεδομένων (DDG) είναι ένα κατευθυνόμενο πολλαπλό γράφημα με κάθε κορυφή που αντιπροσωπεύει μια δήλωση και μια ακμή, e ∈ E, από τον κόμβο S_i έως S_j που αντιπροσωπεύει μια πολυεδρική εξάρτηση από μια επανάληψη του S_i σε μια επανάληψη του S_j: χαρακτηρίζεται από ένα πολυεδρικό, D_e, που ονομάζεται πολυεξάνιο εξάρτησης και συλλαμβάνει ακριβείς πληροφορίες εξάρτησης που αντιστοιχούν στο e. Το πολυεδρικό εξάρτησης είναι στο άθροισμα των διαστάσεων των χώρων επαναλήψεων πηγής και στόχου και στον αριθμό των παραμέτρων του προγράμματος. Τουλάχιστον μία από τις προσπελάσεις πηγής και στόχου πρέπει να είναι μια εγγραφή.

```
for(t=0; t<=-T1; t++)
    for(i=1; i<=-N2; i++)
        for(j=1; j<=-N2; j++)
            a[i][j]= ( a[-i1][-j1] + a[-i1][j] + a[-i1][j+1]
            + a[i][-j1] + a[i][j] + a[i][j+1] + a[i+1][-j1] + a[i+1][j] + a[i+1][
    j+1] ) / 9.0
```

Για παράδειγμα, στο τμήμα κώδικα παραπάνω, η εξάρτηση μεταξύ της εγγραφής a[i] [j] στο $\vec{s}$ = (t, i, j) και της ανάγνωσης στο $\vec{t}$ = t ', i', j ') σε ένα [i'-1] [j'-1] δίνεται από το πολυεδρικό εξάρτησης, D_e = $\vec{s}$, $\vec{t}$, $\vec{p}$, 1), που αποτελεί συνδυασμό των ακόλουθων εξισώσεων και ανισοτήτων:

$$i' = i + 1,\ j' = j + 1,\ t' = t \tag{3.7}$$

$$0 \leq t \leq T\text{–}1,\ 1 \leq i \leq N\text{–}3,\ 1 \leq j \leq N\text{–}3 \tag{3.8}$$

3.2.3 Intel C++ Compiler

Μια άλλη επιλογή για έναν αυτόματο μεταγλωττιστή παραλλαγής που είναι διαθέσιμος, αλλά όχι ανοικτού κώδικα σε αντίθεση με τους άλλους δύο, είναι ο επεξεργαστής Intel C ++. Η λειτουργία αυτόματου παραλληλισμού του μεταγλωττιστή Intel C ++ μεταφράζει αυτόματα τα σειριακά τμήματα του προγράμματος εισόδου σε σημασιολογικά ισοδύναμο κώδικα πολλαπλών νημάτων. Η αυτόματη παραλληλοποίηση καθορίζει τους βρόχους που είναι ικανοί να μοιραστούν τους υποψηφίους, εκτελεί την ανάλυση ροής δεδομένων για να επαληθεύσει τη σωστή παράλληλη εκτέλεση και χωρίζει τα δεδομένα για δημιουργία γενικών κωδικωνίων όπως απαιτείται για τον προγραμματισμό των οδηγιών OpenMP. Οι εφαρμογές OpenMP και αυτόματης παραλληλοποίησης παρέχουν τα κέρδη από την κοινή μνήμη σε συστήματα πολλαπλών επεξεργαστών.

Οι μεταγλωττιστές της Intel βελτιστοποιούνται στα συστήματα υπολογιστών που χρησιμοποιούν επεξεργαστές που υποστηρίζουν αρχιτεκτονικές της Intel. Έχουν σχεδιαστεί για να ελαχιστοποιούν τους πάγκους και να παράγουν κώδικα που εκτελείται στον ελάχιστο δυνατό αριθμό κύκλων. Ο επεξεργαστής Intel C ++ υποστηρίζει τρεις ξεχωριστές τεχνικές υψηλού επιπέδου για τη βελτιστοποίηση του προγραμματισμένου προγράμματος: βελτιστοποίηση μεταξύ διαδικασιών (IPO), βελτιστοποίηση καθοδηγούμενου από προφίλ (PGO) και βελτιστοποιήσεις υψηλού επιπέδου (HLO). Ο μεταγλωττιστής Intel C ++ στα προϊόντα Parallel Studio XE υποστηρίζει επίσης εργαλεία, τεχνικές και επεκτάσεις γλώσσας για την προσθήκη και τη διατήρηση παραλληλισμού εφαρμογών σε επεξεργαστές IA-32 και Intel 64 και επιτρέπει την επεξεργασία για επεξεργαστές και συνεργαζόμενους επεξεργαστές Intel Xeon Phi.

Η βελτιστοποίηση με καθοδήγηση βάσει προφίλ αναφέρεται σε έναν τρόπο βελτιστοποίησης, όπου ο μεταγλωττιστής έχει τη δυνατότητα πρόσβασης σε δεδομένα από μια δοκιμή δείγματος του προγράμματος σε ένα αντιπροσωπευτικό σύνολο εισόδων. Τα δεδομένα δείχνουν ποιες περιοχές του προγράμματος εκτελούνται συχνότερα και ποιες περιοχές εκτελούνται λιγότερο συχνά. Όλες οι βελτιστοποιήσεις επωφελούνται από τα σχόλια που καθοδηγούνται από το προφίλ επειδή είναι λιγότερο εξαρτημένα από τα ευρετικά στοιχεία κατά τη λήψη των αποφάσεων συλλογής.

Οι βελτιστοποιήσεις υψηλού επιπέδου είναι βελτιστοποιήσεις που εκτελούνται σε μια έκδοση του προγράμματος που αντιπροσωπεύει περισσότερο τον πηγαίο κώδικα. Αυτό περιλαμβάνει την εναλλαγή βρόχου, τη σύντηξη βρόχων, την σχάση βρόχου, την εκτύλιξη βρόχου, την προφόρτιση δεδομένων και πολλά άλλα.

Η διεπιχειρησιακή βελτιστοποίηση εφαρμόζει τις τυπικές βελτιστοποιήσεις του μεταγλωττιστή (όπως η συνεχής διάδοση) αλλά χρησιμοποιεί ένα ευρύτερο πεδίο που μπορεί να περιλαμβάνει πολλαπλές διαδικασίες, πολλαπλά αρχεία ή ολόκληρο το πρόγραμμα.

3.2.4 CETUS

Ο Cetus είναι μια υποδομή μεταγλωττιστή για τον μετασχηματισμό των προγραμμάτων από πηγή σε πηγή. Δημιουργήθηκε από την ανάγκη για ένα ερευνητικό περιβάλλον μεταγλωττιστή που διευκολύνει την ανάπτυξη διεργαστηριακών τεχνικών ανάλυσης και παραλληλισμού για προγράμματα C, C ++ και Java. Το Cetus είναι ένα σύνολο κατηγοριών ενδιάμεσης αναπαράστασης (IR) και διαβάσεων βελτιστοποίησης και δεν περιέχει κανένα ιδιόκτητο κώδικα που βασίζεται σε ελεύθερα διαθέσιμα εργαλεία. Για τη δημιουργία ενός αναλυτή Cetus εξετάστηκαν οι γεννήτριες συνόλων Yacc και Bison, οι οποίες χρησιμοποιούν lex ή flex για σάρωση, και το Antlr, το οποίο είναι συνδεδεμένο με τη δική του γεννήτρια σαρωτή.
Ο Cetus έχει τους ακόλουθους στόχους:

- Η εσωτερική αναπαράσταση (IR) είναι ορατή από τον συγγραφέα περάσματος (χρήστης) μέσω μιας διεπαφής, την οποία θα αναφερθούμε ως IR-API. Ο σχεδιασμός ενός απλού, εύχρηστου IR-API, που είναι επεκτάσιμο για μελλοντικές δυνατότητες - ειδικά για την υποστήριξη άλλων γλωσσών - είναι το πιο δύσκολο έργο μηχανικής.

- Πρέπει να είναι εύκολο να γράψετε μετασχηματισμούς πηγής-πηγής και περάσματα βελτιστοποίησης. Η υλοποίηση είναι μια ιεραρχική ταξινόμηση αντικειμένων με ελάχιστο αριθμό ονομάτων μεθόδων IR-API (χρησιμοποιώντας εικονικές λειτουργίες και συνεπή ονομασία), εύχρηστες μεθόδους IR traversal και πληροφορίες που μπορούν να συναχθούν από άλλα δεδομένα αυστηρά κρυμμένα από το χρήστης.

- Η ευκολία του εντοπισμού σφαλμάτων μπορεί να είναι καθοριστική για την επιτυχία οποιουδήποτε έργου μεταγλωττιστή που χρησιμοποιεί την υποδομή. Το IR-API θα πρέπει να καταστήσει αδύνατη τη δημιουργία αντιφατικών αναπαραστάσεων προγραμμάτων, αλλά εξακολουθούμε να χρειαζόμαστε εργαλεία που συλλαμβάνουν κοινά λάθη και περιβάλλοντα που καθιστούν εύκολη την ανίχνευση σφαλμάτων σε περίπτωση εμφάνισης προβλημάτων.

- Το Cetus θα πρέπει να λειτουργεί σε πολλαπλές πλατφόρμες χωρίς καμία ή ελάχιστη τροποποίηση. Η φορητότητα της υποδομής σε μια μεγάλη ποικιλία πλατφορμών θα κάνει τον Cetus χρήσιμο σε μια μεγαλύτερη κοινότητα.

3.3 Πειραματισμός και σύγκριση

3.3.1 Πειραματισμός εργαλείων

Για την αξιολόγηση των εργαλείων χρησιμοποιήσαμε τη μηχανή αμμοβολής, μέρος των πόρων CSLab, που είναι εξοπλισμένη με 4 επεξεργαστές Intel Xeon E5-4620 (Sandy Bridge) σε αρχιτεκτονική NUMA. Κάθε επεξεργαστής διαθέτει 8 πυρήνες, παρέχοντας συνολικά 64 κλωστές διαθέσιμες για εκτέλεση.

Η σουίτα αναφοράς «polybench» χρησιμοποιήθηκε για τη σύγκριση των εργαλείων, δεδομένου ότι παρέχει ένα σύνολο σημείων αναφοράς ειδικά σχεδιασμένα για να δοκιμάζουν τις τεχνικές παραλληλοποίησης. Συγκεκριμένα, στη σουίτα διατίθεται μια σειρά εφαρμογών που σχετίζονται με υπολογιστικές μεθόδους, που καλύπτουν απόλυτα τις ανάγκες της αξιολόγησης. Τα κριτήρια αναφοράς παρουσιάζονται συνοπτικά στον Πίνακα 3.1.

Οι παράλληλες εκδόσεις των σημείων αναφοράς εκτελέστηκαν στο μηχάνημα αμμοβολής (4 x Intel Xeon E5-4620 - Sandy Bridge) του εργαστηρίου χρησιμοποιώντας 1, 2, 4, 8, 16, 32 και 64 νήματα. Από αυτό το πείραμα, εξήχθη μια σειρά από αποτελέσματα σχετικά με την επιτάχυνση που παρατηρείται μετά την παραλληλισμό, καθώς και την επεκτασιμότητα των παραλληλισμένων προγραμμάτων, όταν διατίθενται πρόσθετοι υπολογιστικοί πόροι. Τα αποτελέσματα παρουσιάζονται στο Παράρτημα Α.

Γενικές παρατηρήσεις

Όσον αφορά την δυνατότητα κλιμάκωσης των επιδόσεων όπως παρουσιάζεται στο Παράρτημα Α, μπορούμε να παρατηρήσουμε ότι και τα δύο εργαλεία δείχνουν μια καλή συμπεριφορά κλιμακώσεως, ενώ καθώς αυξάνουμε τον αριθμό των χρησιμοποιούμενων νημάτων, ο χρόνος εκτέλεσης μειώνεται αναλόγως. Στις περισσότερες περιπτώσεις, το PLuTo αποδεικνύεται ότι έχει καλύτερη απόδοση από το ROSE. Μπορούμε να υποθέσουμε ότι αυτό προκαλείται κυρίως λόγω των μετασχηματισμών που πραγματοποιεί το PLuTo για να βελτιώσει τις λειτουργίες πρόσβασης μνήμης και έτσι να βελτιώσει τη συνολική απόδοση ακόμη και όταν χρησιμοποιεί 1 νήμα.

Για να εισαγάγουμε μια πιο διαφωτιστική επισκόπηση της αποτελεσματικότητας των εργαλείων, έχουμε ταξινομηθεί η συμπεριφορά απόδοσης σε οκτώ κατηγορίες ανάλογα με το συνολικό κέρδος και την επεκτασιμότητα που παρατηρούμε κατά την εκτέλεση των κριτηρίων αναφοράς. Για λόγους απεικόνισης μπορούμε να θεωρήσουμε ως αρχικό χρόνο εκτέλεσης την απόδοση των σημείων αναφοράς όταν εκτελούμε την έκδοση ROSE με ένα νήμα (δηλαδή το πρώτο σημείο σε κάθε μπλε γραφική παράσταση), αφού το ROSE δεν εκτελεί μετασχηματισμούς κώδικα και έτσι τρέχει αυτή την έκδοση χρησιμοποιώντας ένα Το νήμα είναι λίγο πολύ η σειριακή / αρχική έκδοση.

Οι οκτώ κατηγορίες ορίζονται ως εξής:

Benchmark	Description
2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)
3mm	3 Matrix Multiplications ((A*B)*(C*D))
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
head-3d	Heat equation over 3D data domain
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Πίνακας 3.1: Η σουίτα μετροπρογραμμάτων polybench

1. **High scalability**
   Οι περιπτώσεις όπου ο χρόνος εκτέλεσης γίνεται γρήγορα μικρότερος με την αύξηση του αριθμού των νημάτων που χρησιμοποιούνται

2. **Low scalability**
   Οι περιπτώσεις όπου ο χρόνος εκτέλεσης γίνεται μικρότερος με αργό ρυθμό με την αύξηση του αριθμού των νημάτων που χρησιμοποιούνται

3. **No scalability, good performance**
   Οι περιπτώσεις όπου δεν παρατηρείται καμία κλιμακοσιμότητα καθώς αυξάνουμε τον αριθμό των νημάτων, αλλά παρ'όλα αυτά η απόδοση του μετροπρογράμματος είναι καλή λόγω βελτιστοποιήσεων στον κώδικα (μόνο στην περίπτωση του PLuTo)

4. **First worse, then better**
   Οι περιπτώσεις όπου οι βελτιστοποιήσεις στον κώδικα προκαλούν μεγάλη καθυστέριση στην αρχή, αλλά με την αύξηση του αριθμού των treads, η απόδοση γίνεται καλύτερη από αυτή της αρχικής έκδοσης (μόνο για το PLuTo)

5. **Early saturation**

Οι περιπτώσεις όπου μετά την εισαγωγή ενός μικρού αριθμού νημάτων, η απόδοση δεν βελτιώνεται με την προσθήκη περισσότερων

6. **First better, then worse**
   Οι περιπτώσεις όπου μετά την εισαγωγή ενός μικρού αριθμού νημάτων, η απόδοση επιδεινώνεται με την προσθήκη περισσότερων νημάτων

7. **No performance gain**
   Οι περιπτώσεις όπου δεν υπάρχει κέρδος απόδοσης με την εισαγωγή περισσότερων νημάτων

8. **Scaling backwards**
   Οι περιπτώσεις όπου η προσθήκη των νημάτων προκαλεί μόνο χαμηλότερη απόδοση

Τα αποτελέσματα παρουσιάζονται χρησιμοποιώντας τις παραπάνω κατηγορίες στον Πίνακα 3.2.

	High scalability	Low scalability	No scalability, good performance	First worse, then better
ROSE	16	2	0	0
PLuTo	6	3	7	1

	Early saturation	First better, then worse	No performance gain	Scaling backwards
ROSE	0	3	4	5
PLuTo	1	3	6	2

Πίνακας 3.2: Αξιολόγηση των μετροπρογραμμάτων της σουίτας polybench

Λαμβάνοντας υπόψη τον πίνακα 3.2, μπορούμε να δούμε ότι τα μισά από τα benchmarks βελτιστοποιήθηκαν με επιτυχία είτε με βελτιστοποιήσεις κώδικα που βελτιώνουν τις δυνατότητες πρόσβασης σε μνήμη και παραλληλισμό είτε με την εισαγωγή πολλαπλών νημάτων που επέτρεψαν τα εκτελέσιμα να τρέχουν παράλληλα καθήκοντα νωρίτερα. Αυτός είναι ένας καλός δείκτης για την αποτελεσματικότητα των εργαλείων και τον αντίκτυπο της αξιοποίησής τους από τους προγραμματιστές εφαρμογών που περιλαμβάνουν μεγάλα υπολογιστικά φορτία.

Από την άλλη πλευρά, μπορούμε να παρατηρήσουμε ότι περίπου το 1/3 των κριτηρίων αξιολόγησης φαίνεται να έχει ελάχιστο κέρδος από τη χρήση των εργαλείων ή, σε ορισμένες περιπτώσεις, να επιδεικνύει ακόμη και μια μεγάλη επιβάρυνση που εισάγεται με την προσθήκη πολλαπλών νημάτων. Φυσικά, υπάρχουν και μερικά αποτελέσματα στο μεσαίο έδαφος, όπου μπορούμε να δούμε κάποια βελτίωση με τη χρήση ενός μικρού ποσού νημάτων και άλλων όπου χρειαζόμαστε ακόμη περισσότερα νήματα για να δούμε κάποια βελτίωση.

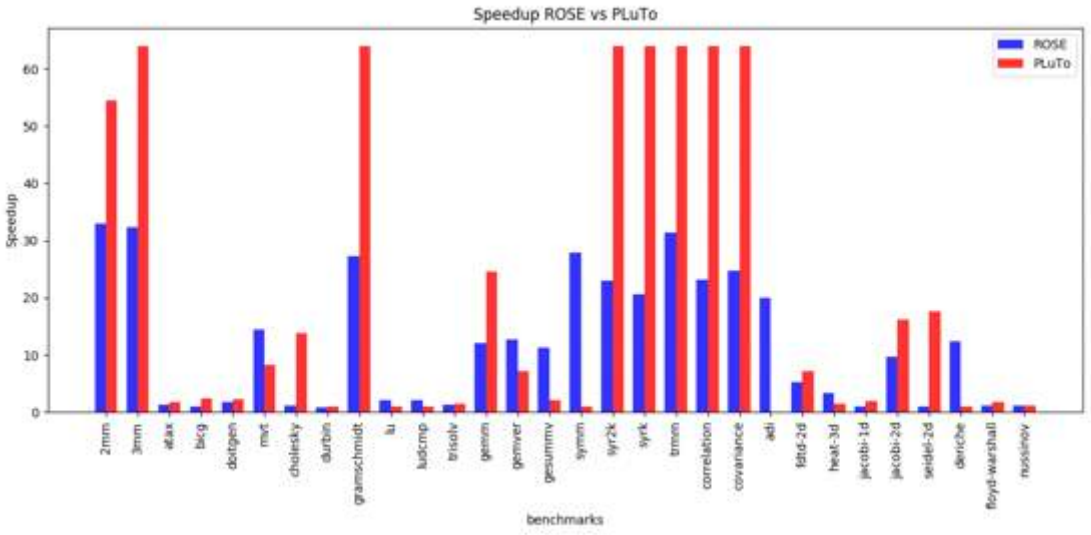
Ανάλυση επιτάχυνσης

Στο Σχήμα 3.4, η συνολική επιτάχυνση του κάθε μετροπρογράμματος έχει υπολογιστεί από τον τύπο:

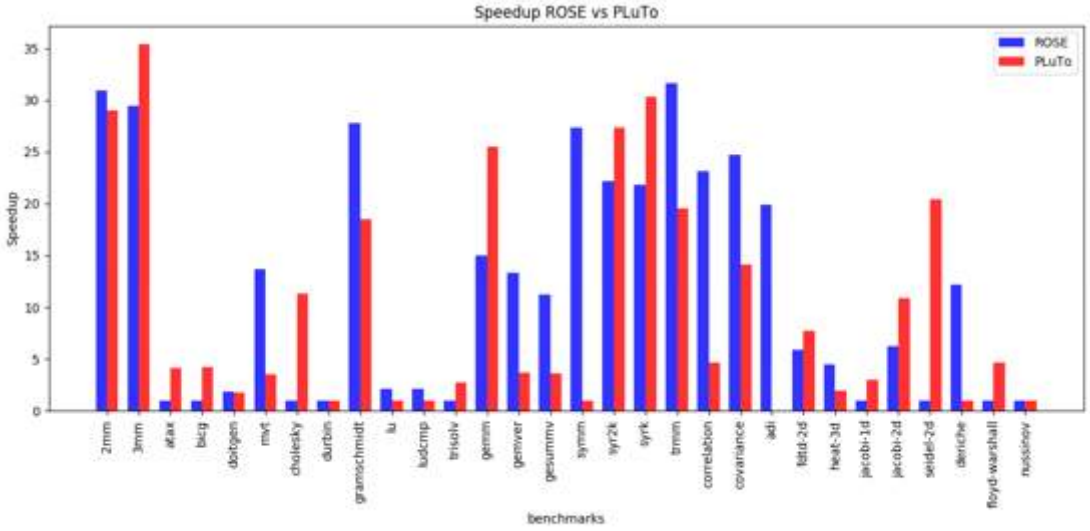
$$S_{speedup} = \frac{Initial\ Execution\ Time}{min_x(Parallelized\ version\ running\ using\ x\ threads)} \tag{3.9}$$

όπου x = 1,2,4,8,16,32,64 νήματα

Όπως αναμενόταν, το PLuTo επιδεικνύει μερικά εξαιρετικά κέρδη απόδοσης σε σύγκριση με τα αποτελέσματα ROSE (Τα σχήματα 3.4 έχουν εξομαλυνθεί σε περικοπές ταχύτητας μεγαλύτερες από x64 για να διευκολύνουν την απεικόνιση.



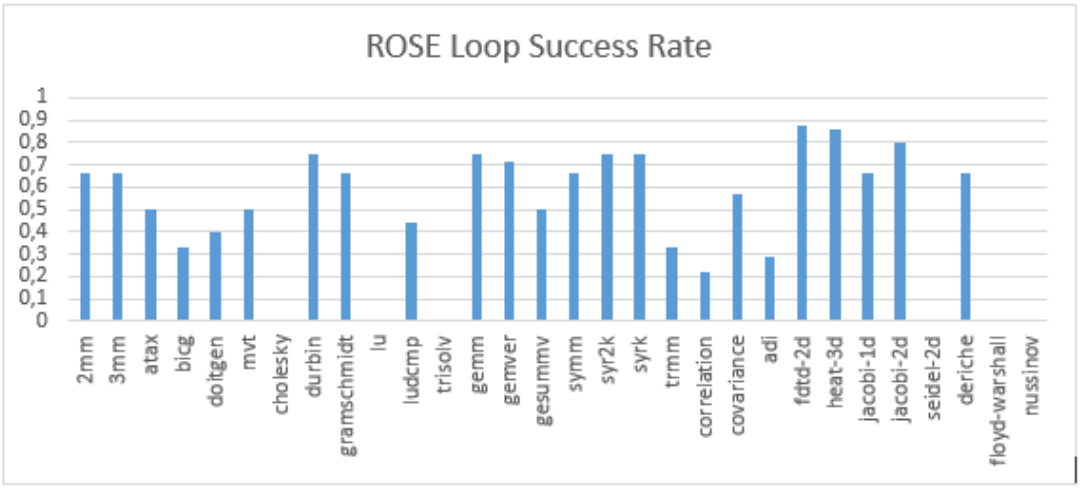
Σχήμα 3.4: Τελικό κέρδος σε επιτάχυνση λόγω βελτιστοποιήσεων και παραλληλοποίησης



Σχήμα 3.5: Επιτάχυνση σε σύγκριση με την απόδοση της παραλληλοποιημένης έκδοσης κώδικα, με τη χρήση ενός νήματος

Για τη μέτρηση της επεκτασιμότητας που προσφέρεται από κάθε εργαλείο, συγκρίναμε επίσης την καλύτερη απόδοση κάθε έκδοσης του κώδικα με την απόδοση που επιτυγχάνεται από την εκτέλεση της ίδιας έκδοσης χρησιμοποιώντας το νήμα. Με αυτόν τον τρόπο καταφέραμε να εξετάσουμε τους μετασχηματισμούς βρόχων και τις βελτιστοποιήσεις της τοπικής μνήμης, προσδιορίζοντας τα πραγματικά κέρδη απόδοσης που επιτύχαμε από την αύξηση του αριθμού των νημάτων για κάθε έκδοση. Τα αποτελέσματα φαίνονται στο Σχήμα 3.5.

Εδώ μπορούμε να παρατηρήσουμε ότι η διαφορά που παρατηρήσαμε στη συνολική απόδοση των δύο εργαλείων δεν είναι τόσο μεγάλη όσο και πριν. Αν αγνοήσουμε τα κέρδη απόδοσης από τους μετασχηματισμούς κώδικα, μπορούμε να δούμε ότι το ROSE φαίνεται να βελτιώνεται πολύ καλύτερα από ότι στο παρελθόν σε σχέση με το PLuTo, του οποίου η απόδοσή του εξακολουθεί να είναι μεγάλη σε ορισμένες περιπτώσεις, αλλά όχι τόσο μεγάλη ώστε να θεωρείται καλύτερη επιλογή. Συγκεκριμένα, μπορούμε να δούμε τις περιπτώσεις όπου το ROSE ξεπερνά το PLuTo και παρατηρούμε ότι υπάρχει



Σχήμα 3.6: Απόδοση του ROSE στην παραλληλοποίηση βρόχων

μεγάλη διαφορά (βλέπε mvt, gramschmidt, gemver, gessumv, symm, trmm, deriche), ενώ αντίθετα μπορούμε να δούμε ότι το PLuTo είναι καλύτερο, Το ROSE βρίσκεται λίγο πιο κάτω, πράγμα που σημαίνει ότι το κέρδος είναι παρόμοιο.

3.3.2 Εφαρμογές από τον 'πραγματικό κόσμο'

Εργαλεία παραλληλοποίησης

Ο PLuTo parallelizer έχει αποδειχθεί ότι είναι ένα αποτελεσματικό εργαλείο για την παραλληλισμό των εφαρμογών με τη λειτουργία μετασχηματισμού κώδικα που εισάγει καθώς και τις εγχύσεις OpenMP στον κώδικα. Ωστόσο, όταν μιλάμε για εφαρμογές σε πραγματικό κόσμο, πρέπει να έχουμε κατά νου ότι οι γλώσσες υψηλότερου επιπέδου προτιμώνται συνήθως για υπολογιστικές εφαρμογές, καθιστώντας την υποστήριξη της PLuTo για τη γλώσσα χαμηλού επιπέδου C μια λύση περισσότερο προσανατολισμένη στην έρευνα.

Από την άλλη πλευρά, ο μεταγλωττιστής ROSE είναι σε θέση να υποστηρίξει εφαρμογές C ++ και όπως τονίσαμε στην προηγούμενη παράγραφο, το κέρδος που έχουμε στη γενική περίπτωση όταν χρησιμοποιούμε το PLuTo αντί να μην είναι μεγάλο όταν μιλάμε για ένα μεγάλο αριθμό νημάτων . Το εργαλείο autoPar του ROSE λειτουργεί κυρίως με έγχυση του κώδικα με pragmas OpenMP στις περιοχές που θεωρούνται εφαρμόσιμες από άποψη που σχετίζεται με την εξάρτηση. Για να ελεγχθεί η δυνατότητα εφαρμογής του εργαλείου, πραγματοποιήθηκε μια δοκιμή αξιολόγησης προκειμένου να μετρηθούν οι συνολικοί βρόχοι που προσδιορίζονται από το εργαλείο ως παράλληλο. Ο λόγος για τον οποίο θεωρήθηκε σημαντικό να πραγματοποιηθεί μια τέτοια ανάλυση ήταν η εξέταση του εργαλείου σχετικά με την αποτελεσματικότητά του σε διάφορα είδη βρόχων με διαφορετικές εξάρσεις. Έτσι, αν το εργαλείο είναι σε θέση να παραλληλισθεί ένα μεγάλο μέρος των αναγνωρισμένων βρόχων μπορεί να θεωρηθεί εφαρμόσιμο σε ένα ευρύ φάσμα υπολογιστικών εργασιών.

Όπως φαίνεται στην Εικόνα 3.6, η πλειονότητα των αναγνωρισμένων βρόχων στα σημεία αναφοράς πολυβανίου θεωρήθηκε ότι δεν εξαρτώνται από την εξάρτηση, με αποτέλεσμα το ποσοστό παραλληλισμού του βρόχου να υπερβαίνει τα 60% για τα περισσότερα σημεία αναφοράς, ποσοστό είναι κοντά στο 50%. Αυτά τα αποτελέσματα, σε συνδυασμό με την πραγματική επιτάχυνση που παρατηρήθηκε στα σημεία αναφοράς, ήταν αρκετά ενθαρρυντικά ώστε να μας επιτρέψουν να προχωρήσουμε στην επόμενη φάση δοκιμών όπου δοκιμάσαμε μια σειρά εφαρμογών πραγματικού κόσμου χρησιμοποιώντας το ROSE για να αξιολογήσουμε την απόδοσή του.

Δοκιμάζοντας τις δυνατότητες του μεταγλωττιστή ROSE, χρησιμοποιήσαμε μια εφαρμογή που σχετίζεται με την κατακόρυφη επιτήρηση που επικεντρώνεται στην ανάλυση των δορυφορικών εικό-

```
vector<vector<int32_t> > replaceBlock(vector<vector<int32_t> > &mat_1,
                                      int32_t firstRow,
                                      int32_t lastRow,
                                      int32_t firstColumn,
                                      int32_t lastColumn,
                                      vector<vector<int32_t> > &mat_2)
{
    vector<vector<int32_t> > result = mat_1;
    int32_t mat2_i = 0;
    int32_t i, j;
    for (i = firstRow; i <= lastRow; i++) {
        int32_t mat2_j = 0;
        for (j = firstColumn; j <= lastColumn; j++) {
            result[i][j] = mat_2[mat2_i][mat2_j];
            mat2_j++;
        }
        mat2_i++;
    }
    return result;
}
```

Σχήμα 3.7: Κομμάτι κώδικα από την εφαρμογή surveillance - ταυτοποίηση μεταβλητής επανάληψης

νων. Ο κώδικας της εφαρμογής, λόγω της φύσης της, περιλαμβάνει μεγάλα ποσά βρόχων και υπολογισμών πίνακα που μας επιτρέπει να αξιολογήσουμε εύκολα την απόδοση του εργαλείου. Με τη δοκιμή του εργαλείου ενάντια σε αυτά τα χαρακτηριστικά κώδικα, έχουμε την ευκαιρία να επιθεωρήσουμε ορισμένα προβλήματα που συμβαίνουν με ρεαλιστικές εφαρμογές όπου η χρήση εργαλείων παραλληλισμού θα δημιουργούσε σημαντικά κέρδη απόδοσης.

Όταν ασχολούμαστε με πραγματικές εφαρμογές, πρέπει να λάβουμε υπόψη ότι ο κώδικας που μελετάμε δεν είναι πάντα βελτιστοποιημένος και σε πολλές περιπτώσεις δεν είναι εφικτό να παράγουμε τα αποτελέσματα που περιμένουμε. Ένα τέτοιο παράδειγμα μπορεί να παρουσιαστεί στο Σχήμα 3.7.

Στο Σχήμα 3.7, μπορούμε εύκολα να παρατηρήσουμε ότι οι δύο βρόχοι μπορούν εύκολα να παραλληλιστούν. Ωστόσο, οι μεταβλητές `mat2_i, mat2_j` δημιουργούν εξαρτήσεις που δεν μπορούν εύκολα να παρατηρηθούν από τα αυτόματα εργαλεία. Αυτό οφείλεται στο γεγονός ότι οι μεταβλητές δεν αναγνωρίζονται ως iterators του βρόχου (που θα λύσουν τις εξαρτήσεις), παρόλο που στην πραγματικότητα είναι, έτσι ώστε ο χειρισμός τους εντός του βρόχου να μην επιτρέπει στο εργαλείο να εγγυηθεί την παράλληλη παραλλαγή τους χωρίς εξάρτηση. Αντίθετα, αν ο προγραμματιστής είχε χρησιμοποιήσει τις λειτουργίες του i και j αντί των μεταβλητών (`mat2_i = i - firstRow, mat2_j = j - firstColumn` για τον παραλληλιστή και παρατηρηθούσε, οι εξαρτήσεις θα επιλύονταν.

Ένα άλλο πρόβλημα που μπορούμε να παρατηρήσουμε εύκολα στο προαναφερθέν τμήμα κώδικα είναι η ρητή χρήση δυναμικών δομών δεδομένων, διανυσμάτων, τα οποία επιτρέπουν στον προγραμματιστή να τροποποιεί δυναμικά τον διαθέσιμο χώρο για τα δεδομένα δίνοντας μεγαλύτερη ευελιξία στη χειραγώγηση της δομής. Από την άλλη πλευρά, η χρήση τέτοιων δομών υποδηλώνει τη δυναμική τους κατάσταση και συνεπώς δημιουργεί πρόσθετες εξαρτήσεις. Στο συγκεκριμένο παράδειγμα, μπορούμε ακόμη να παρατηρήσουμε ότι η δυναμική φύση των διανυσμάτων δεν χρησιμοποιείται καθόλου, πράγμα που σημαίνει ότι ο προγραμματιστής θα μπορούσε να χρησιμοποιήσει εύκολα τις στατικές δομές δεδομένων όπως οι πίνακες για την υλοποίηση αυτής της λειτουργίας, επιτρέποντας έτσι την παράλληλη παραλλαγή του.

Αυτά τα ”μικρά” αλλά σημαντικά προβλήματα προκύπτουν εύκολα σε εφαρμογές που έχουν σχεδιαστεί υπό πραγματικές συνθήκες και πρέπει να λαμβάνονται σοβαρά υπόψη κατά την αξιολόγηση ενός αυτόματου παραλληλιστή. Για παράδειγμα, ο μεταγλωττιστής ROSE που επιλέξαμε να χρησιμοποιήσουμε δεν εφαρμόζει μετασχηματισμούς στον κώδικα για την εξάλειψη τέτοιων εξαρτήσεων, ενώ το PLuTo εφαρμόζει πολύπλοκα μοντέλα μετασχηματισμού (δηλ. Πολυεδρικά) για να παραλληλίσουν τον κώδικα πιο αποτελεσματικά.

Κοινά προβλήματα

Σε αυτή την ενότητα, απαριθμούμε τα πιο σημαντικά χαρακτηριστικά που μπορούν να αναγνωριστούν στον κώδικα ικανά να δημιουργήσουν ανεπίλυτες εξαρτήσεις και να παρεμποδίσουν συνολικά τη διαδικασία παραλληλισμού:

- Έμμεση πρόσβαση στη μνήμη: Το πρόβλημα στην περίπτωση αυτή εμφανίζεται όταν έχετε πρόσβαση σε διευθύνσεις μνήμης που δεν είναι γνωστές σε χρόνο μετεγλώττισης. Παραλληλοποίηση κώδικα που περιλαμβάνει τέτοια χαρακτηριστικά μπορεί να προκαλέσει πρόσβαση στον ίδιο χώρο μνήμης πολλές φορές παράλληλα σε τυχαία σειρά προκαλώντας ασυνέπειες.

- Παρενέργειες: Οι κλήσεις συναρτήσεων με άγνωστες παρενέργειες στον κώδικα εμποδίζουν την παράλληλοποίησή του, καθώς ο μεταγλωττιστής δεν έχει την πλήρη εικόνα της διαχείρισης χώρου μνήμης στις λειτουργίες της συνάρτησης.

- Εξαρτήσεις ροής δεδομένων: Οι εξαρτήσεις των δεδομένων στη ροή εντολών του κώδικα δεν επιτρέπουν τη σωστή παραλληλισμό του, δημιουργώντας περιορισμούς στη σειρά εκτέλεσης των εντολών και στην παράλληλη εκτέλεση τους.

- Δυναμικές συστοιχίες (arrays) / δομές δεδομένων: Η χρήση δυναμικών δομών δεδομένων περιλαμβάνει σημαντικό περιορισμό στην παραλληλοποίηση του κώδικα. Όταν επεξεργάζεται δυναμικά το χώρο μνήμης, ο μεταγλωττιστής δεν έχει καμία εικόνα της κατάστασης της δομής, συνεπώς οι οδηγίες που εκτελούνται παράλληλα μπορεί να προκαλέσουν ασυνέπειες και πρόσβαση σε μη δεσμευμένο χώρο μνήμης.

Είναι προφανές ότι τα αυτόματα εργαλεία παραλληλοποίησης και τεχνικές που είναι διαθέσιμες δεν μπορούν να δημιουργήσουν τις απαιτούμενες συνθήκες για την πλήρη επίλυση των προαναφερθέντων προβλημάτων. Ορισμένα από αυτά μπορούν να επιλυθούν πλήρως ή εν μέρει, ενώ άλλα απαιτούν συγκεκριμένες συμβιβασμούς που αντιδρούν στην αυτοματοποίηση της διαδικασίας, καθώς δεν υπάρχει σαφής τρόπος μεταφοράς των γνώσεων που ανήκει ο προγραμματιστής στον παραλληλιστή. Στο επόμενο τμήμα, θα περιγράψουμε κάποιες από αυτές τις συμφωνίες και τα κέρδη που θα μπορούσαν να προσφέρουν στη διαδικασία παραλληλισμού.

3.3.3 Εισαγωγή σε ένα εκτεταμένο μοντέλο προγραμματισμού

Οι εξαρτήσεις ροής δεδομένων είναι μερικά από τα πιο κοινά και σημαντικά χαρακτηριστικά που μπορούν να μπλοκάρουν τη διαδικασία παραλληλισμού. Ωστόσο, υπάρχουν τρόποι αντιμετώπισης εν μέρει με κατάλληλους μετασχηματισμούς. Η χρήση εργαλείων όπως PLuTo, ικανών για τέτοιες βελτιστοποιήσεις, μπορεί να εξαλείψει πολλές εξαρτήσεις και να ξεκλειδώσει πολλές δυνατότητες παραλληλισμού.

Από την άλλη πλευρά, οι συνήθεις πρακτικές κωδικοποίησης περιλαμβάνουν ορισμένα χαρακτηριστικά (έμμεση διεύθυνση, κλήσεις λειτουργίας, δυναμικοί φορείς) που αποκρύπτουν ένα μεγάλο μέρος της λειτουργικότητας του προγράμματος καθιστώντας το πιο περίπλοκο και εμποδίζοντας την ανάλυσή του από τον παραλληλιστή. Φυσικά, υπάρχουν περιπτώσεις όπου χρησιμοποιούνται τόσο πολύπλοκα χαρακτηριστικά, αλλά λόγω της φύσης της εφαρμογής, δεν υπάρχουν πραγματικές εξαρτήσεις. Ωστόσο, λόγω της έλλειψης κατανόησης της λειτουργικότητας από τον παραλληλιστή, αυτές οι εξαρτήσεις εξακολουθούν να εντοπίζονται και να μην επιλύονται. Προς αυτή την κατεύθυνση προτείνεται εδώ ένα εκτεταμένο μοντέλο προγραμματισμού, ώστε ο χρήστης να μπορεί εύκολα να περιλαμβάνει πληροφορίες σχετικά με τον κώδικα που δεν μπορεί να εξαχθεί ρητά. Με αυτό τον τρόπο, ο προγραμματιστής είναι σε θέση να ”αλληλεπιδράσει” με τον παραλληλιστή και να προσθέσει λεπτομέρειες σχετικά με τις λειτουργίες πρόσβασης στη μνήμη, τη στατικότητα των διανυσμάτων και τις παρενέργειες της λειτουργίας, προκειμένου να ξεκλειδώσουν περισσότερες δυνατότητες παραλληλισμού. Το μοντέλο περιλαμβάνει αυτές τις πληροφορίες με τη μορφή σημείων pragma όπως περιγράφεται παρακάτω:

```
for (int i = startRow; i <= endRow; i++) {
    for (int j = startCol; j <= endCol; j++) {
        if (cond[i][j]) {
            // Parallelisable Loop
            for (int k = 0; k < rowBlockIndex.size(); k++) {
                result[i + rowBlockIndex[k]][j + colBlockIndex[k]] = true;
            }
        }
    }
}
```

Σχήμα 3.8: Κομμάτι κώδικο από τη surveillance εφαρμογή - έμμεση προσπέλαση μνήμης

#pragma aeolus function/loop static-vectors
Εγγυάται ότι η ακόλουθη συνάρτηση / βρόχος περιλαμβάνει μόνο στατικούς vectors, πράγμα που σημαίνει ότι δεν υπάρχει δυναμική κατανομή μνήμης κατά την εκτέλεση.

#pragma aeolus loop no-aliasing
Εγγυάται ότι η συνάρτηση / βρόχος δεν περιλαμβάνει aliasing δείκτη, που σημαίνει πρόσβαση στην ίδια διεύθυνση μνήμης από διαφορετικούς δείκτες.

#pragma aeolus function no-side-effects
Εγγυάται ότι η ακόλουθη συνάρτηση δεν περιλαμβάνει παρενέργειες, που σημαίνει ότι δεν υπάρχει γραφή στις διευθύνσεις μνήμεις ή σε streams, κι έτσι η ροή του προγράμματος δεν επηρεάζεται πέρα από την τιμή επιστροφής της συνάρτησης.

Το επόμενο τμήμα κώδικα εμφανίζει ένα καλό παράδειγμα ψευδούς ψευδαισθήματος στην πράξη όπου η χρήση των συστοιχιών 'rowBlockIndex' και 'colBlockIndex' για έμμεση πρόσβαση στη συστοιχία 'result' δημιουργεί ψευδείς εξάρσεις που δεν μπορούν να επιλυθούν με τις τρέχουσες τεχνικές παραλληλοποίησης.

Προσθέτοντας εδώ το σχολιασμό «no-aliasing» μπορεί να καθοδηγήσει τον παραλληλιστή ότι οι εξάρσεις που σχετίζονται με την έμμεση πρόσβαση μπορούν να εξαλειφθούν με ασφάλεια.

Η έξοδος αυτής της εργασίας σχετικά με τη χρήση αυτού του εκτεταμένου μοντέλου προγραμματισμού για τη βελτιστοποίηση των αποτελεσμάτων που λαμβάνονται από το ROSE είναι ένα εργαλείο Auto-Parallelization που αναφέρεται και πάλι στο πλαίσιο του κεφαλαίου 5.

3.3.4 Εκτεταμένη Σουίτα Benchmark

Προκειμένου να ελεγχθεί περαιτέρω η βελτίωση που επιτυγχάνεται κατά τη χρήση του μοντέλου εκτεταμένου προγραμματισμού, εξήγαμε ένα σύνολο λειτουργιών που απαιτούν υπολογιστικές ενέργειες που περιέχουν τέτοια εξαρτήματα από πραγματικές εφαρμογές. Χρησιμοποιήσαμε αυτά τα τμήματα για να συμπεριλάβουμε μερικά πραγματικά παραδείγματα κώδικα που μπορούν να βρεθούν σε εφαρμογές και να επεκτείνουμε τη λίστα αναφοράς μας με πρόσθετα χαρακτηριστικά κώδικα που πρέπει να υποστηρίξει ένα εργαλείο παραλληλισμού. Μερικά παραδείγματα από αυτόν τον κατάλογο μπορείτε να βρείτε παρακάτω.

```
1 vector<double> replaceIndexByVector(vector<double> &vec, vector<int32_t> &
    index_vec, vector<double> &value) {
2     int32_t i;
3     int32_t index_size = index_vec.size();
4     vector<double> result = vec;
5     for (i = 0; i < index_size; i++)
6         result[index_vec[i]] = value[i];
7     return result;
8 }
```

Στο παραπάνω τμήμα, εμφανίζεται ένας κωδικός από ένα σημείο αναφοράς, συμπεριλαμβανομένων των χαρακτηριστικών έμμεσης διευθυνσιοδότησης. Εδώ, ο διάνυσμα 'αποτελέσματος' είναι έμμεσα προσπελάσιμος με το διάνυσμα '$index_vec$' ως ευρετήριο. Για λογικούς λόγους, γίνεται φανερό ότι δεν υπάρχει αλληλεπικάλυψη μεταξύ διαφορετικών επαναλήψεων βρόχου, οπότε όταν εισάγουμε τον σχολιασμό 'no-aliasing' οι εξάρσεις εξαλείφονται. Επιπλέον, μπορούμε να προσθέσουμε τον σχολιασμό 'static-vector' για να δηλώσουμε τη στατικότητα των χρησιμοποιούμενων διανυσμάτων κατά την εκτέλεση του κώδικα.

```
int32_t square(int32_t x) {
    return x*x;
}
int32_t square_sum(vector<int32_t> &vec) {
    int32_t i;
    int32_t vec_size = vec.size();
    int32_t result = 0;
    for (i = 0; i < vec_size; i++)
        result += square(vec[i]);
    return result;
}
```

Ομοίως, στο παραπάνω τμήμα μπορούμε να δούμε πώς η χρήση των σχολιασμών 'no-side-effects' μπορεί να επιλύσει εξαρτήσεις που συνδέονται με τις άγνωστες παρενέργειες των αποκαλούμενων λειτουργιών.

Κεφάλαιο 4

Παράλληλο Προγραμματιστικό Μοντέλο

Όπως τονίσαμε σε προηγούμενες ενότητες, η ανάπτυξη μιας παράλληλης εφαρμογής σε μη ομοιόμορφες αρχιτεκτονικές μπορεί να είναι ένα πολύ σύνθετο και χρονοβόρο έργο. Περιλαμβάνει την προσαρμογή διαφόρων μοντέλων προγραμματισμού και διαφορετικών πρωτοκόλλων επικοινωνίας καθώς και την ενσωμάτωση διαφορετικών αρχιτεκτονικών. Σε αυτή την ενότητα θα αναλύσουμε τη διαδικασία που πρέπει να ακολουθήσουμε κατά την ανάπτυξη μιας παράλληλης εφαρμογής και τη συγκρίνουμε με αυτήν που χρησιμοποιούμε όταν χρησιμοποιούμε ένα ενιαίο μοντέλο παράλληλου προγραμματισμού. Στη συνέχεια, θα παρουσιάσουμε το σχεδιασμό ενός μοντέλου παράλληλου προγραμματισμού και τις λεπτομέρειες που κάνουν την εφαρμογή του.

4.1 Ένα Παράλληλο Προγραμματιστικό Μοντέλο: Σχεδιασμός

Προκειμένου να διευκολυνθεί η βελτιστοποίηση των αποτελεσμάτων ανάπτυξης από την άποψη της απόδοσης και της οικονομικής απόδοσης, καθώς και η ευελιξία του συστήματος να εκμεταλλευτεί διάφορα μέρη του διαθέσιμου υλικού, αναλαμβάνουμε ένα σύνολο ξεχωριστών συστατικών εφαρμογών που εκτελούνται ανεξάρτητα, σε συνέργεια χρησιμοποιώντας μια καθορισμένη διεπαφή (η οποία περιγράφεται στην ενότητα 4.1.3) για την εκτέλεση μιας συγκεκριμένης λειτουργικότητας. Η σημασία αυτής της απόφασης έγκειται στο γεγονός ότι το εργαλείο θα πρέπει να είναι σε θέση να διανείμει το φορτίο υπολογισμού με διαφορετικούς τρόπους στο διαθέσιμο υλικό για να βελτιστοποιήσει τη συνολική απόδοση της εφαρμογής.

Για παράδειγμα, ας εξετάσουμε μια μικρή εφαρμογή με δύο στοιχεία A και B και ένα σύστημα 2 CPU με μια διαθέσιμη GPU. Μια πιθανή χαρτογράφηση θα περιλάμβανε κάθε συνιστώσα που λειτουργούσε σε μια (διαφορετική) CPU. Μια διαφορετική θα πρότεινε το στοιχείο A να εκτελεστεί στη CPU1 και το στοιχείο B στην CPU2 με το τελευταίο να χρησιμοποιεί τη GPU για να εκτελέσει μέρος του φορτίου υπολογισμού σε αυτό.

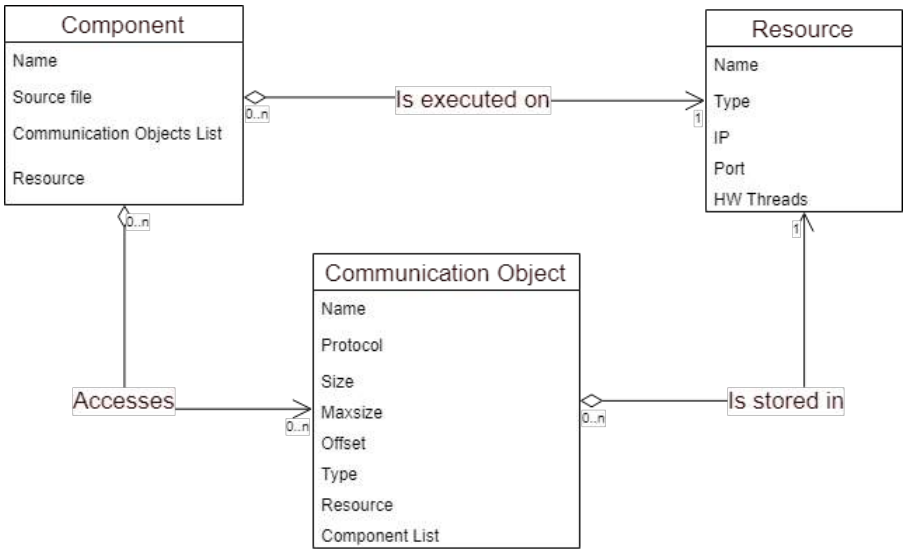
Έτσι, ο χρήστης μπορεί να δοκιμάσει διαφορετικές αντιστοιχίσεις μεταξύ των στοιχείων και των διαθέσιμων πόρων για να αναπτύξει την εφαρμογή και να οδηγήσει σε μια αποτελεσματική απόφαση που να ικανοποιεί τις απαιτήσεις τους.

Για να δομηθεί καλύτερα αυτό το μοντέλο, έχει καθοριστεί ένα σύνολο από κλάσεις στις οποίες δηλώνονται όλες οι απαραίτητες πληροφορίες τόσο για τα κομμάτια της εφαρμογής όσο και για τις ανταλλαγές δεδομένων που πραγματοποιούνται μεταξύ τους. Αυτές οι κλάσεις περιγράφουν δύο συγκεκριμένες οντολογίες: τα κομμάτια της εφαρμογής (Components) και τα αντικείμενα επικοινωνίας της εφαρμογής (Communication objects).

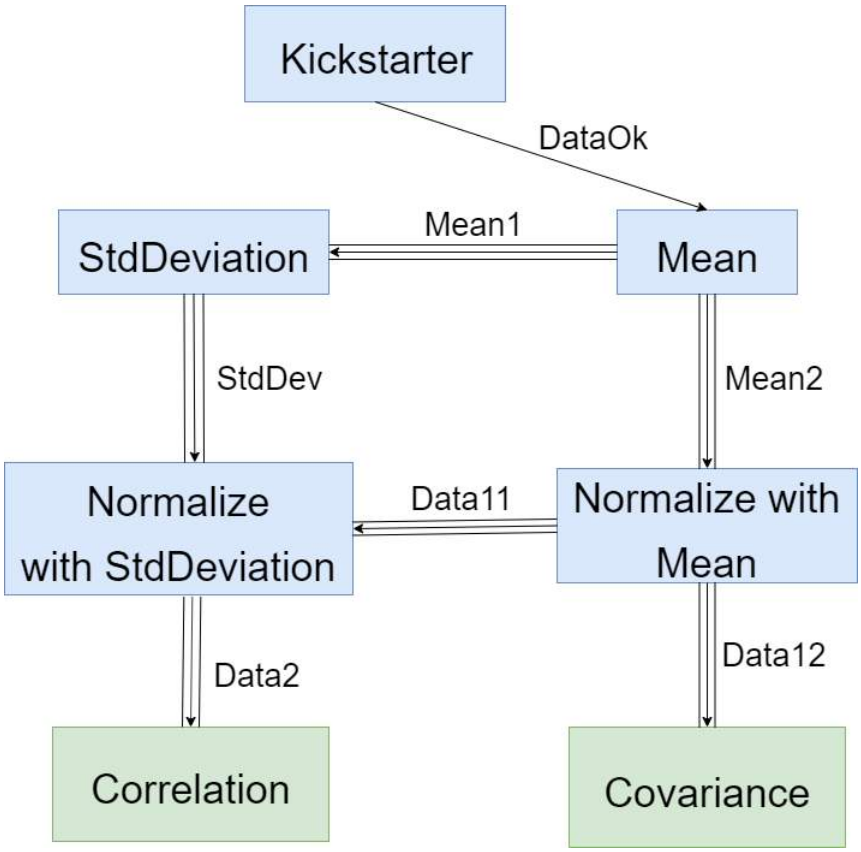
Components: Ένα component ορίζεται ως ξεχωριστό, ανεξάρτητο μέρος της εφαρμογής με συγκεκριμένα χαρακτηριστικά όπως το όνομα και το αναγνωριστικό που είναι σε θέση να αλληλεπιδρά με άλλα συστατικά μέσω αντικειμένων επικοινωνίας.

Communication Objects: Ένα communication object ορίζεται είτε ως συναλλαγή δεδομένων μεταξύ συστατικών με συγκεκριμένα χαρακτηριστικά όπως μέγεθος δεδομένων, τύπος δεδομένων, προέλευση και στόχος ή ως ενέργεια συγχρονισμού που στοχεύει στο συντονισμό των στοιχείων που το δεσμεύουν.

Οι κλάσεις παρουσιάζονται στο Σχήμα 4.1.



Σχήμα 4.1: Διάγραμμα Κλάσεων Μοντέλου Εφαρμογής



Σχήμα 4.2: Παράδειγμα Component Network

Στο σχήμα 4.2, μπορούμε να δούμε ένα παράδειγμα του τρόπου με τον οποίο οι περιπτώσεις αυτών των κλάσεων μπορούν να σχηματίσουν ένα δίκτυο του οποίου τα μέλη σχετίζονται μεταξύ τους με τον τρόπο που περιγράφεται στο Σχήμα 4.1.

4.1.1 Μοντέλο Εφαρμογής

Για να μπορέσει ο χρήστης να περιγράψει την εφαρμογή του χρησιμοποιώντας το μοντέλο που παρουσιάστηκε παραπάνω και να το υποβάλει ως είσοδο για το εργαλείο, έχει σχεδιαστεί ένα αρχείο XML ακολουθώντας αυτό το μοντέλο, το οποίο περιλαμβάνει όλες τις απαραίτητες πληροφορίες που απαιτούνται για τα στοιχεία και τα αντικείμενα επικοινωνίας της αίτησης. Ένα παράδειγμα αυτού του αρχείου εμφανίζεται παρακάτω:

```
<application name="tutorial" value="general-purpose">

<!--================== Components description================ -->

<component name="C0" type="asynchronous">
    <source file="C0.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>
<component name="C1" type="asynchronous">
    <source file="C1.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>
<component name="C2" type="asynchronous">
    <source file="C2.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>

<!--==================Communication objects description================ -->

<comm-object item-size="16793807" name="Queue0" object-class="FIFO" size="5" type
    ="Queue">
        <source name="C0" port-name="inQueue00" type="in"/>
        <target name="C1" port-name="outQueue01" type="out"/>
        <target name="C2" port-name="outQueue02" type="out"/>
</comm-object>
<comm-object item-size="4" name="Shared0" object-class="shared" size="10" type="
    Shared-Memory">
        <source name="C2" port-name="inShared0" type="in"/>
        <target name="C1" port-name="outShared0" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal0" object-class="signal" size="1" type="
    Signal">
        <source name="C0" port-name="inSignal0" type="in"/>
        <target name="C2" port-name="outSignal0" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal1" object-class="signal" size="1" type="
    Signal">
        <source name="C1" port-name="inSignal1" type="in"/>
        <target name="C2" port-name="outSignal1" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal2" object-class="signal" size="1" type="
    Signal">
        <source name="C2" port-name="inSignal2" type="in"/>
        <target name="C1" port-name="outSignal2" type="out"/>
</comm-object>

</application>
```

Μια εφαρμογή που χρησιμοποιεί το μοντέλο παράλληλου προγραμματισμού ορίζεται από δύο διαφορετικές οντότητες, στοιχεία λογισμικού και αντικείμενα επικοινωνίας. Ένα άλλο παράδειγμα εμφανίζεται στον παρακάτω τομέα:

```
<applicationname="Example_Application"xsi:noNamespaceSchemaLocation="./aeolus.xsd
    ">
<component name="A"  type="asynchronous">
   <implementation id="1" model="any" target-HW="CPU">
```

```
        <source lang="c" file="CB.h" path="src\components"/>
    </implementation>
</component>
<component name="B" type="asynchronous" >
    <implementation id="1" model="any" target-HW="CPU">
        <source lang="c" file="CB.h" path="src\components"/>
    </implementation>
</component>
<comm-object name ="B" type="Buffer" object-class="FIFO" size="128" item-size="8"
    >
    <source name="A" port-name="inA1" type="in"/>
    <target name="B" port-name="outB1" type="out"/>
</comm-object>
<comm-object name ="S" type="Signal" object-class="Control">
    <source name="A" port-name="inA2" type="in"/>
    <target name="B" port-name="outB2" type="out"/>
</comm-object>
<comm-object name ="Sh" type="Shared Memory" object-class="Memory" size="1024"
     item-size="32">
    <source name="A" port-name="inA3" type="in"/>
    <target name="B" port-name="outB3" type="out"/>
</comm-object>
</application>
```

Συστατικά (Components) της εφαρμογής

Μια εφαρμογή AEOLUS χωρίζεται σε διαφορετικά μέρη λογισμικού που εκτελούνται παράλληλα και ξεκινούν από την πλατφόρμα στην αρχή της εκτέλεσης. Κάθε component αντιστοιχεί σε ένα συγκεκριμένο πηγαίο αρχείο όπως ορίζεται στο Component Network, το οποίο πρέπει να έχει ένα σημείο εισόδου για την εκτέλεση του στοιχείου. Τα στοιχεία μπορούν να επαναχρησιμοποιηθούν πολλές φορές, εφόσον αυτό ορίζεται ρητά στο δίκτυο συνιστωσών. Ξεκινούν ως ξεχωριστές και ανεξάρτητες λειτουργίες.

