Τοπολογία και συγχρονισμός στο OpenMP για συστήματα NUMA πάρα πολλών πυρήνων

Γεώργιος Ζ. Ζάχος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

Πανεπιστήμιο Ιωαννίνων

Σεπτέμβριος 2021

ΑΦΙΕΡΩΣΗ

Στην Πανωραία

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΡΟΓΡΑΜΜΑΤΩΝ

ΠΕΡΙΛΗΨΗ

Γεώργιος Ζ. Ζάχος, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος 2021.
Τοπολογία και συγχρονισμός στο OpenMP για συστήματα NUMA πάρα πολλών πυρήνων.
Επιβλέπων: Βασίλειος Β. Δημακόπουλος, Αναπληρωτής Καθηγητής.

Η ολοένα και αυξανόμενη ανάγκη για μεγαλύτερη επεξεργαστική ισχύ οδήγησε στη δημιουργία των συστημάτων *μη ομοιόμορφης προσπέλασης μνήμης* (NUMA) τα οποία αποτελούν αρχιτεκτονική εξέλιξη των κοινών *συμμετρικών πολυεπεξεργαστών* (SMPs) και τα οποία είναι εφοδιασμένα με μεγάλους αριθμούς επεξεργαστικών πυρήνων. Τα συστήματα αυτά παρέχουν κοινόχρηστο χώρο διευθύνσεων και συνεπώς επιτρέπουν την ανάπτυξη προγραμμάτων με διαδεδομένες *διεπαφές προγραμματισμού εφαρμογών* (APIs) όπως το OpenMP. Λόγω της πολύπλοκης αρχιτεκτονικής οργάνωσης των συστημάτων NUMA, η επίτευξη των βέλτιστων δυνατών επιδόσεων συνήθως απαιτεί την εκμετάλλευση πληροφοριών που σχετίζονται με την τοπολογία του υποκείμενου συστήματος, δηλαδή με το πώς είναι οργανωμένο το υλικό. Ήδη από την έκδοση 4.0 του OpenMP, άρχισαν να προδιαγράφονται λειτουργίες όπως τα OpenMP places και OpenMP processor binding policies οι οποίες σχετίζονται με την τοπολογία και επιτρέπουν στο χρήστη να ελέγξει τον τρόπο ανάθεσης των νημάτων στα διαθέσιμα επεξεργαστικά στοιχεία. Στα πλαίσια της παρούσας διπλωματικής εργασίας, έγινε πλήρης υλοποίηση των λειτουργιών OpenMP places και OpenMP processor binding policies. Επιπλέον, επανασχεδιάστηκαν οι λειτουργίες συγχρονισμού και συγκεκριμένα οι *κλήσεις φραγής* (barriers), ώστε να λειτουργούν αποδοτικά σε συστήματα NUMA.

ABSTRACT

Georgios Z. Zachos, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, September 2021.
Topology and synchronization in OpenMP for NUMA manycore systems.
Advisor: Vassilios V. Dimakopoulos, Associate Professor.

Εκτεταμένη περίληψη της εργασίας στην αντίθετη γλώσσα από αυτήν του κειμένου.
Αν το κείμενο είναι στα Ελληνικά τότε αυτή η σελίδα πρέπει να είναι στα Αγγλικά.
Αν το κείμενο είναι στα Αγγλικά τότε αυτή η σελίδα πρέπει να είναι στα Ελληνικά.

Προτεινόμενο: 2 σελίδες.
Μέγιστο: 4 σελίδες.

ΚΕΦΑΛΑΙΟ 1

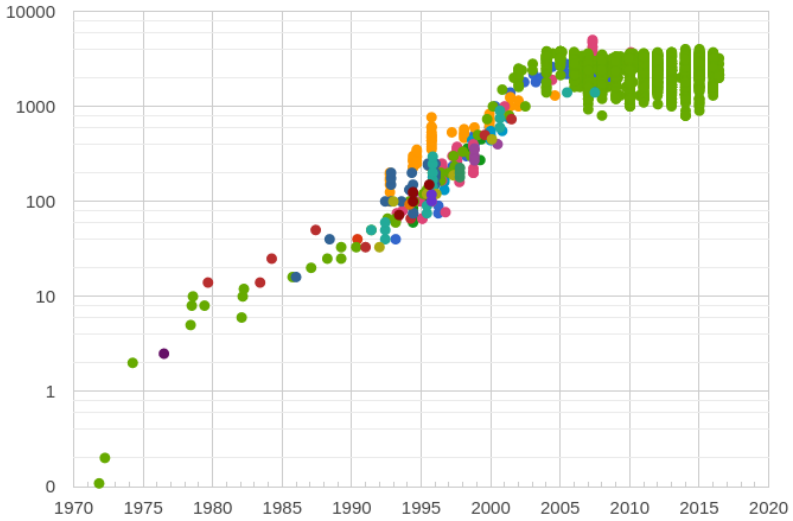
Εισαγωγή

1.1 Η ανάγκη για παράλληλα συστήματα

Η βασική ιδέα της οργάνωσης των ηλεκτρονικών υπολογιστών με τη μορφή που αυτοί είναι γνωστοί έως και σήμερα, βασίζεται στην αρχιτεκτονική von Neumann όπως αυτή περιγράφηκε το 1945 από τον μαθηματικό John von Neumann. Βάσει αυτής της περιγραφής, υπάρχει μία μονάδα επεξεργασίας η οποία επικοινωνεί με το τμήμα μνήμης, εκτελώντας εντολές και τροποποιώντας δεδομένα. Ο επεξεργαστής λαμβάνει μία-μία τις εντολές από τη μνήμη και τις εκτελεί προσπελαύνοντας ή/και τροποποιώντας δεδομένα που βρίσκονται στη μνήμη όταν αυτό καθορίζεται από την εντολή, με αυτό τον κύκλο να επαναλαμβάνεται συνεχώς. Το μοντέλο προγραμματισμού που χρησιμοποιείται στη συγκεκριμένη αρχιτεκτονική είναι γνωστό και ως σειριακό μοντέλο.

Η εποχή της πληροφορίας που ξεκίνησε στα μέσα του $20^{ού}$ αιώνα και οδήγησε σε μία οικονομία βασισμένη στην τεχνολογία της πληροφορίας, καθώς και η ολοένα και μεγαλύτερη χρήση των Η/Υ σε κάθε πτυχή της ζωής του ανθρώπου είχαν ως αποτέλεσμα την έναρξη ενός αγώνα ταχύτητας με σκοπό την κατασκευή όλο και πιο γρήγορων υπολογιστών.

Η κλιμάκωση της συχνότητας (frequency scaling) που αφορά την αύξηση της συχνότητας ενός επεξεργαστή με στόχο την επίτευξη μεγαλύτερης επίδοσης στα συστήματα που τον χρησιμοποιούν, αποτέλεσε τον κύριο λόγο αύξησης των επιδόσεων των εμπορικών Η/Υ από τα μέσα του 1980 έως και περίπου τα τέλη του 2004, όταν αυτή η προσπάθεια προσέκρουσε στο τείχος της ισχύος (power wall) όπως φαίνεται στο Σχήμα 1.1. Αυτό συνέβει καθώς η αύξηση της συχνότητας οδήγησε στην



(Η απεικόνιση προήλθε από την ιστοσελίδα του CPUDB [1])

Σχήμα 1.1: Η εξέλιξη της συχνότητας λειτουργίας (σε MHz) των μικροεπεξεργαστών.

αύξηση της καταναλισκόμενης ισχύος η οποία με τη σειρά της είχε ως αποτέλεσμα την αύξηση του κόστους λειτουργίας αλλά και την οδήγηση του υλικού στα όριά του με χαρακτηριστικό παράδειγμα την υπερθέρμανσή του.

Για να ξεπεραστεί το τείχος της ισχύος, οι κατασκευαστές των επεξεργαστών άρχισαν να ενσωματώνουν παραπάνω του ενός επεξεργαστικούς πυρήνες στο ίδιο ολοκληρωμένο κύκλωμα επεξεργαστή. Η αρχιτεκτονική αυτή βελτίωση είχε ως αποτέλεσμα τη γέννηση των πρώτων πολυπύρηνων (multicore) επεξεργαστών οι οποίοι έκαναν δυνατή την ταυτόχρονη εκτέλεση εντολών.

Στη σημερινή εποχή, τα πολυπύρηνα συστήματα είναι ευρέως διαδεδομένα καθώς απαντώνται από κινητά τηλέφωνα και ενσωματωμένους υπολογιστές χαμηλού κόστους και μεγέθους όσο μια πιστωτική κάρτα τραπέζης, μέχρι και υπολογιστικά συστήματα πολύ υψηλών επιδόσεων που διεξάγουν επιστημονικούς υπολογισμούς.

1.2 Κατηγορίες Παράλληλων Συστημάτων

Όταν μιλάμε για παράλληλα συστήματα ουσιαστικά αναφερόμαστε σε συστήματα που διαθέτουν μία συλλογή από επεξεργαστικές μονάδες που επικοινωνούν και συνεργάζονται για τη λύση ενός προβλήματος. Το πρόβλημα χωρίζεται σε επιμέρους εργασίες οι οποίες με τη σειρά τους ανατίθενται στις επεξεργαστικές μονάδες για

εκτέλεση. Παρόλο που η ιδέα των πολλαπλών επεξεργαστικών μονάδων φαίνεται απλή, προκύπτουν ζητήματα τα οποία σχετίζονται πλην άλλων με τον τρόπο:

- επικοινωνίας, συγχρονισμού και συντονισμού των επεξεργαστικών μονάδων
- διαμοιρασμού των εργασιών και
- προγραμματισμού ανάλογα με την οργάνωση του εκάστοτε συστήματος.

1.2.1 Ταξινομία του Flynn

Μία αρχική κατηγοριοποίηση των παράλληλων συστημάτων μπορεί να γίνει βάσει της ταξινομίας του Flynn και η οποία περιλαμβάνει τις εξής κατηγορίες:

- SISD (single-instruction single-data)
- SIMD (single-instruction multiple-data)
- MIMD (mutiple-instruction multiple-data)

Στην κατηγορία SISD ανήκουν οι κλασικοί σειριακοί υπολογιστές οι οποίοι εκτελούν μία εντολή τη φορά (single-instruction) επάνω σε ένα δεδομένο (single-data). Στην κατηγορία SIMD συναντάμε επίσης υπολογιστές οι οποίοι εκτελούν μία εντολή τη φορά, αλλά μπορούν να την εφαρμόσουν ταυτόχρονα σε πολλαπλά δεδομένα (multiple-data) ώστε να εκμεταλλευτούν την παραλληλία επιπέδου δεδομένων (data-level parallelism). Στη σύγχρονη εκδοχή τους απαντώνται κυρίως στις κάρτες γραφικών (GPUs) που χρησιμοποιούνται πολύ συχνά ως επιταχυντές υπολογισμών. Τέλος, η κατηγορία MIMD θεωρείται ως η κατηγορία των καθαρά παράλληλων υπολογιστών οι οποίοι μπορούν και εκτελούν ταυτόχρονα πολλαπλές εντολές, με κάθε μία να ασχολείται με διαφορετικό δεδομένο.

Όπως θα δούμε στη συνέχεια, οι υπολογιστές MIMD, μπορούν να κατηγοριοποιηθούν περαιτέρω βάσει του πώς είναι οργανωμένη η μνήμη τους.

1.2.2 Ταξινόμηση βάσει της οργάνωσης της μνήμης

Η ταξινόμηση των παράλληλων υπολογιστών σχετικά με το πώς είναι οργανωμένη η μνήμη μπορεί να γίνει είτε βάσει της φυσικής οργάνωσης της μνήμης είτε βάσει της εικόνας/άποψης που έχει ο προγραμματιστής για αυτή.

Σχετικά με την πρώτη κατηγοριοποίηση, οι παράλληλοι υπολογιστές διακρίνονται σε υπολογιστές με (φυσικά) κοινόχρηστη μνήμη (γνωστοί και ως πολυεπεξεργαστές) και σε υπολογιστές με (φυσικά) κατανεμημένη μνήμη.

Όσον αφορά την εικόνα που έχει ο προγραμματιστής για την οργάνωση της μνήμης, οι υπολογιστές μπορούν να διαχωριστούν σε υπολογιστές με κοινόχρηστο και σε υπολογιστές με κατανεμημένο χώρο διευθύνσεων. Αξίζει να σημειωθεί ότι η εικόνα από προγραμματιστική άποψη δεν ταυτίζεται απαραίτητα με την φυσική οργάνωση της μνήμης καθώς με χρήση ειδικού υλικού ή λογισμικού ένα σύστημα με φυσικά κατανεμημένη μνήμη μπορεί να παρέχει κοινόχρηστο χώρο διευθύνσεων.

Συστήματα κοινόχρηστης μνήμης

Τα συστήματα κοινόχρηστης μνήμης (SMM - Shared Memory Machines [2]) αποτελούνται από επεξεργαστές, κοινόχρηστη μνήμη (γνωστή και ως καθολική) η οποία είναι καθολικά προσβάσιμη από όλους τους επεξεργαστές και ένα δίκτυο διασύνδεσης για την επικοινωνία των επεξεργαστών με τη μνήμη. Από άποψη φυσικής οργάνωσης, η μνήμη μπορεί να αποτελείται από περισσότερα του ενός τμήματα/μέρη (modules) τα οποία όμως παρέχουν έναν κοινόχρηστο χώρο διευθύνσεων που είναι προσβάσιμος από όλους τους επεξεργαστές. Οι επεξεργαστές επικοινωνούν, συνεργάζονται και ανταλλάσσουν δεδομένα διαβάζοντας ή γράφοντας σε κοινόχρηστες μεταβλητές που βρίσκονται αποθηκευμένες στη μνήμη.

Το δίκτυο διασύνδεσης επεξεργαστών-μνήμης μπορεί να είναι ένας απλός δίαυλος (bus), ένα διακοπτικό δίκτυο (π.χ. crossbar) ή κάποιο δίκτυο πολλαπλών επιπέδων (π.χ. δίκτυο Δέλτα). Στην περίπτωση που το δίκτυο διασύνδεσης είναι δίαυλος, τότε αναφερόμαστε σε αυτά τα συστήματα ως *συμμετρικοί πολυεπεξεργαστές* (SMPs - Symmetric Multiprocessors). Οι συμμετρικοί πολυεπεξεργαστές διαθέτουν μία κοινόχρηστη μνήμη η οποία απέχει εξίσου από όλους τους επεξεργαστές και συνεπώς χαρακτηρίζονται ως συστήματα ομοιόμορφης προσπέλασης μνήμης (UMA - Uniform Memory Access). Όπως είναι φυσικό, αν οι επεξεργαστές είναι πολυπύρηνοι και διαθέτουν ιεραρχία από πολύ μικρές αλλά ταυτόχρονα πολύ γρήγορες μνήμες γνωστές ως κρυφές μνήμες (caches), τότε ο κάθε πολυπύρηνος επεξεργαστής αποτελεί ένα σύστημα SMP καθώς η πρόσβαση στην κρυφή μνήμη είναι πιο γρήγορη από την πρόσβαση στην κύρια μνήμη μέσω του διαύλου.

Επειδή το εύρος ζώνης (bandwidth) του διαύλου είναι σταθερό ανεξάρτητα από το πλήθος των επεξεργαστών που είναι συνδεδεμένοι σε αυτόν, όσο πιο πολλοί επε-

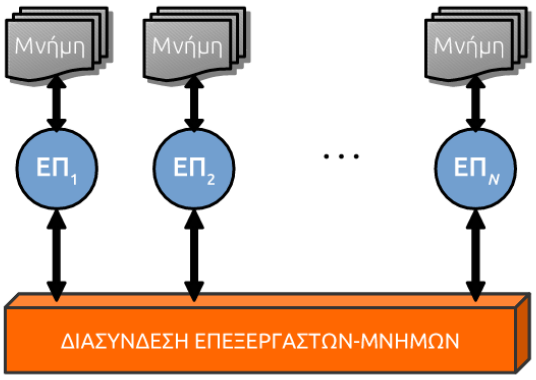
ξεργαστές είναι συνδεδεμένοι, τόσο πιο πολύ υποβαθμίζεται η αποδοτικότητα του δικτύου λόγω των συγγρούσεων πρόσβασης στο κοινό μέσο και συνεπώς τόσο περισσότερο καθυστερεί η εξυπηρέτηση προσπελάσεων στη μνήμη, αυξάνοντας έτσι τον σχετικό (effective) χρόνο προσπέλασης. Για την αποδοτική υποστήριξη μεγαλύτερου αριθμού επεξεργαστών καταφεύγουμε στη χρήση κρυφής μνήμης ή άλλου τύπου δίκτυου διασύνδεσης.

Συστήματα κατανεμημένης μνήμης

Τα συστήματα κατανεμημένης μνήμης (DMM - Distributed Memory Machines) αποτελούνται από επεξεργαστικά στοιχεία (γνωστά ως κόμβοι) και από ένα δίκτυο διασύνδεσης το οποίο επιτρέπει την επικοινωνία μεταξύ των κόμβων (Σχήμα 1.2). Κάθε κόμβος περιέχει επεξεργαστή, τοπική μνήμη και ίσως περιφερειακές συσκευές. Η τοπική μνήμη κάθε κόμβου είναι απευθείας προσβάσιμη μόνο από τον επεξεργαστή του ίδιου κόμβου, ενώ όταν κάποιος επεξεργαστής χρειάζεται να προσπελάσει την τοπική μνήμη που βρίσκεται σε κάποιον άλλο κόμβο, αυτό επιτυγχάνεται με μεταβίβαση μηνυμάτων μέσω του δικτύου διασύνδεσης.

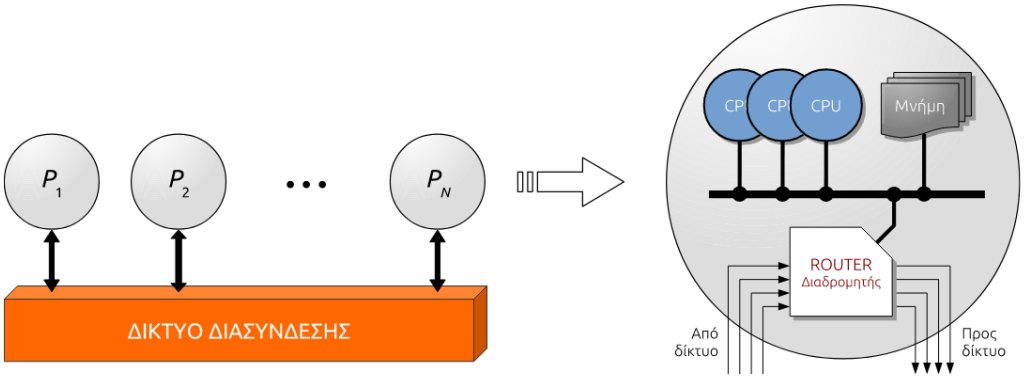
Παράλληλα συστήματα μεγαλύτερου μεγέθους μπορούν να υλοποιηθούν χρησιμοποιώντας συμμετρικούς πολυεπεξεργαστές ως κόμβους του δικτύου διασύνδεσης (Σχήμα 1.3). Σε αυτά τα συστήματα, καθίσταται δυνατή η παροχή ενός κοινόχρηστου χώρου διευθύνσεων με χρήση κατάλληλων πρωτοκόλλων συνοχής τα οποία αποκρύπτουν από τον χρήστη του συστήματος την κατανεμημένη οργάνωση της φυσικής μνήμης. Η αρχιτεκτονική που μόλις περιγράφηκε είναι γνωστή και ως κατανεμημένη κοινόχρηστη μνήμη (DSM - Distributed Shared Memory), ενώ τα συστήματα αυτά ονομάζονται συστήματα μη ομοιόμορφης προσπέλασης μνήμης (NUMA - Non-Uniform Memory Access). Σε περίπτωση ύπαρξης ιεραρχίας κρυφών μνημών στους κόμβους, χρειάζεται η χρήση ενός πρωτοκόλλου συνοχής κρυφής μνήμης (cache coherence protocol) το οποίο θα εξασφαλίζει ανά πάσα στιγμή ότι οποιαδήποτε προσπέλαση μνήμης θα επιστρέφει την πιο πρόσφατη τιμή. Τέτοια συστήματα είναι γνωστά ως cc-NUMA (Cache coherent NUMA). Τη σημερινή εποχή, οι όροι NUMA και cc-NUMA είναι συνήθως συνώνυμοι.

Συλλογές υπολογιστών που είναι συνδεδεμένοι μέσω δικτύου διασύνδεσης αφιερωμένου αποκλειστικά στη μεταξύ τους επικοινωνία ονομάζονται συστάδες (clusters). Οι συστάδες υπολογιστών χρησιμοποιούνται ευρέως λόγω της διαθεσιμότητας δικτύων διασύνδεσης υψηλών επιδόσεων όπως το Switched Gigabit Ethernet, το



(Απεικόνιση από το σύγγραμμα Παράλληλα Συστήματα και Προγραμματισμός [3])

Σχήμα 1.2: Οργάνωση συστημάτων κατανεμημένης μνήμης.



(Απεικόνιση από το σύγγραμμα Παράλληλα Συστήματα και Προγραμματισμός [3])

Σχήμα 1.3: Οργάνωση συστημάτων NUMA.

Infiniband, το Myriret κ.ά. Πολλαπλές συστάδες υπολογιστών μπορούν να διασυνδεθούν μεταξύ τους και να δημιουργήσουν συστήματα πλέγματος (grids).

Το πλήθος των κόμβων και το πλήθος των επεξεργαστών στα συστήματα κατανεμημένης μνήμης μπορεί να φτάσει τις εκατοντάδες χιλιάδες και κάποια εκατομμύρια αντίστοιχα, σε αντίθεση με τα συστήματα κοινόχρησης μνήμης όπου το πλήθος των επεξεργαστών περιορίζεται σε μερικές δεκάδες. Προφανώς, η ικανότητα κλιμάκωσης των κατανεμημένων συστημάτων εξαρτάται από την σωστή επιλογής της τοπολογίας του δικτύου διασύνδεσης.

1.3 Προγραμματισμός Παράλληλων Συστημάτων

1.3.1 Συστήματα κοινόχρηστης μνήμης

Ο προγραμματισμός των συστημάτων κοινόχρηστης μνήμης συνήθως βασίζεται στη χρήση νημάτων, δηλαδή σε ξεχωριστές ακολουθίες ελέγχου (στοίβα + μετρητής προγράμματος) που μοιράζονται δεδομένα με τα υπόλοιπα νήματα μέσω κοινόχρηστου χώρου διευθύνσεων. Η ύπαρξη κοινόχρηστων δεδομένων εγείρει ζητήματα όπως αυτό της ταυτόχρονης προσπέλασής τους από διαφορετικά νήματα, καθώς κάτι τέτοιο θα μπορούσε να οδηγήσει σε συνθήκες ανταγωνισμού (race conditions). Όταν υπάρχουν συνθήκες ανταγωνισμού, το τελικό αποτέλεσμα των ταυτόχρονων προσπελάσεων μνήμης εξαρτάται από τις σχετικές ταχύτητες εκτέλεσεις των νημάτων. Για την αποφυγή συνθηκών ανταγωνισμού και την εξασφάλιση της ακεραιότητας των δεδομένων χρησιμοποιείται ο αμοιβαίος αποκλεισμός (mutual exclusion), βάσει του οποίου μόνο ένα νήμα κάθε φορά μπορεί να εκτελεί εντολές που τροποποιούν κοινόχρηστες μεταβλητές.

Η πιο διαδεδομένη βιβλιοθήκη νημάτων και μοντέλο εκτέλεσης είναι αυτό των POSIX threads (pthreads) το οποίο παρέχει κλήσεις διαχείρισης (π.χ. δημιουργία, τερματισμό) νημάτων, κλειδαριές (mutexes), μεταβλητές συνθήκης (condition variables) και κλήσεις συγχρονισμού νημάτων όπως για παράδειγμα κλήσεις φραγής (barriers). Ο προγραμματιστής έχει την πλήρη ευθύνη για τη δημιουργία των νημάτων, την ανάθεση εργασιών σε αυτά, προεραιτικά την αντιστοίχηση των νημάτων σε επεξεργαστές, πιθανώς την συλλογή των επιμέρους αποτελεσμάτων και την σύνθεση του τελικού αποτελέσματος από αυτά, καθώς και τον τερματισμό τους. Επιπλέον, είναι υπεύθυνος για τον συγχρονισμό των νημάτων και την αποφυγή συνθηκών ανταγωνισμού με χρήση κατάλληλων προγραμματιστικών δομών.

Για τη διευκόλυνση του προγραμματισμού, έχουν αναπτυχθεί εργαλεία πιο υψηλού επιπέδου όπως το OpenMP (Open Multi-Processing) [4]. Το OpenMP είναι μία διεπαφή προγραμματισμού εφαρμογών (API - Application Programming Interface) η οποία αποτελείται από οδηγίες (directives) προς τον μεταφραστή, ρουτίνες βιβλιοθήκης και μεταβλητές περιβάλλοντος (environment variables) και συντελεί στην συγγραφή πολυνηματικού κώδικα για συστήματα κοινόχρηστης μνήμης. Πολύ σημαντικό είναι το γεγονός ότι το OpenMP δίνει τη δυνατότητα παραλληλοποίησης του υπάρχοντα σειριακού κώδικα χωρίς την τροποποίησή του, παρά μόνο με την προσθήκη των ειδικών οδηγιών οι οποίες μπορούν να αγνοηθούν σε περίπτωση που

δεν υποστηρίζονται από κάποιο μεταφραστή. Επίσης, η διαχείριση των νημάτων γίνεται αυτόματα, ενώ όλα τα υπόλοιπα ζητήματα που σχετίζονται με τον συγχρονισμό, ανάθεση εργασιών κλπ απλοποιούνται σε μεγάλο βαθμό. Η απλότητα του OpenMP αλλά και όλες οι διευκολύνσεις που προσφέρει, το κάνουν να βρίσκεται στην κορυφή των προτιμήσεων για προγραμματισμό συστημάτων κοινόχρηστης μνήμης, αφού μπορεί να χρησιμοποιηθεί ακόμα και από προγραμματιστές χωρίς ιδιαίτερη εμπειρία ή γνώσεις σχετικές με την παράλληλη επεξεργασία. Περισσότερα για το πρότυπο OpenMP θα ειπωθούν στο Κεφάλαιο 2.

