Εκτέλεση κώδικα OpenMP σε απομακρυσμένες συσκευές σε ένα cluster

Ιωάννης Κούσιας

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Δημακόπουλος

Ιωάννινα, Ιούλιος 2025

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Περίληψη

Η παρούσα διπλωματική εργασία επικεντρώνεται στην υλοποίηση μηχανισμών απομακρυσμένης εκτέλεσης και κατανομής φόρτου σε ετερογενή συστήματα τύπου cluster, αξιοποιώντας και επεκτείνοντας τον μεταγλωττιστή OMPi που αναπτύσσεται στο Πανεπιστήμιο Ιωαννίνων. Στόχος είναι η ενίσχυση της αποδοτικότητας και της ευχρηστίας κατά την παράλληλη εκτέλεση τμημάτων κώδικα σε πολλαπλές συσκευές, όπως CPUs, GPUs και άλλους επιταχυντές.

Στο πλαίσιο της εργασίας, υλοποιήθηκε ένας επεκτάσιμος μηχανισμός κατανομής συσκευών (device scheduling), με υποστήριξη στρατηγικών τύπου **Round Robin** και **Low Workload**, που επιτρέπουν την έξυπνη ανάθεση αποστολών εκτέλεσης στις διαθέσιμες συσκευές ανάλογα με τη σειρά ή τον τρέχον φόρτο τους. Παράλληλα, ενσωματώθηκε μηχανισμός **profiling** στον πυρήνα του runtime συστήματος, επιτρέποντας την παρακολούθηση και καταγραφή χρόνων εκτέλεσης σε κρίσιμα σημεία της διαδικασίας απομακρυσμένης εκτέλεσης.

Η εργασία συμβάλλει τόσο στη λειτουργική επέκταση του OMPi, όσο και στη βελτίωση της προσβασιμότητας σε τεχνικές απομακρυσμένης εκτέλεσης μέσω ενός φιλικού προς τον προγραμματιστή περιβάλλοντος. Η ορθότητα και αποδοτικότητα της υλοποίησης αξιολογούνται μέσω πειραμάτων σε σύγχρονα ετερογενή περιβάλλοντα.

Λέξεις-Κλειδιά: απομακρυσμένη εκτέλεση, κατανομή φόρτου, parallel scheduling, profiling, clusters, OMPi, ετερογενή συστήματα.

Abstract

This thesis focuses on the implementation of remote code execution mechanisms and load distribution strategies in heterogeneous cluster systems, through modifications to the OMPi compiler developed at the University of Ioannina. The main objective is to improve both performance and usability when executing parallelizable code segments across multiple computational devices, such as CPUs, GPUs, and hardware accelerators.

As part of this work, an extensible **device scheduling mechanism** was developed, supporting both **Round Robin** and **Low Workload** strategies. These approaches enable dynamic task assignment to available devices based on execution order or current load. Additionally, a **profiling system** was integrated into the runtime layer, allowing accurate monitoring and logging of execution times at critical stages of the remote offloading process.

The proposed contributions enhance the functionality of the OMPi system and simplify access to distributed computing techniques by providing a user-friendly interface for developers. The implementation is evaluated through performance measurements on modern heterogeneous cluster environments.

Keywords: remote execution, load balancing, parallel scheduling, profiling, clusters, OMPi, heterogeneous systems.

Πίνακας Περιεχομένων

Κεφάλαιο 1. Εισαγωγή

1.1 Παράλληλος προγραμματισμός και αρχιτεκτονικές συστημάτων

Ο παράλληλος προγραμματισμός αποτελεί ένα βασικό πεδίο της υπολογιστικής επιστήμης, επιτρέποντας την ταυτόχρονη εκτέλεση πολλών υπολογιστικών διεργασιών [1]. Η βασική του ιδέα είναι η διάσπαση της εργασίας σε ανεξάρτητα τμήματα, τα οποία μπορούν να εκτελεστούν ταυτόχρονα, βελτιώνοντας την απόδοση και την αποτελεσματικότητα των συστημάτων. Σε αντίθεση με τον σειριακό προγραμματισμό, όπου οι εντολές εκτελούνται διαδοχικά, στον παράλληλο προγραμματισμό επιδιώκεται η εκμετάλλευση των διαθέσιμων πόρων, όπως πολυπύρηνοι επεξεργαστές (CPUs), κάρτες γραφικών (GPUs) ή άλλους επιταχυντές.

Ανάλογα με την αρχιτεκτονική της μνήμης, τα υπολογιστικά συστήματα που υποστηρίζουν την εκτέλεση παράλληλου κώδικα διακρίνονται σε τρεις κύριες κατηγορίες [1]. Η πρώτη αφορά τα συστήματα κοινόχρηστης μνήμης. Σε αυτά, πολλές μονάδες επεξεργασίας έχουν άμεση πρόσβαση σε έναν ενιαίο, φυσικό χώρο μνήμης. Αυτή η αρχιτεκτονική διευκολύνει την ανταλλαγή δεδομένων μεταξύ των επεξεργαστών, καθώς όλοι μπορούν να διαβάζουν και να γράφουν στα ίδια τμήματα της μνήμης χωρίς την ανάγκη ρητής επικοινωνίας. Ωστόσο, απαιτείται προσεκτικός συγχρονισμός για την αποφυγή συγκρούσεων και αδιεξόδων κατά την ταυτόχρονη πρόσβαση σε κοινά δεδομένα. Τέτοιου τύπου συστήματα συναντώνται συνήθως σε πολυπύρηνους υπολογιστές και servers, όπου η αρχιτεκτονική του hardware επιτρέπει την κοινή χρήση της μνήμης με υψηλές ταχύτητες.

Η δεύτερη κατηγορία περιλαμβάνει τα συστήματα κατανεμημένης μνήμης. Σε αυτά, κάθε υπολογιστική μονάδα, ή αλλιώς κόμβος, διαθέτει τη δική της ξεχωριστή τοπική μνήμη, στην οποία έχουν πρόσβαση μόνο οι επεξεργαστές του ίδιου κόμβου. Η επικοινωνία μεταξύ των κόμβων και η ανταλλαγή δεδομένων γίνεται μέσω ανταλλαγής

μηνυμάτων, συνήθως με τη χρήση εξειδικευμένων πρωτοκόλλων ή βιβλιοθηκών όπως το MPI (Message Passing Interface). Η αρχιτεκτονική αυτή προσφέρει μεγάλη επεκτασιμότητα, καθώς μπορεί να υποστηρίξει μεγάλο αριθμό υπολογιστικών μονάδων χωρίς τα προβλήματα συμφόρησης που παρουσιάζει η κοινόχρηστη μνήμη. Ωστόσο, η ανάπτυξη παράλληλου λογισμικού για τέτοια συστήματα είναι συνήθως πιο περίπλοκη, καθώς απαιτείται ρητός σχεδιασμός για την επικοινωνία και τον συγχρονισμό των διεργασιών.

Τέλος, υπάρχει η κατηγορία των υβριδικών συστημάτων, τα οποία συνδυάζουν χαρακτηριστικά και από τα δύο προηγούμενα μοντέλα. Σε αυτά τα συστήματα, σε τοπικό επίπεδο μπορεί να υφίσταται κοινόχρηστη μνήμη ανάμεσα σε επεξεργαστές του ίδιου υπολογιστικού κόμβου, ενώ η επικοινωνία μεταξύ διαφορετικών κόμβων γίνεται μέσω ανταλλαγής μηνυμάτων, όπως συμβαίνει στα συστήματα κατανεμημένης μνήμης.

Η επιλογή αρχιτεκτονικής επηρεάζει σημαντικά τόσο την πολυπλοκότητα του παράλληλου προγραμματισμού όσο και την τελική απόδοση του συστήματος, καθώς καθορίζει κρίσιμους παράγοντες όπως ο τρόπος συγχρονισμού, η μεταφορά δεδομένων και η στρατηγική κατανομής των εργασιών στους επεξεργαστές.

Για την ανάπτυξη παράλληλων εφαρμογών σε τέτοια συστήματα, έχουν αναπτυχθεί διάφορα προγραμματιστικά πρότυπα, όπως το MPI, που χρησιμοποιείται κυρίως σε συστήματα κατανεμημένης μνήμης, αλλά και το OpenMP, το οποίο επιτρέπει την ανάπτυξη παράλληλου κώδικα σε περιβάλλοντα κοινόχρηστης μνήμης. Το OpenMP προσφέρει έναν απλό και επεκτάσιμο τρόπο για την παραλληλοποίηση εφαρμογών και θεωρείται ιδιαίτερα φιλικό προς τον προγραμματιστή, καθώς δεν απαιτεί ρητή διαχείριση μνήμης ή επικοινωνίας. Τα τελευταία χρόνια, το OpenMP έχει εξελιχθεί σημαντικά, υποστηρίζοντας και την εκτέλεση σε ετερογενή περιβάλλοντα μέσω μηχανισμών offloading σε επιταχυντές όπως GPUs, επεκτείνοντας έτσι τη χρήση του και πέρα από τα όρια της κοινόχρηστης μνήμης [2].

1.2 Υπολογιστικοί πόροι στον παράλληλο προγραμματισμό

Ο σύγχρονος παράλληλος προγραμματισμός στηρίζεται κατά μεγάλο μέρος στην αξιοποίηση εξειδικευμένων υπολογιστικών πόρων, με σκοπό την επιτάχυνση της εκτέλεσης απαιτητικών εφαρμογών. Οι σημαντικότεροι από αυτούς τους πόρους είναι οι πολυπύρηνες κεντρικές μονάδες επεξεργασίας (CPUs), οι μονάδες επεξεργασίας γραφικών (GPUs) και οι ειδικοί επιταχυντές.

Οι πολυπύρηνες CPUs αποτελούν την εξέλιξη των παραδοσιακών επεξεργαστών, περιλαμβάνοντας πολλούς ανεξάρτητους πυρήνες μέσα στο ίδιο φυσικό ολοκληρωμένο κύκλωμα. Κάθε πυρήνας έχει τη δυνατότητα να εκτελεί εντολές αυτόνομα, επιτρέποντας την ταυτόχρονη εκτέλεση πολλαπλών νημάτων (threads). Αν και οι CPUs σχεδιάζονται για υψηλή απόδοση σε σειριακό και γενικού σκοπού κώδικα, η πολυπυρηνικότητα τους παρέχει επαρκή δυνατότητα για παράλληλη εκτέλεση, ιδιαίτερα σε εφαρμογές που μπορούν να επιμερίσουν τα δεδομένα ή τις διεργασίες τους.

Οι GPUs, αρχικά σχεδιασμένες για την επεξεργασία γραφικών, αποτελούν πλέον έναν από τους κυριότερους πόρους για υπολογιστικά φορτία γενικού σκοπού. Σε αντίθεση με τις CPUs, οι οποίες έχουν λίγους αλλά ισχυρούς πυρήνες, οι GPUs διαθέτουν εκατοντάδες ή και χιλιάδες απλούστερους πυρήνες, ειδικά βελτιστοποιημένους για την εκτέλεση πολλών παρόμοιων υπολογισμών παράλληλα. Αυτή η χαρακτηριστική μαζική παραλληλία καθιστά τις GPUs εξαιρετικά αποδοτικές για προβλήματα που μπορούν να αναλυθούν σε πολλά και επαναλαμβανόμενα βήματα, όπως η επεξεργασία εικόνας, η προσομοίωση φυσικών συστημάτων και η εκπαίδευση μοντέλων μηχανικής μάθησης. Ως αποτέλεσμα, οι GPUs αποτελούν σήμερα μία από τις κυρίαρχες επιλογές για την υλοποίηση παράλληλου προγραμματισμού υψηλής απόδοσης.

Εκτός από τις CPUs και τις GPUs, υπάρχουν και εξειδικευμένοι επιταχυντές, όπως τα FPGAs (Field-Programmable Gate Arrays) και τα ASICs (Application-Specific Integrated Circuits), οι οποίοι έχουν σχεδιαστεί για να εκτελούν πολύ συγκεκριμένες λειτουργίες με εξαιρετικά υψηλή ταχύτητα και ενεργειακή απόδοση [7], [8]. Αν και απαιτούν περισσότερο εξειδικευμένο σχεδιασμό και προγραμματισμό, οι επιταχυντές αυτοί βρίσκουν εφαρμογή σε τομείς όπως η επεξεργασία σήματος, η ασφάλεια, και η τεχνητή νοημοσύνη, όπου η απόδοση σε ειδικά καθορισμένα καθήκοντα υπερτερεί της γενικότητας.

Η επιλογή ανάμεσα σε CPUs, GPUs και επιταχυντές εξαρτάται από τις απαιτήσεις της εκάστοτε εφαρμογής και το είδος των υπολογιστικών προβλημάτων που πρέπει να επιλυθούν. Ωστόσο, η τάση των τελευταίων ετών δείχνει καθαρά μια στροφή προς τις GPUs, λόγω της εξαιρετικής τους ικανότητας να διαχειρίζονται μεγάλες ποσότητες παράλληλων δεδομένων και να επιτυγχάνουν υψηλή υπολογιστική απόδοση σε πληθώρα εφαρμογών πέραν του γραφικού τομέα. Η αυξανόμενη ανάγκη για επεξεργασία σε μεγάλη κλίμακα έχει οδηγήσει στην ανάπτυξη συστημάτων που συνδυάζουν πολλαπλούς τέτοιους υπολογιστικούς πόρους – όπως πολυπύρηνες CPUs και GPUs – σε ετερογενή περιβάλλοντα. Τέτοια περιβάλλοντα συγκροτούνται συχνά σε μορφή υπολογιστικών clusters, τα οποία προσφέρουν την απαραίτητη υποδομή για την

εκτέλεση πολύπλοκων και υψηλών υπολογιστικών φορτίων με παράλληλο τρόπο. Η μελέτη της αρχιτεκτονικής, των χαρακτηριστικών και της αξιοποίησης των clusters αποτελεί κρίσιμο βήμα στην κατανόηση και την εφαρμογή του παράλληλου προγραμματισμού σε πραγματικές συνθήκες.

1.3 Clusters πολυπλοκότητα και μηχανισμοί διαχείρισης

Ένα cluster είναι μια μορφή κατανεμημένου συστήματος, που αποτελείται από πολλούς αυτόνομους υπολογιστικούς κόμβους. Κάθε κόμβος διαθέτει δική του CPU, μνήμη και αποθηκευτικό χώρο, και συνδέεται μέσω δικτύου με τους υπόλοιπους κόμβους. Ο στόχος είναι η ενιαία αξιοποίηση των υπολογιστικών πόρων για την εκτέλεση εργασιών μεγάλης κλίμακας.

Τα clusters βασίζονται σχεδόν αποκλειστικά σε κατανεμημένη μνήμη, πράγμα που σημαίνει ότι απαιτείται ρητή διαχείριση της τοποθέτησης των δεδομένων όσο και της επικοινωνίας μεταξύ των εμπλεκόμενων κόμβων, συνήθως μέσω πρωτοκόλλων όπως το MPI. Σε πιο πολύπλοκες υλοποιήσεις, αξιοποιούνται και υβριδικά μοντέλα, όπου εντός κάθε κόμβου μπορεί να υπάρχει κοινόχρηστη μνήμη, ενώ η επικοινωνία μεταξύ κόμβων απαιτεί αποστολή και λήψη δεδομένων.

Οι προκλήσεις που εμφανίζονται σε clusters σχετίζονται με την ετερογένεια των πόρων, τη δυναμική κατανομή τους και την ανάγκη για αποδοτικό συγχρονισμό και μεταφορά πληροφορίας. Καθώς τα υπολογιστικά περιβάλλοντα εξελίσσονται σε πολυπύρηνα και κατανεμημένα, η προγραμματιστική πολυπλοκότητα αυξάνεται. Δεν αρκεί πλέον μόνο η δημιουργία αποδοτικών αλγορίθμων, αλλά απαιτούνται και μηχανισμοί διαχείρισης πόρων, όπως το profiling και το scheduling, ώστε να εξασφαλιστεί η ορθή αξιοποίηση του συστήματος.

Το profiling είναι η διαδικασία παρακολούθησης της εκτέλεσης καθώς και της καταγραφής χρόνων των κλήσεων ενός προγράμματος με σκοπό την αναγνώριση καθυστερήσεων, την κατανάλωση πόρων και την αξιολόγηση των επιδόσεων. Σε κατανεμημένα συστήματα, το profiling βοηθά στον εντοπισμό του βέλτιστου τρόπου εκτέλεσης, αποκαλύπτοντας κρίσιμες πληροφορίες όπως:

- Χρόνος εκτέλεσης ανά κόμβο,
- Κόστος επικοινωνίας,
- Αποτελεσματικότητα του παράλληλου κώδικα.

Από την άλλη πλευρά, το scheduling είναι η στρατηγική με την οποία κατανέμονται οι εργασίες στους διαθέσιμους πόρους. Σε συστήματα με πολλούς κόμβους οι οποίοι δεν διαθέτουν ισοδύναμους πόρους, ο τρόπος που οργανώνεται η εκτέλεση των τμημάτων του προγράμματος επηρεάζει σημαντικά την συνολική απόδοση. Οι δύο παραπάνω μηχανισμοί (profiling και scheduling) είναι αναπόσπαστα εργαλεία για την επίτευξη αποδοτικής εκτέλεσης σε μεγάλης κλίμακας υπολογιστικά περιβάλλοντα.

Αν και παραδοσιακά η ανάπτυξη εφαρμογών για clusters βασιζόταν στο MPI για την υλοποίηση της επικοινωνίας μεταξύ των κόμβων, σύγχρονες προσεγγίσεις προσπαθούν να απλοποιήσουν τη διαδικασία παραλληλοποίησης και να αποκρύψουν τη διαχείριση της κατανεμημένης εκτέλεσης από τον χρήστη. Ένα τέτοιο παράδειγμα είναι η επέκταση του OpenMP με μηχανισμούς remote offloading, που επιτρέπουν την εκτέλεση παράλληλων περιοχών όχι μόνο σε τοπικούς πυρήνες, αλλά και σε απομακρυσμένους κόμβους ενός cluster. Στο πλαίσιο αυτό, ο OMPi, ένας ερευνητικός μεταγλωττιστής που υποστηρίζει το πρότυπο OpenMP, έχει ενσωματώσει μηχανισμούς για την απομακρυσμένη εκτέλεση χωρίς να απαιτείται από τον προγραμματιστή να γράψει κώδικα MPI. Αυτή η εξέλιξη ανοίγει τον δρόμο για υψηλού επιπέδου προγραμματισμό σε κατανεμημένα περιβάλλοντα, αξιοποιώντας τη γνωστή σύνταξη του OpenMP.