Αντικείμενα επικοινωνίας

Για να ενεργοποιηθεί η επικοινωνία μεταξύ των διαφόρων εξαρτημάτων της εφαρμογής, το μοντέλο προγραμματισμού AEOLUS ορίζει μια άλλη οντότητα που καθορίζει συγκεκριμένα τις ανταλλαγές δεδομένων ή τις λειτουργίες συντονισμού που απαιτούνται από την εφαρμογή. Τέσσερις τύποι Αντικειμένων Επικοινωνίας έχουν αναπτυχθεί, Κοινόχρηστο, ουρά, σήμα, Mutex (περιγράφονται στην επόμενη ενότητα), οι οποίοι υλοποιούνται από τις αντίστοιχες διεπαφές. Τα πρωτόκολλα έχουν ενισχυθεί περαιτέρω για τη χρήση συγκεκριμένων τύπων δεδομένων για κάθε αντικείμενο επικοινωνίας, που ορίζεται από το περιβάλλον προγραμματισμού AEOLUS. Ειδικότερα, παρέχονται οι ακόλουθοι τύποι:

- aeolus_shared
- aeolus_queue
- aeolus_signal
- aeolus_mutex

4.1.2 Μοντέλο ανάπτυξης

Εκτός από το σχεδιασμό που θα πρέπει να ακολουθήσει η εφαρμογή για να ολοκληρώσει τη λειτουργικότητά της, θα πρέπει να παρέχεται από το χρήστη ένα μοντέλο ανάπτυξης, για τον εντοπισμό των διαφόρων στόχων υλικού και πλατφορμών που είναι διαθέσιμες και για τη χαρτογράφηση των συστατικών στοιχείων εφαρμογών σε αυτά για εκτέλεση σύμφωνα με κάθε ιδιότητες του στοιχείου.

Στοιχεία/Πόροι Υλικού (Hardware Elements)

Προκειμένου να δημιουργηθεί ένα πραγματικό μοντέλο των διαφορετικών σχεδίων ανάπτυξης που μπορούν να εκτελεστούν, πρέπει να ορίσουμε ένα σύνολο στοιχείων υλικού στα οποία μπορούν να αναπτυχθούν τα διάφορα συστατικά στοιχεία της εφαρμογής. Έτσι, απαριθμούμε και περιγράφουμε εδώ, τα διάφορα στοιχεία που υποστηρίζονται από το μοντέλο.

- **CPU – only (SMPs) processing nodes**
  Αυτά τα στοιχεία περιλαμβάνουν μόνο επεξεργαστές γενικής χρήσης (έναν ή περισσότερους) που λειτουργούν μόνο σε κοινή μνήμη. Αυτές οι περιγραφές περιλαμβάνουν πληροφορίες όπως τον πυρήνα του κάθε επεξεργαστή, τη συχνότητα κλπ.

- **CPU – GPU processing nodes**
  Παρόμοια με το μοντέλο CPU - only, το στοιχείο αυτό θεωρεί μια πρόσθετη συσκευή GPU συνδεδεμένη στον κόμβο και είναι εύκολα προσβάσιμη από τον επεξεργαστή γενικού σκοπού του κόμβου. Τα components της εφαρμογής που έχουν αντιστοιχιστεί για να εκτελεστούν σε αυτά τα στοιχεία χρησιμοποιούν την CPU για την εκτέλεση και την εκφόρτωση συγκεκριμένων υπολογισμών στη συσκευή για την αξιοποίηση των πόρων που είναι διαθέσιμοι για επιτάχυνση.

- **CPU – FPGA processing nodes**
  Παρόμοια με το μοντέλο μόνο CPU, αυτό το στοιχείο εξετάζει πρόσθετους λογικούς πόρους FPGA που μπορούν να διευκολύνουν τις λειτουργίες πρόσβασης μνήμης και να επιταχύνουν συγκεκριμένους υπολογισμούς. Και πάλι, τα εξαρτήματα λογισμικού λειτουργούν κυρίως στις λειτουργίες CPU και εκφορτώνουν συναρτήσεις που μπορούν να βελτιστοποιηθούν στους πόρους υλικού που είναι διαθέσιμοι στις FPGAs.

Περιγραφή πλατφόρμας

Η περιγραφή της πλατφόρμας μεταφέρει πληροφορίες υψηλού επιπέδου σχετικά με το υλικό, όπως το CPU, τα κανάλια μνήμης και επικοινωνίας στο σύστημα ή άλλες ιδιότητες που περιγράφουν πιο συγκεκριμένες συσκευές προορισμού όπως GPUs, FPGAs κ.λπ. Σκοπός αυτής της περιγραφής είναι να επιτρέψει στον AEOLUS να χειριστεί εύκολα μια ποικιλία αρχιτεκτονικών. Ένα παράδειγμα του αρχείου XML εμφανίζεται παρακάτω :.

```
<platform name="localhost" xsi:noNamespaceSchemaLocation="./hw_development.xsd">

<!-- localhost -->
<device name="DevelopmentMachine" type="CPU-SMP" reliability="5">
        <processing-node name="unit1" type="CPU-SMP" architecture="SMP"> <!--
    Intel Core i7-6700K -->
                <processor name="local" type="INTEL-COREi7">
                        <configuration name="core number" value="4"/>
                        <configuration name="cpu frequency" value="4.0" unit="GHz
    "/>
                        <configuration name="bytespercycle" value="1"/>
                        <memory name="LOCALMEM1" type="RAM" size="2048" size-unit
    ="MB" access-time="1" access-time-unit="ns/word"/> <!--assuming 8-byte word
    -->
                </processor>
        </processing-node>

        <comm_interface name="enp0s3" type="Ethernet Network">
                <configuration name="speed" value="100" units="MBit/s" ip="
    localhost" user="demo"/>
        </comm_interface>

</device>

</platform>
```

Το αρχείο XML της πλατφόρμας XML χρησιμοποιείται για τον καθορισμό της αρχιτεκτονικής του συστήματος, συμπεριλαμβανομένων όλων των χαρακτηριστικών που είναι χρήσιμα για την ανάπτυξη. Παρακάτω παρουσιάζεται μια περιγραφή πλατφόρμας CPU-FPGA:

```
<platform>
<device name="CPU–FPGA device" type="CPU–FPGA" reliability="2">
  <processing–node name="UIZynq–unit1" type="CPU–SMP" architecture="SMP">
    <processor name="UIZynq–P1" type="ARM–Cortex">
      <configuration name="core number" value="2"/>
      <configuration name="cpu frequency" value="800" unit="MHz"/>
      <configuration name="bytespercycle" value="1"/>
      <memory name="UIZynq–SM3" type="DDR3" size="1024" size–unit="MB" access–time="8" access–time–unit="ns/word"/>
    </processor>
  </processing–node>
  <processing–node name="UIZynq–unit2" type="FPGA" brand="Xilinx">
    <fpgalogic name="UIZynq–PL1" type="xc7z045ffg900–2">
      <resource name="UIZynq–LC" type="logiccell"/>
      <resource name="UIZynq–LUT" type="lookuptables"/>
      <resource name="UIZynq–LUTRAM" type="lookuptablesRAM"/>
      <resource name="UIZynq–FF" type="flipflop"/>
      <resource name="UIZynq–BRAM" type="blockRAM"/>
      <resource name="UIZynq–DSP" type="digitalsignalprocessing"/>
      <resource name="UIZynq–BUFG" type="bufferglobal"/>
      <configuration name="UIZynq–maxfrequency" value="200" units="MHz"/>
      <memory name="UIZynq–SM1" type="DDR3" size="1024" size–unit="MB" access–time="8" access–time–unit="ns/word"/>
    </fpgalogic>
  </processing–node>

  <local_bus name="UIZynq–AXI" type="AXI4" throughput="15" throughput–time–unit="Bytes/ns"/> <!––15GB/s––>
  <comm_interface name="UIZynq–EXT" type="Ethernet Network">
    <configuration name="UIZynq–speed" value="100" units="MBit/s" ip="192.168.1.2" user="external"/>
  </comm_interface>
</device>
</platform>
```

Σχέδιο Εκτέλεσης (Deployment Plan)

Προκειμένου να αναπτυχθεί η εφαρμογή στην διαθέσιμη υποδομή υλικού, ο χρήστης προσπαθεί να αντιστοιχίσει το συστατικό δίκτυο μέσω της περιγραφής της πλατφόρμας. Το αποτέλεσμα αυτής της διαδικασίας χαρτογράφησης είναι ένα σχέδιο ανάπτυξης. Οι αναπτύξεις εκπροσωπούνται στην ίδια μορφή που βασίζεται σε XML όπως συζητήθηκε προηγουμένως. Ένα παράδειγμα φαίνεται παρακάτω:

```
<deployment name="tutorial_Mapping" xmlns="DE" xsi:noNamespaceSchemaLocation="../models/deployment.xsd">

<!––            Estimated Execution Time:705.0  local:5.0mW  ––>
<!––            Estimated Reliability:5.0––>
<!––            Component Deployment          ––>
<target–application name="tutorial"/>
<target–hw–platform name="DevelopmentMachine"/>
<mapping name="component_C0_map" type="processing">
        <component name="C0" comp_id="0" subcomponents="1" secure="false"/>
        <CPU processor–name="local" device–type="CPU–SMP" CPU–type="SMP"/>
</mapping>
<mapping name="component_C1_map" type="processing">
        <component name="C1" comp_id="1" subcomponents="1" secure="false"/>
        <CPU processor–name="local" device–type="CPU–SMP" CPU–type="SMP"/>
```

```
15 </mapping>
16 <mapping name="component_C2_map" type="processing">
17         <component name="C2" comp_id="2" subcomponents="1" secure="false"/>
18         <CPU processor-name="local" device-type="CPU-SMP" CPU-type="SMP"/>
19 </mapping>
20
21 </deployment>
```

4.1.3 Προγραμματιστική Διεπαφή (Programming Interface)

Για να διευκολυνθεί η χρήση των πρωτοκόλλων επικοινωνίας, παρέχεται μια διεπαφή στον προγραμματιστή και περιγράφεται στις επόμενες παραγράφους. Παρατηρήστε τη διαφορά μεταξύ της διεπαφής χρήστη των πρωτοκόλλων και της πραγματικής διεπαφής των λειτουργιών πρωτοκόλλου που περιγράφεται στην ενότητα 'Υλοποίηση'. Τα API απαιτούν πληροφορίες για τις οποίες είναι υπεύθυνος ο Διαχειριστής Εκτέλεσης (Deployment Manager), προκειμένου να αυτοματοποιήσουν τη διαδικασία ανάπτυξης. Οι λεπτομέρειες σχετικά με το θέμα αυτό συζητούνται στην αντίστοιχη ενότητα όπου περιγράφεται η λειτουργικότητα του Διαχειριστή Εκτέλεσης.

Πρωτόκολλο Κοινής Μνήμης

Το πρωτόκολλο κοινής μνήμης (Shared Protocol) περιγράφει λειτουργίες ανταλλαγής δεδομένων μέσω της μνήμης που είναι ουσιαστικά κοινόχρηστη. Παρέχει στον χρήστη τη δυνατότητα να χειρίζεται τις μεταβλητές ως κοινές στο περιβάλλον του ολοκληρωμένου συστήματος. Το πρωτόκολλο χρησιμοποιεί το μοντέλο σταθερότητας της μνήμης που σημαίνει ότι οποιοσδήποτε διαβάζει ή γράφει την άμεση ενημέρωση της κοινόχρηστης μνήμης, οπότε ο προγραμματιστής είναι υπεύθυνος να καλέσει κατάλληλες λειτουργίες συγχρονισμού προκειμένου να παράσχει την απαραίτητη συνέπεια.

Επίσης, ο χρήστης δεν έχει άμεση πρόσβαση στην κοινόχρηστη μνήμη, οπότε η λειτουργία συγχρονισμού πρέπει να χρησιμοποιηθεί για τη μεταφορά δεδομένων προς και από αυτό. Εμφανίζεται εδώ:

*bool aeolus_synchronize(void *item, int dir)*
Προκαλεί την τοπική προβολή του στοιχείου να ενημερωθεί σύμφωνα με τα αντίστοιχα δεδομένα στην κοινή μνήμη, εάν η μεταβλητή dir έχει την τιμή 0. Στην αντίθετη περίπτωση, η κοινόχρηστη μνήμη ενημερώνεται σύμφωνα με την τοπική προβολή του στοιχείου.

Πρωτόκολλο Ουράς

Το Πρωτόκολλο Ουράς (Queue Protocol) παρέχει κατάλληλες λειτουργίες που επιτρέπουν στο χρήστη να διαχειρίζεται την επικοινωνία μεταξύ των εξαρτημάτων που συνδέονται με συγκεκριμένα αντικείμενα επικοινωνίας με τη μορφή ουρών και ειδικότερα με τη μορφή αποκλεισμού FIFOs αυθαίρετου μεγέθους. Οι λειτουργίες του πρωτοκόλλου Queue παρέχουν στο χρήστη τη δυνατότητα αποθήκευσης και προσπέλασης στοιχείων σε μια δομή ουράς και την καταμέτρηση του αριθμού των στοιχείων που υπάρχουν στην ουρά σε συγκεκριμένη χρονική στιγμή. Οι λειτουργίες που αναφέρονται στο τρέχον στάδιο είναι οι εξής:

*bool aeolus_queue_get(void *item)*
Pulls an item from the queue, storing it in the address pointed by item. Returns true if the transaction was successful.

*bool aeolus_queue_put(void *item)*
Pushes to the queue the item in the address pointed by item. Returns true if the transaction was successful.

*bool aeolus_queue_peek(void *item)*
Gets an item from the queue, storing it in the address pointed by item but not extracting it from the queue. Returns true if the transaction was successful.

*uint32_t aeolus_queue_count(void *queue)*
Returns the number of items that exist in the queue at the time of the call.

Note: The Queue Protocol has been developed to support object transfers of variable length. For that purpose, the objects are passed to the queue functions serialized including the size of the object in the 4 first bytes. Although the developers are free to develop their own functions for serialization, some prototype ones have been developed and are provided for the facilitation of the process.

Πρωτόκολλο σημάτων

Εκτός από την ανταλλαγή και την ανταλλαγή δεδομένων, περιλαμβάνεται επίσης το Πρωτόκολλο Σήμανσης το οποίο παρέχει στο χρήστη δυνατότητες σηματοδότησης, επιτρέποντας την συντονισμένη εκτέλεση στοιχείων και τμημάτων μέσα σε κάθε στοιχείο. Αυτές οι λειτουργίες συμπληρώνουν επίσης τις λειτουργίες των κοινόχρηστων πρωτοκόλλων και των ουρών, καθώς χρειάζεται επίσης να συγχρονίσουν την εκτέλεση τους μεταξύ των στοιχείων. Οι λειτουργίες επιτρέπουν στα εξαρτήματα να ανταλλάσσουν σήματα καθώς και να δημιουργούν φραγμούς συγχρονισμού, όπως περιγράφεται παρακάτω:

bool aeolus_wait(int src)
Blocks the current thread/process until the signal in question is notified by the thread/process with id=src.

bool aeolus_notify(int dst)
If dst = -1 unblock a random single thread/process waiting on the signal. Else unblock the thread/process with id=dst.

bool aeolus_notifyall()
Unblock all threads/processes waiting on the signal.

bool aeolus_barrier()
In addition, a barrier function is provided able to wait until all threads or processes before that call, have finished their work.

Πρωτόκολλο Mutex

Τα mutexes χρησιμοποιούνται για την επιβολή αμοιβαίου αποκλεισμού μεταξύ ενός συνόλου στοιχείων σε μια κρίσιμη περιοχή όπου χρησιμοποιούνται κοινά δεδομένα. Εφαρμόζονται σε επίπεδο συνιστωσών και δεν σχετίζονται με τα mutex σε επίπεδο OS από άλλα API, αλλά η χρήση τους είναι παρόμοια με αυτή που περιγράφεται στα πρότυπα POSIX. Ισχύει για τύπους δείκτη και πίνακες. Οι λειτουργίες API που επιτρέπουν τη χρήση του mutex περιγράφονται παρακάτω:

*bool aeolus_mutex_lock(void *mutex)*
Block until the current mutex can be owned by the requesting component.

*bool aeolus_mutex_unlock(void *mutex)*
Release the current mutex, if it is currently owned. Any components waiting on ‘aeolus_mutex_lock’ can then recontest for the mutex. If multiple components are blocked then a random one is awarded the lock.

*bool aeolus_mutex_trylock(void *mutex)*
Attempt to lock the mutex, but do not block if the attempt was unsuccessful. Returns true if the mutex was locked and false if not.

4.2 Ένα παράλληλο πρότυπο προγραμματισμού: Εφαρμογή

Το σημείο διαίρεσης της εφαρμογής σε ξεχωριστά στοιχεία αφορά την παράλληλη εκτέλεση τους, καταλαμβάνοντας διαφορετικές διαδικασίες σε διαφορετικούς κόμβους επεξεργασίας. Ωστόσο, η πλήρης ενσωμάτωση των συστατικών σε μια ενιαία εφαρμογή απαιτεί την υλοποίηση της μεταξύ τους επικοινωνίας. Έτσι, ένα σύνολο πρωτοκόλλων επικοινωνίας έχει σχεδιαστεί για να παρέχει στον χρήστη επαρκή μεταφορά δεδομένων, καθώς και λειτουργίες συγχρονισμού που θα επιτρέπουν τη διαφάνεια (στον χρήστη) να επιτρέπουν τις απαραίτητες ενέργειες επικοινωνίας και συντονισμού μεταξύ των διαφόρων συνιστωσών. Στις επόμενες ενότητες θα συζητήσουμε το σχεδιασμό και την εφαρμογή του καθενός από αυτά τα πρωτόκολλα.

4.2.1 Υποστήριξη μεταφοράς δεδομένων

Η διεπαφή προγραμματισμού που θα παρουσιαστεί επιτρέπει την επικοινωνία internode μεταξύ των εξαρτημάτων για πολύ πολύπλοκες υλοποιήσεις χωρίς την παρέμβαση του χρήστη. Ωστόσο, η μορφή των προς μεταφορά δεδομένων απαιτεί την υιοθέτηση ενός συγκεκριμένου μοντέλου που ακολουθεί ένα σύνολο κανόνων (κατευθυντήριων γραμμών). Για να διατηρηθεί το μοντέλο αυτό όσο το δυνατόν πιο χαλαρό, έχουν καταβληθεί όλες οι προσπάθειες για την επέκταση της υποστήριξης δομών δεδομένων που μπορούν να μετακινηθούν μεταξύ των διαφόρων συνιστωσών.

Αυτόματη υποστήριξη

Οι πρωταρχικοί τύποι δεδομένων υποστηρίζονται αυτόματα από τη διεπαφή προγραμματισμού. Αυτό περιλαμβάνει τους τύπους: char, short, int, long, float, διπλό και όλα τα προσόντα των παραπάνω (υπογεγραμμένα, μη υπογεγραμμένα, μακρά και μακρά και αυτά που δηλώνονται const).

Εκτεταμένη υποστήριξη

- Δομές (structs) και ενώσεις των παραπάνω τύπων
- Πίνακες των παραπάνω τύπων (συμπεριλαμβανομένων των υποστιριζόμενων δομών και ενώσεων) που δηλώνονται με στατικά οριζόμενο μέγεθος, που καθορίζεται κατά τη μεταγλώτισση. Αυτό συμπεριλαμβάνει πολυδιάστατους πίνακες, όπου όλες οι διαστάσεις έχουν στατικά οριζόμενο μέγεθος, που καθορίζεται κατά τη μεταγλώτισση.
- Οι πίνακες μεταβλητού μήκους των παραπάνω τύπων, αφού το μέγεθος τους μπορεί να μεταβληθεί από το συντελεστή sizeof() σε χρόνο μεταγλώτισσης.
- Στιγμιότυπα κλάσεων των παραπάνω
- Αναφορές στα παραπάνω

Ακριβώς, ένας πίνακας με στατικό μέγεθος μεταγλώττισης είναι πίνακες όπου το sizeof() μπορεί να επιλυθεί σε μια τιμή στο χρόνο μεταγλώττισης χωρίς να χρειάζεται να δημιουργήσει κώδικα χρόνου εκτέλεσης. Για τη χρήση πρωτογενών, οι χρήστες ενθαρρύνονται να χρησιμοποιούν τύπους που περιλαμβάνονται στη βιβλιοθήκη stdint.h, λόγω του σταθερού τους μεγέθους μεταξύ διαφορετικών αρχιτεκτονικών. Σε μια διαφορετική περίπτωση, ο Διαχειριστής Εκτέλεσης θα αναλάβει την επίλυση των ασυνεπειών τύπου πριν από την ανάπτυξη.

Οι τύποι δείκτη δεν υποστηρίζονται αυτόματα επειδή δεν έχουν νόημα όταν μεταφέρονται σε άλλο χώρο μνήμης. Αυτός ο περιορισμός ισχύει και για τις δομές με πεδία δείκτη. Το ίδιο ισχύει και για

τους φορείς C ++ και για άλλες δομές των οποίων το μέγεθος είναι μεταβλητό και δεν μπορεί να γίνει γνωστό κατά τη μεταγλώττιση.

Αντιμετώπιση ανεπιθύμητων καταστάσεων

Οι περιπτώσεις όπου τα δεδομένα που πρέπει να μεταφερθούν ακολουθούν μια πιο σύνθετη δομή από εκείνα που αναφέρθηκαν παραπάνω, μπορούν επίσης εύκολα να χειριστούν ο χρήστης συμπεριλαμβάνοντας τη σειριακή ανάλυση των δεδομένων σε μια στατική δομή πριν από τη μεταφορά όπως φαίνεται στο παρακάτω παράδειγμα .

```
struct X *data_out; // Unsupported data type

int component_A() {
    initialize(data_out);

    #pragma queue out object_out
    char object_out[100]; // Supported data type

    serialize_X(data_out,object_out);
    queue_put(object_out);
    return 0;
}

struct X *data_in; // Unsupported data type

int component_B() {
    #pragma queue out object_in
    char object_in[100]; // Supported data type

    queue_get(object_in);
    deserialize_X(object_in,data_in);
    return 0;
}
```

Γενικά, η διασύνδεση προγραμματισμού παρέχει στον χρήστη ευρεία υποστήριξη για κάθε είδους μεταφορές. Πρέπει να δοθεί προσοχή στους περιορισμούς (που περιγράφονται παραπάνω) που προέρχονται από τη μεταγωγή χώρου διευθύνσεων κατά τη μετακίνηση αντικειμένων μεταξύ διαφορετικών τμημάτων της αρχιτεκτονικής υλικού.

4.2.2 Λειτουργίες πρωτοκόλλου επικοινωνίας

Ανάλογα με τον τύπο φυσικής μνήμης που απαιτείται για την επικοινωνία, χρησιμοποιούνται διάφορες βιβλιοθήκες όπως περιγράφεται παρακάτω.

Στην περίπτωση που περισσότερα από ένα εξαρτήματα εκτελούνται σε ένα σύστημα κοινόχρηστης μνήμης, τα συγκεκριμένα στοιχεία θα εκτελούνται ως διαφορετικά νήματα που μοιράζονται τον ίδιο χώρο μνήμης. Τα αντικείμενα επικοινωνίας που χειρίζονται μόνο από στοιχεία στον ίδιο κόμβο επεξεργασίας υλοποιούνται με τη χρήση των pthreads μαζί με άλλες λειτουργίες POSIX (όπως mutexes και μεταβλητές συνθηκών) για να διατηρηθεί το κόστος επικοινωνίας χαμηλό καθώς και το αποτύπωμα μνήμης.

Για τα στοιχεία που αλληλεπιδρούν με το αντικείμενο όταν εκτελούνται σε διαφορετικούς κόμβους, δηλαδή ένα σύστημα κατανεμημένης μνήμης, απαιτείται μια τεχνική μετάδοσης μηνυμάτων. Για το λόγο αυτό επιλέχθηκε η βιβλιοθήκη OpenMPI λόγω της αποδοτικότητας και της απλότητας της σε διαφορετικά είδη εφαρμογών πραγματικού κόσμου. Συγκεκριμένα, η εκτέλεση διαφορετικών στοιχείων θα υλοποιηθεί ως διαφορετικές διεργασίες MPI και η επικοινωνία μεταξύ τους θα εκμεταλλευτεί πολλές από τις ήδη υπάρχουσες λειτουργίες διεπαφής του OpenMPI που είναι διαθέσιμες.

Στις επόμενες παραγράφους, επισυνάπτεται μια περιγραφή της εφαρμογής των πρωτοκόλλων μαζί με ορισμένες από τις βασικές λειτουργίες των πρωτοκόλλων, προκειμένου να παρουσιαστεί μια καλή

εικόνα της λειτουργικότητας της Διεπαφής. Περισσότερες λεπτομέρειες σχετικά με την υλοποίηση θα συζητηθούν στην επόμενη ενότητα όπου περιγράφεται ο Deployment Manager, λόγω της μεγάλης εξάρτησης της διασύνδεσης προγραμματισμού από αυτήν.

Κοινό πρωτόκολλο - Εφαρμογή κοινής μνήμης

Το κοινόχρηστο πρωτόκολλο υλοποιείται διατηρώντας μια περιοχή στη διαθέσιμη μνήμη ως κοινό τμήμα μεταξύ των διαφόρων συνιστωσών.

```
if(dir == 0) {
    pthread_mutex_lock(&obj->lock);
    memcpy(local_data,shared_data,dims[0]*data_size);
    pthread_mutex_unlock(&obj->lock);
}
else if(dir == 1) {
    pthread_mutex_lock(&obj->lock);
    memcpy(shared_data,local_data,dims[0]*data_size);
    pthread_mutex_unlock(&obj->lock);
}
```

Πρωτόκολλο κοινής μνήμης - Υλοποίηση κατανεμημένης μνήμης

Όταν υπάρχουν κοινά δεδομένα μεταξύ διαφορετικών κόμβων ακολουθείται μια άλλη στρατηγική. Δημιουργείται ένας ουσιαστικά κοινόχρηστος χώρος διευθύνσεων μεταξύ των διαδικασιών στις οποίες εκτελούνται τα στοιχεία. Η μέθοδος συγχρονισμού χρησιμοποιείται για τη λήψη / εισαγωγή οποιωνδήποτε δεδομένων από / προς την κοινόχρηστη μνήμη.

Στην εφαρμογή χαμηλού επιπέδου, τα δεδομένα αποθηκεύονται σε διαφορετικές μνήμες σε όλη την πλατφόρμα υλικού και οι λειτουργίες RMA που παρέχονται από το πρωτόκολλο MPI χρησιμοποιούνται για τη δημιουργία της ουσιαστικά κοινής περιοχής στον χώρο διευθύνσεων των διαδικασιών. Ειδικότερα, οι λειτουργίες MPI_Get() και MPI_Put() χρησιμοποιούνται για την υλοποίηση των κινήσεων δεδομένων που απαιτούνται για να είναι μονόπλευρη. Αυτό σημαίνει ότι ο επεξεργαστής του οποίου η μνήμη είναι πραγματικά ενημερωμένη (στόχος) δεν χρειάζεται να είναι ενεργή και μόνο ο επεξεργαστής που ενημερώνει τη μνήμη (προέλευση) πρέπει να ενεργήσει.

Παρακάτω, εμφανίζεται η δημιουργία ενός τέτοιου περιβάλλοντος, μαζί με την εφαρμογή της λειτουργίας aeolus_synchronize() API.

Η δημιουργία της εμφανιζόμενης συνάρτησης initialize() είναι μέρος της λειτουργικότητας του Deployment Manager και θα συζητηθεί πλήρως στην ενότητα 5. Εδώ παρουσιάζεται η κατανομή του απαιτούμενου παραθύρου κοινής μνήμης καθώς και η απαραίτητη λειτουργία για την ενημέρωση της τοπικής μνήμης με την κοινή έκδοση των δεδομένων (ή αντίστροφα).

```

void initialize() {
    MPI_Init(argc, argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    buf = (int *)malloc(10*sizeof(int));
    MPI_Win_create(buf, 10*sizeof(int), sizeof(int), MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
}
```

```
void aeolus_synchronize(void *local, void *shared, int dir, int caller, int owner
    , int disp, int size, MPI_Datatype type, MPI_Win win) {
    if(dir == 0) {
        if(caller != owner) {
            MPI_Win_lock(MPI_LOCK_SHARED,owner,0,win);
            MPI_Get(shared,size,type,owner,0,size,type,win);
            MPI_Win_unlock(owner,win);
        }
```

```
        memcpy(local,shared,size*disp);
    }
    else if(dir == 1) {
        memcpy(shared,local,size*disp);
        if(caller != owner) {
            MPI_Win_lock(MPI_LOCK_EXCLUSIVE,owner,0,win);
            MPI_Put(shared,size,type,owner,0,size,type,win);
            MPI_Win_unlock(owner,win);
        }
    }
    else {
        perror("Error, not a valid direction");
    }
}
```

Πρωτόκολλο ουράς - Εφαρμογή κοινής μνήμης

Το πρωτόκολλο ουράς χρησιμοποιεί επίσης μια κοινόχρηστη περιοχή μνήμης για να διατηρήσει τα δεδομένα της ουράς. Επιπλέον, χρησιμοποιείται μια κοινή μεταβλητή δείκτη για την πρόσβαση στην ουρά και την επεξεργασία των δεδομένων της.

```
void aeolus_queue_get(aeolus_queue *queue, void* it, int data_size) {
  while(queue->count == 0);
  pthread_mutex_lock(&queue->lock);
  aeolusQnode *tmp = queue->front;
  queue->front = queue->front->next;
  if(queue->front == NULL)
    queue->rear = NULL;
  queue->count--;
  memcpy(it,tmp->item,data_size);
  pthread_mutex_unlock(&queue->lock);
  free(tmp);
}
```

```
bool aeolus_queue_put(aeolus_queue *queue, void* it) {
  aeolusQnode *new_node = generate_aeolus_Qnode(it);
  pthread_mutex_lock(&queue->lock);
  if(queue->rear == NULL) {
    queue->front = queue->rear = new_node;
  }
  else {
    queue->rear->next = new_node;
    queue->rear = new_node;
  }
  queue->count++;
  pthread_mutex_unlock(&queue->lock);
  return true;
}
```

Πρωτόκολλο ουράς - Εφαρμογή κατανεμημένης μνήμης

Το πρωτόκολλο ουράς, όπως και στο κοινόχρηστο πρωτόκολλο, χρησιμοποιεί λειτουργίες RMA για την αποθήκευση και την ανταλλαγή δεδομένων. Συγκεκριμένα, κάθε αντικείμενο επικοινωνίας αποθηκεύεται στο χώρο διεύθυνσης μιας συγκεκριμένης διεργασίας σύμφωνα με το Σχέδιο Ανάπτυξης και χρησιμοποιεί τις λειτουργίες MPI _Get() και MPI _Put() για την πρόσβαση και την αποστολή δεδομένων στην πλατφόρμα υλικού.

```
// queue_get function

prev_front = queue->front;

```

```
if(queue->front == queue->rear)
        queue->front = queue->rear = -1;
else
        queue->front = (queue->front + queue->disp) % info_offset;
queue->count--;
set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->count);

if(caller != queue->owner) {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
        MPI_Get(queue->qdata+prev_front,queue->disp,MPI_UNSIGNED_CHAR,queue->
    owner,prev_front,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
        MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,
    queue->owner,info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,queue->win);
        MPI_Win_unlock(queue->owner,queue->win);
}
memcpy(item,queue->qdata+prev_front,queue->disp);
```

```
// queue_put function

if(queue->rear == -1) queue->front = queue->rear = 0;
else queue->rear = (queue->rear + queue->disp) % info_offset; queue->count++;
set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->count);
memcpy(queue->qdata+queue->rear,item,queue->disp);

if(caller != queue->owner) {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
        MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,
    queue->owner,info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,queue->win);
        MPI_Put(queue->qdata+queue->rear,queue->disp,MPI_UNSIGNED_CHAR,queue->
    owner,queue->rear,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
        MPI_Win_unlock(queue->owner,queue->win);
}
```

Πρωτόκολλο Σημάτων - Υλοποίηση Κοινής Μνήμης

Το Πρωτόκολλο Σημάτων (Signal Protocol) χρησιμοποιεί τις δομές μεταβλητών συνθήκης που ορίζονται στην POSIX για την ανταλλαγή σημάτων μεταξύ των components.

```
void aeolus_notify(aeolus_signal *curSignal, bool *rd, int aeolus_src) {
    pthread_mutex_lock(&curSignal->lock);

    curSignal->ready = *rd;
    if(curSignal->ready == true) {
       if(pthread_cond_signal(&curSignal->cond) != 0) {
              fprintf(stderr,"Failed to send signal\n");
              exit(0);
       }
    }
    else {
    perror("Signal called with value 0");
    exit(1);
    }
    pthread_mutex_unlock(&curSignal->lock);
}
```

```
void aeolus_wait(aeolus_signal *curSignal, bool *rd, int aeolus_cmpid) {
    pthread_mutex_lock(&curSignal->lock);
    while(curSignal->ready == 0) {
        if(pthread_cond_wait(&curSignal->cond,&curSignal->lock) != 0) {
            fprintf(stderr, "failed to wait the condition variable\n");
            exit(0);
        }
        else {
```

```
        curSignal->ready = 1;
        *rd = true;
      }
    }
    pthread_mutex_unlock(&curSignal->lock);
}
```

Πρωτόκολλο Σημάτων - Υλοποίηση Κατανεμημένης Μνήμης

Όταν εκτελούνται σε διαφορετικούς κόμβους, οι λειτουργίες MPI_Send(), MPI_Recv(), MPI_Bcast() χρησιμοποιούνται για τη σηματοδότηση και MPI_Barrier() για τη λειτουργία aeolus_barrier().

```
void aeolus_notify(int *rd, int dst, MPI_Comm comm) {
  MPI_Send(rd, 1, MPI_INT, dst, 0, comm);
}
```

```
int aeolus_wait(int *rd, int src, MPI_Comm comm) {
    MPI_Status status;
    MPI_Recv(rd, 1, MPI_INT, src, MPI_ANY_TAG, comm, &status);
    return status.MPI_ERROR;
}
```

```
void aeolus_notifyall(int *rd, int src, MPI_Comm comm) {
  MPI_Bcast(rd,1,MPI_INT,src,comm);
}
```

```
void aeolus_barrier(MPI_Comm comm) {
  MPI_Barrier(comm);
}
```

Πρωτόκολλο Αμοιβαίου Αποκλεισμού - Υλοποίηση Κοινής Μνήμης

Το Πρωτόκολλο Αμοιβαίου Αποκλεισμού χρησιμοποιεί την αντίστοιχη λειτουργικότητα που είναι διαθέσιμη στις βιβλιοθήκες της POSIX.

```
bool aeolus_mutex_lock(aeolus_mutex *mutex) {
    if(!pthread_mutex_lock(&mutex->lock))
        return true;
    else
        return false;
}
```

```
bool aeolus_mutex_unlock(aeolus_mutex *mutex) {
    if(!pthread_mutex_unlock(&mutex->lock))
        return true;
    else
        return false;
}
```

```
bool aeolus_mutex_trylock(aeolus_mutex *mutex) {
    if(!pthread_mutex_unlock(&mutex->lock))
        return true;
    else
        return false;
}
```

Πρωτόκολλο Αμοιβαίου Αποκλεισμού - Υλοποίηση Κατανεμημένης Μνήμης

Όπως και με το πρωτόκολλο ουράς, το πρωτόκολλο αμοιβαίου αποκλεισμού διατηρεί μια λίστα προτεραιότητας σε έναν συγκεκριμένο κόμβο, ο οποίος δηλώνεται από το σχέδιο ανάπτυξης. Έτσι, κάθε διαδικασία χρησιμοποιεί λειτουργίες RMA για πρόσβαση στον ιδιοκτήτη του mutex. Εάν η κλειδαριά αποκτάται από άλλη διαδικασία, τότε η διαδικασία κλήσης αποκλείει τη χρήση του αποκλεισμού MPI_Recv() μέχρι να ολοκληρωθεί η κλειδαριά. Από την άλλη άκρη, η διαδικασία που συγκρατεί την κλειδαριά χρησιμοποιεί MPI_Send() για να δώσει την κλειδαριά σε μία από τις διαδικασίες που την περιμένουν.

```
// lock function

// Try to acquire lock in one access epoch
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
    MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */, 1,
    MPI_CHAR, mutex->win);
    MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->numprocs,
     MPI_CHAR, mutex->win);
    MPI_Win_unlock(mutex->home, mutex->win);

    assert(waitlist[mutex->ID] == 1);

    // Count the 1's
    for (i = 0; i < mutex->numprocs; i++) {
        if (waitlist[i] == 1 && i != mutex->ID) {
            // We have to wait for the lock
            // Dummy receive, no payload
            MPI_Recv(&lock, 0, MPI_CHAR, MPI_ANY_SOURCE, mutex->tag, mutex->comm,
     MPI_STATUS_IGNORE);
            break;
        }
    }
```

```
// unlock function

MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */, 1,
    MPI_CHAR, mutex->win);
MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->numprocs,
    MPI_CHAR, mutex->win);
MPI_Win_unlock(mutex->home, mutex->win);
assert(waitlist[mutex->ID] == 0);
// If there are other processes waiting for the lock, transfer ownership
next = (mutex->ID + 1 + mutex->numprocs) % mutex->numprocs;
for (i = 0; i < mutex->numprocs; i++, next = (next + 1) % mutex->numprocs) {
    if (waitlist[next] == 1) {
    // Dummy send, no payload
        MPI_Send(&lock, 0, MPI_CHAR, next, mutex->tag, mutex->comm);
        break;
    }
}
```

4.2.3 Αρχικοποίηση

Ο χρήστης μπορεί να αποκτήσει έναν δείκτη στη δομή του αντικειμένου επικοινωνίας μέσω μιας μεθόδου αρχικοποίησης, ώστε να μπορεί να καλέσει τις υπόλοιπες μεθόδους API χρησιμοποιώντας αυτόν τον δείκτη. Οι λειτουργίες αρχικοποίησης εμφανίζονται παρακάτω:

*aeolus_shared *aeolus_shared_init(char *comm_object_port)*

*aeolus_queue *aeolus_queue_init(char *comm_object_port)*

*aeolus_signal *aeolus_signal_init(char *comm_object_port)*

*aeolus_mutex *aeolus_mutex_init(char *comm_object_port)*

όπου comm_object_port είναι το όνομα της θύρας του αντικειμένου όπως ορίζεται στο δίκτυο συνιστωσών. Το όνομα της θύρας σχετίζεται αυστηρά με το αρχείο προέλευσης και όχι με την οντότητα συνιστωσών, επομένως τα στοιχεία που συνδέονται με το ίδιο αρχείο προέλευσης θα πρέπει να χρησιμοποιούν το ίδιο όνομα θύρας στο δίκτυο συνιστωσών για τα αντίστοιχα αντικείμενα.

Μια σημαντική σημείωση εδώ είναι ότι ο χρήστης πρέπει να δηλώσει τις μεταβλητές αντικειμένου επικοινωνίας προκειμένου να χρησιμοποιήσει τις αντίστοιχες λειτουργίες πρωτοκόλλου. Οποιεσδήποτε αρχικοποιήσεις αντικειμένων επικοινωνίας θα πρέπει να γίνονται πάντα τοπικά μέσα στη λειτουργία του στοιχείου. Από την άλλη πλευρά, ο δείκτης του αντικειμένου μπορεί να περάσει σε οποιαδήποτε λειτουργία χρειάζεται ο χρήστης για να καλέσει τις λειτουργίες πρωτοκόλλου (aeolus_queue_get, aeolus_synchronize etc.).

4.2.4 Λειτουργίες αρχείων

Το μοντέλο προγραμματισμού AEOLUS έχει επεκταθεί με λειτουργίες αρχείων που πρόκειται να επιτρέψουν τη λειτουργία αρχείων σε μια μη ομοιόμορφη αρχιτεκτονική με διαφορετική μνήμη όπου τα αρχεία μπορεί να βρίσκονται οπουδήποτε. Οι πρόσθετες λειτουργίες είναι καθρέφτες των λειτουργιών του αρχείου POSIX, προκειμένου να μεγιστοποιηθεί η συμβατότητα. Τα αντικείμενα ροής που επιστρέφονται από τις λειτουργίες είναι συμβατά με τις άλλες λειτουργίες εισόδου / εξόδου από το πρότυπο C library, fprintf, fscanf, snprintf, sprintf, sscanf.

*FILE * aeolus_fopen (const char * filename, const char * mode)*

*int aeolus_fclose (FILE * stream)*

*int aeolus_fflush (FILE * stream)*

*size_t aeolus_fwrite (const void * ptr, size_t size, size_t count, FILE * stream)*

*size_t aeolus_fread (void * ptr, size_t size, size_t count, FILE * stream)*

*int aeolus_fgetpos (FILE * stream, fpos_t * pos)*

*int aeolus_fseek (FILE * stream, long int offset, int origin)*

*int aeolus_fileno(FILE *stream)*
Returns the integer file descriptor associated with the stream pointed to by stream.

*FILE *aeolus_fdopen(int fd, const char *mode)*
Συνδέει μια ροή με τον υπάρχοντα περιγραφέα αρχείου, fd. Η λειτουργία της ροής (μία από τις τιμές: “r”, “r+”, “w”, “w+”, “a”, “a+”) πρέπει να είναι συμβατή με τη λειτουργία του περιγραφέα αρχείων. Η ένδειξη θέσης αρχείου της νέας ροής έχει οριστεί σε αυτήν που ανήκει στο fd και οι ενδείξεις σφαλμάτων και τέλους του αρχείου διαγράφονται. Οι τρόποι ”w” ή ”w +” δεν προκαλούν περικοπή του αρχείου. Ο περιγραφέας του αρχείου δεν είναι διπλωμένος και θα κλείσει όταν κλείσει η δέσμη ενεργειών που δημιουργήθηκε από το aeolus_fdopen(). Το αποτέλεσμα της εφαρμογής aeolus_fdopen () σε ένα αντικείμενο κοινόχρηστης μνήμης είναι απροσδιόριστο.

int aeolus_get_fd_flags(int fd, int value)
Ανακτά την τρέχουσα θέση στη ροή.

*long int aeolus_ftell(FILE *stream)*
Επιστρέφει την τρέχουσα τιμή του δείκτη θέσης της ροής. Για δυαδικές ροές, αυτός είναι ο αριθμός των bits από την αρχή του αρχείου. Για τις ροές κειμένου, η αριθμητική τιμή μπορεί να χρησιμοποιηθεί για την επαναφορά της θέσης στην ίδια θέση αργότερα χρησιμοποιώντας το aeolus_fseek().

4.2.5 Monitoring library

Έχει επίσης δημιουργηθεί μια διεπαφή παρακολούθησης για τον χρήστη να παρακολουθεί συγκεκριμένα τμήματα της εφαρμογής του. Ο τύπος aeolus_monitor ορίζεται για να παρέχει πρόσβαση στις λειτουργίες παρακολούθησης που παρέχει το AEOLUS.

*aeolus_monitor *aeolus_monitor_init()*
Επιστρέφει έναν δείκτη στην οθόνη που χρησιμοποιείται για την αποθήκευση μετρήσεων σε τοπικά αρχεία.

*void aeolus_mf_start(aeolus_monitor *monitor)*
Καταγράφει την έναρξη του component. Αν δεν χρησιμοποιηθεί, θα ξεκινήσει αυτόματα η έναρξη της εκτέλεσης, η οποία μπορεί να μην προσδιορίσει επακριβώς την έναρξη του υπολογιστικά έντονου τμήματος του component.

*void aeolus_mf_user_metric(aeolus_monitor *monitor, char *metric_name, int value)*
Καταγράφει μια μετρική που ο χρήστης μπορεί να θεωρήσει χρήσιμη.

*void aeolus_mf_end(aeolus_monitor *monitor)*
Καταγράφει το τέλος εκτέλεσης του component. Εάν δεν χρησιμοποιηθεί, θα καταχωρηθεί αυτόματα το τέλος της εκτέλεσης, το οποίο μπορεί να μην προσδιορίσει επακριβώς το τέλος του υπολογιστικού εντατικού τμήματος του component.

Σημείωση: Όπως συμβαίνει με τις αρχικοποιήσεις των communication objects, οι αρχικοποιήσεις παρακολούθησης θα πρέπει να πραγματοποιούνται πάντα τοπικά μέσα στη βασική συνάρτηση του component, ενώ ο δείκτης του monitor μπορεί να περάσει σε οποιαδήποτε λειτουργία χρειάζεται ο χρήστης για να καλέσει τις λειτουργίες παρακολούθησης.

4.2.6 Επισημειώσεις παραλληλοποίησης

Το μοντέλο προγραμματισμού έχει επεκταθεί ώστε να περιλαμβάνει βοηθητικές επισημειώσεις για τη στατική ανάλυση του κώδικα για την ενίσχυση των αποτελεσμάτων παραλληλισμού των εργαλείων παραλληλοποίησης (όπως παρουσιάζεται στην ενότητα 3.3.3). Αυτά παρέχουν πρόσθετες πληροφορίες που είναι πολύ περίπλοκες για να εξαχθούν αυτόματα και επομένως μπορούν να χρησιμοποιηθούν από τους προγραμματιστές για να βελτιώσουν την ανάπτυξη του κώδικα τους, χωρίς να επιβάλλουν κατευθυντήριες γραμμές σε όλους τους προγραμματιστές.

#pragma aeolus function/loop static-vectors
Εγγυάται ότι η ακόλουθη λειτουργία / βρόχος περιλαμβάνει μόνο στατικούς vectors, πράγμα που σημαίνει ότι δεν υπάρχει δυναμική κατανομή μνήμης κατά την εκτέλεση.

#pragma aeolus loop no-aliasing
Εγγυάται ότι η λειτουργία / βρόχος δεν περιλαμβάνει aliasing δείκτη, που σημαίνει πρόσβαση στην ίδια διεύθυνση μνήμης από διαφορετικούς δείκτες.

#pragma aeolus function no-side-effects
Εγγυάται ότι η ακόλουθη λειτουργία δεν περιλαμβάνει παρενέργειες, που σημαίνει ότι δεν υπάρχει γραφή στις διευθύνσεις ή τις ροές μνήμης, έτσι η ροή του προγράμματος δεν επηρεάζεται περισσότερο από την τιμή επιστροφής της λειτουργίας.

Η πλατφόρμα θα επαληθεύσει αυτά τα pragmas (όταν είναι δυνατόν) για να βοηθήσει τον προγραμματιστή και θα εμφανίσει σφάλμα εάν παραβιαστεί, αλλά γενικά θεωρούνται ως εγγυήσεις από τον προγραμματιστή στην πλατφόρμα.

Κεφάλαιο 5

Μια υποδομή για Ανάπτυξη, Βελτιστοποίηση και Εκτέλεση παράλληλων εφαρμογών (AEOLUS)

5.1 Ιδέα

Τα πλεονεκτήματα της ετερογενούς πληροφορικής, όπως ήδη αναφέρθηκε, βασίζονται στον συνδυασμό διαφορετικών τεχνολογιών για την εκμετάλλευση των διαφόρων περιοχών παράλληλου κώδικα που συμπεριφέρονται βέλτιστα σε διαφορετικούς τύπους επεξεργαστών. Έτσι, αυτή η ποικιλία των εργαλείων που απαιτούνται είναι η κύρια δυσκολία για τον προγραμματιστή που είναι υπεύθυνος να έχει επαρκή γνώση αυτών των εργαλείων ή να αποκτήσει αυτή τη γνώση για να επιτύχει το στόχο τους.

Η λύση που προτείνεται εδώ ασχολείται με την αρχιτεκτονική και λειτουργικότητα μιας Υποδομής για την Ανάπτυξη και Βελτιστοποίηση Εφαρμογών σε Παράλληλες Αρχιτεκτονικές (AEOLUS), το οποίο παρέχει πολυδιάστατες δυνατότητες παραλληλισμού που θα ελαχιστοποιούν το χάσμα γνώσης μεταξύ του προγραμματιστή και των διαφόρων συστημάτων που μπορούν να χρησιμοποιηθούν. Με στόχο την ανάπτυξη εφαρμογών σε ένα ετερογενές περιβάλλον, τα εργαλεία θα εκμεταλλευτούν τεχνολογίες αιχμής, βοηθώντας ταυτόχρονα την ανάπτυξη σε πολλαπλά συστατικά υλικού. Το κύριο συστατικό της σειράς εργαλείων θα παρέχει τον πυρήνα του συστήματος ανάπτυξης και ένα σύνολο πρόσθετων ενοτήτων θα ολοκληρώσει την αρχιτεκτονική του πλαισίου με περισσότερες λειτουργίες με στόχο την περαιτέρω βελτιστοποίηση της εφαρμογής ή της χρήσης του συστήματος.

5.2 Αρχιτεκτονική

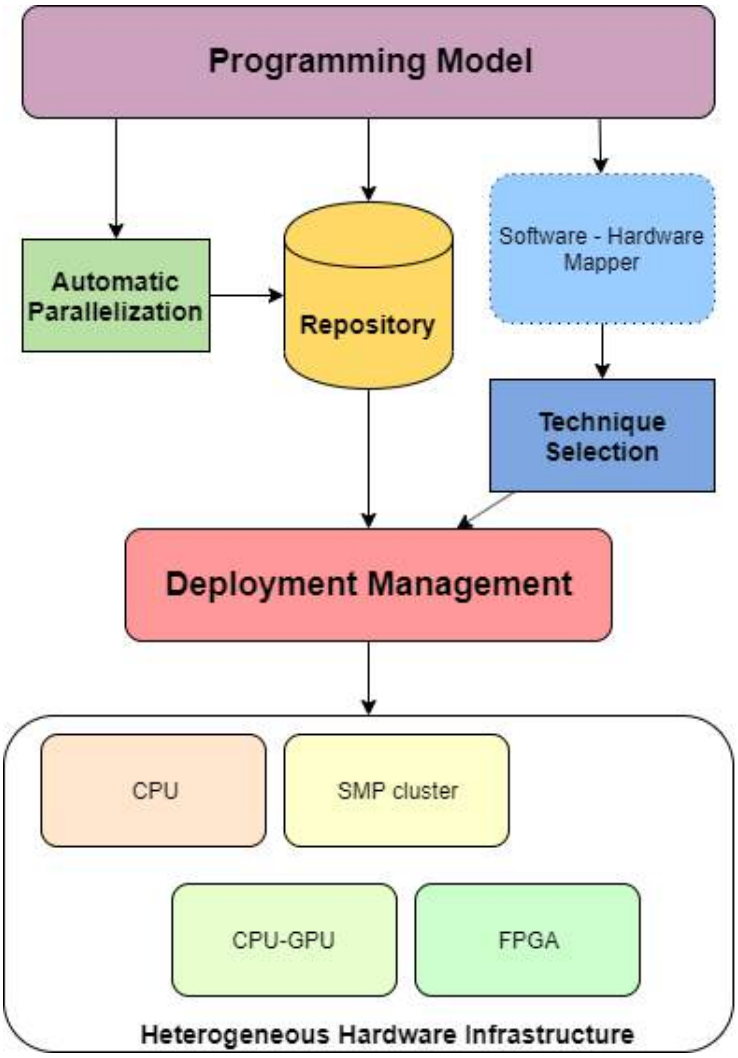
Η βάση του συστήματος είναι ένα στοιχείο ανάπτυξης που ονομάζεται Deployment Manager, το οποίο είναι υπεύθυνο για την ανάπτυξη της εφαρμογής στην υποδομή υλικού και την υλοποίηση της επικοινωνίας μεταξύ των διαφόρων εξαρτημάτων υλικού. Η ενσωμάτωση της εφαρμογής ολοκληρώνεται με τη χρήση συγκεκριμένων μεθοδολογιών που υποστηρίζονται από εξωτερικές βιβλιοθήκες που επιλέγονται σύμφωνα με τις απαιτήσεις της ανάπτυξης από ένα άλλο στοιχείο, το Technique Selection. Έχει αναπτυχθεί μια πρόσθετη ενότητα για να υποστηρίξει την πολυδιάστατη βελτιστοποίηση κώδικα και παραλληλισμό, που αναφέρεται ως Auto-Parallelization Tool.

Μια επισκόπηση της αρχιτεκτονικής του συστήματος εμφανίζεται στο σχήμα 5.1, ενώ εξηγείται μια σύντομη περιγραφή του toolflow.

Όπως απεικονίζεται στο σχήμα 5.1, Η AEOLUS, όπως υλοποιείται στο πλαίσιο αυτής της εργασίας, αποτελείται από τέσσερα βασικά στοιχεία:

1. Programming Model
   Ένα σύνολο κανόνων και κατευθυντήριων γραμμών που καθορίζουν τη διαδικασία ανάπτυξης και υλοποίησης των συστατικών λογισμικού κατά τρόπο που να εγγυάται την επιτυχή ανάπτυξη και παραλληλισμό της εφαρμογής.

2. Automatic Parallelization Tool
   Ένα εργαλείο για την ανάλυση του κώδικα και την παραγωγή παραλληλισμένων εκδόσεων των components.



Σχήμα 5.1: Η αρχιτεκτονική του AEOLUS

3. Technique Selection
   Ενα εργαλείο για την επιλογή των κατάλληλων υλοποιήσεων και των δύο στοιχείων και του κώδικα επικοινωνίας σύμφωνα με το συγκεκριμένο σχέδιο ανάπτυξης.

4. Deployment Manager
   Ένα εργαλείο για την υλοποίηση της εκτέλεσης των στοιχείων της εφαρμογής στην υποδομή υλικού σύμφωνα με το Deployment Plan και τα αποτελέσματα από το Technique Selection. Επιπλέον, έχει αναπτυχθεί μια διεπαφή προγραμματισμού (ένα σύνολο APIs) για την υλοποίηση της επικοινωνίας / συγχρονισμού μεταξύ των στοιχείων της εφαρμογής.

Η ροή του εργαλείου ξεκινάει με τη στατική παραλληλισμό των μεμονωμένων εξαρτημάτων της εφαρμογής με αποτέλεσμα την παραγωγή παράλληλων εκδόσεων των εξαρτημάτων και την αποθήκευση τους σε μια δεξαμενή που μπορεί να προσεγγιστεί από άλλες μονάδες. Σύμφωνα με τη χαρτογράφηση των στοιχείων της εφαρμογής στα διάφορα εξαρτήματα υλικού (σχέδιο ανάπτυξης), η Τεχνική Επιλογή επιλέγει τις κατάλληλες εκδόσεις που θα βελτιστοποιήσουν την απόδοση της εφαρμογής στην υποδομή υλικού. Επιπλέον, επιλέγεται μια στρατηγική ανάπτυξης, σύμφωνα με το σχέδιο ανάπτυξης, το οποίο αντιστοιχεί άμεσα στην επιλογή των API επικοινωνίας που πρόκειται να χρησιμοποιηθούν για την υλοποίηση της αλληλεπίδρασης των συστατικών στοιχείων. Τέλος, τα στοιχεία μαζί με τις εξωτερικές βιβλιοθήκες που απαιτούνται για την επικοινωνία τους μεταβιβάζονται στο Διαχειριστή Εκτέλεσης ως σύνολο αρχείων πηγαίου κώδικα για την πραγματική υλοποίηση των τε-

χνικών ανάπτυξης που επιλέχθηκαν στο προηγούμενο στάδιο. Το αποτέλεσμα του toolflow αποτελείται από ένα σύνολο εκτελέσιμων αρχείων και αρχείων διαμόρφωσης στους αντίστοιχους στόχους που τελικά εκτελούνται και παρακολουθούνται. Η συνολική διαδικασία καθοδηγείται από τη δομή που έχει οριστεί στο μοντέλο προγραμματισμού.

5.3 Μοντέλο προγραμματισμού

Το μοντέλο προγραμματισμού που ορίσαμε στο τμήμα 7 αποτελεί τον πυρήνα του πλαισίου. Οι χρήστες αναπτύσσουν την εφαρμογή τους ακολουθώντας συγκεκριμένες οδηγίες που ορίζονται στο μοντέλο προγραμματισμού, δημιουργώντας μια εφαρμογή που, ανάλογα με τις ανάγκες της, μπορεί να αναπτυχθεί σε μια σειρά διαφορετικών πλατφορμών υλικού. Το μοντέλο προγραμματισμού:

- υποστηρίζει τη στατική παραλληλοποίηση διαφορετικών components ενισχύοντας τις δυνατότητες του εργαλείου με βοηθητικές επισημειώσεις από το χρήστη
- παρέχει διάφορες επιλογές για τη στρατηγική εκτέλεσης (τεχνικές / τεχνολογίες) ακόμη και για ένα ενιαίο Deployment Plan
- δημιουργεί τις συνθήκες υπό τις οποίες τα στοιχεία μπορούν να ενσωματωθούν και να αναπτυχθούν ως μία ενιαία εφαρμογή
- παρέχει μια σειρά πρωτοκόλλων επικοινωνίας που επιτρέπουν επικοινωνία και συγχρονισμό μεταξύ των components
- Παρέχει μια ποικιλία από διαφορετικές λειτουργίες που μπορούν να βελτιώσουν την εμπειρία του χρήστη, όπως η παρακολούθηση της εκτέλεσης των στοιχείων του λογισμικού, καθώς και η πρόσβαση σε αρχεία σε διαφορετικά μηχανήματα

Έχουμε ορίσει το μοντέλο προγραμματισμού που χρησιμοποιείται από το AEOLUS στην ενότητα 5. Σε αυτήν την παράγραφο, περιγράφουμε τον τρόπο με τον οποίο η υποδομή εκμεταλλεύεται αυτό το μοντέλο για να υποστηρίξει παραλληλισμό και κατανομή φορτίου της εφαρμογής στο διαθέσιμο υλικό. Περισσότερες λεπτομέρειες για τις διαφορετικές πτυχές της δίνονται στα αντίστοιχα τμήματα των εργαλείων που τα εκμεταλλεύονται.

