Υλοποίηση νέων πολιτικών δρομολόγησης για βρόχους του OpenMP

Σπύρος Μαντέλος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

Πανεπιστήμιο Ιωαννίνων

Ιούνιος 2022

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΓΛΩΣΣΑΡΙ

OpenMP	(Open Multi-Processing) Διεπαφή προγραμματισμού για συστήματα κοινόχρηστης μνήμης
MPI	(Message Passing Interface) Διεπαφή προγραμματισμού για συστήματα κατανεμημένης μνήμης
OMPi	Παραλληλοποιητικός μεταφραστής της γλώσσας C που υποστηρίζει τις οδηγίες του OpenMP.
schedule	Λέξη κλειδί/φράση του OpenMP για την επιλογή μεθόδου δρομολόγησης. Κυριολεκτικά πρόγραμμα ή δρομολόγιο.
chunk	Σύνολο επαναλήψεων που ανατίθενται για εκτέλεση σε ένα νήμα. Κυριολεκτικά μεγάλο κομμάτι.
atomics	Αναφέρεται σε συναρτήσεις που πραγματοποιούνται με μία μόνο εντολή και έτσι δεν είναι απαραίτητος ο αμοιβαίος αποκλεισμός. Γενικά μπορεί να αναφέρεται και στους τύπους μεταβλητών όπου η ανάγνωση και η γραφή γίνονται με μία εντολή.

ΠΕΡΙΛΗΨΗ

Σπύρος Μαντέλος, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2022.
Υλοποίηση νέων πολιτικών δρομολόγησης για βρόχους του OpenMP.
Επιβλέπων: Δημακόπουλος Βασίλειος, Αναπληρωτής Καθηγητής.

Το OpenMP έχει εδραιωθεί ως μία από τις πλέον διαδεδομένες μεθόδους παράλληλου προγραμματισμού για συστήματα κοινόχρηστης μνήμης. Με τη χρήση οδηγιών προς τον μεταφραστή, ο προγραμματιστής ελέγχει τη συμπεριφορά του προγράμματος. Μία από τις επιλογές που καλείται να κάνει ο προγραμματιστής είναι η μέθοδος με την οποία διαχωρίζονται και διαμοιράζονται ανάμεσα στα διαθέσιμα νήματα οι επαναλήψεις ενός βρόχου που εκτελείται παράλληλα. Η επιλογή αυτή είναι σημαντική ώστε να επιτευχθεί ο ταχύτερος δυνατός χρόνος μέσα από την ισορροπία φόρτου ανάμεσα στα νήματα διατηρώντας παράλληλα το χαμηλότερο δυνατό κόστος δρομολόγησης. Το OpenMP υποστηρίζει τρεις μεθόδους αλλά στη βιβλιογραφία έχουν αναπτυχθεί και προταθεί αρκετές διαφορετικές προσεγγίσεις. Στο πλαίσιο της παρούσας διπλωματικής εργασίας δημιουργήθηκε υποδομή για την ενσωμάτωση επιπλέον μεθόδων δρομολόγησης στον παραλληλοποιητικό μεταφραστή OMPi. Στη συνέχεια, χρησιμοποιώντας την υποδομή αυτή, μελετήθηκαν και υλοποιήθηκαν τέσσερεις νέες πολιτικές. Επιπροσθέτως, δημιουργήθηκε η δυνατότητα προσθήκης ετικετών στις δομές του OpenMP και χρησιμοποιήθηκαν ως ένας καινούριος τρόπος επιλογής της πολιτικής δρομολόγησης κατά την εκτέλεση του προγράμματος. Οι νέες μέθοδοι δοκιμάστηκαν και συγκρίθηκαν με τις υπάρχουσες ώστε να δοθεί μία εικόνα για τις επιδώσεις και τη χρησιμότητά τους.

ABSTRACT

Spyros Mantelos, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, June 2021.
Implementing new loop schedules in OpenMP.
Advisor: Vassilios Dimakopoulos, Associate Professor.

OpenMP has been establised as one of the most widespread parallel programming methods for shared memory systems. With the use of compiler directives, the programmer controls the behavior of the program. One of the choices the programmer has to make is the method by which the iterations of a parallel loop are divided and distributed across the available threads. This choice is important in order to achieve the fastest possible time by having a good load balance across the threads while maintaing the scheduling overhead as low as possible. OpenMP supports three methods, but in the literature there have been different approaches developed and proposed. In this thesis the infrastructure for integrading additional scheduling methods was created for the parallel compiler OMPi. Afterward, using this infrastructure four new politics were studied and implemented. In addition, the ability to add a tag for any OpenMP construct was created and used as a new way for selecting the scheduling method during the programs runtime. The new methods where tested and compared with the existing ones in order to gain an image for their speed and usefulness.

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Δημιουργία παράλληλων συστημάτων

Η ανάγκη των αρχαίων Βαβυλώνιων για εύκολους και γρήγορους υπολογισμούς, λόγω της ανάπτυξης του εμπορίου τους, τους οδήγησε στην δημιουργία του πρώτου υπολογιστή, του άβακα. Από τότε, ανάλογα με τις ανάγκες και τεχνολογικές δυνατότητες της κάθε εποχής έχουν υπάρξει πολλές ιδέες και μηχανισμοί που οδήγησαν σταδιακά στη δημιουργία των ηλεκτρονικών υπολογιστών με τη μορφή που έχουν σήμερα. Η αρχιτεκτονική τους, περιγράφηκε πρώτη φορά από τον von Neumann το 1945 και τον επόμενο χρόνο σχεδιάστηκε με βάση αυτή ο πρώτος ηλεκτρονικός υπολογιστής που ονομάστηκε ENIAC.