1.4 Αντικείμενο εργασίας

Η παρούσα διπλωματική εργασία έχει ως βασικό αντικείμενο τη μελέτη, τον σχεδιασμό και την υλοποίηση ενός επεκτάσιμου συστήματος απομακρυσμένης εκτέλεσης κώδικα σε κατανεμημένο περιβάλλον υπολογιστικού cluster. Το σύστημα απευθύνεται σε παράλληλες εφαρμογές που έχουν αναπτυχθεί με χρήση του προτύπου OpenMP, το οποίο αποτελεί ευρέως διαδεδομένο εργαλείο για την ανάπτυξη παραλληλοποιημένου κώδικα σε συστήματα κοινόχρηστης μνήμης.

Η εργασία βασίζεται στον OMPi, έναν ερευνητικό μεταφραστή που υποστηρίζει το πρότυπο OpenMP και περιλαμβάνει ήδη μηχανισμούς για την απομακρυσμένη εκτέλεση παράλληλων περιοχών. Σκοπός της παρούσας υλοποίησης είναι η επέκταση αυτής της υφιστάμενης λειτουργικότητας, με την ενσωμάτωση πρόσθετων μηχανισμών που επιτρέπουν την παρακολούθηση της απόδοσης και την αποδοτική κατανομή του υπολογιστικού έργου.

Συγκεκριμένα, η επέκταση αυτή εστιάζει στα εξής:

- Μηχανισμό profiling σε χρόνο εκτέλεσης, ο οποίος συλλέγει κρίσιμα δεδομένα απόδοσης, όπως χρόνους εκτέλεσης των παράλληλων περιοχών,

κόστη επικοινωνίας μεταξύ των κόμβων, καθώς και τη χρήση διαθέσιμων πόρων. Οι πληροφορίες αυτές είναι καθοριστικές για την αξιολόγηση και βελτιστοποίηση της συμπεριφοράς του συστήματος.

- Μηχανισμό scheduling, ο οποίος είναι υπεύθυνος για τη στρατηγική ανάθεσης των υπολογιστικών πυρήνων στους κόμβους του cluster. Στην παρούσα εργασία υποστηρίζονται δύο στρατηγικές κατανομής:
    - round-robin, κατά την οποία οι εργασίες κατανέμονται με κυκλικό τρόπο στους κόμβους, ανεξάρτητα από τον φόρτο τους.
    - load-based, που βασίζεται στον δυναμικό προσδιορισμό του φόρτου κάθε κόμβου, ώστε να επιτυγχάνεται καλύτερη εξισορρόπηση και αποδοτικότητα.

Ο απώτερος στόχος είναι η ανάπτυξη ενός αυτόνομου και διαφανούς συστήματος απομακρυσμένης εκτέλεσης, το οποίο να επιτρέπει στον προγραμματιστή να αξιοποιεί αποτελεσματικά τους υπολογιστικούς πόρους ενός cluster, χωρίς να απαιτείται εκτεταμένη γνώση ή τροποποίηση του κώδικα. Η προσέγγιση αυτή διευρύνει τις δυνατότητες του συνδυασμού OMPi και OpenMP, καθιστώντας τα εργαλεία αυτά κατάλληλα για χρήση σε κατανεμημένα περιβάλλοντα.

1.5 Δομή του υπόλοιπου κειμένου

Η δομή της εργασίας έχει ως εξής:

- **Κεφάλαιο 2:** Παρουσίαση των υπαρχόντων εργαλείων και τεχνολογιών, OMPi και OpenMP.
- **Κεφάλαιο 3:** Αναλύεται η ανάγκη για εξέλιξη της απομακρυσμένης εκτέλεσης κώδικα.
- **Κεφάλαιο 4:** Περιγράφεται η υλοποίηση του profiler και του μηχανισμού scheduling.
- **Κεφάλαιο 5:** Παρουσιάζονται πειραματικά αποτελέσματα, συγκρίσεις και ανάλυση απόδοσης.
- **Κεφάλαιο 6:** Σύνοψη και προτάσεις για μελλοντική εργασία.

Κεφάλαιο 2. OpenMP και OMPi

2.1 Εισαγωγή στο OpenMP

Η ταχεία πρόοδος στον τομέα της επιστήμης των υπολογιστών και η εμφάνιση απαιτητικών εφαρμογών σε τομείς όπως η βιοπληροφορική, η μηχανική μάθηση, η επεξεργασία σήματος και οι αριθμητικές προσομοιώσεις, έχουν δημιουργήσει την ανάγκη για αξιοποίηση όλο και μεγαλύτερης υπολογιστικής ισχύος. Ωστόσο, οι φυσικοί και τεχνολογικοί περιορισμοί στην αύξηση της συχνότητας των CPUs, καθώς και ζητήματα που σχετίζονται με την κατανάλωση ενέργειας και την απόδοση ανά πυρήνα, καθιστούν επιτακτική τη μετάβαση σε παραλληλία και ετερογένεια.

Σε αυτό το πλαίσιο, η εξέλιξη των συστημάτων με πολλούς πυρήνες (multi-core) και η ενσωμάτωση επιταχυντών όπως οι κάρτες γραφικών (GPUs), προσφέρουν σημαντικά πλεονεκτήματα. Παράλληλα, δημιουργείται η ανάγκη για κατάλληλα εργαλεία και πρότυπα προγραμματισμού, τα οποία διευκολύνουν τον αποδοτικό διαμοιρασμό εργασιών σε πολλαπλούς επεξεργαστικούς πόρους. Το OpenMP αποτελεί ένα από τα σημαντικότερα εργαλεία, καθώς επιτρέπει στον προγραμματιστή να εκμεταλλευτεί τόσο τις δυνατότητες πολυνηματικής εκτέλεσης σε κοινόχρηστη μνήμη όσο και τις υπολογιστικές ικανότητες εξωτερικών συσκευών.

Το πρότυπο OpenMP βασίζεται σε compiler directives, οι οποίες ενσωματώνονται απευθείας στον πηγαίο κώδικα, χωρίς την ανάγκη χρήσης πολύπλοκων βιβλιοθηκών ή τροποποίησης του βασικού αλγορίθμου. Η φιλοσοφία του «incremental parallelization» επιτρέπει την σταδιακή μετατροπή ενός σειριακού προγράμματος σε παράλληλο, με ελάχιστες αλλαγές στη δομή του κώδικα.

Η γενική σύνταξη μιας οδηγίας OpenMP είναι:

```
#pragma omp directive [clauses]
```

όπου η directive προσδιορίζει τον τύπο της παράλληλης κατασκευής (όπως parallel, for, sections, target), ενώ οι clauses επιτρέπουν τον έλεγχο της συμπεριφοράς των νημάτων και της διαχείρισης των δεδομένων.

2.1.1 Παράλληλος προγραμματισμός με OpenMP

Η πιο βασική χρήση του OpenMP αφορά την παράλληλη εκτέλεση τμημάτων κώδικα από πολλαπλά νήματα (threads), αξιοποιώντας τους πολλαπλούς πυρήνες των σύγχρονων επεξεργαστών. Αυτό επιτυγχάνεται μέσω της οδηγίας `#pragma omp parallel`, η οποία δηλώνει ότι το τμήμα του κώδικα που ακολουθεί θα εκτελεστεί ταυτόχρονα από μια ομάδα νημάτων [2]:

```
#pragma omp parallel
{
  // Κώδικας που εκτελείται από κάθε νήμα
}
```

Μία συχνή και ιδιαιτέρως χρήσιμη επέκταση αυτής της οδηγίας είναι η parallel for, η οποία εφαρμόζεται σε βρόχους επανάληψης. Χάρη σε αυτήν, οι επαναλήψεις ενός βρόχου μπορούν να κατανεμηθούν αυτόματα στα διαθέσιμα νήματα:

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
  // Παράλληλη εκτέλεση του σώματος του βρόχου
}
```

Η δυνατότητα αυτή είναι κρίσιμη για τη βελτιστοποίηση της απόδοσης υπολογιστικά εντατικών εφαρμογών, καθώς μειώνει τον συνολικό χρόνο εκτέλεσης με την αξιοποίηση της ταυτόχρονης εκτέλεσης.

Για να επιτευχθεί καλύτερη απόδοση και ισορροπημένη κατανομή φόρτου, το OpenMP παρέχει διάφορες στρατηγικές scheduling, οι οποίες καθορίζουν πώς κατανέμονται οι επαναλήψεις του βρόχου στα νήματα:

- **static:** Ο compiler μοιράζει τις επαναλήψεις εκ των προτέρων, ισομερώς ανάμεσα στα διαθέσιμα νήματα. Είναι αποτελεσματικό όταν όλες οι επαναλήψεις απαιτούν παρόμοιο υπολογιστικό φόρτο.
- **dynamic:** Οι επαναλήψεις εκχωρούνται σε blocks δυναμικά κατά τη διάρκεια εκτέλεσης. Όταν ένα νήμα ολοκληρώνει τις τρέχουσες επαναλήψεις του, λαμβάνει νέα. Αυτή η στρατηγική είναι χρήσιμη όταν οι επαναλήψεις έχουν

άνισο κόστος εκτέλεσης, προσφέροντας καλύτερη εξισορρόπηση, με ελαφρύ κόστος σε runtime overhead.

- **guided:** Παρόμοιο με το dynamic, αλλά τα blocks είναι αρχικά μεγάλα και μικραίνουν σταδιακά. Συνδυάζει την απόδοση του static με την εξισορρόπηση του dynamic, ειδικά για μεγάλους βρόχους.
- **auto:** Η επιλογή της στρατηγικής scheduling γίνεται αυτόματα από το runtime σύστημα, σύμφωνα με το εκάστοτε περιβάλλον και συνθήκες.
- **runtime:** Η στρατηγική καθορίζεται μέσω της μεταβλητής περιβάλλοντος OMP_SCHEDULE, επιτρέποντας ευελιξία και τροποποίηση χωρίς επαναμετάφραση του κώδικα.

Η χρήση κατάλληλης στρατηγικής κατανομής επιδρά ουσιαστικά στη συνολική απόδοση, ιδιαίτερα σε περιπτώσεις με μεγάλο αριθμό επαναλήψεων ή μεταβλητό υπολογιστικό κόστος ανά επανάληψη.

Κατά την είσοδο σε μια παράλληλη περιοχή, ο τρόπος με τον οποίο οι μεταβλητές "συμπεριφέρονται" ανά νήμα ελέγχεται μέσω ειδικών clauses. Τα πιο συνήθεις είναι:

- **shared:** Η μεταβλητή είναι κοινή για όλα τα νήματα. Απαιτείται προσοχή όταν γίνεται ταυτόχρονη πρόσβαση, διότι μπορεί να οδηγήσει σε race conditions.
- **private:** Κάθε νήμα δημιουργεί ένα δικό του αντίγραφο της μεταβλητής χωρίς αρχικοποίηση. Οποιεσδήποτε τιμές εντός της παράλληλης περιοχής δεν επηρεάζουν το εξωτερικό περιβάλλον.
- **firstprivate:** Όπως το private, αλλά το κάθε νήμα ξεκινά με την τιμή της μεταβλητής όπως ήταν πριν την είσοδο στην παράλληλη περιοχή. Είναι χρήσιμο όταν απαιτείται αρχική κατάσταση.
- **lastprivate:** Χρησιμοποιείται κυρίως σε parallel for βρόχους. Η τιμή της μεταβλητής από την τελευταία επανάληψη (κατά την εκτέλεση) αποθηκεύεται στο κύριο νήμα μετά την ολοκλήρωση.

Η σωστή επιλογή φράσης είναι απαραίτητη για την ορθή λογική εκτέλεση του προγράμματος και για την αποφυγή σφαλμάτων, όπως λανθασμένα αποτελέσματα ή απροσδιόριστη συμπεριφορά.

2.1.2 Υποστήριξη για συσκευές (devices)

Από την έκδοση 4.0 του OpenMP και έπειτα, το πρότυπο ενσωμάτωσε υποστήριξη για διαθέσιμες υπολογιστικές συσκευές, όπως οι GPUs. Η επέκταση αυτή εισήγαγε τη δυνατότητα offloading, δηλαδή την εκτέλεση επιλεγμένων τμημάτων του

προγράμματος σε διαθέσιμη συσκευή, με στόχο την αξιοποίηση της υψηλής παραλληλίας και της ταχύτητας επιπλέον συσκευών (GPUs, CPUs, accelerators).

Η βασική οδηγία που υλοποιεί το offloading είναι η target, η οποία καθορίζει ότι το αντίστοιχο τμήμα κώδικα θα εκτελεστεί σε διαθέσιμη συσκευή [2]:

```
#pragma omp target
{
  // Κώδικας που εκτελείται στη συσκευή (π.χ. GPU)
}
```

Για την ορθή εκτέλεση του κώδικα σε άλλη συσκευή, είναι απαραίτητη η σωστή μεταφορά δεδομένων ανάμεσα στη CPU του κόμβου από τον οποίο εκτελείται ο κώδικας (host) και τη συσκευή. Το OpenMP προσφέρει τον μηχανισμό map, ο οποίος χρησιμοποιείται ως clause σε directives όπως target και καθορίζει την κατεύθυνση της μεταφοράς (από ή προς τη συσκευή):

```
#pragma omp target map(to: A[0:N]) map(from: B[0:N])
```

Στο παραπάνω παράδειγμα, ο πίνακας Α μεταφέρεται στη συσκευή, εκτελείται κάποια πράξη (π.χ. επεξεργασία ή υπολογισμός) και το αποτέλεσμα αποθηκεύεται στον πίνακα Β, ο οποίος επιστρέφεται στον host.

2.1.3 Διαχείριση δεδομένων κατά το offloading

Κατά τη διαδικασία offloading, η διαχείριση των δεδομένων είναι εξαιρετικά σημαντική, τόσο για λόγους απόδοσης όσο και για την αποφυγή σφαλμάτων (π.χ. μη έγκυρη πρόσβαση σε μνήμη). Το OpenMP παρέχει μηχανισμούς υψηλού επιπέδου για να προσφέρει στον προγραμματιστή έλεγχο πάνω στη μεταφορά, κατοχύρωση και διάρκεια ζωής των δεδομένων σε μια συσκευή.

Κύριες επιλογές περιλαμβάνουν [2]:

- **target data**: Ορίζει μία ευρύτερη περιοχή μέσα στην οποία τα δεδομένα παραμένουν προσβάσιμα στη συσκευή, αποφεύγοντας την ανάγκη για επαναλαμβανόμενες μεταφορές.
- **target enter data / target exit data**: Επιτρέπουν τον ρητό έλεγχο για το πότε τα δεδομένα θα μεταφερθούν προς και από τη συσκευή αντίστοιχα.
- **declare target**: Χρησιμοποιείται για να καταστήσει συναρτήσεις ή μεταβλητές διαθέσιμες σε target regions. Ιδιαίτερα χρήσιμο για σταθερές τιμές ή επαναχρησιμοποιούμενες συναρτήσεις.

Παράδειγμα:

```
#pragma omp target data map(to: A[0:N]) map(from: B[0:N])
{
    #pragma omp target
    for (int i = 0; i < N; ++i) {
        B[i] = A[i] + 1;
    }
    // ενδεχομένως περισσότερος κώδικας
    #pragma omp target
    for (int i = 0; i < N; i++) {
         B[i] = ( B[i] + A[i] ) / 2 ;
    }
}
```

Με αυτόν τον τρόπο, τα δεδομένα A και B διατηρούνται καθ' όλη τη διάρκεια του target data block στη συσκευή, μειώνοντας τον overhead των μεταφορών.

2.1.4 Παράδειγμα χρήσης offloading

Ακολουθεί ένα ενδεικτικό και απλό παράδειγμα που δείχνει την πρακτική εφαρμογή των δυνατοτήτων offloading σε έναν αριθμητικό υπολογισμό:

```
#pragma omp target map(to: A[0:N]) map(from: B[0:N])
for (int i = 0; i < N; ++i) {
    B[i] = A[i] * 2.0;
}
```

Στο παράδειγμα αυτό, γίνεται επεξεργασία του πίνακα A στη συσκευή, με την έξοδο να αποθηκεύεται στον πίνακα B και να επιστρέφεται στον host. Το πλεονέκτημα του OpenMP είναι ότι η όλη διαδικασία offloading γίνεται με τρόπο συμβατό με τον υπάρχοντα C/C++ κώδικα, χωρίς την ανάγκη εξειδικευμένου προγραμματισμού σε CUDA ή OpenCL.

Η ευκολία αυτή καθιστά το OpenMP ελκυστικό εργαλείο για ακαδημαϊκούς και ερευνητές, οι οποίοι συχνά δεν επιθυμούν να εμπλακούν με την πολυπλοκότητα των χαμηλού επιπέδου APIs, αλλά χρειάζονται παρ' όλα αυτά σημαντική επιτάχυνση υπολογισμών.

2.1.5 Συνοπτική αξιολόγηση

Η εισαγωγή του offloading στο OpenMP αποτελεί σημαντική πρόοδο για την υποστήριξη ετερογενών αρχιτεκτονικών, παρέχοντας μια υψηλού επιπέδου, φορητή και σταδιακά επεκτάσιμη λύση για την αξιοποίηση επιταχυντών. Σε αντίθεση με άλλες τεχνολογίες όπως CUDA, το OpenMP δεν μεταβάλλει σημαντικά τον κώδικα και επιτρέπει γρήγορη μετάβαση από σειριακό σε παράλληλο και τελικώς ετερογενές πρόγραμμα.

Ωστόσο, παραμένουν ορισμένες προκλήσεις:

- **Συμβατότητα:** Το επίπεδο υποστήριξης του OpenMP offloading διαφέρει μεταξύ μεταγλωττιστών (π.χ. OMPi, GCC, LLVM, Intel).
- **Αποσφαλμάτωση:** Ο εντοπισμός σφαλμάτων σε target regions δεν είναι πάντα απλός.
- **Απόδοση:** Η βέλτιστη επίδοση απαιτεί προσεκτική διαχείριση μεταφορών και εκτέλεσης.

Παρά τα παραπάνω, για εφαρμογές όπου η φορητότητα, η απλότητα και η προοδευτική μετάβαση σε επιταχυνόμενα υπολογιστικά μοντέλα είναι επιθυμητές, το OpenMP αποτελεί μία από τις πλέον κατάλληλες και ευέλικτες επιλογές.

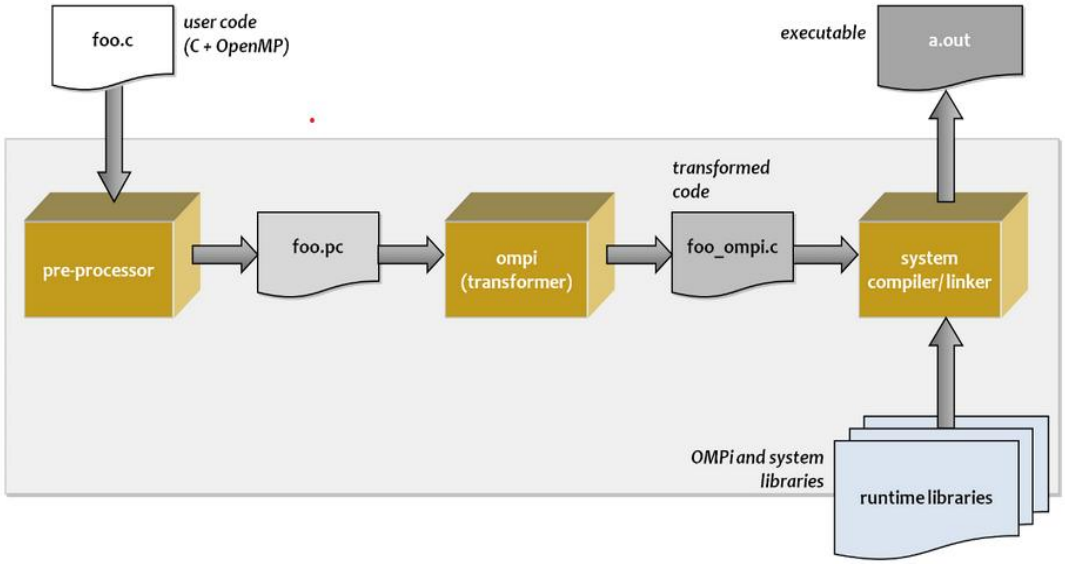
2.2 Εισαγωγή στον OMPi

Ο OMPi αποτελεί έναν ελαφρύ και ανοιχτού κώδικα μεταφραστή της γλώσσας C για το πρότυπο OpenMP, αναπτυγμένο στο Πανεπιστήμιο Ιωαννίνων. Η αρχιτεκτονική του είναι σχεδιασμένη με σκοπό την ευελιξία και την πειραματική χρήση διαφορετικών βιβλιοθηκών νημάτων (threading libraries), διατηρώντας παράλληλα υψηλή απόδοση κατά την εκτέλεση παράλληλων εφαρμογών.

Η λειτουργία του OMPi διαχωρίζεται σε δύο βασικά υποσυστήματα: το τμήμα μεταγλώττισης (compiler) και το σύστημα υποστήριξης εκτέλεσης (runtime system). Κατά την επεξεργασία του πηγαίου κώδικα, οι οδηγίες OpenMP μετασχηματίζονται σε ισοδύναμο πολυνηματικό C κώδικα, στον οποίο οι οδηγίες #pragma αντικαθίστανται από κατάλληλες κλήσεις σε συναρτήσεις του runtime συστήματος. Το τελικό αποτέλεσμα είναι εκτελέσιμος κώδικας, ο οποίος συνδέεται με τη βελτιστοποιημένη runtime βιβλιοθήκη του OMPi, προσφέροντας αποδοτική παράλληλη εκτέλεση.

Ένα βασικό χαρακτηριστικό του μεταφραστή είναι η αφαιρετική προσέγγιση που υιοθετεί για την έννοια του "νήματος". Η επιλογή του είδους των νημάτων που θα

χρησιμοποιηθούν (π.χ. POSIX threads, custom threads κ.λπ.) γίνεται δυναμικά κατά την εκτέλεση, μέσω των μηχανισμών και βιβλιοθηκών που παρέχει το σύστημα υποστήριξης εκτέλεσης. Ο OMPi έχει αποδειχθεί αξιόπιστος και φορητός, καθώς έχει δοκιμαστεί επιτυχώς σε πλήθος λειτουργικών συστημάτων, όπως Linux, Solaris, Irix και Windows (μέσω Cygwin), και μπορεί να προσαρμοστεί εύκολα σε οποιαδήποτε πλατφόρμα συμβατή με POSIX. Η όλη διαδικασία της μεταγλώττισης φαίνεται παρακάτω στο Σχήμα 2.1.



Σχήμα 2.1: Διαδικασία μετάφρασης του ompi.

2.3 Μετασχηματισμός πηγαίου σε πηγαίο κώδικα (source-to-source)

Ο μεταφραστής OMPi υλοποιεί τη μετατροπή πηγαίου κώδικα C με οδηγίες OpenMP σε ισοδύναμο πολυνηματικό κώδικα C, αντικαθιστώντας τις οδηγίες αυτές με κατάλληλες κλήσεις προς ρουτίνες του συστήματος υποστήριξης εκτέλεσης (runtime) του OMPi [3].

Η διαδικασία ξεκινά με τα κλασικά στάδια της λεκτικής και συντακτικής ανάλυσης, τα οποία είναι κοινά σε όλους τους μεταγλωττιστές. Στο πρώτο στάδιο, η λεκτική ανάλυση μετατρέπει τον κώδικα από ακολουθία χαρακτήρων σε ακολουθία λεκτικών μονάδων. Έπειτα, μέσω της συντακτικής ανάλυσης, παράγεται το συντακτικό δέντρο που αναπαριστά τη δομή του προγράμματος.

Ακολουθεί μια διάσχιση του συντακτικού δέντρου, κατά την οποία εντοπίζονται όλες οι οδηγίες OpenMP και αντικαθίστανται με αντίστοιχες κλήσεις σε συναρτήσεις του runtime του OMPi. Το αποτέλεσμα είναι ένας τροποποιημένος πηγαίος κώδικας C, στον

οποίο η παραλληλία επιτυγχάνεται αποκλειστικά μέσω των ρουτινών που παρέχει το σύστημα υποστήριξης εκτέλεσης του μεταφραστή.

Αυτή η ενδιάμεση μορφή του κώδικα αποτελεί το βασικό προϊόν της διαδικασίας μετάφρασης και είναι πλήρως απαλλαγμένη από τις αρχικές #pragma OpenMP οδηγίες.

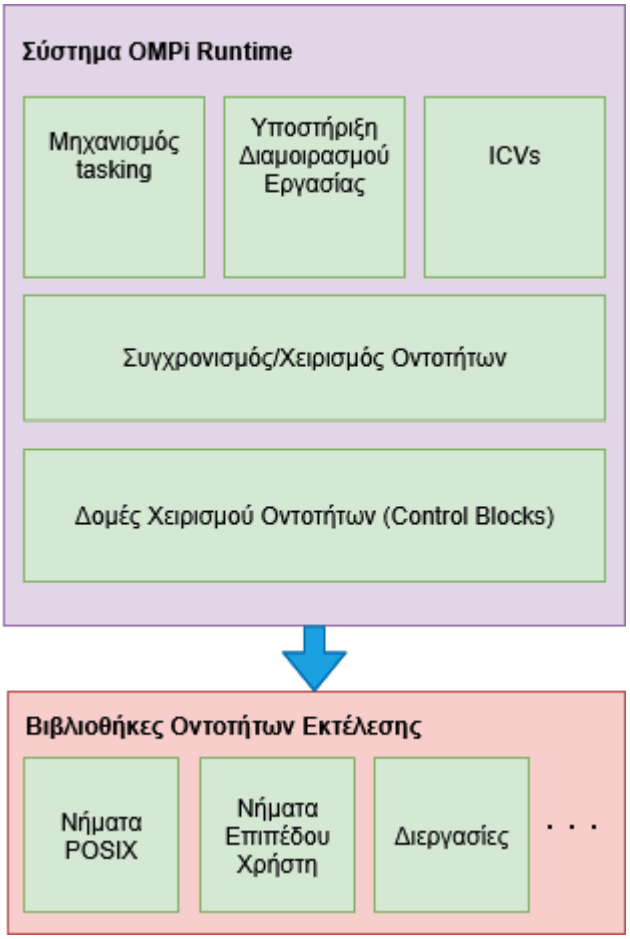
2.4 OMPi runtime

Το σύστημα υποστήριξης εκτέλεσης του OMPi (OMPi Runtime ή ORT) αποτελεί τον δεύτερο βασικό πυλώνα της αρχιτεκτονικής του μεταφραστή, πλάι στο μεταγλωττιστικό στάδιο. Είναι υπεύθυνο για την υποστήριξη της παράλληλης εκτέλεσης του κώδικα που έχει προκύψει από τη μετατροπή των οδηγιών OpenMP, προσφέροντας την απαραίτητη υποδομή για τον χειρισμό νημάτων, τη διαχείριση μεταβλητών περιβάλλοντος, καθώς και την υλοποίηση των απαιτούμενων βιβλιοθηκών του προτύπου όπως μπορούμε να παρατηρήσουμε και στο Σχήμα 2.2 [3].

Η αρχιτεκτονική του ORT είναι δομημένη με τρόπο modular και διακρίνεται σε δύο βασικά τμήματα: το τμήμα **host** και το τμήμα **devices**.

Το τμήμα **host** αφορά στο κύριο σύστημα στο οποίο εκτελείται ο κώδικας (host system). Παρέχει τη λειτουργικότητα για:

- το συντονισμό των νημάτων εκτέλεσης,
- την υλοποίηση των οδηγιών OpenMP με τις αντίστοιχες φράσεις τους,
- τον χειρισμό των μεταβλητών περιβάλλοντος,
- την υποστήριξη των συναρτήσεων runtime όπως ορίζονται από το πρότυπο OpenMP.



Σχήμα 2.2: Η δομή του OMPi runtime για το σύστημα host

Επιπλέον, το τμήμα host είναι υπεύθυνο για την επικοινωνία με τις συσκευές (devices). Μέσω μιας ενσωματωμένης διεπαφής (hostpart), αναλαμβάνει τη διαχείριση βασικών λειτουργιών κάθε συσκευής, όπως:

- αρχικοποίηση και τερματισμό της συσκευής,
- διαχείριση της μνήμης της,
- μεταφορά και εκτέλεση κώδικα σε αυτή.

Οι οντότητες εκτέλεσης (νήματα ή διεργασίες) δεν δημιουργούνται άμεσα από το ORT. Αντίθετα, το ORT βασίζεται σε ξεχωριστές και εξειδικευμένες βιβλιοθήκες, τις **Βιβλιοθήκες Οντοτήτων Εκτέλεσης (Execution Entity Libraries – EELIBS)**. Αυτές οι βιβλιοθήκες παρέχουν την ευελιξία επιλογής της επιθυμητής μορφής παραλληλίας (νήματα, διεργασίες κ.λπ.), ανάλογα με τις ανάγκες του προγραμματιστή ή την υποκείμενη αρχιτεκτονική. Αυτή η modular προσέγγιση επιτρέπει τη διαφανή υποκατάσταση ή επέκταση του τρόπου υλοποίησης των εκτελούμενων οντοτήτων.

Τα παραπάνω αφορούν στο σύστημα που υποστηρίζει την εκτέλεση στον βασικό επεξεργαστή. Ο OMPi διαθέτει ένα επιπλέον διακριτό τμήμα, αυτό των devices το οποίο

αφορά στην υποστήριξη των εντολών #target του OpenMP. Συγκεκριμένα, το τμήμα **devices** του ORT αναλαμβάνει την υποστήριξη της εκτέλεσης σε εξωτερικές ή συνοδές συσκευές (π.χ. GPU). Στον OMPi κάθε τύπος συσκευής ή module υποστηρίζεται από μια ξεχωριστή βιβλιοθήκη που διακρίνεται στα εξής υποσυστήματα:

- **hostpart**: το μέρος της συσκευής που επικοινωνεί με το κύριο σύστημα. Είναι υπεύθυνο για την ανταλλαγή δεδομένων και τον συγχρονισμό με την πλευρά του host.
- **devpart**: το μέρος της βιβλιοθήκης που υλοποιείται και εκτελείται εντός της συσκευής. Προσφέρει την υποστήριξη των οδηγιών OpenMP και των ρουτινών στο εσωτερικό των kernels που εκτελούνται στη συσκευή.

Το επίπεδο υποστήριξης και οι δυνατότητες κάθε συσκευής εξαρτώνται από την αντίστοιχη υλοποίηση της βιβλιοθήκης που τη συνοδεύει. Αυτή η αρχιτεκτονική προσφέρει μεγάλη ευελιξία, επιτρέποντας την εύκολη προσθήκη ή ενσωμάτωση νέων τύπων συσκευών στο σύστημα του OMPi. Για την ενσωμάτωση νέων συσκευών πρέπει να δημιουργηθεί νέο module το οποίο υλοποιεί τη διεπαφή του OMPi [11].

2.4.1 Περιοχές παραλληλίας σε συνδυασμό με target

Κατά τη διαδικασία του μετασχηματισμού, οι οδηγίες `#pragma omp parallel` του αρχικού πηγαίου κώδικα μετατρέπονται σε κλήσεις `ort_execute_parallel()`. Η βιβλιοθήκη από την οποία θα κληθεί η ρουτίνα ορίζεται από το σημείο εμφάνισης της οδηγίας. Για παράδειγμα, αν αυτή εμφανίζεται μέσα σε μια περιοχή target η οποία αφορά τη συσκευή Α, τότε η βιβλιοθήκη που θα χρησιμοποιηθεί θα είναι αυτή της υποστήριξης εκτέλεσης της Α.

2.4.2 Είσοδος σε περιοχή target.

Κατά τη διαδικασία του μετασχηματισμού, οι οδηγίες #pragma omp target του αρχικού πηγαίου κώδικα μετατρέπονται σε ακολουθία κλήσεων προς τη runtime βιβλιοθήκη του OMPi. Ο σκοπός των κλήσεων αυτών είναι να εξασφαλίσουν την ορθή μεταφορά των δεδομένων από και προς τη συσκευή, καθώς και την εκτέλεση του δομημένου τμήματος της περιοχής target στο κατάλληλο περιβάλλον. Ο παρακάτω ψευδοκώδικας παρουσιάζει τα βασικά βήματα του μετασχηματισμού, όπως εφαρμόζονται από τον OMPi, για την εξής απλή περίπτωση:

```
int x = 5;
#pragma omp target map(tofrom: x)
{
```

```
    x = x * 2;
}
printf("x = %d\n", x);
```

Στον μετασχηματισμένο ψευδοκώδικα, η είσοδος στην περιοχή target σηματοδοτεί την αρχικοποίηση του περιβάλλοντος ανταλλαγής δεδομένων με τη συσκευή, μέσω μιας ρουτίνας όπως start_target_data(). Η μεταβλητή x, που αναφέρεται με το χαρακτηριστικό map(tofrom:), δηλώνεται ώστε να μεταφερθεί στη συσκευή στην αρχή και να επιστραφεί πίσω στον host μετά την εκτέλεση. Η λειτουργία αυτή περιγράφεται με την αφαιρετική συνάρτηση handle_data_mapping_to_device(...).

Στη συνέχεια, δημιουργείται μια προσωρινή δομή δεδομένων dev_data, η οποία περιέχει τους απαραίτητους δείκτες προς τις μεταβλητές που θα προσπελαστούν από τον κώδικα της συσκευής. Η διεύθυνση της μεταβλητής x μεταφράζεται κατάλληλα μέσω μιας εσωτερικής λειτουργίας (όπως host2device_address_translation()), ώστε να μπορεί να χρησιμοποιηθεί από τον kernel.

Το δομημένο τμήμα του target (στην περίπτωσή μας η πράξη x = x * 2) υλοποιείται ως ξεχωριστή συνάρτηση, η οποία ορίζεται και μεταγλωττίζεται για εκτέλεση στη συσκευή. Η εκκίνηση αυτής της συνάρτησης πραγματοποιείται μέσω μιας κλήσης offload_kernel(...), η οποία αντιστοιχεί στην κλήση ort_offload_kernel(...) της βιβλιοθήκης του OMPi.

Μετά την ολοκλήρωση της εκτέλεσης στη συσκευή, η δομή dev_data αποδεσμεύεται, και οι τιμές των μεταβλητών που είχαν χαρακτηριστεί ως map(from:) ή map(tofrom:) αντιγράφονται πίσω στον host, ώστε να είναι διαθέσιμες για την υπόλοιπη εκτέλεση του προγράμματος. Η όλη διαδικασία ολοκληρώνεται με την κλήση end_target_data(), η οποία τερματίζει το περιβάλλον μεταφοράς δεδομένων.

Ο συνοπτικός ψευδοκώδικας του μετασχηματισμού έχει ως εξής:

```
int __original_main(...) {
    int x = 5;
    device_env = start_target_data();
    handle_data_mapping_to_device(&x, sizeof(x), TOFROM);
    dev_data = allocate_device_struct();
    dev_data->x = translate_address(&x);
    dev_data->_x_offset = 0;
    offload_kernel(kernel_wrapper_function, dev_data);
```