5.3.1 Συστατικά

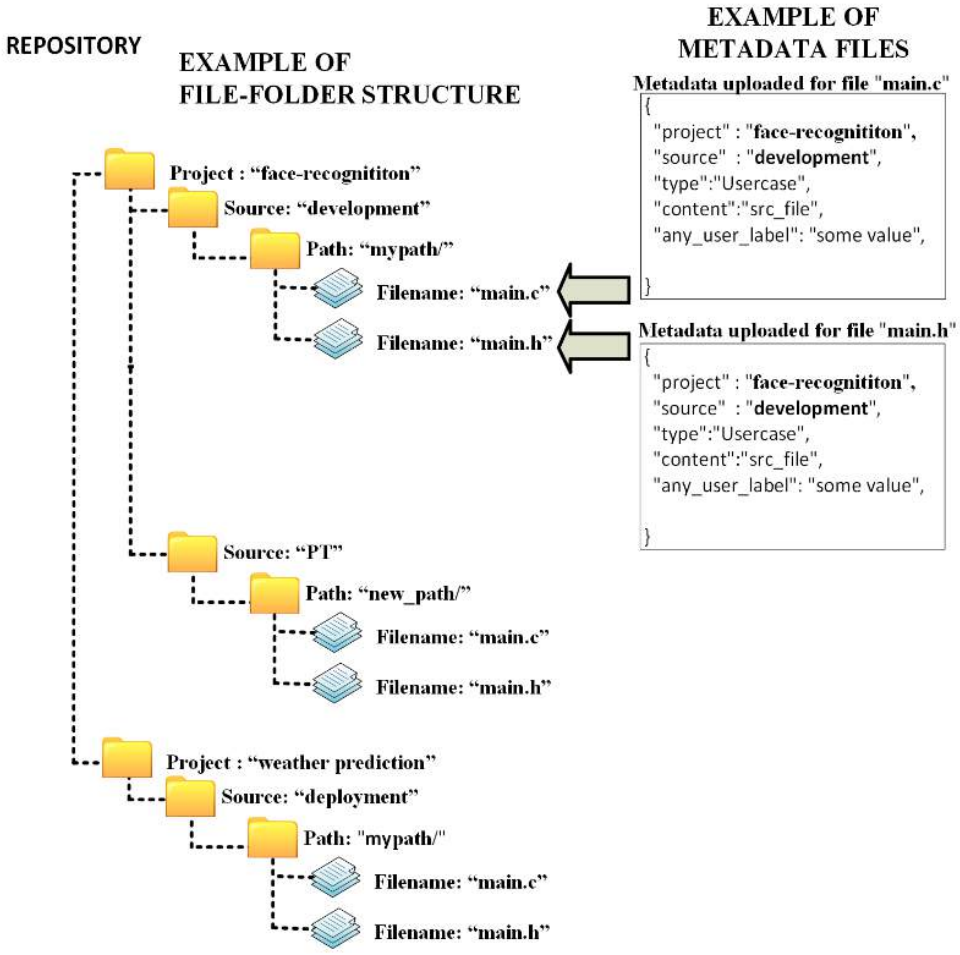
Κάθε στοιχείο αντιστοιχεί σε ένα συγκεκριμένο αρχείο πηγής όπως ορίζεται στο δίκτυο συστατικών στοιχείων, το οποίο πρέπει να έχει ένα σημείο εισόδου για την εκτέλεση του στοιχείου. Τα στοιχεία μπορούν να επαναχρησιμοποιηθούν πολλές φορές, εφόσον αυτό ορίζεται ρητά στο δίκτυο συνιστωσών. Ξεκινούν ως ξεχωριστές και ανεξάρτητες λειτουργίες. Αυτές οι λειτουργίες υπάρχουν στο αποκλειστικό αρχείο προέλευσης του εξαρτήματος με το αντίστοιχο όνομα. Ο προγραμματιστής είναι σε θέση να επιλέξει να μεταβιβάσει τα επιχειρήματα της γραμμής εντολών στη λειτουργία. Τέλος, ο τύπος επιστροφής των λειτουργιών θα πρέπει να είναι κενός*. Επομένως, η υπογραφή της συνάρτησης για ένα στοιχείο με αρχείο προέλευσης comp_example.cpp θα πρέπει να είναι έτσι:

*void *comp_example(int32_t argc, char **argv)*

or

*void *comp_example()*

Οι προγραμματιστές είναι ελεύθεροι να συμπεριλάβουν όλες τις εξωτερικές βιβλιοθήκες που επιθυμούν να χρησιμοποιήσουν, καθώς δεν εντοπίστηκαν ζητήματα συμβατότητας κατά την ανάπτυξη. Από την άλλη πλευρά, προκειμένου τα στοιχεία να χρησιμοποιούν τις βιβλιοθήκες AEOLUS, πρέπει να συμπεριληφθεί το αρχείο κεφαλίδας 'aeolus.h', έτσι ώστε οι απαραίτητες δομές και λειτουργίες να είναι ορατές από την κύρια συνιστώσα του εξαρτήματος.



Σχήμα 5.2: Δομή αποθετηρίου

5.3.2 Αντικείμενα επικοινωνίας

Το AEOLUS ορίζει μια σειρά από βιβλιοθήκες ανάπτυξης που περιγράφονται λεπτομερώς στην ενότητα του Deployment Manager (5.6) και είναι υπεύθυνοι για την εισαγωγή των διαφόρων διαμορφώσεων επικοινωνίας που απαιτούνται για την επιτυχή εκτέλεση της εφαρμογής.

Μία από αυτές τις διαμορφώσεις περιλαμβάνει τον ορισμό των αντικειμένων επικοινωνίας του κοινόχρηστου πρωτοκόλλου. Ο προγραμματιστής είναι σε θέση να μεταφέρει αντικείμενα που ορίζονται από το χρήστη με στατικό μέγεθος. Προκειμένου να ενεργοποιηθούν τέτοιες συναλλαγές, ο χρήστης θα πρέπει να περιλαμβάνει τις βιβλιοθήκες που ορίζουν τέτοιες δομές σε ένα ειδικό αρχείο κεφαλίδας για αυτό το σκοπό. Αυτό το αρχείο ονομάζεται 'aeolus_user_defined_structs.h' και πρέπει να τοποθετηθεί στον κατάλογο src (δείτε την επόμενη ενότητα).

5.3.3 Τοποθέτηση της εφαρμογής του προγραμματιστή στο αποθετήριο του AEOLUS

Το αποθετήριο του AEOLUS χρησιμοποιείται από όλα τα διαφορετικά εργαλεία του πλαισίου. Για το σκοπό αυτό, τα εργαλεία αναλαμβάνουν μια συγκεκριμένη δομή στο χώρο αποθήκευσης που πρέπει να σέβεται ο προγραμματιστής κατά τη φόρτωση οποιωνδήποτε αρχείων προέλευσης, εισόδου ή περιγραφής. Η δομή που ακολουθείται από το χώρο αποθήκευσης περιέχει ένα σύνολο φακέλων επιπέδου-ένα που θεωρούνται οι κατάλογοι έργου. Στο δεύτερο επίπεδο υπάρχει ένα σύνολο καταλόγων που ορίζουν την πηγή (ιδιοκτήτη).

Ο προγραμματιστής πρέπει να τοποθετήσει όλα τα απαραίτητα αρχεία στο χώρο αποθήκευσης

στο *project_name/development* πριν από την έναρξη των εργαλείων. Ειδικότερα, θα πρέπει να ακολουθείται η ακόλουθη δομή:

project_name/

development/

src/

aeolus_user_defined_structs.h

description/

Component-Network.xml

Platform-Description.xml

inputs/

outputs/

5.4 Αυτόματο εργαλείο παραλληλοποίησης

5.4.1 Περιγραφή

Το πρώτο στάδιο της αρχιτεκτονικής του υποδομή αφορά την αυτόματη παραλληλοποίηση των στοιχείων της εφαρμογής. Για το σκοπό αυτό, έχει αναπτυχθεί ένα αυτόματο εργαλείο που αναλύει τον πηγαίο κώδικα συστατικών στοιχείων και το μετασχηματίζει σύμφωνα με τις οδηγίες παραλληλισμού που παρέχει ο χρήστης. Η λειτουργικότητα του εργαλείου επικεντρώνεται στην ανάλυση του κώδικα για τον εντοπισμό παράλληλων περιοχών σε αυτό και την παραγωγή τροποποιημένων εκδόσεων που επιτρέπουν την περαιτέρω εκμετάλλευση των πόρων του συστήματος. Η ανάλυση στοχεύει στην απομόνωση των εξαρτήσεων δεδομένων που ανακαλύφθηκαν κατά την ανάλυση, οι οποίες δεν επηρεάζουν την πιθανή συνάφεια. Οι σχολιασμοί κώδικα μπορούν να παρέχονται από τον προγραμματιστή για να επεκτείνουν την κατανόηση του εργαλείου για τη ροή δεδομένων και να επιλύσουν πολλές από αυτές τις εξαρτήσεις. Λεπτομερής περιγραφή της ροής εργαλείων αναπτύσσεται στις ακόλουθες παραγράφους.

5.4.2 Σχεδιασμός

Ο σκοπός αυτού του εργαλείου στην αρχιτεκτονική AEOLUS είναι η αυτόματη δημιουργία τροποποιημένων εκδόσεων των αρχικών εξαρτημάτων για την εκτέλεσή τους σε διαφορετικά συστήματα για την αξιοποίηση των ειδικών ιδιοτήτων του κώδικα. Οι εκδόσεις που υποστηρίζονται αυτήν τη στιγμή είναι:

OpenMP version: Περιλαμβάνει επισημειώσεις OpenMP που υποδεικνύουν τη χρήση πολλαπλών νημάτων για την εκτέλεση του component σε πολυπύρηνα συστήματα.

CUDA version: Περιλαμβάνει τη διεπαφή (API) της CUDA με την οποία επικοινωνεί με το kernel της GPU (το κομμάτι κώδικα που τρέχει στην συσκευή) επιταχύνοντας την εφαρμογή με χρήση GPU. Η έκδοση αυτή δημιουργείται μόνο όταν διευκρινίζεται ρητά στο Component Network πως είναι δυνατή η δημιουργία της.

Η δημιουργία των ενημερωμένων στοιχείων θα δώσει τη δυνατότητα στο AEOLUS να εκμεταλλευτεί την διαθέσιμη πλατφόρμα υλικού και να επιταχύνει τη λειτουργικότητα της εφαρμογής. Η ροή του εργαλείου περιγράφεται στα παρακάτω βήματα.

Ανάλυση Component Network

Η κύρια δραστηριότητα του εργαλείου είναι η αυτόματη παραλληλοποίηση των στοιχείων και των βιβλιοθηκών που ορίζονται από το χρήστη. Για την υλοποίηση αυτής της αλληλεπίδρασης με το χρήστη, το εργαλείο παίρνει σαν είσοδο το αρχείο XML του Component Network, το οποίο περιέχει όλες τις απαραίτητες πληροφορίες για τα στοιχεία (π.χ. όνομα, αρχείο προέλευσης, εξωτερικές βιβλιοθήκες που χρησιμοποιήθηκαν κλπ.) και αποθηκεύει αυτές τις πληροφορίες χρησιμοποιώντας ένα σύνολο κλάσεων που έχουν σχεδιαστεί για να μοντελοποιούν τα software components και τις μεταφορές δεδομένων μεταξύ τους.

Αυτόματη Παραλληλοποίηση

Μετά την εξαγωγή των απαραίτητων πληροφοριών από το αρχείο XML, το εργαλείο μπορεί να προχωρήσει στην παραλληλισμό του κώδικα. Για το σκοπό αυτό, απαιτείται άλλη αλληλεπίδραση με το χώρο αποθήκευσης για τη λήψη των αρχείων κώδικα της εφαρμογής. Τέλος, χρησιμοποιείται η πραγματική ανάλυση του κώδικα για όλα τα στοιχεία και τις εξωτερικές βιβλιοθήκες (μόνο εκείνες που ορίζονται στο δίκτυο Component) που χρησιμοποιούνται, όπως περιγράφεται στα παρακάτω βήματα.

Προεπεξεργασία με το μοντέλο προγραμματισμού

Χαρακτηριστικά του μοντέλου προγραμματισμού βοηθούν το στάδιο της ανάλυσης κώδικα. Ειδικότερα, η απομόνωση συστατικών στοιχείων του μοντέλου σημαίνει ότι υπάρχουν αποκρίσεις δεδομένων που ανακαλύφθηκαν κατά τη διάρκεια της ανάλυσης, οι οποίες δεν επηρεάζουν τη δυνητική συνάφεια. Για το λόγο αυτό, οι συγκεκριμένοι σχολιασμοί, που περιλαμβάνονται στο μοντέλο προγραμματισμού, χρησιμοποιούνται για την επέκταση της κατανόησης της ροής δεδομένων από το εργαλείο και για την επίλυση πολλών από αυτές τις εξαρτήσεις. Παραδείγματος χάριν, οι στατικοί φορείς vector #pragma aeolus χρησιμοποιούνται για τον χαρακτηρισμό ενός φορέα στατικού μεγέθους, ο οποίος καταργεί με αυτόν τον τρόπο πολλές από τις εξαρτήσεις που προκύπτουν λόγω του μεταβλητού μεγέθους του φορέα. Για να εκμεταλλευτούν αυτές τις ενδείξεις, ένα μεγάλο μέρος της λειτουργικότητας του εργαλείου αφορά την προεπεξεργασία του κώδικα, με στόχο την επίλυση πολλών υφισταμένων εξαρτήσεων του. Το εργαλείο χρησιμοποιεί αυτή την αλληλεπίδραση με τον χρήστη για να επιλύσει μερικές από τις δυσκολίες που δεν έχουν ακόμη επιλύσει οι τελευταίες τεχνικές αυτόματης παραλληλοποίησης.

Παραμετροποίηση OpenMP

Η ανάλυση κώδικα που εκτελείται από το εργαλείο προσδιορίζει παράλληλες περιοχές στον κώδικα και παράγει μια παραλληλισμένη έκδοση που περιέχει επισημάνσεις OpenMP που επιτρέπουν τη χρήση πολλαπλών νημάτων κατά την εκτέλεση του στοιχείου σε ένα σύστημα πολλαπλών CPU. Οι παράλληλες παραλλαγές που παράγονται αποθηκεύονται παράλληλα με το αρχικό στοιχείο.

Μετά την προεπεξεργασία του στοιχείου, το εργαλείο εφαρμόζει μερικές από τις πιο πρόσφατες τεχνικές συλλογής για να μοντελοποιήσει τον πηγαίο κώδικα. Μια αναπαράσταση εξαρτώμενης μήτρας διευρυμένης κατεύθυνσης (EDM) χρησιμοποιείται για την κάλυψη των μη κοινών φωλιών βρόχου που περιβάλλουν μόνο μία από τις δύο δηλώσεις, προκειμένου να χειριστούν μη τέλεια ενωμένους βρόχους. Για τις προσπελάσεις πινάκων εντός των βρόχων, ένας Gaussian αλγόριθμος εξάλειψης χρησιμοποιείται για την επίλυση ενός συνόλου γραμμικών ακέραιων εξισώσεων των μεταβλητών επαγωγής του βρόχου. Η εισαγωγή τέτοιων μοντέλων βοηθά στον εντοπισμό αλλιώς ανιχνεύσιμων μεγάλων δυνατοτήτων παραλληλισμού.

Τα στάδια που περιλαμβάνονται στην ανάλυση είναι τα εξής:

- Εφαρμογή προαιρετικών προσαρμοσμένων μετασχηματισμών με βάση τη σημασιολογία κώδικα εισόδου, όπως η μετατροπή διαδρομών δέντρων σε επαναλήψεις βρόχων σε ομάδες μνήμης.

- Κανονικοποίηση βρόχων, συμπεριλαμβανομένων αυτών που χρησιμοποιούν μεταβλητές επανάληψης.

- Εύρεση βρόχων υπολογισμού υποψήφιου πίνακα με κανονικές μορφές (for omp for) ή βρόχων και συναρτήσεων που λειτουργούν σε μεμονωμένα στοιχεία (for omp task).

- Για κάθε υποψήφιο βρόχο/συνάρτηση:

  – Απόρριψη του υποψηφίου εάν υπάρχουν κλήσεις συναρτήσεων χωρίς γνωστή σημασιολογία ή παρενέργειες.

  – Εκκίνηση ανάλυσης εξαρτήσεων και liveness μεταβλητών.

  – Ταξινόμηση OpenMP μεταβλητών (autoscoping), αναγώριση αναφορών στο τρέχον στοιχείο και εύρεση προσβάσεων εγγραφής που είναι ανεξάρτητες από τη σειρά εκτέλεσης.

  – Εξάλειψη εξαρτήσεων που σχετίζονται με τις autoscoped μεταβλητές, εκείνες που αφορούν μόνο τα τρέχοντα στοιχεία και εξαρτήσεις εξόδου (WAW) που προκαλούνται από ανεξάρτητες από την σειρά εγγραφές.

  – Εισαγωγή των αντίστοιχων επισημειώσεων OpenMP αν δεν υπάρχουν εξαρτήσεις.

- Τοπική αποθήκευση των νέων παραλληλοποιημένων OpenMP εκδόσεων του κώδικα.

Τέλος, το στοιχείο υποβάλλεται σε επεξεργασία για τυχόν αλλαγές που σχολιάστηκαν κατά τη διάρκεια της προεπεξεργασίας του, προκειμένου να διατηρηθεί η αρχική του λειτουργικότητα.

5.4.3 Λεπτομέρειες υλοποίησης

Αξιοποίηση του μοντέλου εφαρμογής

Ένα μεγάλο μέρος της ανάλυσης αφορά την προεπεξεργασία του κώδικα σύμφωνα με τις παρατηρήσεις χρηστών όπως αυτές ορίζονται από το μοντέλο εφαρμογής. Συγκεκριμένα, οι ακόλουθες pragmas αξιοποιούνται από το εργαλείο για τη διευκόλυνση της ανάλυσης:

#pragma aeolus function no-side-effects
Η επισημείωση δηλώνει ότι μια συνάρτηση μπορεί να θεωρηθεί ως ελεύθερη παρενέργειας, που σημαίνει ότι δεν γράφει σε διευθύνσεις μνήμης εκτός της περιοχής της. Αυτό επιτρέπει στο εργαλείο να αγνοεί τυχόν εξαρτήσεις που προέρχονται από οποιεσδήποτε κλήσεις σε αυτή τη λειτουργία.

#pragma aeolus loop no-pointer-aliasing
Η επισημείωση μπορεί να χρησιμοποιηθεί για να δηλώσει ότι οποιοσδήποτε προσπελάσιμος δείκτης / πλέγμα στον επόμενο βρόχο δεν θα επικαλύπτεται με κάποιον άλλο. Αυτές οι πληροφορίες χρησιμοποιούνται για την εξάλειψη εξαρτήσεων που προκύπτουν εξαιτίας έμμεσων προσπελάσεων χρησιμοποιώντας τις τιμές που ορίζονται κατά το χρόνο εκτέλεσης.

#pragma aeolus loop static-vectors
Η επισημείωση μπορεί να χρησιμοποιηθεί για να δηλώσει ότι οποιαδήποτε αντικείμενα κλάσης vector που υπάρχουν στον βρόχο διατηρούν το μέγεθός τους κατά τη διάρκεια όλων των επαναλήψεων, αποφεύγοντας με αυτό τον τρόπο τις εξάρσεις που προκαλούνται από προσβάσεις σε μη κατανεμημένες διευθύνσεις μνήμης. Χρησιμοποιώντας αυτές τις πληροφορίες, το εργαλείο θεωρεί τους vectors που εμφανίζονται στον βρόγχο ανάλυσης ως στατικές συστοιχίες C. Αυτή η ένδειξη βρέθηκε ότι είναι η πιο χρήσιμη αφού η κλάση vector χρησιμοποιείται πολύ στις εφαρμογές C ++ πραγματικού κόσμου.

Ο μεταγλωττιστής ROSE

Το εργαλείο χρησιμοποιεί τον μεταγλωττιστή ROSE κατά τη διάρκεια της ανάλυσης κώδικα σε κάθε στοιχείο ή εξωτερική βιβλιοθήκη που ορίζεται από τον χρήστη. Το ROSE είναι μια υποδομή μεταγλωττιστή ανοιχτού κώδικα για την κατασκευή εργαλείων μετασχηματισμού και ανάλυσης προγραμμάτων πηγής προς πηγή για εφαρμογές μεγάλης κλίμακας. Συγκεκριμένα, το εργαλείο χρησιμοποιεί το εργαλείο autoPar, το οποίο είναι μια εφαρμογή αυτόματης παραλληλοποίησης με το OpenMP και μπορεί να εισάγει αυτόματα τις οδηγίες OpenMP 3.0 σε σειριακό κώδικα εισόδου C / C ++.

Η ενσωμάτωση με το Compiler ROSE έχει τοποθετηθεί καλά στην τελική δομή του εργαλείου με αποτέλεσμα την υιοθέτηση των τελευταίων διαθέσιμων τεχνολογιών, ενώ παράλληλα ικανοποιεί και τους περιορισμούς όσον αφορά τις απαιτήσεις περί χρήσης. Με την υποστήριξη του μοντέλου προγραμματισμού, το εργαλείο είναι σε θέση να επεκτείνει τη λειτουργικότητα του εργαλείου autoPar, εξαλείφοντας πολλές εξάρσεις που καθιστούν τις εφαρμογές του πραγματικού κόσμου δύσκολο να παραλληλιστούν, χρησιμοποιώντας ακόμα προηγμένα μαθηματικά μοντέλα για να διεξάγουν την ανάλυση κώδικα.

5.4.4 Εκτεταμένη λειτουργικότητα

Ο σχεδιασμός του εργαλείου εμφανίζει ρητά τη δυνατότητά του να επεκτείνει την υποστήριξή του στις διάφορες τεχνικές αυτόματης παραλληλοποίησης ή στα εργαλεία που υπάρχουν ή ενδέχεται να υπάρχουν στο μέλλον. Η έκδοση των συστατικών στοιχείων είναι μια αυστηρά δυναμική διαδικασία που ενθαρρύνει την προσθήκη νέων εργαλείων στην αλυσίδα εργαλείων για τη συνεχή εξέλιξη του τελικού σχεδιασμού και της εφαρμογής της εφαρμογής στη διαθέσιμη υποδομή υλικού.

Σε αυτή τη διατριβή, η υποστήριξη παραλληλισμού του OpenMP έχει συμπεριληφθεί στο σχεδιασμό του πρωτοτύπου που δείχνει μόνο μία από τις περιπτώσεις που μπορεί να αντιμετωπιστεί από ένα τέτοιο εργαλείο. Επιπλέον έρευνα μπορεί να ενισχύσει τα αποτελέσματα της λειτουργικότητας του εργαλείου, εισάγοντας μια σειρά από διαφορετικές τεχνικές που απευθύνονται σε ένα ευρύτερο φάσμα εφαρμογών καθώς και σε στόχους υλικού που μπορούν να συμβάλουν στη βελτιστοποιημένη εκτέλεση των στοιχείων.

Για να το επεξεργαστούμε, η εργαλειοθήκη που περιγράψαμε σε αυτήν την ενότητα σχεδιάστηκε ειδικά για να είναι επαρκώς αρθρωτή, επιτρέποντας τη χρήση πρόσθετων εργαλείων καθώς γίνονται και άλλες βελτιώσεις. Έτσι, για να συμπεριληφθούν περισσότερες παράλληλες εκδόσεις των στοιχείων, τα εξωτερικά εργαλεία μπορούν απλά να αποθηκεύσουν τις νέες εκδόσεις στον αντίστοιχο κατάλογο και να τροποποιήσουν αναλόγως το συστατικό δίκτυο όπως περιγράφηκε παραπάνω.

5.4.5 Διεπαφές για ανάπτυξη και εκτέλεση εφαρμογών σε ετερογενείς πλατφόρμες

Εκτός από την επικοινωνία μεταξύ διαφορετικών εξαρτημάτων, η επικοινωνία με διαφορετικούς τύπους συσκευών απαιτείται επίσης για να ενισχυθεί η υποστήριξη της ετερογενούς ανάπτυξης του συστήματος. Αυτή είναι μια προσέγγιση που δεν ερευνήθηκε σε βάθος κατά τη διάρκεια της εξέτασης αυτής της διατριβής και, ως εκ τούτου, μπορεί να υποστηρίξει μόνο πολύ απλοϊκές περιπτώσεις. Περιλαμβάνεται εδώ, μόνο ως απόδειξη της έννοιας, για να υποστηρίξει την ποικιλομορφία των τεχνικών που μπορούν να χρησιμοποιηθούν στο πλαίσιο της χρήσης της υποδομής.

Μια διεπαφή χρήστη για υποστήριξη GPU

Μια προσπάθεια υποστήριξης αυτού του είδους των συστημάτων επικεντρώνεται σε ένα συγκεκριμένο API που προορίζεται να βοηθήσει τον χρήστη με κινήσεις δεδομένων προς και από τη συσκευή καθώς και με την ανάπτυξη του κώδικα του πυρήνα που επιτρέπει την επιτάχυνση της πληροφορικής. Αυτό το API περιλαμβάνει ένα σύνολο σημείων pragma ειδικά σχεδιασμένων ώστε να επιτρέπουν την εισαγωγή μεταβλητών εισόδου / εξόδου καθώς και τις λειτουργίες του πυρήνα που πρέπει να καλούνται. Ως αποτέλεσμα, παράγονται οι αντίστοιχες λειτουργίες CUDA, συμπληρώνοντας τον πηγαίο κώδικα εξαρτήματος για ανάπτυξη σε μια GPU.

Η μορφή των pragmas που θα χρησιμοποιηθούν ορίζεται ως εξής:

Defining a kernel input:
#pragma aeolus kernel <function> kernelin <name> type=<type> size=<size>

Defining a kernel output:
#pragma aeolus kernel <function> kernelout <name> type=<type> size=<size>

Ανάπτυξη σε συσκευές GPU

Με τη χρήση αυτού του API, το εργαλείο μπορεί να μετασχηματίσει τον πηγαίο κώδικα του εξαρτήματος παράγοντας μια έκδοση του αντίστοιχου συστατικού που μπορεί να αναπτυχθεί σε μια GPU. Παρακάτω, εμφανίζεται ένα παράδειγμα των μετασχηματισμών κώδικα για την κατανόηση των αποτελεσμάτων μιας τέτοιας ανάλυσης. Πρωτότυπη μορφή του κωδικού εξαρτήματος:

```
int *vec_add(int *a, int *b) {
    int c[10000];

    #pragma aeolus kernel vec_add_CUDA kernelin a type=int size=10000
    #pragma aeolus kernel vec_add_CUDA kernelin b type=int size=10000

    for(int i=0; i<10000; i++)
        c[i] = a[i] + b[i];

    #pragma aeolus kernel vec_add_CUDA kernelout c type=int size=10000

    return c;
}
```

Εδώ είναι ορατή η χρήση του API GPU, ορίζοντας τις συστοιχίες a και b ως είσοδοι και τον πίνακα c ως έξοδο για τη λειτουργία του πυρήνα vec_add_CUDA(). Τα αποτελέσματα της ανάλυσης και η παραγωγή της έκδοσης GPU του στοιχείου εμφανίζονται παρακάτω:

```
int *vec_add(int *a, int *b) {

int c[10000];

    //Declare pointers for the GPU to use
int *dev_a, *dev_b, *dev_c;

    //Allocate memory on the GPU
    cudaMalloc((void**)&dev_a, 10000*sizeof(int));
    cudaMalloc((void**)&dev_b, 10000*sizeof(int));
    cudaMalloc((void**)&dev_c, 10000*sizeof(int));

    //Copy the arrays a and b to the GPU
    cudaMemcpy( dev_a, a, 10000*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy( dev_b, b, 10000*sizeof(int), cudaMemcpyHostToDevice);

    //Launch the kernel with 100/128 thread blocks of 128 threads
    dim3 block(128);
    dim3 grid((10000+ 127)/128);
    vec_add_CUDA<<<grid, block>>>(dev_a, dev_b, dev_c, 10000);

    //Copy the array c back from the GPU to the CPU
    cudaMemcpy( c, dev_c, 10000*sizeof(int), cudaMemcpyDeviceToHost);

    //Free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return c;
}
```

5.5 Επιλογή Τεχνικών - Technique Selection

5.5.1 Περιγραφή

Ο σκοπός της δημιουργίας μιας δέσμης διαφορετικών παραλληλισμένων εκδόσεων των συστατικών στοιχείων καθώς και διαφορετικών εκδόσεων των API επικοινωνίας έγκειται στη βελτιστοποίηση της απόδοσης της εφαρμογής, ανάλογα με τη φύση των στοιχείων και τους διαθέσιμους στόχους υλικού. Προκειμένου να επιτευχθεί αυτός ο στόχος, υπάρχει άμεση ανάγκη για έναν μηχανισμό απόφασης που θα επιλέξει τις κατάλληλες υλοποιήσεις που σε συνδυασμό με το συγκεκριμένο σχέδιο ανάπτυξης θα έχουν ως αποτέλεσμα την αύξηση της απόδοσης και την ελαχιστοποίηση του αποτυπώματος ισχύος και μνήμης.

5.5.2 Επιλογή έκδοσης κώδικα

Στο πρώτο στάδιο της λειτουργικότητάς του, το εργαλείο επιλέγει την καλύτερη τεχνολογία παραλληλισμού για κάθε στοιχείο (π.χ. OpenMP για συστήματα CPU πολλαπλών πυρήνων, CUDA ή OpenCL για GPU, πυρήνες IP για FPGA κ.λπ.) σύμφωνα με το σχέδιο ανάπτυξης και δημιουργεί το πραγματικό σύνολο των στοιχείων που θα χρησιμοποιηθούν κατά την ανάπτυξη.

Σε αυτή τη διατριβή, οι τεχνολογίες που υποστηρίζονται από το εργαλείο αυτόματης παραλληλοποίησης AEOLUS περιλαμβάνουν μόνο την έκδοση OpenMP των εξαρτημάτων. Ωστόσο, όπως περιγράφεται στην ενότητα 5.4.5, παρέχεται ένα API για την ανταλλαγή και την ανάπτυξη δεδομένων σε στόχους GPU, που επιτρέπει την εκμετάλλευση των πυρήνων που έχουν αναπτυχθεί από τον χρήστη CUDA. Έτσι, στην περίπτωση που ένα στοιχείο σχεδιάζεται να τρέξει σε ένα σύστημα CPU, επιλέγεται η έκδοση του OpenMP, επιτρέποντας τη δυνατότητα παράλληλης εκτέλεσης του στοιχείου χρησιμοποιώντας πολλαπλά νήματα. Από την άλλη πλευρά, εάν το σχέδιο ανάπτυξης αναθέτει το στοιχείο σε μια GPU για να επιταχύνει την εκτέλεση του, επιλέγεται η έκδοση CUDA (που παρέχεται από τον χρήστη), υποθέτοντας ότι το API χρησιμοποιήθηκε με επιτυχία στον κώδικα συνιστωσών.

```
public void TechSel {
    boolean gpu_flag=false, fpga_flag=false;
    for(int i=0; i<Components.size(); i++) {
        if(Components.get(i).getLoops().size() == 0) {
            Components.get(i).setFinalVersion“(”OpenMP);
            continue;
        }
        else {
            for(int j=0; j<Components.get(i).getLoops().size(); j++) {
                if(Components.get(i).getLoops().get(j).RunsOnGpu()) {
                    gpu_flag = true;
                }
                else if( Components.get(i).getLoops().get(j).RunsOnFpga) {
                    fpga_flag = true;
                }
            }
        }
        if(gpu_flag && !fpga_flag) {
            Components.get(i).setFinalVersion“(”CUDA);
        }
        else if(!gpu_flag && fpga_flag) {
            Components.get(i).setFinalVersion“(”VHDL);
        }
    }
}
```

Η διαδικασία της επιλογής που περιγράφεται παραπάνω αφορά τη μετάφραση του σχεδίου εγκατάστασης από μια εκπροσώπηση υλικού χαμηλότερου επιπέδου των συστατικών στοιχείων σε ένα λογισμικό υψηλότερου επιπέδου. Οι τελικές εκδόσεις των συστατικών στοιχείων αποθηκεύονται ξεχωριστά από τις άλλες διαθέσιμες εκδόσεις και είναι έτοιμες να βελτιωθούν και να αναπτυχθούν από

το Deployment Manager (ενότητα 5.6).

5.5.3 Επιλογή τεχνικής εκτέλεσης

Το μοντέλο εφαρμογής ορίζει ότι ένα σύνολο στοιχείων θα λειτουργεί ανεξάρτητα και παράλληλα, ενώ αλληλεπιδρούν μεταξύ τους, για να αποτελέσει τη συνολική λειτουργικότητα της εφαρμογής. Αυτός είναι ένας γενικός ορισμός που επιτρέπει κάποια ευελιξία στην τεχνολογία που μπορεί να χρησιμοποιηθεί για την ανάπτυξη της εφαρμογής, καθώς και για την επικοινωνία μεταξύ των στοιχείων. Έτσι, ένας μηχανισμός λήψης αποφάσεων είναι απαραίτητος για να αποφασιστούν τα πρότυπα τεχνολογίας που πρέπει να ακολουθήσει η εφαρμογή του.

Ωστόσο, αυτή η απόφαση δεν έχει ένα τόσο ευρύ φάσμα επιλογών, λόγω της υψηλής αξιοπιστίας του προς το σχέδιο ανάπτυξης που καθορίζει τη θέση τους πάνω στην πλατφόρμα υλικού. Τα στοιχεία που αντιστοιχίζονται στον ίδιο κόμβο επεξεργασίας χρησιμοποιούν τον ίδιο χώρο διευθύνσεων, επομένως οι βιβλιοθήκες όπως το pthreads και το OpenMP μπορούν να χρησιμοποιηθούν παράλληλα σε ξεχωριστά θέματα. Από την άλλη πλευρά, τα στοιχεία που έχουν εκχωρηθεί για εκτέλεση σε διαφορετικούς κόμβους πρέπει να χρησιμοποιούν άλλους τύπους βιβλιοθηκών για να λειτουργούν σε ένα περιβάλλον όπου οι κόμβοι εργάζονται σε διαφορετικό χώρο διευθύνσεων. Η χρήση μιας εφαρμογής MPI όπως MPICH και OpenMPI είναι μια κοινή λύση σε τέτοιες περιπτώσεις ειδικά για γλώσσες χαμηλού επιπέδου, όπως το C.

Στο πλαίσιο αυτής της εργασίας θα συζητήσουμε μια εφαρμογή του εισαγόμενου συστήματος χρησιμοποιώντας μόνο pthreads και OpenMPI (βλέπε κεφάλαιο 5.8: Deployment Manager), αλλά όπως και με τις υπόλοιπες μονάδες AEOLUS έχουν αναπτυχθεί μόνο βασικές λειτουργίες, έτσι ώστε μελλοντικά οι επεκτάσεις μπορούν να προστεθούν διευρύνοντας την υποστήριξη του συστήματος για περισσότερες βιβλιοθήκες που μπορούν να βελτιστοποιήσουν τα αποτελέσματα της ανάπτυξης. Η στρατηγική που θα ακολουθήσουμε σχετικά με την τεχνολογική επιλογή της εγκατάστασης σχετίζεται αυστηρά με τις οδηγίες του σχεδίου ανάπτυξης σχετικά με τα αντικείμενα επικοινωνίας. Κάθε αντικείμενο επικοινωνίας προσεγγίζεται από ένα σύνολο στοιχείων που μπορούν να εκτελεστούν στον ίδιο ή σε άλλο χώρο μνήμης, οπότε η κατανομή αυτών των οντοτήτων σε όλο το υπολογιστικό συνεχές καθορίζει τον χαρακτήρα εφαρμογής τους. Συγκεκριμένα, μελετώνται δύο διαφορετικές περιπτώσεις σχετικά με την υλοποίηση ενός αντικειμένου επικοινωνίας:

- **Όλα τα στοιχεία που σχετίζονται με το αντικείμενο εκτελούνται σε ένα σύστημα κοινής μνήμης**
  Στη συνέχεια, τα δεδομένα μπορούν να ανταλλαγούν με ασφάλεια ως μια διαμοιραζόμενη δομή στη μνήμη. Τα στοιχεία εκτελούνται ως ξεχωριστά νήματα χρησιμοποιώντας τη βιβλιοθήκη pthreads μαζί με τα πρότυπα POSIX για την πρόσβαση στα δεδομένα.

- **Τα στοιχεία που σχετίζονται με το αντικείμενο εκτελούνται σε διαφορετικά συστήματα μνήμης**
  Ένα πρωτόκολλο μετάδοσης μηνυμάτων είναι απαραίτητο για τη μεταφορά των δεδομένων. Τα στοιχεία εκτελούνται ως ξεχωριστές διεργασίες MPI χρησιμοποιώντας τη βιβλιοθήκη OpenMPI, ενώ τα API που παρέχονται από το MPI χρησιμοποιούνται επίσης για την πρόσβαση στα δεδομένα.

Σύμφωνα με τις παραπάνω αποφάσεις, οι αντίστοιχες υλοποιήσεις των API επιλέγονται και αποθηκεύονται μαζί με την τελική έκδοση των συστατικών.

5.5.4 Εκτεταμένη λειτουργικότητα

Ο σχεδιασμός αυτής της μονάδας εμφανίζει σαφώς τη δυνατότητά της να επεκτείνει την υποστήριξή της στις διάφορες τεχνικές ή βιβλιοθήκες που υπάρχουν ή ενδέχεται να υπάρχουν στο μέλλον. Η επέκτασή της συμβαδίζει με το εργαλείο αυτόματης παραλληλοποίησης, το οποίο αφήνει περιθώρια για μελλοντικές προσθήκες, με αποτέλεσμα να διαλέξετε πιο τροποποιημένες εκδόσεις των αρχικών

εξαρτημάτων. Όπως και με τη συμπερίληψη των εκδόσεων CUDA που υποστηρίζονται από το GPU API, περισσότερες εκδόσεις μπορούν επίσης να παρέχονται από τον χρήστη, υποθέτοντας ότι υπάρχει η αντίστοιχη υποστήριξη για την ανάπτυξή τους στην διαθέσιμη υποδομή υλικού.

5.6 Διαχειριστής Εκτέλεσης

5.6.1 Περιγραφή

Ας υποθέσουμε μια εφαρμογή που ακολουθεί το μοντέλο που περιγράφηκε στην ενότητα 4 και μια περίπλοκη ετερογενή πλατφόρμα υλικού που περιλαμβάνει διαφορετικές συσκευές και αρχιτεκτονικές στις οποίες πρόκειται να εκτελεστεί η εφαρμογή. Σε ετερογενή υπολογιστικά συστήματα, ο προγραμματιστής είναι υπεύθυνος για την υλοποίηση της ανάπτυξης της εφαρμογής στα διάφορα εξαρτήματα υλικού, εξασφαλίζοντας ταυτόχρονα τις ανάγκες επικοινωνίας που μπορεί να προκύψουν μεταξύ των στοιχείων του λογισμικού ή των εξαρτημάτων και των εξωτερικών συσκευών. Λαμβάνοντας υπόψη όλες τις διαφορετικές βιβλιοθήκες και εργαλεία που πρέπει να χρησιμοποιηθούν και τις διαφορετικές διαμορφώσεις που πρέπει να πραγματοποιηθούν για την ανάπτυξη μιας εφαρμογής σε ένα ετερογενές περιβάλλον, μπορεί να είναι μια πολύ σύνθετη και χρονοβόρα διαδικασία.

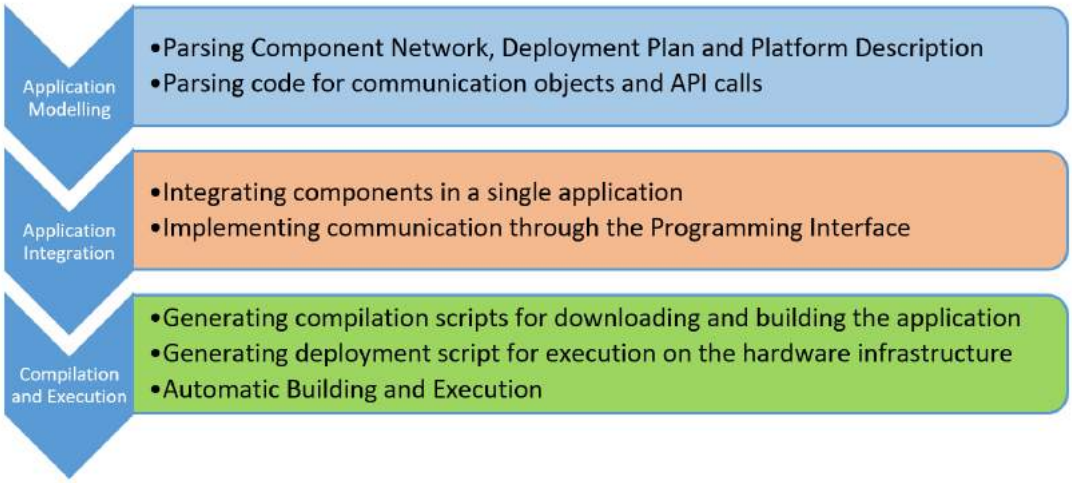
Ο στόχος του AEOLUS είναι να ελαχιστοποιήσει την προσπάθεια και τις γνώσεις που απαιτούνται από τον προγραμματιστή για την ανάπτυξη μιας εφαρμογής που εκτελείται σε ετερογενή συστήματα. Η ιδέα που συζητάμε εδώ είναι για μια ξεχωριστή ενότητα της αρχιτεκτονικής του πλαισίου που θα θέσει ως προγραμματιστής να κάνει όλες τις απαραίτητες διαμορφώσεις για να αναπτύξει τα στοιχεία παράλληλα σε σχέση με την επικοινωνία μεταξύ τους. Για την υλοποίηση ενός τέτοιου παράγοντα, ο κώδικας όπου πρέπει να οριστούν οι απαραίτητες δομές και λειτουργίες πρέπει να δημιουργηθεί κατά τον χρόνο σύνταξης, έτσι ώστε ολόκληρη η εφαρμογή να μπορεί να συντονιστεί κατά την εκτέλεση του χρόνου. Αλλά η μείωση της προσπάθειας του ανθρώπινου παράγοντα σημαίνει τη δημιουργία ενός τεχνητού μέρους που είναι υπεύθυνο για όλα τα τεχνικά μέρη της ανάπτυξης που ο χρήστης δεν μπορεί ή δεν πρέπει να είναι.

Σε αυτήν την ενότητα θα συζητηθεί η λειτουργικότητα ενός εργαλείου που αναφέρεται ως ”Διαχειριστής Εκτέλεσης”. Οι διαφορετικές προσαρμογές και βελτιώσεις κώδικα εκτελούνται ως τελικό στάδιο της ανάπτυξης της εφαρμογής, ακολουθώντας την ανάλυση και τα αποτελέσματα των προηγουμένως συζητημένων εργαλείων και μοντέλων εφαρμογών όπως ορίζονται από τον χρήστη. Επιπλέον, παράγεται ένα σύνολο σεναρίων που αντιστοιχούν στην παραπάνω διαδικασία, υλοποιώντας την πραγματική συλλογή και ανάπτυξη των στοιχείων στο διαθέσιμο υλικό.

5.6.2 Προδιαγραφές σχεδίασης και διεπαφές

Όπως ήδη αναφέρθηκε, ο υπεύθυνος ανάπτυξης είναι υπεύθυνος για μια σειρά ενεργειών σχετικά με την ενσωμάτωση των στοιχείων και την αυτόματη σύνταξη εκτέλεση τους, προκειμένου να διευκολυνθεί η διαδικασία ανάπτυξης. Σε αυτή την ενότητα, αυτά τα βήματα θα περιγραφούν λεπτομερώς, ενώ στην επόμενη ενότητα θα συμπεριληφθούν και ορισμένες λεπτομέρειες εφαρμογής χαμηλότερου επιπέδου. Στο σχήμα 5.3, απεικονίζεται η λειτουργικότητα του εργαλείου.

Όπως φαίνεται στο σχήμα 5.3, η ροή εργαλείου του Διαχειριστή Εκτέλεσης μπορεί να χωριστεί σε τρία στάδια. Πρώτον, υπάρχει η μοντελοποίηση της εφαρμογής όπου χρησιμοποιείται ένα σύνολο κατηγοριών Java για την περιγραφή των στοιχείων, ο τρόπος επικοινωνίας μεταξύ τους, η χαρτογράφηση τους στην πλατφόρμα υλικού και πληροφορίες σχετικά με τα δεδομένα ή τα σήματα που χρειάζονται για την ανταλλαγή. Δεύτερον, θα πραγματοποιηθεί η δημιουργία των αναγκαίων αρχείων, τα οποία θα επιτρέψουν την ενσωμάτωση των στοιχείων μεταξύ τους, με τον τρόπο αυτό την επικοινωνία μεταξύ των διαφόρων τμημάτων της εφαρμογής. Τέλος, θα δημιουργηθεί δέσμη δέσμης ενεργειών για την κατασκευή και τοποθέτηση των δυαδικών αρχείων στις αντίστοιχες μηχανές.



Σχήμα 5.3: Ροή εκτέλεσης του Διαχειριστή Εκτέλεσης

Μοντελοποίηση εφαρμογών

Για την ενορχήστρωση της ανάπτυξης σύμφωνα με τις οδηγίες του σχεδίου ανάπτυξης και της ενότητας επιλογής τεχνικής, ο υπεύθυνος ανάπτυξης πρέπει να συλλέξει όλες τις απαραίτητες πληροφορίες για να μοντελοποιήσει την εφαρμογή ανά συνιστώσα συνοδευόμενη από τα αντίστοιχα αντικείμενα επικοινωνίας. Αυτές οι πληροφορίες είναι κατάλληλα ενσωματωμένες στα αρχεία XML του Component Network and Deployment Plan καθώς και μέσα στον πηγαίο κώδικα με τη μορφή σημείων pragma.

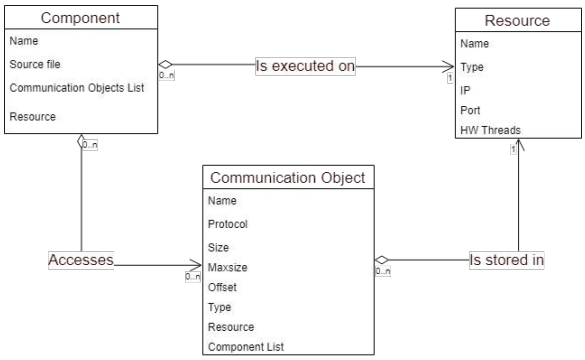
1. Ανάλυση αρχείων μοντέλων XML

Πρώτον, οι απαραίτητες πληροφορίες εξάγονται από το δίκτυο συνιστωσών προκειμένου να κατασκευαστεί ένα μοντέλο εφαρμογής χρησιμοποιώντας ένα σύνολο κατηγοριών Java. Παρέχονται πληροφορίες όπως τα ονόματα των συστατικών στοιχείων, τα αρχεία προέλευσης και οι διαδρομές στο χώρο αποθήκευσης, καθώς και η τοποθέτηση των στοιχείων μέσα στο δίκτυο των στοιχείων, που σημαίνει τις αλληλεπιδράσεις τους με άλλα στοιχεία και τα αντίστοιχα αντικείμενα επικοινωνίας. Επιπλέον, βασικές πληροφορίες σχετικά με τα αντικείμενα επικοινωνίας λαμβάνονται επίσης για να συμπεριληφθούν στο δίκτυο. Χαρακτηριστικά που χαρακτηρίζουν αυτά τα αντικείμενα επικοινωνίας, όπως τα ονόματα, το μέγεθος, η πηγή και ο στόχος τους, εξάγονται από το δίκτυο συνιστωσών προκειμένου να δημιουργηθούν οι αντίστοιχες οντότητες στο μοντέλο.

Η περιγραφή της πλατφόρμας αναλύεται επίσης για τη συλλογή των δεδομένων που αφορούν τα διαθέσιμα στοιχεία υλικού. Ο σχεδιασμός της αρχιτεκτονικής του συστήματος περιλαμβάνεται στο μοντέλο εφαρμογής καθώς και πληροφορίες για κάθε μεμονωμένο μηχάνημα ή συσκευή. Έτσι, τα χαρακτηριστικά όπως η συχνότητα, το εύρος ζώνης, η μνήμη cache, ο αριθμός πυρήνων, η διεύθυνση IP συγκεντρώνονται για τον συντονισμό της ανάπτυξης στην υποδομή υλικού. Η συσχέτιση μεταξύ των στοιχείων του λογισμικού και των αντικειμένων επικοινωνίας τους, καθώς και των εξαρτημάτων υλικού της πλατφόρμας, επιτυγχάνεται με την ανάλυση του Σχεδίου Ανάπτυξης. Κάθε συστατικό στοιχείο και αντικείμενο επικοινωνίας χαρτογραφείται σε έναν συγκεκριμένο στοιχείο υλικού, οπότε και αυτή η σύνδεση περιλαμβάνεται στο μοντέλο εφαρμογής συμπληρώνοντας τη δομή του.

2. Αναγνώριση αντικειμένου επικοινωνίας

Το AEOLUS είναι υπεύθυνο για τη δημιουργία ενός κοινού περιβάλλοντος για τα συστατικά που πρόκειται να εκτελεστούν, σε ένα υψηλότερο λογικό επίπεδο από αυτό στο οποίο λειτουργούν τα στοιχεία. Όλα τα κοινά δεδομένα βρίσκονται στο χώρο ονομάτων που μοιράζεται μεταξύ όλων των στοιχείων που αποτελούν την εφαρμογή. Εάν δύο στοιχεία κοινόχρηστων δεδομένων ονομάζονται



Σχήμα 5.4: Μοντέλο εφαρμογής

τα ίδια, αντιμετωπίζονται ως το ίδιο στοιχείο από την υποδομή. Ο χώρος ονομάτων κοινών δεδομένων ορίζεται ως το σύνολο στοιχείων κοινόχρηστων δεδομένων σε όλα τα στοιχεία του συστήματος. Αυτό δεν πρέπει να συγχέεται με τους χώρους ονομάτων στη γλώσσα C που περιορίζονται σε ένα μεμονωμένο στοιχείο.

```
1 // Declaring a communication object called the_data, linked to the // local
      variable data
2 #pragma aeolus shared in the_data offset=896
3 uint8_t data[128];
```

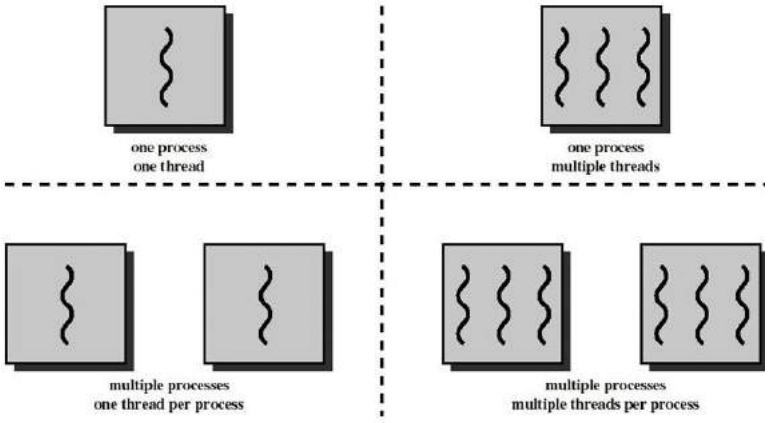
Στο παραπάνω παράδειγμα, η μεταβλητή the_data βρίσκεται στο κοινόχρηστο χώρο ονομάτων δεδομένων σε όλο το σύστημα, τα δεδομένα είναι τοπικά στο στοιχείο C σύμφωνα με τους κανονικούς κανόνες οριοθέτησης του C.

Προκειμένου να προσδιοριστούν τα δεδομένα επικοινωνίας και να συνδεθούν οι τοπικές μεταβλητές που αφορούν τα ίδια αντικείμενα επικοινωνίας, ένα προκαταρκτικό συμπλήρωμα του κώδικα θα ξεκινήσει την ανάλυση. Συγκεκριμένα, θα χρησιμοποιηθούν δύο σχολιασμοί pragma που παρέχονται από το μοντέλο προγραμματισμού, έτσι ώστε τα αντικείμενα επικοινωνίας να μπορούν να δηλωθούν σε περιβάλλον υψηλότερου επιπέδου. Όταν δηλώνεται ένα αντικείμενο επικοινωνίας, δηλώνεται επίσης η αντίστοιχη πληροφορία, όπως το μέγεθός του, το όνομα της τοπικής μεταβλητής στο οποίο αποθηκεύονται τα δεδομένα, το πρωτόκολλο που χρησιμοποιείται για τη μεταφορά κλπ. Σύμφωνα με το πρωτόκολλο που έχει οριστεί για ένα το αντικείμενο επικοινωνίας, ο τρόπος με τον οποίο γίνεται πρόσβαση σε αυτό το αντικείμενο, αλλάζει και οι απαιτήσεις συνέπειας των δεδομένων. Μερικοί εξασφαλίζουν διαδοχική συνέπεια ή ισχυρότερη, ενώ άλλοι δεν ορίζουν καμία συνέπεια και ο προγραμματιστής πρέπει να χρεώσει με το χέρι συγχρονισμό.

Το μοντέλο εφαρμογής συμπληρώνεται με τις πληροφορίες που προέρχονται από τον πηγαίο κώδικα, οπότε ο Διαχειριστής Εκτέλεσης μπορεί να προχωρήσει στην υλοποίηση της πραγματικής ανάπτυξης. Η αρχιτεκτονική του μοντέλου που δημιουργήθηκε ακολουθεί τη δομή του διαγράμματος στο Σχήμα 5.4.

Κατασκευή μιας εφαρμογής πολλαπλών διεργασιών

Υπάρχουν πολλές βιβλιοθήκες που μπορούν να χρησιμοποιηθούν για την παράλληλη εκτέλεση των στοιχείων. Καθοδηγούμενη από την επιλογή βιβλιοθήκης της ενότητας επιλογής τεχνικής, ο υπεύθυνος ανάπτυξης αναπτύσσει την υλοποίηση της συντονισμένης εκτέλεσης των στοιχείων, μαζί με την επικοινωνία που απαιτείται μεταξύ τους. Το μοντέλο προγραμματισμού εγγυάται την ανεξαρτησία των στοιχείων και έχει σχεδιαστεί ειδικά έτσι ώστε οι μέθοδοι επικοινωνίας που χρησιμοποιούνται μπορούν να επιλεγούν μεταξύ ποικίλων επιλογών, συμπεριλαμβανομένων των διαφορετικών βιβλιοθηκών, τεχνολογιών και πρωτοκόλλων. Επομένως, τα στοιχεία θα πρέπει να μπορούν να εκτελούνται ως ανεξάρτητες διεργασίες που μπορούν να επικοινωνούν μέσω του δικτύου, μέσω μηχανισμών υποδοχής ή ακόμα και χρησιμοποιώντας τοπικό ή cloud storage για να μιλούν ο ένας στον άλλο μέσω του



Σχήμα 5.5: Διαφορετικές υλοποιήσεις του SPMD μοντέλου με χρήση νημάτων και διεργασιών

Programming Interface. Οι διαφορετικές εφαρμογές μπορεί να διαφέρουν ανάλογα με τις διαφορετικές απαιτήσεις που συναντώνται κατά την ανάπτυξη.

Το AEOLUS έχει σχεδιαστεί για να χρησιμοποιεί την ποικιλία εργαλείων που έχουν αναπτυχθεί για τη βελτιστοποίηση της απόδοσης σύμφωνα με τις πολυδιάστατες απαιτήσεις όπως ο χρόνος εκτέλεσης και η κατανάλωση ενέργειας. Για να ακολουθήσει αυτό το κίνητρο, ο Διαχειριστής Εκτέλεσης υλοποιεί την ανάπτυξη των στοιχείων σε σχέση με αυτές τις απαιτήσεις. Για το σκοπό αυτό, η διεπαφή προγραμματισμού υιοθέτησε το πρωτόκολλο MPI για εκτέλεση σε συστήματα διανεμημένης μνήμης και ένα σύνολο τεχνικών δημιουργίας νήματος και συγχρονισμού για περαιτέρω παραλληλισμό σε κάθε κόμβο και συντονισμό μεταξύ τους.

Η αρχιτεκτονική της εφαρμογής έχει σχεδιαστεί για να ακολουθεί το μοντέλο SPMD για την εκτέλεση των διαφορετικών εργασιών που ζητούνται από τον χρήστη παράλληλα. Μια αφηρημένη (και εύκολη ακολουθία) υλοποίηση της ανάπτυξης των συστατικών στοιχείων υιοθετεί πλήρως αυτό το μοντέλο και υλοποιεί την κορυφαία δεξιά θήκη από το Σχήμα 5.5, στο οποίο το πρόγραμμα δημιουργεί πολλαπλά νήματα για την εκκίνηση και την εκτέλεση των στοιχείων. Αυτό το μοντέλο μπορεί να εφαρμοστεί μόνο στην περίπτωση ενός συστήματος κοινόχρηστης μνήμης, όταν ο χρήστης αποφασίσει ότι όλα τα στοιχεία θα πρέπει να εκτελούνται στον ίδιο κόμβο/SMP.

Σε μια παρόμοια απλή περίπτωση, όταν ο χρήστης έχει δώσει εντολή στην εφαρμογή να εκτελείται σε ένα πιο περίπλοκο περιβάλλον από ένα και μόνο μηχάνημα (π.χ. να χρησιμοποιούν διαφορετικούς κόμβους ενός συστήματος συμπλέγματος), απαιτείται ένα πρωτόκολλο μετάδοσης μηνυμάτων για την ανταλλαγή δεδομένων σε μια κατανεμημένη μνήμη space, έτσι ώστε να υιοθετηθεί το πρωτόκολλο MPI. Αυτή είναι η κάτω αριστερή περίπτωση στο Σχήμα 5.5. Σε αυτήν την περίπτωση, τα διαφορετικά στοιχεία εκτελούνται το καθένα ως ξεχωριστή διαδικασία MPI, σύμφωνα με το σχέδιο ανάπτυξης, όλα λειτουργούν παράλληλα. Η εφαρμογή Διεπαφής Προγραμματισμού χρησιμοποιεί τις αντίστοιχες λειτουργίες MPI για τη μεταφορά δεδομένων και σημάτων συγχρονισμού μεταξύ των στοιχείων. Ο Διαχειριστής Εκτέλεσης είναι υπεύθυνος για τη δημιουργία των απαραίτητων αρχείων για τη δέσμευση των στοιχείων σε μια εφαρμογή MPI και για τη δήλωση των δομών που απαιτούνται για την υλοποίηση της επικοινωνίας. Αυτά τα αρχεία περιγράφονται λεπτομερώς στην ενότητα 'Υλοποίηση'.