Το μοντέλο von Neumann, περιγράφει μία μονάδα επεξεργασίας που επικοινωνεί με ένα τμήμα μνήμης. Αρχικά ο επεξεργαστής λαμβάνει την επόμενη εντολή, την εκτελεί και όταν χρειαστεί τροποποιεί κάποια δεδομένα στη μνήμη. Η μνήμη, ο επεξεργαστής και η επικοινωνία μεταξύ τους έχουν βελτιωθεί σε τεράστιο βαθμό με τη

βελτίωση της τεχνολογίας εκτοξεύοντας τις δυνατότητες την υπολογιστών. Ο ENIAC καταλάμβανε έναν ολόκληρο όροφο και μπορούσε να εκτελέσει 5000 πράξεις το δευτερόλεπτο ενώ ένας σημερινός υπολογιστής μπορεί να κάνει δισεκατομμύρια πράξεις καταλαμβάνοντας ελάχιστο χώρο.

Το σημαντικότερο στοιχείο για τη βελτίωση της ταχύτητας ενός υπολογιστή, αποτελεί ο επεξεργαστής. Οι πρώτοι επεξεργαστές σε ηλεκτρονικούς υπολογιστές αποτελούνταν από λυχνίες κενού. Με την ανακάλυψη των τρανζίστορ, οι επεξεργαστές έγιναν γρηγορότεροι και ταυτόχρονα μικρότεροι σε μέγεθος. Επιπλέον μειώθηκε η κατανάλωση ενέργειας και αντιμετωπίστηκε σε μεγάλο βαθμό η υπερθέρμανσή τους. Τα τρανζίστορ συνέχισαν να μειώνονται σε μέγεθος και να αυξάνεται η συχνότητα λειτουργίας τους. Όμως η διαρκής αναβάθμιση των επεξεργαστών ήρθε αντιμέτωπη με κάποια προβλήματα. Αρχικά λόγω φυσικών περιορισμών τα τελευταία χρόνια έχει μειωθεί ο ρυθμός με τον οποίο αυξάνεται η ταχύτητά τους. Ακόμη, λόγω της αύξησης συχνοτήτων λειτουργίας και την ταυτόχρονη μείωση των διαστάσεων του, υπήρξε ραγδαία αύξηση της κατανάλωσης ενέργειας.

Η λύση για τη βελτίωση των επιδόσεων χωρίς την ολοκληρωτική αλλαγή του μοντέλου ήταν η προσθήκη περισσότερων επεξεργαστών σε έναν υπολογιστή. Έτσι δημιουργήθηκαν τα παράλληλα συστήματα, όπου πετυχαίνουμε καλύτερες επιδόσεις με τη χρήση απλών και δοκιμασμένων επεξεργαστών που δουλεύουν την ίδια στιγμή. Βέβαια, η απλή χρήση περισσότερων επεξεργαστών δεν είναι αρκετή. Τα σειριακά προγράμματα που γράφονταν κατά κύριο λόγο μέχρι τότε πρέπει να προσαρμοστούν ώστε να χρησιμοποιούν όλους τους επεξεργαστές όπου αυτό είναι δυνατό.

1.2 Είδη παράλληλων συστημάτων και ο προγραμματισμός τους

Ο προγραμματισμός των παράλληλων συστημάτων είναι πιο σύνθετος από τον σειριακό προγραμματισμό και διαφέρει ανάλογα με την αρχιτεκτονική του συστήματος. Σε ένα παράλληλο πρόγραμμα πρέπει οι επεξεργαστές να συντονιστούν, να γίνει διαμοιρασμός των δεδομένων και συλλογή των αποτελεσμάτων. Ανάλογα με τον τρόπο επικοινωνίας, το πλήθος και τις δυνατότητες των επεξεργαστών, υπάρχουν πολλές κατηγορίες παράλληλων συστημάτων. Μπορούμε να διαχωρίσουμε δύο αρχιτεκτονικές με βάση την οργάνωση της μνήμης: τα συστήματα κοινόχρηστης μνή-

μης και τα συστήματα κατανεμημένης μνήμης.

1.2.1 Κοινόχρηστη Μνήμη

Σε αυτή την κατηγορία έχουμε μία κοινή μνήμη στην οποία συνδέονται πολλαπλοί επεξεργαστές. Η επικοινωνία μεταξύ τους, επιτυγχάνεται με τη χρήση κοινόχρηστων μεταβλητών τις οποίες διαβάζουν ή μεταβάλλουν κατάλληλα οι επεξεργαστές. Καθώς η επικοινωνία γίνεται με τη χρήση ενός μέσου, συνήθως διαύλου, όταν προστεθεί μεγάλος αριθμός επεξεργαστών δημιουργείται συμφόρηση και καθυστέρηση της επικοινωνίας. Όμως, οι δυνατότητες που προσφέρουν με λίγους πυρήνες (σήμερα συνήθως 4 έως 12) είναι αρκετές για τους προσωπικούς υπολογιστές.