1.3.2 Συστήματα κατανεμημένης μνήμης

Ο προγραμματισμός των συστημάτων κατανεμημένης μνήμης συνήθως βασίζεται στη μεταβίβαση μηνυμάτων μεταξύ αυτόνομων διεργασιών (προγράμματα υπό εκτέλεση) οι οποίες βρίσκονται σε διαφορετικούς κόμβους και δεν μοιράζονται κοινόχρηστες μεταβλητές, αλλά πραγματοποιούν μεταξύ τους αποστολή και λήψη μηνυμάτων. Η μη ύπαρξη κοινόχρηστων δεδομένων εξαλείφει την ανάγκη για αμοιβαίο αποκλεισμό αλλά ταυτόχρονα δυσκολεύει τον συγχρονισμό μεταξύ των διεργασιών. Ο προγραμματιστής είναι υπεύθυνος να καθορίσει πότε σταματούν οι υπολογισμοί και πότε ξεκινούν οι επικοινωνίες, το περιεχόμενο, τον αποστολέα και τους παραλήπτες των μηνυμάτων, καθώς και να αποφασίσει τον τύπο της επικοινωνίας που θα χρησιμοποιήσει (σύγχρονη ή ασύγχρονη).

Στις σύγχρονες επικοινωνίες μία διεργασία που θέλει να στείλει (λάβει) δεδομένα μπλοκάρει στην αντίστοιχη κλήση αποστολής (λήψης) μέχρις ότου η διαδικασία παραλήπτης (αποστολέας) παραλάβει (αποστείλει) τα δεδομένα. Ο τρόπος λειτουργίας των σύγχρονων επικοινωνιών τις καθιστά ιδανικές για την επίτευξη συγχρονισμού μεταξύ των διεργασιών. Αντίθετα, στις ασύγχρονες επικοινωνίες, η διεργασία δεν μπλοκάρει αλλά συνεχίζει κανονικά την εκτέλεση της.

Λόγω της φύσης του μοντέλου μεταβίβασης μηνυμάτων, ο προγραμματιστής θα πρέπει να δώσει προσοχή στο πώς θα ελαχιστοποιήσει την επικοινωνία μεταξύ διαφορετικών κόμβων, καθώς η προσπέλαση της τοπικής μνήμης κάθε κόμβου είναι πολύ πιο γρήγορη απ' ότι η προσπέλαση της μνήμης άλλων κόμβων. Γι' αυτό το λόγο πρέπει να σχεδιαστεί με σύνεση ο διαμοιρασμός των δεδομένων ανάμεσα στους κόμβους, καθώς και η ανάθεση φόρτου σε κάθε διεργασία ώστε εκτός από την αποφυγή καθυστερήσεων σε απομακρυσμένες προσπελάσεις, να αποφευχθεί

και συμφόρηση του δίκτυου.

Δημοφιλές πρότυπο μεταβίβασης μηνυμάτων αποτελεί το MPI (Message Passing Interface) με τις δύο πιο γνωστές υλοποιήσεις του να είναι το Open MPI και το MPICH. Το MPI υποστηρίζει τη μεταβίβαση μηνυμάτων στις γλώσσες C, C++ και Fortran με στόχο την ανάπτυξη μεταφέρσιμων παράλληλων εφαρμογών μεγάλης κλίμακας. Μεγάλο προτέρημα αποτελεί η απόκρυψη των πληροφοριών χαμηλού επιπέδου όπως ο τύπος του υποκείμενου δικτύου, το λειτουργικό σύστημα του κάθε κόμβου, αλλά και η ύπαρξη βοηθητικών προγραμματιστικών δομών, όπως οι συλλογικές και μη επικοινωνίες που μπορούν να πραγματοποιηθούν χωρίς τη γνώση δικτυακού προγραμματισμού (sockets).

1.4 Αντικείμενο της Διπλωματικής Εργασίας

Το αντικείμενο της παρούσας διπλωματικής εργασίας είναι η επίτευξη καλύτερων επιδόσεων στον ερευνητικό μεταφραστή OMPi, μέσω της εκμετάλλευσης πληροφοριών που σχετίζονται με την τοπολογία του συστήματος. Ο OMPi είναι ένας μεταφραστής πηγαίου σε πηγαίο κώδικα (source-to-source) που υποστηρίζει τη διεπαφή προγραμματισμού εφαρμογών OpenMP και αναπτύσσεται από την Ομάδα Παράλληλης Επεξεργασίας του Πανεπιστημίου Ιωαννίνων.

Στην έκδοση 4.0 του OpenMP προστέθηκε η δυνατότητα ανάθεσης (binding) των νημάτων σε σύνολα από συγκεκριμένους επεξεργαστές (τα λεγόμενα OpenMP places) ώστε να εκτελεστούν σε αυτά. Η ανάθεση των νημάτων σε OpenMP places γίνεται βάσει των διάφορων διαθέσιμων πολιτικών (γνωστές ως processor binding policies). Ο χρήστης έχει τη δυνατότητα προσδιορισμού της λίστας των διαθέσιμων OpenMP places αλλά και της πολιτικής ανάθεσης των νημάτων σε αυτά. Στο OpenMP 5.1 (Νοέμβριος 2020) επεκτάθηκαν περαιτέρω οι τρόποι προσδιορισμού των OpenMP places οι οποίοι περιγράφονται με λεπτομέρεια στην Ενότητα 3.4. Στα πλαίσια της διπλωματικής εργασίας υλοποιήθηκε πλήρης υποστήριξη των προαναφερθέντων λειτουργιών και οι οποίες συμμορφώνονται με τις προδιαγραφές του OpenMP 5.1.

Επιπρόσθετα, ανεξάρτητα από το πρότυπο OpenMP, έγινε εκμετάλλευση της τοπολογίας για τη βελτίωση των κλήσεων φραγής που υλοποιούνται στο μεταφραστή OMPi. Συγκεκριμένα, σε περίπτωση που αναγνωρίζεται το υποκείμενο σύστημα ως

σύστημα μη ομοιόμορφης προσπέλασης μνήμης (NUMA), τότε εφαρμόζεται ένας νέος αλγόριθμος για τις κλήσεις φραγής που λαμβάνει υπόψην του την τοπολογία για την επίτευξη ταχύτερου συγχρονισμού των νημάτων.

1.5 Διάρθρωση της Διπλωματικής Εργασίας

Η διπλωματική εργασία είναι οργανωμένη με τον ακόλουθο τρόπο:

- Κεφάλαιο 2: Παρουσίαση της διεπαφής προγραμματισμού εφαρμογών OpenMP και περιγραφή της σύνταξης των πιο διαδεδομένων οδηγιών. Αναφορά στους διάφορους μεταφραστές που υποστηρίζουν το OpenMP και περιγραφή του μεταφραστή OMPi.

- Κεφάλαιο 3: Εισαγωγή στην τοπολογία, περιγραφή των συστημάτων NUMA, εργαλεία για την αναγνώριση και εκμετάλλευση της τοπολογίας και η χρήση της τελευταίας στα πλαίσια του OpenMP. Παρουσίαση των λειτουργιών του OpenMP που σχετίζονται με την τοπολογία και υλοποιήθηκαν στον μεταφραστή OMPi.

- Κεφάλαιο 4: Περιγραφή του τρόπου επίτευξης συγχρονισμού με χρήση κλήσεων φραγής (barriers) και παρουσίαση των ιδιαίτερων χαρακτηριστικών τους στα πλαίσια του OpenMP. Επεξήγηση της υλοποίησης των κλήσεων φραγής στον μεταφραστή OMPi και των βελτιώσεων που έγιναν για την επίτευξη καλύτερων επιδόσεων σε πολυπύρηνα συστήματα NUMA.

- Κεφάλαιο 5: Παρουσίαση και ανάλυση των αποτελεσμάτων από την εκτέλεση διάφορων πειραμάτων για τη μέτρηση της απόδοσης του μεταφραστή OMPi μετά την υλοποίηση των λειτουργιών που σχετίζονται με την τοπολογία και των βελτιώσεων στις κλήσεις φραγής.

- Κεφάλαιο 6: Σύνοψη της διπλωματικής εργασίας και αναφορά σε μελλοντικές εργασίες.

ΚΕΦΑΛΑΙΟ 2

Η διεπαφή προγραμματισμού εφαρμογών OpenMP

2.1 Εισαγωγή στο OpenMP

Όπως αναφέρθηκε ήδη στην Υποενότητα 1.3.1, η διεπαφή προγραμματισμού εφαρμογών OpenMP (Open Multi-Processing) αναπτύχθηκε για τη διευκόλυνση της ανάπτυξης πολυνηματικών εφαρμογών για συστήματα κοινόχρηστης μνήμης. Οι γλώσσες οι οποίες υποστηρίζονται είναι οι C, C++ και Fortran. Το OpenMP αποτελείται από:

- Οδηγίες (directives): Συνιστούν οδηγίες προς τον μεταφραστή για το πώς και τι να εκτελέσει πολυνηματικά. Στις γλώσσες C/C++ χρησιμοποιείται ο μηχανισμός που είναι γνωστός ως pragmas και απευθύνεται στον προεπεξεργαστή. Αυτές οι οδηγίες προστίθενται στο υπάρχον σειριακό πρόγραμμα και μπορούν να αγνοηθούν από έναν μεταφραστή που δεν τις υποστηρίζει. Αυτό είναι μεγάλο προσόν καθώς το ίδιο πρόγραμμα μπορεί να εκτελεστεί σειριακά ή παράλληλα.

- Ρουτίνες βιβλιοθήκης: σύνολο συναρτήσεων οι οποίες βοηθούν στη διαχείριση των χαρακτηριστικών των νημάτων και του περιβάλλοντος εκτέλεσης. Για παράδειγμα, η συνάρτηση `omp_set_num_threads(int)` καθορίζει το πλήθος των νημάτων που θα συμμετάσχουν σε επερχόμενη πολυνηματική εκτέλεση (παράλληλη περιοχή).

- Μεταβλητές περιβάλλοντος: χρησιμοποιούνται για τον καθορισμό διάφορων

χαρακτηριστικών των νημάτων και του περιβάλλοντος εκτέλεσης. Οι τιμές των μεταβλητών περιβάλλοντος οριστικοποιούνται στην αρχή της εκτέλεσης και χρησιμοποιούνται ως προκαθορισμένες τιμές. Κάποιες από αυτές τις προκαθορισμένες αυτές τιμές μπορούν να τροποποιηθούν σε χρόνο εκτέλεσης με χρήση των διαθέσιμων ρουτινών βιβλιοθήκης.

Από τη στιγμή που η παραλληλοποίηση ενός σειριακού προγράμματος μπορεί να γίνει με την απλή προσθήκη οδηγιών στο υπάρχοντα κώδικα, η διαδικασία της παραλληλοποίησης απλοποιείται σε μεγάλο βαθμό και μπορεί να γίνει σταδιακά (π.χ. παραλληλοποίηση ενός βρόγχου for τη φορά) και χωρίς τη χρήση διαφορετικής λογικής, όπως για παράδειγμα θα γινόταν με χρήση των POSIX Threads.

Ένα από τα σχετικά καινούρια χαρακτηριστικά του OpenMP είναι η δυνατότητα αποστολής κώδικα για εκτέλεση σε συσκευές όπως κάρτες γραφικών γενικού σκοπού (GPGPUs), συνεπεξεργαστές (coprocessors) ή διάφορους άλλους επιταχυντές. Το πλεονέκτημα είναι ότι ο προγραμματιστής δεν χρειάζεται να μάθει να προγραμματίζει σε γλώσσες προγραμματισμού χαμηλού επιπέδου όπως OpenCL, CUDA κλπ για να αξιοποιήσει την επεξεργαστική ισχύ των διαθέσιμων συσκευών. Αυτό το χαρακτηριστικό είναι ιδιαίτερα βοηθητικό σε συστήματα υπολογισμών υψηλών επιδόσεων (High Performance Computing - HPC) όπου υπάρχει η τάση να εξοπλίζεται ένα σύνολο των υπολογιστικών κόμβων με ισχυρούς επιταχυντές για την επίτευξη μεγαλύτερων επιδόσεων. Για παράδειγμα, ο υπερυπολογιστής Aris του Εθνικού Δικτύου Υποδομών και Έρευνας (ΕΔΥΤΕ - GRNET) διαθέτει πλην άλλων:

- 18 κόμβους με 2 επεξεργαστές Intel Xeon E5-2660v3 και 2 συνεπεξεργαστές Intel Xeon Phi 7120P.
- 44 κόμβους με 2 επεξεργαστές Intel Xeon E5-2660v3 και 2 κάρτες γραφικών NVIDIA K40.
- 1 κόμβο με 2 επεξεργαστές Intel E5-2698v4 και 8 κάρτες γραφικών NVIDIA V100 για εκτέλεση προγραμμάτων μηχανικής μάθησης.

Λαμβάνοντας υπόψην ότι η διαχείριση των νημάτων μετατίθεται από τον προγραμματιστή στο μεταφραστή, ότι απλοποιούνται ουσιώδη ζητήματα ενός παράλληλου προγράμματος όπως η επίτευξη συγχρονισμού και αμοιβαίου αποκλεισμού, καθώς επίσης ότι μπορούν να αξιοποιηθούν επιταχυντές χωρίς γνώση του πως προγραμματίζονται, γίνεται εύκολα αντιληπτό ότι το OpenMP είναι ένα προσιτό ερ-

γαλείο ακόμα και για άτομα χωρίς μεγάλη εμπειρία στον παράλληλο προγραμματισμό. Αυτό το χαρακτηριστικό του OpenMP το κάνει ιδιαίτερα διαδεδομένο σε χρήστες που το υπόβαθρό τους διαφέρει από αυτό της επιστήμης της πληροφορικής, όπως για παράδειγμα φυσικοί, χημικοί, αστρονόμοι κόκ. Ταυτόχρονα όμως, η απόκρυψη των λεπτομερειών χαμηλού επιπέδου είναι πιθανό να καταστήσει σε ορισμένες περιπτώσεις μη εφικτή την εξασφάλιση της μέγιστης αποδοτικότητας του παραλληλισμού.

2.2 Το προγραμματιστικό μοντέλο του OpenMP

Το OpenMP βασίζεται στη χρήση πολλαπλών νημάτων όπως άλλωστε συνηθίζεται στον προγραμματισμό συτημάτων κοινόχρηστης μνήμης καθώς και στο προγραμματιστικό μοντέλο fork-join που συναντάται στις διεργασίες.

Στο μοντέλο fork-join, η εκτέλεση ξεκινάει σειριακά από ένα νήμα (γνωστό ως αρχηγός - master) και σε προκαθορισμένα σημεία όπου απαιτείται παράλληλη εκτέλεση, δημιουργούνται επιπλέον νήματα τα οποία μαζί με το νήμα-αρχηγό συμμετέχουν στον παράλληλο υπολογισμό. Τα σημεία στα οποία πραγματοποιείται παράλληλη εκτέλεση είναι γνωστά ως παράλληλες περιοχές (parallel sections/regions). Μόλις ο παράλληλος υπολογισμός τελειώσει τα νήματα που δημιουργήθηκαν τερματίζουν και η εκτέλεση συνεχίζεται σειριακά από το νήμα-αρχηγό.

Ενδιαφέρον είναι το γεγονός ότι υποστηρίζονται αυθαίρετα πολλές εμφωλευμένες παράλληλες περιοχές, δηλαδή ένα οποιοδήποτε νήμα το οποίο συμμετέχει στην εκτέλεση μιας παράλληλης περιοχής μπορεί να αποτελέσει με τη σειρά του νήμα-αρχηγός και να δημιουργήσει μία νέα παράλληλη περιοχή. Οι εμφωλευμένες παράλληλες περιοχές μπορούν να χρησιμοποιηθούν για την ανάθεση εργασιών με το επιθυμητό μέγεθος κόκκου παραλληλίας (granularity) σε κάθε νήμα.

Είναι χρήσιμο να αναφερθεί πως όταν ξεκινάει να εκτελείται μία διεργασία, χρησιμοποιείται ένα νήμα για την εκτέλεση των εντολών σειριακά και το οποίο νήμα στα πλαίσια του OpenMP ονομάζεται αρχικό νήμα (initial thread).

2.3 Εισαγωγή στη διεπαφή προγραμματισμού OpenMP

Το πλήθος των σελίδων των προδιαγραφών του OpenMP τείνει να αυξάνεται εκθετικά στις τελευταίες εκδόσεις με αποτέλεσμα να είναι αδύνατο να περιγραφούν όλες οι δυνατότητές του στα πλαίσια μίας διπλωματικής εργασίας. Για το λόγο αυτό, θα γίνει περιγραφή ενός υποσυνόλου των διαθέσιμων λειτουργιών, πολλές από τις οποίες αποτελούν τις πιο συνηθισμένες και καλύπτουν τις ανάγκες της πλειοψηφίας των διαθέσιμων εφαρμογών OpenMP. Αυτές οι πιο διαδεδομένες λειτουργίες είναι εικοσιμία (21) στο πλήθος και αποτελούν το λεγόμενο OpenMP Common Core @Ref.

2.3.1 Σύνταξη οδηγιών (directives)

Η γενική μορφή μίας οδηγίας στο OpenMP είναι της μορφής:

```
#pragma omp directive-name [[,] clause [[,] clause] ... ] <new-line>
```

Κάθε οδηγία ξεκινάει υποχρεωτικά με το `#pragma omp` και ακολουθεί το όνομά της που καθορίζει ποιά λειτουργία θα εκτελεστεί στην περιοχή του κώδικα που ακολουθεί. Υπάρχουν διάφορες διαθέσιμες οδηγίες οι οποίες μπορούν να χρησιμοποιηθούν, πλην άλλων, για τη δημιουργία παράλληλης ομάδας, το συγχρονισμό των νημάτων και τον ορισμό της πολιτικής με την οποία τα νήματα θα ανατεθούν στους διαθέσιμους επεξεργαστές.

Στη συνέχεια τοποθετούνται προαιρετικά φράσεις (clauses) οι οποίες παραμετροποιούν τις συνθήκες υπό τις οποίες θα εκτελεστεί η λειτουργία που ορίζει η οδηγία. Για παράδειγμα, η φράση `num_threads` μπορεί να χρησιμοποιηθεί σε συνδιασμό με την οδηγία δημιουργίας παράλληλης ομάδας για να καθορίσει το επιθυμητό πλήθος των νημάτων που θα συμμετάσχουν στην παράλληλη εκτέλεση. Η σειρά με την οποία αναγράφονται οι φράσεις δεν έχει σημασία.

Το τέλος της οδηγίας σηματοδοτείται από αλλαγή γραμμής (newline).

2.3.2 Η οδηγία parallel

Η οδηγία `parallel` συντάσσεται ως εξής:

```
#pragma omp parallel [clause [[,] clause] ... ] <new-line>
structured-block
```

Όταν ένα οποιοδήποτε νήμα συναντήσει μία οδηγία `parallel`, δημιουργείται μία ομάδα νημάτων η οποία εκτελεί την παράλληλη περιοχή. Μια παράλληλη περιοχή υποδηλώνει ένα τμήμα κώδικα το οποίο προορίζεται για πολυνηματική εκτέλεση. Στη συγκεκριμένη περίπτωση, ο κώδικας που περιέχεται στο δομημένο τμήμα κώδικα (structured block) που ακολουθεί την οδηγία `parallel` είναι αυτός που εν τέλει θα εκτελεστεί παράλληλα. Ως δομημένο τμήμα κώδικα ορίζεται μία εντολή ή μία ακολουθία εντολών που περικλείονται από άγκιστρα.

Το πλήθος των νημάτων (N) που συμμετέχουν σε μια παράλληλη ομάδα είναι σταθερό καθόλη τη διάρκειά της. Το νήμα το οποίο συνάντησε την οδηγία `parallel` λαμβάνει το ρόλο του αρχηγού (master[1]) της ομάδας, ενώ συμμετέχει και αυτό στον παράλληλο υπολογισμό μαζί με τα υπόλοιπα $N-1$ νήματα που δημιουργήθηκαν.

Μέσα σε μία παράλληλη περιοχή μπορούν να χρησιμοποιηθούν τα αναγνωριστικά των νημάτων για την αναγνώριση του κάθε νήματος. Τα αναγνωριστικά είναι ακέραιοι αριθμοί και συγκεκριμένα για μία ομάδα N νημάτων, η τιμή τους κυμαίνεται από μηδέν (για τον αρχηγό της ομάδας) έως και ένα λιγότερο από το μέγεθος της ομάδας, δηλαδή $N-1$.

Καθώς ο χρόνος που χρειάζεται ένα νήμα για να εκτελέσει την παράλληλη περιοχή εξαρτάται από πολλούς παράγοντες, όπως για παράδειγμα το πόσο δίκαιη είναι η κατανομή του φόρτου μεταξύ των νημάτων της ομάδας, στο τέλος της παράλληλης περιοχής υπονοείται μία κλήση φραγής (barrier). Η κλήση αυτή εξασφαλίζει ότι τα νήματα θα περιμένουν στην κλήση φραγής μέχρις ότου όλα τα νήματα να φτάσουν σε αυτό το σημείο πριν τους επιτραπεί να τερματίσουν και το νήμα-αρχηγός συνεχίσει την εκτέλεση του υπόλοιπου προγράμματος που ακολουθεί της παράλληλης περιοχής.

Την οδηγία `parallel` μπορεί προεραιτικά να ακολουθούν φράσεις διαμοιρασμού δεδομένων που θα δούμε στην Υποενότητα 2.3.3 καθώς και οι φράσεις `num_threads`, `redution` και `proc_bind` που περιγράφονται αμέσως μετά.

Η φράση num_threads

Η φράση `num_threads` δέχεται ως παράμετρο έναν ακέραιο αριθμό και καθορίζει το επιθυμητό πλήθος των νημάτων που θα εκτελέσουν την παράλληλη περιοχή.

[1]Στο πρότυπο του OpenMP 5.1 (Νοέμβριος 2020) καθορίζεται αλλαγή της ονομασίας master σε primary. Στο παρόν κείμενο θα ακολουθηθεί η ορολογία master (αρχηγός) για λόγους συμβατότητας με τους υπάρχοντες πόρους της Ομάδας Παράλληλης Επεξεργασίας του Πανεπιστημίου Ιωαννίνων.

Η φράση reduction

Η φράση `reduction` χρησιμοποιείται για τη διεκπεραίωση μιας αριθμητικής πράξης από τα νήματα της ομάδας πάνω σε μια κοινόχρηστη μεταβλητή, εξασφαλίζοντας την αποφυγή συνθηκών ανταγωνισμού. Η απλουστευμένη σύνταξη της φράσης είναι η εξής:

reduction(*reduction-identifier : list***)**

Ο `reduction-identifier` είναι η πράξη η οποία θέλουμε να εφαρμοστεί πάνω στις μία ή περισσότερες μεταβλητές που αναγράφονται στη λίστα `list` και είναι διαχωρισμένες μεταξύ τους με κόμμα. Οι διαθέσιμες πράξεις είναι οι `+`, `-`, `*`, `&`, `|`, `^`, `&&` και `||`.

Η φράση proc_bind

Η φράση `proc_bind` χρησιμοποιείται για τον ορισμό της πολιτικής με την οποία τα νήματα θα ανατεθούν στους διαθέσιμους επεξεργαστές. Οι διαθέσιμες επιλογές είναι οι `master/primary`, `close` και `spread`. Περισσότερες λεπτομέρειες θα δούμε στην Υποενότητα 3.4 όπου θα ασχοληθούμε λεπτομερώς με την τοπολογία του υποκείμενου συστήματος και τον τρόπο ανάθεσης των νημάτων OpenMP στους επεξεργαστές.

2.3.3 Φράσεις διαμοιρασμού δεδομένων

Οι φράσεις διαμοιρασμού δεδομένων χρησιμοποιούνται για να δηλώσουν τον τρόπο διαμοιρασμού μίας ή περισσότερων μεταβλητών μεταξύ των νημάτων ως εξής:

- `shared`: Οι μεταβλητές είναι κοινόχρηστες.
- `private`: Οι μεταβλητές είναι ιδιωτικές καθώς δημιουργείται από ένα αντίγραφο για κάθε νήμα.
- `firstprivate`: Οι μεταβλητές είναι ιδιωτικές και καθεμιά αρχικοποιείται στην τιμή της αντίστοιχης αρχικής μεταβλητής.
- `lastprivate`: Οι μεταβλητές είναι ιδιωτικές και μετά το πέρας της εκτέλεσης η τιμή της καθεμιάς τους θα χρησιμοποιηθεί για την ενημέρωση της τιμής της αντίστοιχης αρχικής μεταβλητής.

- `default`: Καθορίζει την προκαθορισμένη πολιτική διαμοιρασμού με διαθέσιμες επιλογές να είναι οι `shared`, `firstprivate`, `private`, `none`.

2.3.4 Οδηγίες Διαμοιρασμού Εργασίας

Το OpenMP παρέχει τη δυνατότητα κατανομής του φόρτου εργασίας ανάμεσα στα νήματα της ομάδας μέσω των οδηγιών που καθορίζουν περιοχές διαμοιρασμού εργασίας (worksharing regions).

Η βασική διαφορά μίας περιοχής διαμοιρασμού εργασίας με μία παράλληλη περιοχή είναι ότι στην πρώτη σε αντίθεση με τη δεύτερη, δεν δημιουργούνται νέα νήματα αλλά χρησιμοποιούνται τα υπάρχοντα. Βάσει αυτής της παρατήρησης συμπεραίνουμε ότι οι περιοχές διαμοιρασμού έχουν νόημα όταν εντοπίζονται εντός παράλληλων περιοχών. Είναι πιθανό μία παράλληλη ομάδα που αποτελείται από ένα μόνο νήμα να συναντήσει μία οδηγία περιοχής διαμοιρασμού εργασίας. Όπως είναι προφανές, σε αυτή την περίπτωση, η εκτέλεση είναι σειριακή και όχι παράλληλη.

Στο τέλος των περιοχών διαμοιρασμού εργασίας, όπως και στο τέλος των παράλληλων περιοχών, υπονοείται μία κλήση φραγής για την επίτευξη συγχρονισμού μεταξύ των νημάτων, με τη διαφορά ότι στις πρώτες η κλήση φραγής μπορεί να παραληφθεί με τη χρήση της φράσης `nowait`.