Μια πιο περίπλοκη εφαρμογή θα προσπαθήσει να μειώσει τα γενικά έξοδα που προκαλούνται από τη δημιουργία διαφορετικών διαδικασιών MPI, ενώ παράλληλα θα εκμεταλλευτεί τις δυνατότητες πολλών πυρήνων των διαθέσιμων μηχανημάτων. Για το σκοπό αυτό, η εκτέλεση των στοιχείων που εκτελούνται στον ίδιο χώρο μνήμης υλοποιείται ως διαφορετικά νήματα της ίδιας διαδικασίας. Το αποτέλεσμα είναι παρόμοιο με το κατώτατο δικαίωμα στην εικόνα 5.5. Ένα σύνολο διαδικασιών MPI ξεκινάει (ένα για κάθε διαφορετικό μηχάνημα), καθένα από τα οποία δημιουργεί μια σειρά από θέματα για κάθε στοιχείο που έχει αντιστοιχιστεί στο χώρο της μνήμης του. Αυτή είναι η πιο γενική

περίπτωση των παραπάνω, όπου η παράλληλη παραλλαγή των διαφορετικών συνιστωσών επιτυγχάνεται σε επίπεδο τοπικής μνήμης από μια οπτική γωνία, αλλά και σε επίπεδο κατανεμημένης μνήμης από ένα πολυεπεξεργασμένο.

Όλες οι προσεγγίσεις έχουν τα δικά τους οφέλη και τα μειονεκτήματά τους όσον αφορά τη συμπερίληψη επιπλέον χρόνου επιβάρυνσης, κατανάλωσης ενέργειας ή αποτυπώματος μνήμης. Η καλύτερη προσέγγιση πρέπει να αποφασιστεί με βάση τα διαφορετικά σχέδια και απαιτήσεις της εφαρμογής που αναπτύσσεται. Αυτός είναι ο λόγος για τον οποίο η AEOLUS ενθαρρύνει διαφορετικές επαναλήψεις ανάπτυξης για τη δοκιμή διαφορετικών χαρτογραφιών και τεχνικών, βελτιστοποιώντας τις επιδόσεις του συστήματος χρησιμοποιώντας πολλαπλά αντικειμενικά κριτήρια.

Ενσωμάτωση με τη διεπαφή προγραμματισμού

Η υλοποίηση του Διεπαφή Προγραμματισμού εξαρτάται σε μεγάλο βαθμό από το μοντέλο εφαρμογής που έχει επιλεγεί για την εφαρμογή όπως εξηγείται στην προηγούμενη παράγραφο. Αυτός είναι ο λόγος για τον οποίο κατά τη διάρκεια της ενσωμάτωσης των εξαρτημάτων της εφαρμογής δημιουργούνται επίσης οι απαραίτητες δομές ακολουθώντας τα πρότυπα που προτείνονται από την Τεχνική Επιλογή για την υλοποίηση της επικοινωνίας μεταξύ των συνιστωσών στο περιβάλλον υλικού που υποδεικνύεται από το Σχέδιο Ανάπτυξης. Λεπτομέρειες σχετικά με τις προαναφερόμενες δομές που χρησιμοποιούνται μπορούν να βρεθούν στην ενότητα ”Υλοποίηση”.

Δημιουργία και ανάπτυξη δυαδικών αρχείων

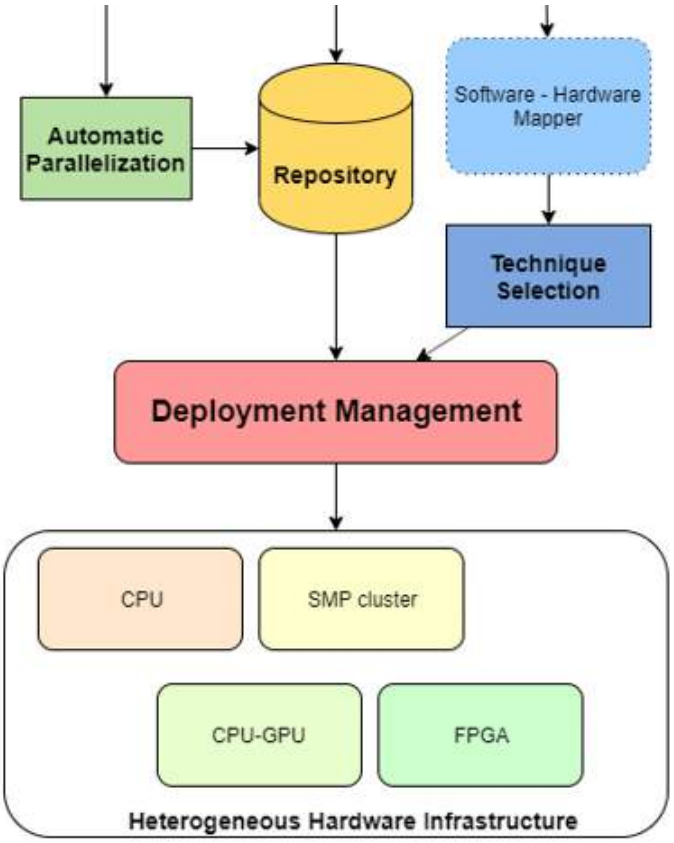
Η ανάπτυξη των στοιχείων στη συγκεκριμένη υποδομή υλικού είναι η βασική λειτουργία του Διαχειριστή Εκτέλεσης. Ενώ αυτό ισχύει, η πραγματική ανάπτυξη είναι μια ενέργεια που έχει εκχωρηθεί στο χρήστη μέσω ενός συνόλου σεναρίων που παράγονται από το εργαλείο. Κατευθυνόμενο από το Deployment Plan, το DM δημιουργεί τα scripts μεταγλώττισης προκειμένου να δημιουργήσει τα δυαδικά αρχεία εφαρμογών στις αντίστοιχες μηχανές / συσκευές όταν ο χρήστης επιλέξει να τις εκτελέσει. Η διαδικασία συλλογής συνεχίζεται σε σχέση με την ελαχιστοποίηση της περιττής δημιουργίας δυαδικών αρχείων και των μεταφορών αρχείων από ή προς το χώρο αποθήκευσης. Τα παρακάτω βήματα δείχνουν τις ενέργειες που ολοκληρώνονται με την εκτέλεση των σεναρίων.

1. Script Μεταγλώττισης

1. Check for the necessary binaries on the target machines.
2. If not found on machines
   (a) Check on Repository.
   (b) If found on Repository
      i. Get binary from Repository
   (c) Else
      i. Get project source folder from Repository
      ii. Compile source files and generate corresponding binary
      iii. Put on Repository
3. Get necessary input files (check if they already exist on the machines)

Παρατηρήστε ότι εξαιτίας του τεράστιου μεγέθους ορισμένων αρχείων, αποφεύγονται περιττές μεταφορές ελέγχοντας τοπικά τα αρχεία όταν χρειάζεται. Αυτό ισχύει τόσο για τα δυαδικά αρχεία και τα αρχεία εισόδου.

Επιπλέον, έχει συμπεριληφθεί στη διαδικασία η λειτουργικότητα Checksum. Σε περίπτωση που το ζητούμενο αρχείο εντοπιστεί σε τοπικό επίπεδο αλλά με διαφορετική τιμή ελέγχου αθροίσματος από εκείνη που υπάρχει στο χώρο αποθήκευσης, το αρχείο θεωρείται κατεστραμμένο και απορρίπτεται. Αυτό ισχύει τόσο για τα δυαδικά αρχεία όσο και για τα αρχεία εισόδου.



Σχήμα 5.6: Τοποθέτηση του Διαχειριστή Εκτέλεσης στην αρχιτεκτονική του AEOLUS

2. Script Εκτέλεσης

Δημιουργείται επίσης δέσμη ενεργειών για την έναρξη της εκτέλεσης των συστατικών στοιχείων στην ετερογενή αρχιτεκτονική υλικού. Αυτό το σενάριο πρέπει να περιέχει τις οδηγίες ανάπτυξης σχετικά με το πού πρέπει να εκτελείται κάθε στοιχείο. Λόγω της τοποθεσίας της εφαρμογής που προωθείται από το μοντέλο προγραμματισμού, ο Διαχειριστής Εκτέλεσης θα πρέπει να καταβάλει κάθε δυνατή προσπάθεια για να εγγυηθεί για τη διασύνδεση των εξαρτημάτων λογισμικού με τα εξαρτήματα υλικού στην πλατφόρμα υλικού. Για το σκοπό αυτό, η βιβλιοθήκη OpenMPI παρέχει ειδικές λειτουργίες κατά την εκτέλεση, οι οποίες αξιοποιούνται πλήρως από την AEOLUS για τη δέσμευση συγκεκριμένων διαδικασιών σε συγκεκριμένους πυρήνες επεξεργαστών, υποδοχές κλπ.

Τα προαναφερθέντα σενάρια είναι διαθέσιμα στο χρήστη για εκτέλεση στο απομακρυσμένο μηχάνημα όπου εκτελείται ο Διαχειριστής Εκτέλεσης.

Αλληλεπίδραση με άλλα εργαλεία

Όπως συμβαίνει με όλα τα εργαλεία AEOLUS, ο Διαχειριστής Εκτέλεσης είναι μέρος της συνολικής αρχιτεκτονικής και πρέπει να επικοινωνεί με τα άλλα εργαλεία για να ολοκληρώσει τη λειτουργικότητά του. Η τοποθέτησή του στην πλατφόρμα φαίνεται στο Σχήμα 5.6.

Όπως περιγράφεται στο Σχήμα 5.6, το ίδιο το εργαλείο χρειάζεται μόνο να αλληλεπιδράσει με το Repository, αποκτώντας τα αρχεία που απαιτούνται για την ανάπτυξη. Αυτά περιλαμβάνουν:

- Η περιγραφή πλατφόρμας, όπου ορίζεται η υποδομή υλικού
- Το δίκτυο συνιστωσών, όπου περιγράφεται η δομή της εφαρμογής
- Το Σχέδιο Ανάπτυξης, όπου περιγράφεται η χαρτογράφηση που σχεδιάστηκε από τον Χάρτη Πολλαπλών Στόχων

- Τα αρχεία προέλευσης που διαμένουν στο χώρο αποθήκευσης

Οι πραγματικές αλληλεπιδράσεις που κάνει το εργαλείο με τις άλλες ενότητες προκύπτουν από τα σενάρια που δημιουργούνται και περιλαμβάνουν αρκετές αλληλεπιδράσεις με το Repository για την λήψη / τοποθέτηση δυαδικών αρχείων και αρχείων εισόδου / εξόδου.

5.6.3 Λεπτομέρειες υλοποίησης

Αναγνώριση αντικειμένων επικοινωνίας

Τα αντικείμενα επικοινωνίας που χρησιμοποιούνται από την εφαρμογή πρέπει να δηλώνονται σαν οποιαδήποτε άλλη μεταβλητή. Για μια δήλωση αντικειμενικής επικοινωνίας χρησιμοποιείται ένας σχολιασμός pragma ορίζοντας το πρωτόκολλο του αντικειμένου, το όνομά του κλπ.
Οι σχολιασμοί pragma περιγράφονται εδώ λεπτομερώς:

> *#pragma <queue | shared | signal | mutex> <in | out | inout> name maxsize=maxsize offset=offset*
> Δηλώνοντας ένα αντικείμενο χρησιμοποιώντας αυτήν την pragma, η τοπική μεταβλητή που ακολουθεί τον σχολιασμό συνδέεται με το αντικείμενο. Το μέγεθος και ο τύπος του αντικειμένου καθορίζονται από τα αντίστοιχα χαρακτηριστικά της τοπικής μεταβλητής.

Πληροφορίες που περιλαμβάνονται στη δήλωση τοπικής μεταβλητής:

- Τοπικό όνομα μεταβλητής
- Μέγεθος αντικειμένου
- Τύπος αντικειμένου

Στον σχολιασμό pragma, ορίζονται επίσης τα ακόλουθα χαρακτηριστικά:

- Όνομα αντικειμένου επικοινωνίας
- Πρωτόκολλο (ουρά ή κοινή χρήση)
- Κατεύθυνση του αντικειμένου (χρησιμοποιείται ως είσοδος, έξοδος ή και τα δύο)
- Μέγιστη ποσότητα αντικειμένων που μπορούν να εισαχθούν στην ουρά (μόνο για αντικείμενα πρωτοκόλλου ουράς αναμονής) - προαιρετικά
- Απόκλιση για την πρόσβαση συγκεκριμένων δεδομένων ενός πίνακα που είναι αποθηκευμένο στην κοινόχρηστη μνήμη. Χρήσιμο για την αποφυγή της μετακίνησης μεγάλων κομματιών δεδομένων μεταξύ των στοιχείων χωρίς πραγματικό λόγο. (μόνο για αντικείμενα κοινόχρηστου πρωτοκόλλου) - προαιρετικά

Στο ακόλουθο παράδειγμα, μερικές από τις λειτουργίες πρωτοκόλλου που ορίστηκαν στη διεπαφή προγραμματισμού χρησιμοποιούνται μεταξύ των στοιχείων CA και CB για να βοηθήσουν στην κατανόηση των προαναφερθέντων σημείων pragma. Στο περιβάλλον κάθε συστατικού γίνεται μια δήλωση αντικειμένου επικοινωνίας (pragma), στην οποία είναι ορατό ότι το όνομα είναι το ίδιο για τα αντικείμενα που αναφέρονται στο ίδιο αντικείμενο, ενώ το όνομα των τοπικών μεταβλητών δεν έχει σχέση με το άλλο. Ας εξετάσουμε το αντικείμενο που έχει δηλωθεί ως στοιχείο κοινόχρηστης μνήμης χρησιμοποιώντας το αναγνωριστικό 'the_data'. Αυτό σημαίνει ότι το όνομα 'the_data' αναφέρεται στην ίδια σειρά 1024 αντικειμένων και του τύπου uint8_t, που δηλώνονται ότι υπάρχουν στη μνήμη και μοιράζονται μεταξύ των δύο στοιχείων. Οι τοπικές μεταβλητές που χρησιμοποιούν το όνομα 'δεδομένα' είναι δύο διαφορετικές μεταβλητές που υπάρχουν σε διαφορετικά εύρη της εφαρμογής, αλλά ορίζονται για να διατηρούν το ίδιο κομμάτι δεδομένων χρησιμοποιώντας το κοινόχρηστο πρωτόκολλο.

```
// Component A

#pragma aeolus signal out ready
bool startb;

#pragma aeolus shared out the_data
uint8_t data[1024];

#pragma aeolus queue in sum
uint32_t input_sum;

void CB() {
   construct_input_data(data);
   aeolus_synchronize(data,1); // 1 is for updating the shared memory
   aeolus_signal(&startb);
   aeolus_queue_get(&input_sum);
}

// Component B

#pragma aeolus signal in ready
bool startme;

// Getting the last 128 items from the ''the_data comm. object
#pragma aeolus shared in the_data offset=896
uint8_t data[128];

#pragma aeolus queue out sum
uint32_t output_sum;

void CB() {
   while(true) {
      aeolus_wait(&startme);
      aeolus_synchronize(data,0); // 0 is for updating the local memory
      for(int i=0; i<128; i++)
         output_sum += data[i];
      aeolus_queue_put(&output_sum);
   }
}
```

Στο παραπάνω παράδειγμα, η εφαρμογή χρησιμοποιεί το κοινόχρηστο πρωτόκολλο για να μεταφέρει δεδομένα από το CA component to component CB. Η κλήση στη συνάρτηση aeolus_synchronize() με το χαρακτηριστικό κατεύθυνσης που έχει οριστεί στην τιμή '1' ενημερώνει την κοινόχρηστη μνήμη μεταξύ των δύο στοιχείων με τις τιμές της μεταβλητής δεδομένων στο στοιχείο CA, ενώ στην άλλη πλευρά το στοιχείο CB καλεί aeolus_synchronize() με το χαρακτηριστικό κατεύθυνσης που έχει οριστεί στο '0' για την ενημέρωση της τοπικής μνήμης με τις τιμές στην κοινόχρηστη μνήμη. Προκειμένου να διασφαλιστεί η συνάφεια των δεδομένων, ο προγραμματιστής χρησιμοποίησε το πρωτόκολλο σήματος για τον συντονισμό των συναλλαγών μεταξύ των στοιχείων. Το πρωτόκολλο ουράς χρησιμοποιείται επίσης για τη μετακίνηση του αποτελέσματος των υπολογισμών που γίνονται στο CB πίσω στο CA. Δεν απαιτούνται ενέργειες συγχρονισμού εδώ, καθώς το πρωτόκολλο ουράς μπορεί να εγγυηθεί για τη συνέπεια των δεδομένων.

Η ευθύνη του Διαχειριστή Εκτέλεσης κατά τη διάρκεια αυτού του σταδίου έγκειται στην ανάλυση του κώδικα εντοπισμού των προαναφερθέντων pragmas προκειμένου να ολοκληρωθεί το μοντέλο εφαρμογής σε επίπεδο κώδικα πηγής, όπως σχεδιάστηκε από τον προγραμματιστή συμπεριλαμβανομένων των στοιχείων, των αντικειμένων επικοινωνίας και του κώδικα σχετικές με αυτές, δηλαδή τις δηλώσεις pragma και τις αντίστοιχες κλήσεις API.

Κατά τη διάρκεια αυτής της διαδικασίας, οι πληροφορίες εξάγονται από τις pragmas και γίνονται ορισμένες τροποποιήσεις στις κλήσεις API. Έτσι, μια φάση προ-σύνταξης εκτελείται στον πηγαίο κώδικα κάθε στοιχείου κατά την οποία γίνονται οι ακόλουθες ενέργειες:

1. Αν εντοπιστεί μια δήλωση αντικειμένου επικοινωνίας, εξάγονται από αυτήν οι αντίστοιχες πληροφορίες και το pragma διαγράφεται. Ο Διαχειριστής Εκτέλεσης αποθηκεύει αυτές τις πληροφορίες για να τροποποιήσει ανάλογα τις κλήσεις API που μπορεί να προκύψουν μετά την πράξη. Επομένως, είναι σημαντικό η δήλωση pragma να προχωρήσει σε οποιαδήποτε χρήση των τοπικών μεταβλητών που συνδέονται με το αντικείμενο, όπως και με μια κανονική μεταβλητή δήλωση.

2. Εάν προσδιοριστεί μια κλήση API, προστίθενται συγκεκριμένα επιχειρήματα, παρέχοντας τις απαραίτητες πληροφορίες που πρόκειται να ολοκληρώσουν τη λειτουργικότητά της.

Ενσωμάτωση με τη διεπαφή προγραμματισμού

Η διεπαφή προγραμματισμού χρησιμοποιεί ένα σύνολο δομών για τον χειρισμό και τη μεταφορά των αντικειμένων επικοινωνίας. Αυτές οι δομές αντιπροσωπεύουν τα αντικείμενα επικοινωνίας σε χαμηλότερο επίπεδο για να διευκολύνουν την υλοποίηση των μεταφορών δεδομένων μεταξύ των στοιχείων. Αποθηκεύουν τις απαραίτητες πληροφορίες σε συγκεκριμένα πεδία και έχουν πρόσβαση από τα API για να ολοκληρώσουν τις λειτουργίες τους.

Τα αντικείμενα που μοιράζονται σε έναν χώρο διανεμημένης μνήμης χρησιμοποιούν τις ακόλουθες δομές:

```
// QUEUE PROTOCOL
typedef struct aeolus_queue {
        uint8_t *qdata;
        uint32_t front, rear, count;
        uint32_t disp, maxsize, owner;
        MPI_Win win;
} aeolus_queue;

// SIGNAL PROTOCOL
typedef struct {
        int sig;
        MPI_Comm comm;
} aeolus_signal;

// MUTEX PROTOCOL
typedef struct aeolus_mutex {
        int numprocs, ID, home, tag;
        MPI_Comm comm;
        MPI_Win win;
        unsigned char *waitlist;
} aeolus_mutex;
```

While objects that live in a shared memory space use the next structures:

```
// SHARED PROTOCOL
typedef struct {
        pthread_mutex_t lock;
        int size;
        void *data;
} aeolus_shared_object;

// SIGNAL PROTOCOL
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int ready;
        int source;
        int target;
} aeolus_signal;

// MUTEX PROTOCOL
```

```
typedef struct {
        pthread_mutex_t lock;
} aeolus_mutex;

// QUEUE PROTOCOL
typedef struct phnode {
        void* item;
        struct phnode *next;
} aeolusQnode;

typedef struct {
        aeolusQnode *front;
        aeolusQnode *rear;
        pthread_mutex_t lock;
        volatile int count;
} aeolus_queue;
```

Προκειμένου να παρέχεται διαφάνεια στον χρήστη καθώς και δυνατότητα διαμορφώσεως στην εφαρμογή της εφαρμογής, ο προγραμματιστής χρησιμοποιεί μια απλή διασύνδεση (βλέπε κεφάλαιο 7.1.3) για την κλήση των αντίστοιχων λειτουργιών πρωτοκόλλου. Σύμφωνα με το Σχέδιο Ανάπτυξης, οι κλήσεις API που προσδιορίζονται τροποποιούνται για να ταιριάζουν στις αντίστοιχες υλοποιήσεις που υποδεικνύονται από την επιλογή Τεχνικής. Αυτές οι εφαρμογές μπορούν να βρεθούν στο προσάρτημα 1.

Κατασκευή μιας εφαρμογής πολλαπλών διεργασιών

Η υλοποίηση της παραγόμενης εφαρμογής θα ακολουθήσει την προσέγγιση Single-Program-Multiple Data (SPMD) χρησιμοποιώντας τη βιβλιοθήκη OpenMPI. Αυτό σημαίνει ότι ένα ενιαίο πρόγραμμα είναι χτισμένο και τρέχει στην υποδομή υλικού, ακολουθώντας μια διαφορετική διαδρομή ροής ελέγχου για κάθε διαφορετικό στοιχείο της διαδικασίας. Το πρωτόκολλο MPI επιλέχθηκε για την αποτελεσματικότητά του τόσο στις αρχιτεκτονικές UMA όσο και στις αρχιτεκτονικές NUMA, λόγω των στρατηγικών της τοπικής μνήμης που ακολουθεί, καθώς και για την εφαρμογή του σε ετερογενή περιβάλλοντα.

Συγκεκριμένα, μια αρχή αρχικοποίησης είναι το σημείο εκκίνησης της εφαρμογής, όπου κατανέμονται όλες οι απαραίτητες δομές, συμπεριλαμβανομένων των αντικειμένων επικοινωνίας, συνοδευόμενες από μια λειτουργία οριστικοποίησης που ανακατεύει κάθε χρησιμοποιούμενο χώρο και τελειώνει την εκτέλεση. Ένα κοινό αρχείο κεφαλίδας χρησιμοποιείται για τη δημιουργία του περιβάλλοντος AEOLUS, κρατώντας τις πληροφορίες υψηλότερου επιπέδου που απαιτούνται για την υλοποίηση. Για την παραγωγή αυτών των αρχείων προέλευσης χρησιμοποιείται το μοντέλο εφαρμογής που έχει δημιουργηθεί στο προηγούμενο στάδιο για την παροχή των αντίστοιχων πληροφοριών (π.χ. τοπικές μεταβλητές, πληροφορίες αντικειμένου επικοινωνίας κ.λπ.).

Ένα παράδειγμα της συνάρτησης αρχικοποίησης εμφανίζεται στο ακόλουθο τμήμα:

```
#include "aeolus.h"

void invoke_component(int id) {
    if(id == 0) {
        CA();
    }
    else if(id == 1) {
        CB();
    }
    return;
}

void initialize(int *argc, char*** argv) {
        MPI_Init(argc, argv);
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
17          buf = (int *)malloc(10*sizeof(int));
18          MPI_Win_create(buf, 10*sizeof(int), sizeof(int), MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
19          signal = aeolus_signal_init(MPI_COMM_WORLD);
20          aeolus_mutex_init(&mutex0,0);
21  }
22
23  void finalize() {
24      aeolus_mutex_destroy(mutex0);
25      MPI_Win_free(&win);
26      MPI_Finalize();
27  }
28
29  int main(int argc, char** argv) {
30      initialize(&argc,&argv);
31      invoke_component(world_rank);
32      finalize();
33      return 0;
34  }
```

Όπως απεικονίζεται παραπάνω, ένα σύνολο λειτουργιών που δημιουργούνται από το Διαχειριστή Εκτέλεσης ρυθμίζει το περιβάλλον AEOLUS διαθέτοντας χώρο για τις απαραίτητες δομές και προετοιμάζοντας τις αντίστοιχες τιμές / αντικείμενα, συντονίζει την εκτέλεση και τελικά καθαρίζει το περιβάλλον.

Οι μεταβλητές που χρησιμοποιούνται για την υλοποίηση των αντικειμένων επικοινωνίας στο παραπάνω τμήμα δηλώνονται στο 'aeolus.h', το οποίο εμφανίζεται παρακάτω:

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "signalAPI.h"
5  #include <string.h>
6  #include "mutexAPI.h"
7  #include "sharedAPI.h"
8  #include "queueAPI.h"
9
10 MPI_Win win;
11 aeolus_mutex *mutex0;
12 aeolus_signal *signal;
13 int *buf;
14 int world_rank, world_size;
15
16 void CA();
17 void CB();
```

Παρατηρήστε ότι ο τύπος των πληροφοριών που αποθηκεύονται στις δηλωμένες δομές διαφέρει από την εφαρμογή στην υλοποίηση. Για την υλοποίηση ενός κοινόχρηστου χώρου μνήμης, η βιβλιοθήκη POSIX χρησιμοποιείται για την υλοποίηση του API. Σε αυτήν την περίπτωση, το αρχείο 'aeolus.c' μοιάζει με αυτό:

```
1  void aeolus_init() {
2          int i, sig=0, que=0, shr=0;
3          phsignal = (aeolus_signal **) malloc(NUM_SIGNALS*sizeof(aeolus_signal *))
       ;
4          phqueue = (aeolus_queue **) malloc(NUM_QUEUES*sizeof(aeolus_queue *));
5          phshared = (aeolus_shared_object **) malloc(NUM_SHARED*sizeof(
       aeolus_shared_object *));
6
7          signal_index = (int *) malloc(AEOLUS_NUMOFCOMMS*sizeof(int));
8          queue_index = (int *)malloc(AEOLUS_NUMOFCOMMS*sizeof(int));
9          shared_index = (int *) malloc(AEOLUS_NUMOFCOMMS*sizeof(int));
10
11         for(i=0; i<AEOLUS_NUMOFCOMMS; i++) {
```

```
12                  if(commobj_type[i] == SIGNAL) {
13                          phsignal[sig] = initialize_aeolus_signal(aeolus_source_id
    [i], aeolus_target_id[i]);
14                          signal_index[i] = sig++;
15                  }
16                  else if(commobj_type[i] == QUEUE) {
17                          phqueue[que] = initialize_aeolus_queue();
18                          queue_index[i] = que++;
19                  }
20                  else if(commobj_type[i] == SHARED) {
21                          phshared[shr] = initialize_aeolus_shared_object(
    commobj_datasize[shr], commobj_size[shr]);
22                          shared_index[i] = shr++;
23                  }
24          }
25  }
```

Οι πιο σύνθετες καταστάσεις μπορούν να προκύψουν όπως περιγράφηκε προηγουμένως, όπου η εφαρμογή μπορεί να περιλαμβάνει συνδυασμό των δύο προσεγγίσεων, την ανάπτυξη των στοιχείων τόσο στις διαδικασίες MPI όσο και στα ξεχωριστά θέματα, βελτιστοποιώντας την ανάπτυξη ανάλογα με το σχεδιασμό της εφαρμογής καθώς και τις απαιτήσεις την εκτέλεσή του.

Δημιουργία και ανάπτυξη δυαδικών αρχείων

Τα σενάρια που δημιουργούνται προετοιμάζουν τους τοπικούς καταλόγους του υλικού για ανάπτυξη. Τα δυαδικά αρχεία ή τα αρχεία προέλευσης μεταφορτώνονται (ανάλογα με τα προηγούμενα χρησιμοποιούμενα δυαδικά αρχεία για να αποφευχθεί η ανασυγκρότηση του ίδιου κώδικα), να συγκεντρωθούν και να τοποθετηθούν στις αντίστοιχες θέσεις των μηχανών-στόχων. Εάν δημιουργούνται νέες εκδόσεις των δυαδικών αρχείων, μεταφορτώνονται στο χώρο αποθήκευσης για μελλοντική επαναχρησιμοποίηση. Το Makefile, που βρίσκεται στον κατάλογο 'src' του Repository, τροποποιείται επίσης για να συμπεριλάβει τα αντίστοιχα περιτυλίγματα μεταγλωττιστή (nvcc, mpicc), τις απαραίτητες βιβλιοθήκες (OpenMP, pthreads) και τα τροποποιημένα αρχεία προέλευσης. Παρακάτω παρουσιάζεται ένα παράδειγμα της δέσμης σύνταξης:

```
1
2  PROJECT_NAME="test_project";
3  LOCAL_DIR="~/AEOLUS/Repository/${PROJECT_NAME}/DM";
4  BIN_PATH_ON_REPO="bin";
5  BIN_NAME="myapp-bin20181010";
6  FILE_PATH="${LOCAL_DIR}/${BIN_PATH_ON_REPO}/${BIN_NAME}";
7
8  cd ~/AEOLUS;
9
10 # CHECK FOR CORRESPONDING BINARY
11 if [ -f $FILE_PATH ]
12 then
13         echo "File exists"
14 else
15         # TRY GETTING BINARY FROM REPO
16         response_code=`Get_file.sh ${PROJECT_NAME} ${BIN_PATH_ON_REPO}/${BIN_NAME
    }`;
17         if [ ! "${response_code}" -eq 200 ]
18         then
19                 # GET SOURCE FILES FROM REPO
20                 ~/AEOLUS/Repository/scripts/Get_dir.sh ${PROJECT_NAME} src;
21                 rm src.zip;
22                 # GENERATE BINARIES
23                 cd ${LOCAL_DIR}/src;
24                 make
25                 mv ${BIN_NAME} ${LOCAL_DIR}/${BIN_PATH_ON_REPO}
26                 cd ~/AEOLUS
```

```
                # PUT BINARY ON REPO
                Put_file.sh ${FILE_PATH} ${BIN_PATH_ON_REPO};
        fi;
fi;

INPUT_PATH_ON_REPO="inputs";
INPUT_NAME="input_data.txt";
INPUT_FILE="${LOCAL_DIR}/${INPUT_PATH_ON_REPO}/${INPUT_NAME}";

# PLACE INPUT FILES AT THE CORRESPONDING LOCATIONS
# CHECK FOR CORRESPONDING FILE
if [ -f $INPUT_FILE ]
then
        echo "File exists"
else
        # GET FILE FROM REPO
        response_code=`Get_file.sh ${PROJECT_NAME} ${INPUT_PATH_ON_REPO}/${
    INPUT_FILE}`;
fi;
```

Το Script Εκτέλεσης είναι υπεύθυνο για την πραγματική εκτέλεση των στοιχείων στις αντίστοιχες θέσεις. Αυτή η δέσμη ενεργειών έχει σχεδιαστεί σύμφωνα με τη χαρτογράφηση που περιγράφεται στο Σχέδιο ανάπτυξης και την Περιγραφή πλατφόρμας που παρέχει τη θέση δικτύου κάθε μηχανής ή συσκευής. Οι μηχανές έχουν πρόσβαση μέσω SSH από το μηχάνημα ανάπτυξης, όπου εκτελείται ο Διαχειριστής Εκτέλεσης. Επομένως, θεωρούνται απαραίτητες διαμορφώσεις για την επιτυχή ανάπτυξη της εφαρμογής.

```
# EXECUTE PROGRAM
# mpirun <affinity options>
# --host <IP> -np <number of processes> <executable_name> :
# .
# .
# .
# --host <IP> -np <number of processes> <executable_name>