Τα προγράμματα γίνονται παράλληλα με τη χρήση διεργασιών ή νημάτων. Καθώς η μνήμη είναι κοινή ανάμεσα σε όλα τα νήματα, πρέπει να γίνουν οι απαραίτητες ενέργειές έτσι ώστε να μη δημιουργηθεί πρόβλημα σε καταστάσεις όπως η προσπάθεια να προσπελάσουν και να μεταβάλλουν ταυτόχρονα την ίδια διεύθυνση μνήμης πολλά νήματα. Ο προγραμματιστής είναι υπεύθυνος να εξασφαλίσει τον αμοιβαίο αποκλεισμό ανάμεσα στα νήματα στις περιοχές αυτές, ώστε ένα νήμα τη φορά να βρίσκεται εκεί. Η πολυπλοκότητα ενός προγράμματος όταν η δημιουργία των νημάτων και οι κλήσεις για τη διασφάλιση της ορθής του λειτουργίας γίνονται από τον προγραμματιστή, μπορεί να αυξηθεί αρκετά. Μία πλατφόρμα που αυτοματοποιεί αυτή τη διαδικασία είναι το OpenMP του θα μελετηθεί στη συνέχεια.

1.2.2 Κατανεμημένη Μνήμη

Από την άλλη υπάρχουν και παράλληλα συστήματα που αποτελούνται από πολλές ομάδες επεξεργαστή και μνήμης που συνδέονται μεταξύ τους ως ένα δίκτυο. Ένα τέτοιο σύστημα μπορεί να δημιουργηθεί από τη διασύνδεση πολλών προσωπικών υπολογιστών μεταξύ τους. Σε αντίθεση με τα συστήματα κοινόχρηστης μνήμης, δεν δημιουργούνται το ίδιο συχνά προβλήματα με την κλιμάκωση του συστήματος.

Σε αυτή την περίπτωση είναι αδύνατη η επικοινωνία μέσω κοινόχρηστων μεταβλητών, αφού ο κάθε επεξεργαστής έχει άμεση πρόσβαση στη δική του μόνο μνήμη. Συνεπώς η επικοινωνία γίνεται με ανταλλαγές μηνυμάτων μέσα στο δίκτυο που δημιουργείται με τη διασύνδεσή τους. Το πιο διαδεδομένο πρότυπο για τον προγραμματισμό τέτοιων συστημάτων είναι το MPI. Το MPI απλοποιεί σε έναν βαθμό

την ανταλλαγή μηνυμάτων ανάμεσα στους κόμβους του δικτύου ώστε να μην χρειαστεί ο προγραμματιστής να γνωρίζει λεπτομέρειες για τη μορφή του.

1.3 OpenMP

Όπως αναφέρθηκε προηγουμένως το OpenMP είναι ο πλέον διαδεδομένος τρόπος προγραμματισμού για συστήματα κοινόχρηστης μνήμης. Οι γλώσσες τις οποίες υποστηρίζει είναι η C, η C++ και η Fortran. Ένα σημαντικό πλεονέκτημά του είναι ότι ο τελικός κώδικας έχει μικρές διαφορές από το σειριακό πρόγραμμα και συνήθως μπορεί να εκτελεστεί και σειριακά.

Ο προγραμματιστής με τη χρήση οδηγιών προς τον μεταφραστή (directives) επιλέγει μία λειτουργία και της δίνει τις κατάλληλες παραμέτρους. Μία οδηγία αποτελείται από μία γραμμή που ξεκινάει με `#pragma` και έτσι δεν μπλέκεται με τον υπόλοιπο κώδικα. Συνεπώς η διαδικασία μετατροπής ενός προγράμματος σε παράλληλο είναι σε μεγάλο βαθμό ανεξάρτητη από τον κώδικα. Για τη χρήση του OpenMP, κατά τη μετάφραση του προγράμματος πρέπει να δηλωθεί η επιλογή `-fopenmp`, διαφορετικά οι οδηγίες αγνοούνται και το πρόγραμμα εκτελείται σειριακά.

1.4 Στόχοι και συνεισφορά της εργασίας

Η παρούσα εργασία είχε ως στόχο να εμπλουτίσει τον μεταφραστή OMPi ώστε να προσφέρει στους προγραμματιστές περισσότερες επιλογές σχετικά με τη δρομολόγηση των βρόχων του OpenMP. Επιπλέον, προστέθηκε η λειτουργία δημιουργίας ετικετών στο OpenMP ώστε να υπάρχει μεγαλύτερη ευελιξία στις επιλογές του προγραμματιστή κατά την εκτέλεση της εφαρμογής.

Η διπλωματική εργασία πραγματοποιήθηκε δημιουργώντας τις παραπάνω επεκτάσεις στον παραλληλοποιητικό μεταφραστή OMPi [1]. Ο OMPi είναι ένας μεταφραστής πηγαίου σε πηγαίο κώδικα για την γλώσσα C και διαθέτει ένα σύνθετο σύστημα χρόνου εκτέλεσης. Είναι δυνατή η χρήση διαφορετικών βιβλιοθηκών νημάτων, και προσφέρει ανταγωνιστικές επιδόσεις.