Οδηγία sections

Χρησιμοποιείται για την κατανομή μη επαναληπτικών (non-iterative) εργασιών και συντάσσεται ως εξής:

```
#pragma omp sections [clause [[,] clause] ... ] <new-line>
{
[ #pragma omp section <new-line>
<structured-block> ]
[ #pragma omp section <new-line>
<structured-block> ]
...
}
```

Κάθε δομημένο τμήμα κώδικα που ακολουθεί μία οδηγία `#pragma omp section` που περιέχεται μέσα στην οδηγία `sections` θα ανατεθεί σε ένα νήμα και θα εκτε-

λεστεί ακριβώς μία φορά. Συνήθεις φράσεις αποτελούν οι `private`, `firstprivate`, `lastprivate` και `reduction`.

Οδηγία single

Η οδηγία αυτή καθορίζει ότι το δομημένο τμήμα κώδικα που την ακολουθεί θα εκτελεστεί μόνο από ένα νήμα (όχι απαραίτητα το νήμα-αρχηγό) και η σύνταξή της είναι η εξής:

```
#pragma omp single [clause [[,] clause] ... ] <new-line>
<structured-block>
```

Συνήθεις φράσεις που ακολουθούν είναι οι `private` και `firstprivate`.

2.3.5 Οδηγίες Διαμοιρασμού Εργασίας Βρόγχου

Οι οδηγίες διαμοιαρμού εργασίας βρόγχου είναι αντίστοιχες με τις οδηγίες διαμοιρασμού εργασίας που είδαμε στην Υποενότητα 2.3.4, με τη διαφορά ότι αφορούν την κατανομή των επαναλήψεων ενός βρόγχου στα νήματα.

Οδηγία for

Χρησιμοποιείται για την κατανομή των επαναλήψεων ενός βρόγχου[2] `for` και συντάσσεται ως εξής:

```
#pragma omp for [clause [[,] clause] ... ] <new-line>
<loop-nest>
```

Την οδηγία αυτή ακολουθεί υποχρεωτικά βρόγχος `for` ενώ συνήθεις φράσεις αποτελούν οι `private`, `firstprivate`, `lastprivate`, `reduction` και `schedule`.

Η φράση `schedule` χρησιμοποιείται για τον καθορισμό της πολιτικής με την οποία θα διαμοιραστούν οι επαναλήψεις ενός βρόγχου `for` στα νήματα και η απλουστευμένη σύνταξη της είναι η ακόλουθη:

```
schedule(kind[, chunk_size])
```

[2]Η σύνταξη του βρόγχου δεν μπορεί να είναι τόσο αυθαίρετη όσο επιτρέπει το συντακτικό των γλωσσών C/C++, αλλά στα πλαίσια αυτής της εργασίας δεν θα μας απασχολήσει αυτό το ζήτημα.

Η τιμή `kind` καθορίζει τον τρόπο διαμοιρασμού των επαναλήψεων ενώ η τιμή `chunk_size` καθορίζει το μέγεθος του κόκκου παραλληλίας (granularity) που αναλαμβάνει το κάθε νήμα. Η χρονοδρομολόγηση των επαναλήψεων γίνεται όπως περιγράφεται ακολούθως:

- `static`: Συνεχόμενες επαναλήψεις διασπώνται σε τμήματα μεγέθους όσο η τιμή `chunk_size` και ανατίθενται κυκλικά (round-robin) στα νήματα βάσει του αναγνωριστικού τους. Σε περίπτωση που δεν έχει καθοριστεί συγκεκριμένη τιμή `chunk_size`, το σύνολο των επαναλήψεων χωρίζεται σε τόσα ισομεγέθη τμήματα όσα και το πλήθος των νημάτων.

- `dynamic`: Συνεχόμενες επαναλήψεις διασπώνται σε τμήματα μεγέθους όσο η τιμή `chunk_size` και ανατίθενται στα νήματα όταν αυτά ζητήσουν το επόμενο τμήμα προς εκτέλεση. Η δυναμική ανάθεση συνεχίζεται μέχρι να ανατεθούν όλα τα τμήματα. Σε περίπτωση που δεν έχει καθοριστεί συγκεκριμένη τιμή `chunk_size`, τότε τα τμήματα έχουν μέγεθος ίσο με 1.

- `guided`: Η πολιτική αυτή μοιάζει στην πολιτική `dynamic` με τη διαφορά ότι το μέγεθος των τμημάτων δεν είναι σταθερό. Συγκεκριμένα, για `chunk_size` ίσο με 1 (k), το μέγεθος του πρώτου τμήματος ισούται με το πλήθος όλων των επαναλήψεων διαιρεμένο με το πλήθος των νημάτων, με το μέγεθος των επόμενων τμημάτων να μειώνεται εκθετικά μέχρι να ισούται με 1 (k).

- `auto`: Η επιλογή πολιτικής και μεγέθους κόκκου παραλληλίας μεταφέρεται από τον προγραμματιστή στο μεταφραστή ή στο σύστημα χρόνου εκτέλεσης (runtime).

- `runtime`: Η πολιτική και το μέγεθος κόκκου παραλληλίας καθορίζονται σε χρόνο εκτέλεσης βάσει της τιμής της μεταβλητής περιβάλλοντος `OMP_SCHEDULE`.

Στην περίπτωση των πολιτικών `auto` και `runtime` απαγορεύεται ο προσδιορισμός τιμής `chunk_size`.

2.3.6 Η οδηγία task

Η οδηγία αυτή ορίζει μία εργασία προς εκτέλεση και συντάσσεται ως εξής:

```
#pragma omp task [clause [[,] clause] ... ] <new-line>
<structured-block>
```

Όταν ένα νήμα συναντήσει την οδηγία `task`, δημιουργεί μία νέα εργασία που αποτελείται από τον κώδικα του δομημένου τμήματος κώδικα και το περιβάλλον δεδομένων (data environment) που χρειάζεται η εργασία για να διεκπεραιωθεί. Η εκτέλεση της εργασίας μπορεί να πραγματοποιηθεί από οποιοδήποτε νήμα και δεν είναι γνωστό πότε θα ξεκινήσει. Να σημειωθεί ότι υποστηρίζονται εμφωλευμένες οδηγίες `task`.

Συνήθεις φράσεις αποτελούν οι `default`, `private`, `firstprivate` και `shared`.

2.3.7 Οδηγίες συγχρονισμού

Στα προγράμματα OpenMP ως προγράμματα για συστήματα κοινόχρηστης μνήμης, σημαντικός είναι ο ρόλος του συγχρονισμού για την εξασφάλιση της συνέπειας των δεδομένων και της ορθότητας του προγράμματος. Για το λόγο αυτό, το OpenMP παρέχει έτοιμες λειτουργίες συγχρονισμού ως οδηγίες.

Οδηγία atomic

Απλουστευμένη σύνταξη:

```
#pragma omp atomic <new-line>
<statement>
```

Εξασφαλίζει ότι μια θέση μνήμης προσβαίνεται ατομικά εξαλείφοντας την πιθανότητα πολλαπλών ταυτόχρονων προσβάσεων από διαφορετικά νήματα.

Οδηγία barrier

Σύνταξη:

```
#pragma omp barrier <new-line>
```

Η γνωστή κλήση φραγής η οποία εξασφαλίζει ότι όλα τα νήματα της ομάδας θα περιμένουν σε αυτό το σημείο πριν μπορέσουν να συνεχίσουν την εκτέλεση με τις εντολές που ακολουθούν.

Οδηγία critical

Απλουστευμένη σύνταξη:

```
#pragma omp critical <new-line>
<structured-block>
```

Εξασφαλίζει ότι το δομημένο τμήμα κώδικα που ακολουθεί θα εκτελείται από ένα νήμα μόνο τη φορά.

Οδηγία taskwait

Απλουστευμένη σύνταξη:

```
#pragma omp taskwait <new-line>
```

Εξασφαλίζει ότι οι εργασίες-παιδιά της τρέχουσας εργασίας (task) θα έχουν ολοκληρωθεί πριν συνεχιστεί η εκτέλεση μετά από αυτό το σημείο.

2.3.8 Ρουτίνες βιβλιοθήκης χρόνου εκτέλεσης

- `void omp_set_num_threads(int num_threads)`: Θέτει την τιμή της παραμέτρου `num_threads` ως το πλήθος των νημάτων που θα χρησιμοποιηθούν σε επερχόμενες παράλληλες περιοχές. Εξαίρεση αποτελούν οι παράλληλες περιοχές που χρησιμοποιούν τη φράση `num_threads` η οποία υπερισχύει.
- `int omp_get_num_threads(void)`: Επιστρέφει το πλήθος των νημάτων που συνιστούν την τρέχουσα ομάδα.
- `int omp_get_thread_num(void)`: Επιστρέφει το αριθμητικό αναγνωριστικό του καλούντος νήματος μέσα στο πλαίσιο της τρέχουσας ομάδας.
- `double omp_get_wtime(void)`: Επιστρέφει το χρόνο (wall clock) σε δευτερόλεπτα που παρήλθε μετά από μία δεδομένη αλλά ταυτόχρονα αυθαίρετη στιγμή στο παρελθόν.

2.3.9 Μεταβλητές περιβάλλοντος

Η μεταβλητή περιβάλλοντος `OMP_NUM_THREADS` μπορεί να χρησιμοποιηθεί για τον ορισμό ενός προκαθορισμένου πλήθους νημάτων που θα συμμετέχουν στις παράλληλες περιοχές του προγράμματος. Η τελική απόφαση για το πλήθος των νημάτων που θα χρησιμοποιηθούν σε μία παράλληλη περιοχή καθορίζεται σε φθίνουσα προτεραιότητα από:

1. Τη μεταβλητή περιβάλλοντος `OMP_NUM_THREADS`
2. Τη ρουτίνα βιβλιοθήκης χρόνου εκτέλεσης `omp_set_num_threads`
3. Τη φράση `num_threads` της οδηγίας `parallel`

2.4 Μεταφραστές OpenMP

Η διεπαφή που ορίζεται από το πρότυπο του OpenMP υλοποιείται από διάφορους εμπορικούς και ερευνητικούς μεταφραστές. Προμηθευτές μεταφραστών για C/C++ που υποστηρίζουν το OpenMP είναι οι εξής [5, 6]:

- AMD:
  - Ο AOMP είναι βασισμένος στον LLVM/Clang και υποστηρίζει την εκτέλεση κώδικα σε πολλαπλές κάρτες γραφικών/επιταχυντές.
  - Ο AOCC είναι επίσης βασισμένος στον clang/LLVM και υποστηρίζει πλήρως το OpenMP 4.5 και μερικώς το OpenMP 5.0.
- ARM: Ο μεταφραστής της ARM παρέχει πλήρη υποστήριξη για το OpenMP 3.1 και υποστηρίζει το OpenMP 4.0/4.5 χωρίς τη δυνατότητα εκτέλεσης κώδικα σε συσκευές η οποία βρίσκεται υπό ανάπτυξη.
- Barcelona Supercomputing Center: Ο Mercurium είναι ένας ερευνητικός μεταφραστής πηγαίου σε πηγαίο κώδικα (source-to-source) ο οποίος υποστηρίζει σχεδόν πλήρως το OpenMP 3.1 καθώς και χαρακτηριστικά νεότερων εκδόσεων που σχετίζονται με τον μηχανισμό tasking του OpenMP.
- Fujitsu: Οι μεταφραστές για τον υπερυπολογιστή PRIMEHPC FX100 της Fujitsu υποστηρίζουν το OpenMP 3.1.
- GNU: Ο GCC υποστηρίζει πλήρως το OpenMP 4.5 (έκδοση 6) και μερικώς το OpenMP 5.0. Επίσης, σε συστήματα Linux υποστηρίζει την εκτέλεση κώδικα σε κάρτες γραφικών NVIDIA (nvptx) και τις κάρτες Fiji και Vega της AMD Radeon (GCN).
- HPE: Το Cray Compiling Environment (CCE) παρέχει πλήρη υποστήριξη για το OpenMP 4.5 και μερική υποστήριξη για το OpenMP 5.0.

- IBM: Ο μεταφραστής XL C/C++ για Linux υποστηρίζει πλήρως το OpenMP 4.5.

- Intel: Οι μεταφραστές της Intel υποστηρίζουν πλήρως το OpenMP 4.5 και μερικώς το OpenMP 5.0.

- LLNL Rose Research Compiler: Ο ROSE είναι ένας ερευνητικός μετφραστής πηγαίου σε πηγαίο κώδικα που υποστηρίζει το OpenMP 3.0 και κάποια χαρακτηριστικά του OpenMP 4.0 που σχετίζονται με την εκτέλεση κώδικα σε κάρτες γραφικών/επιταχυντές της NVIDIA.

- LLVM: Ο Clang παρέχει υποστήριξη για το OpenMP 4.5 με περιορισμένη υποστήριξη για την εκτέλεση κώδικα σε συσκευές. Επίσης, υποστηρίζεται μεγάλο μέρος του OpenMP 5.0 και μικρό μέρος του OpenMP 5.1.

- Siemens: Ο Sourcery CodeBench (AMD GCN) Lite για συστήματα x86_64 GNU/Linux είναι βασισμένος στον GCC, παρέχει πλήρη υποστήριξη για το OpenMP 4.5, μερική υποστήριξη για το OpenMP 5.0. και επιτρέπει την εκτέλεση κώδικα σε κάρτες γραφικών AMD Radeon (GCN) όπως οι Fiji, gfx900 Vega 10 και gfx906 Vega 20.

- NVIDIA HPC Compiler: Οι μεταφραστές NVIDIA HPC παρέχουν πλήρη υποστήριξη του OpenMP 3.1 και μερική υποστήριξη του OpenMP 5.0 για συστήματα Linux/x86-64, Linux/OpenPOWER, Linux/Arm. Επίσης, σε κάρτες γραφικών της NVIDIA υποστηρίζεται μερικώς το OpenMP 5.0.

- OMPi Research Compiler: Ο OMPi είναι ερευνητικός μεταφραστής ανοιχτού κώδικα για τη γλώσσα C που επιτρέπει τη χρήση διάφορων βιβλιοθηκών νημάτων και συσκευών. Υποστηρίζει πλήρως την εκτέλεση κώδικα σε συσκευές βάσει του OpenMP 4.5, καθώς επίσης και υποσύνολο των λειτουργιών του OpenMP 5.0.

- OpenUH Research Compiler: Ο OpenUH είναι ερευνητικός μεταφραστής και υποστηρίζει πλήρως το OpenMP 2.5 και σχεδόν πλήρως το OpenMP 3.0 σε συστήματα Linux.

- Oracle: Οι μεταφραστές του Oracle Developer Studio υποστηρίζουν το OpenMP 4.0.

- PGI: Οι μεταφραστές NVidia HPC υποστηρίζουν μερικώς το OpenMP 5.0.
- Texas Instruments:
  - Ο μεταφραστής TI cl6x υποστηρίζει το OpenMP 3.0. (C66x).
  - Το βασισμένο στον GCC Linaro toolchain υποστηρίζει το OpenMP 4.5 (Cortex-A15).
  - Ο μεταφραστής TI clacc υποστηρίζει το OpenMP 3.0 και εκτέλεση κώδικα σε συσκευές βάσει του OpenMP 4.0 (Cortex-A15+C66x-DSP).

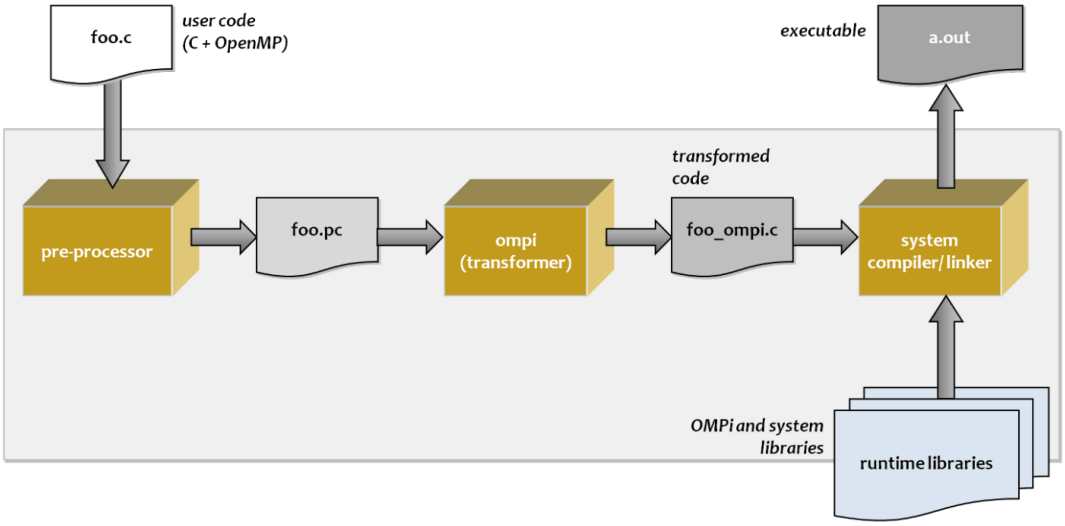
  Να σημειωθεί ότι η παρεχόμενη υποτήριξη αφορά προϊόντα system on a chip (SoC) της Texas Instruments.

2.4.1 Ο μεταφραστής OMPi

Ο μεταφραστής OMPi αναπτύσσεται από την Ομάδα Παράλληλης Επεξεργασίας του Πανεπιστημίου Ιωαννίνων από το 2001 και είναι ένας μεταφραστής για τη γλώσσα C που υποστηρίζει τη διεπαφή προγραμματισμού εφαρμογών OpenMP. Ο OMPi είναι οργανωμένος σε δύο βασικά μέρη, τον μεταφραστή (compiler) και το σύστημα χρόνου εκτέλεσης (runtime).

Διαδικασία μετάφρασης προγραμμάτων χρήστη

Η πλήρης διαδικασία μετάφρασης που ακολουθεί ο OMPi φαίνεται στο Σχήμα 2.1. Το πρόγραμμα εισόδου είναι πηγαίος κώδικας σε γλώσσα C που περιλαμβάνει οδηγίες OpenMP. Όπως συμβαίνει με όλα τα προγράμματα C, αρχικά περνάνε από το στάδιο της προεπεξεργασίας. Η έξοδος αυτού του πρώτου σταδίου τροφοδοτείται στο τμήμα του μεταφραστή (compiler) του OMPi, ο οποίος κάνει λεκτική και συντακτική ανάλυση. Από τη συντακτική ανάλυση προκύπτει το αφηρημένο συντακτικό δέντρο (AST - Abstract Syntax Tree) το οποίο χρησιμεύει στην αντικατάσταση των οδηγιών OpenMP με κλήσεις συναρτήσεων του συστήματος χρόνου εκτέλεσης, οι οποίες υλοποιούν τις λειτουργίες που καθορίζουν οι αντίστοιχες οδηγίες. Στη συνέχεια, ο μετασχηματισμένος κώδικας δίνεται ως είσοδος σε έναν απλό μεταφραστή για γλώσσα C (π.χ. GCC) ώστε να παραχθεί το αντικείμενο πρόγραμμα. Τέλος, από τη σύνδεση (linking) του αντικείμενου προγράμματος με τη βιβλιοθήκη χρόνου εκτέλεσης του OMPi και τις βιβλιοθήκες του συστήματος, προκύπτει το τελικό εκτελέσιμο αρχείο.



Σχήμα 2.1: Η διαδικασία μετάφρασης του μεταφραστή OMPi.

Σύστημα χρόνου εκτέλεσης (runtime)

Το σύστημα χρόνου εκτέλεσης παρέχει όλες τις απαραίτητες βοηθητικές συναρτήσεις που απαιτούνται σε χρόνο εκτέλεσης. Παράδειγμα τέτοιων συναρτήσεων είναι οι συναρτήσεις που υλοποιούν αμοιβαίο αποκλεισμό και συγχρονισμό όπως κλειδαριές και κλήσεις φραγής, συναρτήσεις δημιουργίας οντοτήτων εκτέλεσης, συναρτήσεις που διαβάζουν τις μεταβλητές περιβάλλοντος κλπ.

Αποτελείται από τα τμήματα host και devices. Το πρώτο τμήμα αφορά την υποστήριξη εκτέλεσης κώδικα στο κύριο σύστημα (host), δηλαδή εκεί όπου ξεκίνησε η εκτέλεση του προγράμματος, ενώ το δεύτερο τμήμα αφορά την εκτέλεση κώδικα σε συσκευές όπως κάρτες γραφικών γενικού σκοπού (devices). Το τμήμα host αποτελείται με τη σειρά του από το ORT (**O**MPi **R**un**T**ime) και τις βιβλιοθήκες οντοτήτων εκτέλεσης (EELIBs - **E**xecution **E**ntity **LIB**raries).

Η οντότητα εκτέλεσης είναι μία αφηρημένη έννοια που χρησιμοποιείται για την απόκρυψη των λεπτομερειών υλοποίησης των EELIBs ώστε αυτά να είναι εντελώς ανεξάρτητα από τον υπόλοιπο κώδικα του OMPi. Το ORT διαχειρίζεται και συντονίζει τις οντότητες εκτέλεσης σε υψηλό επίπεδο, δηλαδή μέσω μίας διεπαφής προγραμματισμού εφαρμογών (API) που παρέχουν όλα τα EELIBs. Από τη μεριά τους, τα EELIBs διαχειρίζονται τις οντότητες εκτέλεσης σε χαμηλό επίπεδο ανάλογα με τις λεπτομέρειες υλοποίησης της εκάστοτε βιβλιοθήκης. Για να γίνει πιο κατανοητό, το ORT με μία κλήση της συνάρτησης `othr_request()` ζητάει από

το EELIB να δημιουργήσει ένα συγκεκριμένο πλήθος οντοτήτων εκτέλεσης, χωρίς όμως να γνωρίζει λεπτομέρειες για τις οντότητες αυτές (π.χ. αν είναι νήματα POSIX, νήματα PSTHREADS ή διεργασίες). Αντίθετα, το EELIB γνωρίζει όλες τις λεπτομέρειες υλοποίησης και πραγματοποιεί τις κατάλληλες κλήσεις για τη δημιουργία των οντοτήτων (π.χ. `pthread_create()` στην περίπτωση των νημάτων POSIX).

Η ιδιαίτερη αρχιτεκτονική σχεδίαση του συστήματος χρόνου εκτέλεσης και ο τρόπος αλληλεπίδρασής του με τα EELIBS καθιστούν δυνατή τη χρήση πολλών διαφορετικών τύπων οντοτήτων εκτέλεσης. Επίσης σημαντικό χαρακτηριστικό είναι ότι ο οποιοσδήποτε μπορεί να αναπτύξει και να χρησιμοποιήσει τη δική του EELIB χωρίς να ασχοληθεί με τον κώδικα του OMPi.

ΚΕΦΑΛΑΙΟ 3

Τοπολογία Συστήματος

Η δημιουργία φορητού (portable) λογισμικού, δηλαδή λογισμικού το οποίο μπορεί να χρησιμοποιηθεί σε συστήματα διαφορετικά μεταξύ τους, είναι δυνατή με τη χρήση αφαιρέσεων (abstractions) για το υποκείμενο σύστημα. Οι αφαιρέσεις αυτές αποκρύπτουν λεπτομέρειες της οργάνωσης του υλικού του εκάστοτε συστήματος, με την έννοια ότι από τη σκοπιά του προγραμματιστή δεν θα πρέπει να μας ενδιαφέρουν ζητήματα χαμηλού επιπέδου, όπως για παράδειγμα η οργάνωση της μνήμης από φυσική άποψη (κοινόχρηστη ή κατανεμημένη) που αναφέραμε στην Υποενότητα 1.2.2, αλλά το τι εικόνα έχουμε για αυτή.

Η αφαιρετική όμως εικόνα που έχει ο προγραμματιστής για το σύστημα που καλείται να προγραμματίσει δεν του επιτρέπει να εκμεταλλευτεί στο μέγιστο τις δυνατότητές του και άρα να επιτύχει την καλύτερη δυνατή επίδοση. Για παράδειγμα, σε ένα σύστημα δύο κόμβων με κατανεμημένη φυσική οργάνωση μνήμης που όμως παρέχει κοινόχρηστο χώρο διευθύνσεων, ναι μεν ο ένας κόμβος μπορεί να προσβεί οποιαδήποτε θέση μνήμης στον άλλο κόμβο, παρόλα αυτά το κόστος της απομακρυσμένης πρόσβασης θα είναι πολύ μεγαλύτερο από αυτό της πρόσβασης στην τοπική μνήμη.

Στον προγραμματισμό παράλληλων συστημάτων που στόχος μας είναι η επίτευξη όσο το δυνατόν καλύτερων επιδόσεων, συχνά καταφεύγουμε στην εκμετάλλευση πληροφοριών που σχετίζονται με την τοπολογία. Για το λόγο αυτό γίνονται προσπάθειες δημιουργίας μεταφέρσιμου λογισμικού που όμως λαμβάνει υπόψην του την οργάνωση του υλικού σε χρόνο εκτέλεσης.

3.1 Η εξέλιξη των βασικών στοιχείων ενός συστήματος

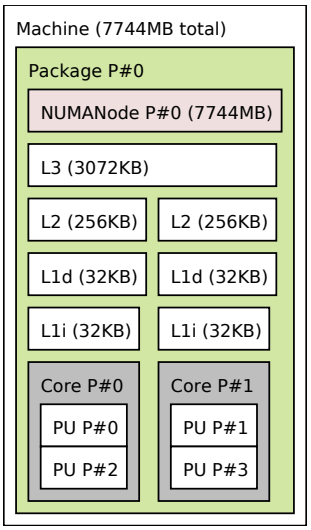
Τα πιο σημαντικά στοιχεία που αποτελούν ένα υπολογιστικό σύστημα είναι αυτά που αφορούν τα υποσυστήματα επεξεργαστή και μνήμης.

Ένα απλό σειριακό σύστημα αποτελούνταν από έναν επεξεργαστή (ΚΜΕ - Κεντρική Μονάδα Επεξεργασίας, CPU - Central Processing Unit) ο οποίος περιείχε έναν επεξεργαστικό πυρήνα (ή απλώς πυρήνας, core) και επικοινωνούσε με την κύρια μνήμη μέσω ενός διαύλου. Με την εμφάνιση των πολυπύρηνων (multicore) οργανώσεων περισσότεροι του ενός πυρήνες ενσωματώθηκαν στον ίδιο επεξεργαστή, ενώ με την πάροδο του χρόνου προστέθηκε και ιεραρχία κρυφών μνημών.