# example: 2 components executed on different machines
mpirun -nolocal --host 192.282.77.22 -np 1 executable :
                --host 192.151.23.34 -np 1 executable
```

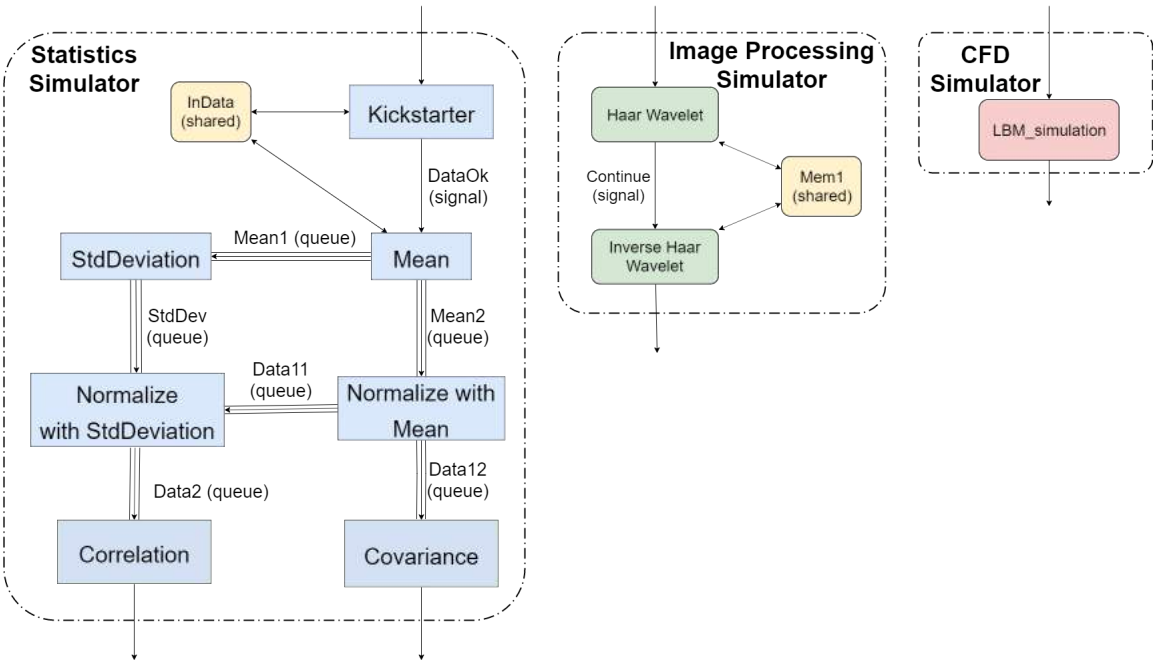
Το OpenMPI παρέχει μια λίστα επιλογών που επιτρέπουν την εκτέλεση συνθέσεων συγγένειας. Με τη δέσμευση συγκεκριμένων πόρων σε συγκεκριμένες διεργασίες, υλοποιείται η ανάπτυξη που προτείνεται στο Deployment Plan, ενώ επιπλέον βελτιστοποιήσεις τοπικών χώρων επιτρέπουν περαιτέρω βελτιώσεις στην εκμετάλλευση των πόρων μνήμης.

5.7 Αξιολόγηση

Η αξιολόγηση της αλυσίδας εργαλείων θα βασιστεί σε μια πραγματική περίπτωση χρήσης που ενοποιεί χαρακτηριστικά από διαφορετικά επιστημονικά πεδία για την επικύρωση των αποτελεσμάτων. Σε αυτή την ενότητα περιγράφουμε αυτήν την εφαρμογή και πώς τα εξαρτήματά της είναι κατάλληλα για τη δοκιμή των διαφόρων εργαλείων. Επιπλέον, παρέχουμε μια επισκόπηση των δοκιμών που εκτελέσαμε προκειμένου να αξιολογήσουμε την υποδομή και την προκύπτουσα παράλληλη εφαρμογή από την άποψη μιας σειράς λειτουργικών και μη λειτουργικών απαιτήσεων.

5.7.1 Περιγραφή περίπτωσης χρήσης

Η εφαρμογή που χρησιμοποιήθηκε για την αξιολόγηση του τελικού αποτελέσματος της αλυσίδας εργαλείων περιγράφεται εδώ. Τρεις διαφορετικοί προσομοιωτές θεωρούνται ότι καλύπτουν ένα ευρύ φάσμα χαρακτηριστικών που μπορούν να βρεθούν σε πραγματικές και υπολογιστικά εντατικές εφαρμογές. Αυτοί οι προσομοιωτές περιέχουν υπολογισμούς από τα πεδία Στατιστικής, Ανάλυσης εικόνας



Σχήμα 5.7: Αξιολόγηση μοντέλου εφαρμογής περίπτωσης χρήσης

και δυναμικής ελέγχου ροής (CFD). Συγκεκριμένα, τα στοιχεία και η μεταξύ τους επικοινωνία παρουσιάζονται λεπτομερώς στο Σχήμα 5.7, όπου ορίζεται το μοντέλο εφαρμογής (Network Component) της περίπτωσης χρήσης.

Συγκεκριμένα, ο Στατιστικός Προσομοιωτής έχει σχεδιαστεί για να περιλαμβάνει πολλά μικρότερα στοιχεία που ανταλλάσσουν εκτενώς πληροφορίες χρησιμοποιώντας τα κοινόχρηστα, ουρά και πρωτόκολλα σημάτων όπως ορίζονται και εφαρμόζονται στο πλαίσιο του μοντέλου προγραμματισμού. Ο προσομοιωτής ανάλυσης εικόνων αποτελείται από δύο στοιχεία υπολογιστικής έντασης που χρησιμοποιούν επίσης ένα υποσύνολο της διασύνδεσης προγραμματισμού για την ανταλλαγή δεδομένων. Ο προσομοιωτής CFD περιλαμβάνει ένα μόνο στοιχείο το οποίο υλοποιεί έναν υπολογιστικό CFD υπολογισμό, ο οποίος παρέχει δύο διαφορετικές εκδόσεις, μία CPU μόνο και μία CPU-GPU.

Το Component Network έχει σχεδιαστεί με αυτόν τον τρόπο για να παρέχει μια ποικιλία χαρακτηριστικών που θεωρούνται σημαντικά για την εξέταση των διαφορετικών χαρακτηριστικών των εργαλείων. Για παράδειγμα, με τον Στατιστικό προσομοιωτή αξιολογούμε την πολυπλοκότητα που μπορεί να υποστηριχθεί από το μοντέλο προγραμματισμού καθώς και τη λειτουργικότητα των λειτουργιών πρωτοκόλλου προγραμματισμού που χρησιμοποιούνται εκτενώς σε αυτό το μέρος. Οι άλλοι δύο προσομοιωτές επικεντρώνονται στο φορτίο υπολογισμού που δημιουργείται κατά την εκτέλεση των στοιχείων τους και χρησιμοποιούνται για την απόκτηση μετρήσεων από τις παράλληλες παραλλαγές του OpenMP που παράγονται από το εργαλείο αυτόματης παραμετροποίησης. Για την περίπτωση του CFD Simulator, η έκδοση CPU-GPU έχει επίσης συμπεριληφθεί στη διαδικασία για να δοκιμάσει την ολοκλήρωση της πλατφόρμας με παράλληλες εκδόσεις εκτός από το OpenMP.

Ανάπτυξη με χρήση του AEOLUS

Η βελτιστοποίηση της εφαρμογής, από την άποψη του χρόνου εκτέλεσης (ή άλλων στόχων που δεν μελετήσαμε εδώ - π.χ. αποδοτικότητα ισχύος), είναι ο προφανής στόχος ενός προγραμματιστή που χρησιμοποιεί μια παράλληλη υποδομή. Στο πλαίσιο αυτής της εργασίας, ωστόσο, εστιάζουμε στην αρχιτεκτονική του συστήματος και την προκύπτουσα εφαρμογή αφού χρησιμοποιήσουμε τα καθορισμένα χαρακτηριστικά του πλαισίου. Οι κύριοι δείκτες που εξετάσαμε για την αξιολόγηση της σειράς εργαλείων περιλαμβάνουν τη σωστή λειτουργικότητα της προκύπτουσας εφαρμογής και την ικανότητά της να προσαρμόζεται σε μια σειρά διαφορετικών αρχιτεκτονικών. Επιπρόσθετα, με-

τρήσαμε την ατομική απόδοση των τριών προσομοιωτών προκειμένου να αξιολογήσουμε τον τρόπο βελτιστοποίησης αυτών και να υπολογίσουμε τυχόν overhead που μπορεί να εισάγεται από την υποδομή.

Λειτουργική δοκιμή

Για τη λειτουργική δοκιμή της εφαρμογής, εξετάσαμε μια σειρά από διαφορετικές δοκιμές (αντιστοιχίσεις components σε διαφορετικές διεργασίες) για να δοκιμάσουμε τις διάφορες υλοποιήσεις ανάπτυξης και επικοινωνίας. Συγκεκριμένα, χρησιμοποιήσαμε το ακόλουθο σύνολο αντιστοιχιών για την ανάπτυξη του Στατιστικού Προσομοιωτή:

Δοκιμή 1:

Διεργασία 0: Όλα τα components

Δοκιμή 2:

Διεργασία 0: Kickstarter, Mean, Normalize_with_mean, Covariance

Διεργασία 1: StdDeviation, Normalize_with_stddev, Correlation

Δοκιμή 3:

Διεργασία 0: Kickstarter, Mean

Διεργασία 1: StdDeviation, Normalize_with_mean, Normalize_with_stddev

Διεργασία 2: Covariance

Διεργασία 3: Correlation

Ανάλογα με το διαθέσιμο κοινόχρηστο περιβάλλον, χρησιμοποιήθηκαν οι αντίστοιχες υλοποιήσεις επικοινωνίας για την ανταλλαγή δεδομένων και σημάτων, ενώ τα συστατικά έχουν αρχίσει ως θέματα ή διεργασίες ανάλογα. Αξιολόγηση απόδοσης

Η AEOLUS δοκιμάστηκε για τα κέρδη απόδοσης σε δύο από τους προσομοιωτές. Τα προγράμματα δοκιμάστηκαν και επικυρώθηκαν με τη συμβολή του εργαλείου χρησιμοποιώντας μια Intel i7 CPU 12 νημάτων. Για τις δοκιμές, μια σειρά μικρών, μεσαίων και μεγάλων συνόλων δεδομένων εισόδου, όπου χρησιμοποιείται η εκμετάλλευση διαφορετικών ποσοτήτων διαθέσιμων κλωστών με αποτέλεσμα μικρά και μεγάλα φορτία υπολογισμού. Η σειριακή έκδοση της εφαρμογής θεωρήθηκε ως η γραμμή βάσης, ενώ η βελτίωση έχει υπολογιστεί σε σύγκριση με τον πολυνηματικό παράλληλο κώδικα ως:

$$PerformanceImprovement = \frac{serial - parallel}{serial}\ 100\% \tag{5.1}$$

Σε γενικές γραμμές, ο προκύπτων παράλληλος κώδικας φάνηκε να προσθέτει κάποια επιβάρυνση στην εφαρμογή, έτσι όταν το φορτίο υπολογισμού ήταν σχετικά μικρό, το γενικό ήταν εμφανές, ενώ για μεγαλύτερα φορτία υπολογισμού η επιτάχυνση της εφαρμογής υπερνικά σημαντικά τα γενικά έξοδα, επιτυγχάνοντας θετικά αποτελέσματα σε 84 %. Τα συγκεκριμένα αποτελέσματα που παρουσιάζουν βελτίωση παρουσιάζονται στον Πίνακα 5.1.

Μετροπρόγραμμα Surveillance			
Μέγεθος εισόδου \ # νημάτων	**3 νήματα**	**6 νήματα**	**12 νήματα**
Μικρό	-10%	-19%	-22%
Μεσαίο	57%	70%	13%
Μεγάλο	62%	80%	84%
Μετροπρόγραμμα CFD			
Μέγεθος εισόδου \ # νημάτων	**3 νήματα**	**6 νήματα**	**12 νήματα**
Μικρό	-18%	-11%	26%
Μεσαίο	-8%	4%	29%
Μεγάλο	59%	8%	30%

Πίνακας 5.1: Αξιολόγηση των αποτελεσμάτων για τα μετροπρογράμματα Surveillance και CFD

Κεφάλαιο 6

Επίλογος

Σε αυτό το κεφάλαιο, θα προσπαθήσουμε να συνοψίσουμε το έργο που έχει γίνει στο πλαίσιο αυτής της διατριβής, μαζί με τα συμπεράσματα που έγιναν κατά το σχεδιασμό της αρχιτεκτονικής και την ανάπτυξη των διαφόρων τμημάτων της εργαλειομηχανής που περιγράφηκε στο τα προηγούμενα κεφάλαια. Θα συμπεριλάβουμε επίσης μερικές από τις ιδέες που δημιουργήθηκαν από τις ανάγκες ενός παράλληλου πλαισίου εφαρμογής, όπως αυτό που περιγράφεται σε αυτή τη διατριβή, καθώς και ορισμένες επεκτάσεις του εργαλείου που θα μπορούσαν να αποδειχθούν πολύ χρήσιμες για τη διαδικασία ανάπτυξης και εκτέλεσης εφαρμογών.

6.1 Συμπέρασμα

Οι κύριοι πυλώνες αυτής της εργασίας μπορούν να περιγραφούν εν συντομία σε τρία διαφορετικά μέρη. Για το πρώτο μέρος, μελετήσαμε μερικά από τα πλέον σύγχρονα εργαλεία και τεχνικές που χρησιμοποιούνται για την αυτόματη παραλληλοποίηση κώδικα. Για το δεύτερο μέρος σχεδιάσαμε ένα μοντέλο παράλληλου προγραμματισμού που επιτρέπει την ανάπτυξη μιας παράλληλης εφαρμογής. Στο τρίτο και τελευταίο μέρος της διατριβής αναπτύξαμε την προκύπτουσα σειρά εργαλείων από τα δύο προηγούμενα μέρη που αποτελούν μια υποδομή για την ανάπτυξη μιας παράλληλης εφαρμογής από την ανάπτυξή της μέχρι την αυτόματη παραλληλοποίηση της με την ολοκλήρωση και την υλοποίησή της.

6.1.1 Αυτόματη Παραλληλοποίηση

Κατά τη διάρκεια της μελέτης μας για τα εργαλεία αυτόματης παραλληλοποίησης, εξετάσαμε δύο διαφορετικά εργαλεία, το ROSE Compiler και το PLuTo, ένα αυτόματο παραλληλιστή. Αναλύσαμε τις διαφορές μεταξύ των δύο εργαλείων και εκτελέσαμε μια διαδικασία δοκιμής μεταξύ τους για να συγκρίνουμε την αποδοτικότητά τους. Για το σκοπό αυτό, χρησιμοποιήσαμε το Polybench, μια πλούσια σουίτα συγκριτικής αξιολόγησης που περιλαμβάνει μια σειρά από παραδείγματα τμημάτων κώδικα κατάλληλα για παραλληλισμό. Η ανάλυση των εργαλείων έδειξε ένα σαφές πλεονέκτημα για τον παραλληλιστή PLuTo, ο οποίος χρησιμοποιεί αποτελεσματικούς μετασχηματισμούς κώδικα όχι μόνο για να παραλληλοποιήσει περισσότερα τμήματα κώδικα αλλά και για να βελτιστοποιήσει την πρόσβαση στη μνήμη για να ελαχιστοποιήσει τις λειτουργίες ανάγνωσης και εγγραφής. Εντούτοις, μια αξιοσημείωτη παρατήρηση είναι ότι, αν και το PLuTo παρουσίασε καλύτερα αποτελέσματα από το ROSE γενικά, το τελευταίο αποδείχθηκε ότι υποστηρίζει έναν μεγαλύτερο εύρος εφαρμογών (δεδομένου ότι είναι ένα εργαλείο συμβατό με τη C ++), ενώ παράλληλα αποδίδει καλύτερα από το πρώτο σε ορισμένες περιπτώσεις, οπότε το κέρδος που επιτυγχάνεται με τη χρήση του PLuTo αντί του ROSE δεν είναι τόσο μεγάλο.

Για να επεκτείνουμε τις δοκιμές μας, χρησιμοποιήσαμε μια σειρά πιο περίπλοκων παραδειγμάτων που εξάγονται από εφαρμογές πραγματικού κόσμου για την επέκταση της σουίτας μετροπρογραμμάτων. Από την άποψη αυτή, θεωρήσαμε χρήσιμο να προτείνουμε ένα εκτεταμένο μοντέλο προγραμματισμού που θα επέτρεπε στον χρήστη να εντοπίσει περιοχές στον κώδικα που μπορεί να είναι παράλληλες, αλλά λόγω ειδικών χαρακτηριστικών ορισμένες εξαρτήσεις δεν μπορούν να επιλυθούν από τον

παραλληλιστή. Το μοντέλο απέδειξε ότι επιλύει ορισμένες εξαρτήσεις που ήταν αδύνατον να βρεθούν χρησιμοποιώντας τα εργαλεία αυτόματης παραλληλοποίησης.

6.1.2 Παράλληλη Προγραμματική Μοντελοποίηση

Για το σχεδιασμό ενός μοντέλου παράλληλου προγραμματισμού, πρέπει πρώτα να εξετάσουμε τη γενική προσέγγιση που ακολουθείται κατά το σχεδιασμό μιας παράλληλης εφαρμογής. Με αυτό τον τρόπο, περιγράφουμε στην πραγματικότητα τι γίνεται από το backend που θα υποστήριζε ένα μοντέλο προγραμματισμού. Αυτή η ενότητα είναι αφιερωμένη στις λεπτομέρειες σχεδιασμού και υλοποίησης ενός τέτοιου μοντέλου. Το μοντέλο προγραμματισμού που περιγράφουμε βασίζεται στο σχεδιασμό της εφαρμογής ακολουθώντας μια διαμοιρασμένη δομή των στοιχείων του λογισμικού που επικοινωνούν μεταξύ τους χρησιμοποιώντας ένα καθορισμένο σύνολο API που αφαιρούν την εφαρμογή τους από το χρήστη χαμηλού επιπέδου. Αυτή η προσέγγιση επιτρέπει την ανάπτυξη των εξαρτημάτων σε διάφορα μηχανήματα / στοιχεία υλικού που βελτιστοποιούν την απόδοση ανάλογα με τα διαφορετικά χαρακτηριστικά τους.

6.1.3 Η υποδομή AEOLUS

Η υποδομή AEOLUS είναι το πραγματικό προϊόν αυτής της εργασίας. Συνδυάζει το αποτέλεσμα από τα άλλα δύο μέρη για να διαμορφώσει μια σειρά εργαλείων που οι προγραμματιστές μπορούν να αξιοποιήσουν για να σχεδιάσουν και να βελτιστοποιήσουν την εφαρμογή τους ακολουθώντας τις οδηγίες που καθορίζονται από το πρότυπο προγραμματισμού. Περιλαμβάνει τη συνολική αρχιτεκτονική του πλαισίου και τις αλληλεπιδράσεις μεταξύ των εργαλείων, καθώς και το σχεδιασμό και την υλοποίησή τους.

Η υποδομή κατασκευάζεται με βάση το πρότυπο προγραμματισμού. Ο προγραμματιστής ξεκινά την υλοποίηση σχεδιάζοντας το Component Network (το οποίο περιγράφει την εφαρμογή), με βάση την προσέγγιση που βασίζεται στις συνιστώσες που περιγράφεται και την Περιγραφή πλατφόρμας (η οποία περιγράφει το υποκείμενο υλικό που είναι διαθέσιμο). Συνδυάζοντας αυτά, μπορούν να επιλέξουν μια συγκεκριμένη ανάπτυξη και να το απεικονίσουν σε ένα Σχέδιο Ανάπτυξης. Ο κώδικας μπορεί να ρυθμιστεί σύμφωνα με τις απαιτήσεις της εφαρμογής. Από εκεί, ένα αυτόματο εργαλείο παραλληλισμού μπορεί να αναλάβει την παραγωγή παράλληλων εκδόσεων των συστατικών (χρησιμοποιώντας OpenMP, CUDA ή άλλες βιβλιοθήκες) για εκτέλεση σε πολυνηματικές CPU ή άλλο υλικό. Στη συνέχεια, μια ενότητα επιλογής τεχνικών θα επιλέξει τις αντίστοιχες υλοποιήσεις API και τις εκδόσεις εξαρτημάτων σύμφωνα με το Σχέδιο Ανάπτυξης, το οποίο στη συνέχεια αναπτύσσεται από τον Διαχειριστή Εκτέλεσης, ο οποίος ενσωματώνει, καταρτίζει και εκτελεί την εφαρμογή στους αντίστοιχους στόχους του μηχανήματος.

Σκοπός αυτής της μελέτης είναι να παράσχει το σχεδιασμό μιας ολοκληρωμένης λύσης για την ανάπτυξη παράλληλων εφαρμογών για μια υποδομή που μπορεί να αναπτυχθεί, ενώ αναπτύσσονται διαφορετικές τεχνολογίες και μοντέλα. Ορισμένα από τα εργαλεία μπορούν να αντικατασταθούν, άλλα μπορούν να βελτιωθούν, ενώ η συνολική αρχιτεκτονική μπορεί να επεκταθεί, εισάγοντας νέα εργαλεία στην υποδομή που μπορούν να προσφέρουν μια ευρύτερη δέσμη υποστηριζόμενων περιπτώσεων χρήσης.

6.2 Μελλοντική δουλέια

Κατά την ανάπτυξη και αξιολόγηση της προκύπτουσας σειράς εργαλείων, υπήρξαν πολλές ιδέες που προέκυψαν, ιδέες που θα διευκόλυναν την εμπειρία του προγραμματιστή, καθώς και την επέκταση των δυνατοτήτων που προσφέρουν τα εργαλεία, βελτιώνοντας παράλληλα την απόδοση της παράλληλης εφαρμογής. Σε αυτό το τελευταίο τμήμα, θέτουμε ορισμένες από αυτές τις ιδέες που παρέχουν κίνητρο για την περαιτέρω επέκταση της δουλειάς μας στην υποδομή, το οποίο έχει δυνατότητες για ορισμένες πολύ ελπιδοφόρες και ενδιαφέρουσες βελτιώσεις.

6.2.1 Αυτόματη Παραλληλοποίηση

Οι προσπάθειες για αυτή τη διατριβή επικεντρώθηκαν στην αυτόματη παραλληλοποίηση σε μια κλίμακα πολλαπλών θορύβων χρησιμοποιώντας το μοντέλο προγραμματισμού OpenMP. Παρόλο που τα διαθέσιμα εργαλεία δείχνουν κυρίως πολλά υποσχόμενα αποτελέσματα χρησιμοποιώντας αυτό το μοντέλο, υπάρχουν πολλές συνεχιζόμενες εργασίες με άλλες πολυπολικές τεχνολογίες ή ακόμα και τεχνολογίες για εξωτερική εκμετάλλευση υλικού παρόμοια με CUDA και OpenCL. Με το πέρασμα του χρόνου, εμφανίζονται πολλές νέες αναδυόμενες τεχνολογίες, τόσο για το τμήμα προγραμματισμού όσο και για το υλικό. Νέες συσκευές και νέες διεπαφές πρόκειται να χρησιμοποιηθούν για τη συνεχή επιτάχυνση εφαρμογών και υπολογισμών εκφόρτωσης που μπορούν να προσαρμοστούν σε συγκεκριμένο υλικό.

Προκειμένου να ενσωματωθούν και να αξιοποιηθούν όλες αυτές οι διαφορετικές λύσεις, το Μοντέλο Προγραμματισμού (που περιγράφεται στην ενότητα 4) έχει σχεδιαστεί ειδικά για να υποστηρίζει διαφορετικές τεχνολογίες και συσκευές, επιτρέποντας τη διεξαγωγή ποικίλων εφαρμογών και υλοποιήσεων, ώστε ο προγραμματιστής να μπορεί να δοκιμάσει διαφορετικές λύσεις βελτιστοποιώντας την απόδοση της εφαρμογής. Καθώς εισάγονται νέα εργαλεία αυτόματης παραλληλοποίησης, μπορούν να δημιουργηθούν και να εξεταστούν διαφορετικές εκδόσεις των στοιχείων λογισμικού για ανάπτυξη.

Για το σκοπό αυτό, το εργαλείο αυτόματης παραλληλοποίησης μπορεί να επεκταθεί για να υποστηρίξει περισσότερες τεχνολογίες παραλληλισμού για κοινή και κατανεμημένη μνήμη ή για εξωτερικές συσκευές / υλικό. Ομοίως, η ενότητα επιλογής τεχνικής μπορεί να επεκταθεί για να επιλέξει διαφορετικές εκδόσεις των εξαρτημάτων για τις ίδιες ή διαφορετικές συσκευές / μηχανές, καθώς και διαφορετικές βιβλιοθήκες επικοινωνίας για μεταφορά δεδομένων και σήματα συγχρονισμού, σύμφωνα με την επιλεγμένο τρόπο εκτέλεσης.

6.2.2 Βελτιστοποίηση διανομής πόρων

Όπως έχει ήδη αναφερθεί, το AEOLUS επιτρέπει την ανάπτυξη των εξαρτημάτων της συγκεκριμένης εφαρμογής λογισμικού σε διαφορετικά στοιχεία υλικού, έτσι ώστε ο προγραμματιστής να μπορεί να δοκιμάσει διαφορετικές υλοποιήσεις της εφαρμογής τους με στόχο τη βελτιστοποίηση της απόδοσης. Ωστόσο, αυτές οι διαφορετικές εκτελέσεις δοκιμών βασίζονται πολύ στη διαίσθηση και στο πόσο ο προγραμματιστής εξοικειώνεται με το υποκείμενο υλικό. Για να παρατηρήσουν κάποια βελτίωση, οι προγραμματιστές πρέπει είτε να είναι αρκετά τυχεροί ώστε να δοκιμάσουν κάποιο μοντέλο εκτέλεσης που να δείχνει κέρδη απόδοσης, είτε πρέπει πραγματικά να γνωρίζουν τι κάνουν (υποθέτοντας βεβαίως μια μεγάλη εφαρμογή πολλαπλών στοιχείων με μεγάλο αριθμό πιθανών μοντέλων εκτέλεσης).

Ένα εργαλείο διανομής πόρων μπορεί να κάνει ακριβώς αυτό, μπορεί να αναπτυχθεί για να χρησιμοποιήσει τη διαίσθηση των εμπειρογνωμόνων, η οποία δεν είναι κάτι που ο μέσος χρήστης κατέχει, να αποδίδει με έμφαση τα διάφορα συστατικά σε συγκεκριμένα στοιχεία υλικού. Ανάλογα με τα χαρακτηριστικά των εξαρτημάτων και τα διαφορετικά κριτήρια / απαιτήσεις του χρήστη (ταχύτητα, ισχύς κ.λπ.), το εργαλείο μπορεί να προτείνει ή να δοκιμάσει αυτόματα διαφορετικές εφαρμογές που μπορούν να είναι αρκετά ακριβείς και να βελτιστοποιήσουν με τον καιρό την τελική υλοποίηση η εφαρμογή.

6.2.3 Αποθετήριο αρχείων

Αν και η τρέχουσα αρχιτεκτονική του πλαισίου επιτρέπει την αλληλεπίδραση μεταξύ διαφορετικών μηχανών σε σχέση με το χρήστη και την ανταλλαγή σημάτων, ο χρήστης πρέπει να έχει πλήρη επίγνωση του πού να τοποθετήσει τα απαραίτητα αρχεία που χρησιμοποιούνται από τα εξαρτήματα της εφαρμογής.

Δύο ενδεικτικές προσεγγίσεις σχετικά με την υλοποίηση περιγράφονται εδώ:

- Ένας κεντρικός διακομιστής αποθετηρίων: Ένας απομακρυσμένος διακομιστής που χρησιμοποιείται για αποθήκευση αρχείων. Αυτή η λύση παρέχει ένα σύνολο API που επιτρέπουν στην εφαρμογή να αποκτά πρόσβαση εξ αποστάσεως στο διακομιστή και να διαβάζει / μεταφορτώνει / ανεβάζει τα αρχεία που χρειάζονται.

- Ένα κατανεμημένο αποθετήριο: Ένα σύνολο API που μπορεί να παρέχει πρόσβαση στα αρχεία που διαμένουν σε διαφορετικά μηχανήματα. Ο χώρος αποθήκευσης σε αυτήν την περίπτωση διανέμεται σε όλες τις διαφορετικές συσκευές / μηχανές που ορίζονται στην Περιγραφή πλατφόρμας. Το Μοντέλο Προγραμματισμού είναι υπεύθυνο για την αφαίρεση του χρήστη από οποιεσδήποτε μεταφορές / προσπελάσεις χαμηλού επιπέδου μεταξύ των διαφορετικών μηχανών, παρέχοντας την εικόνα ενός ενιαίου ομοιόμορφου αποθετηρίου με τον οποίο μπορεί να έχει πρόσβαση χρησιμοποιώντας την ίδια διεπαφή με αυτή του κεντρικού αποθετηρίου.

6.2.4 Αξιολόγηση απόδοσης

Για το σχεδιασμό του μοντέλου προγραμματισμού, εξετάσαμε την ανάγκη του χρήστη να παρακολουθεί την εφαρμογή του, όσον αφορά διάφορες πτυχές της απόδοσης (χρόνος εκτέλεσης, ισχύς κ.λπ.). Για το λόγο αυτό έχουν σχεδιαστεί τα αντίστοιχα API που δίνουν τη δυνατότητα στο χρήστη να συλλέγει πληροφορίες σχετικά με την εκτέλεση και να το αποθηκεύει όπου κι αν προτιμά (αρχείο, εκτύπωση στην έξοδο κ.λπ.). Οι αντίστοιχες υλοποιήσεις αυτών των λειτουργιών μπορούν να επεκταθούν σε διαφορετικά είδη συσκευών και υλικού για τη μέτρηση διαφορετικών παραμέτρων για να διευκολυνθεί η αξιολόγηση της απόδοσης της εφαρμογής σε κάθε ανάπτυξη.

Οι πληροφορίες που συλλέγονται από τις διάφορες εκτελέσεις μπορούν να παράσχουν πολύ χρήσιμες πληροφορίες για εργαλεία όπως το εργαλείο διανομής πόρων που αναφέραμε προηγουμένως και για τον χρήστη επίσης. Έτσι, η οργανωμένη αποθήκευση και συνάθροιση αυτών των δεδομένων είναι μεγάλη υπόθεση. Τα προαναφερθέντα API παρακολούθησης μπορούν να επεκταθούν ώστε να παρέχουν αυτά τα δεδομένα αυτόματα σε μια πλατφόρμα παρακολούθησης, η οποία με τη σειρά της μπορεί να παρέχει δυνατότητες αποθήκευσης, ανάλυσης και απεικόνισης, διευκολύνοντας τη συνολική αξιολόγηση της εφαρμογής.

6.2.5 Διεπαφή διαχείρισης

Η διαχείριση και η εκτέλεση των εργαλείων καθώς και η ανάπτυξη της εφαρμογής σύμφωνα με τις προδιαγραφές του Μοντέλου Προγραμματισμού μπορεί να είναι σύνθετες εργασίες. Μια διεπαφή διαχείρισης για την παρακολούθηση και διαχείριση των διαφόρων λειτουργιών των εργαλείων και για τη διευκόλυνση της διαδικασίας ανάπτυξης μέσω ενός γραφικού περιβάλλοντος χρήστη μπορεί να απλοποιήσει την πολυπλοκότητα που δέχεται ο χρήστης. Μια σειρά πρόσθετων λειτουργιών μπορεί να εισαχθεί σαν αυτό που προσφέρει νέες δυνατότητες εκμετάλλευσης της σειράς εργαλείων. Μερικά παραδείγματα απαριθμούνται παρακάτω:

- Εκτέλεση των εργαλείων χρησιμοποιώντας μια γραφική διεπαφή για τη διευκόλυνση της χρήσης τους

- Ανάπτυξη της εφαρμογής παρέχοντας μια διασύνδεση που μπορεί να απομακρύνει την πολυπλοκότητα των οδηγιών του μοντέλου προγραμματισμού, επιτρέποντας στους χρήστες να προσαρμόζονται εύκολα με το μοντέλο καθώς συνεχίζονται

- Παρακολούθηση της ανάπτυξης της εφαρμογής σε πραγματικό χρόνο, συμπεριλαμβανομένων των διαφόρων σταδίων της ανάλυσής της (παραλληλισμός, κατανομή πόρων, σύνταξη κ.λ.π.)

- Παρακολούθηση της εκτέλεσης της εφαρμογής σε πραγματικό χρόνο, παρουσιάζοντας χρήσιμες αναλύσεις σχετικά με τις διάφορες υλοποιήσεις, ενώ μετρώνται επίσης διαφορετικές παράμετροι

National Technical University of Athens
School of Electrical and Computer Engineering
Devision of Computer Science

A Study for Application Development, Optimization and Deployment on Parallel Architectures

MASTER THESIS

IOANNIS E. TZANETTIS

Supervisor : Georgios Goumas
Assistant Professor NTUA

Athens, November 2019

National Technical University of Athens
School of Electrical and Computer Engineering
Devision of Computer Science

A Study for Application Development, Optimization and Deployment on Parallel Architectures

MASTER THESIS

IOANNIS E. TZANETTIS

Supervisor : Georgios Goumas
Assistant Professor NTUA

Approved by the examining committee on the November 11, 2019.

Georgios Goumas Assistant Professor NTUA	Nectarios Koziris Professor NTUA	Nikolaos Papaspyrou Professor NTUA

Athens, November 2019

Ioannis E. Tzanettis
Electrical and Computer Engineer

Abstract

Recent advancements on computationally intensive algorithms in various fields like Image Processing, Control-Flow-Dynamics and Machine Learning showcase the massively increased need for computational power. On the other hand, High Performance Computing is under continuous development offering advanced hardware architectures and parallel programming models and protocols that increase the performance and decrease the cost of such tasks. Still, the exploitation of these technologies is only available to specialized users, while average programmers can only deal with a limited set of technologies that have been developed enough to include a simple interface able to abstract all complexity.

In this Diploma Thesis, we studied a series of different technologies that combined can facilitate the development, optimisation and deployment of a parallel application. Focus was given to the design of the architecture of a framework, which takes the form of a multi-component toolchain offering a variety of different functionalities. The utilization of such a framework aims to assist the user to implement highly efficient applications, able to be deployed on a variety of different architectures and memory spaces.

A Parallel Programming Model has been designed guiding the user through the application design procedure. This model abstracts the programmer from the low-level implementation, taking care of all communication and deployment tasks. The integration and deployment of the application is the responsibility of the Deployment Manager tool that has been developed to implement the low-level directives and library inclusions. The Programming Interface is part of the Programming Model and mainly consists of a set of libraries developed to allow communication between the software components of the application. The Auto-Parallelization tool is the result of an extended study of the state-of-the-art techniques utilized in the field and exploits the design of the Programming Model to automatically parallelize parts of the application that can be computationally intensive. The Technique Selection decision tool exploits user directives for assigning parts of the application on the different hardware components and selects the corresponding technologies to implement the low-level communication and deployment functionalities.

The main objectives of this thesis can be summarized into three parts namely chapters 3, 4 και 5. At the first part, we discuss current parallelization technologies, focusing specifically on the performance evaluation of the PLuTo parallelizer and the ROSE Compiler, while also extending relevant work to cover more cases and optimize performance. At the second part, we describe in detail the parallel programming model that is used for the integration of the different tools. Finally, at the third part, we discuss the integration details of such a framework (AEOLUS) and how their interaction can provide the outcome in the form of an integrated and optimised parallel application.

Key words

Parallel Systems, Computer architecture, Parallel Programming, Framework, Parallelization

Acknowledgements

I would like to thank my professor, Georgios Goumas, for all his valuable help throughout this period. Not only did he provide me with guidelines and support on the subject of this thesis, but also he introduced me to the field of Operating and Parallel Systems through the multiple courses that I studied under his supervision. His interest and enthusiasm have driven me to face my work in the same way.

Furthermore, I would like to give my thanks to my company WINGS ICT Solutions along with the PHANTOM Horizon 2020 project which I have worked with and exchanged information, while providing a big part of this thesis to its development. Specifically, I would like to thank my managers Panagiotis Vlacheas and Panagiotis Demestichas who have supported me every step of the way with their advice, their understanding and trust throughout this period.

Additionally, I would like to thank all my friends who have supported and motivated me all this time and especially Konstantinos, Haris and Stathis whom I have co-progressed with during the last years of my studies. I would also like to express my deep gratitude for my mother, father, brothers and sister who have stood by my side, supporting me and advising me through all the difficult times. Special thanks to Eleni who was always patiently by my side to listen to me and help me whenever I needed it.

Through these few lines, I wish to acknowledge everyone who has helped me complete this work during this intense period of my life with their moral support, input and advice.

Ioannis E. Tzanettis,

Athens, November 11, 2019

National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Computing Systems Laboratory, November 2019.

URL: `http://artemis.cslab.ece.ntua.gr/`

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays, computing tasks tend to become more and more demanding every day. The rise of machine learning, data science and image processing, as well as the demand for real-time applications show clearly the need for more computation power and reduced power consumption. This is where the value of heterogeneous systems becomes evident. Current and future research extensively focuses to the ability to combine computation power of high performance systems (e.g. clusters) with low consumption of smaller, energy-efficient devices (e.g. embedded devices, FPGAs etc.) and utilize it to promote the development of highly efficient systems able to optimally execute all kinds of applications.

On the other side, developing parallel applications for different devices is a time consuming and difficult procedure, not meant for the average programmer who is only familiar with serial programming. Plenty of libraries and frameworks have been developed to support parallel programming using multiple threads, processes, cores or different accelerators. However, developing a multi-dimensional application that can exploit different levels of a parallel architecture including external accelerators demands familiarity with multiple such frameworks and can result in widely increased effort for the programmer requiring extensive debugging and reformatting of the application to adjust to the existing architecture.

Although designing heterogeneous parallel applications is a complicated and multi-dimensional task, a big part of its procedure can be automated assuming partial guidance by the user. A framework that can facilitate parallel application design should be aware of different languages and programming models, be able to exploit different architectures, while at the same time providing a high level programming model abstracting the user from the low level implementations.

1.2 Objectives

The idea behind a framework that can facilitate and enhance the procedure of parallel application development aims to abstract the high levels of difficulty that exist when dealing with complex system architectures. Different environments mean different programming styles, interfaces and even languages, which in many cases makes heterogeneous deployment extremely complicated for the average programmer. An intermediate “agent” between the programmer and the available target system should take up the low-level implementations of the application relieving the former from the responsibility of dealing with the complexity that is created and making fine-tuned parallelism available to a less expert crowd.

The problem we will be facing is based on the design and the architecture of a toolchain that will allow the user to design a parallel application while following specific guidelines, as well as the implementation of some key components that constitute the core of the framework, on which more components can be built and developed completing its functionality. The aim of this thesis is to provide a framework sufficient enough to support the design of parallel applications consisting of multiple components that can be executed in parallel, as well as their individual parallelization

and finally their deployment on a distributed and parallel architecture consisting of multiple computer nodes each with its own multicore capabilities. The design of the programming model will be designed in a way, so that a heterogeneous architecture, also including accelerators (e.g. GPUs) or FPGAs, can also be considered if such devices are available.

Chapter 2

Background

2.1 Moore's Law

Moore's Law asserts that the number of transistors on a microchip doubles every two years, though the cost of computers is halved. In other words, we can expect that the speed and capability of computers will increase every couple of years, and that their cost will be less and less. Another tenet of Moore's Law is that this growth in the microprocessor industry is exponential—meaning that it will expand steadily and rapidly over time. To break down the law even further, it specifically stated that the number of transistors on an affordable CPU would double every two years (which is essentially the same thing that was stated before) but 'more transistors' is more accurate.

However, if you were to look at processor speeds from the 1970's to 2009 and then again in 2010, one may think that the law has reached its limit or is nearing the limit. In the 1970's processor speeds ranged from 740 KHz to 8MHz. From 2000 – 2009 there has not really been much of a speed difference as speed ranges from 1.3 GHz to 2.8 GHz, which suggests that speed has barely doubled within a 10 year span. This is because we are looking at the speed and not the number of transistors; in 2000 the number of transistors in the CPU numbered 37.5 million, while in 2009 the number went up to an outstanding 904 million; this is why it is more accurate to apply the law to transistors than to speed.

But Moore's Law may not be able to go on indefinitely. The high tech industry might love its talk of exponential growth and a digitally-driven "end of scarcity," but there are physical limits to the ability to continually shrink the size of components on a chip. Already the billions of transistors on the latest chips are invisible to the human eye. If Moore's Law was to continue through 2050, engineers will have to build transistors from components that are smaller than a single atom of hydrogen. It's also increasingly expensive for companies to keep up. Building fabrication plants for new chips costs billions. As a result of these factors, many people predict Moore's Law will peter out some time in the early 2020s, when chips feature components that are only around 5 nanometers apart.

2.2 Parallel Architectures

To overcome the effect of Moore's Law's decline, the scientific community has been working more and more on the exploitation of multiple cores/processors/machines to execute tasks in parallel. The speed of individual CPUs may not increase as fast as before, but the collective work of computing devices aims to fill that gap and take up to support the increasingly demanding computing tasks that modern algorithms include. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

2.2.1 Multi-core computing

A multi-core processor is a processor that includes multiple processing units (called "cores") on the same chip. This processor differs from a superscalar processor, which includes multiple execution

units and can issue multiple instructions per clock cycle from one instruction stream (thread); in contrast, a multi-core processor can issue multiple instructions per clock cycle from multiple instruction streams. Each core in a multi-core processor can potentially be superscalar as well - that is, on every clock cycle, each core can issue multiple instructions from one thread.

Concurrent multithreading (e.g. Intel's Hyper-Threading) was an early form of pseudo-multi-coreism. A processor capable of concurrent multithreading includes multiple execution units in the same processing unit (i.e. it has a superscalar architecture) and can issue multiple instructions per clock cycle from multiple threads. Temporal multithreading on the other hand includes a single execution unit in the same processing unit and can issue one instruction at a time from multiple threads.

2.2.2 Shared Memory Architectures

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists.

SMPs use Uniform Memory Architecture (UMA) meaning that all processors access the memory with the same bandwidth and the same latency. There are machines that do not follow the same approach where memory is accessed non-uniformly. Machines that use Non-Uniform Memory Architecture (NUMA) show different latency when different processors access the memory.

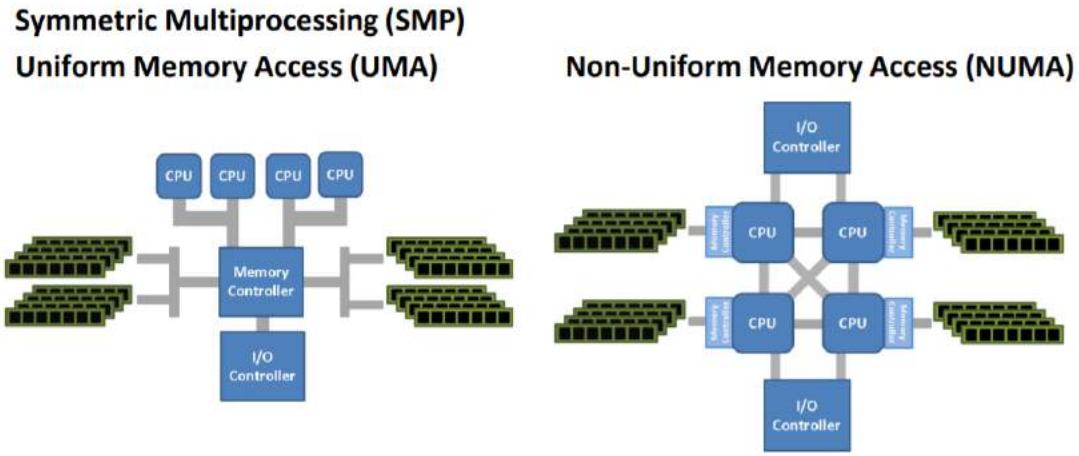


Figure 2.1: SMP vs NUMA architectures

2.2.3 Distributed Memory Architectures

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel.

Distributed architectures also include hybrid solutions that use multiple SMPs connected together through a network.

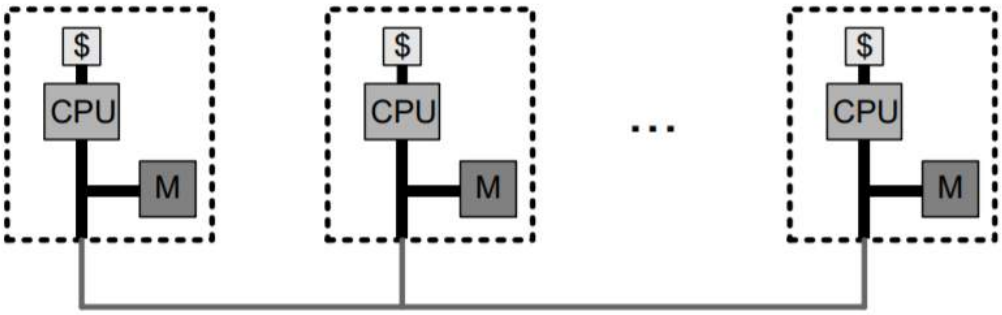


Figure 2.2: Distributed memory architectures

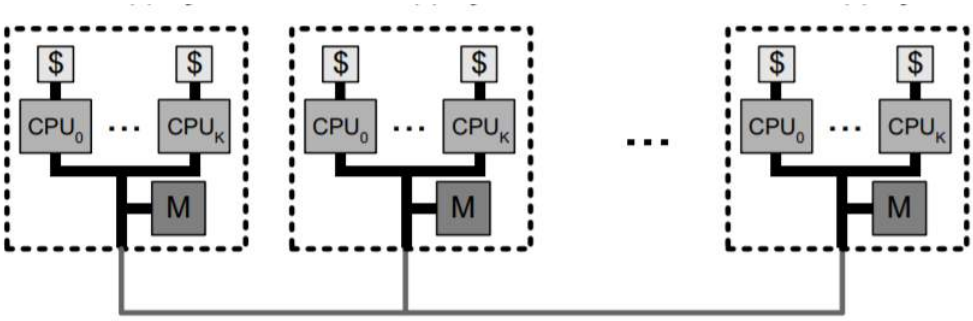


Figure 2.3: Hybrid architectures

2.2.4 Cluster computing

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. 87% of all Top500 supercomputers are clusters.

Because grid computing systems (described below) can easily handle embarrassingly parallel problems, modern clusters are typically designed to handle more difficult problems—problems that require nodes to share intermediate results with each other more often. This requires a high bandwidth and, more importantly, a low-latency interconnection network. Many historic and current supercomputers use customized high-performance network hardware specifically designed for cluster computing, such as the Cray Gemini network. As of 2014, most current supercomputers use some off-the-shelf standard network hardware, often Myrinet, InfiniBand, or Gigabit Ethernet.

2.2.5 Graphics Processing Units

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics and image processing. Their highly parallel structure makes them more efficient than general-purpose central processing units (CPUs) for algorithms that process large blocks of data in parallel. In a personal computer, a GPU can be present on a video card or embedded on the motherboard. In certain CPUs, they are embedded on the CPU die.

2.2.6 Field-Programmable devices

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term ”field-programmable”. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-Specific Integrated Circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

FPGAs contain an array of programmable logic blocks, and a hierarchy of ”reconfigurable interconnects” that allow the blocks to be ”wired together”, like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software.

2.3 Performance Modelling

Optimally, the speedup from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speedup. Most of them have a near-linear speedup for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl’s law:

$$S_{latency} = \frac{1}{1 - p + p/s} \tag{2.1}$$

where $S_{latency}$ is the potential speedup in latency of the execution of the whole task; s is the speedup in latency of the execution of the parallelizable part of the task; p is the percentage of the execution time of the whole task concerning the parallelizable part of the task before parallelization.

Since $S_{latency}$ < 1/(1 - p), it shows that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (serial) parts. If the non-parallelizable part of a program accounts for 10% of the runtime (p = 0.9), we can get no more than a 10 times speedup, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule.

Amdahl’s law only applies to cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, Gustafson’s law gives a less pessimistic and more realistic assessment of parallel performance:

$$S = 1 - p + sp \tag{2.2}$$

Both Amdahl’s law and Gustafson’s law assume that the running time of the serial part of the program is independent of the number of processors. Amdahl’s law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also independent of the number of processors, whereas Gustafson’s law assumes that the total amount of work to be done in parallel varies linearly with the number of processors.

2.4 Parallelization Technologies

When developing a parallel application, the programmer needs to take into account a series of parameters that are going to determine the tools and techniques that they will have to use for the implementation. The computing resources and architectures that are available narrow down the choices that can be used concerning the programming models that can be exploited. On the other hand, the needs of the application as well as its nature guide the programmer towards specific models that may be more beneficial for each specific case. In Table 2.1, we categorize the different approaches that can be followed and the different advantages/disadvantages that exist for each one.

		Architecture	
		Shared Memory	**Distributed Memory**
Programming Model	**Shared address space**	(1) + Easy implementation + Easy programming + High performance	(3) + Easy programming - Difficult implementation - Low performance
	Distributed address space	(2) + Easy implementation + High performance - Difficult programming	(4) + Easy implementation + High performance - Difficult programming

Table 2.1: Architectures and programming models

1. The processing elements executing the stream have immediate access to the same memory and can also refer to memory addresses using the same pointers. POSIX multi-threading is such an example, where the different threads are using the same memory and memory space.

2. The processing elements have access to the same memory, but cannot use the same memory references (pointers) since they use a different memory space. So, they have to use different mechanisms to transfer data other than pointers, hence the difficulty in programming. Different UNIX processes use the same memory, but refer to a different memory space.

3. The processing elements don't have access to the same physical memory, but can refer to the same memory space. Since memory accessing is indirect there is a cost to performance, but also to the implementation since there should be a middleware layer handling all memory operations.

4. The processing elements have access to neither the same physical memory, nor the same memory space. So, the user needs to implement any communication that is necessary for the application, making programming a difficult task as with the use of processes on the same memory space. An example of this approach is the Message Passing Interface (MPI) and its implementations (OpenMPI, MPICH).

2.4.1 Single perspective techniques

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems, including Linux, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It

uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

The CUDA platform is designed to work with programming languages such as C, C++, and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL. When it was first introduced by Nvidia, the name CUDA was an acronym for Compute Unified Device Architecture, but Nvidia subsequently dropped the common use of the acronym.

MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

Official implementations:

1. The initial implementation of the MPI 1.x standard was MPICH, from Argonne National Laboratory (ANL) and Mississippi State University. IBM also was an early implementor, and most early 90s supercomputer companies either commercialized MPICH, or built their own implementation. LAM/MPI from Ohio Supercomputer Center was another early open implementation. ANL has continued developing MPICH for over a decade, and now offers MPICH-3.2, implementing the MPI-3.1 standard.

2. Open MPI (not to be confused with OpenMP) was formed by the merging FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI, and is found in many TOP-500 supercomputers.

2.4.2 Hardware Description Languages (HDL)

Verilog

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits.

A subset of statements in the Verilog language are synthesizable. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip-flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations

to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bitstream file for an FPGA).

VHDL

VHDL (VHSIC-HDL) (Very High Speed Integrated Circuit Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.
VHDL is commonly used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, only if it is part of the logic design. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design.

2.4.3 Heterogeneous Computing

Heterogeneous computing can be defined as the coordinated use of different types of machines, networks, and interfaces to maximize their combined performance and/or cost-effectiveness. These systems benefit from the variety of parallelization capabilities that different kinds of processors, coprocessors and other targets like graphics processing units (GPU) or field-programmable devices can offer to different parts of the computation load. So, different advantages that different parts of hardware can offer are combined and used, each for the computation part where they are most valuable, optimizing the overall efficiency of the system.

OpenCL

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies programming languages (based on C99 and C++11) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

OpenACC

OpenACC (for open accelerators) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

As in OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions. Like OpenMP 4.0 and newer, OpenACC can target both the CPU and GPU architectures and launch computational code on them.

Chapter 3

Automatic Parallelization

The rise of computing in the last decades has caused the field of automatic parallelization to emerge, aiming to the complete hand-over of any optimization or parallelization tasks currently handled by the programmer to state-of-the-art compiling tools. Such tools are responsible, not only for the analysis and compilation of the code, but also its transformation to forms that allow the exploitation of the system's memory specifications as well as the available resources that can be used for task allocation and acceleration. A lot of research has been concentrated to the development of such techniques, directed to different computing environments, architectures and programming languages.

Although parallelization can be considered as the allocation of completely independent tasks to different processing resources, the most popular part of automatic parallelization mainly refers to the analysis and parallelization of loop statements, which include similar computation tasks executed on different data. This is mostly caused because of the complexity that arises when analyzing non-uniform tasks, but also because a big portion of the computing load is concentrated on these parts of the code. On the other hand, analyzing tasks that refer to the same actions but on different memory regions can be modeled a lot easier using existent compiling tools. Of course, different researchers in the field explore all possible directions that may provide fruitful results, but in the scope of this thesis we will not concentrate on approaches that diverge from loop-centric parallelization.

This section begins with a discussion of some of the most popular approaches that are considered in the field and it continues with the introduction and comparison of tools that have been developed using these techniques for the optimization and parallelization of manually-written programs. Finally, an extended programming model will be presented and used for the successful integration of the tools with a real-world use case application which will be used for their evaluation.

3.1 Code Optimisations

In this paragraph, the basic types of dependencies that can exist in both simple and complex loops are described along with a few techniques that have been proven very helpful concerning code analysis, transformations and parallelization. In specific, some of the most popular dependence tests are introduced, as well as a series of transformations that can be applied in the code, giving strong insight in the latest strategies that state-of-the-art tools follow in automatic parallelization.

3.1.1 Dependences

Two iterations are said to be dependent if they access the same memory location and one of them is a write. A true dependence exists if the source iteration's access is a write and the target's is a read. These dependences are also called read-after-write or RAW dependences, or flow dependences. Similarly, if a write precedes a read to the same location, the dependence is called a WAR dependence or an anti-dependence. WAW dependences are also called output dependences. Read-after-read or RAR dependences are not actually dependences, but they still could be important in characterizing reuse. RAR dependences are also called input dependences. Dependences are an important concept while studying execution reordering since a reordering will only be legal if does not violate the dependences,

i.e., one is allowed to change the order in which operations are performed as long as the transformed program has the same execution order with respect to the dependent iterations.

Due to the multiple dependencies that may occur in the code, an automatic parallelization tool cannot safely produce parallel code before checking for possible dependencies across different iterations of the loops. Mathematically a dependence can be described with the following definition:

Definition 3.1. Dependence existence A dependence exists if there exist iteration vectors $\vec{k}$ and $\vec{j}$ such that:

$$\vec{L} \leq \vec{k} \leq \vec{j} \leq \vec{U} \; and \; f_i(\vec{k}) = g_i(\vec{j}), \; 1 \leq i \leq d. \tag{3.1}$$

where $\vec{L}$ and $\vec{U}$ the lower and upper bound vectors of the loop. Another approach to describe the identification of a dependence relies on the check of the iteration vector where if it's found to be lexicographic-ally positive a dependence can be assumed to exist.

Some examples of dependence testing can facilitate the understanding of these definitions. Some powerful techniques about dependence testing have been developed and here we will present some of the most popular.

The Lamport test

The Lamport test is a simple test for index expressions involving a single index variable, and with the coefficients of the index variable all being the same, i.e.:

$$A[\ldots, b * i + c_1, \ldots] = A[\ldots, b * i + c_2, \ldots] \tag{3.2}$$

If there are i_1 and i_2 such that:

$$L \leq i_1 \leq i_2 \leq U \; and \; b * i_1 + c_1 = b * i_2 + c_2 \Rightarrow i_2 - i_1 = (c_1 - c_2)/b \tag{3.3}$$

then there is a dependence in the loop. Note that an integer solution exists only if $(c_1 - c_2)/b$ is an integer. Then the dependence distance is:

$$d = \frac{c_1 - c_2}{b}, \; if \; L \leq d \leq U \tag{3.4}$$

- If d>0 then the dependence is a true dependence
- If d=0 then the dependence is loop-independent
- If d<0 then the dependence is an anti-dependence

The GCD test

The GCD test is based on following theorem.

Theorem 3.1. *Equation 3.5 has an integer solution x_1, x_2,..., x_n iff the Greatest Common Divisor (GCD) of a_1, a_2,..., a_n divides c.*

$$\sum_{i=0}^{n} a_i x_i = c \tag{3.5}$$

e.g.: $2 * x_1 - 2 * x_2 = 1$. GCD(2,-2) = 2 and 2 cannot divide 1. So, there is no integer solution for the equation above.

It is difficult to analyze array references in compile time to determine data dependency (whether they point to same address or not). A simple and sufficient test for the absence of a dependence is the greatest common divisor (GCD) test. It is based on the observation that if a loop carried dependency exists between $X[a * i + b]$ and $X[c * i + d]$ (where X is the array; a, b, c and d are integers, and i is the loop variable), then GCD (c, a) must divide (d – b). The assumption is that the loop must be normalized – written so that the loop index/variable starts at 1 and gets incremented by 1 in every iteration. For example, in the following loop, a=2, b=3, c=2, d=0 and GCD(a,c)=2 and (d-b) is -3. Since 2 does not divide -3, no dependence is possible.

```
1 for(i=1; i<=100; i++)
2     X[2*i+3] = X[2*i] + 50;
```

The Banerjee test

The Banerjee test [2] is based on the Intermediate Value Theorem. It is another widely used test that calculates the possible minimum and maximum values an expression on the LHS of a linear equation can achieve, given bounds on each of the variables involved. Once the minimum and maximum of the expression is known, the test checks whether the constant on the RHS of the equation falls between these extreme values. If it does not, then no dependence exists. If it does fall in the range, we know only that a real solution to the linear equation exits. However, we cannot conclude that a dependence exists, since there may not in fact be an integer solution to the equation. This inability to distinguish between real and integer solutions makes the Banerjee test, though highly efficient, an inexact test. In addition to providing a no or maybe answer to a dependence test, the Banerjee test can be used to generate direction vector information.

3.1.2 Loop transformations

In compiler theory, loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops; as such, many compiler optimization techniques have been developed to make them faster.

Loop optimization can be viewed as the application of a sequence of specific loop transformations to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). Evaluating the benefit of a transformation or sequence of transformations can be quite difficult within this approach, as the application of one beneficial transformation may require the prior use of one or more other transformations that, by themselves, would result in reduced performance. Common loop transformations include:

- Fission
- Fusion
- Interchange
- Inversion
- Reversal
- Skewing
- Splitting
- Tiling
- Vectorization
- Unrolling

3.1.3 OpenMP annotations

We discussed about the usage of the OpenMP programming model in a previous section. The ease of use and understanding that is offered by OpenMP make it a preferable solution for many automatic parallelization tools which use OpenMP pragma annotations to execute tasks in parallel after identifying the ones that can be considered parallelizable. The annotations are placed before the parallelized code segments, splitting them to different threads.

3.2 The Polyhedral Model

The Polyhedral model for compiler optimization is a powerful mathematical framework based on parametric linear algebra and integer linear programming. It provides an abstraction to represent nested loop computation and its data dependencies using integer points in polyhedra.

Complex execution-reordering, that can improve performance by parallelization as well as locality enhancement, is captured by affine transformations in the polyhedral model. The model has reached a level of maturity in various aspects – in particular, as a powerful intermediate representation for performing transformations, and code generation after transformations.

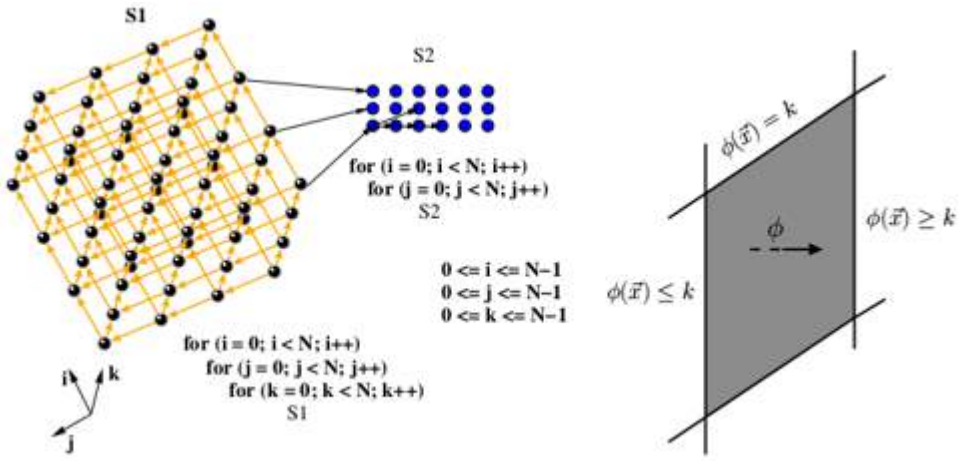


Figure 3.1: The polyhedral model

Some definitions are displayed below to facilitate the presentation of the techniques that the model relies on:

Definition 3.2. Affine Hyperplane The set X of all vectors $\vec{x} \in Z^n$ such that $h.\vec{x} = k$, for $k \in Z$, is an affine hyperplane.

Definition 3.3. Polyhedron The set of all vectors $\vec{x} \in Z^n$ such that $A\vec{x} + b \geq 0$, where A is an integer matrix, defines a (convex) integer polyhedron. A polytope is a bounded polyhedron.

Definition 3.4. Polyhedral representation of programs Given a program, each dynamic instance of a statement, S, is defined by its iteration vector $\tilde{i}$ which contains values for the indices of the loops surrounding S, from outermost to innermost. Whenever the loop bounds are linear combinations of outer loop indices and program parameters (typically, symbolic constants representing problem sizes), the set of iteration vectors belonging to a statement define a polytope. Let D_S represent the polytope and its dimensionality be m_S. Let $\vec{p}$ be the vector of program parameters.

3.2.1 Polyhedral dependences

The dependence model considered is referred to at [3][4][5][6]. Dependences are determined precisely through dataflow analysis, but we consider all dependences including anti (write-after-read),

output (write-after-write) and input (read-after-read) dependencies, i.e., input code does not require conversion to single assignment form.

Definition 3.5. Data Dependence Graph The Data Dependence Graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, $e \in E$, from node S_i to S_j representing a polyhedral dependence from a dynamic instance of S_i to one of S_j : it is characterized by a polyhedron, P_e, called the dependence polyhedron that captures the exact dependence information corresponding to e. The dependence polyhedron is in the sum of the dimensionalities of the source and target statement's polyhedra (with dimensions for program parameters as well). Let $\vec{s}$ represent the source iteration and $\vec{t}$ be the target iteration pertaining to a dependence edge e. It is possible to express the source iteration as an affine function of the target 102 iteration, i.e., to find the last conflicting access. This affine function is also known as the h-transformation, and will be represented by he for a dependence edge e. Hence, $\vec{s} = h_e(\vec{t})$. The equalities corresponding to the h-transformation are a part of the dependence polyhedron and can be used to reduce its dimensionality.

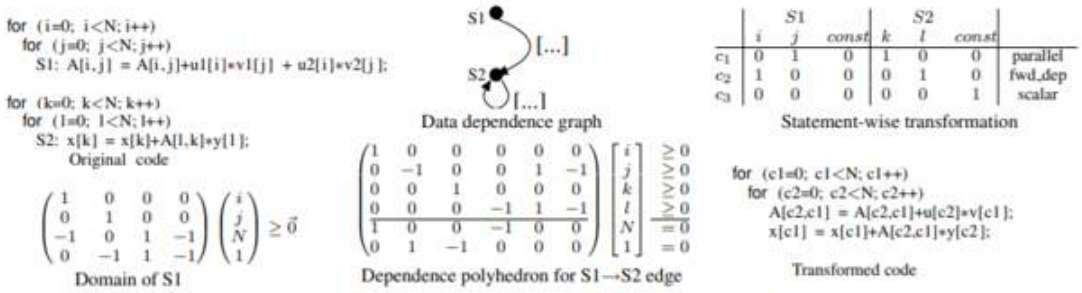


Figure 3.2: Polyhedral representation example

Figure 3.2 shows the polyhedral representation of a simple code. Let S_1, S_2, . . . , S_n be the statements of the program. A one-dimensional affine transform for statement S_k is defined by:

$$\phi_{s_k}(\vec{i}) = [c_1 \ldots c_{m_{s_k}}](\vec{i}) + c_0, c_i \in Z \tag{3.6}$$

ϕ_{s_k} can also be called an affine hyperplane, or a scattering function when dealing with the code generator. A multi-dimensional affine transformation for a statement is represented by a matrix with each row being an affine hyperplane.
Dependence satisfaction: An affine dependence with polyhedron P_e is satisfied at a level l iff the following condition is satisfied:

$$\forall k(1 \leq k \leq l-1) : \phi^k_{s_j}(\vec{t}) - \phi^k_{s_i}(\vec{s}) \geq 0, < \vec{s}, \vec{t} > \in P_e \tag{3.7}$$

and

$$\phi^l_{s_j}(\vec{t}) - \phi^l_{s_i}(\vec{s}) \geq 0, < \vec{s}, \vec{t} > \in P_e \tag{3.8}$$

3.2.2 Loop transformations

Latest research indicates the importance of the polyhedral model in the area of automatic parallelization, which is used for transforming the code in such a way where existing dependencies are resolved and new parallelizable code segments appear.

There are two common approaches to loop transformation. In the first approach, researchers have considered compound transformations as a series of individual transformations on the loop nest. This approach is applicable to general loop nests, but it is not clear how to choose the best combination of transformations to apply to a given loop nest. A second approach considers transformations to be

matrix transformations on an iteration space. This approach is elegant but applicable only in limited cases, when the dependences can be summarized as distance vectors. The polyhedral model combines the mathematical rigor in the matrix transformation model with the greater generality of the sequence of individual transformations approach. Loop interchange (permutation), reversal and skewing are unified as unimodular transformations, and the dependence vectors incorporate both distance and direction information. This unification provides a general test to determine if the code obtained via a compound transformation is legal, as opposed to a specific legality test for each individual elementary transformation. Thus the loop transformation problem can be formulated as directly solving for the transformation that maximizes some objective function, while satisfying a set of constraints. Using this theory, multiple algorithms have been developed for improving the parallelism and locality of a loop nest via loop transformations. Tiling, while not a unimodular transformation, is also considered and described as a commonly used technique in the automatic parallelization field.

Permutation, reversal and skewing are three well known transformations that can be represented as unimodular matrices.

Permutation: A permutation σ on a loop nest transforms iteration $(p_1 \ldots p_n)$ to $(p_{\sigma_1} \ldots p_{\sigma_n})$. This transformation can be expressed in matrix form as I_σ, the n×n identity matrix I with rows permuted by σ. The loop interchange above is an n = 2 example of the general permutation transformation.

Reversal: Reversal of the $i_t h$ loop is represented by the identity matrix, but with the $i_t h$ diagonal element equal to 1 rather than 1. For example, the matrix representing loop reversal of the outermost loop of a two-deep loop nest is:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Skewing: Skewing loop I_j by an integer factor f with respect to loop I_i maps the iteration:

$$(p_1 \ldots p_{i-1}, p_i, p_{i+1} \ldots p_{j-1}, p_j, p_{j+1} \ldots p_n) \tag{3.9}$$

to:

$$(p_1 \ldots p_{i-1}, p_i, p_{i+1} \ldots p_{j-1}, p_j + f p_i, p_{j+1} \ldots p_n) \tag{3.10}$$

The transformation matrix T that produces skewing is the identity matrix, but with the element tj,i equal to f rather than zero. Since i < j, T must be lower triangular.

Tiling: The tiling transformation despite the fact that it is not a unimodular transformation, is a primary transformation for improving locality, and can also be employed to expose coarse-grain parallelism. Tiling encompasses the well-known transformations of strip-mine and interchange [7] and unroll-and-jam [8]. As with unimodular transformations, the programmer must consider under what conditions the transformation is legal, and how applying it to a loop nest changes the loop nest. The most popular tiling methods consider only rectangular tiling. However, non-rectangular tiles can be achieved by combining (rectangular) tiling with unimodular loop transformations.

3.3 Automatic Parallelization Tools

3.3.1 ROSE Compiler – autoPar tool

ROSE [14] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++ and Fortran applications. Since it preserves the representation of high-level abstractions, no required information to recognize such abstractions is lost and the associated semantics can be reliably inferred. ROSE allows even non-expert users to exploit compiler techniques to address the analysis and transformation of abstractions.

A parallelizer was designed in the context of the development of the ROSE project using the compiler to automatically parallelize target loops and functions by introducing either omp for or omp task, and other required OpenMP directives and clauses. For input programs with existing OpenMP directives, the tool will double check the correctness when the right option is turned on.

It is designed to handle both conventional loops operating on primitive arrays and modern applications using high-level abstractions. The parallelizer uses the following algorithm:

1. Preparation and Preprocessing
   (a) Read a specification file for known abstractions and semantics.
   (b) Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
   (c) Normalize loops, including those using iterators.
   (d) Find candidate array computation loops with canonical forms (for omp for) or loops and functions operating on individual elements (for omp task).
2. For each candidate:
   (a) Skip the target if there are function calls without known semantics or side effects.
   (b) Call dependence analysis and liveness analysis.
   (c) Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
   (d) Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.
   (e) Insert the corresponding OpenMP constructs if no dependencies remain.

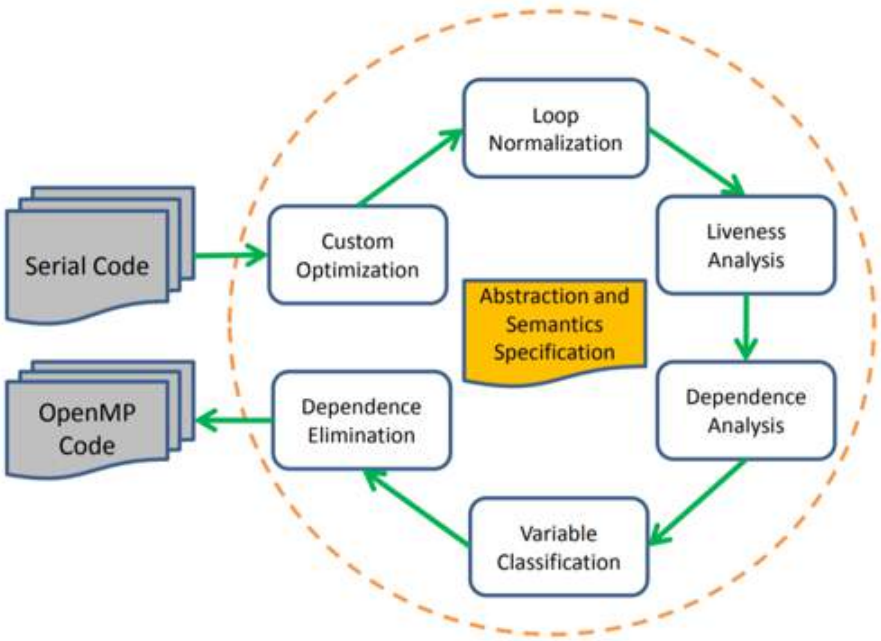


Figure 3.3: ROSE Compiler toolflow

The key idea of the algorithm is to capture dependencies within a target and eliminate them later on as much as possible based on various rules. Parallelization is safe if there are no remaining dependencies. Semantics of abstractions are used in almost each step to facilitate the transformations and analyses, including recognizing function calls as variable references, identifying the current element being accessed, and ensuring if there are constraints for the ordering of write accesses to shared variables.

3.3.2 PLuTO parallelizer

PLuTO is an automatic parallelization tool based on the polyhedral model. The tool transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling. OpenMP parallel code for multicores can be automatically generated from sequential C program sections. Outer (communication-free), inner, or pipelined parallelization is achieved purely with OpenMP parallel for pragmas; the code is also optimized for locality and made amenable for auto-vectorization.

As already discussed, the polyhedral model stands as a complete model for analyzing the data flow and applying any transformations that are considered necessary for exploiting parallelism. However, an approach to automatically find good transformations for communication-optimized coarse-grained parallelization together with locality optimization has been a key missing link. PLuTO has been developed to offer exactly that; a new automatic transformation framework that solves the above problem. This approach works by finding good affine transformations through a powerful and practical linear cost function that enables efficient tiling and fusion of sequences of arbitrarily nested loops. This in turn allows simultaneous optimization for coarse-grained parallelism and locality. Synchronization-free parallelism and pipelined parallelism at various levels can be extracted. The framework can be targeted to different parallel architectures, like general-purpose multicores, the Cell processor, GPUs, or embedded multiprocessor SoCs.

The described framework has been implemented in [10][11], into a new end-to-end transformation tool, PLuTO, which can automatically generate parallel code from regular C program sections. Experimental results from the implemented system show significant performance improvement for single core and multicore execution over state-of-the-art research compiler frameworks as well as the best native production compilers. For several dense linear algebra kernels, code generated from Pluto beats, by a significant margin, the same kernels implemented with sequences of calls to highly-tuned libraries supplied by vendors. The system also allows empirical optimization to be performed in a much wider context than has been attempted previously. In addition, PLuTo can serve as the parallel code generation backend for several high-level domain-specific languages.

The polyhedral compiler framework is an abstraction for analysis and transformation of programs. It captures the execution of a program in a static setting by representing its instances as integer points inside parametric polyhedra. Most publicly available tools and compilers that use this framework extract such a representation from C, C++, and Fortran programs.

Polyhedral representation of programs: Let S_1, $S_2 \ldots S_n$ be the statements of the program. Each dynamic instance of a statement, S, is identified by its iteration vector i that contains values for indices of the loops surrounding S, from outermost to innermost. Whenever the loop bounds are affine functions of outer loop indices and program parameters, the set of iteration vectors belonging to a statement form a convex polyhedron called its domain or index set. Let I_S be the index set of S and let its dimensionality be m_S. Let p be the vector of program parameters. Program parameters are not modified anywhere in the portion of code we are trying to model.

A function f on a domain I_S is called an affine function if it can be represented in the following form:

$$f(\vec{i}) = [c_1 c_2 \ldots c_{m_S}](\vec{i}) + c_0, i \in I_S \tag{3.11}$$

Regular data accesses in a statement are represented as multi-dimensional affine functions of domain indices. Codes that satisfy these constraints are also known as affine loop nests. Polyhedral dependences: The data dependence graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, e $\in$ E, from node S_i to S_j representing a polyhedral dependence from an iteration of S_i to an iteration of S_j: it is characterized by a polyhedron, D_e, called the dependence polyhedron that captures exact dependence information corresponding to e. The dependence polyhedron is in the sum of the dimensionalities of the source and target iterations spaces, and the number of program parameters. At least one of the source and target accesses has to be a write.

```
for(t=0; t<=-T1; t++)
    for(i=1; i<=-N2; i++)
        for(j=1; j<=-N2; j++)
            a[i][j]= ( a[-i1][-j1] + a[-i1][j] + a[-i1][j+1]
            + a[i][-j1] + a[i][j] + a[i][j+1] + a[i+1][-j1] + a[i+1][j] + a[i+1][
    j+1] ) / 9.0
```

For example, in the code segment above, the dependence between the write a[i][j] at $\vec{s}$ = (t,i,j) and the read at $\vec{t}$ = (t',i',j') at a[i′−1][j′−1] is given by the dependence polyhedron, D_e=($\vec{s}$, $\vec{t}$, $\vec{p}$, 1), which is a conjunction of the following equalities and inequalities:

$$i' = i + 1,\ j' = j + 1,\ t' = t \tag{3.12}$$

$$0 \leq t \leq T\text{–}1,\ 1 \leq i \leq N\text{–}3,\ 1 \leq j \leq N\text{–}3 \tag{3.13}$$

3.3.3 Intel C++ Compiler

Another choice for an automatic parallelization compiler that is available, but not open-source unlike the other two, is the Intel C++ Compiler. The auto-parallelization feature of the Intel C++ Compiler automatically translates serial portions of the input program into semantically equivalent multi-threaded code. Automatic parallelization determines the loops that are good sharing candidates, performs the data-flow analysis to verify correct parallel execution, and partitions the data for threaded code generation as is needed in programming the OpenMP directives. The OpenMP and auto-parallelization applications provide the performance gains from shared memory on multiprocessor systems.

Intel compilers are optimized to computer systems using processors that support Intel architectures. They are designed to minimize stalls and to produce code that executes in the fewest possible number of cycles. The Intel C++ Compiler supports three separate high-level techniques for optimizing the compiled program: interprocedural optimization (IPO), profile-guided optimization (PGO), and high-level optimizations (HLO). The Intel C++ compiler in the Parallel Studio XE products also supports tools, techniques and language extensions for adding and maintaining application parallelism on IA-32 and Intel 64 processors and enables compiling for Intel Xeon Phi processors and coprocessors.

Profile-guided optimization refers to a mode of optimization where the compiler is able to access data from a sample run of the program across a representative input set. The data would indicate which areas of the program are executed more frequently, and which areas are executed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions.

High-level optimizations are optimizations performed on a version of the program that more closely represents the source code. This includes loop interchange, loop fusion, loop fission, loop unrolling, data prefetch, and more.

Interprocedural optimization applies typical compiler optimizations (such as constant propagation) but using a broader scope that may include multiple procedures, multiple files, or the entire program.

3.3.4 CETUS

Cetus is a compiler infrastructure for the source-to-source transformation of programs. It was created out of the need for a compiler research environment that facilitates the development of interprocedural analysis and parallelization techniques for C, C++, and Java programs. Cetus is a set of intermediate representation (IR) classes and optimization passes and does not contain any proprietary code relying on freely available tools. For creating a Cetus parser the parser generators Yacc and Bison were considered, which use lex or flex for scanning, and Antlr, which is bundled with its own scanner generator.
Cetus has the following goals:

- The Internal Representation (IR) is visible to the pass writer (the user) through an interface, which we will refer to as the IR-API. Designing a simple, easy-to-use IR-API, that is extensible for future capabilities – especially to support other languages – is the most difficult engineering task.

- It must be easy to write source-to-source transformations and optimization passes. The implementation is an object-oriented class hierarchy with a minimal number of IR-API method names (using virtual functions and consistent naming), easy-to-use IR traversal methods, and information that can be inferred from other data strictly hidden from the user.

- Ease of debugging can be decisive for the success of any compiler project that makes use of the infrastructure. The IR-API should make it impossible to create inconsistent program representations, but we still need tools that catch common mistakes and environments that make it easy to track down bugs if problems occur.

- Cetus should run on multiple platforms with no or minimal modification. Portability of the infrastructure to a wide variety of platforms will make Cetus useful to a larger community.

3.4 Experimentation and comparison

3.4.1 Tool experimentation

For the evaluation of the tools we used the sandman machine, part of the CSLab resources, that is equipped with 4 Intel Xeon E5-4620 (Sandy Bridge) processors in a NUMA architecture. Each processor holds 8 cores, providing a total of 64 threads available to use for execution.

The ‘polybench’ benchmark suite was used for the comparison of the tools, as it provides a set of benchmarks specifically designed to test parallelization techniques. In specific, a series of computation-intensive, math-related applications is available in the suite, which covers perfectly the needs of the evaluation. The benchmarks are briefly presented in Table 3.1.

Benchmark	Description
2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)
3mm	3 Matrix Multiplications ((A*B)*(C*D))
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
head-3d	Heat equation over 3D data domain
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Table 3.1: The polybench benchmark suite

The parallelized versions of the benchmarks were executed on the sandman machine (4 x Intel Xeon E5-4620 - Sandy Bridge) of the lab using 1, 2, 4, 8, 16, 32 and 64 threads. From this experiment, a series of results were extracted concerning the speed-up that is observed after parallelization, as well as the scalability of the parallelized programs when additional computing resources are available. The results are illustrated in Appendix A.

General Observations

Regarding the benchmarks' scalability as it is illustrated in Appendix A, we can observe that both tools show a good scalability behaviour, while as we increase the number of threads used, execution time falls accordingly. In most cases, PLuTo proves to perform better than ROSE. We can assume that this is mostly caused due to the transformations performed by PLuTo to improve memory access operations and thus improving the overall performance even when using 1 thread.

To introduce a more enlightening overview of the tools' efficiency, we have sorted performance behavior in eight categories according to the overall gain and the scalability that we observe during the benchmarks' execution. For visualization purposes we can assume as initial execution time the performance of the benchmarks when running the ROSE version with one thread (i.e. the first point on each blue plot), since ROSE doesn't perform any code transformations and thus running this version using one thread is pretty much the serial/initial version.

The eight categories are defined as follows:

1. **High scalability**
   The cases where execution time gets lower with a fast rate as we use more threads

2. **Low Scalability**
   The cases where execution time gets lower with a slow rate as we use more threads

3. **No scalability, good performance**
   The cases where no scalability is observed as we increase the number of threads, but performance is better due to code optimizations (only for PLuTo)

4. **First worse, then better**
   The cases where code optimizations provide a big overhead at first, but with the usage of more threads performance get better than the original (only for PLuTo)

5. **Early saturation**
   The cases where after introducing a small number of threads, performance doesn't get better with the addition of more threads

6. **First better, then worse**
   The cases where after introducing a small number of threads, performance gets worse with the addition of more threads

7. **No performance gain**
   The cases where there is no performance gain when introducing more threads

8. **Scaling backwards**
   The cases where the addition of threads only causes lower performance

The results are visualized using the above categories in Table 3.2.

	High scalability	Low scalability	No scalability, good performance	First worse, then better
ROSE	16	2	0	0
PLuTo	6	3	7	1

	Early saturation	First better, then worse	No performance gain	Scaling backwards
ROSE	0	3	4	5
PLuTo	1	3	6	2

Table 3.2: polybench benchmarks evaluation

Considering Table 3.2, we can see that half of the benchmarks were successfully optimized either by code optimizations improving memory accesses and parallelization capabilities, or by the introduction of multiple threads which allowed the executables to run in parallel and complete their tasks sooner. This is a good indicator about the tools' efficiency and the impact of their utilization by developers of applications that include big computational loads.

On the other hand, we can notice that around 1/3 of the benchmarks appear to have minimal gain from the utilization of the tools, or in some cases even demonstrate a big overhead introduced by the addition of multiple threads. Of course, there are also some middle-ground results where we can see some improvement with the utilization of a small amount of threads and others where we need even more threads to see some improvement.

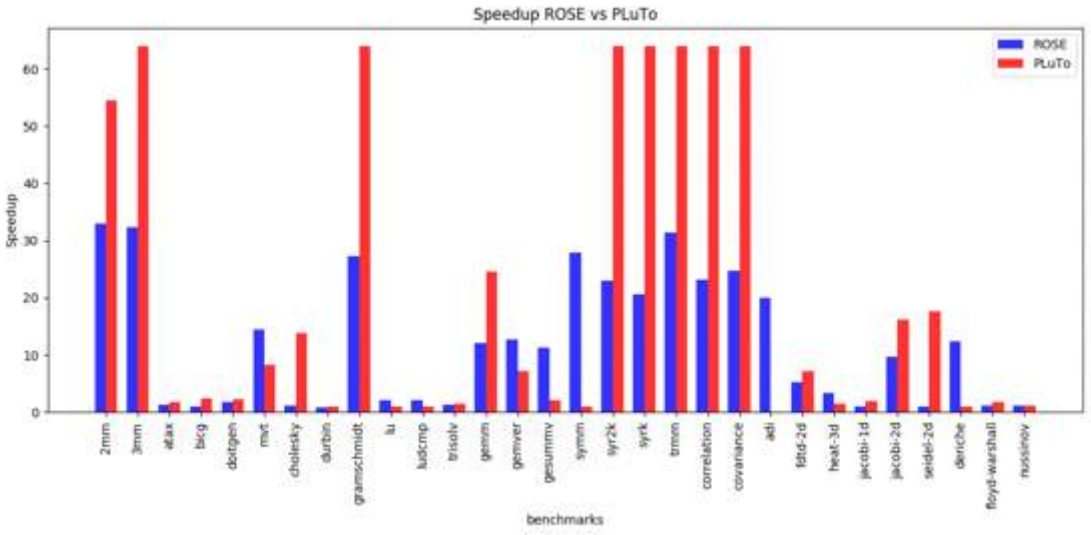


Figure 3.4: Overall speedup gained from parallelization/optimization

Speedup Analysis

In Figure 3.4, the overall speedup of each benchmark has been calculated from:

$$S_{speedup} = \frac{Initial\ Execution\ Time}{min_x(Parallelized\ version\ running\ using\ x\ threads)} \tag{3.14}$$

where x = 1,2,4,8,16,32,64 threads

As expected, PLuTo demonstrates some extraordinary performance gains in comparison to ROSE (Figure 3.4 results have been normalized to cut-off speedups greater than x64 to facilitate visualization).

For measuring the scalability that is offered by each tool, we also compared the best performance of each version of the code to the performance obtained from running the same version using on thread. That way we were able to look past the loop transformations and memory locality optimizations identifying the actual performance gains we got from increasing the number of threads for each version. The results are shown in Figure 3.5.

Here we can notice that the difference that we observed in the overall performance of the two tools is not as big now as before. By ignoring the performance gains from the code transformations, we can see the ROSE appears to scale a lot better than before in comparison with PLuTo whose performance gain is still big in some cases, but not that big to be considered as a better choice. Specifically, we can see the cases where ROSE outperforms PLuTo and notice that there is a big difference (see mvt, gramschmidt, gemver, gessumv, symm, trmm, deriche), whereas in the opposite case we can see that where PLuTo is better, ROSE stands a little below, meaning that the gain is similar.

3.4.2 Real-World Applications

Parallelization Tools

The PLuTo Parallelizer has proved to be an effective tool for parallelizing applications with the code transformation functionality it introduces as well as the OpenMP injections to the code. However, when we talk about real-world applications we should keep in mind that higher-level languages are usually preferred for computationally intensive applications, making the support of PLuTo for the C low-level language a more research-oriented solution.

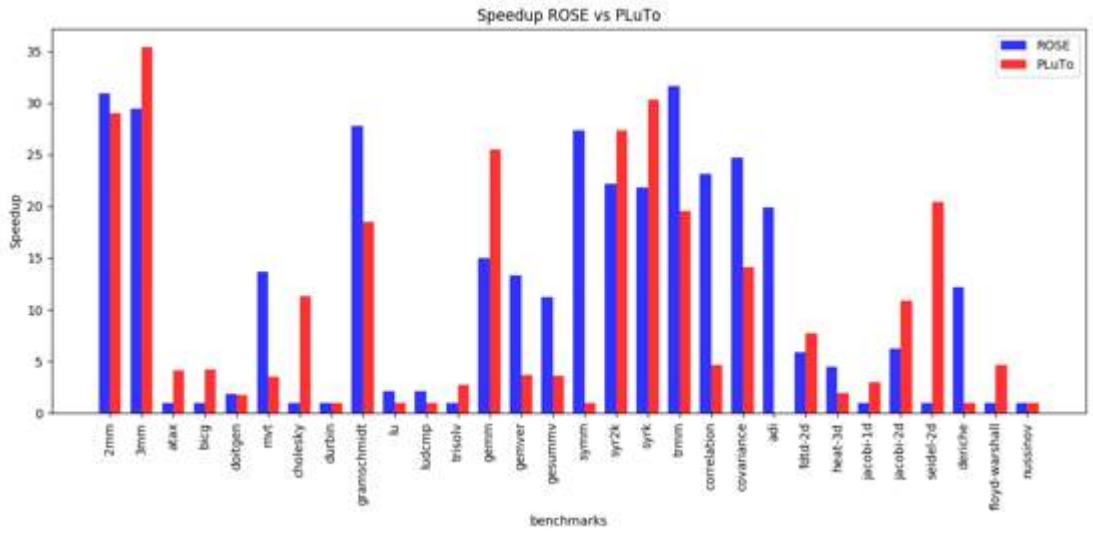


Figure 3.5: Speedup when compared with 1-threaded execution of parallelized version

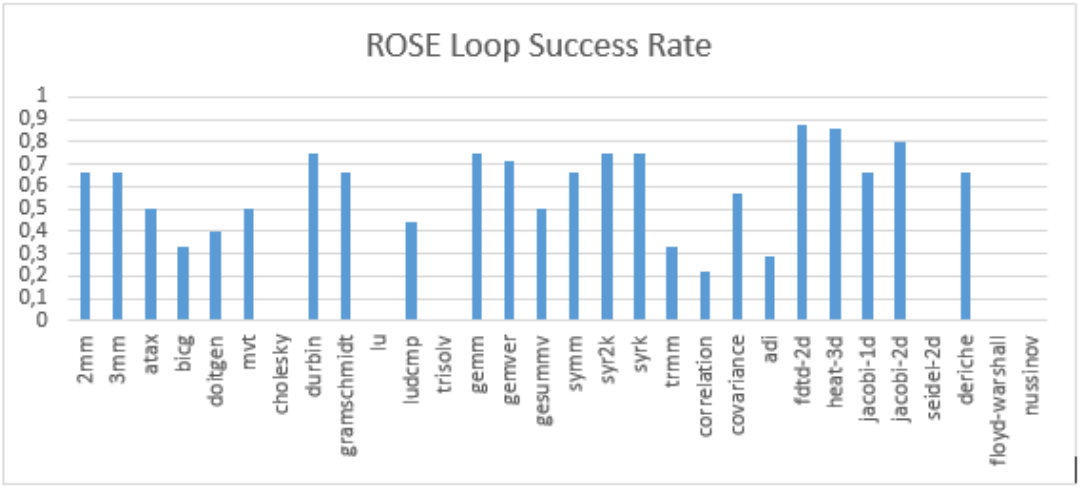


Figure 3.6: ROSE loop parallelization success rate

On the other hand, the ROSE Compiler is able to support C++ applications and as we stressed in the previous paragraph, the gain that we get in the general case when using PLuTo instead is not a big one when we talk about a big amount of threads. The autoPar tool of ROSE mainly works by injecting the code with OpenMP pragmas in the regions that are considered applicable from a dependence-related point of view. To test the tool's applicability, an evaluation test has been conducted in order to measure the total loops that are identified by the tool as parallelizable. The reason it was considered important for such an analysis to take place was the examination of the tool concerning its effectiveness on different kinds of loops with different dependences. Thus, if the tool is able to parallelize a big portion of the identified loops it can be considered applicable in a wide range of computation tasks.

As displayed in Figure 3.6, the majority of the identified loops in the polybench benchmarks were considered dependence-free, resulting in a loop parallelization rate more than 60% for most of the benchmarks, while the average loop parallelization rate is close to 50%. These results combined with the actual speed-up that was observed on the benchmarks were encouraging enough to allow us to move on to the next testing phase where we tested a series of real-world applications using ROSE to evaluate its performance.