Αναπτύσσεται από την Ομάδα Παράλληλης Επεξεργασίας του Πανεπιστημίου

Ιωαννίνων, είναι ανοιχτού κώδικα και υποστηρίζει τη διεπαφή προγραμματισμού εφαρμογών OpenMP. Μεταφράζοντας ένα πρόγραμμα γραμμένο σε γλώσσα C, ο OMPi παράγει ένα μετασχηματισμένο πολυνηματικό πρόγραμμα. Έχει δοκιμαστεί σε ένα πλήθος μηχανημάτων Linux, Solaris, Irix και Windows, ενώ μεταφέρεται εύκολα σε οποιοδήποτε μηχάνημα υποστηρίζει νήματα POSIX.

1.4.1 Προσθήκη μεθόδων δρομολόγησης

Έχουν περιγραφεί πολλές διαφορετικές μέθοδοι δρομολόγησης των επαναλήψεων ενός βρόχου `for`. Με διαφορετικούς τρόπους η κάθε μία προσπαθεί να πετύχει την καλύτερη ισορροπία διαμοιρασμού ανάμεσα στα νήματα δαπανώντας όσο το δυνατό λιγότερο χρόνο για τη δρομολόγηση. Καθώς κάθε επεξεργαστής μπορεί να έχει διαφορετικό φόρτο και ταχύτητα, δεν υπάρχει κάποιος εύκολος τρόπος για να γίνει αυτό. Συνεπώς ανάλογα με την εφαρμογή, το σύστημα και τον τρόπο διαμοιρασμού των επαναλήψεων, εμφανίζονται διαφορετικές επιδόσεις.

Έτσι, πέρα από τις τρεις βασικές μεθόδους δρομολόγησης που προσφέρει το OpenMP (static, dynamic και guided) υπάρχουν προβλήματα που ένας διαφορετικός διαμοιρασμός είναι πιθανώς αποδοτικότερος. Με την παρούσα διπλωματική εργασία επιλέχθηκαν και υλοποιήθηκαν οι ακόλουθοι νέοι μέθοδοι:

- Trapezoid Self Scheduling
- Taper Method
- Fixed Size Chunking
- Factorial

Επιπλέον δημιουργήθηκε γενική υποδομή με την οποία μελλοντικά μπορούν πολύ εύκολα να υλοποιηθούν επιπλέον πολιτικές δρομολόγησης. Στη συνέχεια έγιναν κάποιες μετρήσεις για τις επιδόσεις όλων των δυνατών επιλογών και αξιολογήθηκαν τα αποτελέσματα.

Αναφέρθηκε ότι σε μία οδηγία του OpenMP ο προγραμματιστής μπορεί να περάσει επιπλέον επιλογές. Μία τέτοια επιλογή όταν παραλληλοποιείται ένας βρόχος `for` είναι το `schedule`. Με βάση αυτό επιλέγεται μία από τις τρεις βασικές μεθόδους δρομολόγησης του OpenMP.

Επιπλέον, το OpenMP προσφέρει δύο ακόμη επιλογές. Το `runtime` στο οποίο ο προγραμματιστής θα επιλέξει στη συνέχεια μία μέθοδο με μία μεταβλητή περιβάλλοντος και το `auto` με το οποίο επιλέγεται αυτόματα μία μέθοδος από την εκάστοτε υλοποίηση του OpenMP (πρόκειται για χαρακτηριστικό implementation-specific). Προσθέτοντας μία καινούρια μεταβλητή περιβάλλοντος όταν επιλεχθεί το `auto` στον OMPi, ο προγραμματιστής μπορεί να επιλέξει μία από τις νέες μεθόδους.

1.4.2 Προσθήκη ετικετών

Κατά τη διάρκεια της εργασίας διαπιστώθηκε ένας περιορισμός στον έλεγχο των επιλογών δρομολόγησης που έχει ένας προγραμματιστής OpenMP κατά την εκτέλεση του προγράμματος. Από εκεί δημιουργήθηκε η ιδέα της προσθήκης ενός νέου μηχανισμού με τον οποίο θα δίνονται περισσότερες επιλογές ελέγχου στον προγραμματιστή μέσω μεταβλητών περιβάλλοντος.

Ως αποτέλεσμα, σαν δεύτερο μέρος της διπλωματικής εργασίας, υλοποιήθηκε η δυνατότητα προσθήκης ετικετών στις οδηγίες του OpenMP. Στη συνέχεια με τις ετικέτες αυτές ο προγραμματιστής θα έχει τη δυνατότητα να θέτει διαφορετικές παραμέτρους για κάθε κομμάτι του προγράμματος που επιθυμεί. Σε πρώτο στάδιο οι ετικέτες χρησιμοποιήθηκαν για την επιλογή μεθόδου δρομολόγησης κάποιου βρόχου αλλά ο μηχανισμός είναι γενικός και μπορεί να εφαρμοστεί άμεσα και σε άλλες λειτουργίες του OpenMP.

1.5 Δομή της εργασίας/διάρθρωση του κειμένου

Η εργασία ακολουθεί την ακόλουθη δομή:

- Κεφάλαιο 2: Περιγραφή του OpenMP και του μεταφραστή OMPi.
- Κεφάλαιο 3: Παρουσίαση της δρομολόγησης στους βρόχους στο OpenMP και ανάλυση των νέων μεθόδων που υλοποιήθηκαν.
- Κεφάλαιο 4: Εμφάνιση των προβλημάτων που μας οδήγησαν στην υλοποίηση ετικετών στο OpenMP. Περιγραφή της υλοποίησης και του τρόπου χρήσης των ετικετών.