Στα πλαίσια μιας άλλης αρχιτεκτονικής βελτίωσης, με μικρή αύξηση του μεγέθους του κυκλώματος, μπόρεσαν να ενσωματωθούν μέσα σε έναν πυρήνα 2 ή περισσότερα νήματα υλικού (H/W threads, logical processors - λογικοί επεξεργαστές), ώστε να αξιοποιηθούν καλύτερα οι λειτουργικές μονάδες του επεξεργαστή, προσφέροντας με αυτό τον τρόπο τη δυνατότητα παράλληλης εκτέλεσης.

Τη σημερινή εποχή, μπορούμε να συναντήσουμε μεγάλα συστήματα που διαθέτουν πολλαπλούς επεξεργαστές, όπου κάθε επεξεργαστής είναι πολυπύρηνος, περιλαμβάνει ιεραρχία κρυφών μνημών και τοπική μνήμη, ενώ κάθε πυρήνας διαθέτει πολλαπλά H/W threads.

Στο Σχήμα 3.1 φαίνεται η οργάνωση ενός επεξεργαστή που διαθέτει 2 πυρήνες με 2 H/W threads και μία ιεραρχία κρυφών μνημών τριών επιπέδων (L1, L2, L3) στην οποία η κρυφή μνήμη του πρώτου επιπέδου διακρίνεται σε κρυφή μνήμη δεδομένων (L1d) και εντολών (L1i). Η οπτικοποίηση της τοπολογίας έγινε με το λογισμικό Portable Hardware Locality (hwloc) με το οποίο θα ασχοληθούμε περαιτέρω στην Υποενότητα 3.3.1.



Σχήμα 3.1: Η τοπολογία ενός Intel® Core™ i3-7100U.

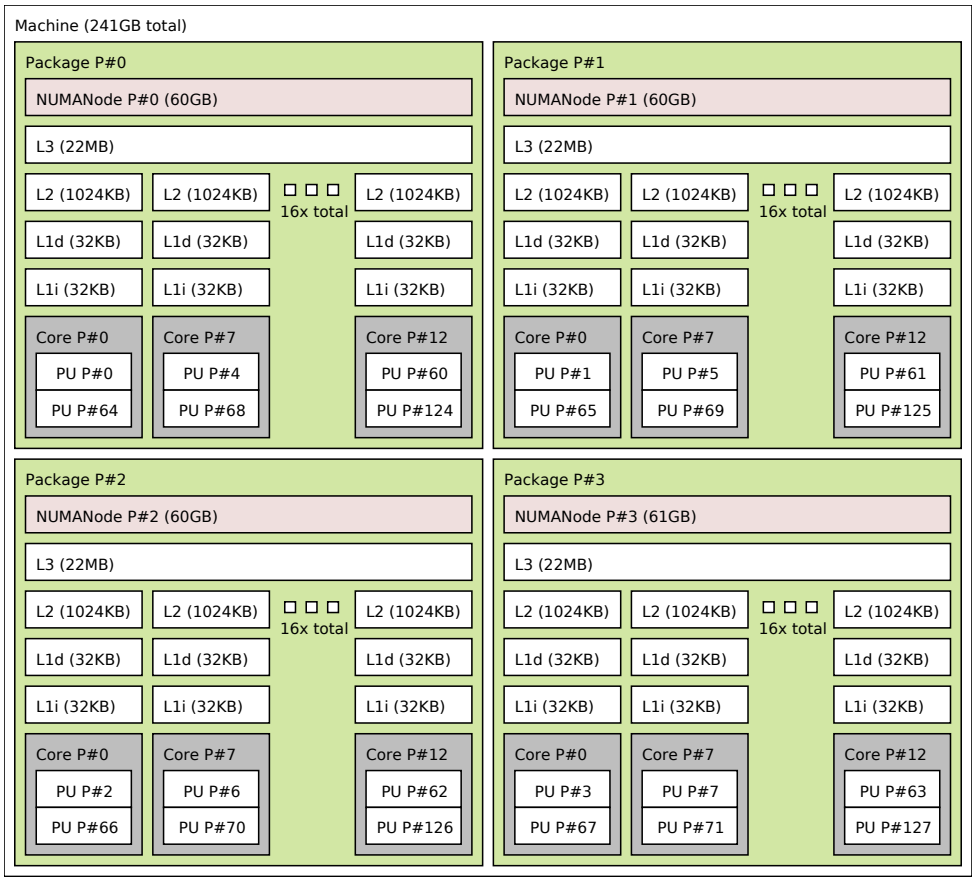
3.2 Συστήματα NUMA

Στην Υποενότητα 1.2.2 περιγράφηκε η οργάνωση των συστημάτων ανομοιόμορφης προσπέλασης μνήμης (NUMA), τα οποία αποτελούν αρχιτεκτονική εξέλιξη των συμμετρικών πολυεπεξεργαστών (SMPs) που επιτρέπει την κατασκευή μεγαλύτερων παράλληλων συστημάτων.

Η πιο συχνή υλοποίηση των συστημάτων NUMA είναι ως ένα σύνολο κόμβων τύπου SMP που διαθέτουν ιεραρχία κρυφών μνημών συνδεδεμένους μεταξύ τους με ένα δίκτυο διασύνδεσης [7]. Το δίκτυο διασύνδεσης μπορεί να είναι δακτύλιος (ring), διακοπτικό δίκτυο τύπου crossbar, από σημείο σε σημείο (point-to-point) ή πλέγμα (mesh).

Η εξασφάλιση της συνοχής των κρυφών μνημών συνήθως επιτυγχάνεται με τη χρήση ενός πρωτοκόλλου εντός του κόμβου και ενός δεύτερου πρωτοκόλλου μεταξύ των κόμβων. Όπως αναφέρθηκε νωρίτερα, ένας κόμβος SMP βασίζεται σε δίαυλο, ενώ η επικοινωνία μεταξύ των κόμβων γίνεται με πιο πολύπλοκα δίκτυα διασύνδεσης. Για το λόγο αυτό, το πρωτόκολλο συνοχής εντός του κόμβου είναι τύπου παρακολούθησης (snooping protocol), ενώ αυτό μεταξύ των κόμβων βασίζεται σε καταλόγους (directory-based protocol). Φυσικά, σε διαφορετικές οργανώσεις συστημάτων NUMA μπορούμε να συναντήσουμε και άλλες στρατηγικές επίτευξης συνοχής, όπως αυτή της μη αποθήκευσης κοινόχρηστων δεδομένων στις κρυφές μνήμες.

Όπως γίνεται γρήγορα αντιληπτό, ο προγραμματισμός αυτών των συστημάτων



Σχήμα 3.2: Η τοπολογία ενός Dell PowerEdge R840 με 4 Intel® Xeon® Gold 6130.

συνήθως απαιτεί να εστιάσουμε στην τοπικότητα, δηλαδή τη χρήση πόρων που βρίσκονται στη μικρότερη δυνατή απόσταση και την μείωση της επικοινωνίας μεταξύ των κόμβων [8]. Για την επίτευξη της τοπικότητας και άρα τη συγγραφή πιο αποδοτικών προγραμμάτων, η εκμετάλλευση πληροφορίας που σχετίζεται με την τοπολογία είναι απαραίτητη.

Στο Σχήμα 3.2 φαίνεται η οργάνωση του `parade`, ενός NUMA συστήματος με τέσσερις κόμβους τύπου SMP που διαθέτει η Ομάδα Παράλληλης Επεξεργασίας του Πανεπιστημίου Ιωαννίνων και η οργάνωσή του αποτελεί την πλέον συνηθισμένη.

3.3 Βοηθητικά Εργαλεία

Για την εκμετάλλευση της τοπολογίας ενός συστήματος έχουν αναπτυχθεί διάφορα εργαλεία όπως η βιβλιοθήκη Portable Hardware Locality (hwloc) και η `libnuma`.

3.3.1 hwloc

Η βιβλιοθήκη Portable Hardware Locality παρέχει μία φορητή αφαίρεση (abstraction) της τοπολογίας που διαθέτουν υπολογιστικά συστήματα με σύγχρονες αρχιτεκτονικές. Ο κύριος λόγος χρήσης της είναι η συγκέντρωση πληροφοριών σχετικά με πολύπλοκα παράλληλα συστήματα, με στόχο την κατάλληλη και αποδοτική εκμετάλλευσή τους [9].

Μεγάλο πλεονέκτημα της hwloc αποτελεί η υποστήριξη όλων των γνωστών λειτουργικών συστημάτων, όπως για παράδειγμα Linux, Solaris, AIX, HP-UX, NetBSD, FreeBSD, Darwin / OS X και Microsoft Windows.

Τα στοιχεία που απαρτίζουν την τοπολογία

Η hwloc μπορεί να αναγνωρίσει τα εξής στοιχεία υλικού:

- Κόμβους συστημάτων NUMA (NUMA nodes)
- Υποδοχές επεξεργαστών (packages, πρώην sockets)
- Κρυφές μνήμες (caches)
- Πυρήνες (cores)
- H/W threads (PUs - Processing Units)
- Σύσκευές εισόδου-εξόδου όπως:
  - Συσκευές PCI
  - Επιταχυντές OpenCL, CUDA και Xeon Phi
  - Προσαρμογείς δικτύου (network interfaces) όπως InfiniBand

Με την ορολογία της hwloc στη διάθεση μας, ας ανατρέξουμε στο Σχήμα 3.2 που απεικονίζεται η τοπολογία του συστήματος parade. Στο σχήμα αυτό βλέπουμε 4 επεξεργαστές ο καθένας εκ των οποίων διαθέτει 16 πυρήνες και ιεραρχία κρυφών μνημών τριών επιπέδων (L1i & L1d, L2, L3). Τα επίπεδα ένα και δύο των κρυφών μνημών είναι κοινά ανά πυρήνα, ενώ το επίπεδο τρία είναι κοινό για όλους τους πυρήνες του επεξεργαστή. Επίσης, κάθε πυρήνας διαθέτει δύο H/W threads τα οποία μοιράζονται τους πόρους του πυρήνα (π.χ. κρυφές μνήμες).

Η διεπαφή προγραμματισμού εφαρμογών (API)

Η βασική διεπαφή προγραμματισμού εφαρμογών (API) της hwloc είναι διαθέσιμη μέσω του αρχείου `hwloc.h`, ενώ για την επιτυχή μετάφραση χρειάζεται το όρισμα `-lhwloc`.

Η τοπολογία του υποκείμενου συστήματος αναπαρίσταται ως μία δεντρική δομή από αντικείμενα και αποθηκεύεται σε μία μεταβλητή τύπου `hwloc_topology_t`. Στη ρίζα του δέντρου υπάρχει πάντα ένα αντικείμενο τύπου Machine που αναπαριστά ολόκληρο το υποκείμενο μηχάνημα/σύστημα, ενώ στα φύλλα του δέντρου υπάρχουν πάντα PUs ή αλλιώς H/W threads που αποτελούν το μικρότερο διαθέσιμο επεξεργαστικό στοιχείο. Καμία άλλη παραδοχή δεν είναι δυνατόν να γίνει για τη μορφή της τοπολογίας (π.χ. βάθος, τύποι αντικειμένων που την απαρτίζουν κλπ).

Για τη διαχείριση της μεταβλητής τύπου `hwloc_topology_t` χρησιμοποιούνται οι εξής βασικές συναρτήσεις των οποίων η λειτουργία είναι εύκολα κατανοητή από το όνομά τους:

- `hwloc_topology_init()`
- `hwloc_topology_load()`
- `hwloc_topology_destroy()`

Αφού αρχικοποιηθεί και φορτωθεί η τοπολογία με τις δύο πρώτες κλήσεις, μπορεί να γίνει αναδρομική διάσχιση του δέντρου για την ανακάλυψη των διαφορετικών συστατικών του στοιχείων (packages, NUMA nodes, caches, cores, PUs). Η hwloc διαθέτει πλήθος χρήσιμων συναρτήσεων που διευκολύνουν είτε την αναδρομική διάσχιση, είτε τη διάσχιση ενός μόνο επιπέδου και επιτρέπουν:

- την εύρεση του αντικειμένου στη ρίζα της τοπολογίας (`hwloc_get_root_obj()`).
- την εύρεση του πλήθους των αντικειμένων που έχουν συγκεκριμένο τύπο ή βάθος (`hwloc_get_nbobjs_by_type()` και `hwloc_get_nbobjs_by_depth()`).
- την εύρεση αντικειμένων με συγκεκριμένο τύπο ή σε συγκεκριμένο βάθος (`hwloc_get_obj_by_type()` και `hwloc_get_obj_by_depth()`).
- την εύρεση του επόμενου στη σειρά αντικειμένου με συγκεκριμένο τύπο ή βάθος (`hwloc_get_next_obj_by_type()` και `hwloc_get_next_obj_by_depth()`).

Άλλες δυνατότητες της hwloc

Πέρα από τη συγκέντρωση πληροφοριών που σχετίζονται με την τοπολογία του υποκείμενου συστήματος, η hwloc μπορεί να χρησιμοποιηθεί για την ανάθεση νημάτων (thread binding) σε επεξεργαστές και τη διαχείριση μνήμης με δυνατότητα δέσμευσης τμημάτων μνήμης σε συγκεκριμένους κόμβους NUMA (memory binding). Παρόλα αυτά, η χρήση των διαθέσιμων λειτουργιών υπόκειται στις δυνατότητες του εκάστοτε λειτουργικού συστήματος.

3.3.2 libnuma

Η libnuma είναι μία βιβλιοθήκη που παρέχει μία διεπαφή προγραμματισμού εφαρμογών (API) για την πολιτική NUMA (NUMA policy) του Linux Kernel.

Η πολιτική NUMA του Linux αφορά τη δέμευση μνήμης σε συγκεκριμένους κόμβους NUMA ώστε να μπορεί το πρόγραμμα που εκτελείται να την προσβεί όσο το δυνατόν πιο γρήγορα [10]. Η προκαθορισμένη πολιτική του Linux είναι η δέσμευση μνήμης να πραγματοποιείται στον τοπικό κόμβο του νήματος που την προκάλεσε [11, `mm/mempolicy.c`]. Ένα νήμα προκαλεί δέσμευση μνήμης όταν γράψει για πρώτη φορά σε μία εικονική διεύθυνση[1] που ανήκει σε σελίδα[2] (page) της εικονικής μνήμης η οποία δεν έχει αντιστοιχιστεί σε σελίδα φυσικής μνήμης. Αυτή η πολιτική είναι γνωστή και ως first-touch.

Η πολιτική first-touch είναι αποδοτική όταν οι προσβάσεις πραγματοποιούνται από νήματα που εκτελούνται στον κόμβο όπου έχει δεσμευτεί η μνήμη. Υπάρχουν όμως περιπτώσεις που μπορεί να οδηγηθούμε σε κακές επιδόσεις. Μία τέτοια περίπτωση είναι όταν μία πολυνηματική εφαρμογή εκτελείται σε πολλαπλούς κόμβους αλλά μόνο ένα νήμα κάνει τις αρχικοποιήσεις της μνήμης. Κάτι τέτοιο θα οδηγήσει στη δέσμευση της μνήμης μόνο σε ένα κόμβο και άρα τα νήματα που είναι τοποθετημένα στους υπόλοιπους κόμβους θα αναγκαστούν να πραγματοποιούν απομακρυσμένες προσβάσεις με αποτέλεσμα να παρατηρηθούν αυξημένες καθυστερήσεις. Τη λύση σε αυτό το πρόβλημα μπορεί να δώσει η libnuma, καθώς επιτρέπει στο

[1] Το λειτουργικό σύστημα χρησιμοποιεί έναν μηχανισμό γνωστό ως εικονική μνήμη (virtual memory) ώστε να μπορεί να παρέχει σε κάθε διεργασία ένα συνεχόμενο χώρο (εικονικών) διευθύνσεων ικανοποιητικού μεγέθους (π.χ. 4 Gigabytes για συστήματα 32 bit), ακόμα και αν αυτός ο χώρος δεν είναι διαθέσιμος στη φυσική μνήμη (π.χ. 8 Gigabytes).

[2] Η εικονική και φυσική μνήμη είναι χωρισμένες σε τμήματα, συνήθως μεγέθους 4,096 Bytes, που ονομάζονται σελίδες (pages).

νήμα που κάνει όλες τις αρχικοποιήσεις να να δεσμεύσει μνήμη σε οποιοδήποτε κόμβο και συνεπώς να κατανείμει τη μνήμη λαμβάνοντας υπόψην από ποιό κόμβο πρόκειται να προέλθουν οι περισσότερες προσπελάσεις.

Η προκαθορισμένη πολιτική μπορεί να αλλάξει για συγκεκριμένα τμήματα μνήμης ή σε επίπεδο νήματος/διεργασίας. Στην πρώτη περίπτωση η πολιτική είναι ίδια για όλα τα νήματα/διεργασίες. Στη δεύτερη περίπτωση, η αλλαγή επηρεάζει μόνο το τρέχον νήμα/διεργασία και η νέα πολτική κληρονομείται σε νήματα/διεργασίες-παιδιά που ίσως δημιουργηθούν μεταγενέστερα. Οι διαθέσιμες πολιτικές είναι οι εξής:

- page interleaving: Οι σελίδες δεσμεύονται κυκλικά (round-robin) σε όλους τους διαθέσιμους κόμβους (ή ένα υποσύνολό τους).
- preferred node allocation: Οι σελίδες δεσμεύονται στον επιθυμητό κόμβο.
- local allocation: Οι σελίδες δεσμεύονται στον τοπικό κόμβο όπου εκτελείται η διεργασία ή το νήμα.
- allocation only on specific nodes: Οι σελίδες δεσμεύονται μόνο σε συγκεκριμένους κόμβους.

Είναι σημαντικό να διευκρινιστεί ότι η διαχείριση μνήμης στη libnuma γίνεται σε επίπεδο σελίδων. Για παράδειγμα, όταν ο χρήστης αιτείται δέσμευση μνήμης συγκεκριμένου μεγέθους, αυτό το μέγεθος στρογγυλοποιείται προς τα πάνω ώστε να είναι πολλαπλάσιο του μεγέθους σελίδας (page size, συνήθως 4,096 Bytes). Επίσης, η ανομοιόμορφη προσπέλαση μνήμης που προκύπτει λόγω της ύπαρξης ιεραρχίας κρυφών μνημών αγνοείται κατά τη διαδικασία ορισμού των κόμβων NUMA [12].

Η διεπαφή προγραμματισμού εφαρμογών (API)

Η βασική διεπαφή προγραμματισμού εφαρμογών (API) της libnuma είναι διαθέσιμη μέσω του αρχείου `numa.h`, ενώ για την επιτυχή μετάφραση χρειάζεται το όρισμα `-lnuma`.

Πριν οποιαδήποτε κλήση συνάρτησης που παρέχει η libnuma, είναι υποχρεωτικό να κληθεί η συνάρτηση `numa_available()` και σε περίπτωση που επιστραφεί η τιμή -1 οι συναρτήσεις της libnuma είναι απροσδιόριστες (undefined).

Οι ακόλουθες συναρτήσεις δεσμεύουν μνήμη χρησιμοποιώντας κάποια από τις διαθέσιμες πολιτικές, χωρίς όμως να αλλάξουν την προκαθορισμένη πολιτική:

- `numa_alloc_onnode()`: Δεσμεύει μνήμη σε συγκεκριμένο κόμβο.
- `numa_alloc_local()`: Δεσμεύει μνήμη στον τοπικό κόμβο.
- `numa_alloc_interleaved()` και `numa_alloc_interleaved_subset()`: Δεσμεύουν μνήμη κυκλικά σε όλους τους κόμβους ή σε ένα υποσύνολό τους αντίστοιχα.
- `numa_alloc()`: Δεσμεύει μνήμη βάσει της τρέχουσας πολιτικής.

Η απελευθέρωση της μνήμης που δεσμεύτηκε με τις παραπάνω συναρτήσεις γίνεται μέσω της συνάρτησης `numa_free()`, ενώ η συνάρτηση `numa_realloc()` επιτρέπει την αλλαγή του μεγέθους της ήδη δεσμευμένης μνήμης.

Συναρτήσεις αλλαγής της προκαθορισμένης πολιτικής του τρέχοντος νήματος ή διεργασίας:

- `numa_set_localalloc()`: Αλλάζει την πολιτική σε local allocation.
- `numa_set_preferred()`: Αλλάζει την πολιτική σε preferred node allocation και θέτει ως προτιμητέο κόμβο αυτόν που περνιέται ως παράμετρος.
- `numa_set_interleave_mask()`: Αλλάζει την πολιτική σε page interleaving και η δέσμευση πραγματοποιείται κυκλικά στους κόμβους που καθορίζονται μέσω της μάσκας που περνιέται ως παράμετρος.
- `numa_set_membind()`: Αλλάζει την πολιτική ώστε η δέσμευση μνήμης να πραγματοποιείται μόνο από τους κόμβους που καθορίζονται μέσω της μάσκας που περνιέται ως παράμετρος.

Άλλες χρήσιμες συναρτήσεις:

- `numa_max_node()`: Επιστρέφει το μέγιστο αναγνωριστικό κόμβου που είναι διαθέσιμο στο υποκείμενο σύστημα.
- `numa_num_configured_nodes()`: Επιστρέφει το πλήθος των κόμβων του υποκείμενου συστήματος συμπεριλαμβανομένων και των απενεργοποιημένων (disabled) κόμβων.
- `numa_get_mems_allowed()`: Επιστρέφει μία μάσκα που αναπαριστά τους κόμβους στους οποίους η τρέχουσα διεργασία μπορεί να δεσμεύσει μνήμη.

- `numa_num_configured_cpus()`: Επιστρέφει το πλήθος των επεξεργαστών του υποκείμενου συστήματος συμπεριλαμβανομένων και όσων είναι απενεργοποιημένοι (disabled).
- `numa_node_size()`: Επιστρέφει το μέγεθος της μνήμης ενός κόμβου και προεραιτικά (μέσω παραμέτρου) το μέγεθος της διαθέσιμης προς δέσμευση μνήμης.
- `numa_pagesize()`: Επιστρέφει το μέγεθος της σελίδας σε bytes.
- `numa_node_of_cpu()`: Επιστρέφει το αναγνωριστικό του κόμβου στον οποίο ανήκει ένας επεξεργαστής[3].

Άλλες δυνατότητες της libnuma

Η libnuma υποστηρίζει μεταξύ άλλων, λειτουργίες όπως η αλλαγή της πολιτικής ήδη δεσμευμένης μνήμης, περιορισμού της εκτέλεσης ενός νήματος/διεργασίας σε συγκεκριμένο σύνολο κόμβων ή επεξεργαστών, εύρεσης της απόστασης μεταξύ δύο κόμβων, μετακίνησης σελίδων μεταξύ κόμβων, καθώς και πληθώρα συναρτήσεων διαχείρισης μασκών που χρησιμοποιούνται για την αναπαράσταση συνόλων από κόμβους ή επεξεργαστές.

[3]Οτιδήποτε ορίζει το λειτουργικό σύστημα ως επεξεργαστή. Στην περίπτωση του Linux Kernel, οποιοδήποτε επεξεργαστικό στοιχείο που μπορεί να εκτελέσει μία διεργασία, όπως για παράδειγμα ένα H/W thread.

3.4 Χρήση τοπολογίας στο OpenMP

Στην έκδοση 4.0 του OpenMP προστέθηκε η δυνατότητα ανάθεσης (binding) των νημάτων σε σύνολα από συγκεκριμένους επεξεργαστές, τα λεγόμενα OpenMP places. Η ανάθεση αυτή γίνεται βάσει των διάφορων διαθέσιμων πολιτικών που περιγράφονται στις προδιαγραφές του OpenMP και είναι γνωστές ως processor binding policies. Με αυτό τον τρόπο το OpenMP δίνει τη δυνατότητα εκμετάλλευσης της τοπολογίας του υποκείμενου συστήματος ώστε να μπορούν να συγγραφούν πιο αποδοτικά παράλληλα προγράμματα.

Σε αυτό το σημείο χρειάζεται να διευκρινιστεί ότι επεξεργαστής (processor) είναι μία αυθαίρετη έννοια που προσδιορίζεται από την εκάστοτε υλοποίηση του OpenMP και αφορά μονάδα του υλικού στην οποία μπορούν να εκτελεστούν ένα ή περισσότερα νήματα. Τυπικά, ένας επεξεργαστής αντιστοιχεί σε ένα H/W thread.

Κατά το binding η εκτέλεση των νημάτων περιορίζεται σε ένα υποσύνολο των διαθέσιμων επεξεργαστών και μπορεί να χρησιμοποιηθεί για την επίτευξη καλύτερων επιδόσεων. Καλύτερες επιδόσεις μπορούμε να πετύχουμε, για παράδειγμα, περιορίζοντας τις μετακινήσεις των νημάτων μεταξύ διαφορετικών επεξεργαστών και άρα μειώνοντας[4] το πλήθος χρονοβόρων διαδικασιών όπως η διαχείριση αστοχιών κρυφής μνήμης (cache misses). Επίσης, το binding μπορεί να χρησιμοποιηθεί σε πολύπλοκα συστήματα (π.χ. NUMA) για την καλύτερη κατανομή των νημάτων ανάλογα την παράλληλη εφαρμογή.

Υποθέτοντας ένα σύστημα NUMA, αν η παράλληλη εφαρμογή απαιτεί συνεργασία μεταξύ των νημάτων για την εκτέλεση μίας εργασίας, τότε συμφέρει τα νήματα να τοποθετηθούν κοντά[5] το ένα στο άλλο ώστε να μπορούν να ανταλλάσσουν πιο γρήγορα δεδομένα μέσω της κρυφής και τοπικής μνήμης. Αν τα νήματα τοποθετηθούν σε διαφορετικούς κόμβους NUMA, το επικοινωνιακό φορτίο στο δίκτυο διασύνδεσης θα είναι αυξημένο λόγω του συντονισμού, συγχρονισμού και της μεταξύ τους επικοινωνίας, καθώς και λόγω της κίνησης που δημιουργεί το πρωτοκόλλο συνοχής των κρυφών μνημών. Σε αντίθετη περίπτωση, αν οι εργασίες που ανατίθενται στα νήματα είναι ανεξάρτητες μεταξύ τους, τότε βολεύει η διασπορά των νημάτων στους κόμβους ώστε να μπορεί το κάθε νήμα να εκμεταλλευτεί τους πόρους ολόκληρου κόμβου.