```
vector<vector<int32_t> > replaceBlock(vector<vector<int32_t> > &mat_1,
                                      int32_t firstRow,
                                      int32_t lastRow,
                                      int32_t firstColumn,
                                      int32_t lastColumn,
                                      vector<vector<int32_t> > &mat_2)
{
    vector<vector<int32_t> > result = mat_1;
    int32_t mat2_i = 0;
    int32_t i, j;
    for (i = firstRow; i <= lastRow; i++) {
        int32_t mat2_j = 0;
        for (j = firstColumn; j <= lastColumn; j++) {
            result[i][j] = mat_2[mat2_i][mat2_j];
            mat2_j++;
        }
        mat2_i++;
    }
    return result;
}
```

Figure 3.7: Code segment from surveillance application - iterator identification

Testing the capabilities of the ROSE Compiler, we used an application related to the Surveillance vertical that concentrates on the analysis of satellite images. The application's code, due to its nature, includes big amounts of loops and array calculations which allows us to easily evaluate the tool's performance. Testing the tool against such code characteristics, we get the chance to inspect certain problems that occur with realistic applications where the use of parallelization tools would create significant performance gains.
When dealing with real applications we need to consider that the code we study is not always optimized and in many cases it's not feasible to produce the results that we expect. Such an example can be shown in Figure 3.7.

In Figure 3.7, we can easily observe that the two loops can be easily parallelized. However, the variables `mat2_i, mat2_j` create dependencies that can not be easily observed by automatic tools. This is because the variables are not identified as iterators of the loop (which would resolve the dependencies), even though they actually are, so their manipulation inside the loop does not allow the tool to guarantee their dependence-free parallelization. On the contrary, if the programmer had used functions of i and j instead of the variables (`mat2_i = i - firstRow, mat2_j = j - firstColumn`) for the array index, the iterator usage would be obvious for the parallelizer to notice and the dependencies would be resolved.

Another problem that we can easily observe in the aforementioned code segment is the explicit use of dynamic data structures, vectors, which allow the programmer to dynamically modify the allocated space for the data giving more flexibility to the manipulation of the structure. On the downside, the use of such structures implies their dynamic state and thus creates additional dependencies. In the specific example, we can even notice that the dynamic nature of the vectors is not used at all, which means that the programmer could have easily used static data structures such as arrays to implement this function, thus allowing its parallelization.

Such "small" but significant problems easily come up in applications designed under real circumstances and should be strongly considered when evaluating an automatic parallelizer. For example, ROSE Compiler that we chose to use doesn't apply any transformations to the code to eliminate such dependencies, while PLuTo applies complex transformation models (i.e. polyhedral) to parallelize the code more efficiently.

Common Dependencies

In this section, we list the most significant characteristics that can be identified in the code able to create unresolved dependencies and hinder the parallelization procedure overall:

- Indirect addressing: The problem in this case appears when accessing memory addresses that are not known at compile time. Parallelization of code that includes such characteristics may cause accessing the same memory space multiple times in parallel in a random order causing inconsistencies.
- Side effects: Calling functions with unknown side-effects in the code hinders its parallelization, since the compiler does not have the complete picture of the memory space management in the function's operations.
- Data flow dependencies: Data dependencies in the code's instruction flow do not allow its correct parallelization, creating constraints in the order of these instructions, hence to their parallel execution.
- Dynamic arrays/data structures: Usage of dynamic data structures includes a major constraint for code parallelization. When manipulating memory space in a dynamic manner, the compiler has no view of the structure's state, thus instructions running in parallel may cause inconsistencies and access on unallocated memory space.

It becomes apparent that the automatic parallelization tools and techniques that are available cannot create the required circumstances for the complete resolution of the aforementioned problems. Some of them can be completely or partially resolved, while others require specific tradeoffs that act against the automation of the procedure since there is not an explicit way to transfer the knowledge owned by the programmer to the parallelizer. In the next section, we will describe some of these tradeoffs and the gains they could offer to the parallelization procedure.

3.4.3 Introduction to an extended programming model

Data flow dependencies are some of the most common and significant characteristics that can block the parallelization procedure. However, there are ways to deal with them partially through appropriate transformations. Using tools such as PLuTo, capable of such optimizations can eliminate many dependencies and unlock many parallelization capabilities.

On the other hand, common coding practices include certain characteristics (indirect addressing, function calls, dynamic vectors) that obscure a big part of the program's functionality making it more complex and thus, hindering its analysis by the parallelizer. Of course, there are cases where such complex characteristics are used but due to the nature of the application, no actual dependences exist. However, due to the lack of understanding of the functionality by the parallelizer, these dependencies are still identified and not resolved.

To this direction, an extended programming model is being proposed here, to enable the user to easily include information about the code that cannot be extracted explicitly. In this way, the programmer is able to "interact" with the parallelizer and add details about memory access operations, staticality of vectors and function side-effects in order to unlock more parallelization capabilities. The model includes this information in the form of pragma annotations as described below:

#pragma aeolus function/loop static-vectors
Guarantees that the following function/loop includes only static vectors, meaning that there is no dynamic memory allocation during its execution.

#pragma aeolus loop no-aliasing
Guarantees that the function/loop does not include pointer aliasing, meaning access to the same memory address from different pointers.

```
for (int i = startRow; i <= endRow; i++) {
    for (int j = startCol; j <= endCol; j++) {
        if (cond[i][j]) {
            // Parallelisable Loop
            for (int k = 0; k < rowBlockIndex.size(); k++) {
                result[i + rowBlockIndex[k]][j + colBlockIndex[k]] = true;
            }
        }
    }
}
```

Figure 3.8: Code segment from surveillance application - indirect addressing

#pragma aeolus function no-side-effects
Guarantees that the following function does not include any side-effects, meaning that there is no writing on memory addresses or streams, thus the flow of the program is not affected any more than the function's returning value.

The following code segment displays a good example of false aliasing in practice where the use of the 'rowBlockIndex' and 'colBlockIndex' arrays to indirectly access the 'result' array creates false dependences that cannot be resolved with the current parallelization techniques.

Adding here the 'no-aliasing' annotation can instruct the parallelizer that the dependences related to indirect access can be safely eliminated.

The output of this work regarding the utilization of this extended programming model for the optimisation of the results obtained by ROSE is an Auto-Parallelization Tool that is referenced again in the context of chapter 5.

3.4.4 Extended Benchmark Suite

In order to further test the improvement obtained when using the extended programming model, we extracted a set of computationally intensive functions containing such parts from real-world applications. We used these parts to include some real examples of code that can be found in applications and extend our benchmark list with additional code characteristics that a parallelization tool should support. Some examples from this list can be found below.

```
1 vector<double> replaceIndexByVector(vector<double> &vec, vector<int32_t> &
    index_vec, vector<double> &value) {
2     int32_t i;
3     int32_t index_size = index_vec.size();
4     vector<double> result = vec;
5     for (i = 0; i < index_size; i++)
6         result[index_vec[i]] = value[i];
7     return result;
8 }
```

In the above segment, code from a benchmark including indirect addressing characteristics is shown. Here, the 'result' vector is indirectly accessed with the '$index_vec$' vector as index. For logic-related reasons, it becomes obvious that there is no overlapping between different loop iterations, so when inserting the 'no-aliasing' annotation the dependencies are eliminated. Additionally, we can add the 'static-vector' annotation to declare the staticality of the used vectors during the execution of the code.

```
1 int32_t square(int32_t x) {
2     return x*x;
3 }
```

```
int32_t square_sum(vector<int32_t> &vec) {
    int32_t i;
    int32_t vec_size = vec.size();
    int32_t result = 0;
    for (i = 0; i < vec_size; i++)
        result += square(vec[i]);
    return result;
}
```

Similarly, in the above segment we can see how the use of the ‘no-side-effects’ annotations can resolve dependencies that are bound to the unknown side-effects of called functions.

Chapter 4

Parallel Programming Modelling

As we stressed in previous sections, developing a parallel application in non-uniform architectures can be a very complex and time-consuming task. It includes the adaptation of various programming models and different communication protocols as well as the integration of different architectures. In this section, we will analyze the procedure that needs to be followed when developing a parallel application and compare it with the one when using a uniform parallel programming model. Next, we will present the design of a parallel programming model and the details that make its implementation.

4.1 A Parallel Programmming Model: Design

In order to facilitate the optimization of the deployment results in terms of performance and cost effectiveness as well as the flexibility of the system to exploit different parts of the available hardware, we assume a set of separate application components that are executed independently, but also work in synergy using a defined interface (which is described in section 4.1.3) to perform a certain functionality. The meaning of this decision resides on the fact that the tool should be able to distribute the computation load in different ways on the available hardware to optimize the application's overall performance.

For example, let's consider a small application with two components A and B and a 2-CPU system with an available GPU. A possible mapping would include each component running on a (different) CPU. Another one would suggest that component A is executed on CPU1 and component B on CPU2 with the latter using the GPU to execute part of the computation load on it.

So, the user can try different mappings between the components and the available resources to deploy the application and result in an efficient decision that satisfies their requirements.

To better structure this model, a set of classes has been defined in which all the necessary information is declared for both the software components and the data exchanges that take place between them. These classes describe two particular ontologies: the application components and the communication objects of the application.

Components: A component is defined as a separate, independent part of the application with specific attributes like name and ID that is able to interact with other components through communication objects.

Communication Objects: A communication object is defined either as a data transaction between components, with specific attributes like data size, data type, origin and target or as a synchronization action that aims to the coordination of the components that commit it. The classes are presented in Figure 4.1.

In Figure 4.2, we can see an example of how instances of those classes can form a network whose members relate to each other in the way that is described in Figure 4.1.

4.1.1 Application Model

In order for the user to be able to describe their application using the model presented above and submit it as input for the tool, an XML file has been designed following that model including all the necessary

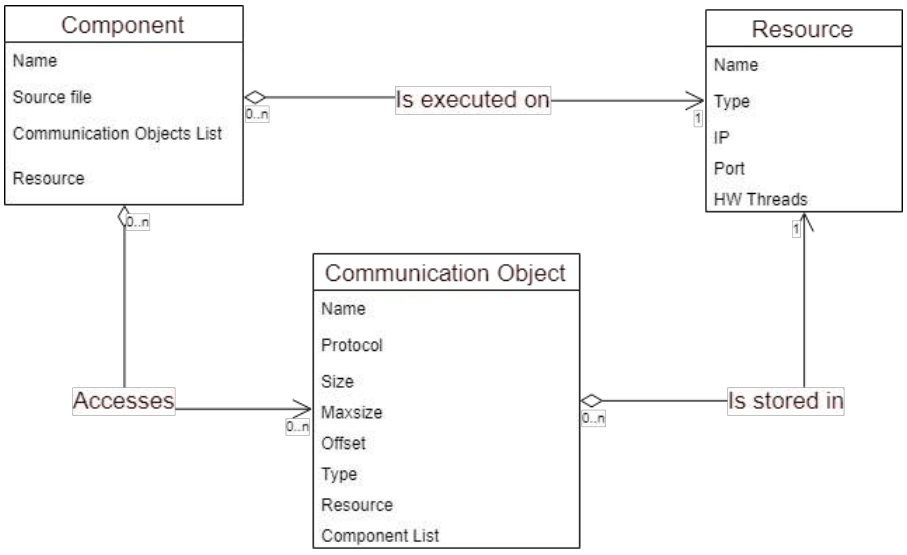


Figure 4.1: Application Model class diagram

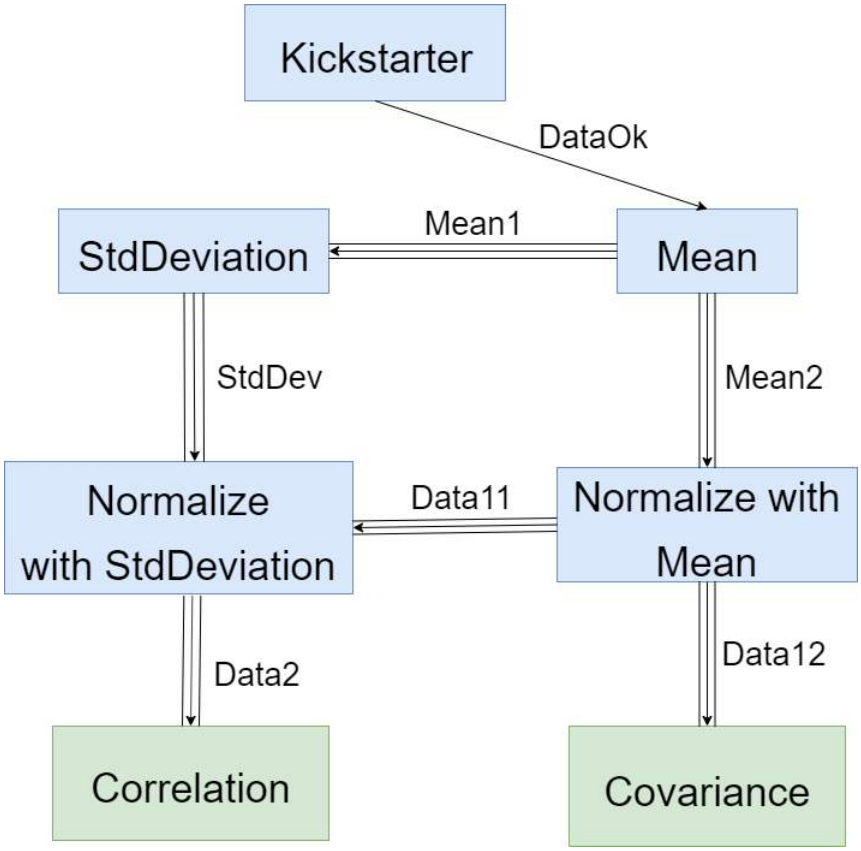


Figure 4.2: Example component network schema

information that is needed about the components and the communication objects of the application. An example of this file is displayed below:

```
<application name="tutorial" value="general-purpose">

<!--=================== Components description================= -->

```

```
<component name="C0" type="asynchronous">
    <source file="C0.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>
<component name="C1" type="asynchronous">
    <source file="C1.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>
<component name="C2" type="asynchronous">
    <source file="C2.cpp" lang="cpp" path="src"/>
    <devices CPU="yes" FPGA="no" GPU="no"/>
</component>

<!--==================Communication objects description================ -->

<comm-object item-size="16793807" name="Queue0" object-class="FIFO" size="5" type
    ="Queue">
        <source name="C0" port-name="inQueue00" type="in"/>
        <target name="C1" port-name="outQueue01" type="out"/>
        <target name="C2" port-name="outQueue02" type="out"/>
</comm-object>
<comm-object item-size="4" name="Shared0" object-class="shared" size="10" type="
    Shared-Memory">
        <source name="C2" port-name="inShared0" type="in"/>
        <target name="C1" port-name="outShared0" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal0" object-class="signal" size="1" type="
    Signal">
        <source name="C0" port-name="inSignal0" type="in"/>
        <target name="C2" port-name="outSignal0" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal1" object-class="signal" size="1" type="
    Signal">
        <source name="C1" port-name="inSignal1" type="in"/>
        <target name="C2" port-name="outSignal1" type="out"/>
</comm-object>
<comm-object item-size="1" name="Signal2" object-class="signal" size="1" type="
    Signal">
        <source name="C2" port-name="inSignal2" type="in"/>
        <target name="C1" port-name="outSignal2" type="out"/>
</comm-object>

</application>
```

An application using the Parallel Programming Model is defined by two different entities, software components and communication objects. Another example is shown in the segment below:

```
<applicationname="Example_Application"xsi:noNamespaceSchemaLocation="./aeolus.xsd
    ">
<component name="A"  type="asynchronous">
   <implementation id="1" model="any" target-HW="CPU">
      <source lang="c" file="CB.h" path="src\components"/>
   </implementation>
</component>
<component name="B" type="asynchronous" >
   <implementation id="1" model="any" target-HW="CPU">
      <source lang="c" file="CB.h" path="src\components"/>
   </implementation>
</component>
<comm-object name ="B" type="Buffer" object-class="FIFO" size="128" item-size="8"
    >
   <source name="A" port-name="inA1" type="in"/>
   <target name="B" port-name="outB1" type="out"/>
```

```
</comm-object>
<comm-object name ="S" type="Signal" object-class="Control">
   <source name="A" port-name="inA2" type="in"/>
   <target name="B" port-name="outB2" type="out"/>
</comm-object>
<comm-object name ="Sh" type="Shared Memory" object-class="Memory" size="1024"
    item-size="32">
   <source name="A" port-name="inA3" type="in"/>
   <target name="B" port-name="outB3" type="out"/>
</comm-object>
</application>
```

Components

An AEOLUS application is split into different software components that execute in parallel and are launched by the platform at the beginning of the execution. Each component corresponds to a specific source file as defined in the component network, which should have an entry point for the component's execution. Components can be reused multiple times as long as this is explicitly defined in the Component Network. They are initiated as separate and independent functions.

Communication Objects

To enable the communication between the different application components, the AEOLUS Programming Model defines another entity to specifically define the data exchanges or the coordination functionalities that are required by the application. Four types of Communication Objects have been developed, Shared, Queue, Signal, Mutex (described in the next section), which are implemented by the corresponding interfaces. The protocols have been furtherly enhanced to use specific data types for each communication object, defined by the AEOLUS Programming Interface. In specific, the following types are provided:

- aeolus_shared
- aeolus_queue
- aeolus_signal
- aeolus_mutex

4.1.2 Deployment Model

Except from the design that the application should follow to complete its functionality, a deployment model should be provided by the user as well, for identifying the various hardware targets and platforms that are available and for mapping the application components on them for execution according to each component's properties.

Hardware Elements/Resources

In order to create an actual model of the different deployment plans that can be executed, we have to define a set of hardware elements on which the different application components can be deployed. So, we enumerate and describe here, the different elements that are supported by the model.

- **CPU – only (SMPs) processing nodes**
  These elements only include general – purpose processors (one or more) that only work on shared memory. These descriptions include information like the core number of each processor, its frequency etc.

- **CPU – GPU processing nodes**
  Similar to the CPU – only model, this element considers an additional GPU device connected to the node and easily accessed by the general – purpose processor of the node. Software components that are mapped to be deployed on these elements use the CPU for their execution and offload specific calculations on the device for exploiting the resources that are available for acceleration.

- **CPU – FPGA processing nodes**
  Similar to the CPU – only model, this element considers additional FPGA logic resources that can facilitate memory access operations and accelerate specific calculations. Again, the software components mainly run on the CPU and offload specific functions that can be optimized on the hardware resources that are available on the FPGAs.

Platform Description

The platform description carries high-level information about the hardware, such as the CPU, memory and communication channels in the system or other properties that describe more specific target devices like GPUs, FPGAs etc. The goal of this description is to allow AEOLUS to handle easily a variety of architectures. An example of the XML file is displayed below:.

```
<platform name="localhost" xsi:noNamespaceSchemaLocation="./hw_development.xsd">

<!-- localhost -->
<device name="DevelopmentMachine" type="CPU-SMP" reliability="5">
        <processing-node name="unit1" type="CPU-SMP" architecture="SMP"> <!--
    Intel Core i7-6700K -->
                <processor name="local" type="INTEL-COREi7">
                        <configuration name="core number" value="4"/>
                        <configuration name="cpu frequency" value="4.0" unit="GHz
    "/>
                        <configuration name="bytespercycle" value="1"/>
                        <memory name="LOCALMEM1" type="RAM" size="2048" size-unit
    ="MB" access-time="1" access-time-unit="ns/word"/> <!--assuming 8-byte word
    -->
                </processor>
        </processing-node>

        <comm_interface name="enp0s3" type="Ethernet Network">
                <configuration name="speed" value="100" units="MBit/s" ip="
    localhost" user="demo"/>
        </comm_interface>

</device>

</platform>
```

The Platform Description XML file is used to define the system architecture including all characteristics that are useful for the deployment. Below, a CPU-FPGA platform description is also depicted:

```
<platform>
<device name="CPU-FPGA device" type="CPU-FPGA" reliability="2">
  <processing-node name="UIZynq-unit1" type="CPU-SMP" architecture="SMP">
    <processor name="UIZynq-P1" type="ARM-Cortex">
      <configuration name="core number" value="2"/>
      <configuration name="cpu frequency" value="800" unit="MHz"/>
      <configuration name="bytespercycle" value="1"/>
      <memory name="UIZynq-SM3" type="DDR3" size="1024" size-unit="MB" access-
    time="8" access-time-unit="ns/word"/>
    </processor>
```

```
  </processing-node>
  <processing-node name="UIZynq-unit2" type="FPGA" brand="Xilinx">
    <fpgalogic name="UIZynq-PL1" type="xc7z045ffg900-2">
      <resource name="UIZynq-LC" type="logiccell"/>
      <resource name="UIZynq-LUT" type="lookuptables"/>
      <resource name="UIZynq-LUTRAM" type="lookuptablesRAM"/>
      <resource name="UIZynq-FF" type="flipflop"/>
      <resource name="UIZynq-BRAM" type="blockRAM"/>
      <resource name="UIZynq-DSP" type="digitalsignalprocessing"/>
      <resource name="UIZynq-BUFG" type="bufferglobal"/>
      <configuration name="UIZynq-maxfrequency" value="200" units="MHz"/>
      <memory name="UIZynq-SM1" type="DDR3" size="1024" size-unit="MB" access-
    time="8" access-time-unit="ns/word"/>
    </fpgalogic>
  </processing-node>

  <local_bus name="UIZynq-AXI" type="AXI4" throughput="15" throughput-time-unit="
    Bytes/ns"/> <!--15GB/s-->
  <comm_interface name="UIZynq-EXT" type="Ethernet Network">
    <configuration name="UIZynq-speed" value="100" units="MBit/s" ip="192.168.1.2
    " user="external"/>
  </comm_interface>
</device>
</platform>
```

Deployment Plan

In order to deploy the application on the available hardware infrastructure, the user attempts to map the component network over the platform description. The result of this mapping process is a deployment plan. Deployments are represented in the same XML-based format as previously discussed. An example is shown below:

```
<deployment name="tutorial_Mapping" xmlns="DE" xsi:noNamespaceSchemaLocation="../
    models/deployment.xsd">

<!--            Estimated Execution Time:705.0  local:5.0mW   -->
<!--            Estimated Reliability:5.0-->
<!--           Component Deployment             -->
<target-application name="tutorial"/>
<target-hw-platform name="DevelopmentMachine"/>
<mapping name="component_C0_map" type="processing">
        <component name="C0" comp_id="0" subcomponents="1" secure="false"/>
        <CPU processor-name="local" device-type="CPU-SMP" CPU-type="SMP"/>
</mapping>
<mapping name="component_C1_map" type="processing">
        <component name="C1" comp_id="1" subcomponents="1" secure="false"/>
        <CPU processor-name="local" device-type="CPU-SMP" CPU-type="SMP"/>
</mapping>
<mapping name="component_C2_map" type="processing">
        <component name="C2" comp_id="2" subcomponents="1" secure="false"/>
        <CPU processor-name="local" device-type="CPU-SMP" CPU-type="SMP"/>
</mapping>

</deployment>
```

4.1.3 Programming Interface

To facilitate the usage of the communication protocols, an interface is provided to the programmer and is described in the following paragraphs. Notice the difference between the user interface of the

protocols and the actual interface of the protocol functions that is described in the implementation section. The APIs demand information for which the Deployment Manager is responsible in order to automate the deployment procedure. The details on that subject are discussed in the corresponding section where the functionality of the Deployment Manager is described.

Shared Protocol

The shared protocol describes functions for exchanging data through the virtually shared memory. It provides the user with the ability to manipulate variables as shared in the integrated system environment. The protocol makes use of the relaxed memory consistency model meaning that any reads or writes instantly update the shared memory, so the programmer is responsible to call suitable synchronization functions in order to provide the necessary consistency.

Also, the user doesn't have direct access to the shared memory, so the synchronize function needs to be used in order to move data to and from it. It is shown here:

*bool aeolus_synchronize(void *item, int dir)* Causes the local view of the item to be updated according to the corresponding data on the shared memory, if the dir variable has the value 0. On the opposite case, the shared memory is updated according to the local view of the item.

Queue Protocol

The Queue Protocol provides suitable functions that enable the user to manage the communication between components that are linked with specific communication objects in the form of queues and more specifically the form of blocking FIFOs of arbitrary size. The functions of the Queue protocol provide the user with the ability store and access elements in a queue structure and to count the number of the elements that exist in the queue at a specific time. The functions addressed in the current stage are the following:

*bool aeolus_queue_get(void *item)*
Pulls an item from the queue, storing it in the address pointed by item. Returns true if the transaction was successful.

*bool aeolus_queue_put(void *item)*
Pushes to the queue the item in the address pointed by item. Returns true if the transaction was successful.

*bool aeolus_queue_peek(void *item)*
Gets an item from the queue, storing it in the address pointed by item but not extracting it from the queue. Returns true if the transaction was successful.

*uint32_t aeolus_queue_count(void *queue)*
Returns the number of items that exist in the queue at the time of the call.

Note: The Queue Protocol has been developed to support object transfers of variable length. For that purpose, the objects are passed to the queue functions serialized including the size of the object in the 4 first bytes. Although the developers are free to develop their own functions for serialization, some prototype ones have been developed and are provided for the facilitation of the process.

Signal Protocol

Apart from sharing and interchanging data, the Signal Protocol is also included which provides the user with signalling facilities, enabling the coordinated execution of components and sections inside each component. These functions also complement the functions of the Shared and Queue Protocols

since they also need to synchronize their execution between components. The functions enable the components to exchange signals as well as create synchronization barriers, as described below:

bool aeolus_wait(int src)
Blocks the current thread/process until the signal in question is notified by the thread/process with id=src.

bool aeolus_notify(int dst)
If dst = -1 unblock a random single thread/process waiting on the signal. Else unblock the thread/process with id=dst.

bool aeolus_notifyall()
Unblock all threads/processes waiting on the signal.

bool aeolus_barrier()
In addition, a barrier function is provided able to wait until all threads or processes before that call, have finished their work.

Mutex Protocol

Mutexes are used to enforce mutual exclusion between a set of components on a critical region where shared data is used. They are enforced at the component level and are unrelated to OS-level mutexes from other APIs, but their usage is similar to the one described in the POSIX standards. Applies to pointer and array types. The API functions that enable mutex usage are described below:

*bool aeolus_mutex_lock(void *mutex)*
Block until the current mutex can be owned by the requesting component.

*bool aeolus_mutex_unlock(void *mutex)*
Release the current mutex, if it is currently owned. Any components waiting on 'aeolus_mutex_lock' can then recontest for the mutex. If multiple components are blocked then a random one is awarded the lock.

*bool aeolus_mutex_trylock(void *mutex)*
Attempt to lock the mutex, but do not block if the attempt was unsuccessful. Returns true if the mutex was locked and false if not.

4.2 A Parallel Programming Model: Implementation

The point of dividing the application into separate components is about their parallel execution occupying different processes on different processing nodes. However, the total integration of the components into a single application requires the implementation of their in-between communication. So, a set of communication protocols have been designed to provide the user with sufficient data transfer as well as synchronization functions that are going to transparently (to the user) allow the necessary communication and coordination actions between the different components. In the following sections, we will discuss the design and implementation of each one of these protocols.

4.2.1 Data Transfer Support

The programming interface that will be presented allows internode communication between the components for very complex deployments without the user's intervention. However, the form of the data to be transferred requires the adoption of a specific model that follows a set of rules (guidelines). In order to keep this model as relaxed as possible, all efforts have been made for the extension of the support of data structures that can be moved among the different components.

Automatic Support

Primitive data types are all automatically supported by the Programming Interface. This includes the types: char, short, int, long, float, double and all qualified versions of the above (signed, unsigned, long, and long long, and those declared const).

Extended Support

- Structs and unions of the above types
- Arrays of the above types (including supported structs and unions) that are declared with a compile-time static size. This includes multidimensional arrays, where all dimensions have a compile-time static size.
- Variable-length arrays of the above types, since their size can be manipulated by the sizeof() operator at compile-time.
- Class instances of the above
- References to the above

Precisely, an array with a compile-time static size are arrays where sizeof() can resolve to a value at compile-time without having to generate run-time code. For the use of primitives, users are encouraged to use types included in the stdint.h library, because of their constant size among different architectures. In a different case, the Deployment Manager will take up to resolve type inconsistencies before the deployment.

Pointer types are not automatically supported because they are not meaningful when transferred to another memory space. This restriction also applies to structs with pointer fields. The same stands for C++ vectors and other structures whose size is variable and cannot be known at compile-time.

Handling Unsupported Situations

Cases where the data that needs to be transferred follows a more complex structure than the ones mentioned above, can also be easily handled by the user by including serialization of the data into a static structure (array) before the transfer like shown in the example below.

```
1  struct X *data_out; // Unsupported data type
2
3  int component_A() {
4      initialize(data_out);
5
6      #pragma queue out object_out
7      char object_out[100]; // Supported data type
8
9      serialize_X(data_out,object_out);
10     queue_put(object_out);
11     return 0;
12 }
13
14 struct X *data_in; // Unsupported data type
15
16 int component_B() {
17     #pragma queue out object_in
18     char object_in[100]; // Supported data type
19
20     queue_get(object_in);
21     deserialize_X(object_in,data_in);
```

```
22      return 0;
23  }
```

In general, the Programming Interface provides to the user a wide support for all kinds of transfers. Attention should be paid to the limitations (described above) coming from the address space switching when moving objects among different parts of the hardware architecture.

4.2.2 Communication Protocol Functions

Depending on the type of physical memory that is required for the communication, different libraries are used as described below.

In the case that more than one components are executed on a single shared memory system, the particular components are going to be executed as different threads sharing the same memory space. The communication objects that are manipulated only by components on the same processing node are implemented with the use of pthreads along with other POSIX functionalities (like mutexes and condition variables) to keep the communication cost low as well as the memory footprint.

For components that interact with the object when executed on different nodes, meaning a distributed memory system, a Message Passing technique is required. For this reason the OpenMPI library was chosen due to its efficiency and simplicity on different kinds of real world applications. In specific, the execution of different components will be implemented as different MPI processes and the communication between them will exploit a lot of the already existing OpenMPI interface functions that are available.

In the following paragraphs, a description of the protocols implementation' is attached along with some of the key functions of the protocols, in order to display a good view of the Interface's functionality. More details about the implementation will be discussed in the next section where the Deployment Manager module is described, due to the major dependence of the Programming Interface with it.

Shared Protocol - Shared Memory Implementation

The Shared Protocol is implemented by keeping an area in the available memory as a shared segment between the different components.

```
1  if(dir == 0) {
2      pthread_mutex_lock(&obj->lock);
3      memcpy(local_data,shared_data,dims[0]*data_size);
4      pthread_mutex_unlock(&obj->lock);
5  }
6  else if(dir == 1) {
7      pthread_mutex_lock(&obj->lock);
8      memcpy(shared_data,local_data,dims[0]*data_size);
9      pthread_mutex_unlock(&obj->lock);
10 }
```

Shared Protocol - Distributed Memory Implementation

When the shared data exists between different nodes another strategy is followed. A virtually shared address space is created between the processes that the components are executed on. The synchronize method is used to get/put any data from/to the shared memory.

At the low-level implementation, data is stored on different memories across the hardware platform and RMA operations provided by the MPI protocol are used to create the virtually shared area in the processes' address space. In specific, the MPI_Get() and MPI_Put() functions are exploited for the implementation of data movements which are needed to be one-sided. This means that the processor whose memory is actually updated (target) doesn't have to be active and only the processor that updates the memory (origin) needs to act.

Below, the creation of such an environment is displayed, along with the implementation of the the aeolus_synchronize() API function.

The generation of the displayed initialize() function is part of the Deployment Manager's functionality and will be fully discussed in section 5. Here, the allocation of the required shared memory window is presented, as well as the necessary function to update the local memory with the shared version of the data (or vice versa).

```
1
2  void initialize() {
3      MPI_Init(argc, argv);
4      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
5      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
6      buf = (int *)malloc(10*sizeof(int));
7      MPI_Win_create(buf, 10*sizeof(int), sizeof(int), MPI_INFO_NULL,
       MPI_COMM_WORLD, &win);
8  }
```

```
1  void aeolus_synchronize(void *local, void *shared, int dir, int caller, int owner
       , int disp, int size, MPI_Datatype type, MPI_Win win) {
2      if(dir == 0) {
3          if(caller != owner) {
4              MPI_Win_lock(MPI_LOCK_SHARED,owner,0,win);
5              MPI_Get(shared,size,type,owner,0,size,type,win);
6              MPI_Win_unlock(owner,win);
7          }
8          memcpy(local,shared,size*disp);
9      }
10     else if(dir == 1) {
11         memcpy(shared,local,size*disp);
12         if(caller != owner) {
13             MPI_Win_lock(MPI_LOCK_EXCLUSIVE,owner,0,win);
14             MPI_Put(shared,size,type,owner,0,size,type,win);
15             MPI_Win_unlock(owner,win);
16         }
17     }
18     else {
19         perror("Error, not a valid direction");
20     }
21 }
```

Queue Protocol - Shared Memory Implementation

The Queue Protocol uses a shared memory area as well to keep the data of the queue. In addition, a common pointer variable is used to access the queue and manipulate its data.

```
1  void aeolus_queue_get(aeolus_queue *queue, void* it, int data_size) {
2    while(queue->count == 0);
3    pthread_mutex_lock(&queue->lock);
4    aeolusQnode *tmp = queue->front;
5    queue->front = queue->front->next;
6    if(queue->front == NULL)
7      queue->rear = NULL;
8    queue->count--;
9    memcpy(it,tmp->item,data_size);
10   pthread_mutex_unlock(&queue->lock);
11   free(tmp);
12 }
```

```
bool aeolus_queue_put(aeolus_queue *queue, void* it) {
  aeolusQnode *new_node = generate_aeolus_Qnode(it);
  pthread_mutex_lock(&queue->lock);
  if(queue->rear == NULL) {
    queue->front = queue->rear = new_node;
  }
  else {
    queue->rear->next = new_node;
    queue->rear = new_node;
  }
  queue->count++;
  pthread_mutex_unlock(&queue->lock);
  return true;
}
```

Queue Protocol - Distributed Memory Implementation

The Queue Protocol, similarly to the Shared Protocol, uses RMA operations to store and exchange data. In specific, each communication object is stored in a specific process's address space according to the Deployment Plan and uses the MPI_Get() and MPI_Put() functions to access and send data across the hardware platform.

```
// queue_get function

prev_front = queue->front;

if(queue->front == queue->rear)
        queue->front = queue->rear = -1;
else
        queue->front = (queue->front + queue->disp) % info_offset;
queue->count--;
set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->count);

if(caller != queue->owner) {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
        MPI_Get(queue->qdata+prev_front,queue->disp,MPI_UNSIGNED_CHAR,queue->
    owner,prev_front,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
        MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,
    queue->owner,info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,queue->win);
        MPI_Win_unlock(queue->owner,queue->win);
}
memcpy(item,queue->qdata+prev_front,queue->disp);
```

```
// queue_put function

if(queue->rear == -1) queue->front = queue->rear = 0;
else queue->rear = (queue->rear + queue->disp) % info_offset; queue->count++;
set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->count);
memcpy(queue->qdata+queue->rear,item,queue->disp);

if(caller != queue->owner) {
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
        MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,
    queue->owner,info_offset,3*sizeof(uint32_t),MPI_UNSIGNED_CHAR,queue->win);
        MPI_Put(queue->qdata+queue->rear,queue->disp,MPI_UNSIGNED_CHAR,queue->
    owner,queue->rear,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
        MPI_Win_unlock(queue->owner,queue->win);
}
```

Signal Protocol - Shared Memory Implementation

The Signal Protocol uses POSIX conditional variables to transfer signals between components.

```
void aeolus_notify(aeolus_signal *curSignal, bool *rd, int aeolus_src) {
    pthread_mutex_lock(&curSignal->lock);

    curSignal->ready = *rd;
    if(curSignal->ready == true) {
       if(pthread_cond_signal(&curSignal->cond) != 0) {
              fprintf(stderr,"Failed to send signal\n");
              exit(0);
       }
    }
    else {
    perror("Signal called with value 0");
    exit(1);
    }
    pthread_mutex_unlock(&curSignal->lock);
}
```

```
void aeolus_wait(aeolus_signal *curSignal, bool *rd, int aeolus_cmpid) {
    pthread_mutex_lock(&curSignal->lock);
    while(curSignal->ready == 0) {
        if(pthread_cond_wait(&curSignal->cond,&curSignal->lock) != 0) {
            fprintf(stderr, "failed to wait the condition variable\n");
            exit(0);
        }
        else {
        curSignal->ready = 1;
        *rd = true;
      }
    }
    pthread_mutex_unlock(&curSignal->lock);
}
```

Signal Protocol - Distributed Memory Implementation

When executing on different nodes the MPI_Send(), MPI_Recv(), MPI_Bcast() functions are used for signaling and MPI_Barrier() for the aeolus_barrier() function.

```
void aeolus_notify(int *rd, int dst, MPI_Comm comm) {
  MPI_Send(rd, 1, MPI_INT, dst, 0, comm);
}
```

```
int aeolus_wait(int *rd, int src, MPI_Comm comm) {
    MPI_Status status;
    MPI_Recv(rd, 1, MPI_INT, src, MPI_ANY_TAG, comm, &status);
    return status.MPI_ERROR;
}
```

```
void aeolus_notifyall(int *rd, int src, MPI_Comm comm) {
  MPI_Bcast(rd,1,MPI_INT,src,comm);
}
```

```
void aeolus_barrier(MPI_Comm comm) {
  MPI_Barrier(comm);
}
```

Mutex Protocol - Shared Memory Implementation

The Mutex Protocol uses the corresponding functionality provided by the POSIX standards.

```
bool aeolus_mutex_lock(aeolus_mutex *mutex) {
    if(!pthread_mutex_lock(&mutex->lock))
        return true;
    else
        return false;
}
```

```
bool aeolus_mutex_unlock(aeolus_mutex *mutex) {
    if(!pthread_mutex_unlock(&mutex->lock))
        return true;
    else
        return false;
}
```

```
bool aeolus_mutex_trylock(aeolus_mutex *mutex) {
    if(!pthread_mutex_unlock(&mutex->lock))
        return true;
    else
        return false;
}
```

Mutex Protocol - Distributed Memory Implementation

As with the Queue Protocol, the Mutex Protocol keeps a priority list on a specific node, denoted by the Deployment Plan. So, every process uses RMA operations to access the owner of the mutex. If the lock is acquired by another process, then the calling process blocks using the blocking MPI_Recv() until the lock is available finishes. On the other end, the process holding the lock uses MPI_Send() to give the lock to one of the processes waiting for it.

```
// lock function

// Try to acquire lock in one access epoch
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
    MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */, 1,
    MPI_CHAR, mutex->win);
    MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->numprocs,
     MPI_CHAR, mutex->win);
    MPI_Win_unlock(mutex->home, mutex->win);

    assert(waitlist[mutex->ID] == 1);

    // Count the 1's
    for (i = 0; i < mutex->numprocs; i++) {
        if (waitlist[i] == 1 && i != mutex->ID) {
            // We have to wait for the lock
            // Dummy receive, no payload
            MPI_Recv(&lock, 0, MPI_CHAR, MPI_ANY_SOURCE, mutex->tag, mutex->comm,
     MPI_STATUS_IGNORE);
```