- Κεφάλαιο 5: Παράθεση των πειραματικών αποτελεσμάτων από τη χρήση των νέων μεθόδων σε εφαρμογές με τον OMPi.
- Κεφάλαιο 6: Σύνοψη της διπλωματικής εργασίας και συμπεράσματα.

ΚΕΦΑΛΑΙΟ 2

OpenMP και OMPi

2.1 Γενική εισαγωγή για το OpenMP

Το OpenMP [2] αποτελείται κυρίως από ένα σύνολο οδηγιών με τις οποίες ο προγραμματιστής δημιουργεί παράλληλες περιοχές και τις συντονίζει. Η χρησιμότητά του οφείλεται στο ότι η χρήση του είναι αρκετά απλή καθώς αναλαμβάνει τη δημιουργία και το συντονισμό των νημάτων αυτόματα με βάση τις οδηγίες του προγραμματιστή.

Από την πρώτη έκδοσή του το OpenMP έχει εξελιχθεί σε μεγάλο βαθμό. Σήμερα το σύνολο των λειτουργιών του περιλαμβάνει παράλληλες περιοχές, διαμοιρασμό εργασίας, περιβάλλον δεδομένων, συγχρονισμό και μεταβλητές περιβάλλοντος και συναρτήσεις. Η χρήση όλων αυτών δίνει τη δυνατότητα ελέγχου του προγράμματος σε μεγάλο βαθμό.

Οι *παράλληλες δομές ελέγχου* είναι υπεύθυνες για τον έλεγχο της ροής του προγράμματος. Με αυτές δημιουργούνται παράλληλες περιοχές με ομάδες νημάτων. Στο τέλος μίας παράλληλης περιοχής τα νήματα καταστρέφονται και η ροή συνεχίζει σαν ένα κανονικό σειριακό πρόγραμμα. Ακόμη, μία παράλληλη περιοχή μπορεί να περιέχει και άλλες παράλληλες περιοχές

Άλλο ένα μέρος του, είναι υπεύθυνο για τον *διαμοιρασμό της εργασίας ανάμεσα* στα νήματα. Εδώ ανήκει η δομή του `for` και οι μέθοδοι δρομολόγησης. Είναι σημαντικό να επιτευχθεί όσο το δυνατό καλύτερος διαμοιρασμός ώστε να μην έχουμε νήματα που βρίσκονται σε αδράνεια ενώ άλλα δουλεύουν. Από την άλλη, ο διαχωρισμός της εργασίας στα νήματα έχει ένα κόστος που πρέπει να διατηρηθεί χαμηλό.

Το *περιβάλλον δεδομένων* αναλαμβάνει τη σωστή διαχείριση των μεταβλητών από τα νήματα. Με αυτό καθορίζονται οι κοινές μεταβλητές και οι ιδιωτικές μεταβλητές του κάθε νήματος ώστε να αποφευχθούν προβλήματα από τη χρήση των ίδιων διευθύνσεων μνήμης από όλα τα νήματα.

Ένα άλλο τμήμα του είναι το κομμάτι του *συγχρονισμού*, το οποίο είναι υπεύθυνο για τον συντονισμό των νημάτων. Περιλαμβάνει κρίσιμες περιοχές και φράγματα. Είναι απαραίτητο για να εξασφαλιστεί ότι κάποιες λειτουργίες γίνονται από ένα νήμα την φορά, ή ότι τα νήματα θα βρεθούν ταυτόχρονα σε κάποιο σημείο του προγράμματος πριν ξεκινήσουν να εκτελούν το κώδικα που ακολουθεί.

Τέλος οι συναρτήσεις χρόνου εκτέλεσης και οι μεταβλητές περιβάλλοντος αποτελούν ένα σύνολο τρόπων με τους οποίους ο προγραμματιστής ελέγχει τη λειτουργία του προγράμματος κατά την εκτέλεσή του και όχι κατά τη μετάφραση. Για παράδειγμα, μπορεί να ορίσει τον αριθμό των νημάτων ή τη μέθοδο δρομολόγησης για τους βρόχους του προγράμματος.

2.2 Προγραμματισμός με το OpenMP

Μία οδηγία (directive) του OpenMP ξεκινά με το πρόθεμα `#pragma omp` και είναι ο βασικός μηχανισμός για τον έλεγχο της συμπεριφοράς ενός προγράμματος που το χρησιμοποιεί. Στη συνέχεια ακολουθεί ένα όνομα που σχετίζεται με τη λειτουργία την οποία ο προγραμματιστής θέλει να χρησιμοποιήσει και μία σειρά από φράσεις (clauses) που ελέγχουν πρόσθετες επιλογές. Μία οδηγία δεν μπορεί να περιέχει άλλον κώδικα και ολοκληρώνεται με αλλαγή γραμμής. Έτσι η γενική σύνταξή τους είναι η εξής:

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

Οι φράσεις μπορεί να εμφανίζονται με οποιαδήποτε σειρά. Στη συνέχεια παρουσιάζονται οι σημαντικότερες οδηγίες και φράσεις που σχετίζονται με το αντικείμενο της διπλωματικής αυτής.