[4] Υπό προϋποθέσεις, όπως για παράδειγμα, κατά τη διάρκεια διακοπής (interupt) να μην εκτελεστεί στον ίδιο επεξεργαστή άλλο νήμα που θα γεμίσει την κρυφή μνήμη με δικά του δεδομένα.

[5] Βάσει της τοπολογίας του συστήματος.

3.4.1 OpenMP Places

Όπως αναφέρθηκε ήδη, τα OpenMP places, στα οποία από εδώ και στο εξής θα αναφερόμαστε απλά ως places, είναι σύνολα επεξεργαστών τα οποία χρησιμοποιούνται για την ανάθεση των νημάτων OpenMP σε αυτά βάσει των processor binding policies.

Κάθε place είναι ένα υποσύνολο των διαθέσιμων επεξεργαστών του υποκείμενου συστήματος και κάθε επεξεργαστής αναπαρίσταται μέσω ενός μοναδικού μη αρνητικού ακέραιου αριθμητικού αναγνωριστικού, η σημασία του οποίου καθορίζεται από την εκάστοτε υλοποίηση του OpenMP.

Όλα τα places μαζί αποτελούν το αρχικό place partition, το οποίο είναι η λίστα με τα places που είναι διαθέσιμα στο σύστημα χρόνου εκτέλεσης του OpenMP. Κάθε νήμα διαθέτει δικό του αντίγραφο του place partition και το οποίο ανάλογα την πολιτική ανάθεσης των νημάτων σε επεξεργαστές, μπορεί να είναι υποσύνολο του αρχικού place partition. Με τις πολιτικές ανάθεσης θα ασχοληθούμε στην Υποενότητα 3.4.2.

Η μεταβλητή περιβάλλοντος `OMP_PLACES` χρησιμοποιείται για την αρχικοποίηση του αρχικού place partition. Οι έγκυρες τιμές της `OMP_PLACES` έχουν δύο πιθανές μορφές:

- Αφηρημένο όνομα (abstract name): Περιγράφει με αφηρημένο τρόπο ένα σύνολο places (π.χ. `cores`).
- Ρητή λίστα (explicit list): Περιγράφει κάθε place με ρητό τρόπο χρησιμοποιώντας μη αρνητικούς ακέραιους αριθμούς.

Η ρητή λίστα, μπορεί να οριστεί ως ένα σύνολο ενός ή περισσότερων places χωρισμένων μεταξύ τους με κόμμα. Κάθε place περιγράφεται είτε ως ένας αριθμός, είτε ως ένα σύνολο αριθμών που περικλείονται από άγκιστρα.

Για τον ορισμό των places μπορούν επίσης να χρησιμοποιηθούν διαστήματα (intervals) μέσω του συμβολισμού `<lower-bound>:<length>:<stride>` για να περιγράψουν την εξής ακολουθία αριθμών: `<lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> + (<length> - 1) * <stride>`.

Αν η τιμή του `<stride>` παραλείπεται, τότε θεωρείται ίση με τη μονάδα. Τα διαστήματα είναι δυνατό να χρησιμοποιηθούν και για την περιγραφή ακολουθιών από places. Ο τελεστής "`!`" επιτρέπει την εξαίρεση του αριθμού/place που ακολουθεί αμέσως μετά.

Εναλλακτικά, τα ακόλουθα αφηρημένα ονόματα μπορούν να χρησιμοποιηθούν για την περιγραφή των places:

- `threads`: Κάθε place αντιστοιχεί σε ένα H/W thread.
- `cores`: Κάθε place αντιστοιχεί σε έναν πυρήνα αποτελούμενος από ένα ή περισσότερα H/W threads.
- `ll_caches`: Κάθε place αντιστοιχεί σε ένα σύνολο πυρήνων που μοιράζονται το τελευταίο επίπεδο κρυφής μνήμης (OpenMP 5.1).
- `numa_domains`: Κάθε place αντιστοιχεί σε ένα σύνολο πυρήνων των οποίων η πιο κοντινή μνήμη σε αυτούς:
  - είναι η ίδια μνήμη και
  - σε παρόμοια απόσταση από τους πυρήνες (OpenMP 5.1).
- `sockets`: Κάθε place αντιστοιχεί σε μία υποδοχή ολοκληρωμένου κυκλώματος επεξεργαστή (socket).

Υπάρχει η δυνατότητα τα αφηρημένα ονόματα να ακολουθούνται από ένα ακέραιο νούμερο μέσα σε παρενθέσεις για τον προσδιορισμό του πλήθους των places που θα αποτελέσουν το αρχικό place partition (π.χ. `threads(8)`). Σε περίπτωση που ζητηθούν λιγότερα places σε σχέση με τα διαθέσιμα, η υλοποίηση του OpenMP αποφασίζει ποιά places θα περιληφθούν στο place partition. Αν ζητηθούν περισσότερα places, η υλοποίηση του OpenMP καθορίζει το μέγεθος του place partition, δηλαδή το πλήθος των places που το απαρτίζουν.

Η υλοποίηση του OpenMP καθορίζει επίσης την ακριβή σημασία των αφηρημένων ονομάτων, ενώ δίνεται η δυνατότητα να ορίσει και επιπλέον ονόματα.

Οι παρακάτω εντολές `export`[6] αναθέτουν τιμή στη μεταβλητή `OMP_PLACES`, ορίζοντας και οι πέντε ακριβώς τα ίδια τέσσερα places:

- `export OMP_PLACES=threads`
- `export OMP_PLACES="threads(4)"`
- `export OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"`
- `export OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"`

[6] Σύνταξη συμβατή με το Bourne Again SHell (BASH).

- `export OMP_PLACES="{0:4}:4:4"`

Για τους συντακτικούς κανόνες βάσει των οποίων προκύπτουν όλες οι έγκυρες τιμές της μεταβλητής `OMP_PLACES` ανατρέξτε στο Παράρτημα Β.

3.4.2 OpenMP Processor Binding Policies

Η πολιτική ανάθεσης των νημάτων OpenMP σε places, καθορίζεται μέσω της μεταβλητής περιβάλλοντος `OMP_PROC_BIND`, ενώ μπορεί να παρακαμφθεί με χρήση της φράσης `proc_bind` που είδαμε στην Υποενότητα 2.3.2.

Η σύνταξη της μεταβλητής περιβάλλοντος OMP_PROC_BIND

Οι δυνατές τιμές της `OMP_PROC_BIND` είναι `true`, `false`, ή λίστα από τις τιμές `master`, `primary`, `close`, `spread` διαχωρισμένων με κόμμα. Κάθε στοιχείο της λίστας καθορίζει την πολιτική που θα χρησιμοποιηθεί στην εμφωλευμένη παράλληλη περιοχή του αντίστοιχου επιπέδου/βάθους. Για παράδειγμα, η τιμή `"spread,close,master"` καθορίζει την πολιτική ανάθεσης στο πρώτο, δεύτερο και τρίτο επίπεδο παραλληλίας σε `spread`, `close` και `master` αντίστοιχα.

Σε περίπτωση που η τιμή της `OMP_PROC_BIND` είναι `false`, τα νήματα ενδέχεται να μετακινηθούν ανάμεσα στα places, και οι φράσεις `proc_bind` αγνοούνται. Σε αντίθετη περίπτωση, τα νήματα δεν μετακινούνται[7] ανάμεσα στα places, ενώ το αρχικό νήμα ανατίθεται στο πρώτο place του αρχικού place partition.

Αν η τιμή της `OMP_PROC_BIND` είναι `true`, η πολιτική ανάθεσης καθορίζεται από την υλοποίηση του OpenMP.

Παραδείγματα εντολών `export` που αναθέτουν τιμή στη μεταβλητή `OMP_PROC_BIND`:

- `export OMP_PROC_BIND=false`
- `export OMP_PROC_BIND=close`
- `export OMP_PROC_BIND="spread,close"`

Περιγραφή των πολιτικών

Οι διαθέσιμες πολιτικές είναι οι `master` ή αλλιώς `primary`, η `close` και η `spread`.

[7] Συνίσταται αλλά δεν είναι απαραίτητα υποχρεωτικό.

Η πιο απλή πολιτική είναι η `master/primary` κατά την οποία όλα τα νήματα ανατίθεται στο place του πρωταρχικού νήματος (primary thread). Πρωταρχικό νήμα είναι κάθε νήμα που έχει αναγνωριστικό ίσο με μηδέν και μπορεί να είναι είτε το αρχικό νήμα (initial thread), είτε ένα οποιοδήποτε νήμα που συνάντησε την οδηγία `parallel`. Το πρωταρχικό νήμα ταυτίζεται με το νήμα-πατέρα (parent thread), δηλαδή το νήμα που δημιούργησε την παράλληλη ομάδα.

Στην πολιτική `close` τα νήματα τοποθετούνται σε places που βρίσκονται κοντά στο place του νήματος-πατέρα. Θεωρώντας ως T το πλήθος των νημάτων της παράλληλης ομάδας και ως P το πλήθος των places που ανήκουν στο place partition του νήματος-πατέρα, η τοποθέτηση γίνεται ως εξής:

- $T \leq P$: Το πρωταρχικό νήμα τοποθετείται στο place του νήματος-πατέρα. Το νήμα με το αμέσως μικρότερο αναγνωριστικό τοποθετείται στο επόμενο place του place partition, κόκ. Σε περίπτωση που συναντηθεί το τέλος του place partition, η τοποθέτηση των νημάτων να συνεχίζεται κυκλικά από το πρώτο place του.

- $T > P$: Κάθε place p θα περιέχει S_p νήματα με συνεχόμενα αναγνωριστικά, όπου $\lfloor \frac{T}{P} \rfloor \leq S_p \leq \lceil \frac{T}{P} \rceil$. Τα πρώτα S_0 νήματα (συμπεριλαμβανομένου του πρωταρχικού νήματος) τοποθετούνται στο place του νήματος-πατέρα. Τα επόμενα S_1 νήματα τοποθετούνται στο επόμενο place του place partition, με την τοποθέτηση να συνεχίζεται κυκλικά από το πρώτο place του place partition σε περίπτωση που συναντηθεί το τέλος του. Όταν το P δεν διαιρεί ακριβώς το T, ο ακριβής αριθμός των νημάτων σε κάθε place καθορίζεται από την υλοποίηση του OpenMP.

Ο σκοπός της πολιτικής `spread` είναι να κατανείμει όσο το δυνατόν πιο αραιά T νήματα στα P places του place partition του νήματος-πατέρα. Η αραιή κατανομή επιτυγχάνεται χωρίζοντας το place partition σε T τμήματα (subpartitions) όταν $T \leq P$, ή P τμήματα όταν $T > P$. Σε κάθε τμήμα τοποθετείται ένα νήμα ($T \leq P$) ή ένα σύνολο νημάτων ($T > P$). Σε αντίθεση με τις υπόλοιπες πολιτικές που δεν τροποποιούν το place partition του κάθε νήματος, στην πολιτική `spread` ως place partition του κάθε νήματος τίθεται το αντίστοιχο τμήμα στο οποίο τοποθετήθηκε. Με την τροποποίηση του place partition του κάθε νήματος, καθορίζεται το σύνολο των places το οποίο θα χρησιμοποιηθεί για την εκτέλεση τυχών εμφωλεμένων παράλληλων περιοχών. Η τοποθέτηση των νημάτων γίνεται ως εξής:

- $T \leq P$: Το place partition του νήματος-πατέρα χωρίζεται σε T τμήματα καθένα από τα οποία περιέχει $\lfloor \frac{P}{T} \rfloor$ ή $\lceil \frac{P}{T} \rceil$ συνεχόμενα places. Σε κάθε τμήμα τοποθετείται ένα μόνο νήμα για να εκτελεστεί. Το πρωταρχικό νήμα τοποθετείται στο τμήμα που περιλαμβάνει το place του νήματος-πατέρα. Το νήμα με το αμέσως μικρότερο αναγνωριστικό τοποθετείται στο πρώτο place του επόμενου τμήματος, κόκ. Αν απαιτείται, μετά την τοποθέτηση νήματος στο τελευταίο νήμα, η τοποθέτηση συνεχίζεται κυκλικά από το πρώτο τμήμα.

- $T > P$: Το place partition του νήματος-πατέρα χωρίζεται σε P τμήματα καθένα από τα οποία περιέχει ένα μόνο place. Σε κάθε τμήμα τοποθετούνται S_p νήματα με συνεχόμενα αναγνωριστικά, όπου $\lfloor \frac{T}{P} \rfloor \leq S_p \leq \lceil \frac{T}{P} \rceil$. Τα πρώτα S_0 νήματα (συμπεριλαμβανομένου του πρωταρχικού νήματος) τοποθετούνται στο τμήμα που περιέχει το place του νήματος-πατέρα. Τα επόμενα S_1 νήματα τοποθετούνται στο πρώτο place του επόμενου τμήματος, με την τοποθέτηση να συνεχίζεται κυκλικά από το πρώτο τμήμα. Όταν το P δεν διαρεί ακριβώς το T, ο ακριβής αριθμός των νημάτων σε κάθε τμήμα καθορίζεται από την υλοποίηση του OpenMP.

Η υλοποίηση του OpenMP καθορίζει τι θα συμβεί σε περίπτωση που η τοποθέτηση των νημάτων δεν υποστηρίζεται ή αποτύχει.

3.4.3 Ρουτίνες χρόνου εκτέλεσης

Στις προδιαγραφές του OpenMP έχουν προστεθεί οι εξής ρουτίνες χρόνου εκτέλεσης που μπορούν να χρησιμοποιηθούν από τα προγράμματα χρήστη:

- `omp_get_proc_bind`: Επιστρέφει την πολιτική ανάθεσης νημάτων σε επεξεργαστές που θα χρησιμοποιηθεί σε επερχόμενες παράλληλες περιοχές στις οποίες όμως η πολιτική δεν παρακάμπτεται μέσω της φράσης `proc_bind`.

- `omp_get_num_places`: Επιστρέφει το πλήθος των places στο αρχικό place partition.

- `omp_get_place_num_procs`: Επιστρέφει το πλήθος των επεξεργαστών που βρίσκονται σε συγκεκριμένο place του αρχικού place partition.

- `omp_get_place_proc_ids`: Επιστρέφει λίστα με τα αναγνωριστικά των επεξεργαστών που βρίσκονται σε συγκεκριμένο place του αρχικού place partition.

- `omp_get_place_num`: Επιστρέφει το place στο οποίο το τρέχον νήμα είναι τοποθετημένο/εκτελείται.
- `omp_get_partition_num_places`: Επιστρέφει το πλήθος των places στο place partition του τρέχοντος νήματος.
- `omp_get_partition_place_nums`: Επιστρέφει λίστα με τα αναγνωριστικά των επεξεργαστών που βρίσκονται σε όλα τα places του place partition του τρέχοντος νήματος.

3.4.4 Υλοποίηση στο μεταφραστή OMPi

Στα πλαίσια της παρούσας διπλωματικής εργασίας, προστέθηκε στον μεταφραστή OMPi πλήρης υλοποίηση όλων των λειτουργιών που περιγράφηκαν στις Υποενότητες 3.4.1 και 3.4.2.

Σύντομη περιγραφή

Κατά την εκκίνηση του συστήματος χρόνου εκτέλεσης του OMPi, μέσω της συνάρτησης `ort_initialize()` αρχικοποιούνται όλα τα απαραίτητα δεδομένα, συμπεριλαμβανομένων των OpenMP places και processor binding policies. Η αρχικοποίηση γίνεται είτε με τις προκαθορισμένες τιμές που αναφέρονται παρακάτω στις λεπτομέρειες υλοποίησης, είτε μέσω των τιμών που έχει αναθέσει ο χρήστης στις μεταβλητές περιβάλλοντος `OMP_PLACES` και `OMP_PROC_BIND`.

Για την αναγνώριση της τοπολογίας του υποκείμενου συστήματος χρησιμοποιήθηκε η βιβλιοθήκη hwloc (Υποενότητα 3.3.1) η οποία καθιστά δυνατή τη μετατροπή/αντιστοίχιση των αφηρημένων ονομάτων `threads`, `cores`, `sockets`, `ll_caches` και `numa_domains` σε σύνολα από αριθμητικά αναγνωριστικά επεξεργαστών. Υλοποιήθηκε γρήγορη διάσχιση[8] για την περίπτωση συμμετρικών τοπολογιών και αναδρομική διάσχιση σε αντίθετη περίπτωση.

Για την αποθήκευση του αρχικού place partition χρησιμοποιήθηκε μία ειδική μορφή διδιάστατου πίνακα, κατά την οποία κάθε γραμμή του αντιστοιχεί σε ένα place, με εξαίρεση την πρώτη γραμμή που διαθέτει μόνο ένα στοιχείο και στο οποίο είναι αποθηκευμένο το μέγεθος του place partition. Παρόμοια, σε κάθε γραμμή, στο

[8] Διάσχιση μόνο του επιπέδου που περιέχει τα επιθυμητά αντικείμενα. π.χ. NUMA nodes.

πρώτο στοιχείο της είναι αποθηκευμένο το πλήθος των επεξεργαστών που περιέχονται στο συγκεκριμένο place και στη συνέχεια ακολουθούν τα αναγνωριστικά τους. Η αναφορά σε ένα place γίνεται μέσω ενός ακέραιου αριθμού στο εύρος $[0, N_p)$, όπου N_p το μέγεθος του αρχικού place partition, δηλαδή το πλήθος των places που είναι διαθέσιμα στο σύστημα χρόνου εκτέλεσης.

Καθώς το αρχικό place partition είναι αδύνατο να αλλάξει σε χρόνο εκτέλεσης, κάθε νήμα χρειάζεται να αποθηκεύει μόνο δύο τιμές (`pfrom` και `pto`) για την αναπαράσταση του δικού του place partition. Οι τιμές αυτές καθορίζουν ένα υποσύνολο, όχι απαραίτητα γνήσιο, του αρχικού place partition και πιο συγκεκριμένα ένα κλειστό σύνολο συνεχόμενων places που ορίζεται ως $[pfrom, pto]$. Για παράδειγμα, για $pfrom = 2$ και $pto = 3$, το place partition του νήματος περιλαμβάνει το τρίτο[9] και τέταρτο place. Επιπλέον, κάθε νήμα αποθηκεύει το τρέχον place στο οποίο είναι τοποθετημένο στη μεταβλητή `curr_place`. Η πληροφορία του τρέχοντος place είναι απαραίτητη όχι μόνο για την αποφυγή χρονοβόρων κλήσεων συστήματος ανάθεσης σε περίπτωση που το νήμα είναι ήδη τοποθετημένο στο συγκεκριμένο place, αλλά και για να μπορεί κάθε νήμα να γνωρίζει σε ποιό place είναι τοποθετημένο το νήμα-πατέρας και συνεπώς να υπολογίσει βάσει αυτού πού πρέπει να τοποθετηθεί το ίδιο.

Για να μπορέσει ένα νήμα να προσδιορίσει το place στο οποίο πρέπει να τοποθετηθεί χρειάζεται να γνωρίζει το τρέχον place του νήματος-πατέρα, την ισχύουσα πολιτική ανάθεσης, το αριθμητικό αναγνωριστικό του μέσα στην παράλληλη ομάδα, καθώς και το place partition του.

Στον OMPi, πριν την εκτέλεση μιας παράλληλης περιοχής, το νήμα-πατέρας ζητάει από τη βιβλιοθήκη οντοτήτων εκτέλεσης το απαιτούμενο πλήθος νημάτων και η βιβλιοθήκη είτε τα δημιουργεί εκείνη τη στιγμή, είτε τα αφαιρεί από μία λίστα (pool) με ήδη δημιουργημένα νήματα τα οποία είναι έτοιμα για ανάθεση εργασίας. Αφού το νήμα-πατέρας προετοιμάσει το περιβάλλον εκτέλεσης κατάλληλα, παραδείγματος χάριν αρχικοποιώντας δομές κλήσεων φραγής, αναθέτει στα νήματα (συμπεριλαμβανομένου του εαυτού του) την εκτέλεση της συνάρτησης `ort_ee_dowork()`. Στη συνάρτηση αυτή, τα νήματα αποκτούν[10] και αρχικοποιούν κατάλληλα τη δομή ελέγχου τους (EECB - Execution Entity Control Block) και στη συνέχεια εκτελούν την εργασία που όρισε ο χρήστης.

[9] Λόγω έναρξης αρίθμησης από το μηδέν.

[10] Εάν δεν διαθέτουν ήδη.

Για την αποφυγή της σειριακής εκτέλεσης κώδικα, η ανάθεση νημάτων σε places βάσει της εκάστοτε πολιτικής γίνεται κατανεμημένα από το κάθε νήμα και όχι κεντρικά από το νήμα-πατέρα, αμέσως πριν την εκτέλεση της εργασίας που του ανέθεσε ο χρήστης. Στο τμήμα του συστήματος χρόνου εκτέλεσης, κάθε νήμα υπολογίζει το place στο οποίο θα πρέπει να τοποθετηθεί και στη συνέχεια ζητάει από τη βιβλιοθήκη εκτέλεσης οντοτήτων την πραγματοποίηση της τοποθέτησής του. Αυτό γίνεται μέσω της συνάρτησης `ee_bindme()` η οποία γνωρίζει τις λεπτομέρειες που απαιτούνται για την τοποθέτηση των νημάτων και διεξάγει τις κατάλληλες κλήσεις, όπως για παράδειγμα η κλήση `pthread_setaffinity_np()` στην περίπτωση των POSIX threads.

Λεπτομέρειες Υλοποίησης

Οι αποφάσεις που πάρθηκαν σχετικά με τις λεπτομέρειες υλοποίησης (implementation specifics) που χρειάζεται να καθοριστούν από την εκάστοτε υλοποίηση του OpenMP είναι οι εξής:

- Επεξεργαστής (processor) είναι ότι θεωρεί το λειτουργικό σύστημα ως επεξεργαστή. Τυπικά H/W thread.
- Το αριθμητικό αναγνωριστικό του κάθε επεξεργαστή είναι το φυσικό αναγνωριστικό (physical ID) που παρέχει η hwloc και ταυτίζεται με το αναγνωριστικό που χρησιμοποιεί ο Linux Kernel.
- Όταν χρησιμοποιούνται αφηρημένα ονόματα για τον καθορισμό του αρχικού place partition τότε:
  - Αν ζητηθούν λιγότερα places, έστω N, σε σχέση με το πόσα είναι διαθέσιμα, θα χρησιμοποιηθούν τα N πρώτα places.
  - Αν ζητηθούν περισσότερα places με το πόσα είναι διαθέσιμα, θα χρησιμοποιηθούν ακριβώς όσα places είναι διαθέσιμα.
- Τα αφηρημένα ονόματα `threads`, `cores`, `sockets`, `ll_caches`, `numa_domains` αντιστοιχούν στα αντικείμενα `PUs`, `cores`, `packages`, `caches`[11] και `NUMA nodes` που αναγνωρίζει η hwloc.

[11] Επιλέγεται το αντικείμενο τύπου cache του μεγαλύτερου επιπέδου. π.χ. L3

- Όταν η τιμή της `OMP_PROC_BIND` είναι true, τότε η πολιτική που ακολουθείται είναι η `close`.
- Στις πολιτικές `close` και `spread`, όταν $T > P$ και το P δεν διαιρεί ακριβώς το T, τα πρώτα $N_s = (T \mod P)$ places/τμήματα θα περιέχουν $N_t + 1$ νήματα, ενώ τα υπόλοιπα $P - N_s$ places/τμήματα θα περιέχουν N_t νήματα, όπου $N_t = \lfloor \frac{T}{P} \rfloor$.
- Τα προκαθορισμένα OpenMP places ορίζονται μέσω του αφηρημένου ονόματος `cores`.
- Η ανάθεση (binding) νημάτων σε επεξεργαστές είναι απενεργοποιημένη (false) από προεπιλογή.

ΚΕΦΑΛΑΙΟ 4

Συγχρονισμός με barriers

Όπως έχουμε ήδη αναφέρει σε προηγούμενα κεφάλαια, ο προγραμματισμός συστημάτων κοινόχρηστου χώρου διευθύνσεων βασίζεται συνήθως στη χρήση νημάτων, ο συγχρονισμός μεταξύ των οποίων είναι ιδιαίτερα σημαντικός καθώς εξασφαλίζει τη συνέπεια των δεδομένων και την ορθότητα του προγράμματος.

Μία από τις πιο διαδεδομένες μεθόδους συγχρονισμού είναι η *κλήση φραγής*, γνωστή και ως barrier. Όταν στον πηγαίο κώδικα υπάρχει μία κλήση φραγής και συναντηθεί από ένα νήμα, τότε το νήμα περιμένει σε αυτό το σημείο την άφιξη των υπόλοιπων νημάτων, πριν μπορέσουν όλα μαζί να συνεχίσουν την εκτέλεση του κώδικα που ακολουθεί την κλήση φραγής.

Οι κλήσεις φραγής είναι ιδιαίτερα χρήσιμες για τον διαχωρισμό των φάσεων ενός προγράμματος. Για παράδειγμα, αν θέλουμε να παραλληλοποιήσουμε ένα πρόγραμμα το οποίο υλοποιεί μία επαναληπτική μέθοδο (π.χ. conjugate gradient method), θα πρέπει να χρησιμοποιήσουμε κλήση φραγής ανάμεσα στις επαναλήψεις n και $n+1$ ώστε να μην προχωρήσει κάποιο νήμα στην επανάληψη $n+1$ ενώ δεν έχει ολοκληρωθεί ο υπολογισμός της επανάληψης n.

4.1 Υλοποιήσεις barrier

Παρόλο που η λογική των κλήσεων φραγής είναι αρκετά απλή, υπάρχει πληθώρα αλγορίθμων για την υλοποίησή τους.

Μία πρώτη κατηγορία κλήσεων φραγής αποτελούν οι κεντρικοποιημένες κλήσεις φραγής (centralized barriers). Κάθε νήμα ενημερώνει μία κοινόχρηστη κατάσταση,

ώστε να σημάνει την άφιξη του και έπειτα ελέγχει συνεχώς αυτή την κατάσταση μέχρι να αντιληφθεί ότι έφτασαν όλα τα νήματα. Μόλις φτάσουν όλα τα νήματα, το νήμα συνεχίζει την εκτέλεση του κώδικα που ακολουθεί μετά την κλήση φραγής. Στην πιο απλή της μορφή, η κοινόχρηστη κατάσταση μπορεί να είναι μερικές κοινόχρηστες μεταβλητές που χρησιμοποιούνται για παράδειγμα για τη μέτρηση του πλήθους των νημάτων που αφίχθησαν στον barrier.