```
            break;
        }
    }
```

```
// unlock function

MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */, 1,
    MPI_CHAR, mutex->win);
MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->numprocs,
    MPI_CHAR, mutex->win);
MPI_Win_unlock(mutex->home, mutex->win);
assert(waitlist[mutex->ID] == 0);
// If there are other processes waiting for the lock, transfer ownership
next = (mutex->ID + 1 + mutex->numprocs) % mutex->numprocs;
for (i = 0; i < mutex->numprocs; i++, next = (next + 1) % mutex->numprocs) {
    if (waitlist[next] == 1) {
    // Dummy send, no payload
        MPI_Send(&lock, 0, MPI_CHAR, next, mutex->tag, mutex->comm);
        break;
    }
}
```

4.2.3 Initialization

The user can obtain a pointer to the communication object structure through an initialization method, so that they can call the rest of the API methods using this pointer. The initialization functions are shown below:

*aeolus_shared *aeolus_shared_init(char *comm_object_port)*

*aeolus_queue *aeolus_queue_init(char *comm_object_port)*

*aeolus_signal *aeolus_signal_init(char *comm_object_port)*

*aeolus_mutex *aeolus_mutex_init(char *comm_object_port)*

where comm_object_port is the name of the port of the object as defined in the Component Network. The port name is strictly related to the source file and not to the component entity, so components that are linked to the same source file should use the same port name in the component network for the corresponding objects.

An important note here is that the user needs to declare the communication object variables in order to use the corresponding protocol functions. Any communication object initializations should occur always locally inside the component's function. On the other hand, the object's pointer can be passed in any function the user requires to call the protocol functions (aeolus_queue_get, aeolus_synchronize etc.).

4.2.4 File Operations

The AEOLUS Programming Model has been extended with file operations that are going to enable file operations across a non-uniform architecture with a different memory where files may be located anywhere. The added functions are mirrors of the POSIX file operations in order to maximise compatibility. The stream objects returned by the functions are compatible with the other I/O functions from the C standard library, fprintf, fscanf, snprintf, sprintf, sscanf.

*FILE * aeolus_fopen (const char * filename, const char * mode)*

*int aeolus_fclose (FILE * stream)*

*int aeolus_fflush (FILE * stream)*

*size_t aeolus_fwrite (const void * ptr, size_t size, size_t count, FILE * stream)*

*size_t aeolus_fread (void * ptr, size_t size, size_t count, FILE * stream)*

*int aeolus_fgetpos (FILE * stream, fpos_t * pos)*

*int aeolus_fseek (FILE * stream, long int offset, int origin)*

*int aeolus_fileno(FILE *stream)*
Returns the integer file descriptor associated with the stream pointed to by stream.

*FILE *aeolus_fdopen(int fd, const char *mode)*
Associates a stream with the existing file descriptor, fd. The mode of the stream (one of the values: “r”, “r+”, “w”, “w+”, “a”, “a+”) must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to fd, and the error and end-of-file indicators are cleared. Modes “w” or “w+” do not cause truncation of the file. The file descriptor is not dup’ed, and will be closed when the script created by aeolus_fdopen() is closed. The result of applying aeolus_fdopen() to a shared memory object is undefined.

int aeolus_get_fd_flags(int fd, int value)
Retrieves the current position in the stream.

*long int aeolus_ftell(FILE *stream)*
Returns the current value of the position indicator of the stream. For binary streams, this is the number of bytes from the beginning of the file. For text streams, the numerical value can be used to restore the position to the same position later using aeolus_fseek().

4.2.5 Monitoring library

A monitoring interface has also been provided for the user to monitor specific parts of their application. The aeolus_monitor type is defined to give access to the monitoring functionalities provided by AEOLUS.

*aeolus_monitor *aeolus_monitor_init()*
Returns a pointer to the monitor that is used to stores metrics on local files.

*void aeolus_mf_start(aeolus_monitor *monitor)*
Registers the start of the component. If not used, the start of the execution will be automatically registered which may not identify exactly the start of the computationally intensive part of the component.

*void aeolus_mf_user_metric(aeolus_monitor *monitor, char *metric_name, int value)*
Registers a user defined metric that the user may find useful.

*void aeolus_mf_end(aeolus_monitor *monitor)*
Registers the end of the component. If not used, the end of the execution will be automatically registered which may not identify exactly the end of the computationally intensive part of the component.

Note: Same as with the communication object initializations, monitoring initializations should occur always locally inside the component’s function, while the monitor’s pointer can be passed in any function the user requires to call the monitoring functions.

4.2.6 Parallelization annotations

The programming model has been extended to include assisting annotations for the static analysis of the code to enhance the parallelization results of parallelization tools (as introduced in section 3.4.3). These provide additional information that is too complex to be extracted automatically and therefore can be used by developers to improve deployment of their code, without enforcing guidelines on all component developers.

#pragma aeolus function/loop static-vectors
Guarantees that the following function/loop includes only static vectors, meaning that there is no dynamic memory allocation during its execution.

#pragma aeolus loop no-aliasing
Guarantees that the function/loop does not include pointer aliasing, meaning access to the same memory address from different pointers.

#pragma aeolus function no-side-effects
Guarantees that the following function does not include any side-effects, meaning that there is no writing on memory addresses or streams, thus the flow of the program is not affected any more than the function's returning value.

The platform will verify these pragmas (when possible) in order to assist the developer, and error if they are violated, but in general they are understood as guarantees from the programmer to the platform.

Chapter 5

A Parallel Development, Optimization and Deployment Framework (AEOLUS)

5.1 Idea

The advantages of heterogeneous computing as already mentioned are based on the combination of different technologies for the exploitation of the various parallel code regions that behave optimally on different processor types. Thus, this variation of tools that are required is the main difficulty for the programmer who is responsible to have sufficient knowledge of these tools or to obtain this knowledge in order to achieve their goal.

The solution that is proposed here discusses the architecture and functionality of a framework for Application Development and Optimization in Parallel Architectures (AEOLUS), which provides multidimensional parallelization capabilities that will minimize the knowledge gap between the programmer and the different systems that can be used. Aiming at the development of applications in a heterogeneous environment, the tools will exploit state-of-the-art technologies, while also assisting deployment on multiple hardware components. The main component of the toolchain will provide the core of the deployment system and a set of additional modules will complete the architecture of the framework with more functionalities aiming at further optimizations for the application or the utilization of the system.

5.2 Architecture

The base of the system is a deployment component, called Deployment Manager that is responsible to deploy the application on the hardware infrastructure and implement the communication between the different hardware components. The application integration is completed with the use of specific methodologies supported by external libraries that are selected according to the requirements of the deployment by another component, Technique Selection. An additional module has been developed in order to support multidimensional code optimization and parallelization, referred to as Automatic Parallelization Tool.

An overview of the system architecture is displayed in Figure 5.1, while a brief description of the toolflow is explained.

As illustrated in Figure 5.1, AEOLUS, as implemented in the context of this thesis, consists of four main components:

1. The Programming Model
   A set of rules and guidelines that define the process of the development and implementation of the software components in a way that guarantees the successful deployment and parallelization of the application.

2. An Automatic Parallelization Tool
   A tool for parsing the code and producing parallelized versions of the components.

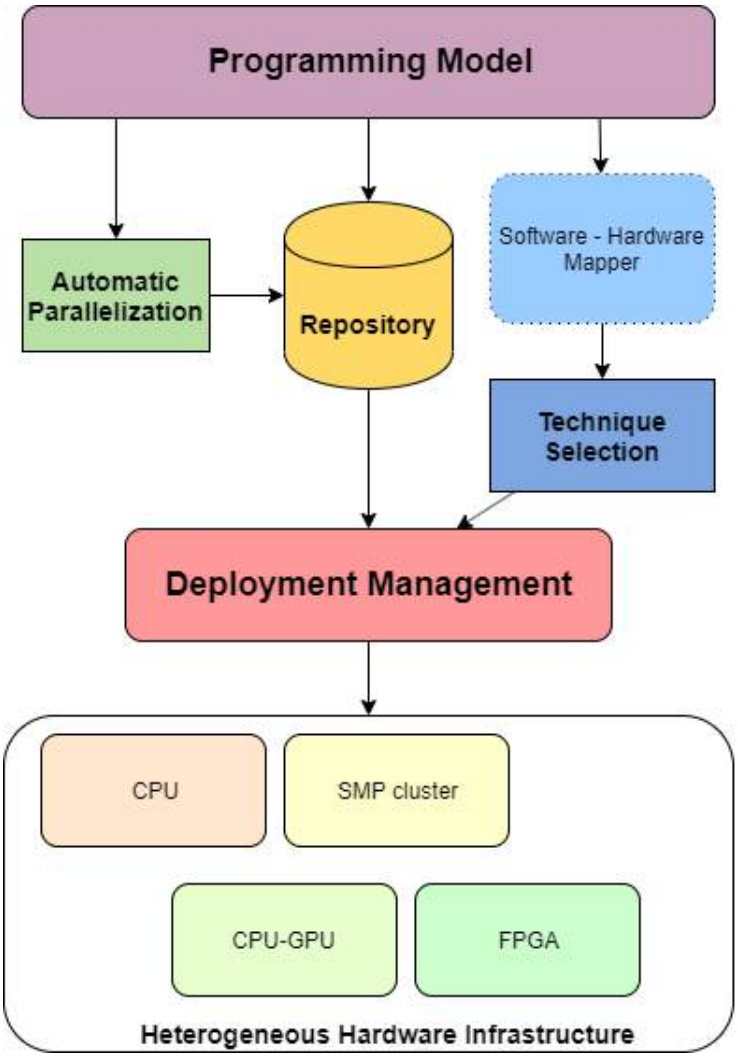


Figure 5.1: AEOLUS architecture

3. The Technique Selection module
   A module to select the suitable implementations of both the components and the communication code according to the given deployment plan.

4. The Deployment Manager
   A component to implement the deployment of the application components on the hardware infrastructure according to the deployment plan and the results from the Technique Selection module.
   Additionally, a Programming Interface (a set of APIs) has been developed for the implementation of the communication/synchronization between the application components.

The tool flow begins with the static parallelization of the individual application components resulting to the generation of parallel versions of the components and their storage in a pool that can be accessed by other modules. According to the mapping of the application components on the different hardware components (deployment plan), Technique Selection picks the suitable versions that will optimize the performance of the application on the hardware infrastructure. Additionally, a deployment strategy is selected, also according to the deployment plan, which directly corresponds to the choice of the communication APIs that are going to be used for the implementation of the components' interaction. Finally, the components, along with the external libraries required for their communication are passed to the Deployment Manager as a set of source code files for the actual implementation of the

deployment techniques that were selected in the previous stage. The result of the toolflow consists of a set of executables and configuration files on the corresponding targets that are finally executed and monitored. The overall procedure is guided by the structure that has been defined in the Programming Model.

5.3 Programming Model

The Programming Model that we defined in section 4 constitutes the core of the framework. Users develop their application by following specific guidelines defined in the Programming Model, creating an application that, according to its needs, can be deployed in a series of different hardware platforms. The Programming Model:

- supports the static parallelization of the different components enhancing the tool's capabilities with user annotations
- provides a variety of different choices for the deployment strategy (techniques/technologies) even for a single deployment plan
- creates the circumstances under which the components can be integrated and deployed as a single application
- provides a series of communication protocols that allow inter-component communication and synchronization
- Provides a variety of different features that can enhance the user's experience like monitoring the execution of the software components as well as accessing files on different machines

We have defined the Programming Model that is used by AEOLUS in section 4. In this paragraph, we describe how the framework exploits this model to support parallelization and load distribution of the application on the available hardware. More details about its different aspects are given in the corresponding sections of the tools that exploit them.

5.3.1 Components

Each component corresponds to a specific source file as defined in the component network, which should have an entry point for the component's execution. Components can be reused multiple times as long as this is explicitly defined in the Component Network. They are initiated as separate and independent functions. These functions exist in the component's dedicated source file with the corresponding name. The developer is able to choose to pass command line arguments to the function. Finally, the return type of the functions should be void*. So, the function's signature for a component with a source file comp_example.cpp should be like this:

*void *comp_example(int32_t argc, char **argv)*

or

*void *comp_example()*

Developers are free to include any external libraries they wish to use, as no compatibility issues were identified during development. On the other hand, in order for the components to use the AEOLUS libraries, the 'aeolus.h' header file should be included, so that the necessary structures and functions are visible from the component main function.

5.3.2 Communication Objects

AEOLUS defines a series of deployment libraries that are thoroughly described in the Deployment Manager section (section 5.6) and are responsible for the introduction of the different communication configurations that are needed for the application's successful execution.

One of these configurations includes the definition of the Shared Protocol's communication objects. The developer is able to transfer objects of user defined structures of static size. In order to enable such transactions, the user should include the libraries that define such structures in a dedicated header file for this purpose. This file is named 'aeolus_user_defined_structs.h' and it should be placed in the src directory (see next section).

5.3.3 Placing developer's application in the AEOLUS Repository

The AEOLUS Repository is used by the all the different tools of the framework. For this purpose the tools assume a specific structure in the Repository that the developer should respect when uploading any source, input or description files. The structure followed by the Repository contains a set of level-one folders which are considered the project directories. In the second level there is a set of directories defining the source (owner).

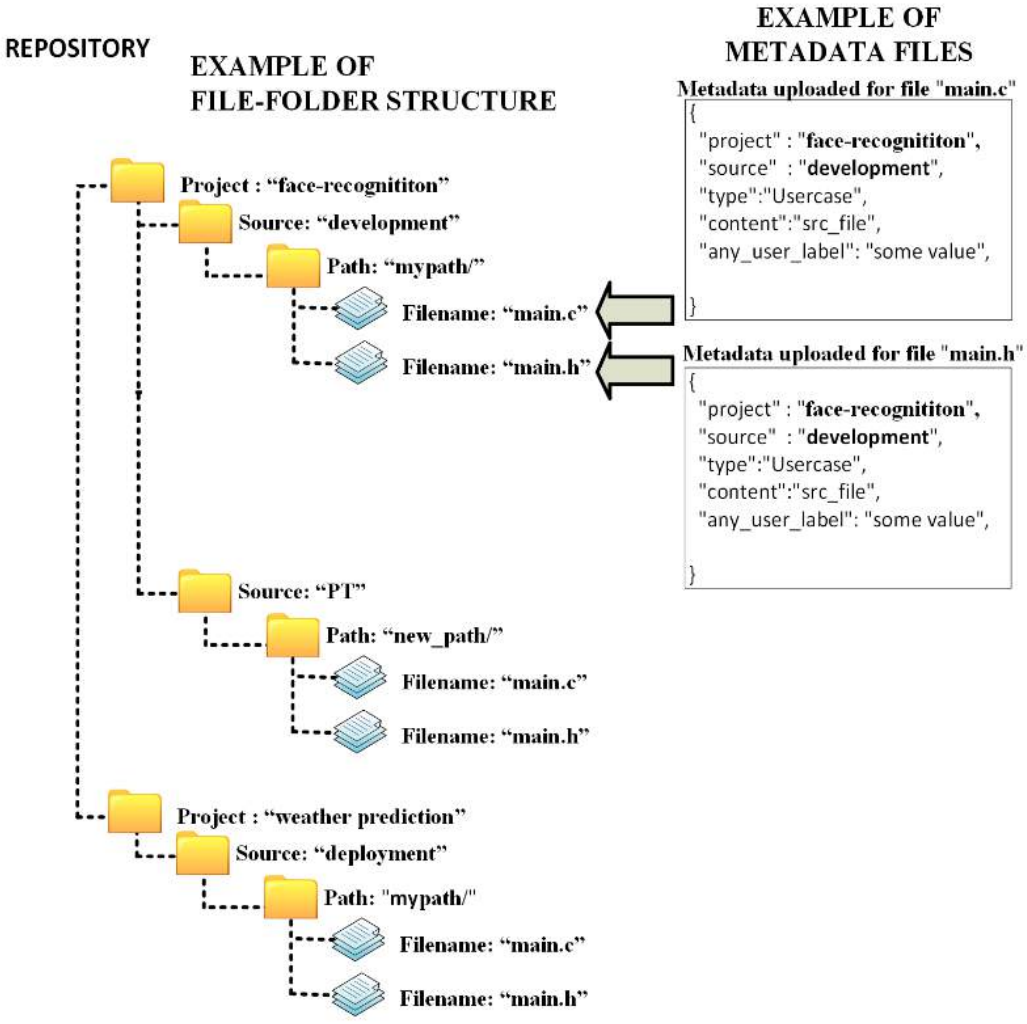


Figure 5.2: Repository structure

The developer must place all the necessary files on the Repository at *project_name/development* before the initiation of the tools. In specific, the following structure should be followed:

project_name/

development/

src/

aeolus_user_defined_structs.h

description/

Component-Network.xml

Platform-Description.xml

inputs/

outputs/

5.4 Automatic Parallelization Tool

5.4.1 Overview

The first stage of the framework's architecture is about the automatic parallelization of the application components. For this purpose, an automatic tool has been developed that analyses component source code and transforms it according to user-provided parallelization directives. The tool's functionality concentrates on the analysis of the code for the identification of parallelizable regions in it and the production of modified versions that enable further exploitation of the system resources. The analysis aims to isolate data dependencies, discovered during the analysis, that don't affect the potential concurrency. Code annotations can be provided by the programmer to extend the tool's understanding of the data flow and resolve many of those dependencies. A detailed description of the tool-flow is developed in the following paragraphs.

5.4.2 Design

The purpose of this tool in the AEOLUS architecture is the automatic generation of modified versions of the original components for their execution on different systems to exploit the code's special properties. The versions that are currently supported are:

OpenMP version: Includes OpenMP annotations that indicate the usage of multiple threads for the component's execution in CPU platforms.

CUDA version: A version including the CUDA API that communicates with the kernel of the GPU (the code that runs on the device) to enable GPU acceleration. This version is generated only when specified in the component network in terms of feasibility (optimality decisions are included in the Deployment Plan).

The generation of the updated components will give the ability to the framework to exploit the available hardware platform and accelerate the application's functionality. The flow of the tool is described in the steps below.

Component Network Analysis

The main activity of the tool is the automatic parallelization of the components and the libraries that are defined by the user. For the implementation of that interaction with the user, the tool gets input from the component network XML file, containing all the necessary information about the components (e.g. name, source file, external libraries used etc.) and stores that information using a set of classes designed to model the different components and the data transfers between them.

Automatic parallelization

After extracting the necessary information from the XML file, the tool can proceed with the parallelization of the code. Then, the actual analysis of the code is invoked for all the components and the external libraries (only the ones that are defined in the Component Network) that are used, as described in the following steps, also shown in the schema below:

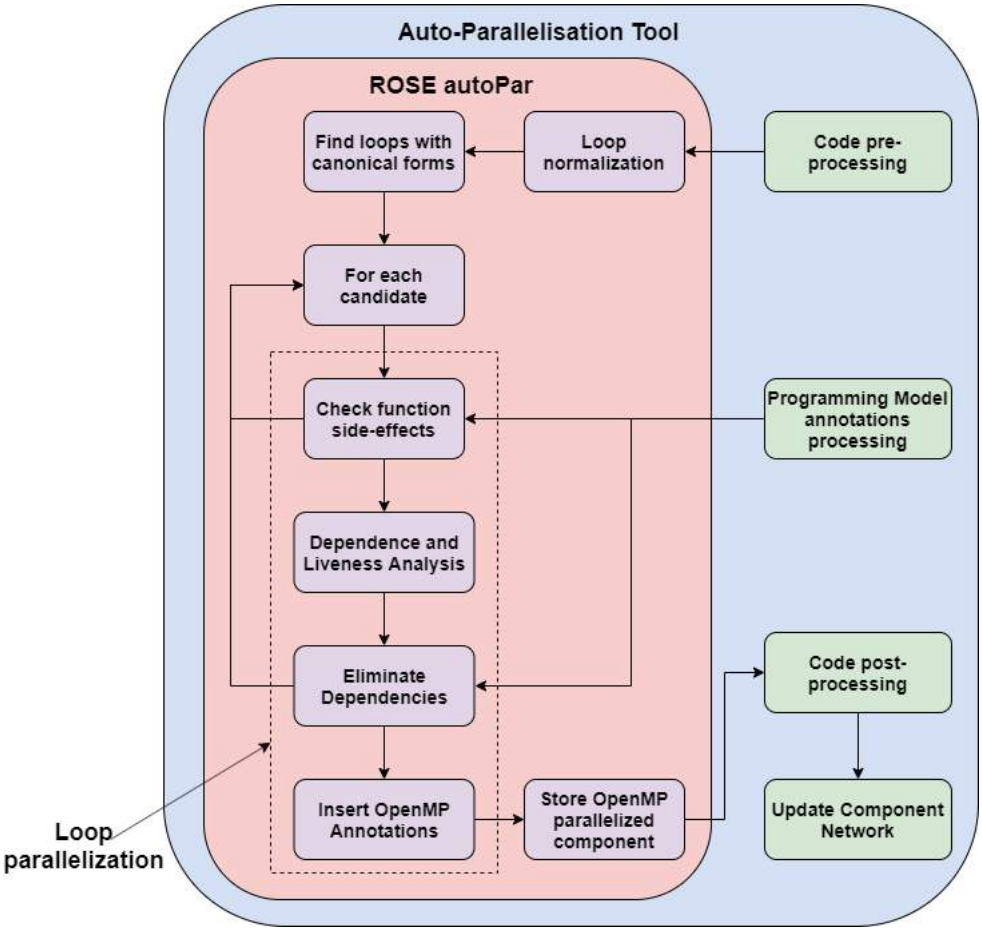


Figure 5.3: Auto-parallelization Tool

Pre-processing with the Programming Model

Characteristics of the programming model assist the code analysis stage. In specific, the component isolation of the model means that there are data dependencies discovered during the analysis that don't affect the potential concurrency. For this reason, specific annotations, included in the programming model, are used to extend the tool's "understanding" of the data flow and resolve many of those dependencies. For instance, the annotation #pragma aeolus static-vectors is used to characterize a vector of static size, resolving in that way lots of the dependences that occur due to the variable size of the vector. In order to exploit these indications, a big part of the tool's functionality concerns the pre-processing of the code, aiming to resolve a lot of its existing dependences. The tool uses this interaction with the user to resolve some of the difficulties that the latest techniques in automatic parallelization have yet to resolve.

OpenMP Parallelization

The code analysis that is run by the tool identifies parallelizable regions in the code and produces a parallelized version, containing OpenMP annotations that allow the use of multiple threads during the execution of the component on a multicore CPU system. The generated parallelized versions are stored alongside the original component.

After pre-processing the component, the tool applies some of the latest compilation techniques [12] to model the source code. An extended direction matrix (EDM) dependence representation is used to cover non-common loop nests that surround only one of the two statements in order to handle non-perfectly nested loops. For array accesses within loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables. Introducing such models assists the localization of otherwise untraceable major parallelization capabilities.

The stages included in the analysis [13] are the following:

- Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
- Normalize loops, including those using iterators.
- Find candidate array computation loops with canonical forms (for omp for) or loops and functions operating on individual elements (for omp task).
- For each candidate:
  - Skip the target if there are function calls without known semantics or side effects.
  - Call dependence analysis and liveness analysis.
  - Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
  - Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.
  - Insert the corresponding OpenMP constructs if no dependencies remain.
- Store the new parallelized OpenMP version locally

Finally, the component is processed for any changes that were annotated during its pre-processing in order to maintain its original functionality.

5.4.3 Implementation Details

Exploitation of the application model

A big part of the analysis concerns the pre-processing of the code according to user annotations as they are defined by the application model. In specific, the following pragmas are exploited by the tool for the facilitation of the analysis:

#pragma aeolus function no-side-effects
This pragma declares that a function can be considered as side-effect-free, meaning that it doesn't write in memory addresses outside its range. That enables the tool to ignore any dependencies originating from any calls to this function.

#pragma aeolus loop no-pointer-aliasing
This pragma can be used to declare that any pointer/array accesses in the following loop will not overlap with each other. This information is used to eliminate dependencies that occur because of indirect array accesses using values that are defined at run-time.

#pragma aeolus loop static-vectors
This pragma can be used to declare that any vector-class objects that exist in the loop retain their size during all iterations avoiding in this way dependences caused by accesses to unallocated memory addresses. Using this information, the tool considers any vectors appearing in the analysed loop as static C arrays. This indication was found to be the most useful since the vector class is used a lot in real-world C++ applications.

The ROSE Compiler

The tool uses the ROSE Compiler during its code analysis on each component or external library that is defined by the user. ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale applications. In specific, the tool employs the autoPar tool which is an implementation of automatic parallelization using OpenMP and can automatically insert OpenMP 3.0 directives into input serial C/C++ code.

The integration with the ROSE Compiler has fitted well on the final structure of the tool resulting in the adoption of the latest technologies available, while also satisfying the constraints regarding the use case requirements. With support from the programming model, the tool is able to extend the functionality of the autoPar tool, eliminating lots of dependences that make real-world applications difficult to parallelize, while still using advanced mathematical models to conduct the code analysis.

5.4.4 Extended functionality

The design of the tool displays explicitly its potential for extending its support among the different automatic parallelization techniques or tools that exist or may exist in the future. The components' version generation is a strictly dynamic procedure that encourages the addition of new tools in the tool chain for the continuous evolution of the application's final design and deployment on the available hardware infrastructure.

In this thesis, the support of OpenMP parallelization has been included in the design of the prototype showing only one of the cases that can be addressed by such a tool. Additional research can surely enhance the results of the tool's functionality, introducing a series of different techniques addressing a bigger range of applications as well as hardware targets that can contribute to the components' optimized execution.

Elaborating this, the toolchain that we describe in this section has been specifically designed to be modular enough, allowing additional tools to be used as further advancements are made. Thus, for including more parallelized versions of the components external tools can simply store the new versions at the corresponding directory and modify accordingly the component network as described above.

5.4.5 Heterogeneous Deployment APIs

Besides from the communication between different components, communication with different types of devices is also needed to enhance the support of heterogeneous system deployment. This is an approach that was not investigated in depth during the development of this thesis, and thus, can support only very simplistic cases. It is included here, only as a proof of concept, to support the diversity of techniques that can be used in the context of the framework's usage.

A user interface for GPU support

An effort to support this kind of systems is concentrated on a particular API that is meant to assist the user with data movements to and from the device as well as with the deployment of the kernel code that enables computing acceleration. This API includes a set of pragma annotations specifically designed to allow the inclusion of input/output variables as well as the kernel functions that need to be called. As a result, the corresponding CUDA functions are generated, completing the component source code for deployment on a GPU.

The format of the pragmas that will be used is defined as follows:

Defining a kernel input:
#pragma aeolus kernel <function> kernelin <name> type=<type> size=<size>

Defining a kernel output:
#pragma aeolus kernel <function> kernelout <name> type=<type> size=<size>

Deployment on GPU devices

With the use of this API the tool is able to transform the component source code producing a version of the corresponding component deployable on a GPU. Below, an example of the code transformations is displayed for understanding the results of such an analysis. Original form of the component code:

```
int *vec_add(int *a, int *b) {
    int c[10000];

    #pragma aeolus kernel vec_add_CUDA kernelin a type=int size=10000
    #pragma aeolus kernel vec_add_CUDA kernelin b type=int size=10000

    for(int i=0; i<10000; i++)
        c[i] = a[i] + b[i];

    #pragma aeolus kernel vec_add_CUDA kernelout c type=int size=10000

    return c;
}
```

Here, the use of the GPU API is visible, defining the arrays a and b as inputs and the array c as output for the kernel function vec_add_CUDA(). The results of the analysis and the generation of the GPU version of the component is displayed below:

```
int *vec_add(int *a, int *b) {

int c[10000];

    //Declare pointers for the GPU to use
int *dev_a, *dev_b, *dev_c;

    //Allocate memory on the GPU
    cudaMalloc((void**)&dev_a, 10000*sizeof(int));
    cudaMalloc((void**)&dev_b, 10000*sizeof(int));
    cudaMalloc((void**)&dev_c, 10000*sizeof(int));

    //Copy the arrays a and b to the GPU
    cudaMemcpy( dev_a, a, 10000*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy( dev_b, b, 10000*sizeof(int), cudaMemcpyHostToDevice);

    //Launch the kernel with 100/128 thread blocks of 128 threads
    dim3 block(128);
    dim3 grid((10000+ 127)/128);
    vec_add_CUDA<<<grid, block>>>(dev_a, dev_b, dev_c, 10000);

    //Copy the array c back from the GPU to the CPU
    cudaMemcpy( c, dev_c, 10000*sizeof(int), cudaMemcpyDeviceToHost);

    //Free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return c;
}
```

5.5 Technique Selection

5.5.1 Overview

The purpose of generating a bunch of different parallelized versions of the components as well as different versions of the communication APIs resides on the optimization of the application's performance depending on the nature of the components and the available hardware targets. In order to achieve this goal, there is a direct need for a decision mechanism that's going to select the suitable implementations that, combined with the given deployment plan, will result to increased performance and minimized power and memory footprint.

5.5.2 Component version selection

At the first stage of its functionality, the tool selects the best-fitting parallelization technology for each component (i.e. OpenMP for multi-core CPU systems, CUDA or OpenCL for GPUs, IP cores for FPGAs etc.) according to the deployment plan and creates the actual set of components that will be used at deployment.

In this thesis, the technologies that are supported by the AEOLUS automatic parallelization tool include only the OpenMP version of the components. However, as described in section 5.4.5, an API is provided for data exchanging and deployment on GPU targets, which allows the exploitation of CUDA-developed kernels provided by the user. Thus, in the case that a component is planned to run on a CPU system the OpenMP version is selected, enabling the possibility of executing the component in parallel using multiple threads. On the other hand, if the deployment plan assigns the component to a GPU to accelerate its execution the CUDA version (provided by the user) is chosen, assuming the API has been successfully used in the component code.

```
public void TechSel {
    boolean gpu_flag=false, fpga_flag=false;
    for(int i=0; i<Components.size(); i++) {
        if(Components.get(i).getLoops().size() == 0) {
            Components.get(i).setFinalVersion“(”OpenMP);
            continue;
        }
        else {
            for(int j=0; j<Components.get(i).getLoops().size(); j++) {
                if(Components.get(i).getLoops().get(j).RunsOnGpu()) {
                    gpu_flag = true;
                }
                else if( Components.get(i).getLoops().get(j).RunsOnFpga) {
                    fpga_flag = true;
                }
            }
        }
        if(gpu_flag && !fpga_flag) {
            Components.get(i).setFinalVersion“(”CUDA);
        }
        else if(!gpu_flag && fpga_flag) {
            Components.get(i).setFinalVersion“(”VHDL);
        }
    }
}
```

The process of the selection described above concerns the translation of the deployment plan from a lower-level hardware representation of the components to a higher-level software one. The final versions of the components are stored separately from the other available versions and are ready to be refined and deployed by the Deployment Manager (section 5.6).

5.5.3 Deployment Selection

The application model defines that a set of components will run independently and in parallel, while interacting with each other, to constitute the application's overall functionality. This a pretty generic definition which allows some flexibility to the technology that can be used for the deployment of the application, as well as for the communication between the components. So, a decision mechanism is necessary to decide the technology patterns that its implementation should follow.

However, this decision has not such a wide range of choices due to its high dependability to the deployment plan which defines their location over the hardware platform. Components that are assigned to the same processing node use the same address space, hence libraries like pthreads and OpenMP can be used parallel execution in separate threads. On the other side, components that are assigned to run on different nodes need to use other types of libraries to function in an environment where nodes work in a different address space. Usage of an MPI implementation like MPICH and OpenMPI is a common solution in such cases especially for low-level languages like C.

In the context of this thesis, we will discuss an implementation of the introduced system using only pthreads and OpenMPI (see section 5.6), but as with the rest of the AEOLUS modules, only core functionalities have been developed, so that future extensions can be added expanding the support of the system for more libraries that may optimize the results of the deployment. The strategy we will be following concerning the technology selection of the deployment is strictly related to the deployment plan's instructions about the communication objects. Every communication object is accessed by a set of components which can execute on the same or another memory space, thus the distribution of these entities across the computing continuum, defines their implementation nature. In specific, two different cases are studied concerning the implementation of a communication object:

- **All the components related to the object are executed on a shared-memory system**
  Then data can be safely exchanged as a shared structure in memory. Components are executed as separate threads using the pthreads library along with the POSIX standards for accessing the data.

- **Components related to the object are executed on different memory systems**
  A message passing protocol is necessary for the transfer of the data. Components are executed as separate MPI processes using the OpenMPI library, while the APIs provided by MPI are also used for accessing the data.

According to the above decisions, the corresponding implementations of the APIs are selected and stored along with the final version of the components.

5.5.4 Extended functionality

The design of this module displays explicitly its potential for extending its support among the different techniques or libraries that exist or may exist in the future. Its extension goes hand-to-hand with the automatic parallelization tool which also leaves room for future additions, resulting in more modified versions of the original components to choose from. As with the inclusion of the CUDA versions supported by the GPU API, more versions can also be provided by the user assuming that there is the corresponding support for their deployment on the available hardware infrastructure.

5.6 Deployment Manager

5.6.1 Overview

Let's assume an application that follows the model that was described in section 4 and a complex heterogeneous hardware platform that includes different devices and architectures where the application is to be executed on. In heterogeneous computing, the programmer is responsible to implement the

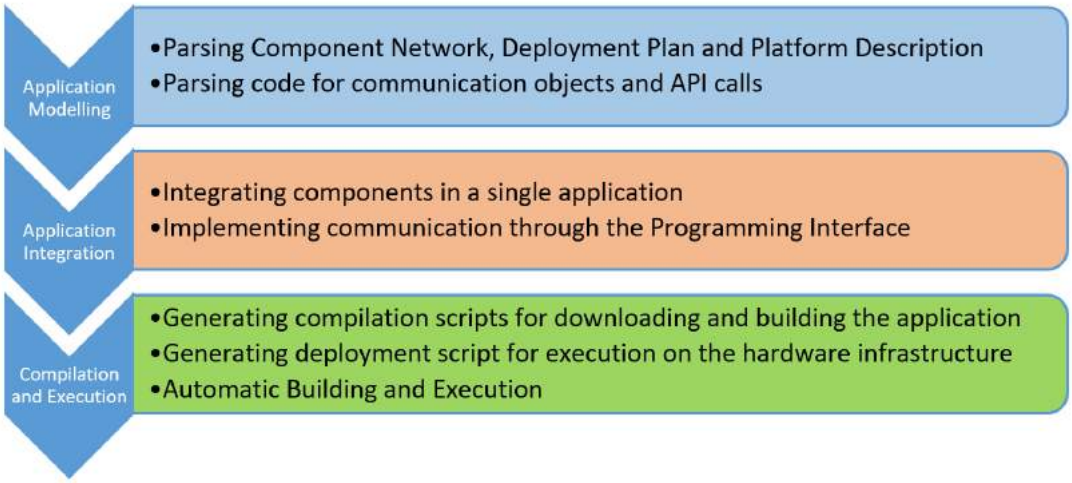


Figure 5.4: Deployment Manager functionality flow

application deployment on the different hardware components, while also ensuring for the communication needs that may occur between the software components or the components and the external devices. Taking into account all the different libraries and tools that need to be used and the different configurations that need to take place in order to deploy an application on a heterogeneous environment, it can be a very complex and time-consuming procedure.

The aim of AEOLUS is to minimize the effort and knowledge that are required from the programmer to develop an application running on heterogeneous systems. The idea in discussion here is about a separate module of the framework's architecture that will pose as the programmer making all the necessary configurations in order to deploy the components in parallel with respect to the communication between them. For implementing such an agent, the code where the necessary structures and functions are defined needs to be generated at compile-time, so that the whole application can be coordinated at run-time. But to reduce the effort contributed by a human agent means the creation of an artificial one that is responsible for all the technical parts of the development that the user cannot or does not have to be.

In this section, the functionality of a tool referred to as the Deployment Manager will be discussed. The different adjustments and code refinements are executed as a final stage of the application's development following the analysis and results of the previously discussed tools and application model as defined by the user. Additionally, a set of scripts corresponding to the aforementioned procedure is generated, implementing the actual compilation and deployment of the components on the available hardware.

5.6.2 Design Specifications and Interfaces

As already mentioned, the Deployment Manager is responsible for a series of actions regarding the integration of the components and their automatic compilation/execution in order to facilitate the deployment procedure. In this section, these steps are going to be described in detail, while in the next section some lower-level implementation details are going to be included as well. In Figure 5.4, the functionality of the tool is illustrated.

As shown in Figure 5.4, the tool flow of the Deployment Manager can be split into three stages. First, there is the modelling of the application where a set of Java classes are used to describe the components, the way they communicate with each other, their mapping on the hardware platform and information about the data or signals they need to exchange. Second, the generation of the necessary files, which enable the integration of the components with each other, will take place implementing in this way the communication between the different parts of the application. Finally, a set of scripts

will be generated for building and placing the binaries on the corresponding machines.

Application Modelling

In order to orchestrate the deployment as instructed by the Deployment Plan and the Technique Selection module, the Deployment Manager needs to collect all the necessary information to model the application per component accompanied by the corresponding communication objects. This information is suitably encapsulated in the Component Network and Deployment Plan XML files as well as inside the source code in the form of pragma annotations.

1. Analyzing XML modelling files

First, the necessary information is extracted from the Component Network in order for an application model to be constructed using a set of Java classes. Information like the components' names, source files and paths on the Repository is obtained, along with the components' positioning inside the component network, meaning their interactions with other components and the corresponding communication objects. Additionally, basic information about the communication objects is also obtained in order to include them in the network. Attributes that characterize these communication objects like their names, size, source and target are extracted from the Component Network in order to generate the corresponding entities in the model.

The Platform Description is also analysed for collecting the data concerning the hardware components that are available. The design of the system architecture is included in the application model as well as information concerning each individual machine or device. So, attributes like frequency, bandwidth, cache memory, core number, IP address are gathered for the coordination of the deployment on the hardware infrastructure. The correlation between the software components and their communication objects, and the hardware components of the platform is achieved with the analysis of the Deployment Plan. Each component and communication object is mapped on a specific hardware resource, thus this connection is also included in the application model completing its structure.

2. Communication object identification

AEOLUS is responsible to create a common environment for the components to be executed in, at a higher logical level than the one that the components work in. All shared data are located in the namespace that is shared between all components that constitute the application. If two shared data items are named the same, they are treated as the same item by the framework. The shared data namespace is defined as the set of shared data items across all of the components of the system. This should not be confused with namespaces in the C language which are confined to an individual component.

```
1 // Declaring a communication object called the_data, linked to the // local
    variable data
2 #pragma aeolus shared in the_data offset=896
3 uint8_t data[128];
```

In the example above, the_data is in the shared data namespace global to the entire system, data is local to the C component under the normal scoping rules of C.

In order to identify the communication data and link the local variables that refer to the same communication objects, a pre-compiling pass of the code will initiate the analysis. In specific, a couple of pragma annotations provided by the programming model will be exploited, so that the communication objects can be declared in a higher-level environment. When a communication object is declared, the corresponding information is declared as well, like its size, the name of the local variable that the data is stored in, the protocol that is used for its transfer etc. According to the protocol that's defined for a communication object, the way that this object is accessed changes, while the consistency requirements of the data is defined as well. Some ensure sequential consistency or stronger, whilst others define no consistency and the programmer must manually force synchronization.

The application model is completed with the information extracted from the source code, thus the Deployment Manager can proceed with the implementation of the actual deployment. The architecture

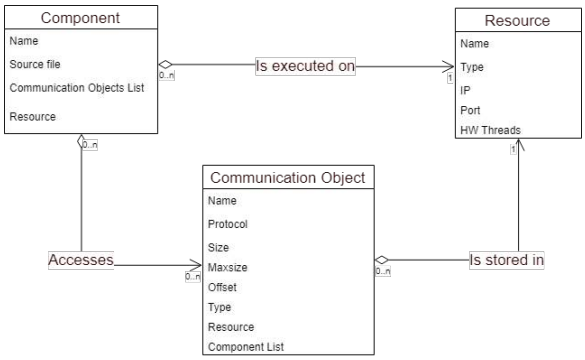


Figure 5.5: Application Model schema

of the model that has been created follows the structure of the diagram in Figure 5.5.

Constructing a multi-process application

There are multiple libraries that can be used for the parallel execution of the components. Guided by the library selection of the Technique Selection module, the Deployment Manager takes up to implement the coordinated execution of the components, along with the communication that is needed between them. The Programming Model, guarantees the independence of the components and has been specifically designed like that, so that the communication methods that are used can be chosen among a variety of options including different libraries, technologies and protocols. So, the components should be able to run as independent processes that can communicate over the network, via socket mechanisms or even using local or cloud storage for talking to each other through the Programming Interface. The different implementations can differ according to the different requirements that are encountered during development.

AEOLUS is designed to use the variety of tools that have been developed in order to optimize performance according to multidimensional requirements such as execution time and power consumption. In order to follow this incentive, the Deployment Manager implements the deployment of the components with respect to these requirements. For this purpose, the Programming Interface has adopted the MPI protocol for execution on distributed-memory systems and a set of thread-spawning and synchronization techniques for further parallelization on each node and coordination between them.

The architecture of the application is designed to follow the SPMD model to execute the different tasks that are requested by the user, in parallel. A naive (and easy to follow) implementation of the components' deployment fully adopts this model and implements the top-right case from Figure 5.6, in which the program spawns multiple threads to initiate and execute the components. This model can only be implemented in the case of a shared-memory system, this is when the user has decided that all the components should be executed on the same node/SMP.

In a similarly simple case when the user has instructed the application to be executed on a more complex environment than a single machine (e.g. to use different nodes of a cluster system), a message-passing protocol is needed for exchanging data in a distributed memory space, so the MPI protocol is adopted. This is the bottom-left case in Figure 5.6. In this case, the different components are executed each one as a separate MPI process, according to the deployment plan, all running in parallel. The Programming Interface implementation uses the corresponding MPI functions to transfer data and synchronization signals between the components. The Deployment Manager is responsible to generate the necessary files to bind the components into an MPI application and declare the structures that are needed for implementing the communication. These files are described in detail in the 'Implementation' section.

A more complex implementation will try to reduce the overhead caused by the creation of different MPI processes, while still exploiting the multi-core capabilities of the available machines. For this purpose, the execution of components running on the same memory space is implemented as different

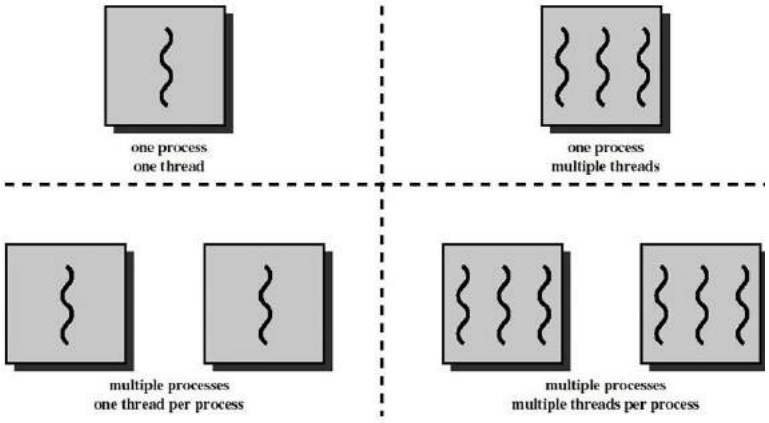


Figure 5.6: Different implementations of an SPMD program using threads and processes

threads of the same process. The result is similar to the bottom-right case in Figure 5.6. A set of MPI processes is initiated (one for each different machine) each one of which spawns a number of threads for every component that is mapped on its memory space. This is the most generic case of the above, where the parallelization of the different components is achieved in a local-memory level from a thread-based perspective, but also in a distributed-memory level from a multi-process one.

All approaches have their own benefits and drawbacks concerning including additional time overhead, power consumption or memory footprint. The best approach is to be decided based on the different designs and requirements of the application that is developed. That is why AEOLUS encourages different deployment iterations for testing different mappings and techniques, optimizing the system's performance by using multi-objective criteria.

Integration with the Programming Interface

The implementation of the Programming Interface is highly dependent on the implementation model that is selected for the application like explained in the previous paragraph. That is why during the integration of the application components, the necessary structures are generated as well, following the patterns suggested by Technique Selection in order to implement the communication between the components in the hardware environment that is indicated by the Deployment Plan. Details about the aforementioned structures that are used can be found at the "Implementation" section.

Binaries generation and deployment

The deployment of the components on the given hardware infrastructure is the core function of the Deployment Manager. While this is true, the actual deployment is an action that is assigned to the user via a set of scripts that are generated by the tool. Guided by the Deployment Plan, the DM creates the compilation scripts in order to build the application binary files on the corresponding machines/devices when the user selects to execute them. The compilation procedure continues with respect to the minimization of unnecessary binaries generation and file transfers from or to the Repository. The following steps show the actions that are completed by the scripts' execution.

1. Compilation Script

1. Check for the necessary binaries on the target machines.
2. If not found on machines
   (a) Check on Repository.

(b) If found on Repository
    i. Get binary from Repository
(c) Else
    i. Get project source folder from Repository
    ii. Compile source files and generate corresponding binary
    iii. Put on Repository

3. Get necessary input files (check if they already exist on the machines)

Notice that due to some files' huge size, unnecessary transfers are avoided by checking locally for the files when needed. This is for both binaries and input files.

Additionally, a Checksum functionality has been included in the procedure. In case the requested file is found locally but with a different checksum value than the one on the Repository, the file is considered corrupted and is discarded. This is for both binaries and input files.

2. Deployment Script

A deployment script is also generated for the initiation of the components' execution on the heterogeneous hardware architecture. This script should contain the deployment directives about where to execute each component. Due to the locality of the application that is promoted by the programming model, the Deployment Manager should make all possible efforts to guarantee for the software components' bond to the hardware components in the hardware platform. For this purpose, the Open-MPI library provides specific functionalities during execution, which are fully exploited by AEOLUS to bind specific processes to specific processor cores, sockets etc.

The aforementioned scripts are available to the user for execution on the remote machine where the Deployment Manager is executed.

Interaction with other tools

As with all the AEOLUS tools, the Deployment Manager is part of the overall architecture and needs to communicate with the other tools to complete its functionality. Its positioning in the platform is shown in Figure 5.7.

As described in Figure 5.7, the tool itself only needs to interact with the Repository, obtaining the files needed for the deployment. These include:

- The Platform Description, where the hardware infrastructure is defined
- The Component Network, where the structure of the application is described
- The Deployment Plan, where the mapping designed by the Multi-Objective Mapper is described
- The source files residing on the Repository

The actual interactions that the tool makes with the other modules occur through the scripts that are generated and includes several interactions with the Repository for getting/putting binaries and input/output files.

5.6.3 Implementation Details

Communication Object Identification

The communication objects that are used by the application have to be declared like any other variable. For a communication object declaration, a pragma annotation is used defining the protocol of the object, its name etc.

The pragma annotations are described here in detail:

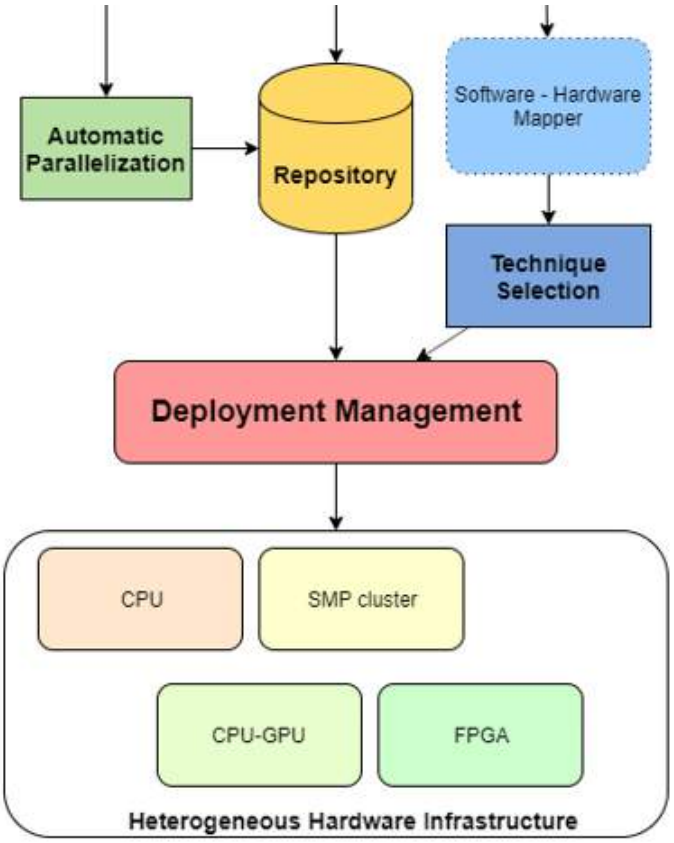


Figure 5.7: Deployment Manager positioning in the AEOLUS architecture

#pragma <queue | shared | signal | mutex> <in | out | inout> name maxsize=maxsize offset=offset
By declaring an object using this pragma, the local variable following the annotation is linked with the object. The size and type of the object is defined by the corresponding attributes of the local variable.

Info included in the local variable declaration:

- Local variable name
- Object size
- Object type

In the pragma annotation, the following attributes are also defined:

- Communication object name
- Protocol (queue or shared)
- Direction of the object (used as input, output or both)
- Maximum amount of objects that can be inserted in the queue (only for queue protocol objects) - optional
- Offset for accessing specific data of an array stored in the shared memory. Helpful for avoiding moving big chunks of data between components for no real reason. (only for shared protocol objects) - optional

In the following example, some of the protocol functions that were defined in the Programming Interface are used between the components CA and CB to assist the understanding of the aforementioned pragma annotations. In every component's environment a communication object declaration (a pragma) takes place, in which it is visible that the name is the same for the objects that refer to the same item, while the name of the local variables are irrelevant to each other. Let's examine the object declared as a shared-memory item using the identifier 'the_data'. This means that the name 'the_data' refers to the same array of 1024 items and of type uint8_t, declared to exist in the memory, shared between the two components. The local variables using the name 'data' are two different variables existing in different ranges of the application, but defined to hold the same chunk of data using the shared protocol.

```
// Component A

#pragma aeolus signal out ready
bool startb;

#pragma aeolus shared out the_data
uint8_t data[1024];

#pragma aeolus queue in sum
uint32_t input_sum;

void CB() {
   construct_input_data(data);
   aeolus_synchronize(data,1); // 1 is for updating the shared memory
   aeolus_signal(&startb);
   aeolus_queue_get(&input_sum);
}

// Component B

#pragma aeolus signal in ready
bool startme;

// Getting the last 128 items from the ''the_data comm. object
#pragma aeolus shared in the_data offset=896
uint8_t data[128];

#pragma aeolus queue out sum
uint32_t output_sum;

void CB() {
   while(true) {
      aeolus_wait(&startme);
      aeolus_synchronize(data,0); // 0 is for updating the local memory
      for(int i=0; i<128; i++)
         output_sum += data[i];
      aeolus_queue_put(&output_sum);
   }
}
```

In the example above, the application uses the shared protocol to transfer data from component CA to component CB. The call to the aeolus_synchronize() function with the direction attribute set to the value '1' updates the shared memory between the two components with the values of the data variable in component CA, while on the other side component CB calls aeolus_synchronize() with the direction attribute set to '0' to update the local memory with the values in the shared memory. In order to guarantee for the concurrency of the data, the programmer has used the signal protocol to coordinate the transactions between the components. The queue protocol is also used for moving the result of the calculations made in CB back to CA. No synchronization actions are needed here since

the queue protocol can guarantee for the consistency of the data.

The responsibility of the Deployment Manager during this stage resides on parsing the code for locating the aforementioned pragmas in order to complete the application model in a source-code level, as it was designed by the programmer including the components, the communication objects and the code related to them, meaning the pragma declarations and the corresponding API calls.

During this procedure, information is extracted from the pragmas and certain modifications are made to the API calls. So, a pre-compiling phase is executed on each component's source code during which the following actions are made:

1. If a communication object declaration is identified, the corresponding information is extracted from it and the pragma is deleted. The Deployment Manager stores this information to modify accordingly any API calls that may occur after the pragma. Thus, it is important that the pragma declaration proceeds any use of the local variables connected to the object, same as with a normal variable declaration.

2. If an API call is identified, specific arguments are added to it, providing the necessary information that are going to complete its functionality.

Integration with the Programming Interface

The Programming Interface uses a set of structures to manipulate and transfer the communication objects. These structures represent the communication objects in a lower level to facilitate the implementation of the data transfers between the components. They store the necessary information in specific fields and are accessed by the APIs in order to complete their functionalities.

Objects shared in a distributed memory space use the following structures:

```
// QUEUE PROTOCOL
typedef struct aeolus_queue {
        uint8_t *qdata;
        uint32_t front, rear, count;
        uint32_t disp, maxsize, owner;
        MPI_Win win;
} aeolus_queue;

// SIGNAL PROTOCOL
typedef struct {
        int sig;
        MPI_Comm comm;
} aeolus_signal;

// MUTEX PROTOCOL
typedef struct aeolus_mutex {
        int numprocs, ID, home, tag;
        MPI_Comm comm;
        MPI_Win win;
        unsigned char *waitlist;
} aeolus_mutex;
```

While objects that live in a shared memory space use the next structures:

```
// SHARED PROTOCOL
typedef struct {
        pthread_mutex_t lock;
        int size;
        void *data;
} aeolus_shared_object;

// SIGNAL PROTOCOL
```

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t cond;
        int ready;
        int source;
        int target;
} aeolus_signal;

// MUTEX PROTOCOL
typedef struct {
        pthread_mutex_t lock;
} aeolus_mutex;

// QUEUE PROTOCOL
typedef struct phnode {
        void* item;
        struct phnode *next;
} aeolusQnode;

typedef struct {
        aeolusQnode *front;
        aeolusQnode *rear;
        pthread_mutex_t lock;
        volatile int count;
} aeolus_queue;
```

In order to provide transparency to the user as well as configurability to the implementation of the application, a simple interface (see section 4.1.3) is used by the programmer to invoke the corresponding protocol functions. According to the Deployment Plan, the API calls that are identified are modified to fit the corresponding implementations pointed out by Technique Selection. These implementations can be found in Appendix B.

Constructing a multi-process application

The implementation of the produced application will follow the Single-Program-Multiple-Data (SPMD) approach using the OpenMPI library. That means that a single program is built and run on the hardware infrastructure, following a different control-flow path for every different process-component. The MPI protocol was selected for its efficiency in both UMA and NUMA architectures due to the memory locality strategies that it follows, as well as for its application on heterogeneous environments.

In particular, an initialization function stands as the starting point of the application, where all the necessary structures are allocated, including the communication objects, accompanied by a finalization function that deallocates any used space and ends execution. A common header file is used for creating the AEOLUS environment, holding the higher-level information that's is needed for the implementation. For the generation of these source files, the application model that has been created in the previous stage is used for providing the corresponding information (e.g. local variables, communication object info etc).

An example of the initialization function is shown in the following segment:

```
#include "aeolus.h"

void invoke_component(int id) {
    if(id == 0) {
        CA();
    }
    else if(id == 1) {
        CB();
    }
    return;
}
```

```
12
13 void initialize(int *argc, char*** argv) {
14         MPI_Init(argc, argv);
15         MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16         MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
17         buf = (int *)malloc(10*sizeof(int));
18         MPI_Win_create(buf, 10*sizeof(int), sizeof(int), MPI_INFO_NULL,
       MPI_COMM_WORLD, &win);
19         signal = aeolus_signal_init(MPI_COMM_WORLD);
20         aeolus_mutex_init(&mutex0,0);
21 }
22
23 void finalize() {
24     aeolus_mutex_destroy(mutex0);
25     MPI_Win_free(&win);
26     MPI_Finalize();
27 }
28
29 int main(int argc, char** argv) {
30     initialize(&argc,&argv);
31     invoke_component(world_rank);
32     finalize();
33     return 0;
34 }
```

As illustrated above, a set of functions, generated by the Deployment Manager, sets up the AEOLUS environment by allocating space for the necessary structures and initializing the corresponding values/objects, coordinates the execution and finally clears the environment space afterwards.

The variables that are used for the implementation of the communication objects in the above segment are declared in the ‘aeolus.h’, which is displayed below:

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "signalAPI.h"
5 #include <string.h>
6 #include "mutexAPI.h"
7 #include "sharedAPI.h"
8 #include "queueAPI.h"
9
10 MPI_Win win;
11 aeolus_mutex *mutex0;
12 aeolus_signal *signal;
13 int *buf;
14 int world_rank, world_size;
15
16 void CA();
17 void CB();
```

Notice that the type of information that is stored in the declared structures differs from implementation to implementation. For a shared memory space implementation, the POSIX library is used for the APIs implementation. In this case, the file ‘aeolus.c’ looks like this:

```
1 void aeolus_init() {
2         int i, sig=0, que=0, shr=0;
3         phsignal = (aeolus_signal **) malloc(NUM_SIGNALS*sizeof(aeolus_signal *))
      ;
4         phqueue = (aeolus_queue **) malloc(NUM_QUEUES*sizeof(aeolus_queue *));
5         phshared = (aeolus_shared_object **) malloc(NUM_SHARED*sizeof(
      aeolus_shared_object *));
6
```

```
        signal_index = (int *) malloc(AEOLUS_NUMOFCOMMS*sizeof(int));
        queue_index = (int *)malloc(AEOLUS_NUMOFCOMMS*sizeof(int));
        shared_index = (int *) malloc(AEOLUS_NUMOFCOMMS*sizeof(int));

        for(i=0; i<AEOLUS_NUMOFCOMMS; i++) {
                if(commobj_type[i] == SIGNAL) {
                        phsignal[sig] = initialize_aeolus_signal(aeolus_source_id[i], aeolus_target_id[i]);
                        signal_index[i] = sig++;
                }
                else if(commobj_type[i] == QUEUE) {
                        phqueue[que] = initialize_aeolus_queue();
                        queue_index[i] = que++;
                }
                else if(commobj_type[i] == SHARED) {
                        phshared[shr] = initialize_aeolus_shared_object(commobj_datasize[shr], commobj_size[shr]);
                        shared_index[i] = shr++;
                }
        }
}
```

More complex situations can arise as described earlier, where the implementation can include a combination of the two approaches, deploying the components on both MPI processes and separate threads, optimizing deployment depending on the design of the application as well as the requirements that where defined for its execution.

Binaries generation and deployment

The scripts that are generated prepare the local directories of the hardware for deployment. The binaries or the source files are downloaded (depending on the previously used binaries to avoid recompiling the same code), compiled and placed on the corresponding locations of the target machines. If new versions of the binaries are generated, they are uploaded on the Repository for future reuse. The Makefile, residing in the 'src' directory on the Repository, is also modified to include the corresponding compiler wrappers (nvcc, mpicc), necessary libraries (OpenMP, pthreads) and the modified component source files. Below an example of the compilation script is displayed:

```

PROJECT_NAME="test_project";
LOCAL_DIR="~/AEOLUS/Repository/${PROJECT_NAME}/DM";
BIN_PATH_ON_REPO="bin";
BIN_NAME="myapp-bin20181010";
FILE_PATH="${LOCAL_DIR}/${BIN_PATH_ON_REPO}/${BIN_NAME}";

cd ~/AEOLUS;

# CHECK FOR CORRESPONDING BINARY
if [ -f $FILE_PATH ]
then
        echo "File exists"
else
        # TRY GETTING BINARY FROM REPO
        response_code=`Get_file.sh ${PROJECT_NAME} ${BIN_PATH_ON_REPO}/${BIN_NAME}`;
        if [ ! "${response_code}" -eq 200 ]
        then
                # GET SOURCE FILES FROM REPO
                ~/AEOLUS/Repository/scripts/Get_dir.sh ${PROJECT_NAME} src;
                rm src.zip;
                # GENERATE BINARIES
```

```
                cd ${LOCAL_DIR}/src;
                make
                mv ${BIN_NAME} ${LOCAL_DIR}/${BIN_PATH_ON_REPO}
                cd ~/AEOLUS
                # PUT BINARY ON REPO
                Put_file.sh ${FILE_PATH} ${BIN_PATH_ON_REPO};
        fi;
fi;

INPUT_PATH_ON_REPO="inputs";
INPUT_NAME="input_data.txt";
INPUT_FILE="${LOCAL_DIR}/${INPUT_PATH_ON_REPO}/${INPUT_NAME}";

# PLACE INPUT FILES AT THE CORRESPONDING LOCATIONS
# CHECK FOR CORRESPONDING FILE
if [ -f $INPUT_FILE ]
then
        echo "File exists"
else
        # GET FILE FROM REPO
        response_code=`Get_file.sh ${PROJECT_NAME} ${INPUT_PATH_ON_REPO}/${
    INPUT_FILE}`;
fi;
```

The deployment script is responsible for the actual execution of the components on the corresponding locations. This script is designed according to the mapping that is described in the Deployment Plan and the Platform Description that provides the network location of each machine or device. The machines are accessed via SSH by the development machine, where the Deployment Manager is executed. Thus, the necessary configurations are assumed for the successful deployment of the application.

```
# EXECUTE PROGRAM
# mpirun <affinity options>
# --host <IP> -np <number of processes> <executable_name> :
# .
# .
# .
# --host <IP> -np <number of processes> <executable_name>