Η οδηγία parallel

Με τη χρήση αυτής της οδηγίας δημιουργείται μία παράλληλη περιοχή. Όταν ένα νήμα τη συναντήσει, δημιουργεί μία ομάδα νημάτων όπου κάθε ένα από αυτά θα εκτελέσει τον κώδικα της παράλληλης περιοχής συμπεριλαμβανομένου και του αρχικού νήματος. Μόλις ολοκληρωθεί η εκτέλεση σε κάθε νήμα, καταστρέφονται όλα εκτός του αρχικού που συνεχίζει τη λειτουργία του. Κάθε νήμα προσδιορίζεται από ένα αναγνωριστικό αριθμό που μπορεί να χρησιμοποιηθεί για τον συντονισμό ή τον διαμοιρασμό εργασίας.

Σε κάθε νήμα, μπορεί να παρουσιάζονται διαφορετικές συνθήκες που θα καθορίσουν τον χρόνο που θα χρειαστεί για να εκτελέσει τον φόρτο που του έχει ανατεθεί. Είναι πιθανό τα νήματα να εκτελούνται σε πυρήνες με διαφορετικές ταχύτητες και φόρτο από άλλες διεργασίες. Ακόμη τα νήματα μπορεί να μην έχουν ισορροπημένο φόρτο μεταξύ τους. Σε πολλές εφαρμογές η ορθότητα του κώδικα που ακολουθεί την παράλληλη περιοχή προαπαιτεί ότι όλα τα νήματα έχουν ολοκληρώσει τη λειτουργία τους πριν εκτελεστεί κάποια άλλη εντολή. Στο τέλος της παράλληλης περιοχής δημιουργείται ένα φράγμα στο οποίο τα νήματα περιμένουν μέχρι να ολοκληρώσουν όλα την παράλληλη περιοχή. Είναι επίσης δυνατό να υπάρχουν εμφωλευμένες παράλληλες περιοχές.

Με τη φράση `num_threads` δηλώνεται ο επιθυμητός αριθμός νημάτων. Υπάρχουν και άλλοι παράγοντες που επηρεάζουν τον πραγματικό αριθμό των νημάτων όπως αν είναι ενεργοποιημένη η δυναμική προσαρμογή μεγέθους στις ομάδες νημάτων ή αν έχει τεθεί άνω όριο στα συνολικά νήματα. Αν δεν υπάρχει αυτή η φράση, ελέγχεται αν δηλώθηκε αριθμός νημάτων με κάποιον άλλο τρόπο κατά την εκτέλεση του προγράμματος. Διαφορετικά ανάλογα με την υλοποίηση εμφανίζονται διαφορετικές συμπεριφορές.

Όταν δημιουργούνται νέα νήματα, έχουν πρόσβαση σε όλες τις μεταβλητές που έχουν δημιουργηθεί πριν την παράλληλη περιοχή, από προεπιλογή. Για τον έλεγχο των χαρακτηριστικών κοινοχρησίας υπάρχουν οι ακόλουθες τρεις φράσεις που δέχονται ως όρισμα μία λίστα μεταβλητών:

- *shared*: Οι μεταβλητές γίνονται κοινόχρηστες ανάμεσα σε όλα τα νήματα.
- *private*: Σε κάθε νήμα δημιουργούνται ιδιωτικές μεταβλητές στις οποίες έχει μόνο αυτό πρόσβαση

- *firstprivate*: Ίδια λειτουργία με το private σε συνδυασμό με αρχικοποίηση των ιδιωτικών μεταβλητών στις τιμές που έχουν στο αρχικό νήμα.

Η οδηγία for

Σε ένα σειριακό πρόγραμμα, οι βρόχοι επαναλήψεων `for` αποτελούν συνήθως σημεία όπου αφιερώνεται σημαντικός χρόνος εκτέλεσης, και επομένως είναι κρίσιμο να παραλληλοποιηθούν. Η οδηγία `for` διανέμει ανάμεσα στα νήματα τις επαναλήψεις του βρόχου `for` που την ακολουθεί υποχρεωτικά. Με την φράση `schedule` δίνεται η δυνατότητα επιλογής της μεθόδου δρομολόγησης. Η χρήση της φράσης αυτής αναλύεται στο επόμενο κεφάλαιο.

Στον Κώδικα 2.1 δίνεται ένα παράδειγμα ενός παράλληλου προγράμματος για τον υπολογισμό της μέσης τιμής ενός πίνακα με αριθμούς με τη χρήση του OpenMP. Στη γραμμή 10 με την οδηγία `#pragma omp parallel` δημιουργείται μία παράλληλη περιοχή. Επιπλέον δίνεται η φράση `num_threads(4)` με την οποία δηλώνεται ότι ο προγραμματιστής επιθυμεί να δημιουργηθούν τέσσερα νήματα για αυτήν την παράλληλη περιοχή. Τέλος, με την οδηγία `reduction(+: sum)` σε κάθε νήμα δημιουργείται ιδιωτική μεταβλητή `sum` στην οποία γίνονται οι πράξεις και μόλις ολοκληρωθεί η παράλληλη περιοχή, οι τιμές όλων των μεταβλητών `sum` προστίθενται στην αρχική μεταβλητή.

Στη γραμμή 12 καλείται μία δεύτερη οδηγία του OpenMP, η οδηγία διαμοιρασμού εργασιών `for`. Αυτή αναλαμβάνει να διαχωρίσει ανάμεσα στα νήματα τις επαναλήψεις του βρόχου που ακολουθεί στη γραμμή 13. Εδώ χρησιμοποιείται μία φράση, η `schedule`, όπου θέτει την μέθοδο διαχωρισμού στην επιλογή `auto`.