Καθώς οι κλήσεις φραγής χρησιμοποιούνται επαναληπτικά, για παράδειγμα όταν υπάρχει ένας barrier στο τέλος ενός βρόγχου `for`, κατά τη διάρκεια μίας κλήσης τα νήματα θα πρέπει να πραγματοποιήσουν δύο ελέγχους. Έναν έλεγχο που θα εξασφαλίζει ότι όλα τα νήματα έφυγαν από την προηγούμενη κλήση φραγής και έναν έλεγχο που θα εξασφαλίζει ότι όλα τα νήματα έφτασαν στην τρέχουσα κλήση φραγής. Ο έλεγχος για το αν τα νήματα έφυγαν από την προηγούμενη κλήση φραγής είναι απαραίτητος καθώς σε αντίθετη περίπτωση μπορεί να δημιουργηθεί πρόβλημα στο διαχωρισμό δύο διαδοχικών κλήσεων φραγής από κάποιο υποσύνολο των νημάτων. Πιο συγκεκριμένα, όταν υπάρχει μία κοινόχρηστη κατάσταση, υπάρχει η πιθανότητα ένα ή περισσότερα νήματα να εγκλωβιστούν στην προηγούμενη κλήση φραγής, την ώρα που κάποια άλλα έχουν μεταβεί στην επόμενη και ως συνέπεια το πρόγραμμα να οδηγηθεί σε αδιέξοδο (deadlock).

Η λύση για την αποφυγή του διπλού ελέγχου με ταυτόχρονη αποφυγή αδιεξόδου, είναι η εισαγωγή της πληροφορίας `sense` η οποία παίρνει τιμές `true` (1) ή `false` (0). Η πληροφορία `sense` είναι κοινόχρηστη και η τιμή της αντιστρέφεται μεταξύ δύο διαδοχικών χρήσεων του barrier. Λόγω αυτής της αντιστροφής, η τιμή `sense` χρησιμοποιείται για το διαχωρισμό μεταξύ διαδοχικών χρήσεων ενός barrier. Η τεχνική αυτή είναι γνωστή ως *sense reversal*, ενώ οι κλήσεις φραγής που τη χρησιμοποιούν είναι γνωστές ως *sense reversing barriers*.

Το μειονέκτημα των κεντρικοποιημένων κλήσεων φραγής είναι οι συνεχόμενες προσβάσεις από όλα τα νήματα σε μία μόνο κοινόχρηστη τοποθεσία. Το πρόβλημα αυτό οδήγησε στην πρόταση υλοποιήσεων που χρησιμοποιούν μεθόδους οπισθοδρόμησης (backoff) όπου ουσιαστικά οι προσβάσεις γίνονται με μεγαλύτερα χρονικά διαστήματα μεταξύ τους. Παράδειγμα μιας τέτοιας μεθόδου αποτελεί η μέθοδος εκθετικής οπισθοδρόμησης (exponential backoff). Αν και με τη χρήση οπισθοδρόμησης παρατηρήθηκε ουσιαστική μείωση της κίνησης στο δίκτυο διασύνδεσης[1], δεν επιτυγ-

[1] Παρατηρήθηκε ταυτόχρονη αύξηση της καθυστέρησης του barrier, δηλαδή του χρόνου μεταξύ της τελευταίας άφιξης και της τελευταίας αναχώρησης από τον barrier.

χάνεται ικανοποιητική κλιμακωσιμότητα των κεντρικοποιημένων κλήσεων φραγής σε μεγάλα συστήματα κάποιων εκατοντάδων επεξεργαστών [13].

Τις συνθήκες υψηλού ανταγωνισμού που υπάρχουν στις κεντρικοποιημένες κλήσεις φραγής μπόρεσε να αντιμετωπίσει ο *combining tree barrier*. Η ιδέα πίσω από αυτή την υλοποίηση είναι η ύπαρξη πολλών μικρών barriers που αποτελούν τους κόμβους-φύλλα μιας δεντρικής δομής δεδομένων και στους οποίους ανατίθενται γνήσια υποσύνολα νημάτων. Κάθε κόμβος τοποθετείται σε διαφορετικά τμήματα μνήμης (memory modules), έτσι ώστε οι προσβάσεις στη μνήμη να μοιράζονται μεταξύ των κόμβων και να μην εστιάζονται σε μία μόνο τοποθεσία όπως συμβαίνει με τους κεντρικοποιημένους barriers. Το τελευταίο νήμα που θα φτάσει σε κάθε barrier-φύλλο (leaf barrier) συνεχίζει διαδίδοντας ενημερώσεις προς τη ρίζα του δέντρου. Όταν κάποιο νήμα καταφέρει να φτάσει στη ρίζα του δέντρου, αυτό σημαίνει ότι όλα τα νήματα που συμμετέχουν στον παράλληλο υπολογισμό έχουν φτάσει στον barrier και η διαδικασία απελευθέρωσής τους από αυτόν μπορεί να ξεκινήσει. Κατά τη διαδικασία απελευθέρωσης, το νήμα που έφτασε στη ρίζα του δέντρου ξεκινά να διαδίδει ενημερώσεις προς τα φύλλα ώστε να επιτραπεί στα νήματα που περιμένουν να συνεχίσουν την εκτέλεση τους.

Ένας παρόμοιος αλγόριθμος με αυτόν του combining tree barrier στον οποίο χρησιμοποιείται δεντρική δομή που έχει τη μορφή δυαδικού δέντρου αποτελεί ο *tournament barrier*. Ο συγχρονισμός μεταξύ των νημάτων πραγματοποιείται μεταξύ δύο νημάτων τη φορά, ενώ το νήμα-εκπρόσωπος που συνεχίζει στο ανώτερο επίπεδο καθορίζεται στατικά και όχι βάσει της σειράς άφιξης, αποφεύγοντας με αυτό τον τρόπο τη χρήση ατομικών εντολών που υλοποιούνται στο υλικό (π.χ. `fetch_and_add()`) και χρησιμοποιούνται από τον combining tree barrier.

Οι παραλλαγές των αλγορίθμων κλήσεων φραγής είναι πάρα πολλές και διαφέρουν άλλες περισσότερο και άλλες λιγότερο μεταξύ τους, ανάλογα με τους στόχους που θέλουν να επιτύχουν αυτοί που τους επινόησαν. Για παράδειγμα, κάποιος μπορεί να προτιμά τη χρήση ή μη ατομικών εντολών που υλοποιούνται στο υλικό (`fetch_and_add()`, `compare_and_swap()` κλπ), τον στατικό ή μη ορισμό των διευθύνσεων μνήμης των κόμβων ενός combining tree barrier, τη χρήση ή μη κλειδαριών (locks) κόκ. Οι διάφορες παραλλαγές στοχεύουν στην επίλυση ζητημάτων που σχετίζονται για παράδειγμα με τις παράλληλες εφαρμογές ή την οργάνωση των υποκείμενων συστημάτων και μπορεί να αφορούν μεταξύ άλλων το πλήθος τον απομακρυσμένων προσβάσεων μνήμης σε συστήματα NUMA, την πολυπλοκότητα χώρου που χρειά-

ζεται ο barrier (π.χ. $O(1)$ για χρήση σταθερού πλήθους κοινόχρηστων μεταβλητών ή $O(N)$ όπου N το πλήθος των νημάτων για χρήση πινάκων με κάθε στοιχείο να αντιστοιχεί σε ένα νήμα) ή το ποσό της κίνησης που δημιουργείται στο δίκτυο διασύνδεσης λόγω των πρωτοκόλλων συνοχής κρυφής μνήμης ανάλογα το είδος αυτών.

4.2 Barriers στο OpenMP

Στη διεπαφή προγραμματισμού εφαρμογών OpenMP που περιγράφηκε στο Κεφάλαιο 2, είδαμε ότι οι κλήσεις φραγής χρησιμοποιούνται ευρέως, τόσο έμμεσα (implicit barrier) στο τέλος παράλληλων περιοχών και των περιοχών διαμοιρασμού εργασίας, όσο και άμμεσα (explicit barrier) μέσω της οδηγίας `barrier`. Συνεπώς, το πόσο αποδοτική ή όχι είναι η υλοποίηση του barrier μπορεί να έχει σημαντικό αντίκτυπο στην συνολική επίδοση της παράλληλης εφαρμογής.

Ενδιαφέρον αποτελεί το γεγονός ότι ο ρόλος των κλήσεων φραγής στα πλαίσια του OpenMP είναι διευρυμένος σε σχέση με τις κλήσεις φραγής που περιγράψαμε μέχρι στιγμής. Αυτό συμβαίνει καθώς ο ρόλος τους εκτός από την επίτευξη συγχρονισμού μεταξύ των νημάτων, περιλαμβάνει και την εξασφάλιση ολοκλήρωσης των εργασιών OpenMP (Υποενότητα 2.3.6), των λεγόμενων και OpenMP tasks ή απλώς tasks, τα οποία δημιουργήθηκαν από την παράλληλη ομάδα. Ένα άλλο χαρακτηριστικό του OpenMP είναι το *cancellation*, δηλαδή η δυνατότητα ακύρωσης της εκτέλεσης μιας περιοχής OpenMP, όπως για παράδειγμα μια παράλληλη περιοχή και η μετάβαση των νημάτων στο τέλος αυτής. Σε περίπτωση που ζητηθεί η ακύρωση της εκτέλεσης μιας παράλληλης περιοχής, τα νήματα επιτρέπεται να μεταβούν στο τέλος της ακόμα και αν δεν έχουν προλάβει να αφιχθούν στην έμμεση κλήση φραγής που βρίσκεται αμέσως πριν αυτό. Οπότε, θα πρέπει οι κλήσεις φραγής που χρησιμοποιούνται στα πλαίσια του OpenMP να υποστηρίζουν την ακύρωση ενός barrier και να επιτρέπουν σε νήματα που έχουν ήδη αφιχθεί να φύγουν από τη διαδικασία αναμονής και να συνεχίσουν την εκτέλεσή τους.

Συνεπώς, οι προϋπάρχοντες αλγόριθμοι κλήσεων φραγής που στοχεύουν μόνο στο συγχρονισμό των νημάτων, είτε θα πρέπει να προσαρμοστούν κατάλληλα για την εξασφάλιση της ολοκλήρωσης των tasks και την υποστήριξη του cancellation, είτε θα πρέπει να επαναδιατυπωθούν από την αρχή για λόγους καλύτερης σχεδίασης, που εν τέλει μπορεί να επηρεάσει την απλότητα και την απόδοση της υλοποίησης.

4.3 Ο barrier του OMPi

Ο αλγόριθμος του barrier που χρησιμοποιείται στο μεταφραστή OMPi έχει προκύψει μετά από πλήθος βελτιώσεων και επανασχεδιάσεων.

Στην πράξη, ο OMPi διαθέτει τρεις διαφορετικούς αλγορίθμους barrier, καθένας από τους οποίους έχει δύο εκδόσεις. Οι εκδόσεις αυτές χρησιμοποιούνται όταν το cancellation είναι ενεργοποιημένο ή όχι. Για λόγους απλότητας και για ευκολότερη κατανόηση των αλγορίθμων, στα πλαίσια αυτής της διπλωματικής εργασίας θα ασχοληθούμε μόνο με τις εκδόσεις που δεν υποστηρίζουν το cancellation. Επιπλέον, για τους ίδιους λόγους θα παραλειφθούν κάποιες λεπτομέρειες υλοποίησης. Οι τρεις διαφορετικοί τύποι barrier είναι οι εξής:

- Parallel Barrier (PB): Χρησιμοποιείται στο τέλος παράλληλων περιοχών και εξασφαλίζει τόσο την ολοκλήρωση της εκτέλεσης των tasks, όσο και το συγχρονισμό των νημάτων.
- Default Barrier (DB): Είναι η πιο απλή μορφή barrier που χρησιμοποιείται εντός των παράλληλων περιοχών και εξασφαλίζει μόνο το συγχρονισμό των νημάτων καθώς υποθέτει ότι δεν υπάρχουν tasks. Σε περίπτωση που δημιουργηθούν tasks, τα νήματα μεταβαίνουν σε μια πιο πολύπλοκη μορφή barrier, τον task barrier.
- Task Barrier (TB): Στον barrier αυτό μεταβαίνουν τα νήματα αφού ησέλθουν πρώτα στον default barrier με την προϋπόθεση ότι δημιουργήθηκαν tasks. Εξασφαλίζει τόσο την ολοκλήρωση των tasks, όσο και τον συγχρονισμό των νημάτων.

Όλα τα δεδομένα του barrier αποθηκεύονται σε μία μεταβλητή τύπου `struct` που ονομάζεται `ort_defbar_t` και οι συναρτήσεις διαχείρισης και χρήσης του barrier είναι οι ακόλουθες:

- `ort_default_barrier_init()`: Αρχικοποιεί τον barrier.
- `ort_default_barrier_wait()`: Χρησιμοποιείται από τα νήματα για να συγχρονιστούν στον barrier.
- `ort_default_barrier_destroy()`: Καταστρέφει τον barrier.

Η ορολογία *default barrier* που χρησιμοποιείται στις παραπάνω συναρτήσεις αφορά τη χρήση του barrier που παρέχεται από τον OMPi και όχι του barrier που πιθανώς παρέχεται από κάποιο EELIB. Γι' αυτό το λόγο δεν θα πρέπει να συγχέεται με τον ένα από τους τρεις τύπους barrier που παρέχει ο OMPi.

Στη δομή ελέγχου EECB του κάθε νήματος υπάρχει ένας barrier ο οποίος όταν το νήμα δημιουργήσει μια παράλληλη περιοχή, δηλαδή γίνει νήμα-αρχηγός, θα χρησιμοποιηθεί για το συγχρονισμό των νημάτων που συμμετέχουν στην ομάδα. Η αρχικοποίηση του barrier γίνεται στη συνάρτηση `prepare_master()` η οποία χρησιμοποιείται για την προετοιμασία του νήματος-αρχηγού για την επικείμενη παράλληλη περιοχή (π.χ. καθορισμός μεγέθους ομάδας, ορισμός της εργασίας που θα εκτελεστεί).

4.3.1 Βασική αρχιτεκτονική

Η σχεδίαση των barriers του OMPi βασίζεται σε πίνακες ακεραίων σταθερού μεγέθους (`MAX_BAR_THREADS`[2]). Κάθε στοιχείο αντιστοιχεί σε ένα μόνο νήμα ώστε να μπορούν να πραγματοποιηθούν προσβάσεις χωρίς τη χρήση κλειδαριών για λόγους αμοιβαίου αποκλεισμού. Με αυτό τον τρόπο, όσο αυξάνεται το πλήθος των νημάτων, ο ανταγωνισμός μεταξύ τους παραμένει ανύπαρκτος. Βέβαια, κάθε πίνακας απαιτεί $O($`MAX_BAR_THREADS`$)$ χώρο, όπου `MAX_BAR_THREADS` το μέγιστο πλήθος των νημάτων που υποστηρίζεται και καθορίζεται κατά το χρόνο μετάφρασης.

Μόλις ένα νήμα φτάσει στον barrier γράφει μία τιμή, έστω 1, στο αντίστοιχο στοιχείο του πίνακα για να σηματοδοτήσει την άφιξή του. Στη συνέχεια, ελέγχει συνεχώς (spins) την τιμή του στοιχείου μέχρι να αλλάξει από 1 σε 0. Αυτό συμβαίνει για όλα τα νήματα, με εξαίρεση το νήμα-αρχηγό (master thread) το οποίο διατρέχει όλες τις θέσεις του πίνακα και ελέγχει αν το αντίστοιχο νήμα έχει φτάσει στον barrier, δηλαδή αν η τιμή του στοιχείου `arrived[x]` ισούται με 1. Μόλις αντιληφθεί ότι όλα τα νήματα έχουν φτάσει, γράφει σε όλες τις θέσεις του πίνακα την τιμή μηδέν η οποία σηματοδοτεί την απελεθέρωση των νημάτων από τον barrier ώστε να συνεχίσουν την εκτέλεσή τους.

Ο ψευδοκώδικας σε γλώσσα C του αλγορίθμου που περιγράφει τη βασική αρχιτεκτονική του barrier φαίνεται στα Προγράμματα 4.1 και 4.2.

Πρόγραμμα 4.1: Απλός barrier για όλα τα νήματα πλην του νήματος-αρχηγού.

[2]Τρέχουσα τιμή ίση με 256.

```
1  arrived[myid] = 1;          /* Mark me as arrived. */
2  while (arrived[myid] == 1)  /* Spin until released. */
3      ;
```

Πρόγραμμα 4.2: Απλός barrier για όλα τα νήματα πλην του νήματος-αρχηγού.

```
1  for every thread (i):
2      while (arrived[i] != 1)  /* Wait until thread i arrives */
3          ;
4
5  for every thread (i):
6      arrived[i] = 0;  /* Release thread i */
```

Ο πίνακας `arrived` θα πρέπει να έχει αρχικοποιηθεί σε μηδέν κατά την αρχικοποίηση του barrier.

4.3.2 Parallel Barrier (PB)

Ο parallel barrier χρησιμοποιείται αποκλειστικά και μόνο στο τέλος των παράλληλων περιοχών ώστε να εξασφαλίσει τόσο την ολοκλήρωση της εκτέλεσης των tasks, όσο και το συγχρονισμό των νημάτων. Στον barrier αυτό χρησιμοποιούνται δύο πίνακες μεγέθους `MAX_BAR_THREADS` αρχικοποιημένοι στο μηδέν.

Αρχικά, όλα τα νήματα πλην του νήματος-αρχηγού μόλις φτάσουν στον barrier γράφουν την τιμή 2 στη θέση `arrived[myid]`, όπου `myid` το αριθμητικό αναγνωριστικό του εκάστοτε νήματος. Στη συνέχεια περιμένουν μέχρι να αλλάξει η τιμή `arrived[myid]`, ελέγχοντας παράλληλα για το αν υπάρχουν tasks και σε περίπτωση που υπάρχουν τα εκτελούν. Η διαδικασία ελέγχου για tasks είναι χρονοβόρα καθώς το κάθε νήμα δεν ελέγχει μόνο τη δική του ουρά (queue) για να δει αν υπάρχουν διαθέσιμα tasks προς εκτέλεση, αλλά ελέγχει και τις ουρές όλων των υπόλοιπων νημάτων ώστε να να "κλέψει" ένα task και να το εκτελέσει. Η τεχνική αυτή είναι γνωστή ως *work stealing* και στην περίπτωση των NUMA συστημάτων όπου πραγματοποιούνται απομακρυσμένες προσβάσεις μνήμης, ο χρόνος που απαιτείται για να διεκπεραιωθεί αυξάνεται σημαντικά.

Μόλις φτάσουν όλα τα νήματα στον barrier θα έχει εξασφαλιστεί ο συγχρονισμός, οπότε στη συνέχεια πρέπει να εξασφαλιστεί ότι όλα τα tasks έχουν ολοκληρωθεί. Αυτό γίνεται μέσω ενός δεύτερου σημείου συγχρονισμού, στο οποίο θα περιμένουν τα νήματα αφού ολοκληρώσουν την εκτέλεση των tasks που πιθανώς ανέλαβαν κατά

την αναμονή τους στο πρώτο σημείο συγχρονισμού.

Ο ψευδοκώδικας σε γλώσσα C του αλγορίθμου του parallel barrier φαίνεται στα Προγράμματα 4.3 και 4.4.

Πρόγραμμα 4.3: Parallel barrier για όλα τα νήματα πλην του νήματος-αρχηγού.

```
1  /* 1st synchronization point */
2  arrived[myid] = 2;
3  while (arrived[myid] == 2)
4      if (task_exist)
5          check_and_execute_tasks();
6
7  if (task_exist)
8      check_and_execute_tasks();
9
10 /*2nd synchronization point */
11 released[myid] = 1;
12 while (released[myid] == 1)
13     ;
```

Πρόγραμμα 4.4: Parallel barrier για το νήμα-αρχηγό.

```
1  /* 1st synchronization point */
2  for every thread (i):
3      while (arrived[i] != 2)
4          if (task_exist)
5              check_and_execute_tasks();
6
7  for every thread (i):
8      arrived[i] = 0;
9
10 /* 2nd synchronization point */
11 for every thread (i):
12     while (released[i] != 1)
13         ;
14
15 for every thread (i):
16     released[i] = 0;
```

Είναι σημαντικό να σημειωθούν τα εξής:

- Το `task_exist` είναι μία μεταβλητή-σημαία (flag) για το αν έχουν φτιαχτεί

tasks.

- Ανάμεσα στα δύο σημεία συγχρονισμού, υπάρχει ένας ακόμα έλεγχος για tasks. Αυτός ο έλεγχος καλύπτει την περίπτωση στην οποία το τελευταίο νήμα λίγο πριν φτάσει στο πρώτο σημείο συγχρονισμού, έφτιαξε ένα task, το νήμα-αρχηγός είδε ότι όλα τα νήματα έφτασαν στον barrier και ξεκίνησε την απελευθέρωσή τους από το πρώτο σημείο συγχρονισμού, πριν προλάβει κάποιο νήμα να ελέγξει για tasks και να εντοπίσει το νέο task.

- Κάθε στοιχείο των πινάκων `arrived` και `released` καταλαμβάνει χώρο όσο μία γραμμή[3] της κρυφής μνήμης για την αποφυγή του φαινομένου *false sharing*[4]. (Ισχύει και για τους τρεις τύπους barrier)

- Όσο τα νήματα περιμένουν την απελευθέρωσή τους από το νήμα-αρχηγό, ανά κάποιο σταθερό χρονικό διάστημα, παραχωρούν την προτεραιότητά τους (yield) σε άλλα νήματα ώστε να εκτελεστούν. Το ίδιο συμβαίνει και για το νήμα-αρχηγό όσο περιμένει την άφιξη ενός νήματος στον barrier. (Ισχύει και για τους τρεις τύπους barrier)

4.3.3 Default Barrier (DB)

Ο default barrier είναι η πιο απλή μορφή barrier και σε αντίθεση με τον parallel barrier χρησιμοποιείται εντός των παράλληλων περιοχών. Σκοπός του είναι να εξασφαλίσει μόνο το συγχρονισμό των νημάτων καθώς υποθέτει ότι δεν υπάρχουν tasks. Σε περίπτωση όμως που δημιουργηθούν tasks, τα νήματα μεταβαίνουν σε μια πιο πολύπλοκη μορφή barrier, τον task barrier που περιγράφεται στην Υποενότητα 4.3.4.

Ο barrier αυτός είναι σε μεγάλο βαθμό ίδιος με τη βασική αρχιτεκτονική που περιγράφηκε στην Υποενότητα 4.3.1, με τη μόνη διαφορά ότι πραγματοποιεί ελέγχους για την ύπαρξη tasks. Ο έλεγχος για tasks πραγματοποιείται με τον ίδιο τρόπο όπως στον parallel barrier και γι' αυτό δεν θα τον περιγράψουμε περαιτέρω.

[3]Cache line ή cache block: Η ποσότητα πληροφορίας που μεταφέρεται μεταξύ κρυφής μνήμης και κύριας μνήμης. Είναι σταθερού μεγέθους και μία τυπική τιμή είναι τα 64 bytes.

[4]Κατά το φαινόμενο του false sharing, διαφορετικά νήματα ενημερώνουν δεδομένα στην ίδια γραμμή της κρυφής μνήμης, με αποτέλεσμα να δημιουργείται πολύ υψηλή κίνηση στο δίκτυο διασύνδεσης λόγω του πρωτοκόλλου συνοχής κρυφής μνήμης.

Ο ψευδοκώδικας σε γλώσσα C του αλγορίθμου του default barrier φαίνεται στα Προγράμματα 4.5 και 4.6.

Πρόγραμμα 4.5: Default barrier για όλα τα νήματα πλην του νήματος-αρχηγού.

```
1  arrived[myid] = 1;
2  while (arrived[myid] == 1)
3      if (task_exist)
4          goto TB;
```

Πρόγραμμα 4.6: Default barrier για το νήμα-αρχηγό.

```
1  for every thread (i):
2      while (arrived[i] != 2)
3          if (task_exist)
4              goto TB;
5
6  if (task_exist)
7      goto TB;
8
9  for every thread (i):
10     arrived[i] = 0;
```

4.3.4 Task Barrier (TB)

Στον task barrier μεταβαίνουν τα νήματα αφού ησέλθουν πρώτα στον default barrier με την προϋπόθεση ότι δημιουργήθηκαν tasks. Εξασφαλίζει τόσο την ολοκλήρωση των tasks, όσο και τον συγχρονισμό των νημάτων.

Αν και παλιότερα ο parallel barrier ήταν πιο απλός από τον task barrier[5], στην τρέχουσα μορφή τους είναι σχεδόν ίδιοι. Η μόνη διαφορά τους είναι ότι ο parallel barrier πρώτα ελέγχει αν έχουν φτιαχτεί tasks και στη συνέχεια ψάχνει να βρει στις ουρές όλων των νημάτων κάποιο task για να εκτελέσει. Ο task barrier ξεκινάει αμέσως την αναζήτηση για tasks, μιας και ο έλεγχος της μεταβλητής-σημαίας έχει ήδη γίνει στον default barrier.

Ο ψευδοκώδικας σε γλώσσα C του αλγορίθμου του task barrier φαίνεται στα Προγράμματα 4.7 και 4.8.

Πρόγραμμα 4.7: Task barrier για όλα τα νήματα πλην του νήματος-αρχηγού.

[5] Αποτελούνταν από ένα σημείο συγχρονισμού αντί για δύο.

```
/* 1st synchronization point */
arrived[myid] = 2;
while (arrived[myid] == 2)
    check_and_execute_tasks();

check_and_execute_tasks();

/*2nd synchronization point */
released[myid] = 1;
while (released[myid] == 1)
    ;
```

Πρόγραμμα 4.8: Task barrier για το νήμα-αρχηγό.

```
/* 1st synchronization point */
for every thread (i):
    while (arrived[i] != 2)
        check_and_execute_tasks();

for every thread (i):
    arrived[i] = 0;

/* 2nd synchronization point */
for every thread (i):
    while (released[i] != 1)
        ;

for every thread (i):
    released[i] = 0;
```

4.3.5 Απαιτήσεις μνήμης

Στο `struct ort_defbar_t` που αποθηκεύονται όλα τα δεδομένα του barrier, υπάρχουν τρεις πίνακες. Οι πίνακες αυτοί είναι οι `arrived` και `released` που είδαμε να χρησιμοποιούνται στους τρεις τύπους barrier που αναφέραμε νωρίτερα, καθώς και ο πίνακας `phase` ο οποίος χρησιμοποιείται για την υλοποίηση της τεχνικής sense reversal που περιγράφηκε στην Ενότητα 4.1. Για λόγους διατήρησης της απλότητας των αλγορίθμων η περιγραφή της τεχνικής αυτής παραλείφθηκε.