# example: 2 components executed on different machines
mpirun -nolocal --host 192.282.77.22 -np 1 executable :
                --host 192.151.23.34 -np 1 executable
```

OpenMPI provides a list of options that enable affinity configurations at execution. By binding specific resources to specific processes, the deployment suggested in the Deployment Plan is implemented, while additional locality optimizations allow further improvements in the exploitation of the memory resources.

Some examples of these options are noted here:

To map processes:

–map-by <foo>
Map to the specified object, defaults to socket. Supported options include *slot*, *hwthread*, *core*, *L1cache*, *L2cache*, *L3cache*, *socket*, *numa*, *board*, *node*, *sequential*, *distance*, and *ppr*. Any object can include modifiers by adding *a :* and any combination of *PE=n* (bind n processing elements to each proc), *SPAN* (load balance the processes across the allocation), *OVERSUBSCRIBE* (allow more processes on a node than processing elements), and *NOOVERSUBSCRIBE*. This includes PPR, where the pattern would be terminated by another colon to separate it from the modifiers.

-nolocal, –nolocal
Do not run any copies of the launched application on the same node as orterun is running. This option will override listing the localhost with –host or any other host-specifying mechanism.

-nooversubscribe, –nooversubscribe
Do not oversubscribe any nodes; error (without starting any processes) if the requested number of processes would cause oversubscription. This option implicitly sets ”max_slots” equal to the ”slots” value for each node.

-bynode, –bynode
Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns MPI_COMM_WORLD ranks in a round-robin, ”by node” manner.

To order processes' ranks in MPI_COMM_WORLD:

–rank-by <foo>
Rank in round-robin fashion according to the specified object, defaults to slot. Supported options include slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, and node.

For process binding:

–bind-to <foo>
Bind processes to the specified object, defaults to core. Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.

5.7 Evaluation

The evaluation of the toolchain will be based on a real-world use case that consolidates characteristics from different scientific fields to validate the results. In this section, we describe this application and how its components are suitable for testing the different tools. Additionally, we provide an overview of the tests that we executed in order to evaluate framework and the resulting parallel application in terms of a series of functional and non-functional requirements.

5.7.1 Use case description

The application that was used for the evaluation of the end-result of the toolchain is described here. Three different simulators are considered covering a wide range of characteristics that can be found in real and computationally intensive applications. These simulators contain calculations from the fields of Statistics, Image Analysis and Control-Flow Dynamics (CFD). In specific, the components and their in-between communication are shown in detail in Figure 5.8, where the application model (Component Network) of the use case is defined.

Specifically, the Statistics Simulator has been designed to include many smaller components that extensively exchange information using the Shared, Queue and Signal Protocols as defined and implemented in the context of the Programming Model. The Image Analysis simulator consists of two computationally intensive components that also use a subset of the Programming Interface to exchange data. The CFD Simulator includes a single component which implements a computationally intensive CFD calculation providing two different versions, a CPU-only and a CPU-GPU one.

The Component Network has been designed in this way to provide a variety of characteristics that are considered important for examining the different features of the tools. For example, with the Statistics Simulator we evaluate the complexity that can be supported by the Programming Model as well as the functionality of the Programming Interface protocol functions that are extensively used in

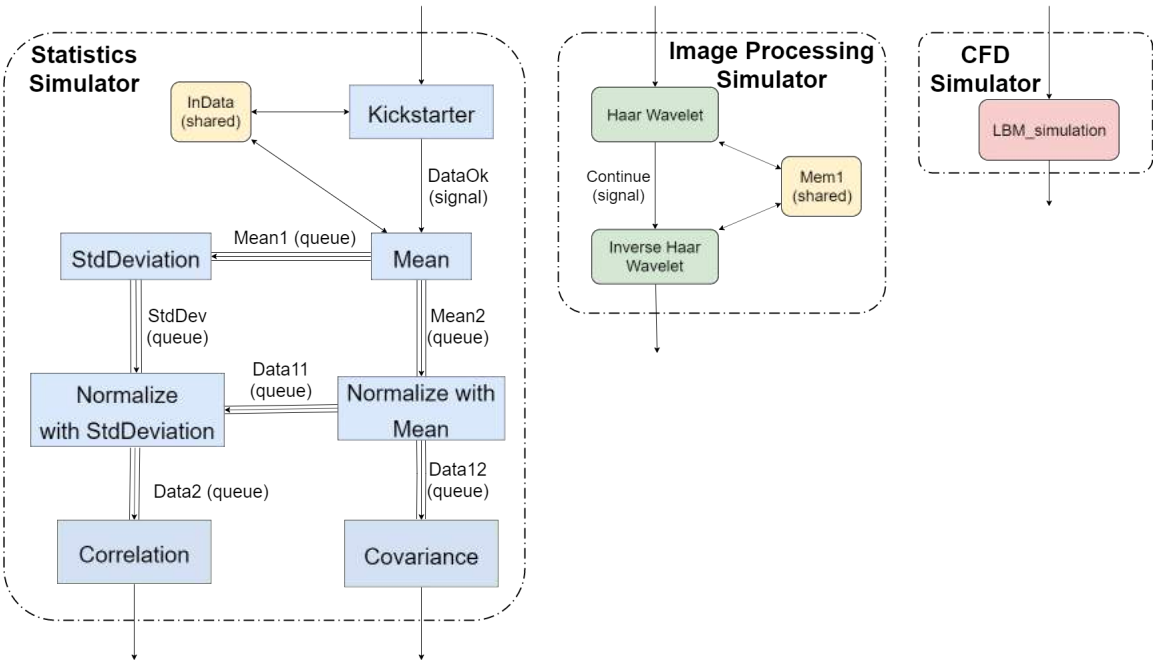


Figure 5.8: Evaluation use case application model

this part. The other two simulators focus on the computation load that is created when executing their components and are used to obtain metrics from the OpenMP parallelized versions generated by the Auto-Parallelization Tool. For the case of the CFD Simulator, the CPU-GPU version has also been included in the process to test the integration of the platform with parallelized versions other than OpenMP.

Deployment using AEOLUS

The optimisation of the application, in terms of execution time (or other objectives that we didn't study here – e.g. power efficiency), is the obvious goal of a programmer using a parallel framework. In the context of this thesis however, we focus on the architecture of the system and the resulting application after utilizing the defined features of the framework. The main indicators that we considered for the evaluation of the toolchain include the correct functionality of the resulting application and its ability to adapt on a series of different architectures. Additionally, we measured the individual performance of the three simulators in order to evaluate how these were optimized and to calculate any possible overhead that may be introduced by the framework.

Functional Testing

For the functional testing of the application, we considered a series of different deployments (mappings of components on different processes) to try the different deployment and communication implementations. In specific, we used the following set of mappings to deploy the Statistics Simulator:

Deployment 1:

Process 0: All components

Deployment 2:

Process 0: Kickstarter, Mean, Normalize_with_mean, Covariance

Process 1: StdDeviation, Normalize_with_stddev, Correlation

Deployment 3:

Process 0: Kickstarter, Mean

Process 1: StdDeviation, Normalize_with_mean, Normalize_with_stddev

Process 2: Covariance

Process 3: Correlation

Depending on the available shared environment, the corresponding communication implementations were used for data and signal exchange, while the components have been initiated as threads or processes accordingly. Performance Evaluation

AEOLUS was tested for its performance gains on two of the simulators. The programs were tested and validated with the contribution of the tool using a 12-thread Intel i7 CPU. For the tests, a series of small, medium and large input data sets where used exploiting different amounts of available threads resulting in small and large computation loads. The serial version of the application was considered as the baseline, while improvement has been calculated in comparison with the multithreaded parallel code as:

$$PerformanceImprovement = \frac{serial - parallel}{serial} 100\% \tag{5.1}$$

In general, the resulting parallel code seemed to add some overhead to the application, so when the computation load was relatively small, the overhead was apparent, while for bigger computation loads, acceleration of the application was significantly overcoming the overhead, reaching positive results up to 84%. Specific results showing improvement are displayed in Table 5.1.

Surveillance benchmark			
Input size \ # of threads	**3 threads**	**6 threads**	**12 threads**
Small	-10%	-19%	-22%
Medium	57%	70%	13%
Large	62%	80%	84%
CFD benchmark			
Input size \ # of threads	**3 threads**	**6 threads**	**12 threads**
Small	-18%	-11%	26%
Medium	-8%	4%	29%
Large	59%	8%	30%

Table 5.1: Surveillance and CFD benchmarks evaluation results

Chapter 6

Epilogue

In this chapter, we will try to summarize the work that has been done in the context of this thesis, along with the conclusions that have been made during the design of the architecture and the development of the different parts of the toolchain that was described in the previous chapters. We will also include some of the ideas that were generated by the needs of a parallel application framework such as the one described in this thesis, as well as some extensions to the toolchain that could be proven very useful to the procedure of the application's development and deployment.

6.1 Conclusion

The main pillars of this thesis can be briefly described in three different parts. For the first part, we studied some of the state of the art tools and techniques that are used for code automatic parallelisation. For the second part, we designed a parallel programming model that enables the development and deployment of a parallel application. At the third and last part of the thesis, we developed the resulting toolchain from the previous two parts that constitute a framework for the deployment of a parallel application from its development to its automatic parallelization to its integration and execution.

6.1.1 Automatic Parallelization

During our study for the automatic parallelization tools, we considered two different tools, the ROSE Compiler and PLuTo, an automatic parallelizer. We analysed the differences between the two tools and run a testing process between them to compare their efficiency. For that purpose, we used Polybench, a rich benchmark suite including a series of example code segments suitable for parallelization. The analysis of the tools showed a clear advantage for the PLuTo parallelizer, which uses efficient code transformations to not only parallelise more code segments, but also optimize memory access to minimize read-write operations. However, a notable observation is that, although PLuTo performed better that ROSE in general, the latter proved to support a wider target (since it is a tool compatible with C++), while also performing better that the former in certain occasions, so the overall gain that is actually obtained by using PLuTo over ROSE is not that big.

In order to extend our testing, we used a series of more complicated examples extracted from real world applications to extend the benchmark suite. To this regard, we considered useful to propose an extended programming model that would enable the user to identify regions in the code that may be parallelisable, but due to specific characteristics certain dependences cannot be resolved by the parallelizer. The model proved to resolve certain dependencies that were impossible to find using automatic parallelisation tools.

6.1.2 Parallel Programming Modelling

For designing a parallel programming model, we first needed to consider the general approach that is followed when designing a parallel application. In this way, we actually describe what is being taken care of by the backend that would support a programming model. This section is devoted on the design and implementation details of such a model. The Programming Model we describe is based on

the design of the application following a modular structure of software components that communicate with each other using a defined set of APIs abstracting their low-level implementation from the user. This approach allows the deployment of the components on different machines/hardware elements optimising performance according to their different characteristics.

6.1.3 The AEOLUS framework

The AEOLUS framework is the actual product of this thesis. It combines the outcome from the other two parts in order to form a toolchain that developers can exploit to design and optimize their application following the guidelines set by the Programming Model. It includes the overall architecture of the framework and the interactions between the tools, as well as their design and implementation.

The framework is constructed based on the Programming Model. The developer starts the implementation by designing the Component Network (which describes the application), based on the component based approach that is described, and the Platform Description (which describes the underlying hardware that is available). Combining those, they can choose a certain deployment and depict this into a Deployment Plan. The code can be instrumented according to the application's requirements. From there, an automatic parallelization tool can take up to produce parallelized versions of the components (using OpenMP, CUDA or other libraries) to be executed on multithreaded CPUs or other hardware. A technique selection module will then select the corresponding API implementations and component versions according to the Deployment Plan, which is then deployed by the Deployment Manager integrating, compiling and executing the application on the corresponding machine targets.

The aim of this study is to provide the design of an end-to-end solution for parallel application development for an infrastructure that can be built on, while different technologies and models are developed. Some of the tools can be replaced, others can be improved, while the overall architecture can be extended, introducing new tools into the framework that can provide a wider package of supported use cases.

6.2 Future Work

During the development and evaluation of the resulting toolchain, there was a lot of ideas that came up, ideas that would facilitate the developer's experience, as well as expanse the capabilities that are offered by the tools, while also improving the parallel application's performance. In this last section, we lay down some of these ideas that provide motivation to further extend our work on the framework which has potential for some very promising and interesting improvements.

6.2.1 Automatic Parallelization

The efforts for this thesis have been concentrated on automatic parallelisation on a multithreading scale using the OpenMP programming model. Although the available tools mostly show promising results using this model, there is a lot of ongoing work with other multicore technologies or even technologies for external hardware exploitation similar to CUDA and OpenCL. As time goes by, a lot of new emerging technologies come up, for both the programming part and the hardware one. New devices and new interfaces are going to be used to continuously accelerate applications and offload calculations that can be suited to specific hardware.

In order to encapsulate and exploit all those different solutions, the Programming Model (described in section 4) has been specifically designed to support different technologies and devices, allowing a variety of implementations and deployments to take place, so that the developer can try different solutions optimising the application's performance. As new automatic parallelisation tools are introduced, different versions of the software components can be created and considered for deployment.

To this end, the Automatic Parallelization tool can be extended to support more parallelization technologies for shared and distributed memory, or for external devices/hardware. Similarly, the

Technique Selection module can be extended to select different versions of the components for the same or different devices/machines, as well as different communication libraries for data transfers and consistency signals, according to the selected deployment.

6.2.2 Resource Distribution Optimization

As already mentioned, AEOLUS allows the deployment of the defined software application components on different hardware elements, so that the developer can try different implementations of their application aiming at performance optimisation. However, these different trial executions rely a lot on intuition and on how much the developer is familiarised with the underlying hardware. In order to actually notice some improvement, developers either need to be lucky enough to get a deployment that shows performance gains, or they really need to know what they are doing (assuming of course a large multi-component application with a large number of possible deployments).

A resource distribution tool can do exactly that, it can be developed to use experts' intuition, which is not something the average user owns, to skillfully assign the different components to specific hardware elements. According to the characteristics of the components and to the different criteria/requirements of the user (speed-up, power etc), the tool can automatically suggest or even try out different deployments that can be accurate enough and optimize over time the final implementation of the application.

6.2.3 File Repository

Although the current architecture of the framework allows the - abstract to the user - interaction between different machines regarding data and signal exchange, the user needs to be fully aware of where to put any necessary files that are used by the application components. In the context of this thesis, we have partially implemented a centralized solution for managing file operations using a local version of a file server, implementing also the corresponding APIs (see section 4.2).

There is a variety of solutions that can be considered, but in every case, support by the programming model is basic requirement. In specific, a set of APIs like the ones we describe in section 4.2 must be defined, as well as a description model including the files' basic information like their location, their size etc.

Two indicative approaches regarding the implementation are described here:

- A centralized repository server: A remote server used for file storage. This solution provides a set of APIs that allow the application to remotely access the server and read/download/upload any files that are needed.

- A distributed repository: A set of APIs developed to provide access to the components for files residing on different machines. The storage space in this case is distributed across all different devices/machines that are defined in the Platform Description. The Programming Model is responsible to abstract the user from any low-level transfers/accesses between the different machines, providing the view of a single uniform repository that he can access using the same interface as with the centralized case.

6.2.4 Performance Evaluation

For the design of the Programming Model, we considered the need of the user to monitor their application, regarding different aspects of performance (execution time, power etc). For this reason, the corresponding APIs have been designed giving the ability to the user to gather information about execution and store it wherever they prefer (file, print at the output etc.). The corresponding implementations of these functions can extend to different kinds of devices and hardware to measure different parameters to facilitate the evaluation of the application's performance at each deployment.

The information gathered from the different executions can provide very useful input for tools like the resource distribution tool we mentioned before, and to the user as well. So, the organized storage and aggregation of this data is a big deal. The aforementioned monitoring APIs can extend to provide this data automatically to a Monitoring Platform, which in turn can provide storing, analysis and visualization capabilities facilitating the overall evaluation of the application.

6.2.5 Management Interface

The management and execution of the tools as well as the development of the application following the specifications of the Programming Model can be complex tasks. A management interface to monitor and manage the different functionalities of the tools and to facilitate the development procedure through a Graphics User Interface can simplify the complexity that is received by the user. A series of additional functionalities can be introduced like this offering new exploitation capabilities of the toolchain. Some examples are enumerated below:

- Execution of the tools using a graphic interface to facilitate their usage
- Development of the application providing an interface that can remove the complexity of the Programming Model guidelines, allowing the users to easily fimiliarise themselves with the model as they go on
- Monitoring the deployment of the application at real-time, including the different stages of its analysis (parallelization, resource allocation, compilation etc.)
- Monitoring of the application's execution at real-time, displaying useful analytics regarding the different deployments while measuring also different parameters

Bibliography

[1] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*. In Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction (CC'08/ETAPS'08), Laurie Hendren (Ed.). Springer-Verlag, Berlin, Heidelberg, 132-146, 2008

[2] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988

[3] P. Feautrier. *Some efficient solutions to the affine scheduling problem: I. one-dimensional time*. Intl. J. of Parallel Programming, 21(5):313–348, 1992.

[4] A. W. Lim, G. I. Cheong, and M. S. Lam. *An affine partitioning algorithm to maximize parallelism and minimize communication*. In ACM Intl. Conf. on Supercomputing, pages 228–237, 1999.

[5] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. *Facilitating the search for compositions of program transformations*. In ACM Intl. Conf. on Supercomputing, pages 151–160, June 2005.

[6] M. Griebl, C. Lengauer, and S. Wetzel. *Code generation in the polytope model*. In IEEE PACT, pages 106–111, 1998.

[7] A. Porterlield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[8] D. Callahan, S, Carr, and K. Kennedy. *Improving register allocation for subscripted variables*. In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990.

[9] Uday Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*. International Conference on Compiler Construction (ETAPS CC), Apr 2008, Budapest, Hungary.

[10] PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. `http://plutocompiler.sourceforge.net.`

[11] Bondhugula, Uday, J. Ramanujam and P. Sadayappan. *PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System*. 2015.

[12] Q. Yi, V. Adve, and K. Kennedy. *Transforming loops to recursion for multi-level memory hierarchies*. In ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada, June 2000.

[13] M. E. Wolf and M. S. Lam. *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*. IEEE Trans. Parallel Distrib. Syst. 2, 4 (October 1991), 452-471. `https://doi.org/10.1109/71.97902`

[14] ROSE Compiler,
`http://plutocompiler.sourceforge.net`

[15] Ajkunic, E. and Fatkić, Hana and Omerovic, E. and Talic, K. and Nosovic, N. *A comparison of five parallel programming models for C++.* January 2012, 1780-1784, ISBN: 978-1-4673-2577-6

[16] A. Castelló, A. J. Peña, S. Seo, R. Mayo, P. Balaji and E. S. Quintana-Ortí *A Review of Lightweight Thread Approaches for High Performance Computing.* 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, 2016, pp. 471-480.

[17] Akhmetova, Dana and Iakymchuk, Roman and Ekeberg, Orjan and Laure *Performance Study of Multithreaded MPI and OpenMP Tasking in a Large Scientific Code.* May 2017, 10.1109/IPDPSW.2017.128, 756-765

[18] Smyk, Adam and Tudruj, Marek *Application of Mixed MPI/OpenMP Programming in a Multi SMP Cluster Computer.* Springer Berlin Heidelberg 2002, 288–296, ISBN: 978-3-540-48086-0

[19] Smith, Lorna, and Mark Bull *Development of mixed mode MPI/OpenMP applications.* Scientific Programming 9.2-3 (2001): 83-98.

[20] Jin, Haoqiang, et al *High performance computing using MPI and OpenMP on multi-core parallel systems.* Parallel Computing 37.9 (2011): 562-575.

[21] Drosinos Nikolaos, and Nectarios Koziris *Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters.* 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.. IEEE, 2004.

[22] Diaz, Javier, Camelia Munoz-Caro, and Alfonso Nino. *A survey of parallel programming models and tools in the multi and many-core era.* IEEE Transactions on parallel and distributed systems 23.8 (2012): 1369-1386.

[23] Chorley, Martin J., and David W. Walker. *Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters.* Journal of Computational Science 1.3 (2010): 168-174.

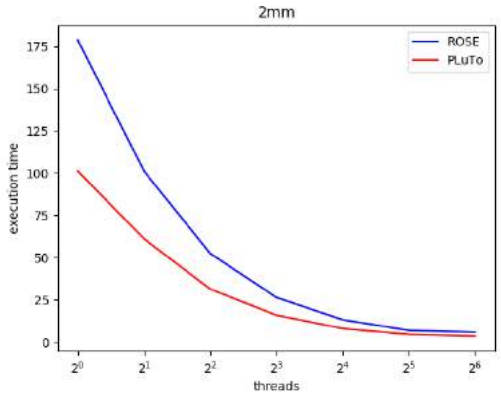
[24] Chan, Michael K., and Lan Yang. *Comparative analysis of openmp and mpi on multi-core architecture.* Proceedings of the 44th Annual Simulation Symposium. 2011.

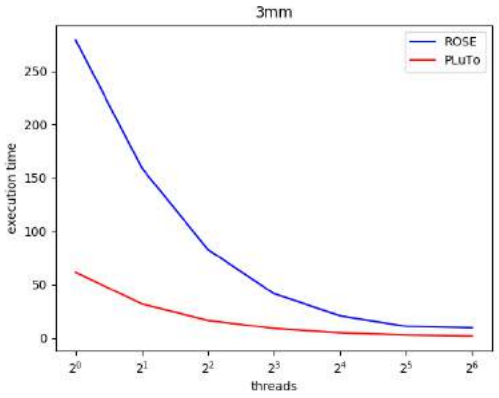
[25] Adhianto, Laksono, and Barbara Chapman. *Performance modeling of communication and computation in hybrid MPI and OpenMP applications.* Simulation Modelling Practice and Theory 15.4 (2007): 481-491.

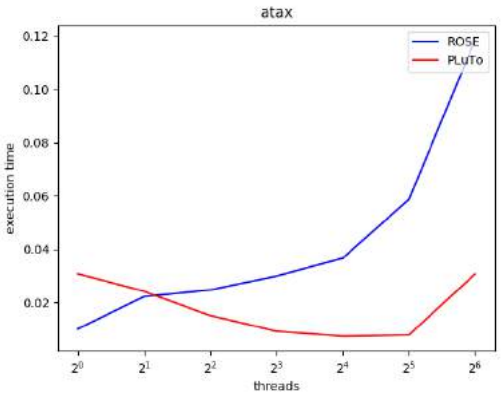
[26] Rabenseifner, Rolf and Hager, Georg and Jost, Gabriele. *Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes.* 10.1109/PDP.2009.43 (2009). 427-436.

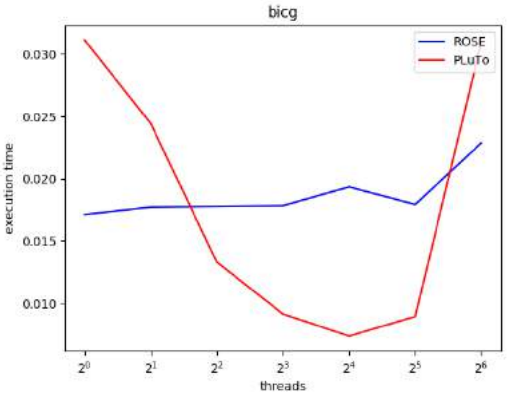
Appendix A

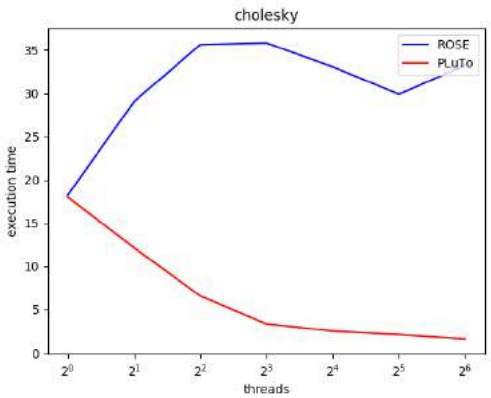
Evaluation - ’polybench’ suite test results

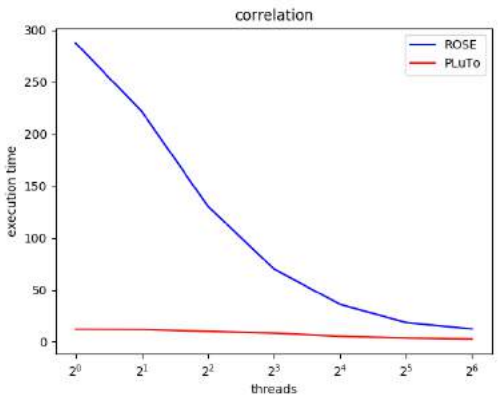


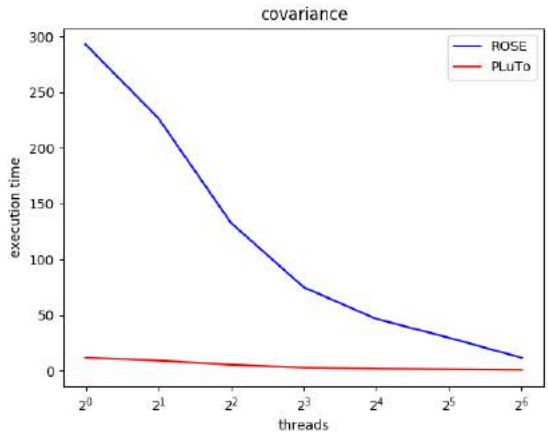


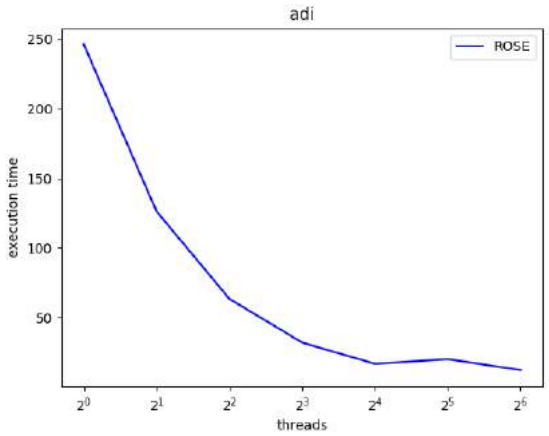


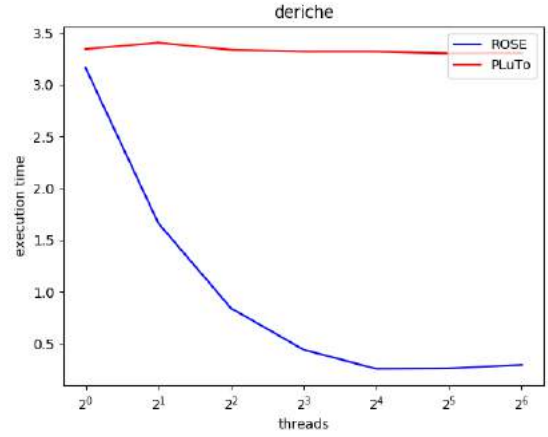


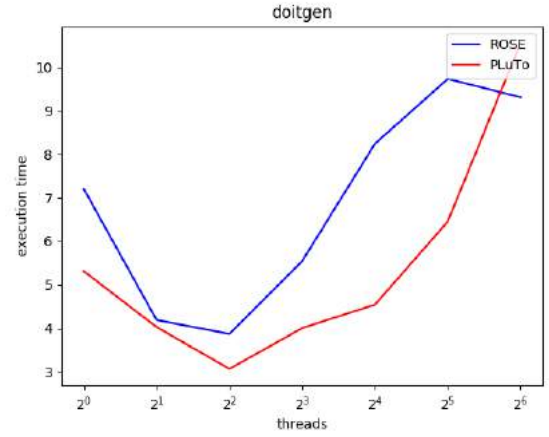


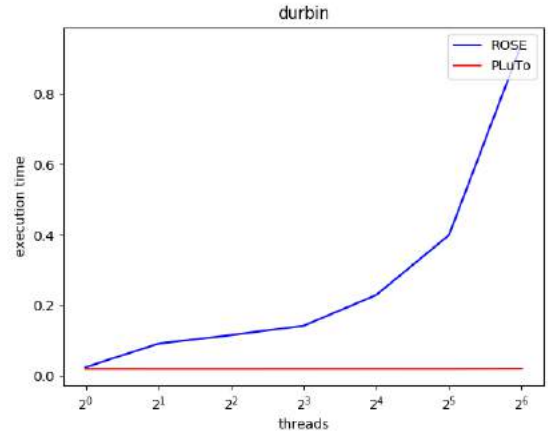


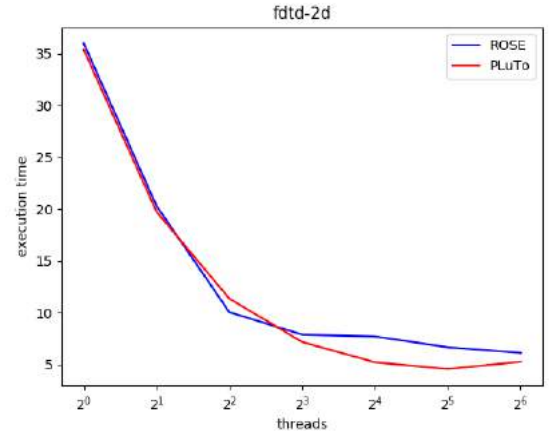

covariance
ROSE
PLuTo
execution time
threads

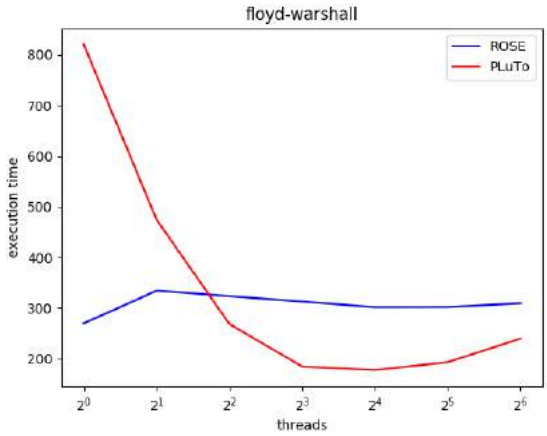

adi
ROSE
execution time
threads

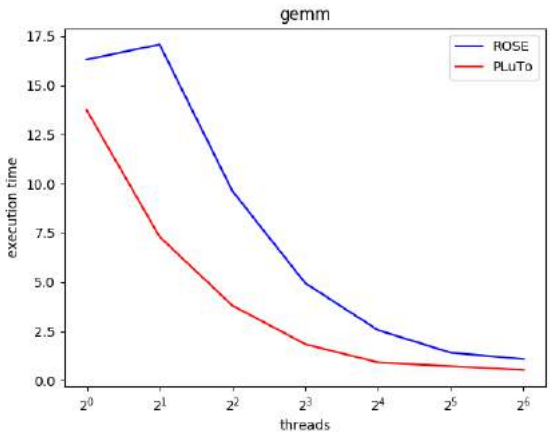

deriche
ROSE
PLuTo
execution time
threads

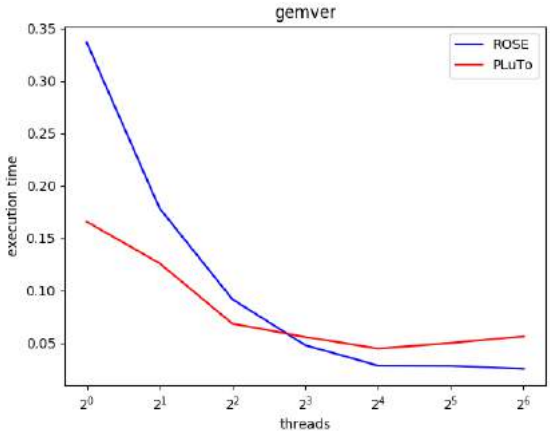

doitgen
ROSE
PLuTo
execution time
threads

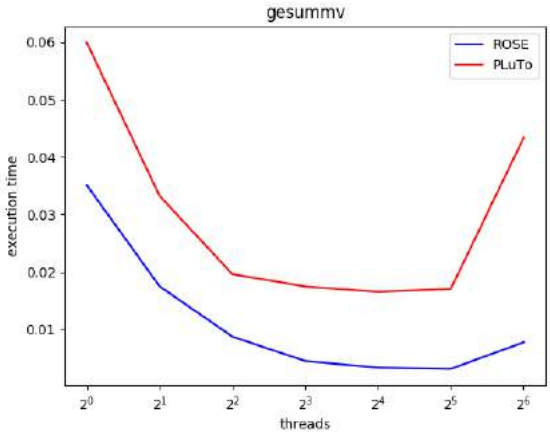

durbin
ROSE
PLuTo
execution time
threads

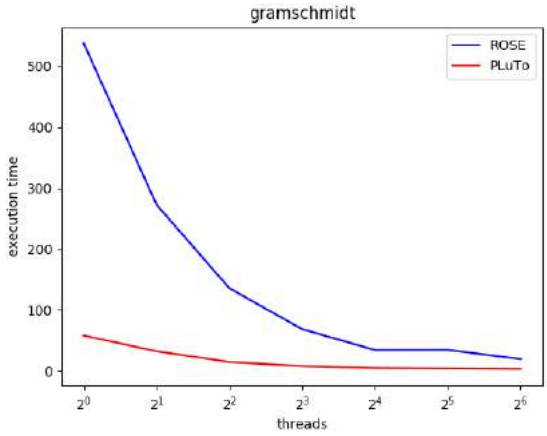

fdtd-2d
ROSE
PLuTo
execution time
threads

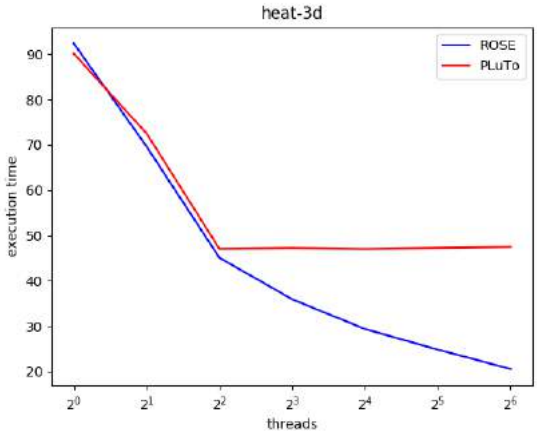

floyd-warshall
ROSE
PLuTo
execution time
threads

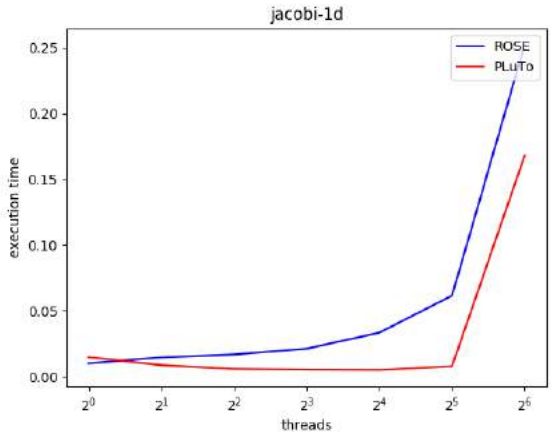

gemm
ROSE
PLuTo
execution time
threads

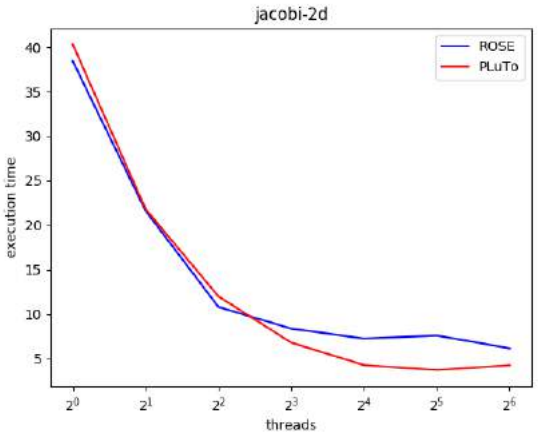

gemver
ROSE
PLuTo
execution time
threads

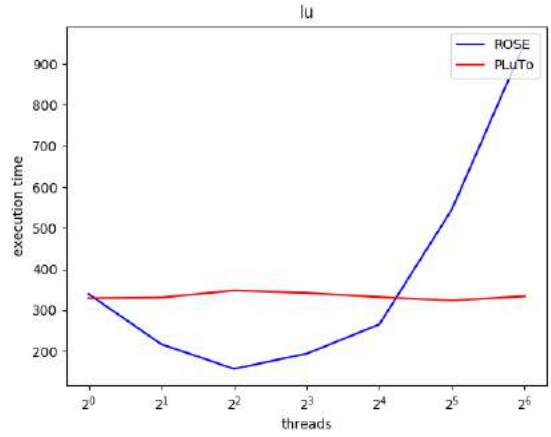

gesummv
ROSE
PLuTo
execution time
threads

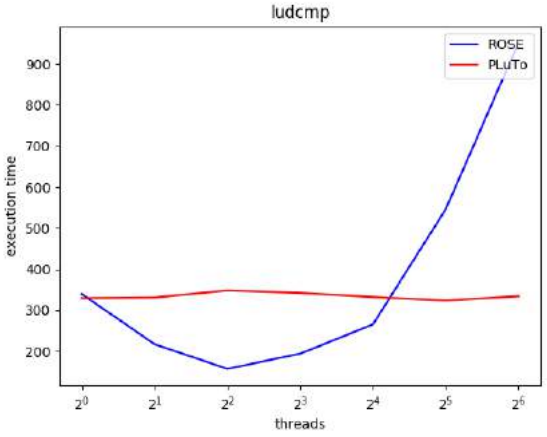

gramschmidt
ROSE
PLuTo
execution time
threads

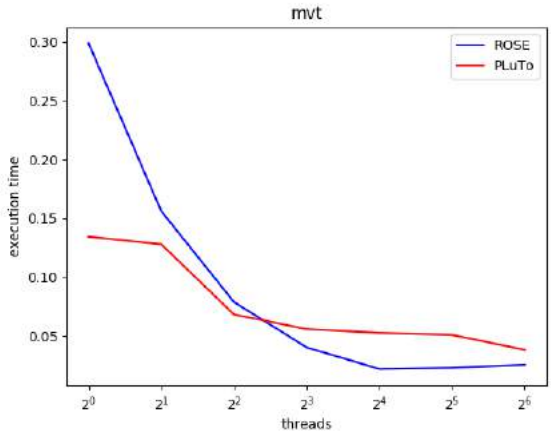

heat-3d
ROSE
PLuTo
execution time
threads

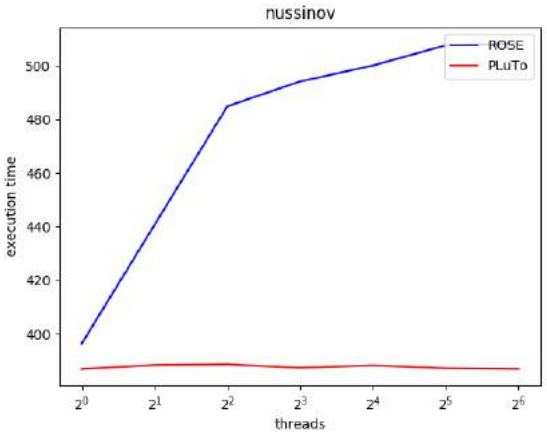

jacobi-1d
ROSE
PLuTo
execution time
0.25
0.20
0.15
0.10
0.05
0.00
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

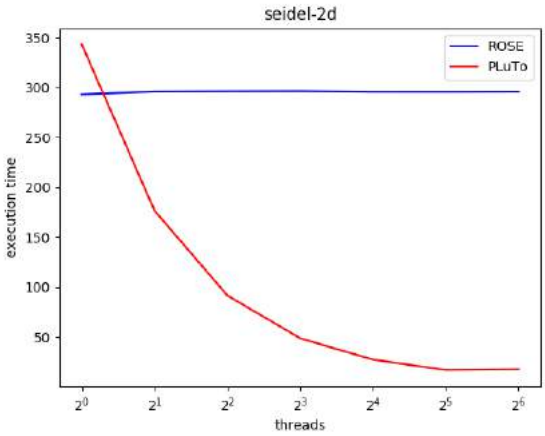

jacobi-2d
ROSE
PLuTo
execution time
40
35
30
25
20
15
10
5
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

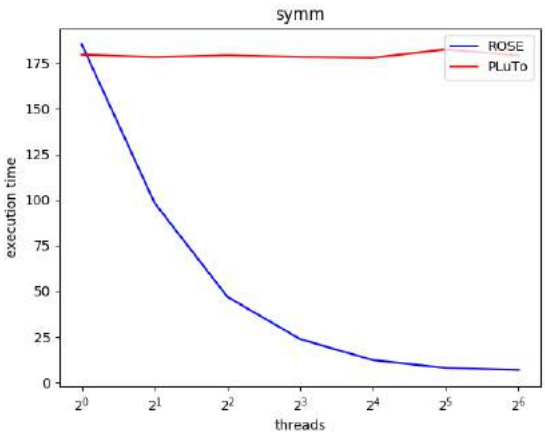

lu
ROSE
PLuTo
execution time
900
800
700
600
500
400
300
200
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

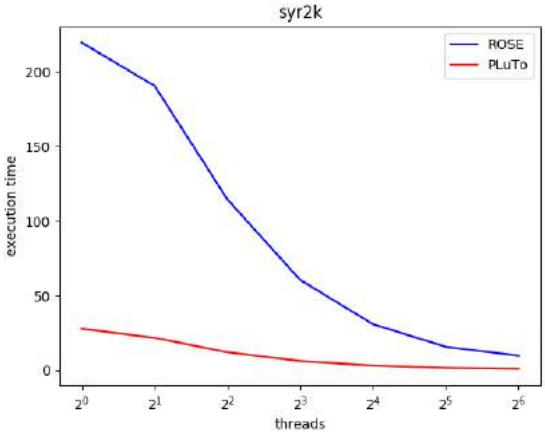

ludcmp
ROSE
PLuTo
execution time
900
800
700
600
500
400
300
200
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

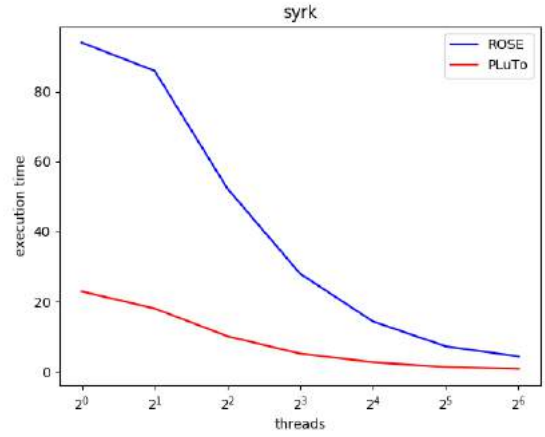

mvt
ROSE
PLuTo
execution time
0.30
0.25
0.20
0.15
0.10
0.05
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

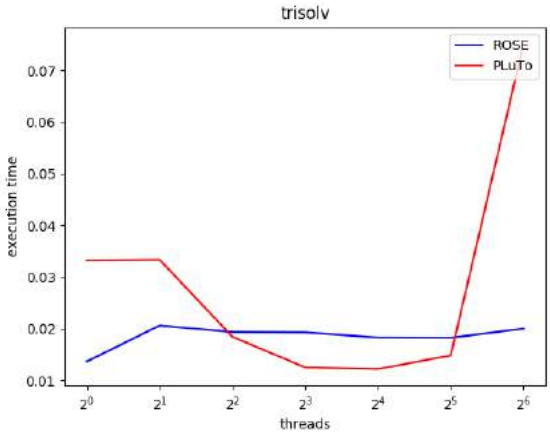

nussinov
ROSE
PLuTo
execution time
500
480
460
440
420
400
2^0 2^1 2^2 2^3 2^4 2^5 2^6
threads

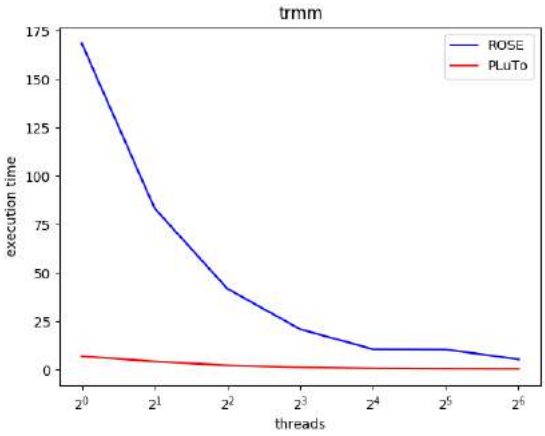

seidel-2d
ROSE
PLuTo
execution time
threads


symm
ROSE
PLuTo
execution time
threads


syr2k
ROSE
PLuTo
execution time
threads


syrk
ROSE
PLuTo
execution time
threads


trisolv
ROSE
PLuTo
execution time
threads


trmm
ROSE
PLuTo
execution time
threads

Appendix B

Implementation of the Programming Interface Communication Protocols

B.1 POSIX Implementation

B.1.1 Shared Protocol

```
#include "sharedAPI.h"

aeolus_shared_object* initialize_aeolus_shared_object(int data_size, int size) {
    aeolus_shared_object *obj=(aeolus_shared_object*)malloc(sizeof(
    aeolus_shared_object));
    int total_size = data_size*size;
    int i;
    void *shared_data = malloc(total_size);
    obj->data = shared_data;
    obj->size = total_size;
    if(pthread_mutex_init(&obj->lock,NULL) != 0)
    {
        perror("mutex init failed\n");
        exit(1);
    }
    return obj;
}

void* aeolus_synchronize(aeolus_shared_object *obj, void *local_data, int dir) {
    if(dir == 0) {
        pthread_mutex_lock(&obj->lock);
        memcpy(local_data,obj->data,obj->size);
        pthread_mutex_unlock(&obj->lock);
    }
    else if(dir == 1) {
        pthread_mutex_lock(&obj->lock);
        memcpy(obj->data,local_data,obj->size);
        pthread_mutex_unlock(&obj->lock);
    }
}
```

B.1.2 Queue Protocol

```
#include "queueAPI.h"

aeolusQnode *generate_ aeolus_Qnode(void* it, int data_size) {
    aeolusQnode *tmp = (aeolusQnode *)malloc(sizeof(aeolusQnode));
    tmp->item = malloc(data_size);
    memcpy(tmp->item,it,data_size);

    tmp->next = NULL;
    return tmp;
```

```
}

/*
 *  Queue Initialization
 */
aeolus_queue *initialize_aeolus_queue() {
        aeolus_queue *queue;
        queue = (aeolus_queue *)malloc(sizeof(aeolus_queue));
        queue->front = queue->rear = NULL;
        queue->count = 0;
        if(pthread_mutex_init(&queue->lock,NULL) != 0)
        {
                perror("mutex init failed\n");
                exit(1);
        }
        return queue;
}

/*
 * Get first item from queue, block if empty
 * @args: queue -= an initialized queue
 *        it -= a pointer to the object we want to get
 *        data_size -= size in bytes of the object
 */
void aeolus_queue_get(aeolus_queue *queue, void* it, int data_size) {
        while(queue->count == 0);
        pthread_mutex_lock(&queue->lock);
        aeolusQnode *tmp = queue->front;
        queue->front = queue->front->next;
        if(queue->front == NULL)
                queue->rear = NULL;
        queue->count--;
        memcpy(it,tmp->item,data_size);
        free(tmp);
        pthread_mutex_unlock(&queue->lock);
}

/*
 * Put item at the end of the queue, block if full
 * @args: queue -= an initialized queue
 *        it -= a pointer to the object we want to insert
 *        data_size -= size in bytes of the object
 */
bool aeolus_queue_put(aeolus_queue *queue, void* it, int data_size) {
        aeolusQnode *new_node = generate_aeolus_Qnode(it,data_size);
        pthread_mutex_lock(&queue->lock);
        if(queue->rear == NULL) {
                queue->front = queue->rear = new_node;
        }
        else {
                queue->rear->next = new_node;
                queue->rear = new_node;
        }
        queue->count++;
        pthread_mutex_unlock(&queue->lock);
        return true;
}

/*
 * Peek for the first item in the queue
 */
void aeolus_queue_peek(aeolus_queue *queue, void *it, int data_size) {
        pthread_mutex_lock(&queue->lock);
```

```
        aeolusQnode *tmp = queue->front;
        memcpy(it,tmp->item,data_size);
        pthread_mutex_unlock(&queue->lock);
}

/*
 * Get the nuber of items currently in the queue
 */
int aeolus_queue_count(aeolus_queue *queue) {
        return queue->count;
}
```

B.1.3 Signal Protocol

```
#include "signalAPI.h"

/*
 *  Signal Initialization
 */
aeolus_signal *initialize_aeolus_signal(int src, int trgt)
{
        aeolus_signal *signal = (aeolus_signal *)malloc(sizeof(aeolus_signal));
        if(pthread_mutex_init(&signal->lock,NULL) != 0)
        {
                perror("mutex init failed\n");
                exit(1);
        }
        if(pthread_cond_init(&signal->cond,NULL) !=0)
        {
                perror("cond init failed\n");
                exit(1);
        }
        signal->source = src;
        signal->target = trgt;
        signal->ready = 0;
        return signal;
}

/*
 * Send a signal to the one waiting for it
 * @args signal -= an initialized signal
 *       rd -= a bool pointer which points to a variable initialized as true
 *       aeolus_src -= the component sending the signal, leave as 0 if uncertain
 */
void aeolus_notify(aeolus_signal *curSignal, bool *rd, int aeolus_src) {
        pthread_mutex_lock(&curSignal->lock);

        curSignal->ready = *rd;
        if(curSignal->ready == true) {
                if(pthread_cond_signal(&curSignal->cond) != 0) {
                        fprintf(stderr,"Failed to send signal\n");
                        exit(0);
                }
        }
        else {
                perror("Signal called with value 0");
                exit(1);
        }
        pthread_mutex_unlock(&curSignal->lock);
}

/*
```

```
 * Wait for a signal
 * @args: signal -= an initialized signal
 *        rd -= a bool pointer
 *        aeolus_cmpid -= the component waiting for the signal, leave as 0 if uncertain
 */
void aeolus_wait(aeolus_signal *curSignal, bool *rd, int aeolus_cmpid) {
        pthread_mutex_lock(&curSignal->lock);
        while(curSignal->ready == 0) {
                if(pthread_cond_wait(&curSignal->cond,&curSignal->lock) != 0) {
                        fprintf(stderr, "failed to wait the condition variable\n");
                        exit(0);
                }
                else {
                        curSignal->ready = 1;
                        *rd = true;
                }
        }
        pthread_mutex_unlock(&curSignal->lock);
}
```

B.1.4 Mutex Protocol

```
#include "mutexAPI.h"

aeolus_mutex *initialize_aeolus_mutex() {
        aeolus_mutex *mutex = (aeolus_mutex *)malloc(sizeof(aeolus_mutex));
        if(pthread_mutex_init(&mutex->lock,NULL) != 0)
        {
                perror("mutex init failed\n");
                exit(1);
        }
        return mutex;
}

bool aeolus_mutex_lock(aeolus_mutex *mutex) {
        if(!pthread_mutex_lock(&mutex->lock))
                return true;
        else
                return false;
}

bool aeolus_mutex_unlock(aeolus_mutex *mutex) {
        if(!pthread_mutex_unlock(&mutex->lock))
                return true;
        else
                return false;
}

bool aeolus_mutex_trylock(aeolus_mutex *mutex) {
        if(!pthread_mutex_unlock(&mutex->lock))
                return true;
        else
                return false;
}
```

B.2 OpenMPI Implementation

B.2.1 Shared Protocol

```c
#include "sharedAPI.h"

void aeolus_synchronize(void *local, void *shared, int dir, int caller, int owner, int disp, int size, MPI_Datatype type, MPI_Win win) {
        if(dir == 0) {
                if(caller != owner) {
                        MPI_Win_lock(MPI_LOCK_SHARED,owner,0,win);
                        MPI_Get(shared,size,type,owner,0,size,type,win);
                        MPI_Win_unlock(owner,win);
                }
                memcpy(local,shared,size*disp);
         }
        else if(dir == 1) {
                memcpy(shared,local,size*disp);
                if(caller != owner) {
                        MPI_Win_lock(MPI_LOCK_SHARED,owner,0,win);
                        MPI_Put(shared,size,type,owner,0,size,type,win);
                        MPI_Win_unlock(owner,win);
                }
        }
        else {
                perror("Error, not a valid direction");
        }
}
```

B.2.2 Queue Protocol

```c
#include "queueAPI.h"

uint8_t *uint32_t_to_uint8_t(uint32_t x) {
        size_t size = sizeof(uint32_t) / sizeof(uint8_t);
        uint8_t *res;
        res = (uint8_t *)malloc(size*sizeof(uint8_t));
        int i;
        for(i=0; i<size; i++) {
                res[i] = (x >> (8*(size-1-i))) & 0xFF;
        }
        return res;
}

uint32_t uint8_t_to_uint32_t(uint8_t *x) {
        size_t size = sizeof(uint32_t);
        uint32_t res=0;
        int i;
        for(i=0; i<size; i++) {
                res <<= 8;
                res |= x[i];
        }
        return res;
}

void get_queue_info(uint8_t *base, uint32_t *front, uint32_t *rear, uint32_t *count) {
        *front = uint8_t_to_uint32_t(base);
        *rear = uint8_t_to_uint32_t(base+sizeof(uint32_t));
        *count = uint8_t_to_uint32_t(base+2*sizeof(uint32_t));
        return;
}

void set_queue_info(uint8_t *init_addr, uint32_t front, uint32_t rear, uint32_t count) {
        // front in the first position
```

```
34          uint8_t *front_p = uint32_t_to_uint8_t(front);
35          memcpy(init_addr,front_p,sizeof(uint32_t));
36          free(front_p);
37          // rear in the second position
38          uint8_t *rear_p = uint32_t_to_uint8_t(rear);
39          memcpy(init_addr+sizeof(uint32_t),rear_p,sizeof(uint32_t));
40          free(rear_p);
41          // count in the third position
42          uint8_t *count_p = uint32_t_to_uint8_t(count);
43          memcpy(init_addr+2*sizeof(uint32_t),count_p,sizeof(uint32_t));
44          free(count_p);
45          return;
46  }
47
48  aeolus_queue* aeolus_queue_init(uint32_t maxsize, uint32_t disp, uint32_t owner,
        MPI_Comm comm) {
49          aeolus_queue *queue = (aeolus_queue *)malloc(sizeof(aeolus_queue));
50          int32_t total_size = maxsize*disp + 3 * (sizeof(uint32_t) / sizeof(
        uint8_t)) ;
51          queue->qdata = (uint8_t *)malloc(total_size);
52  MPI_Win_create(queue->qdata,total_size,sizeof(uint8_t),MPI_INFO_NULL,comm,&queue
        ->win);
53
54          queue->disp = disp;
55          queue->maxsize = maxsize;
56          queue->owner = owner;
57          queue->front = -1;
58          queue->rear = -1;
59          queue->count = 0;
60          // Initialize relevant variables in the last 4 positions of the queue
        data
61          set_queue_info(queue->qdata+maxsize*disp,-1,-1,0);
62
63          return queue;
64  }
65
66  void aeolus_queue_free(aeolus_queue *queue) {
67          MPI_Win_free(&queue->win);
68          free(queue->qdata);
69          free(queue);
70  }
71
72  void aeolus_queue_get(aeolus_queue *queue, uint8_t *item, uint32_t caller) {
73          bool done = false;
74          int info_offset = queue->maxsize*queue->disp;
75          //printf("Caller = %d and Owner = %d\n",caller,queue->owner);
76          while(!done) {
77                  if(caller != queue->owner) {
78                          //printf("Receiver: getting info with MPI cause caller <>
            owner!\n");
79                          MPI_Win_lock(MPI_LOCK_SHARED,queue->owner,0,queue->win);
80                          MPI_Get(queue->qdata+info_offset,3*sizeof(int32_t),
        MPI_UNSIGNED_CHAR,queue->owner,info_offset,3*sizeof(int32_t),
        MPI_UNSIGNED_CHAR,queue->win);
81                          MPI_Win_unlock(queue->owner,queue->win);
82                  }
83                  get_queue_info(queue->qdata+info_offset,&queue->front,&queue->
        rear,&queue->count);
84                  if(queue->count > 0) done = true;
85          }
86
87          int32_t prev_front;
88          prev_front = queue->front;
```

```
89
90          if(queue->front == queue->rear)
91                  queue->front = queue->rear = -1;
92          else
93                  queue->front = (queue->front + queue->disp) % info_offset;
94          queue->count--;
95          set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->
    count);
96
97          if(caller != queue->owner) {
98                  MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
99                  MPI_Get(queue->qdata+prev_front,queue->disp,MPI_UNSIGNED_CHAR,
    queue->owner,prev_front,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
100                 MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),
    MPI_UNSIGNED_CHAR,queue->owner,info_offset,3*sizeof(uint32_t),
    MPI_UNSIGNED_CHAR,queue->win);
101                 MPI_Win_unlock(queue->owner,queue->win);
102         }
103         memcpy(item,queue->qdata+prev_front,queue->disp);
104
105         return;
106 }
107
108 bool aeolus_queue_put(aeolus_queue *queue, uint8_t *item, uint32_t caller) {
109         bool done = false;
110         int info_offset = queue->maxsize*queue->disp;
111
112         while(!done) {
113                 if(caller != queue->owner) {
114                         MPI_Win_lock(MPI_LOCK_SHARED,queue->owner,0,queue->win);
115                         MPI_Get(queue->qdata+info_offset,3*sizeof(int32_t),
    MPI_UNSIGNED_CHAR,queue->owner,info_offset,3*sizeof(int32_t),
    MPI_UNSIGNED_CHAR,queue->win);
116                         MPI_Win_unlock(queue->owner,queue->win);
117                 }
118                 get_queue_info(queue->qdata+info_offset,&queue->front,&queue->
    rear,&queue->count);
119                 if(queue->count < queue->maxsize) done = true;
120         }
121
122         if(queue->rear == -1) queue->front = queue->rear = 0;
123         else queue->rear = (queue->rear + queue->disp) % info_offset; //
    increasing counter: x = ( x + step ) % max
124         queue->count++;
125         set_queue_info(queue->qdata+info_offset,queue->front,queue->rear,queue->
    count);
126         memcpy(queue->qdata+queue->rear,item,queue->disp);
127
128         if(caller != queue->owner) {
129                 MPI_Win_lock(MPI_LOCK_EXCLUSIVE,queue->owner,0,queue->win);
130                 MPI_Put(queue->qdata+info_offset,3*sizeof(uint32_t),
    MPI_UNSIGNED_CHAR,queue->owner,info_offset,3*sizeof(uint32_t),
    MPI_UNSIGNED_CHAR,queue->win);
131                 MPI_Put(queue->qdata+queue->rear,queue->disp,MPI_UNSIGNED_CHAR,
    queue->owner,queue->rear,queue->disp,MPI_UNSIGNED_CHAR,queue->win);
132                 MPI_Win_unlock(queue->owner,queue->win);
133         }
134
135         return true;
136 }
137
138 void aeolus_queue_peek(aeolus_queue *queue, uint8_t *item, uint32_t caller) {
139         int info_offset = queue->maxsize*queue->disp;
```

```
140
141         if(caller != queue->owner) {
142                 MPI_Win_lock(MPI_LOCK_SHARED, queue->owner, 0, queue->win);
143                 MPI_Get(queue->qdata+info_offset, 3*sizeof(int32_t),
    MPI_UNSIGNED_CHAR, queue->owner, info_offset, 3*sizeof(int32_t),
    MPI_UNSIGNED_CHAR, queue->win);
144                 MPI_Win_unlock(queue->owner, queue->win);
145         }
146         get_queue_info(queue->qdata+info_offset, &queue->front, &queue->rear, &queue
    ->count);
147         if(queue->count == 0) {
148                 item = NULL;
149                 return;
150         }
151         if(caller != queue->owner) {
152                 MPI_Win_lock(MPI_LOCK_SHARED, queue->owner, 0, queue->win);
153                 MPI_Get(queue->qdata+queue->front, queue->disp, MPI_UNSIGNED_CHAR,
    queue->owner, queue->front, queue->disp, MPI_UNSIGNED_CHAR, queue->win);
154                 MPI_Win_unlock(queue->owner, queue->win);
155         }
156         memcpy(item, queue->qdata+queue->front, queue->disp);
157         return;
158 }
159
160 uint32_t aeolus_queue_count(aeolus_queue *queue, uint32_t caller) {
161         int info_offset = queue->maxsize*queue->disp;
162         int total_disp = info_offset + 2*sizeof(int32_t);
163         MPI_Win_lock(MPI_LOCK_SHARED, queue->owner, 0, queue->win);
164         MPI_Get(queue->qdata+total_disp, sizeof(int32_t), MPI_UNSIGNED_CHAR, queue->
    owner, total_disp, sizeof(int32_t), MPI_UNSIGNED_CHAR, queue->win);
165         MPI_Win_unlock(queue->owner, queue->win);
166         return uint8_t_to_uint32_t(queue->qdata+total_disp);
167 }
```

B.2.3 Signal Protocol

```
1  #include "signalAPI.h"
2
3  aeolus_signal *aeolus_signal_init(MPI_Comm comm) {
4          aeolus_signal *signal = (aeolus_signal *)malloc(sizeof(aeolus_signal));
5          signal->comm = comm;
6          return signal;
7  }
8
9  void aeolus_notify(aeolus_signal *signal, int dst) {
10         signal->sig = 1;
11         MPI_Send(&signal->sig, 1, MPI_INT, dst, 0, signal->comm);
12         return;
13 }
14
15 int aeolus_wait(aeolus_signal *signal, int src) {
16         MPI_Status status;
17         MPI_Recv(&signal->sig, 1, MPI_INT, src, MPI_ANY_TAG, signal->comm, &status);
18         return status.MPI_ERROR;
19 }
20
21 void aeolus_notifyall(aeolus_signal *signal, int src) {
22         MPI_Bcast(&signal->sig, 1, MPI_INT, src, signal->comm);
23 }
24
25 void aeolus_barrier(MPI_Comm comm) {
26         MPI_Barrier(comm);
```

```
        return;
}
```

B.2.4 Mutex Protocol

```
#include "mutexAPI.h"

#define MPI_MUTEX_MSG_TAG_BASE 1023

struct aeolus_mutex {
        int numprocs, ID, home, tag;
        MPI_Comm comm;
        MPI_Win win;
        unsigned char *waitlist;
};

int aeolus_mutex_init(aeolus_mutex **mutex, int home)
{
        static int tag = MPI_MUTEX_MSG_TAG_BASE;
        int numprocs, ID;
        aeolus_mutex *mtx;

        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &ID);

        mtx = (aeolus_mutex *)malloc(sizeof(aeolus_mutex));
        if (!mtx) {
                fprintf(stderr, "Warning: malloc failed on worker %2d\n", ID);
                return 1;
        }

        mtx->numprocs = numprocs;
        mtx->ID = ID;
        mtx->home = home;
        mtx->tag = tag++;
        mtx->comm = MPI_COMM_WORLD;

        if (ID == mtx->home) {
                // Allocate and expose waitlist
                MPI_Alloc_mem(mtx->numprocs, MPI_INFO_NULL, &mtx->waitlist);
                if (!mtx->waitlist) {
                        fprintf(stderr, "Warning: MPI_Alloc_mem failed on worker
    %2d\n", ID);
                        return 1;
                }
                memset(mtx->waitlist, 0, mtx->numprocs);
                MPI_Win_create(mtx->waitlist, mtx->numprocs, 1, MPI_INFO_NULL,
    mtx->comm, &mtx->win);
        } else {
                // Don't expose anything
                mtx->waitlist = NULL;
                MPI_Win_create(mtx->waitlist, 0, 1, MPI_INFO_NULL, mtx->comm, &
    mtx->win);
        }

        *mutex = mtx;

        MPI_Barrier(MPI_COMM_WORLD);

        return 0;
}

```

```
55 int aeolus_mutex_destroy(aeolus_mutex *mutex)
56 {
57         assert(mutex != NULL);
58
59         int ID;
60
61         MPI_Barrier(MPI_COMM_WORLD);
62
63         MPI_Comm_rank(MPI_COMM_WORLD, &ID);
64
65         if (ID == mutex->home) {
66                 // Free waitlist
67                 MPI_Win_free(&mutex->win);
68                 MPI_Free_mem(mutex->waitlist);
69         } else {
70                 MPI_Win_free(&mutex->win);
71                 assert(mutex->waitlist == NULL);
72         }
73
74         return 0;
75 }
76
77 int aeolus_mutex_lock(aeolus_mutex *mutex)
78 {
79         assert(mutex != NULL);
80
81         unsigned char waitlist[mutex->numprocs];
82         unsigned char lock = 1;
83         int i;
84
85         // Try to acquire lock in one access epoch
86         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
87         MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */,
      1, MPI_CHAR, mutex->win);
88         MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->
    numprocs, MPI_CHAR, mutex->win);
89         MPI_Win_unlock(mutex->home, mutex->win);
90
91         assert(waitlist[mutex->ID] == 1);
92
93         // Count the 1's
94         for (i = 0; i < mutex->numprocs; i++) {
95                 if (waitlist[i] == 1 && i != mutex->ID) {
96                         // We have to wait for the lock
97                         // Dummy receive, no payload
98                         MPI_Recv(&lock, 0, MPI_CHAR, MPI_ANY_SOURCE, mutex->tag,
    mutex->comm, MPI_STATUS_IGNORE);
99                         break;
100                }
101        }
102
103        return 0;
104 }
105
106 int aeolus_mutex_trylock(aeolus_mutex *mutex)
107 {
108        assert(mutex != NULL);
109
110        unsigned char waitlist[mutex->numprocs];
111        unsigned char lock = 1;
112        int i;
113
114        // Try to acquire lock in one access epoch
```

```
115         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
116         MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */,
      1, MPI_CHAR, mutex->win);
117         MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->
    numprocs, MPI_CHAR, mutex->win);
118         MPI_Win_unlock(mutex->home, mutex->win);
119
120         assert(waitlist[mutex->ID] == 1);
121
122         // Count the 1's
123         for (i = 0; i < mutex->numprocs; i++) {
124                 if (waitlist[i] == 1 && i != mutex->ID) {
125                         // Lock is already held, return immediately
126                         return 1;
127                 }
128         }
129
130         return 0;
131 }
132
133 int aeolus_mutex_unlock(aeolus_mutex *mutex)
134 {
135         assert(mutex != NULL);
136
137         unsigned char waitlist[mutex->numprocs];
138         unsigned char lock = 0;
139         int i, next;
140
141         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->home, 0, mutex->win);
142         MPI_Put(&lock, 1, MPI_CHAR, mutex->home, mutex->ID /* &win[mutex->ID] */,
      1, MPI_CHAR, mutex->win);
143         MPI_Get(waitlist, mutex->numprocs, MPI_CHAR, mutex->home, 0, mutex->
    numprocs, MPI_CHAR, mutex->win);
144         MPI_Win_unlock(mutex->home, mutex->win);
145         assert(waitlist[mutex->ID] == 0);
146   // If there are other processes waiting for the lock, transfer ownership
147         next = (mutex->ID + 1 + mutex->numprocs) % mutex->numprocs;
148         for (i = 0; i < mutex->numprocs; i++, next = (next + 1) % mutex->numprocs
    ) {
149                 if (waitlist[next] == 1) {
150                         // Dummy send, no payload
151                         MPI_Send(&lock, 0, MPI_CHAR, next, mutex->tag, mutex->
    comm);
152                         break;
153                 }
154         }
155
156         return 0;
157 }
```