2.2.1 Μεταβλητές περιβάλλοντος

Πολλές από τις επιλογές του OpenMP που γίνονται με φράσεις που αναφέραμε μπορούν να γίνουν και με μεταβλητές περιβάλλοντος. Αυτό έχει το πλεονέκτημα ότι το πρόγραμμα μπορεί να μεταφραστεί μία μόνο φορά και να χρησιμοποιηθεί με διαφορετικές παραμέτρους στη συνέχεια. Ένα παράδειγμα είναι η μεταβλητή `OMP_NUM_THREADS` με την οποία καθορίζεται ο αριθμός των νημάτων που δημιουργούνται στις παράλληλες περιοχές που δεν έχουν τη φράση `num_threads`. Ακόμη υπάρχει η μεταβλητή `OMP_SCHEDULE` με την οποία θέτουμε την μέθοδο δρομολόγησης για όσους βρόχους έχει δηλωθεί ότι η μέθοδος θα επιλεχθεί κατά την εκτέλεση. Όταν ένα πρό-

Κώδικας 2.1: Παράλληλος υπολογισμός μέσης τιμής

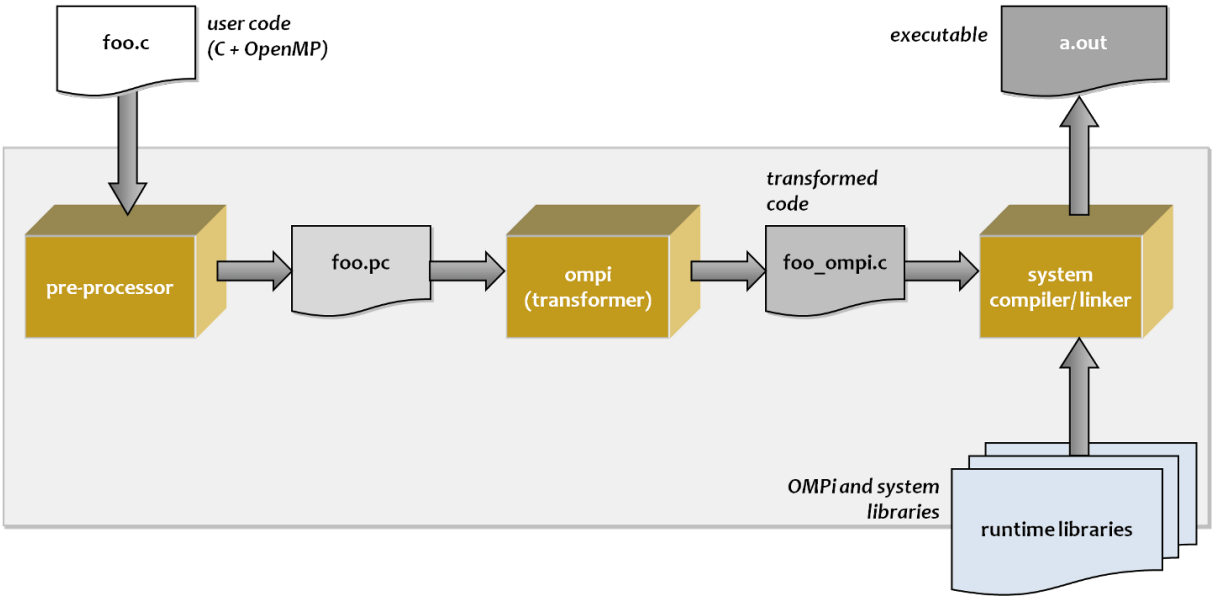
```
#include <stdio.h>

#define N 11
int numbers[] = {1,8,6,8,3,7,11,4,9,10};

void main(){
  int i;
  int sum = 0;

  #pragma omp parallel reduction(+: sum) num_threads(4)
  {
    #pragma omp for schedule(auto)
    for (i = 0; i < N; ++i)
      sum+= numbers[i];
  }
  printf("Mean = %.2f\n", (float)sum/N);
}
```

γραμμα εκτελείται με το OpenMP οι μεταβλητές αυτές διαβάζονται πριν ξεκινήσει η εκτέλεση της εφαρμογής και οι πληροφορίες τους αποθηκεύονται.

2.3 OMPi

Ο OMPi είναι ένας παραλληλοποιητικός μεταφραστής για την γλώσσα C στον οποίο παρέχεται το πρότυπο OpenMP. Έχει δύο τμήματα, τον μεταφραστή και τη βιβλιοθήκη χρόνου εκτέλεσης. Κατά τη μετάφραση ενός προγράμματος, πραγματοποιούνται η λεκτική και η συντακτική ανάλυση και δημιουργείται το συντακτικό δέντρο του αρχικού κώδικα. Στη συνέχεια προσπελάζεται κάθε κόμβος του δέντρου και γίνονται οι απαραίτητοι μετασχηματισμοί ώστε οι οδηγίες του OpenMP να αντιστοιχιστούν σε κώδικα. Μετά από αυτό το στάδιο, παράγεται νέος κώδικας σε γλώσσα C ο οποίος δεν περιέχει πλέον οδηγίες του OpenMP.