```
    free_device_struct(dev_data);
    handle_data_unmapping_from_device(&x);
    end_target_data(device_env);
    printf("x = %d\n", x);
    return 0;
}
```

Αξίζει να σημειωθεί ότι οι συναρτήσεις που εμφανίζονται παραπάνω δεν υφίστανται αυτούσιες στον παραγόμενο κώδικα, αλλά χρησιμοποιούνται για λόγους απλοποίησης και καλύτερης κατανόησης της διαδικασίας μετασχηματισμού. Οι πραγματικές συναρτήσεις που παράγονται από τον OMPi φέρουν ονόματα όπως _ort_map_tdvar(...), _ort_offload_kernel(...), κ.ά., και εξειδικεύονται ανάλογα με τη στοχευμένη συσκευή και την υποστήριξη της βιβλιοθήκης εκτέλεσης.

2.4.3 Υποστήριξη συσκευών OpenMP στον OMPi

Ο OMPi υποστηρίζει την εκτέλεση περιοχών target σε συσκευές, όπως επιταχυντές, GPU ή CPU, σύμφωνα με τις προδιαγραφές του προτύπου OpenMP. Η υποστήριξη αυτή επιτυγχάνεται μέσω μιας σαφώς καθορισμένης διεπαφής μεταξύ του host και των συσκευών, η οποία διαφέρει ανάλογα με τον τύπο και την υλοποίηση κάθε συσκευής [9].

Κατά την ανίχνευση μιας περιοχής target στον κώδικα του χρήστη, ο αντίστοιχος κώδικας μετασχηματίζεται ώστε να πραγματοποιεί τις απαραίτητες κλήσεις προς τις βιβλιοθήκες υποστήριξης της εκάστοτε συσκευής. Ο μετασχηματισμένος κώδικας εξάγεται σε ξεχωριστό αρχείο πηγαίου κώδικα, διασφαλίζοντας έτσι την ανεξάρτητη μεταγλώττισή του με κατάλληλο compiler που ορίζει η συγκεκριμένη συσκευή. Το αρχείο αυτό περιέχει υλοποιήσεις οδηγιών OpenMP σε μορφή C, καθώς και τις απαραίτητες κλήσεις στις συναρτήσεις της συσκευής, ώστε να παράγεται ένα αυτόνομο εκτελέσιμο αρχείο πυρήνα (kernel).

Ο μετασχηματισμένος κώδικας ενσωματώνει την έννοια της διεπαφής **hostpart**, η οποία παρέχει βασικές λειτουργίες όπως αρχικοποίηση και τερματισμό της συσκευής, διαχείριση μνήμης και φόρτωση του κώδικα. Ο αντίστοιχος κώδικας kernel που δημιουργείται, καλεί λειτουργίες της βιβλιοθήκης **devpart**, οι οποίες βρίσκονται στη συσκευή και αναλαμβάνουν την εκτέλεση των οδηγιών OpenMP.

Για τη διαχείριση δεδομένων μεταξύ του host και της συσκευής, εισάγεται η έννοια των **ενδιάμεσων διευθύνσεων**. Επειδή σε πολλές περιπτώσεις οι συσκευές δεν

μοιράζονται κοινό χώρο διευθύνσεων με τον host, οι αρχικές διευθύνσεις δεδομένων μετατρέπονται πρώτα σε ενδιάμεσες. Ο OMPi αποθηκεύει αυτές τις ενδιάμεσες τιμές για μελλοντική χρήση, ενώ στη συσκευή οι διευθύνσεις αυτές μεταφράζονται σε τοπικές, προκειμένου να είναι προσβάσιμες από τον kernel.

Η διαδικασία μεταγλώττισης κάθε πηγαίου αρχείου target καθορίζεται από ένα ειδικό **Makefile**. Για κάθε διαφορετική συσκευή που υποστηρίζεται από το σύστημα, απαιτείται και ένα αντίστοιχο Makefile. Αν το πρόγραμμα περιέχει Χ περιοχές target και υπάρχουν Υ διαφορετικές συσκευές εγκατεστημένες, το σύστημα παράγει συνολικά Χ*Υ εκτελέσιμα αρχεία πυρήνων, ένα για κάθε συνδυασμό περιοχής και συσκευής.

Η αρχιτεκτονική αυτή προσφέρει σημαντική ευελιξία και επεκτασιμότητα, επιτρέποντας την εύκολη ενσωμάτωση νέων συσκευών στο σύστημα, καθώς και τη στοχευμένη εκμετάλλευση των δυνατοτήτων καθεμίας μέσω των κατάλληλων βιβλιοθηκών.

2.4.4 Η διεπαφή hostpart

Η διεπαφή **hostpart** αποτελεί αναπόσπαστο τμήμα του μηχανισμού υποστήριξης συσκευών στον OMPi και επιτρέπει την επικοινωνία μεταξύ του host και των εξωτερικών συσκευών. Η λειτουργία της βασίζεται σε μια κοινή προγραμματιστική διεπαφή (API), η οποία όμως υλοποιείται ξεχωριστά για κάθε συσκευή, λόγω των διαφορετικών χαρακτηριστικών που μπορεί να παρουσιάζουν.

Όταν ο host συναντήσει μια οδηγία target, ενεργοποιεί τη διεπαφή **hostpart** ώστε να εκτελέσει τις εξής βασικές λειτουργίες:

- αρχικοποίηση της συσκευής
- προετοιμασία κοινόχρηστων δομών επικοινωνίας
- μεταφορά και εκτέλεση του κώδικα πυρήνα, καθώς
- τερματισμό της συσκευής με το πέρας της περιοχής.

Η διεπαφή **hostpart** λειτουργεί ως το βασικό εργαλείο σύνδεσης του OMPi με ετερογενείς υπολογιστικές μονάδες και παρέχει τα απαραίτητα εργαλεία για τη σωστή διαχείριση των πόρων και των δεδομένων που εμπλέκονται στις περιοχές target.

Κεφάλαιο 3. Remote Offloading

3.1 Εισαγωγή

Στο πλαίσιο του σύγχρονου παράλληλου και κατανεμημένου υπολογισμού, το remote offloading αποτελεί μια κρίσιμη τεχνολογία που στοχεύει στη βέλτιστη αξιοποίηση των διαθέσιμων υπολογιστικών πόρων μέσω της απομακρυσμένης εκτέλεσης τμημάτων ενός προγράμματος. Συγκεκριμένα, η τεχνική αυτή επιτρέπει την εκφόρτωση υπολογιστικών φορτίων από τον κύριο υπολογιστικό κόμβο (host) σε άλλες, απομακρυσμένες συσκευές όπως CPUs, GPUs ή εξειδικευμένους επιταχυντές που είναι συνδεδεμένες μέσω δικτύου ή ανήκουν σε διαφορετικούς κόμβους ενός υπολογιστικού cluster.

Η ανάγκη για remote offloading έχει αναδειχθεί ως ιδιαίτερα σημαντική τα τελευταία χρόνια, εξαιτίας της εκρηκτικής αύξησης των δεδομένων, της πολυπλοκότητας των εφαρμογών, αλλά και της διάδοσης υβριδικών αρχιτεκτονικών που περιλαμβάνουν ετερογενείς υπολογιστικές μονάδες. Σε τέτοια περιβάλλοντα, η δυνατότητα μεταφοράς μέρους του υπολογιστικού έργου σε απομακρυσμένες μονάδες προσφέρει σημαντικά πλεονεκτήματα όσον αφορά την επέκταση της υπολογιστικής απόδοσης, την ευελιξία στην κατανομή πόρων, και την ανθεκτικότητα του συστήματος σε αποτυχίες.

Το remote offloading δεν περιορίζεται μόνο στην αξιοποίηση εξειδικευμένων επιταχυντών σε τοπικό επίπεδο, αλλά αποτελεί και κεντρική στρατηγική σε υπολογιστικά clusters και υποδομές υψηλών επιδόσεων (High Performance Computing - HPC). Σε αυτά τα περιβάλλοντα, οι κόμβοι είναι συνήθως πολυπύρηνοι υπολογιστές με δυνατότητες παράλληλης επεξεργασίας, οι οποίοι συνεργάζονται μέσω δικτύου. Η υποστήριξη απομακρυσμένης εκφόρτωσης επιτρέπει την αποδοτική εξισορρόπηση φόρτου μεταξύ των κόμβων και την εκμετάλλευση αδρανούς υπολογιστικής ισχύος, αυξάνοντας την αποδοτικότητα και μειώνοντας τους χρόνους εκτέλεσης.

Συγκεκριμένα, το remote offloading καθίσταται απαραίτητο:

- Για την εκμετάλλευση υπολογιστικών πόρων που παραμένουν αδρανείς σε άλλους κόμβους του cluster.
- Για την αποδοτική κατανομή του υπολογιστικού φόρτου μεταξύ host και απομακρυσμένων συσκευών, επιτυγχάνοντας υψηλή αποδοτικότητα.
- Για την υποστήριξη κατανεμημένων εφαρμογών μεγάλης κλίμακας, όπου τα δεδομένα και οι εργασίες δεν μπορούν να περιοριστούν σε έναν μόνο κόμβο.
- Για την ενίσχυση της αξιοπιστίας και ανθεκτικότητας του συστήματος, καθώς σε περίπτωση αστοχίας τοπικών πόρων μπορεί να χρησιμοποιηθούν εναλλακτικές απομακρυσμένες μονάδες.

Κατά συνέπεια, η υποστήριξη remote offloading σε ένα υπολογιστικό σύστημα, και ιδιαίτερα σε clusters, αποτελεί θεμελιώδη τεχνολογικό μηχανισμό για την υλοποίηση επεκτάσιμων και αποδοτικών υπολογιστικών λύσεων.

3.2 Υλοποιήσεις του μηχανισμού του remote offloading

Η υλοποίηση ενός πλήρους μηχανισμού remote offloading αποτελεί μια ιδιαίτερα απαιτητική και πολύπλοκη διαδικασία, η οποία υπερβαίνει κατά πολύ τις τυπικές δυνατότητες ενός παραδοσιακού μεταγλωττιστή. Σε αντίθεση με το κλασικό offloading σε τοπικές συσκευές (όπως μια GPU στον ίδιο κόμβο), η απομακρυσμένη εκτέλεση απαιτεί επιπλέον υποσυστήματα, όπως μηχανισμούς επικοινωνίας δικτύου, μεταφοράς δεδομένων, συγχρονισμού εκτέλεσης, καθώς και δυναμικής διαχείρισης πόρων σε απομακρυσμένους κόμβους.

Για τον λόγο αυτό, οι υλοποιήσεις του remote offloading είναι περιορισμένες, και σε μεγάλο βαθμό παραμένουν στο πλαίσιο ερευνητικών ή ειδικά προσαρμοσμένων έργων. Δεν υπάρχει ακόμα ευρεία υποστήριξη σε δημοφιλείς παραγωγικούς μεταφραστές.

Μέχρι σήμερα, ο OMPi αποτελεί τον μοναδικό μεταφραστή που υποστηρίζει πλήρως και εγγενώς το remote offloading, προσφέροντας ένα ενοποιημένο runtime και μηχανισμούς αποστολής παράλληλων περιοχών σε απομακρυσμένους κόμβους, βασισμένους στο πρότυπο OpenMP. Παράλληλα, το οικοσύστημα LLVM/Clang δίνει τη δυνατότητα ανάπτυξης παρόμοιων δυνατοτήτων μέσω custom υποδομών, αλλά χωρίς να διαθέτει άμεση ή αυτόνομη υποστήριξη.

Στην παρούσα ενότητα εξετάζονται οι διαθέσιμες υλοποιήσεις του μηχανισμού remote offloading, με στόχο την κατανόηση των αρχιτεκτονικών επιλογών και των τεχνικών

προκλήσεων που σχετίζονται με την απομακρυσμένη εκτέλεση παράλληλου κώδικα σε κατανεμημένα περιβάλλοντα.

3.2.1 – Η περίπτωση του LLVM

Ο LLVM είναι ένας από τους πιο διαδεδομένους μεταγλωττιστές με υποστήριξη για parallel και offloading extensions μέσω του Clang/LLVM OpenMP. Παρέχει βασική υποστήριξη για target offloading, συμπεριλαμβανομένης της δυνατότητας εκτέλεσης σε συσκευές όπως NVIDIA GPU μέσω CUDA ή AMD GPU μέσω ROCm. Ωστόσο, η υποστήριξη για remote device offloading, δηλαδή εκφόρτωση σε μη τοπικές συσκευές, είναι σε πειραματικό και περιορισμένο στάδιο [4].

Ο μηχανισμός remote offloading στον LLVM βασίζεται σε:

- Target plugins, όπως libomptarget και libomptarget.rtl.remote (ή custom υλοποιήσεις)[5].
- Τη χρήση μηχανισμών για αποστολή δεδομένων (π.χ. TCP/IP ή gRPC) για επικοινωνία με απομακρυσμένες συσκευές, με τη βοήθεια των βιβλιοθηκών [5].
- Ανάγκη για χειροκίνητη ρύθμιση του runtime, συμπεριλαμβανομένων μεταβλητών περιβάλλοντος, υποστήριξη από τη συσκευή και αντίστοιχου server daemon [6].

Παρά την ισχυρή αρχιτεκτονική του LLVM, η χρήση του remote offloading μηχανισμού παραμένει τεχνικά δύσκολη και μη φιλική για τον τελικό προγραμματιστή [6]. Ορισμένα βασικά προβλήματα είναι:

- Χειροκίνητη διαχείριση των απομακρυσμένων πόρων και του runtime.
- Απουσία δυναμικής επιλογής στόχου: Με βάση το φορτίο ή τη διαθεσιμότητα.
- Πειραματική φύση: Η υποστήριξη του remote offloading θεωρείται ακόμα πειραματική.
- Περιορισμένη ταυτόχρονη εκτέλεση: Δεν υποστηρίζει ταυτόχρονες εκτελέσεις από πολλαπλές εφαρμογές, καθώς είναι συγχρονισμένος και τερματίζεται μετά από μία μόνο εκτέλεση.
- Απαιτήσεις σε βιβλιοθήκες: Η λειτουργία του plugin εξαρτάται από την παρουσία των απαραίτητων βιβλιοθηκών και plugins στον απομακρυσμένο κόμβο.

Αυτό το επίπεδο πολυπλοκότητας περιορίζει την ευρεία χρήση του μηχανισμού και τον καθιστά δύσκολα αξιοποιήσιμο εκτός πολύ εξειδικευμένων περιβαλλόντων. Επιπλέον, η μοναδική διαθέσιμη υλοποίηση για απομακρυσμένο offloading είναι μέσω

του gRPC plugin το οποίο δεν περιλαμβάνεται πλέον στις δημόσιες εκδόσεις. Παρότι παλαιότερα έγινε προσπάθεια εγκατάστασης παλαιότερων εκδόσεων του μεταφραστή, δεν κατέστη δυνατό να επιδιορθωθεί το plugin ώστε να λειτουργήσει ο μηχανισμός και να πραγματοποιηθεί κάποια σύγκριση με τον OMPi [10].

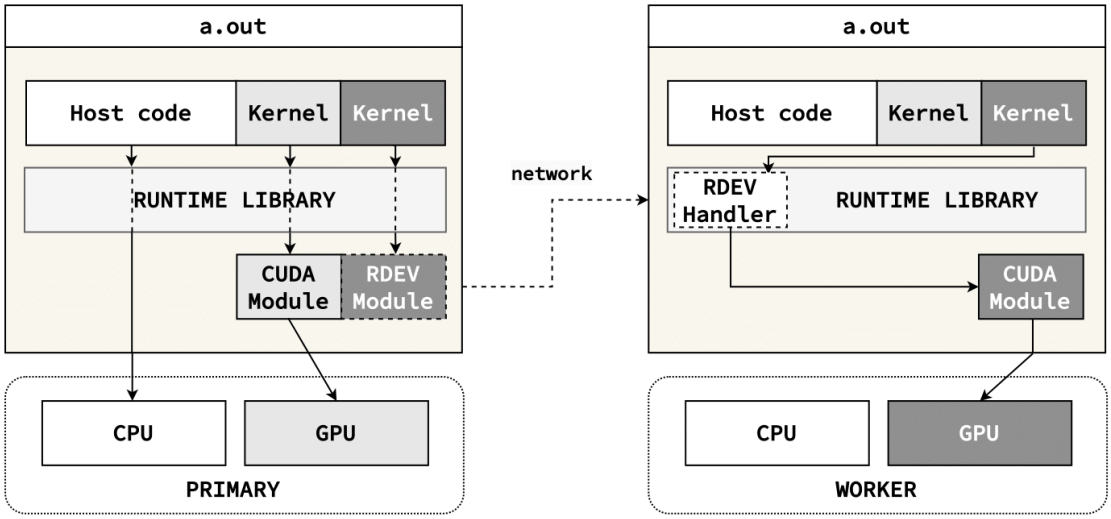
3.2.2 Remote offloading στον OMPi

Ο μηχανισμός remote offloading του OMPi επιτρέπει την εκτέλεση τμημάτων ενός OpenMP προγράμματος σε απομακρυσμένους κόμβους ενός υπολογιστικού συμπλέγματος, σαν να ήταν τοπικές συσκευές [10]. Αυτό επιτυγχάνεται μέσω της εικονικοποίησης των απομακρυσμένων πόρων (όπως CPUs και GPUs), οι οποίοι παρουσιάζονται στον προγραμματιστή ως απλά OpenMP "devices", διαθέσιμα μέσω των τυπικών OpenMP οδηγιών (#pragma omp target).

Η αρχιτεκτονική ακολουθεί ένα μοντέλο master–worker, στο οποίο:

- Ο master κόμβος είναι ο κόμβος που εκτελεί το κύριο μέρος του προγράμματος.
- Οι worker κόμβοι είναι αυτοί στους οποίους γίνεται απομακρυσμένο offloading, δηλαδή εκτελούν τις εντολές του master.

Όταν εκτελείται μια οδηγία target με συγκεκριμένο device, ο OMPi εντοπίζει αν η συγκεκριμένη συσκευή είναι απομακρυσμένη. Αν ναι, με χρήση του προτύπου MPI αποστέλλονται τα απαραίτητα δεδομένα και τις παραμέτρους στον αντίστοιχο worker κόμβο, ο οποίος εκτελεί τον κώδικα τοπικά στη δική του συσκευή (CPU ή GPU) όπως φαίνεται στο σχήμα 3.1. Μετά την ολοκλήρωση της εκτέλεσης, τα αποτελέσματα επιστρέφουν στον master κόμβο, χωρίς ο χρήστης να χρειάζεται να διαχειριστεί το communication layer.



Σχήμα 3.1: Διαδικασία remote offloading με εικονικοποίηση συσκευών [10].

Η διαδικασία αυτή γίνεται διαφανώς για τον χρήστη και δεν απαιτεί αλλαγές στη λογική του προγράμματος, πέρα από τη χρήση των γνωστών OpenMP οδηγιών. Ο OMPi διαχειρίζεται εσωτερικά τη δημιουργία των κατάλληλων δομών, την επικοινωνία μεταξύ κόμβων και την αντιστοίχιση των συσκευών με τις οδηγίες του χρήστη.

Ο χρήστης μπορεί να επιλέξει ποια απομακρυσμένη συσκευή θα χρησιμοποιήσει, είτε με χρήση των αριθμητικών IDs των συσκευών κατευθείαν στον κώδικα της εκάστοτε εφαρμογής, είτε με χρήση της ειδικής διεπαφής του OMPi που επιτρέπει πιο αφηρημένη διαχείριση των πόρων. Επιπλέον, το σύστημα διασφαλίζει ότι οι απομακρυσμένοι πόροι αξιοποιούνται μόνο όταν είναι απαραίτητοι, ώστε να αποφεύγεται περιττό overhead κατά την εκτέλεση.

Με αυτό τον τρόπο, το remote offloading στον OMPi επιτρέπει την επέκταση του μοντέλου του OpenMP πέρα από έναν μόνο κόμβο, αξιοποιώντας αποδοτικά τα διαθέσιμα υπολογιστικά μέσα ενός ολόκληρου συμπλέγματος.

3.3 Ανάγκη για profiling και scheduling

Η υλοποίηση ενός αποδοτικού μηχανισμού remote offloading, όπως φάνηκε από τις παραπάνω προσεγγίσεις, συνοδεύεται από σημαντικές τεχνικές προκλήσεις, τόσο στον σχεδιασμό όσο και στην εκτέλεση. Ο μεγάλος όγκος απαιτούμενης υποδομής, η ανάγκη για έλεγχο των ενδιάμεσων χρόνων (όπως οι καθυστερήσεις μεταφοράς δεδομένων), αλλά και η ύπαρξη πολλαπλών ετερογενών απομακρυσμένων συσκευών, καθιστούν δύσκολη τη βέλτιστη και δυναμική αξιοποίηση των διαθέσιμων πόρων.

Επιπλέον, η απουσία μηχανισμών για αποδοτικό διαμοιρασμό των kernels, ανάλογα με τα χαρακτηριστικά του κώδικα και των απομακρυσμένων κόμβων, οδηγεί σε πιθανά φαινόμενα υποχρησιμοποίησης ή υπερφόρτωσης συγκεκριμένων πόρων. Η στατική εκχώρηση περιοχών προς εκτέλεση, χωρίς δυναμική αξιολόγηση της απόδοσης ή πρόβλεψης του χρόνου ολοκλήρωσης, περιορίζει την αποτελεσματικότητα του remote offloading, ειδικά σε κατανεμημένα περιβάλλοντα με μεταβαλλόμενη διαθεσιμότητα.

Σε αυτό το πλαίσιο, αναδεικνύεται η αναγκαιότητα για την ενσωμάτωση μηχανισμών profiling και scheduling. Η δυνατότητα δυναμικής παρακολούθησης (profiling) της απόδοσης των kernels και των μεταφορών δεδομένων επιτρέπει τη συλλογή κρίσιμων μετρήσεων (latency, bandwidth, CPU/GPU load), οι οποίες μπορούν να αξιοποιηθούν από μηχανισμούς χρονοπρογραμματισμού (scheduling) για τη βέλτιστη κατανομή των εργασιών σε διαθέσιμους κόμβους.

Συνοψίζοντας, η ύπαρξη μηχανισμών profiling και scheduling δεν αποτελεί πολυτέλεια αλλά αναγκαιότητα για την πρακτική και αποδοτική υλοποίηση του remote offloading.

Η πραγματική δυναμική του remote offloading δεν περιορίζεται μόνο στη μεταφορά υπολογιστικών φορτίων, αλλά αναδεικνύεται πλήρως μέσα από την έξυπνη παρακολούθηση και προγραμματισμό της εκτέλεσης. Η κατεύθυνση προς πιο αυτοματοποιημένα και προσαρμοστικά περιβάλλοντα είναι θεμελιώδης για την εξέλιξη και ευρεία υιοθέτηση της τεχνολογίας αυτής στο μέλλον.

Κεφάλαιο 4. Υλοποίηση στον μεταφραστή OMPi

4.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η υλοποίηση που πραγματοποιήθηκε στον μεταφραστή OMPi του Πανεπιστημίου Ιωαννίνων, με στόχο τη βελτίωση της διαχείρισης απομακρυσμένης εκτέλεσης (remote offloading) μέσω μηχανισμών **profiling** και **scheduling**.

Αρχικά, ενσωματώθηκε μηχανισμός **profiling** εντός της βιβλιοθήκης *runtime* του OMPi, με σκοπό την ανάλυση του χρόνου εκτέλεσης κατά τη διαδικασία offloading. Ο στόχος ήταν να εντοπιστούν τα στάδια του offload (όπως μεταφορά δεδομένων, εκκίνηση του kernel, επιστροφή αποτελεσμάτων) που επιβαρύνουν περισσότερο τη συνολική απόδοση. Μέσα από αυτή τη διαδικασία κατέστη εφικτή η εξαγωγή κρίσιμων μετρικών χρόνου για κάθε επιμέρους φάση.

Στη συνέχεια, αναπτύχθηκε μια **αρχική εκδοχή scheduler**, η οποία πραγματοποιεί **διαμοίραση των kernels** προς εκτέλεση στους διαθέσιμους υπολογιστικούς πόρους (devices) εντός των κόμβων ενός cluster.

Η παρούσα υλοποίηση λειτουργεί ως πρώτο βήμα προς την κατεύθυνση ενός ολοκληρωμένου μηχανισμού δυναμικού προγραμματισμού (dynamic scheduling) στο περιβάλλον του OMPi, ανοίγοντας τον δρόμο για περαιτέρω εξελίξεις σε περιβάλλοντα ετερογενών και απομακρυσμένων συστημάτων.

4.2 Profiling στο OMPi runtime

Ο βασικός στόχος της ενσωμάτωσης μηχανισμού *profiling* στον OMPi ήταν η χρονομέτρηση όλων των κρίσιμων σταδίων της διαδικασίας remote offloading, με

σκοπό την πλήρη κατανόηση της συμπεριφοράς και της απόδοσης του συστήματος. Η μέτρηση του χρόνου εκτέλεσης σε φάσεις όπως η μεταφορά δεδομένων προς τον απομακρυσμένο κόμβο, η εκτέλεση του kernel, καθώς και η επιστροφή των αποτελεσμάτων, επιτρέπει τον εντοπισμό των λεγόμενων *bottlenecks*, δηλαδή των σημείων εκείνων που επιβαρύνουν δυσανάλογα τον συνολικό χρόνο εκτέλεσης.

Η ανάγκη για τέτοια ανάλυση γίνεται ιδιαίτερα σημαντική σε κατανεμημένα περιβάλλοντα όπως τα clusters, όπου η απόδοση μπορεί να επηρεάζεται από πολλούς παράγοντες, όπως η συμφόρηση στο δίκτυο, η επιβάρυνση των κόμβων ή η μη βέλτιστη χρήση των διαθέσιμων πόρων.

Επιπλέον, τα αποτελέσματα του profiling δεν χρησιμοποιούνται μόνο για στατιστική παρατήρηση, αλλά μπορούν να αποτελέσουν τη βάση για μελλοντικές στρατηγικές scheduling. Συγκεκριμένα, η καταγραφή χρόνων εκτέλεσης και κόστους επικοινωνίας μπορεί να χρησιμοποιηθεί από έναν scheduler ώστε να λαμβάνει καλύτερες αποφάσεις για την επιλογή του καταλληλότερου απομακρυσμένου κόμβου ή συσκευής για κάθε εργασία. Έτσι, το profiling λειτουργεί και ως μηχανισμός πληροφόρησης για βελτιστοποίηση τόσο σε επίπεδο απόδοσης όσο και σε επίπεδο δυναμικής κατανομής φορτίου.

Συνολικά, η υλοποίηση του profiling αποτελεί το πρώτο βήμα για την ενίσχυση της αποδοτικότητας του remote offloading στον OMPi, προσφέροντας χρήσιμα δεδομένα για την παρακολούθηση, την ανάλυση και, τελικά, τη βελτίωση της συμπεριφοράς του συστήματος κατά την απομακρυσμένη εκτέλεση υπολογιστικά εντατικών εργασιών.

4.2.1 Ενσωμάτωση Profiling στο OMPi runtime

Η υλοποίηση έγινε με στόχο την ελάχιστη παρέμβαση στον υπάρχοντα κώδικα του OMPi runtime. Με βάση αυτή τη λογική η ενσωμάτωση του μηχανισμού profiling βασίζεται στα εξής :

- Ορισμό ελαφριάς δομής αποθήκευσης χρονισμών (TimingInfo) για κάθε συνάρτηση που χρονομετρούμε.
- Καταγραφή σε στατικό πίνακα, με ασφάλεια πρόσβασης (locks) για υποστήριξη πολυνηματικών περιβαλλόντων.
- Χρήση μακροεντολών για εύκολη έναρξη και λήξη χρονομέτρησης, χωρίς να αλλάζει η λογική του κώδικα.
- Δυνατότητα εκτύπωσης ή εξαγωγής σε αρχείο στο τέλος της εκτέλεσης, για μεταγενέστερη ανάλυση.

4.2.2 Δομή αποθήκευσης δεδομένων

Η δομή TimingInfo αποτελεί τη βασική μονάδα αποθήκευσης πληροφοριών εκτέλεσης για κάθε χρονισμένη συνάρτηση. Περιλαμβάνει:

- Τον χρόνο εκτέλεσης.
- Το όνομα της συνάρτησης την οποία χρονομετρούμε (function_name).
- Το όνομα της εσωτερικής συνάρτησης από όπου έγινε η κλήση (func_in_name).
- Το path, το οποίο προορίζεται για μελλοντική χρήση (π.χ. καταγραφή θέσης αρχείου ή τύπου offload).
- Την χρονική στιγμή έναρξης της χρονομέτρησης.
- Την χρονική στιγμή λήξης της χρονομέτρησης.

Η δομή με την βοήθεια ενός κοινού δείκτη λειτουργεί ως στατική προσωρινή αποθήκη για τις τιμές που συλλέγονται κατά την εκτέλεση του runtime.

4.2.3 Συνάρτηση καταγραφής και μακροεντολές χρονισμού

Η συνάρτηση log_timing_info εισάγει με ασφάλεια τα δεδομένα της χρονισμένης εκτέλεσης στη προανφερθείσα δομή, χρησιμοποιώντας μηχανισμό κλειδώματος για αποφυγή ταυτόχρονων εγγραφών σε πολυνηματικά περιβάλλοντα.

Η χρήση δείκτη εξασφαλίζει κυκλική προσπέλαση του πίνακα έως το μέγιστο μέγεθος που έχει οριστεί για τη δομή. Σε περίπτωση υπερχείλισης, εκτυπώνεται προειδοποιητικό μήνυμα στο τερματικό του χρήστη.

Οι μακροεντολές οι οποίες προστέθηκαν μπορούν να ενσωματωθούν σε οποιοδήποτε block κώδικα στο runtime για να μετρήσουν τη διάρκειά εκτέλεσης μιας συνάρτησης. Η υλοποίηση βασίζεται στη συνάρτηση gettimeofday, η οποία καταγράφει τον χρόνο με ακρίβεια μικροδευτερολέπτου.

Παράδειγμα χρήσης:

```
START_TIMING_LOG
// Κώδικας αποστολής δεδομένων σε απομακρυσμένο κόμβο
END_TIMING_LOG("send_data_to_remote_node function name");
```

Με αυτόν τον τρόπο, καταγράφεται ο χρόνος που απαιτείται για κρίσιμες λειτουργίες, όπως επικοινωνία, αποστολή, λήψη και ενεργοποίηση απομακρυσμένων πυρήνων (kernels). Πιο αναλυτικά, η START_TIMING_LOG ξεκινάει τη χρονομέτρηση, ενώ η END_TIMING_LOG τη τερματίζει και καλεί τη συνάρτηση την οποία καταγράφει τα δεδομένα στη στατική δομή.

4.2.4 Εκτύπωση αποτελεσμάτων profiling

Για την εκτύπωση των αποτελεσμάτων προστέθηκε μία ακόμα μακροεντολή PRINTFUNCTIME, η οποία παρέχει δυνατότητα εκτύπωσης των αποτελεσμάτων χρονομέτρησης σε οποιοδήποτε stream (π.χ. stderr, stdout, αρχείο). Για κάθε εγγραφή εμφανίζει το όνομα της συνάρτησης από την οποία κλήθηκαν οι μακροεντολές για τη χρονομέτρηση καθώς και τον αντίστοιχο χρόνο εκτέλεσης και το όνομα της συνάρτησης που χρονομετρήσαμε.

Παράδειγμα:

```
PRINTFUNCTIME(stdout,"%s\n","Remote devices timings: ")
```

4.2.5 Παράδειγμα profiling για offload σε local/remote devices.

Σε αυτή την υποενότητα εξετάζουμε τη χρήση του μηχανισμού profiling για τη σύγκριση της απόδοσης offloading σε τοπική και απομακρυσμένη συσκευή. Ο κώδικας που χρησιμοποιήθηκε εκτελεί τον πολλαπλασιασμό δύο πινάκων επανειλημμένα, αποθηκεύοντας το αποτέλεσμα σε έναν τρίτο πίνακα. Για την εκτέλεση αυτής της διεργασίας, απαιτείται η μεταφορά των τριών πινάκων στη συσκευή (device), η εκτέλεση του υπολογισμού, και στη συνέχεια η επιστροφή του αποτελέσματος στον host.

Παρακάτω παρατίθενται ενδεικτικά αποτελέσματα από το μηχανισμό καταγραφής χρόνου:

- Local Device Offload :

Fucntions	time
_ort_map_tdvar, time (map_var)	0.000001
_ort_map_tdvar, time (map_var)	0.000001
_ort_map_tdvar, time (map_var)	0.000001
offload, time ((*kernel_function)(devdata))	0.003725
_ort_unmap_tdvar, time (unmap_var)	0. 000004
_ort_unmap_tdvar, time (unmap_var)	0. 000004
_ort_unmap_tdvar, time (unmap_var)	0. 000004

- Remote Device Offload :

Fucntions	time
_ort_map_tdvar, time (map_var)	0. 000853
_ort_map_tdvar, time (map_var)	0. 000002
_ort_map_tdvar, time (map_var)	0. 000001
offload, time ((*kernel_function)(devdata))	0. 0.002700
_ort_unmap_tdvar, time (unmap_var)	0. 000005
_ort_unmap_tdvar, time (unmap_var)	0. 000005
_ort_unmap_tdvar, time (unmap_var)	0. 001072

Από τα παραπάνω αποτελέσματα προκύπτει ότι η διαδικασία map (αντιστοίχισης και μεταφοράς δεδομένων) προς και από την τοπική συσκευή είναι σημαντικά ταχύτερη συγκριτικά με την απομακρυσμένη. Παρότι η απομακρυσμένη συσκευή ενδέχεται να είναι θεωρητικά πιο αποδοτική σε υπολογισμούς, στη συγκεκριμένη περίπτωση η διαφορά στον χρόνο εκτέλεσης του πυρήνα είναι αμελητέα, ενώ οι καθυστερήσεις λόγω μεταφοράς δεδομένων (map/unmap) καθιστούν την τοπική εκτέλεση πιο αποδοτική συνολικά.

Επιπλέον, στο παρόν παράδειγμα, έχουν αφαιρεθεί για απλότητα επιπλέον κλήσεις που σχετίζονται με ενδιάμεσες ενέργειες όπως δέσμευση μνήμης και αντιγραφή δεδομένων, οι οποίες επιβαρύνουν κυρίως την απομακρυσμένη συσκευή.

Σε αντίθεση, σε ένα διαφορετικό σενάριο με πιο σύνθετους υπολογισμούς, όπου ο χρόνος εκτέλεσης του πυρήνα θα ήταν το κυρίαρχο μέρος της συνολικής διάρκειας, η επιλογή απομακρυσμένης συσκευής θα ήταν προτιμότερη, καθώς θα μπορούσε να προσφέρει υψηλότερη απόδοση στην εκτέλεση του υπολογιστικού φορτίου.

Η παρούσα υλοποίηση εισάγει έναν ελαφρύ αλλά ταυτόχρονα επεκτάσιμο μηχανισμό καταγραφής χρόνων εντός του OMPi runtime. Ο μηχανισμός αυτός βασίζεται στη χρήση μακροεντολών (macros), γεγονός που καθιστά την ενσωμάτωσή του τόσο διαφανή όσο και ευέλικτη, επιτρέποντας την ενεργοποίησή του μόνο όπου απαιτείται, χωρίς να επηρεάζεται ο βασικός κώδικας ή η λειτουργικότητα του runtime. Επιπλέον, η χρήση ενιαίας δομής δεδομένων για την αποθήκευση των μετρήσεων εξασφαλίζει τη συνεκτικότητα και διευκολύνει τη μελλοντική ανάλυση και αξιοποίησή τους.

Ένα από τα σημαντικότερα πλεονεκτήματα της υλοποίησης αυτής είναι η γενικότητα και η ευελιξία που προσφέρει: ο μηχανισμός μπορεί να χρησιμοποιηθεί για τη

χρονομέτρηση οποιασδήποτε συνάρτησης του runtime — ανεξαρτήτως του αν σχετίζεται άμεσα με διαδικασίες offloading ή άλλες εσωτερικές διεργασίες, όπως π.χ. διαχείριση νημάτων, συγχρονισμοί ή διαμοιρασμός δεδομένων. Αυτό καθιστά το εργαλείο εξαιρετικά χρήσιμο όχι μόνο για σκοπούς αποσφαλμάτωσης και ανάλυσης απόδοσης, αλλά και για πιο σύνθετες διαδικασίες όπως η παρακολούθηση φορτίου κατά την εκτέλεση, η αυτόματη ρύθμιση παραμέτρων ή ακόμα και η δυναμική προσαρμογή (adaptive scheduling) σε πραγματικό χρόνο.

Η αποτύπωση των αποτελεσμάτων είτε μέσω του τερματικού είτε με καταγραφή σε αρχείο, προσφέρει επιπλέον ευελιξία στους χρήστες ή τους ερευνητές που επιθυμούν να εκτελέσουν συγκριτικές μελέτες, πειραματική αξιολόγηση εναλλακτικών υλοποιήσεων, ή να δημιουργήσουν γραφικές απεικονίσεις της απόδοσης. Το γεγονός ότι η υλοποίηση παραμένει ελαφριά σε επίπεδο πόρων και δεν προσθέτει σημαντικό overhead κατά την εκτέλεση, ενισχύει την πρακτική αξία της.

Τέλος, η σχεδίαση του μηχανισμού αφήνει ανοικτό το ενδεχόμενο για περαιτέρω εξέλιξη. Για παράδειγμα, θα μπορούσε να επεκταθεί ώστε να υποστηρίζει αποστολή των μετρήσεων σε εξωτερικό profiler ή ακόμα και ενσωμάτωση με εργαλεία visualization. Επίσης, μπορεί να αποτελέσει τη βάση για αυτοματοποιημένες στρατηγικές βελτιστοποίησης (π.χ. runtime tuning ή feedback-driven compilation), ειδικά σε περιβάλλοντα με ετερογενείς συσκευές (π.χ. τοπικές και απομακρυσμένες μονάδες επιτάχυνσης).

Συνολικά, ο εν λόγω μηχανισμός profiling ενισχύει ουσιαστικά τη δυνατότητα παρακολούθησης και ανάλυσης της συμπεριφοράς του OMPi runtime και συνιστά ένα σημαντικό εργαλείο για την εξέλιξη της πλατφόρμας τόσο σε ερευνητικό όσο και σε πρακτικό επίπεδο.

4.3 Scheduling στον μεταφραστή OMPi

Η έννοια του scheduling, όπως προσεγγίζεται στην παρούσα εργασία, αναφέρεται στην κατανομή των προς εκτέλεση υπολογιστικών εργασιών (kernels) σε απομακρυσμένες συσκευές (devices) εντός ενός κατανεμημένου υπολογιστικού συστήματος. Η κατανομή αυτή σχετίζεται άμεσα με τη λειτουργικότητα του remote offloading, όπως υποστηρίζεται στο runtime του OMPi, και αποτελεί βασικό μηχανισμό για την αποδοτική αξιοποίηση των διαθέσιμων πόρων.

Ο μεταφραστής OMPi παρέχει ένα σύνολο από βασικές κλήσεις, μέσω των οποίων διευκολύνεται η αντιστοίχιση μεταξύ συσκευών, κόμβων και modules, χωρίς να απαιτείται από τον προγραμματιστή να χειρίζεται άμεσα τα device IDs. Τα device ids

αρχικοποιούνται κατά την εγκατάσταση του μεταφραστή στο εκάστοτε σύστημα. Πιο συγκεκριμένα ο τοπικός κόμβος (host), έχει πάντα id μηδέν, ενώ οι υπόλοιποι κόμβοι λαμβάνουν τιμές από ένα έως και τον αριθμό των διαθέσιμων συσκευών συνολικά με τη σειρά που έχουν οριστεί στο αρχείο .ompi_remote_devices. Το αρχείο αυτό χρησιμοποιείται κατά την εγκατάσταση του OMPi για να κάνει τη κατάλληλη ανίχνευση για κόμβους (ips) και διαθέσιμα devices για κάθε κόμβο (cpus, cuda, opencl). Ένα παράδειγμα για ανίχνευση ενός κόμβου με μία συσκευή CPU είναι το εξής:

```
197.100.13.1{
    cpu: 1
}
```

Ενδεικτικά παραδείγματα κλήσεων που παρέχει ο OMPi είναι:

- int ompx_device_get_node(int global_ID, int * local_ID): Δέχεται ως παραμέτρους το καθολικό id της συσκευής (αναγνωριστικό ανάμεσα σε όλα τα διαθέσιμα devices) και αυτό που έχει τοπικά στον κόμβο στον οποίο ανήκει. Επιστρέφει το index του κόμβου αυτού ή -1 αν δεν είναι έγκυρο το global id.
- int ompx_device_get_node(int global_ID, int * local_ID): Δέχεται ως παραμέτρους το καθολικό ID της συσκευής (αναγνωριστικό ανάμεσα σε όλα τα διαθέσιμα devices) και έναν δείκτη στον οποίο επιστρέφεται το τοπικό ID της συσκευής στον κόμβο της. Επιστρέφει το index του κόμβου στον οποίο ανήκει η συσκευή ή -1 αν το global_ID είναι μη έγκυρο.
- char * ompx_device_get_module(int global_ID, int * module_relative): Δέχεται το καθολικό ID μιας συσκευής και έναν προαιρετικό δείκτη στον οποίο επιστρέφεται ο δείκτης της συσκευής εντός του module (module-relative index). Επιστρέφει το όνομα του module στο οποίο ανήκει η συσκευή ή "invalid device" αν το ID δεν είναι έγκυρο.
- int ompx_get_device_of_node(int node_index, int local_ID): Δέχεται το index ενός κόμβου (node) και το τοπικό ID μιας συσκευής στον κόμβο αυτό. Επιστρέφει το καθολικό ID (global device ID) της συσκευής ή -1 σε περίπτωση μη έγκυρης τιμής.
- int ompx_get_device_of_module(char * module_name, int module_relative): Δέχεται το όνομα ενός module και το σχετικό index της συσκευής εντός του module. Επιστρέφει το καθολικό ID της συσκευής ή -1 αν τα ορίσματα δεν είναι έγκυρα ή δεν υπάρχει τέτοια συσκευή.

- int ompx_get_node_num_devices(int node_index): Δέχεται το index ενός κόμβου. Επιστρέφει τον αριθμό των συσκευών στον συγκεκριμένο κόμβο ή -1 αν το index είναι μη έγκυρο.
- int ompx_get_module_num_devices(char * module_name): Δέχεται το όνομα ενός module. Επιστρέφει τον συνολικό αριθμό συσκευών που ανήκουν στο module αυτό ή -1 αν δεν βρεθεί ή δεν έχει καθόλου συσκευές.

Οι κλήσεις αυτές παρέχουν έναν βασικό μηχανισμό αφαίρεσης για την περιγραφή και επιλογή συσκευών στο runtime, διευκολύνοντας σημαντικά την ανάπτυξη στρατηγικών κατανομής φόρτου. Με βάση αυτή την υποδομή, υλοποιήθηκαν δύο διακριτοί αλγόριθμοι για αυτοματοποιημένο device scheduling, που ενσωματώνονται στον μεταφραστή και ενεργοποιούνται δυναμικά κατά την εκτέλεση κάθε target περιοχής.

Αυτό το είδος μηχανισμού είναι ιδιαίτερα χρήσιμο σε σενάρια όπου η εφαρμογή περιλαμβάνει πολλαπλά target kernels με ξεχωριστές απαιτήσεις ή όταν γίνεται μαζική επεξεργασία δεδομένων, όπως για παράδειγμα στην παράλληλη επεξεργασία πλήθους εικόνων. Σε τέτοιες περιπτώσεις, η στατική επιλογή συσκευής από τον χρήστη είναι όχι μόνο δυσκίνητη αλλά και μη αποδοτική. Οι δύο στρατηγικές που προτείνονται επιλύουν αυτό το πρόβλημα, απαλλάσσοντας τον προγραμματιστή από τη διαχείριση των συσκευών και αξιοποιώντας αποδοτικά όλους τους διαθέσιμους πόρους του συστήματος.

4.3.1 Βασικά χαρακτηριστικά υλοποίησης

Ο μηχανισμός scheduling που αναπτύχθηκε στο πλαίσιο του OMPi σχεδιάστηκε έτσι ώστε να λειτουργεί με διαφανή τρόπο για τον τελικό χρήστη. Αυτό σημαίνει ότι δεν απαιτείται καμία τροποποίηση στον πηγαίο κώδικα των εφαρμογών. Ο scheduler ενσωματώνεται στο runtime του OMPi και αξιοποιείται αυτόματα, ενώ μπορεί να παραμετροποιηθεί τόσο μέσω μεταβλητών περιβάλλοντος όσο και μέσω ρητών επιλογών σε οδηγίες `#pragma omp target`.

Η βασική δυνατότητα παραμετροποίησης προσφέρεται μέσω της μεταβλητής περιβάλλοντος:

```
export OMPI_DEVICE_SCHEDULING="cuda"
```

Με τον τρόπο αυτό επιτρέπεται η στοχευμένη χρήση συσκευών cuda, χωρίς ανάγκη επαναμεταγλώττισης ή αλλαγής στον κώδικα της εφαρμογής. Ο OMPi υποστηρίζει τρεις τύπους συσκευών, cuda, opencl και vulkan.

Επιπλέον, υποστηρίζεται ρητή επιλογή στρατηγικής κατανομής εργασιών με χρήση ειδικών αρνητικών τιμών στο πεδίο device() πεδίο της οδηγίας `#pragma omp target,` οι οποίες αντιστοιχούν σε συγκεκριμένες πολιτικές:

```
#pragma omp target device(omp_low_workload_scheduling)
```

Η αντιστοίχιση των τιμών αυτών υλοποιείται μέσω ειδικού enum που χρησιμοποιείται εσωτερικά στο runtime:

- omp_low_workload_scheduling = -70,
- omp_round_robin_scheduling = -71

Πιο συγκεκριμένα η φράση device λαμβάνει θετικές τιμές οι οποίες αντιστοιχούν στα global device ids, όταν όμως περνάμε τις παραπάνω αρνητικές τιμές ενεργοποιείται η αντίστοιχη στρατηγική scheduling. Αυτό συμβαίνει διότι στο πρότυπο OpenMP οι αρνητικές τιμές είναι <<μη νόμιμες>> οι αρνητικές τιμές, συνεπώς δεν θα χρησιμοποιηθούν με στόχο τη χρήση συγκεκριμένου device. Στη παραπάνω περίπτωση η βιβλιοθήκη runtime όταν εντοπίσει αρνητική τιμή αντί να τερματίσει το πρόγραμμα καλεί τη συνάρτηση του scheduler η οποία επιλέγει με βάση τα παραπάνω IDs την αντίστοιχη στρατηγική ή επιστρέφει το id μηδέν του host σε περίπτωση που καμία στρατηγική δεν αντιστοιχεί στον ακέραιο αυτό .

Στρατηγική 1: Round-Robin Scheduling

Πρόκειται για μία απλή και δίκαιη πολιτική κατανομής εργασιών, κατά την οποία οι εργασίες διανέμονται κυκλικά (σε κυκλικό μοτίβο) στις διαθέσιμες συσκευές ενός συγκεκριμένου τύπου όπως GPUs. Η υλοποίηση βασίζεται σε έναν παγκόσμιο μετρητή (counter), ο οποίος αυξάνεται κάθε φορά που εκτελείται ένα target offload, ανεξάρτητα από το είδος ή τη διάρκεια του kernel. Πιο συγκεκριμένα, το index του device αρχικοποιείται στη τρέχουσα τιμή του counter και μετά αυξάνεται. Στη συνέχεια περνάμε σε συνάρτηση ως παράμετρο το όνομα του τύπου (για παράδειγμα CPU) των devices που θέλουμε να εφαρμόσουμε round robin scheduling και το υπόλοιπο της διαίρεσης index_of_next device προς τον συνολικό αριθμό των συσκευών για τον συγκεκριμένο τύπο. Η συνάρτηση αυτή με τις παραπάνω παραμέτρους μας επιστρέφει το global device id του επόμενου device στο οποίο θα γίνει το offload.

Ο αλγόριθμος έχει την εξής μορφή:

```
LOCK
    index_of_next_device = counter
    counter = counter + 1
UNLOCK
```

```
global_device_id = get_device(module_name, index_of_next_device mod
total_devices_in_module)
return global_device_id
```

Η συγκεκριμένη προσέγγιση είναι κατάλληλη για ομογενή περιβάλλοντα, όπου όλες οι συσκευές έχουν παρόμοια υπολογιστική ισχύ και οι εργασίες (kernels) είναι σχετικά ισοδύναμες ως προς τον φόρτο και τη διάρκεια εκτέλεσης. Δεν λαμβάνει υπόψη τον πραγματικό φόρτο εργασίας κάθε συσκευής τη δεδομένη χρονική στιγμή.

Ένα πλεονέκτημα της προσέγγισης είναι ότι απλοποιεί τον τρόπο κατανομής όταν υπάρχουν διαφορετικές κλήσεις που δημιουργούν kernels, οι οποίες ενδέχεται να μην μοιράζονται εύκολα την πληροφορία μεταξύ τους. Επίσης, σε περιβάλλοντα με ετερογενή πλήθος συσκευών (π.χ. clusters με διαφορετικούς τύπους GPUs ή επιταχυντών), η στρατηγική αυτή διευκολύνει τον διαμοιρασμό καθώς ο προγραμματιστής μπορεί να επιλέξει μία ομάδα συσκευών ενός τύπου (module), χωρίς να χρειάζεται να καθορίσει με ακρίβεια τα device ids κάθε φορά. Έτσι επιτυγχάνεται ευκολία χρήσης και μια μορφή στατικής ισορροπίας φορτίου, χωρίς την ανάγκη παρακολούθησης της κατάστασης των συσκευών.

Στρατηγική 2: Low Workload Scheduling

Η στρατηγική αυτή επιδιώκει μια πιο δυναμική εξισορρόπηση του φορτίου μεταξύ των συσκευών, επιλέγοντας κάθε φορά τη συσκευή με το μικρότερο αριθμό ενεργών εργασιών. Η υπόθεση που γίνεται είναι ότι οι εργασίες (kernels) έχουν παρόμοιο φόρτο και διάρκεια, οπότε το τρέχον workload κάθε συσκευής μπορεί να προσεγγιστεί μετρώντας απλώς πόσες εργασίες "τρέχουν" σε κάθε μια.

Η επιλογή της κατάλληλης συσκευής γίνεται με βάση την αναζήτηση στον πίνακα κατάστασης status[], ο οποίος διατηρεί τον αριθμό των ενεργών εργασιών ανά συσκευή. Αν δεν πραγματοποιηθεί ορθή αρχικοποίηση του πίνακα επιστρέφει το global id του default device (host) το οποίο έχει πάντα id ίσο με το μηδέν.

Ο αλγόριθμος υλοποιείται ως εξής:

```
IF status_table IS NULL
    return device_0
LOCK
    choice = 0
    FOR i FROM 1 TO total_devices - 1
        IF module_matches(i, module_name)
            IF status[i] == 0
```

```
                choice = i
                BREAK
            ELSE IF status[i] < status[choice]
                choice = i
    IF choice ≠ 0
        status[choice] = status[choice] + 1
UNLOCK
return choice
```

Ο στόχος της στρατηγικής είναι η βελτιστοποίηση της κατανομής των εργασιών με βάση το τρέχον workload κάθε συσκευής, επιδιώκοντας έτσι καλύτερη αξιοποίηση των διαθέσιμων πόρων. Η μέθοδος βασίζεται στην παραδοχή ότι τα kernels έχουν παρόμοιο φόρτο εργασίας, καθώς δεν υπάρχει πρόσβαση σε λεπτομερή πληροφόρηση όπως ακριβή χρόνο εκτέλεσης ή υπολογιστική απαίτηση κάθε εργασίας.

Ωστόσο, το πλεονέκτημα αυτής της προσέγγισης είναι ότι, ακόμη και χωρίς λεπτομερή παρακολούθηση, μπορεί να προσφέρει σημαντικά καλύτερη εξισορρόπηση σε σχέση με στατικές πολιτικές.

4.3.2 Αρχιτεκτονική και επεκτασιμότητα

Η αρχιτεκτονική της υλοποίησης του μηχανισμού scheduling στον OMPi έχει σχεδιαστεί με γνώμονα τη modular δομή, προκειμένου να προσφέρει ευελιξία, συντηρησιμότητα και επεκτασιμότητα. Ο διαχωρισμός της λογικής κατανομής φορτίου από τον υπόλοιπο κώδικα του runtime επιτρέπει την ανεξάρτητη ανάπτυξη και εξέλιξη των επιμέρους στρατηγικών χωρίς να απαιτούνται παρεμβάσεις στον πυρήνα του συστήματος. Η κάθε στρατηγική μπορεί να προστεθεί με απλό τρόπο μέσω εγγραφής στον μηχανισμό επιλογής, διατηρώντας τον υπόλοιπο κώδικα του runtime αμετάβλητο.

Η ενσωμάτωση της επιλογής συσκευής στο παραγόμενο C code γίνεται με αυτοματοποιημένο τρόπο κατά το στάδιο κατασκευής του AST (Abstract Syntax Tree), χωρίς να απαιτείται παρέμβαση από τον χρήστη.

Η λογική που ενσωματώνεται έχει τη μορφή:

```
if (__ompi_devID < omp_invalid_device)
    __ompi_devID = _get_device_from_scheduling(__ompi_devID);
```

Με αυτόν τον τρόπο, η κάθε περιοχή target γίνεται δυναμική ως προς την επιλογή του device, ακολουθώντας την κατάλληλη στρατηγική που έχει οριστεί μέσω παραμέτρου περιβάλλοντος και μέσω του pragma. Η ενσωμάτωση αυτή είναι πλήρως διαφανής για

τον τελικό χρήστη και επιτρέπει την αξιοποίηση των πλεονεκτημάτων του scheduling χωρίς να απαιτούνται αλλαγές στον πηγαίο κώδικα.

Σημαντικό στοιχείο της υλοποίησης αποτελεί και η ενημέρωση της κατάστασης της συσκευής μετά την ολοκλήρωση κάθε offload. Η δυνατότητα αυτή επιτρέπει στο runtime να πληροφορεί σε πραγματικό χρόνο τη στρατηγική scheduling για την ολοκλήρωση της εκτέλεσης ενός kernel σε μια απομακρυσμένη συσκευή. Με αυτόν τον τρόπο είναι δυνατή η παρακολούθηση του τρέχοντος φόρτου της κάθε συσκευής, για παράδειγμα μέσω της μείωσης του αριθμού ενεργών εργασιών, κάτι που είναι απαραίτητο για την υποστήριξη στρατηγικών τύπου low workload. Επιπλέον, η ύπαρξη αυτής της πληροφορίας ανοίγει τον δρόμο για πιο εξελιγμένες, προσαρμοστικές τεχνικές scheduling στο μέλλον, οι οποίες θα μπορούν να βασίζονται σε metrics όπως το latency, low-workload με χρήση διαφορετικού τύπου συσκευών ή ακόμα και ιστορικά δεδομένα απόδοσης.Η στρατηγική επιλογής device υλοποιείται εσωτερικά μέσω ενός ενιαίου interface, το οποίο βασίζεται σε enumeration (enum) για την αναγνώριση της επιθυμητής πολιτικής κατανομής. Σε αυτό το enum έχουν προστεθεί ειδικές τιμές που αντιστοιχούν σε διαφορετικές στρατηγικές (όπως π.χ. omp_round_robin_scheduling και omp_low_workload_scheduling), και η επιλογή της στρατηγικής γίνεται με βάση αυτή την τιμή. Η χρήση του enum διασφαλίζει τη συνοχή στην υλοποίηση και διευκολύνει την προσθήκη νέων στρατηγικών στο μέλλον, καθώς το μόνο που απαιτείται είναι η καταχώριση μιας νέας τιμής στο enum και η υλοποίηση της αντίστοιχης συνάρτησης επιλογής.

Η συνολική προσέγγιση προσφέρει πολλαπλά οφέλη. Πρωτίστως, ο μηχανισμός είναι πλήρως αυτοματοποιημένος και δεν επιβάλλει τροποποιήσεις στον κώδικα των εφαρμογών. Ο χρήστης έχει τη δυνατότητα να ρυθμίσει τη συμπεριφορά του μηχανισμού μέσω παραμέτρου περιβάλλοντος και με ρητή δήλωση στο #pragma omp target, γεγονός που προσδίδει σημαντική ευελιξία. Επιπλέον, η υλοποίηση είναι συμβατή με περιβάλλοντα υψηλής κλιμάκωσης, όπως ετερογενή clusters που περιλαμβάνουν πολλαπλές συσκευές, και σχεδιάστηκε ώστε να υποστηρίζει μελλοντικές επεκτάσεις που ενσωματώνουν πληροφορίες από profiling, metrics εκτέλεσης ή ακόμη και τεχνικές μηχανικής μάθησης.

Κεφάλαιο 5. Πειραματικά αποτελέσματα

5.1 Εισαγωγή

Σε αυτό το κεφάλαιο παρουσιάζονται τα αποτελέσματα της εκτέλεσης πειραμάτων που πραγματοποιήθηκαν με τη χρήση της τροποποιημένης έκδοσης του μεταγλωττιστή OMPi, ο οποίος ενσωματώνει το μηχανισμό απομακρυσμένης εκτέλεσης και τις νέες στρατηγικές κατανομής φόρτου που παρουσιάστηκαν στο Κεφάλαιο 4. Στόχος των πειραμάτων είναι η αξιολόγηση της αποτελεσματικότητας των δύο στρατηγικών scheduling (round robin και low workload).

Όλα τα πειράματα εκτελέστηκαν σε cluster τύπου Dell PowerEdge R430 που βρίσκεται στο εργαστήριο του Τμήματος Πληροφορικής. Το cluster αποτελείται από 12 κόμβους, καθένας εκ των οποίων διαθέτει 8 CPU cores (Intel Xeon E5-2620 v4) και 16GB μνήμης RAM. Οι κόμβοι είναι διασυνδεδεμένοι μέσω δικτύου 1 Gbps, το οποίο χρησιμοποιείται τόσο για την επικοινωνία μεταξύ των κόμβων όσο και για τη σύνδεσή τους με το διαδίκτυο. Το λειτουργικό σύστημα στους κόμβους είναι Linux με πυρήνα 4.15.0-213-generic, ενώ η έκδοση της βιβλιοθήκης MPI που χρησιμοποιήθηκε είναι η MPICH 4.0.2.

5.2 Gaussian blur

Η εφαρμογή Gaussian Blur πραγματοποιεί θόλωση (blur) εικόνων με βάση μια δοθείσα ακτίνα, η οποία παρέχεται ως όρισμα μέσω της γραμμής εντολών κατά την εκκίνηση του προγράμματος. Επιπλέον, ως δεύτερο όρισμα εισόδου δίνεται το μονοπάτι (path) προς έναν φάκελο που περιέχει εικόνες, οι οποίες ενδέχεται να διαφέρουν μεταξύ τους ως προς το περιεχόμενο και τις διαστάσεις.

Μετά τη λήψη των παραπάνω ορισμάτων, το πρόγραμμα εκτελεί έναν βρόχο for παράλληλα, χρησιμοποιώντας το directive `#pragma omp parallel for`. Εντός αυτού του παράλληλου βρόχου καλείται μία συνάρτηση, στην οποία υλοποιείται ο αλγόριθμος θόλωσης. Στη συνάρτηση αυτή, χρησιμοποιείται η οδηγία `#pragma omp target parallel for` για τη δημιουργία targets που αποστέλλονται σε απομακρυσμένους υπολογιστικούς κόμβους. Συνεπώς για κάθε εικόνα εκτελείται ένας διαφορετικός kernel. Επίσης, επειδή στέλνουμε πολλές εικόνες σε κόμβους του cluster, σε συνδυασμό με τη χρήση της φράσης `num_threads(8)` στην οδηγία target, χρησιμοποιούνται 8 υπολογιστικοί πυρήνες (CPU-cores) ανά κόμβο.

Η επιλογή του κατάλληλου device για την αποστολή της εργασίας πραγματοποιείται με έναν από τους παρακάτω τρεις τρόπους:

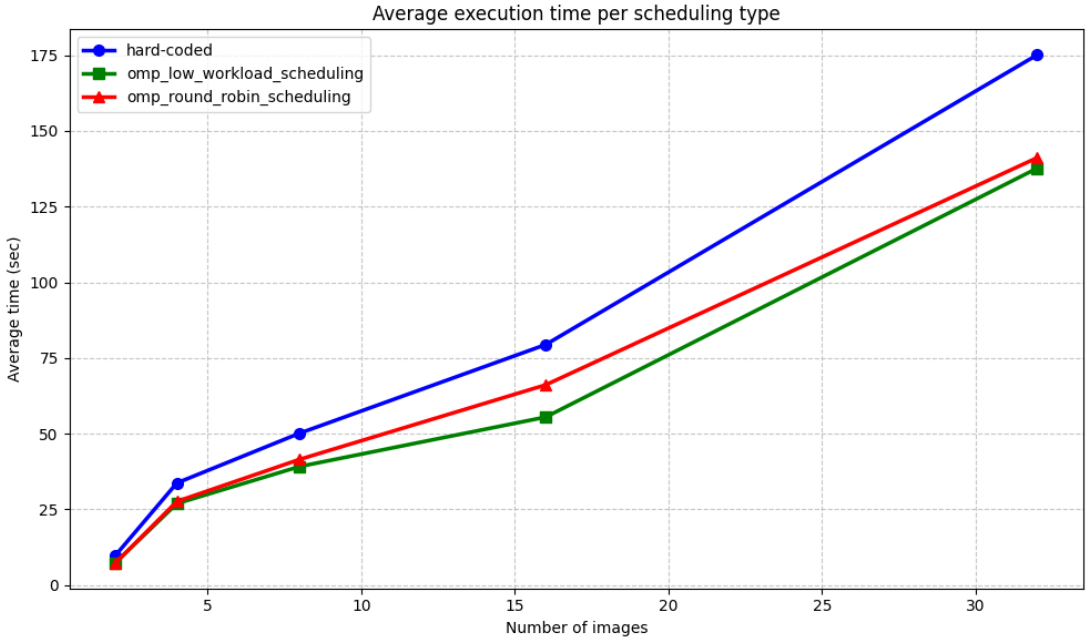
- Hard-coded scheduling: Η κατανομή γίνεται στατικά, με βάση το thread_number. Το device id υπολογίζεται ως εξής: array_of_deviceIds[thread_num % 11]. Συνεπώς στην οδηγία `#pragma omp parallel for` χρησιμοποιούμε τη φράση `num_threads(num_threads(N))` όπου N το πλήθος των συσκευών. Συνεπώς το κάθε νήμα χρησιμοποιεί συγκεκριμένη συσκευή.
- omp_low_workload_scheduling: Δυναμική στρατηγική κατανομής εργασιών που έχει υλοποιηθεί στον OMPi, όπως περιγράφεται στο Κεφάλαιο 4.
- omp_round_robin_scheduling: Εναλλακτική δυναμική στρατηγική, επίσης υλοποιημένη στον OMPi, που κατανέμει τις εργασίες με κυκλικό τρόπο, όπως περιγράφεται στο Κεφάλαιο 4.

5.2.1 Μεταβλητός αριθμός εικόνων

Κάθε μία από τις παραπάνω στρατηγικές δοκιμάστηκε σε πείραμα που επαναλήφθηκε 5 φορές για διαφορετικούς αριθμούς εικόνων (2, 4, 8, 16 και 32). Οι εικόνες που χρησιμοποιήθηκαν είχαν διαστάσεις 1000×1000 pixels και 1500×1500 pixels και σε κάθε εκτέλεση ήταν ίσες σε πλήθος.

Στη Γραφική Παράσταση 5.1 παρουσιάζονται τα αποτελέσματα των πειραμάτων, όπου αποτυπώνονται οι μέσοι χρόνοι εκτέλεσης. Παρατηρείται ότι για μικρό αριθμό εικόνων (2 και 4), οι χρόνοι εκτέλεσης των τριών στρατηγικών δεν εμφανίζουν σημαντικές διαφορές. Ωστόσο, από τις 8 εικόνες και πάνω, οι στρατηγικές κατανομής εργασιών που υλοποιήθηκαν στον OMPi αρχίζουν να υπερτερούν σε απόδοση. Συγκεκριμένα το low workload scheduling εμφανίζει τα καλύτερα αποτελέσματα. Το

αποτέλεσμα αυτό οφείλεται στον καλύτερο διαμοιρασμό φόρτου που επιτυγχάνεται: κόμβοι που ολοκληρώνουν την εκτέλεση των εργασιών τους νωρίτερα μπορούν να συνεχίσουν με άλλες διαθέσιμες, οπότε οι πιο γρήγοροι κόμβοι καταλήγουν να αναλαμβάνουν περισσότερες εργασίες. Η διαφορά αυτή γίνεται ολοένα και πιο εμφανής καθώς αυξάνεται ο αριθμός των εικόνων, γεγονός που αποδεικνύεται ξεκάθαρα στον Πίνακα 5.1, όπου συνοψίζονται οι μέσοι χρόνοι εκτέλεσης για κάθε περίπτωση. Τέλος, ο λόγος που η στρατηγική round robin εμφανίζει καλύτερα αποτελέσματα από τη hard-coded είναι ότι στη τελευταία οι εργασίες εκχωρούνται βάσει του thread_id όπως αναφέρθηκε παραπάνω , συνεπώς όσο αυξάνονται οι εργασίες πολλές από αυτές μένουν σε αναμονή επειδή κάποια νήματα δεν έχουν επιστρέψει για να αναθέσουν την επόμενη στον κόμβο που αντιστοιχεί το device με id τη τιμή array_of_deviceIds[thread_num % 11].



Γραφική παράσταση 5.1: αποτελέσματα χρονομέτρησης gaussian blur για αύξηση αριθμού εικόνων.

	2 εικόνες	**4 εικόνες**	**8 εικόνες**	**16 εικόνες**	**32 εικόνες**
Hard-coded	9.836 sec	33.745 sec	50.196 sec	79.414 sec	175.122 sec
Low workload	7.421 sec	26.9528 sec	39.2009 sec	55.500 sec	137.677 sec
Round Robin	7.326 sec	27.645 sec	41.553 sec	66.121 sec	141.142 sec

5.2.2 Μεταβλητός αριθμός nodes

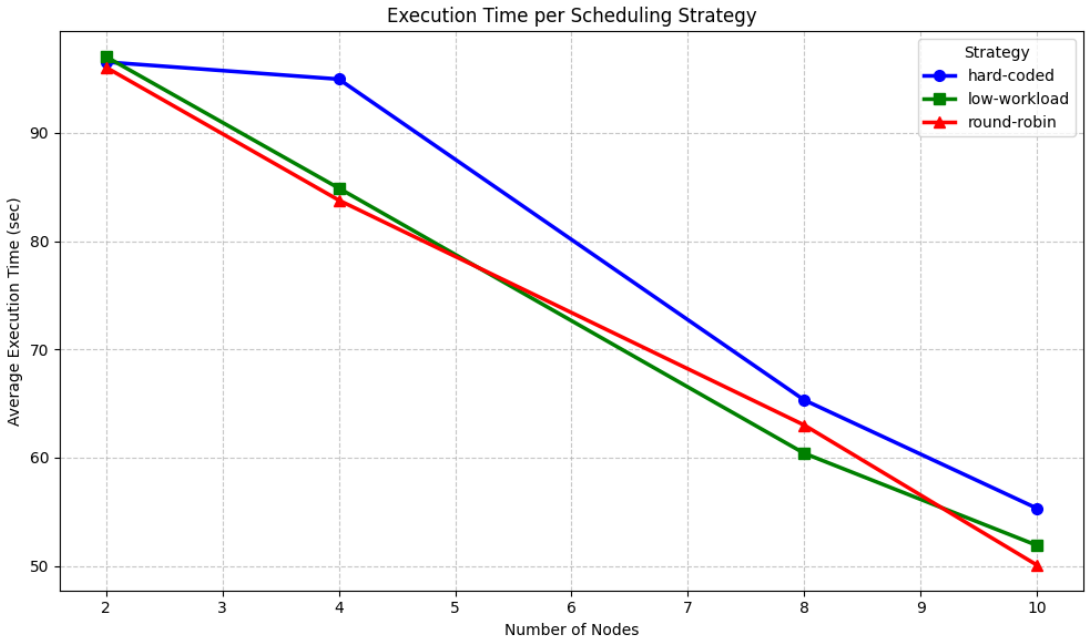
Στο δεύτερο πείραμα χρησιμοποιήθηκαν 128 εικόνες διαστάσεων 500x500 pixels με ακτίνα από 1 έως 5. Ο αριθμός των κόμβων μεταβλήθηκε (2, 4, 8 και 10) με σκοπό την αξιολόγηση της αποτελεσματικότητας των διαφορετικών στρατηγικών scheduling στον OMPi ως προς την εκμετάλλευση των διαθέσιμων υπολογιστικών πόρων.

Από τα αποτελέσματα του Πίνακα 5.2 και της αντίστοιχης Γραφικής Παράστασης 5.2, παρατηρείται ότι οι δυναμικές στρατηγικές scheduling (low-workload και round-robin) του OMPi αποδίδουν συνολικά καλύτερα από τη στατική, χειροκίνητα υλοποιημένη (hard-coded) στρατηγική, ιδιαίτερα όσο αυξάνεται ο αριθμός των κόμβων. Η διαφορά στην απόδοση είναι εμφανής κυρίως στις περιπτώσεις με 4 ή περισσότερους κόμβους.

Η υπεροχή των στρατηγικών scheduling οφείλεται στο γεγονός ότι αξιοποιούν καλύτερα τους διαθέσιμους υπολογιστικούς πόρους. Επειδή οι κόμβοι του συστήματος είναι ισοδύναμοι (ίδιοι επεξεργαστικοί πόροι ανά κόμβο), και οι υπολογιστικές εργασίες (kernels) είναι κατά κύριο λόγο ισοδύναμες από πλευράς υπολογιστικού φόρτου, η ισοκατανομή του φόρτου που πετυχαίνουν οι στρατηγικές round-robin και low-workload αποδίδει πολύ καλά. Η ελαφρώς καλύτερη απόδοση της round-robin σε ορισμένες περιπτώσεις οφείλεται στη σταθερότητα που προσφέρει η κυκλική κατανομή, η οποία φαίνεται να ταιριάζει με το συγκεκριμένο φορτίο.

Αντιθέτως, στη hard-coded υλοποίηση, το index του global device id προσδιορίζεται με βάση το id του νήματος, ανεξαρτήτως του πλήθους των κόμβων ή της τοπικής κατανομής των threads στους κόμβους. Αυτό σημαίνει ότι, σε περιπτώσεις όπου ο αριθμός των κόμβων είναι μικρότερος από τον αριθμό των CPU cores ανά κόμβο (π.χ. 4 κόμβοι σε κόμβους με 8 διαθέσιμους πυρήνες), παραμένουν ανεκμετάλλευτοι πόροι, οι οποίοι όμως αξιοποιούνται πλήρως από τις στρατηγικές του OMPi.

Τέλος, κάθε στρατηγική αξιολογήθηκε με 5 επαναλήψεις και ο πίνακας 5.2 περιέχει τους μέσους όρους των χρόνων εκτέλεσης για κάθε συνδυασμό στρατηγικής και αριθμού κόμβων.



Γραφική παράσταση 5.2: χρόνοι εκτέλεσης gaussian blur για μεταβλητό αριθμό κόμβων

	2 nodes	**4 nodes**	**8 nodes**	**10 nodes**
Hard-coded	96.549 sec	94.969 sec	65.330 sec	55.328 sec
Low workload	97.035 sec	84.895 sec	60.408 sec	51.904 sec
Round Robin	96.050 sec	83.785 sec	63.014 sec	50.087 sec

Πίνακας 5.2: Μέση τιμή χρόνων εκτελέσεων gaussian-blur με αύξηση αριθμού κομβων (σε δευτερόλεπτα).

5.3 Sharpening

Η εφαρμογή Sharpening Filter υλοποιεί έναν αλγόριθμο ενίσχυσης ακμών (edge enhancement), με σκοπό την αύξηση της ευκρίνειας της εικόνας μέσω ενίσχυσης των περιοχών με έντονες μεταβολές φωτεινότητας. Ο αλγόριθμος βασίζεται σε έναν πυρήνα σύγκρισης γειτονικών τιμών (kernel convolution) και χρησιμοποιεί μια προκαθορισμένη μάσκα ενίσχυσης για να επισημάνει τα όρια και να βελτιώσει τη λεπτομέρεια της εικόνας.

Κατά την εκκίνηση της εφαρμογής, δίνονται ως ορίσματα η εικόνα εισόδου και η επιθυμητή ακτίνα μέσω ενός script γραμμένου σε bash.

Το πρόγραμμα περιλαμβάνει παράλληλη υλοποίηση του αλγορίθμου με χρήση OpenMP. Η παράλληλη εκτέλεση επιτυγχάνεται μέσω της οδηγίας:

```
#pragma omp parallel for
```

η οποία χρησιμοποιείται για την επαναληπτική εκτέλεση των υπολογισμών πάνω στην ίδια εικόνα, προκειμένου να αξιολογηθούν οι στρατηγικές κατανομής εργασιών (scheduling) που υλοποιήθηκαν στον OMPi.

Εντός του βρόχου for, καλείται η βασική συνάρτηση που υλοποιεί τον αλγόριθμο sharpening, η οποία περιέχει την οδηγία:

```
#pragma omp target parallel for
```

Η οδηγία αυτή επιτρέπει την εκτέλεση του φίλτρου σε απομακρυσμένους υπολογιστικούς κόμβους (devices). Μέσω της επιλογής device, καθορίζεται σε ποια συσκευή θα εκτελεστεί κάθε kernel, βάσει του device id.

Επιλογές κατανομής kernels σε devices:

Για κάθε εικόνα εκτελείται και διαφορετικός kernel ενώ η εκχώρηση τους στα διαθέσιμα devices γίνεται με βάση τρεις διαφορετικές στρατηγικές:

- Hard-coded Scheduling: Στατική στρατηγική, υλοποιημένη σε επίπεδο εφαρμογής. Με βάση το kernel ID (δηλ. τον αριθμό επανάληψης στο εξωτερικό loop) και τον αριθμό των διαθέσιμων συσκευών (που παρέχεται ως παράμετρος), γίνεται ισοδύναμη κατανομή των kernels στις συσκευές.

  Για παράδειγμα:

  1. Με 2 συσκευές και 4 kernels: κάθε συσκευή εκτελεί από 2 kernels.
  2. Με 3 συσκευές και 5 kernels: οι δύο πρώτες συσκευές εκτελούν από 2 kernels και η τρίτη έναν. Πρώτη θεωρείται η συσκευή με το μικρότερο device id.

  Ο υπολογισμός γίνεται βάσει του παρακάτω κώδικα:

```
if (myid < (kernels_per_device + 1) * leftover_kernels)
        return myid / (kernels_per_device + 1);
else
        return (myid - leftover_kernels) / kernels_per_device;
```

- Low Workload Scheduling: Δυναμική στρατηγική που υλοποιείται στο OMPi, κατά την οποία κάθε νέα εργασία εκχωρείται στη συσκευή με τον χαμηλότερο τρέχοντα φόρτο.

- Round Robin Scheduling: Κυκλική κατανομή, όπου τα kernels εκχωρούνται διαδοχικά στα διαθέσιμα devices με ισομερή τρόπο.

5.3.1 Μεταβολή αριθμού εικόνων

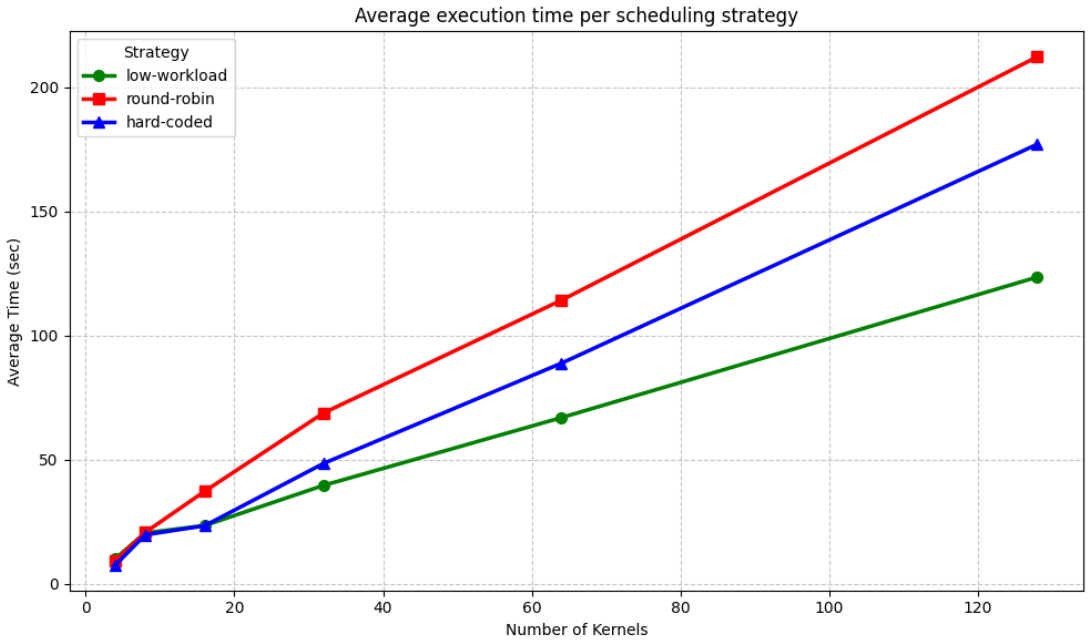
Στο συγκεκριμένο πείραμα μελετάται η επίδραση του αριθμού των εικόνων στην απόδοση του αλγορίθμου sharpening. Εκτελέστηκαν πολλαπλές δοκιμές με διαφορετικό πλήθος kernels: 4, 8, 16, 32, 64 και 128, σε εικόνες διαστάσεων 1024×1024, με σταθερή ακτίνα radius = 8.

Όπως και στα προηγούμενα πειράματα, η στρατηγική low-workload επιτυγχάνει συνολικά την καλύτερη απόδοση, ιδιαίτερα όταν ο αριθμός των εικόνων αυξάνεται σημαντικά. Στο συγκεκριμένο πείραμα αξιολογείται η επίδραση του υπολογιστικού φόρτου στο offloading, χρησιμοποιώντας τον αλγόριθμο sharpening, ο οποίος χαρακτηρίζεται από μικρότερο χρόνο επεξεργασίας ανά εικόνα σε σύγκριση με τον Kuwahara που χρησιμοποιήθηκε στο προηγούμενο πείραμα.

Αυτό έχει ως αποτέλεσμα, για μικρό πλήθος εικόνων, η low-workload να μην είναι τόσο αποδοτική, καθώς ο συνολικός φόρτος είναι χαμηλός και η στρατηγική δεν έχει αρκετό περιθώριο να αξιοποιήσει πλήρως τα χαρακτηριστικά της. Όσο όμως αυξάνεται ο αριθμός των εικόνων, η low-workload παρουσιάζει σταδιακή βελτίωση.

Αντίθετα, οι hard-coded και round-robin στρατηγικές παρουσιάζουν αυξανόμενους χρόνους εκτέλεσης όσο μεγαλώνει ο αριθμός των εικόνων. Αυτό οφείλεται στο γεγονός ότι και οι δύο χρησιμοποιούν ισοδύναμη κατανομή των εικόνων, χωρίς να λαμβάνουν υπόψη τη μεταβαλλόμενη φύση του φόρτου κατά την εκτέλεση. Αν και οι κόμβοι έχουν την ίδια υπολογιστική ισχύ, ο τρόπος με τον οποίο κατανεμόταν η εργασία φαίνεται να επηρεάζει την απόδοση.

Η hard-coded στρατηγική εμφανίζει ελαφρώς καλύτερους χρόνους από τη round-robin. Το πλεονέκτημα αυτό ενδέχεται να οφείλεται σε πιο συνεκτική κατανομή των εικόνων, που μειώνει το overhead της εναλλαγής εργασιών μεταξύ κόμβων, συγκριτικά με την κυκλική κατανομή της round-robin. Τα αναλυτικά αποτελέσματα παρουσιάζονται στον Πίνακα 5.4.



Γραφική παράσταση 5.3 : αποτελέσματα χρονομέτρησης αλγορίθμου sharpening για αύξηση του αριθμού των εικόνων.

	4 kernels	**8 kernels**	**16 kernels**	**32 kernels**	**64 kernels**	**128 kernels**
Hard-coded	7.556 sec	19.729 sec	23.370 sec	48.488 sec	88.876 sec	177.101 sec
Low workload	10.198 sec	20.483 sec	23.449 sec	39.644 sec	66.952 sec	123.573 sec
Round Robin	9.441 sec	20.671 sec	37.263 sec	68.830 sec	114.26 sec	212.357 sec

Πίνακας 5.2: Μέση τιμή χρόνων εκτελέσεων αλγορίθμου sharpening με αύξηση αριθμού εικόνων (σε δευτερόλεπτα).

Κεφάλαιο 6. Επίλογος

6.1 Σύνοψη διπλωματικής εργασίας

Αντικείμενο της παρούσας διπλωματικής εργασίας αποτέλεσε η απομακρυσμένη εκτέλεση κώδικα στους κόμβους ενός υπολογιστικού συστήματος τύπου cluster, με στόχο τη βελτιστοποίηση της αξιοποίησης των υπολογιστικών πόρων και τη διευκόλυνση του χρήστη. Στο πλαίσιο αυτό, αξιοποιήθηκε και επεκτάθηκε η υπάρχουσα υποδομή του μεταφραστή OMPi, ο οποίος υποστηρίζει βασικές δυνατότητες απομακρυσμένου offloading. Η συνεισφορά της εργασίας επικεντρώνεται στην προσθήκη δύο ανεξάρτητων μηχανισμών: profiling και scheduling, οι οποίοι βελτιώνουν την παρακολούθηση της απόδοσης και την κατανομή φόρτου στους διαθέσιμους υπολογιστικούς πόρους.

Ο μηχανισμός profiling παρέχει δυνατότητα μέτρησης και καταγραφής των χρόνων εκτέλεσης επιλεγμένων τμημάτων κώδικα (kernels), σε διαφορετικές συσκευές. Τα αποτελέσματα του profiling μπορούν να αποθηκεύονται ή να εμφανίζονται σε πραγματικό χρόνο μέσω ενός stream (σε αρχείο ή στο τερματικό), δίνοντας στον χρήστη μια σαφή εικόνα της απόδοσης κάθε εκτέλεσης. Η υλοποίηση είναι σχεδιασμένη έτσι ώστε να είναι ελαφριά, επεκτάσιμη και να ενσωματώνεται εύκολα χωρίς τροποποιήσεις στον κώδικα του χρήστη.

Από την άλλη, ο μηχανισμός scheduling αναλαμβάνει τη δυναμική κατανομή των εργασιών στους διαθέσιμους κόμβους του cluster, με βάση απλές αλλά λειτουργικές στρατηγικές. Στην παρούσα υλοποίηση υποστηρίζονται δύο πολιτικές κατανομής:

- Round Robin, η οποία εκχωρεί εργασίες κυκλικά, και
- Low Workload, η οποία επιλέγει τον κόμβο με τον μικρότερο τρέχοντα φόρτο.

Και οι δύο μηχανισμοί έχουν σχεδιαστεί με γνώμονα την ευκολία χρήσης και τη μελλοντική επεκτασιμότητα. Εκτός από την επιλογή στρατιγικής scheduling και τον ορισμό μεταβλητής περιβάλλοντος για την επιλογή του τύπου των devices που επιθυμεί

ο χρήστης να χρησιμοποιηθούν, δεν απαιτούνται αλλαγές στο πρόγραμμα του τελικού χρήστη. Το αποτέλεσμα είναι ένα σύστημα πιο λειτουργικό και προσαρμοστικό, που μπορεί να υποστηρίξει απομακρυσμένη εκτέλεση σε clusters με πολλούς και διαφορετικούς υπολογιστικούς πόρους, χωρίς να επιβαρύνει τον προγραμματιστή.

Συνολικά, η παρούσα εργασία ενισχύει τη λειτουργικότητα του OMPi και καθιστά πιο αποδοτική, προσαρμοστική και εύχρηστη την απομακρυσμένη εκτέλεση υπολογιστικών φορτίων σε clusters, διευρύνοντας έτσι τις δυνατότητές του σε σύγχρονα, απαιτητικά περιβάλλοντα. Επίσης, θέτει τις βάσεις για περαιτέρω επεκτάσεις, τόσο στη σύνδεση των δύο μηχανισμών όσο και στη βελτιστοποίηση της απόδοσης του συστήματος συνολικά.

6.2 Προτάσεις για μελλοντικές εργασίες

Η παρούσα εργασία έθεσε τις βάσεις για έναν ευέλικτο και επεκτάσιμο μηχανισμό απομακρυσμένης εκτέλεσης σε clusters, με δυνατότητες profiling και scheduling. Ωστόσο, υπάρχουν αρκετές δυνατότητες για περαιτέρω βελτιώσεις και επεκτάσεις, οι οποίες θα μπορούσαν να ενισχύσουν τη λειτουργικότητα και την αποδοτικότητα του συστήματος.

Μια άμεση και πρακτική επέκταση αφορά τη σύνδεση των αποτελεσμάτων του μηχανισμού profiling με τον scheduler. Μέχρι τώρα, οι δύο αυτοί μηχανισμοί λειτουργούν ανεξάρτητα. Σε μελλοντικές υλοποιήσεις, τα δεδομένα που συλλέγονται από το profiling (χρόνοι εκτέλεσης, επιβάρυνση συσκευών, αριθμός kernels κ.ά.) θα μπορούσαν να αξιοποιούνται δυναμικά κατά την επιλογή του κατάλληλου κόμβου για την εκτέλεση ενός kernel. Με αυτόν τον τρόπο, θα επιτυγχάνεται πιο έξυπνη και αποδοτική κατανομή του φόρτου εργασίας.

Επιπλέον, υπάρχει περιθώριο για επέκταση των στρατηγικών scheduling, ώστε να λαμβάνονται υπόψη οι ιδιαιτερότητες διαφορετικών τύπων συσκευών (π.χ. CPU, GPU, επιταχυντές). Η διαχείριση ετερογενών συστημάτων αποτελεί πρόκληση, και η υποστήριξη εξειδικευμένων πολιτικών scheduling θα επέτρεπε τη βέλτιστη εκμετάλλευση των πόρων σε clusters με ποικιλία υπολογιστικών μονάδων.

Όσον αφορά το profiling, με τη συσσώρευση αποτελεσμάτων από πολλαπλές εκτελέσεις, μια επόμενη προσθήκη θα μπορούσε να είναι η οπτικοποίηση των δεδομένων μέσω κατάλληλων εργαλείων visualization. Η δημιουργία γραφικών που απεικονίζουν την απόδοση κάθε scheduling πολιτικής, σε συνδυασμό με τον υπολογιστικό φόρτο (π.χ. αριθμό kernels, χρόνο εκτέλεσης, κατανομή στους κόμβους),

θα βοηθούσε τόσο στην αξιολόγηση της συμπεριφοράς του συστήματος όσο και στην ανάλυση από τον χρήστη.

Τέλος, μια πιο φιλόδοξη και ιδανική κατεύθυνση για το μέλλον είναι η ενσωμάτωση τεχνικών μηχανικής μάθησης στον scheduler. Μέσω της χρήσης νευρωνικών δικτύων, ο scheduler θα μπορούσε να προβλέπει την καλύτερη κατανομή των kernels, βασιζόμενος σε ιστορικά δεδομένα από τον μηχανισμό profiling. Ένα τέτοιο σύστημα θα μπορούσε να προσαρμόζεται δυναμικά στις εκάστοτε συνθήκες του cluster, οδηγώντας σε ένα έξυπνο scheduler.

Η υλοποίηση αυτών των επεκτάσεων θα ενίσχυε σημαντικά την προσαρμοστικότητα, αποδοτικότητα και ευχρηστία του συστήματος, καθιστώντας το ένα ολοκληρωμένο εργαλείο απομακρυσμένης εκτέλεσης σε ετερογενή clusters.

Βιβλιογραφία

[1] P.S. Pacheco. An Introduction to Parallel Programming. Morgan Kaufmann Publishers, 2011.

[2] OpenMP ARB. *OpenMP Application Programming Interface v5.2*, 2021.

[3] Parallel Processing Group University of Ioannina

Διαθέσιμο στο: https://paragroup.cse.uoi.gr

[4] ECP Exascale Computing Project. *Remote OpenMP Offloading Strategy Increases Productive Programmability*.

Διαθέσιμο στο: https://www.exascaleproject.org/publication/remote-openmp-offloading-strategy-increases-productive-programmability-reduces-complexity-for-real-world-applications/

[5] D. Shan, J. Cownie, et al. *Towards Efficient Remote OpenMP Offloading*. IWOMP 2022.

[6] LLVM Project. *Clang Offloading Design & Internals*.

Διαθέσιμο στο: https://clang.llvm.org/docs/OffloadingDesign.html

[7] Intel. An Introduction to Accelarator and Parallel Programming.

Διαθέσιμο στο: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-accelerator-and-parallel-programming.html

[8] IBM Corporation. What is Accelerated Computing

Διαθέσιμο στο : https://www.ibm.com/think/topics/accelerated-computing

[9] A. Papadogiannakis, S.N. Agathos, V.V. Dimakopoulos, "OpenMP 4.0 Device Support in the OMPi Compiler", in Proc. IWOMP 2015, 11th International Workshop on OpenMP, Aachen, Germany, Oct. 2015, pp. 202—216.

[10] I.K. Kasmeridis, S. Mantelos, V.V. Dimakopoulos, "Portable OpenMP Offload-
ing Across Clusters", Proceedings ISPDC 2025, 24th International Symposium on

Parallel and Distributed Computing, Rennes, France, July 2025.

[11] I. Kleftakis, V.V. Dimakopoulos, “Experiences with task-based programming using cluster nodes as OpenMP devices”, in *Proc. HPCS 2020/2021, 18th Int'l Conference on High Performance Computing and Simulation*, Barcelona, Spain, Mar. 2021.