Κάθε πίνακας αποτελείται από `MAX_BAR_THREADS` (256) στοιχεία, με κάθε στοιχείο

να απαιτεί μέγεθος όσο μία γραμμή της κρυφής μνήμης (128 bytes για αρχιτεκτονική *x86-64* ή για επεξεργαστές που είναι άγνωστη η αρχιτεκτονική τους). Άρα, κάθε πίνακας απαιτεί $256 \times 128\text{B} = 32\text{KiB}$ και συνολικά και οι τρεις πίνακες καταλαμβάνουν 96KiB ή αλλιώς 24 pages[6].

Επειδή κάθε νήμα διαθέτει barrier, σε ένα σύστημα NUMA στο οποίο δημιουργείται μία παράλληλη ομάδα με 128 νήματα, μόνο οι barriers απαιτούν $128 \times 96\text{KiB} = 12\text{MiB} = 3073$ pages. Σε ένα μικρό σύστημα τύπου SMP, στο οποίο δημιουργούνται 4 νήματα, οι barriers απαιτούν $4 \times 96\text{KiB} = 96$ pages.

Αν στα παραπάνω συνυπολογιστεί ότι υπάρχει υποστήριξη για πολλαπλές ομάδες και εμφωλευμένο παραλληλισμό, γίνεται εύκολα αντιληπτό ότι οι απαιτήσεις σε μνήμη είναι πραγματικά τεράστιες και πιθανότατα οδηγούν και σε μειωμένες επιδόσεις.

4.4 Βελτιστοποίηση απαιτήσεων μνήμης του barrier του OMPi

Στα πλαίσια αυτής της διπλωματικής εργασίας, έγινε προσπάθεια βελτίωσης του barrier σε συστήματα NUMA. Πριν επικεντρωθούμε στις ιδιαιτερότητες των συστημάτων NUMA, προχωρήσαμε σε κάποιες βελτιώσεις που αφορούν τη μείωση των απαιτήσεων μνήμης ανεξαρτήτως του υποκείμενου συστήματος.

Όπως αναφέρθηκε στην Υποενότητα 4.3.5, οι απαιτήσεις μνήμης του OMPi που σχετίζονται με τον barrier είναι ιδιαίτερα αυξημένες. Οι λόγοι που συμβαίνει αυτό είναι οι εξής:

- Ένα τυπικό μέγεθος των γραμμών της κρυφής μνήμης είναι 64 αντί για 128 bytes που χρησιμοποιείται σαν μέγεθος γραμμής από τον OMPi.
- Η εκμετάλλευση του διαθέσιμου χώρου μνήμης από τους πίνακες `arrived`, `released` και `phase` δεν είναι η βέλτιστη δυνατή.
- Το μέγεθος των πινάκων του barrier είναι σταθερό (`MAX_BAR_THREADS`) και καθορίζεται σε χρόνο μετάφρασης του OMPi. Εκτός αυτού, εμμέσως πλην σαφώς, το μέγιστο πλήθος των νημάτων που συμμετέχουν σε μια παράλληλη ομάδα δεν μπορεί να ξεπεράσει το συγκεκριμένο όριο.

[6] Σελίδες εικονικής μνήμης. Μέγεθος στον Linux Kernel: 4096 bytes.

- Όλα τα νήματα διαθέτουν μεταβλητή τύπου barrier (`ort_defbar_t`) καθώς αποτελεί πεδίο της δομής ελέχου τους (EECB), ακόμα κι αν δεν γίνουν ποτέ νήματα-αρχηγοί.

Οι βελτιώσεις που έγιναν είναι οι ακόλουθες:

1. Το μέγεθος των γραμμών της κρυφής μνήμης από 128 bytes άλλαξε σε 64.

2. Επειδή κάθε στοιχείο των πινάκων `arrived`, `released` και `phase` πιάνει χώρο όσο μία γραμμή της κρυφής μνήμης, δηλαδή 64 bytes, αλλά χρησιμοποιούνται μόλις τα τέσσερα[7] (4) από αυτά, οι τρεις πίνακες συγχωνεύτηκαν σε έναν. Ο πίνακας αυτός ονομάστηκε `status`.

3. Στο EECB του κάθε νήματος αποθηκεύεται δείκτης σε μεταβλητή και όχι μεταβλητή τύπου barrier (`ort_defbar_t`), ενώ η δέσμευση του barrier γίνεται δυναμικά σε χρόνο εκτέλεσης μόνο από τα νήματα-αρχηγούς.

4. Η δέσμευση του πίνακα `status` γίνεται δυναμικά ανάλογα το μέγεθος της ομάδας, γεγονός που επιτρέπει πέρα από την εξοικονόμηση μνήμης, την υποστήριξη αυθαίρετου πλήθους νημάτων που συμμετέχουν σε μια παράλληλη ομάδα και όχι το πολύ `MAX_BAR_THREADS` νημάτων.

Σε ένα σύστημα NUMA στο οποίο δημιουργείται μία παράλληλη ομάδα με 128 νήματα οι απαιτήσεις μνήμης σε σελίδες (pages) μετά από κάθε μία από τις παραπάνω βελτιώσεις φαίνονται στον Πίνακα 4.1.

Α/Α Βελτίωσης	Barriers[8]	Πίνακες/Barrier	Στοιχεία/Πίνακα	Μέγεθος στοιχείου	ΣΥΝΟΛΟ[9]
1	128	3	256	64 B	1536
2	128	1	256	64 B	512
3	1	1	256	64 B	4
4	1	1	128	64 B	2

Πίνακας 4.1: Οι απαιτήσεις σε μνήμη του barrier του OMPi μετά από κάθε βελτίωση.

Συνεπώς, ο απαιτούμενος χώρος μνήμης μειώθηκε από τα 12 MiB (3073 pages) στα 8 KiB (2 pages), δηλαδή υπήρξε περίπου μία μείωση της τάξης του 99.93% και η οποία είναι ανεξάρτητη του πλήθους των νημάτων που συμμετέχουν στην παράλληλη ομάδα.

[7]Τυπικό μέγεθος ακεραίου σε συστήματα Linux αρχιτεκτονικής 64-bit.

[9]Συνολικό πλήθος μεταβλητών τύπου barrier (`ort_defbar_t`).

[9]Συνολικό μέγεθος απαιτήσεων μνήμης σε σελίδες (pages).

4.5 Επανασχεδιασμός για συστήματα NUMA

Παρόλο που η υλοποίηση του υπάρχοντα barrier του OMPi είναι καλή για σχετικά μικρά συστήματα[10] τύπου UMA, παρατηρήθηκαν μειωμένες επιδόσεις όσον αφορά μεγαλύτερα συστήματα τύπου NUMA. Όπως ήδη αναφέρθηκε στην Ενότητα 3.2, ο αποδοτικός προγραμματισμός των συστημάτων NUMA απαιτεί να εστιάσουμε στην αρχή της τοπικότητας μέσω της εκμετάλλευσης πληροφοριών που σχετίζονται με την τοπολογία του υποκείμενου συστήματος. Για το λόγο αυτό πραγματοποιήθηκε επανασχεδιασμός του αλγορίθμου του barrier που χρησιμοποιείται στον OMPi ώστε να λαμβάνει υπόψιν του την ύπαρξη κόμβων NUMA (ή απλώς κόμβων).

4.5.1 Σύντομη περιγραφή

Πιο συγκεκριμένα, υλοποιήθηκε ένας αλγόριθμος combining tree barrier δύο επιπέδων ο οποίος χρησιμοποιείται σε περίπτωση που τα νήματα μιας παράλληλης ομάδας είναι τοποθετημένα[11] σε δύο ή περισσότερους κόμβους. Σε κάθε κόμβο τοποθετείται ένας barrier-φύλλο (leaf barrier) στον οποίο συγχρονίζονται τα νήματα τα οποία είναι τοποθετημένα στον κόμβο αυτό.

Καθώς ο υπάρχων αλγόριθμος βασίζεται στη χρήση πινάκων, θα πρέπει το κάθε νήμα να αντιστοιχίζεται σε μία μοναδική θέση του πίνακα του barrier. Μέχρι τώρα, η αντιστοίχιση γινόταν βάσει του μοναδικού αναγνωριστικού του κάθε νήματος (EEID - Execution Entity ID) καθώς ο πίνακας ήταν ένας και χρησιμοποιούνταν από όλα τα νήματα. Επειδή όμως στο νέο αλγόριθμο υπάρχουν πολλαπλοί barriers και το πλήθος των νημάτων σε κάθε κόμβο/barrier μπορεί να διαφέρει, θα πρέπει να γίνει αντιστοίχιση του μοναδικού αναγνωριστικού του κάθε νήματος σε ένα τοπικό αναγνωριστικό (LID - Local ID) με ισχύ εντός του κόμβου. Τα τοπικά αναγνωριστικά χρησιμοποιούνται τόσο στους barriers-φύλλα, όσο και στον barrier-ρίζα (root barrier), ενώ οι πιθανές τιμές τους κυμαίνονται στο εύρος $[0, T_n)$. Στην περίπτωση των barrier-φύλλων το T_n αντιστοιχεί στο πλήθος των νημάτων που είναι τοποθετημένα στον κόμβο n, ενώ στην περίπτωση του barrier-ρίζα αντιστοιχεί στο πλήθος των κόμβων που χρησιμοποιούνται από την παράλληλη ομάδα.

Υπεύθυνοι για το συγχρονισμό εντός των κόμβων είναι τα νήματα με τοπικό αναγνωριστικό ίσο με μηδέν. Αυτά τα νήματα μπορούν να θεωρηθούν ως νήματα-

[10] Συστήματα με περίπου το πολύ 32 H/W threads.

[11] Δηλαδή όταν η processor binding policy είναι `true`, `close`, `spread` ή `master`.

αρχηγοί στο τοπικό επίπεδο των κόμβων. Αφού επιτευχθεί συγχρονισμός σε επίπεδο κόμβου μέσω του barrier-φύλλου, τα τοπικά νήματα-αρχηγοί συνεχίζουν στο επόμενο επίπεδο μεταβαίνοντας στον barrier-ρίζα ώστε να επιτευχθεί και συγχρονισμός σε επίπεδο συστήματος. Στον barrier-ρίζα υπεύθυνος για το συγχρονισμό είναι το νήμα με μοναδικό αναγνωριστικό (EEID) ίσο με μηδέν. Όταν φτάσουν στον barrier-ρίζα όλα τα νήματα-εκπρόσωποι των κόμβων, έχει επιτευχθεί συγχρονισμός σε επίπεδο συστήματος και ξεκινάει η διαδικασία απελευθέρωσης με κατεύθυνση αντίθετη από αυτή της διαδικασίας άφιξης.

4.5.2 Λεπτομέρειες υλοποίησης

Στα πλαίσια της υλοποίησης χρησιμοποιήθηκε η βιβλιοθήκη `libnuma` για την ανάκτηση πληροφοριών όπως το πλήθος των διαθέσιμων κόμβων NUMA, την αντιστοίχιση των αριθμητικών αναγνωριστικών των επεξεργαστών στα αριθμητικά αναγνωριστικά των κόμβων στους οποίους υπάγονται, καθώς και τη δέσμευση των μεταβλητών τύπου barrier σε συγκεκριμένους κόμβους με σκοπό την επίτευξη της τοπικότητας.

Αρχικά, κατά την εκκίνηση του συστήματος χρόνου εκτέλεσης του OMPi και υπό την προϋπόθεση ότι η βιβλιοθήκη `libnuma` είναι διαθέσιμη, πραγματοποείται έλεγχος για την εξακρίβωση των κόμβων στους οποίους επιτρέπεται η δέσμευση μνήμης και για κάθε επεξεργαστή υπολογίζεται και αποθηκεύεται ο κόμβος στον οποίο ανήκει.

Στη συνάρτηση `prepare_master()` κατά την οποία προετοιμάζεται το νήμα-αρχηγός για την επικείμενη παράλληλη περιοχή, γίνεται υπολογισμός διάφορων πληροφοριών που θα χρησιμοποιηθούν για την αρχικοποίηση του barrier. Οι πληροφορίες αυτές αφορούν:

- Τους κόμβους οι οποίοι θα χρησιμοποιηθούν στην παράλληλη περιοχή που ακολουθεί.
- Την ανάθεση τοπικών αναγνωριστικών κάθε νήματος. Ένα τοπικό αναγνωριστικό για τον barrier-φύλλο και ένα για τον barrier-ρίζα.
- Το πλήθος των νημάτων που θα τοποθετηθούν σε κάθε κόμβο.

Ο υπολογισμός των κόμβων που θα χρησιμοποιηθούν από μια παράλληλη ομάδα γίνεται με τον ακόλουθο τρόπο. Για κάθε νήμα υπολογίζεται το place στο οποίο θα τοποθετηθεί βάσει του place partition και της processor binding policy που βρίσκεται

σε ισχύ στη συγκεκριμένη παράλληλη περιοχή. Στη συνέχεια, χρησιμοποιείται ο πρώτος επεξεργαστής του συγκεκριμένου place για την εύρεση του κόμβου στον οποίο ανήκει.

Ο λόγος που χρησιμοποιείται ο πρώτος επεξεργαστής είναι επειδή θεωρούμε ότι όλοι οι επεξεργαστές ενός place ανήκουν στον ίδιο κόμβο, καθώς στη συνάρτηση `prepare_master()` είναι αδύνατο να γνωρίζουμε σε ποιον επεξεργαστή ακριβώς θα τοποθετηθεί το νήμα και συνεπώς ποιος κόμβος θα χρησιμοποιηθεί. Στον OMPi, όταν τα OpenMP places καθορίζονται βάσει αυθαίρετου ονόματος, όλοι οι επεξεργαστές που περιλαμβάνονται σε ένα place ανήκουν στον ίδιο κόμβο. Αν όμως οριστούν μέσω ρητής λίστας από τον χρήστη, υπάρχει η πιθανότητα να δημιουργηθούν places των οποίων οι επεξεργαστές ανήκουν σε διαφορετικούς κόμβους. Σε περίπτωση που το νήμα τοποθετηθεί σε κόμβο διαφορετικό από τον κόμβο του πρώτου επεξεργαστή, ο αλγόριθμος λειτουργεί σωστά με τη μόνη διαφορά ότι θα πραγματοποιούνται απομακρυσμένες προσβάσεις μνήμης για το συγχρονισμό του συγκεκριμένου νήματος.

Αφού υπολογιστούν όλες οι απαραίτητες πληροφορίες που αναφέρθηκαν παραπάνω, πραγματοποιείται έλεγχος για να καθοριστεί αν χρειάζεται η χρήση του combining tree barrier ή απλώς *tree barrier* για συντομία. Αν η επικείμενη παράλληλη ομάδα χρησιμοποιεί δύο ή περισσότερους κόμβους, τότε χρησιμοποιείται ο tree barrier. Σε αντίθετη περίπτωση χρησιμοποιείται ο κλασικός barrier του OMPi που περιγράφηκε στην Ενότητα 4.3.

Η μεταβλητή τύπου δείκτη σε barrier που υπήρχε στη δομή ελέγχου EECB των νημάτων αντικαταστάθηκε με έναν πίνακα δεικτών σε barrier μεγέθους $N+1$, όπου N το πλήθος των διαθέσιμων κόμβων. Οι πρώτοι N δείκτες αντιστοιχούν στους barriers-φύλλα, ενώ ο τελευταίος δείκτης αντιστοιχεί στον barrier-ρίζα. Στη συνάρτηση `prepare_master()` δεσμεύεται δυναμικά μνήμη για τον πίνακα δεικτών καθώς και για κάθε barrier, μόνο μία φορά πριν την πρώτη χρήση του. Επίσης, σε κάθε κλήση της αρχικοποιείται ο barrier κάθε κόμβου που πρόκειται να χρησιμοποιηθεί στην παράλληλη περιοχή που ακολουθεί, καθώς και ο barrier-ρίζα.

Στο `struct ort_defbar_t` προστέθηκε το πεδίο `type` που χρησιμοποιείται για τον καθορισμό του τύπου του barrier και οι πιθανές τιμές είναι:

- `FLAT_BARRIER`: Για τον κλασικό barrier του OMPi.
- `TREE_BARRIER_LOCAL`: Για τους barriers-φύλλα του tree barrier.
- `TREE_BARRIER_ROOT`: Για τον barrier-ρίζα του tree barrier.

4.5.3 Ο αλγόριθμος του νέου barrier

Κατά την υλοποίηση του combining tree barrier που υλοποιήθηκε στον OMPi, τροποποιήθηκαν οι υπάρχουσες συναρτήσεις. Επειδή οι αλλαγές και στους τρεις τύπους barrier είναι ίδιες λόγω της απόκρυψης των λεπτομερειών υλοποίησης της τεχνικής sense reversal, θα γίνει αναφορά μόνο στον νέο αλγόριθμο ενός τύπου και συγκεκριμένα του default barrier.

Στο νέο αλγόριθμο, ο κώδικας του νήματος-αρχηγού εκτελείται από τα τοπικά νήματα-αρχηγούς. Αφού επιτευχθεί συγχρονισμός σε επίπεδο κόμβου, τα τοπικά νήματα-αρχηγοί μεταβαίνουν στον barrier-ρίζα ώστε να επιτευχθεί και συγχρονισμός σε επίπεδο συστήματος.

Ο ψευδοκώδικας σε γλώσσα C του νέου αλγορίθμου του default barrier φαίνεται στα Προγράμματα 4.9 και 4.10.

Πρόγραμμα 4.9: Ο νέος default barrier για όλα τα νήματα πλην των τοπικών νημάτων-αρχηγών.

```
1  local_id = (barrier_type == FLAT_BARRIER) ? my_eeid : my_lid;
2
3  arrived[local_id] = 1;
4  while (arrived[local_id] == 1)
5      if (task_exist)
6          goto TB;
```

Πρόγραμμα 4.10: Ο νέος default barrier για τα τοπικά νήματα-αρχηγούς.

```
1  for every thread (i):
2      while (arrived[i] != 2)
3          if (task_exist)
4              goto TB;
5
6  if (task_exist)
7      goto TB;
8
9  /* At this point, inter-node synchronization has been achieved. In case of
10  * tree barrier, intra-node synchronization should also take place.
11  */
12 if (barrier_type == TREE_BARRIER_LOCAL)
13     goto ROOT_DB;
14
15 for every thread (i):
16     arrived[i] = 0;
```

ΚΕΦΑΛΑΙΟ 5

Πειραματική Αξιολόγηση

Οι διάφορες υλοποιήσεις που πραγματοποιήθηκαν στα πλαίσια της τρέχουσας διπλωματικής εργασίας αξιολογήθηκαν πειραματικά με τη χρήση μετροπρογραμμάτων (benchmarks) τόσο για την επιβεβαίωση της ορθότητας της υλοποίησης, όσο και για τη μέτρηση των επιδόσεων που επιτεύχθηκαν.

5.1 Περιγραφή Συστημάτων

Η εκτέλεση των πειραμάτων έγινε σε δύο υπολογιστικά συστήματα τα οποία αντιπροσωπεύουν συνήθεις αλλά διαφορετικές αρχιτεκτονικές οργανώσεις NUMA. Τα χαρακτηριστικά των συστημάτων αυτών, τόσο από άποψη υλικού, όσο και από άποψη λογισμικού, φαίνονται στους Πίνακες 5.1 και 5.2. Η αρχιτεκτονική των επεξεργαστών όλων των συστημάτων είναι η *x86-64* ενώ το μέγεθος μιας γραμμής της κρυφής μνήμης είναι 64 bytes.

Οι μεταφραστές οι οποίοι χρησιμοποιήθηκαν για τη συγκριτική αξιολόγηση της δουλειάς μας είναι ο *GNU C Compiler* (GCC), ο *Intel C Compiler* (ICC)[1] και ο *Clang*. Ο πηγαίος κώδικας του OMPi μεταφράστηκε με τον GCC, ενώ τα απαιτούμενα πακέτα λογισμικού είναι διαθέσιμα στο Παράρτημα Α.

[1] Εγκαταστάθηκε μέσω των oneAPI Toolkits της Intel®.

Hostname	Proc. Mfr.	NUMA nodes	Sockets	Cores	H/W threads	RAM
parade	Intel	4	4	64	128	256 GB
paragon	AMD	4	2	24	24	16 GB

Πίνακας 5.1: Χαρακτηριστικά υλικού των πειραματικών συστημάτων.

Hostname	OS	GCC	ICC	Clang	hwloc	Linux kernel
parade	CentOS 8	8.4.1	2021.3.0	11.0.0	2.2.0	4.18.0
paragon	CentOS 8	8.4.1	2021.3.0	11.0.0	2.2.0	4.18.0

Πίνακας 5.2: Χαρακτηριστικά λογισμικού των πειραματικών συστημάτων.

5.1.1 Parade

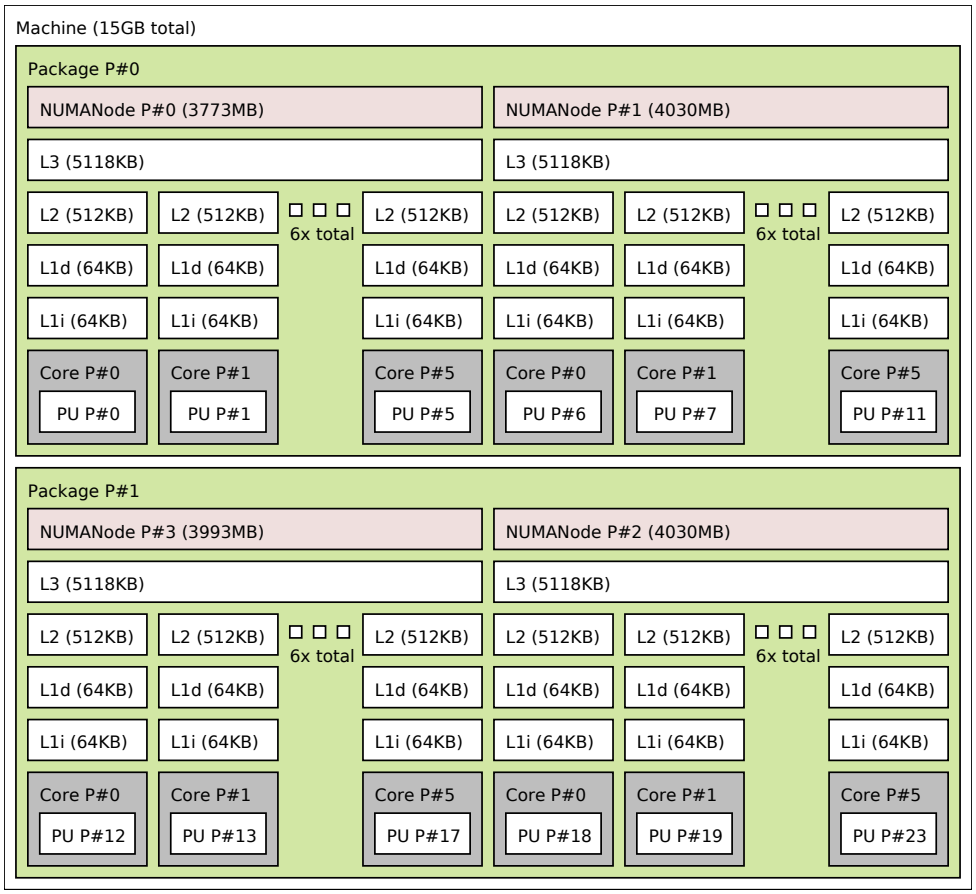
Ο Parade είναι ένα σύστημα Dell PowerEdge R840 με τέσσερις κόμβους NUMA. Κάθε κόμβος διαθέτει 64 GBs μνήμης και έναν επεξεργαστή *Intel® Xeon® Gold 6130* ο οποίος αποτελείται από 12 πυρήνες και ιεραρχία κρυφών μνημών τριών επιπέδων (L1i & L1d, L2, L3). Τα επίπεδα ένα και δύο των κρυφών μνημών είναι κοινά ανά πυρήνα, ενώ το επίπεδο τρία είναι κοινό για όλους τους πυρήνες του επεξεργαστή. Επίσης, κάθε πυρήνας περιέχει δύο H/W threads. Η σχηματική αναπαράσταση της τοπολογίας του είχε δοθεί στο Σχήμα 3.2.

5.1.2 Paragon

Ο Paragon είναι ένα σύστημα με δύο επεξεργαστές *AMD Opteron™ Processor 6166 HE*, καθένας από τους οποίους περιλαμβάνει δύο κόμβους NUMA. Κάθε κόμβος διαθέτει 6 πυρήνες του ενός H/W thread και ιεραρχία κρυφής μνήμης τριών επιπέδων αντίστοιχη με τους κόμβους του συστήματος Parade που είδαμε προηγουμένως. Η σχηματική αναπαράσταση της τοπολογίας του φαίνεται στο Σχήμα 5.1.

Ο λόγος που κάθε επεξεργαστής περιλαμβάνει δύο κόμβους είναι επειδή ουσιαστικά αποτελείται από δύο κυκλώματα επεξεργαστών (dies) των έξι πυρήνων το καθένα, τα οποία συνδέονται μεταξύ τους με ένα δίκτυο διασύνδεσης χαμηλής καθυστέρησης και υψηλού εύρους ζώνης που ονομάζεται HyperTransport, με σκοπό να δημιουργηθεί ένα ολοκληρωμένο κύκλωμα επεξεργαστή με 12 πυρήνες [14]. Κάθε die μπορεί να επικοινωνήσει απευθείας με τη μνήμη[2], οπότε λόγω της ύπαρξης δι-

[2]Επειδή περιλαμβάνει ελεγκτή μνήμης (memory controller).



Σχήμα 5.1: Η τοπολογία του συστήματος Paragon.

κτύου διασύνδεσης μεταξύ των dies, κάθε επεξεργαστής μπορεί να θεωρηθεί ως ένα σύστημα NUMA δύο κόμβων.

5.2 Τοπολογία

Η υλοποίηση των OpenMP places και OpenMP processor binding policies ελέγχθηκε για την ορθότητά της, δηλαδή για το αν ικανοποιεί πλήρως τις προδιαγραφές του OpenMP, τόσο με ιδιόχειρα, όσο και με έτοιμα προγράμματα. Τα έτοιμα προγράμματα ήταν μέρος ενός συνόλου προγραμμάτων ελέγχου ορθότητας τα οποία παρέχονται από τη βιβλιοθήκη *GNU Offloading and Multi Processing Runtime Library* (libgomp) η οποία μεταξύ άλλων, υλοποιεί και τη διεπαφή προγραμματισμού εφαρμογών OpenMP που χρησιμοποιεί ο GCC.

5.3 Barrier

Η υλοποίηση του tree barrier ελέγχθηκε για την ορθότητά της με προγράμματα ελέγχου από τη βιβλιοθήκη libgomp και την *OpenMP Validation Suite V 3.0* [15, 16]. Ο λόγος όμως που αναπτύχθηκε ο tree barrier ήταν οι μειωμένες επιδόσεις του υπάρχοντα barrier του OMPi σε συστήματα NUMA, οπότε αφού εξασφαλίστηκε η ορθότητα της υλοποίησης, εστιάσαμε στο ζήτημα των επιδόσεων με τη χρήση μετροπρογραμμάτων.

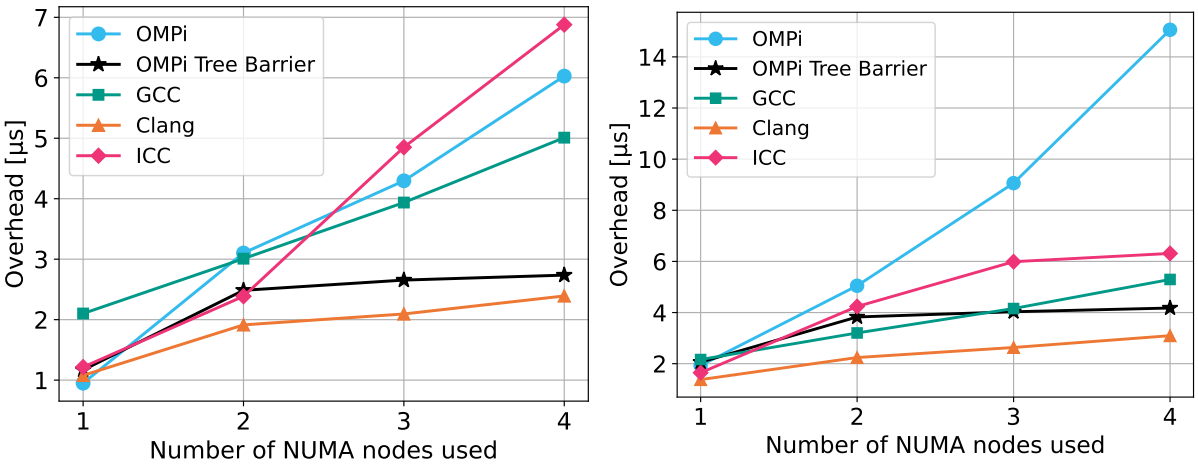
Η *EPCC OpenMP micro-benchmark suite* [17] είναι ένα σύνολο μετροπρογραμμάτων (micro-benchmarks) τα οποία μετράνε το κόστος σε χρόνο (overhead) που απαιτείται για τη διεκπεραίωση λειτουργιών όπως ο συγχρονισμός (π.χ. barrier, κλειδαριές), η *χρονοδρομολόγηση βρόχου* (loop scheduling) και οι πράξεις πινάκων. Η πιο πρόσφατη έκδοση (3.1) υποστηρίζει μέχρι και τις λειτουργίες που προδιαγράφει το OpenMP 3.0.

Τα πειράματα που πραγματοποιήθηκαν επικεντρώθηκαν στη μέτρηση του overhead του barrier, όσο το πλήθος των κόμβων NUMA που χρησιμοποιούνται αυξάνεται. Πιο συγκεκριμένα, ανατίθεται ένα νήμα σε κάθε core ή H/W thread του κόμβου, ενώ πραγματοποιούνται εκτελέσεις με 1 έως N κόμβους, όπου N το πλήθος των διαθέσιμων κόμβων του συστήματος. Με αυτό τον τρόπο ελέγχουμε κατά πόσο η εκάστοτε υλοποίηση barrier είναι κλιμακώσιμη. Οι υλοποιήσεις που συγκρίθηκαν είναι αυτές των μεταφραστών OMPi (κλασική και tree barrier), GCC, ICC και Clang.

5.3.1 Parade

Στο Σχήμα 5.2 φαίνεται το overhead του barrier κάθε μεταφραστή όταν χρησιμοποιούνται 1 έως 4 κόμβοι και ένα νήμα ανά core ή H/W thread αντίστοιχα. Αυτό που παρατηρούμε, είναι ότι όταν `OMP_PLACES="cores"`, για ένα κόμβο ο OMPi έχει το μικρότερο overhead, το οποίο όμως αυξάνει κατά πολύ για περισσότερους κόμβους. Στην περίπτωση που `OMP_PLACES="threads"`, ο OMPi έχει εκθετική αύξηση του overhead για 2 ή περισσότερους κόμβους. Ο tree barrier και στις δύο περιπτώσεις έχει σαφώς καλύτερες επιδόσεις σε σχέση με τον κλασικό barrier του OMPi, ενώ το overhead του αυξάνεται με αρκετά μικρό ρυθμό. Εξαίρεση αποτελεί η περίπτωση που χρησιμοποιείται ένας μόνο κόμβος, κατά την οποία ο tree barrier είναι λίγο πιο αργός από τον κλασικό barrier.

Η σύγκριση που πραγματοποιήθηκε, ναι μεν είναι σωστή από την πλευρά του



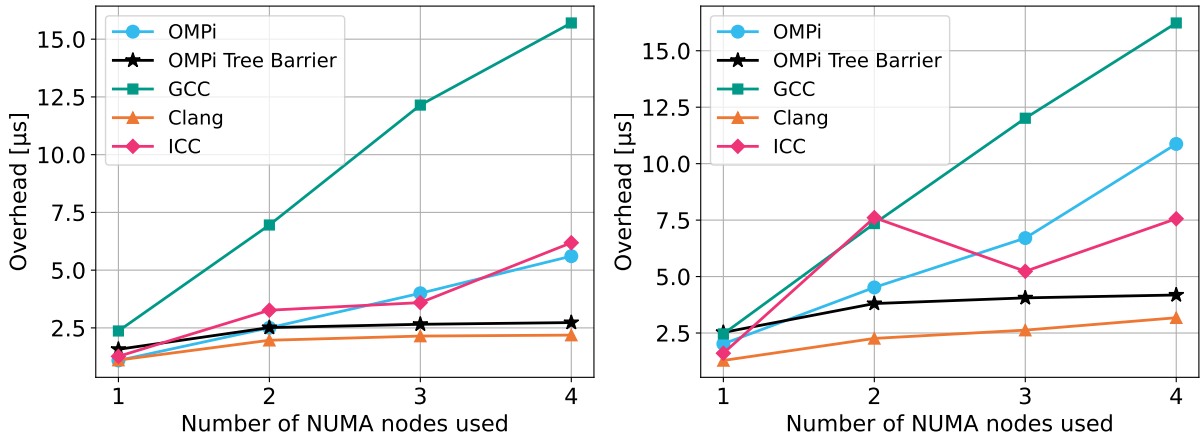
Σχήμα 5.2: Barrier overhead στον Parade (Default places).

χρήστη καθώς χρησιμοποιείται το ίδιο abstract name για τον καθορισμό των places, όμως ο κάθε μεταφραστής μετασχηματίζει με διαφορετικό τρόπο το abstract name σε σύνολα από αναγνωριστικά επεξεργαστών. Για παράδειγμα, όταν `OMP_PLACES=cores` ο GCC τοποθετεί στο place partition με κυκλικό τρόπο (round-robin) ένα core ανά κόμβο. Συνεπώς, αν ο χρήστης επιλέξει ως processor binding policy την `close`, νήματα που θα τοποθετηθούν σε γειτονικά places, θα καταλήξουν σε διαφορετικούς κόμβους NUMA.

Λόγω της ουσιαστικής διαφοράς στο place partition που χρησιμοποιεί το σύστημα χρόνου εκτέλεσης του κάθε μεταφραστή, πραγματοποιήσαμε επιπλέον μετρήσεις κατά τις οποίες ορίσαμε την τιμή της μεταβλητής `OMP_PLACES` χειροκίνητα, βάσει του πώς δημιουργεί το place partition ο κάθε μεταφραστής. Οι μεταφραστές ICC και Clang ορίζουν με τον ίδιο τρόπο το place partition, οπότε προέκυψαν οι τρεις ακόλουθοι τρόποι ορισμού του:

- OMPi places: `OMP_PLACES="ompi-cores"` και `OMP_PLACES="ompi-threads"`.
- GCC places: `OMP_PLACES="gcc-cores"` και `OMP_PLACES="gcc-threads"`.
- Clang/ICC places: `OMP_PLACES="clang-cores"` και `OMP_PLACES="clang-threads"`.

Στα Σχήματα 5.3, 5.4 και 5.5 μπορείτε να δείτε τις μετρήσεις οι οποίες πάρθηκαν με όλους τους μεταφραστές να χρησιμοποιούν το ίδιο place partition. Σε όλες τις περιπτώσεις που χρησιμοποιούνται πολλαπλοί κόμβοι είναι εμφανής η επίτευξη καλύτερης επίδοσης του tree barrier σε σχέση με τον κλασικό barrier του OMPi. Το



Σχήμα 5.3: Barrier overhead στον Parade (OMPi places).

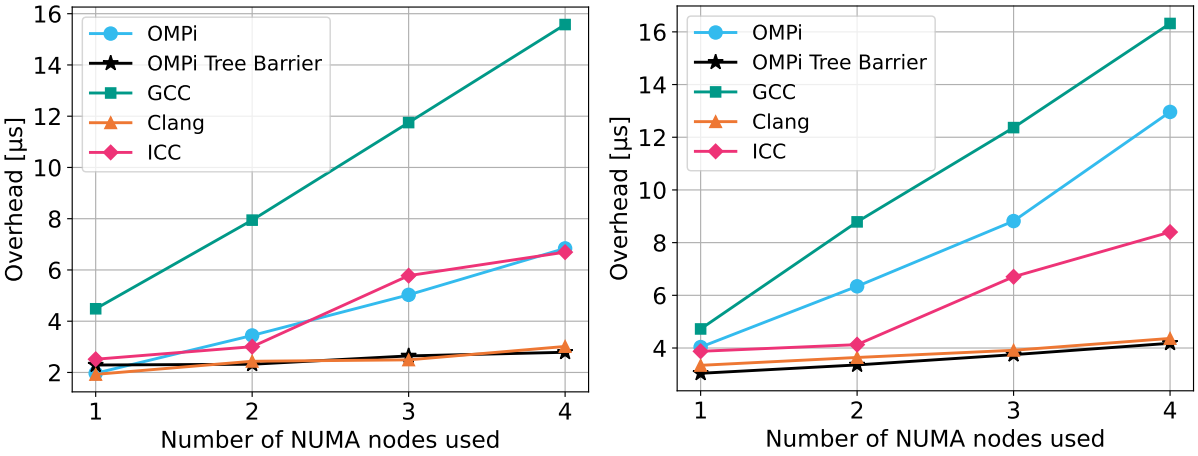
overhead όταν χρησιμοποιούνται δύο κόμβοι είναι σαφώς πιο αυξημένο σε σχέση με όταν χρησιμοποιείται ένας κόμβος, φαινόμενο πλήρως δικαιολογημένο καθώς στην τελευταία περίπτωση δεν πραγματοποιούνται απομακρυσμένες προσβάσεις μνήμης. Όσο το πλήθος των κόμβων αυξάνεται σε τιμές μεγαλύτερες του δύο, παρατηρούμε ότι η αύξηση του overhead είναι πολύ μικρή.

Σε μερικά σχήματα είναι ορατές μη αναμενόμενες συμπεριφορές από τον ICC, όπως για παράδειγμα στο Σχήμα 5.3 όπου διακρίνεται ένα πολύ μεγαλύτερο overhead για δύο κόμβους σε σχέση με τους τρεις κόμβους. Αυτό το φαινόμενο παρατηρήθηκε στην τρέχουσα έκδοση του ICC μετά από την πιο πρόσφατη αναβάθμιση που πραγματοποιήθηκε ώστε οι εκδόσεις των πακέτων λογισμικού να ταυτίζονται με αυτές του συστήματος Paragon και άρα να μην οδηγηθούμε σε εσφαλμένα συμπεράσματα λόγω διαφορών ανάμεσα στις εκδόσεις.

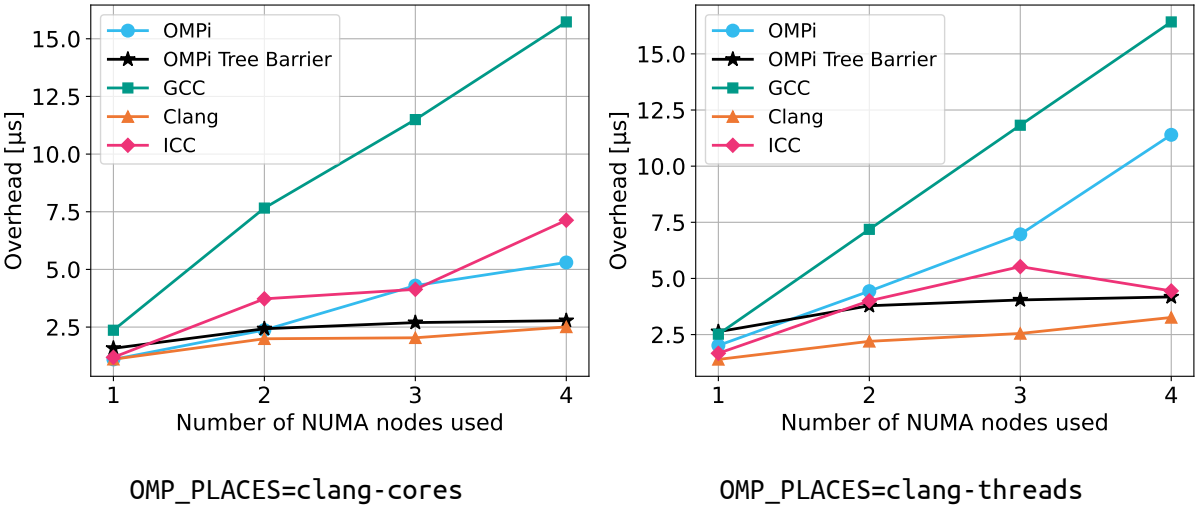
5.3.2 Paragon

Στα Σχήματα 5.6 και 5.7 φαίνονται τα αποτελέσματα των μετρήσεων που πραγματοποιήθηκαν στον Paragon. Επειδή κάθε core διαθέτει μόνο ένα H/W thread, τα abstract names `threads` και `cores` έχουν το ίδιο αποτέλεσμα. Εξαίρεση αποτελούν οι ICC και Clang οι οποίοι εσφαλμένα θεωρούν ότι κάθε επεξεργαστής διαθέτει ένα μόνο core αντί για δώδεκα, γεγονός που θεωρήθηκε ως σφάλμα (bug) και αγνοήθηκε.

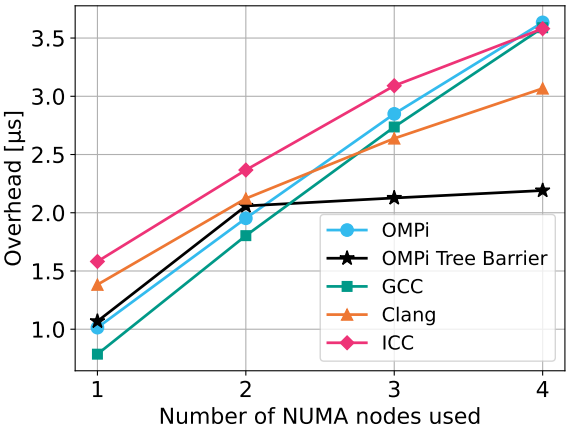
Από τις μετρήσεις στον Paragon παρατηρούμε ότι τα overheads των OMPi, GCC και ICC αυξάνουν σχεδόν γραμμικά και συγκλίνουν για πλήθος τεσσάρων κόμβων. Ο



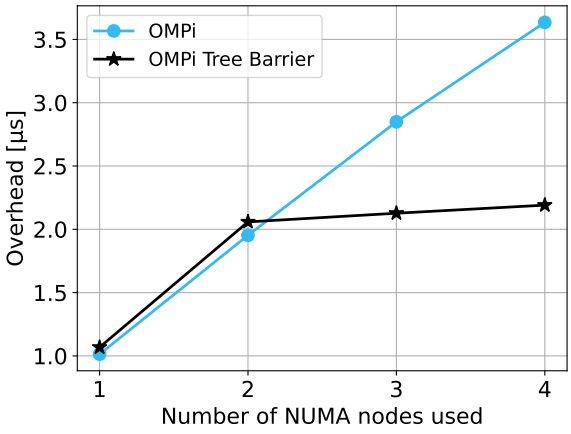
Σχήμα 5.4: Barrier overhead στον Parade (GCC places).



Σχήμα 5.5: Barrier overhead στον Parade (Clang/ICC places).



Σχήμα 5.6: Barrier overhead στον Paragon.



Σχήμα 5.7: Barrier overhead στον Paragon για τους barriers του OMPi.

Clang φαίνεται ότι κλιμακώνει καλύτερα για τρεις και τέσσερις κόμβους από τους OMPi, GCC και ICC, αλλά όχι το ίδιο αποδοτικά όπως συνέβαινε στον Parade. Ο OMPi tree barrier επιδεικνύει την καλύτερη επίδοση για τρεις και τέσσερις κόμβους, με το overhead να αυξάνει με πολύ μικρό ρυθμό όσο αυξάνεται το πλήθος των κόμβων, φαινόμενο που παρατηρήθηκε και στον Parade. Επιπλέον, παρατηρούμε ο tree barrier έχει οριακά μεγαλύτερο overhead σε σχέση με τον κλασικό barrier του OMPi, όχι μόνο για ένα κόμβο αλλά και για δύο. Κάτι τέτοιο πιθανώς συμβαίνει καθώς οι δύο κόμβοι ανήκουν στο ίδιο ενσωματωμένο κύκλωμα επεξεργαστή.

ΚΕΦΑΛΑΙΟ 6

Σύνοψη και Μελλοντική Εργασία

6.1 Ανακεφαλαίωση

Η ανάγκη για όλο και μεγαλύτερη επεξεργαστική ισχύ οδήγησε στη δημιουργία των συστημάτων μη ομοιόμορφης προσπέλασης μνήμης (NUMA), που αποτελούν αρχιτεκτονική εξέλιξη των συμμετρικών πολυεπεξεργαστών (SMPs). Τα συστήματα αυτά παρέχουν κοινόχρηστο χώρο διευθύνσεων και συνεπώς επιτρέπουν τη χρήση διαδεδομένων μοντέλων προγραμματισμού όπως το OpenMP. Λόγω της κατανεμημένης από φυσική άποψη οργάνωση της μνήμης των συστημάτων αυτών, οδηγούμαστε στην εκμετάλλευση πληροφοριών που σχετίζονται με την τοπολογία του υποκείμενου συστήματος για την επίτευξη των βέλτιστων δυνατών επιδόσεων.

Κάτι τέτοιο έρχεται σε αντίθεση με την υψηλού επιπέδου διεπαφή προγραμματισμού εφαρμογών OpenMP, στο πλαίσιο της οποίας χρησιμοποιούνται αφαιρέσεις (abstractions) ώστε να είναι δυνατή η συγγραφή φορητών προγραμμάτων χρήστη. Παρόλα αυτά, λόγω της ανάγκης χρήσης πληροφοριών που σχετίζονται με την τοπολογία του συστήματος, από την έκδοση 4.0 του προτύπου OpenMP και έπειτα, άρχισαν να προδιαγράφονται λειτουργίες που σχετίζονται με την τοπολογία του υποκείμενου συστήματος. Οι λειτουργίες αυτές επιτρέπουν τον ορισμό συνόλων επεξεργαστών (OpenMP places) στα οποία μπορούν να ανατεθούν τα νήματα OpenMP βάσει διάφορων πολιτικών, γνωστών ως OpenMP processor binding policies.

Στα πλαίσια της παρούσας διπλωματικής εργασίας, υλοποιήθηκαν πλήρως τα OpenMP places και OpenMP processor binding policies όπως προδιαγράφονται από την πιο πρόσφατη έκδοση του προτύπου OpenMP και συγκεκριμένα την έκδοση 5.1 (Νοέμβριος 2020). Η ικανότητα ελέγχου του τρόπου ανάθεσης των νημάτων σε

σύνολα από επεξεργαστές, επιτρέπει τον διαμοιρασμό των νημάτων στους διαθέσιμους επεξεργαστές ανάλογα με τη λογική οργάνωση της εργασίας που πρέπει να επιτελέσει το πρόγραμμα χρήστη. Επιπλέον, μειώθηκαν οι απαιτήσεις χώρου του υπάρχοντα barrier και υλοποιήθηκε ένας νέος αλγόριθμος barrier για τον ερευνητικό μεταφραστή OMPi που υποστηρίζει το OpenMP. Σκοπός του νέου αλγορίθμου ήταν η δυνατότητα συγχρονισμού των νημάτων σε συστήματα NUMA με κλιμακώσιμο τρόπο, καθώς το πλήθος των κόμβων αυξάνεται.

Τα μετροπρογράμματα ελέγχου της ορθότητας της υλοποίησης αλλά και μέτρησης του κόστους σε χρόνο (overhead) που απαιτείται για το συγχρονιμό με χρήση barrier, έδειξαν ότι ο νέος barrier έχει τη δυνατότητα να χρησιμοποιείται αποδοτικά για το συγχρονισμό νημάτων που εκτελούνται σε πολλαπλούς κόμβους NUMA. Ιδιαίτερα σημαντικό είναι το γεγονός ότι ο νέος αλγόριθμος barrier σε σχέση με τον κλασικό όχι μόνο είναι λιγότερο κοστοβόρος από άποψη χρόνου, αλλά έχει και καλές ιδιότητες κλιμάκωσης, καθώς για κάθε επιπλέον κόμβο που χρησιμοποιείται, το overhead αυξάνεται με μικρό ρυθμό.

6.2 Μελλοντική Εργασία

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[2] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*. Springer Science & Business Media, 2010.

[3] V. Dimakopoulos, *Parallel Systems and Programming (in Greek)*, 2015.

[4] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[5] Openmp compilers. https://openmp.org/resources/openmp-compilers-tools/.

[6] Ompi compiler. https://paragroup.cse.uoi.gr/wpsite/software/ompi/.

[7] M. Dobson, P. Gaughen, M. Hohnbaum, and E. Focht, "Linux support for numa hardware," in *Proc. Linux Symposium*, vol. 2003. Citeseer, 2003.

[8] M. J. Bligh, M. Dobson, D. Hart, and G. Huizenga, "Linux on numa systems," in *Proceedings of the Linux Symposium*, vol. 1, 2004, pp. 89–102.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 180–186.

[10] A. Kleen, "A numa api for linux," *Novel Inc*, 2005.

[11] L. Torvalds, "Linux kernel," https://github.com/torvalds/linux.

[12] "numa(3) - linux manual page," https://man7.org/linux/man-pages/man3/numa.3.html.

[13] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[14] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE micro*, vol. 30, no. 2, pp. 16–29, 2010.

[15] C. Wang, S. Chandrasekaran, and B. Chapman, "An openmp 3.1 validation testsuite," in *International Workshop on OpenMP*. Springer, 2012, pp. 237–249.

[16] "Openmp validation suite v 3.0," https://github.com/uhhpctools/omp-validation.

[17] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *Proceedings of First European Workshop on OpenMP*, vol. 8. Citeseer, 1999, p. 49.

[18] "Openmp application programming interface v5.1," https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, 2020.

ΠΑΡΑΡΤΗΜΑ Α

Απαιτήσεις λογισμικού του μεταφραστή OMPi

Για τη μετάφραση του OMPi απαιτούνται τα εξής πακέτα λογισμικού:

- `autoconf`
- `automake1.16` (Debian-based distributions), `automake` (CentOS) v1.16
- `bison`
- `flex`
- `gcc` (ή κάποιος άλλος μεταφραστής για τη γλώσσα C)
- `libtool` v2.4.6
- (Προεραιτικά) `libhwloc-dev` (Debian-based distributions), `hwloc-devel` (CentOS)

Για την υποστήριξη των OpenMP places και binding των νημάτων OpenMP στα places απαιτείται επιπλέον το πακέτο:

- `libhwloc-dev` (Debian-based distributions), `hwloc-devel` (CentOS)

Για την υποστήριξη δενδρικού barrier σε NUMA συστήματα απαιτείται επιπλέον το πακέτο:

- `libnuma-dev` (Debian-based distributions), `numactl-devel` (CentOS)

ΠΑΡΑΡΤΗΜΑ Β

Η σύνταξη της μεταβλητής περιβάλλοντος OMP_PLACES

Το σύνολο των έγκυρων τιμών που μπορούν να ανατεθούν στη μεταβλητή `OMP_PLACES` σύμφωνα με το OpenMP 5.1 [18] προκύπτουν βάσει των ακόλουθων συντακτικών κανόνων:

```
<list>         |= <p-list> | <aname>
<p-list>       |= <p-interval> | <p-list>,<p-interval>
<p-interval>   |= <place>:<len>:<stride> | <place>:<len> | <place> | !<place>
<place>        |= {<res-list>} | <res>
<res-list>     |= <res-interval> | <res-list>,<res-interval>
<res-interval> |= <res>:<num-places>:<stride> | <res>:<num-places> | <res>
                  | !<res>
<aname>        |= <word>(<num-places>) | <word>
<word>         |= sockets | cores | ll_caches | numa_domains | threads
                  | <implementation-defined abstract name>
<res>          |= non-negative integer
<num-places>   |= positive integer
<stride>       |= integer
<len>          |= positive integer
```