Ο νέος κώδικας χρησιμοποιεί τη βιβλιοθήκη χρόνου εκτέλεσης στα σημεία που χρειάζεται. Με αυτή δημιουργούνται νήματα, τοποθετούνται κλειδαριές και φράγματα, διαχωρίζονται οι επαναλήψεις ενός βρόχου, και γενικότερα αναλαμβάνει την



Σχήμα 2.1: Μετάφραση προγράμματος από τον OMPi

εκτέλεση του κώδικα παράλληλα. Ο κώδικας αυτός στη συνέχεια μεταφράζεται από κάποιον απλό (μη παραλληλοποιητικό) μεταφραστή C που βρίσκεται στο σύστημα όπως ο gcc. Η διαδικασία που περιγράφεται παραπάνω φαίνεται και στο Σχήμα 2.1.

Τέλος ένα παράδειγμα του κώδικα που παράγεται από τον OMPi με τη μετάφραση του προγράμματος 2.1 παραθέτεται στο παράρτημα Α.

ΚΕΦΑΛΑΙΟ 3

Δρομολόγηση στους βρόχους

3.1 Πρότυπο του OpenMP

Στο προηγούμενο κεφάλαιο έγινε μία εισαγωγή για το OpenMP και κάποιες λειτουργίες του. Μία από τις οδηγίες που παρουσιάστηκαν είναι η οδηγία `for`, που προαιρετικά μπορεί να δεχθεί την φράση `schedule`:

```
#pragma omp for schedule(...)
    for ( ... )
        ...
```

Η οδηγία αυτή αναλαμβάνει να χωρίσει ανάμεσα στα νήματα τις επαναλήψεις του βρόχου που την ακολουθεί. Ο τρόπος με τον οποίο γίνεται ο διαχωρισμός, ελέγχεται με τη φράση `schedule`.

3.1.1 Η φράση schedule

Η επιλογή του `schedule` καθορίζει τον τρόπο με τον οποίο οι επαναλήψεις ενός βρόχου χωρίζονται σε υποσύνολα που ονομάζονται *chunks* και διανέμονται ανάμεσα στα νήματα. Η φράση `schedule` έχει την ακόλουθη σύνταξη:

```
schedule(<modifier><kind>,<chunk_size>)
```

Το <kind> μπορεί να είναι μία από τις ακόλουθες επιλογές: `static`, `dynamic`, `guided`, `runtime` και `auto`. Οι τρεις πρώτες αποτελούν τις μεθόδους που προσφέρονται από το OpenMP. Η επιλογή `runtime`, δηλώνει ότι κάποια μέθοδος θα επιλεχθεί στη συνέχεια κατά την εκτέλεση του προγράμματος. Αυτό γίνεται με βάση την τιμή μίας μεταβλητής περιβάλλοντος που αναφέρεται στην υποενότητα 3.1.2. Τέλος η επιλογή `auto` έχει την σημασία της αυτόματης επιλογής μεθόδου δρομολόγησης και εξαρτάται από την υλοποίηση.

Το <chunk_size> είναι ένας ακέραιος αριθμός που δίνει προαιρετικά ως όρισμα ο προγραμματιστής. Στις μεθόδους `static` και `dynamic`, δηλώνει το μέγεθος των κομματιών που θα δίνονται κάθε φορά σε ένα νήμα. Στην μέθοδο `guided` δηλώνει το ελάχιστο μέγεθος που μπορούν να έχουν τα κομμάτια αυτά. Φυσικά αυτός ο κανόνας πιθανώς δεν ισχύει για το τελευταίο κομμάτι ενός βρόχου.

Τέλος ως modifier υπάρχουν δύο επιλογές που προστέθηκαν σε πρόσφατη έκδοση του OpenMP. Οι δύο modifiers είναι οι `monotonic` και `nonmonotonic`. Διαχωρίζονται από την μέθοδο με την χρήση άνω κάτω τελείας και μπορούν να χρησιμοποιηθούν μόνο στις μεθόδους `dynamic` ή `guided`. Αν δεν δηλωθεί modifier θεωρείται ως προεπιλεγμένος ο `nonmonotonic`.

Αν επιλέξουμε το `schedule static`, οι επαναλήψεις χωρίζονται σε ισομεγέθη chunks που διανέμονται κυκλικά σε όλα τα νήματα με βάση το id τους. Το βασικό πλεονέκτημα της μεθόδου, είναι ότι ο διαχωρισμός των επαναλήψεων και η ανάθεση τους είναι προκαθορισμένη και γίνεται άμεσα και έτσι έχει ελάχιστο κόστος. Όμως, αυτό συνεπάγεται κακή ισορροπία ανάμεσα στα νήματα, καθώς αν δεν επικρατούν ίδιες συνθήκες και ταχύτητες ανάμεσά τους ή αν οι επαναλήψεις του βρόχου δεν έχουν το ίδιο υπολογιστικό φορτίο, τα πιο γρήγορα θα παραμένουν αδρανή περιμένοντας τα υπόλοιπα.

Όταν έχουμε `dynamic schedule`, το κάθε νήμα αναλαμβάνει ένα chunk και ζητάει να του ανατεθεί επόμενο μόλις τελειώσει, μέχρι να ολοκληρωθούν όλες οι επαναλήψεις. Έχει τον αντίστροφο στόχο από την στατική μέθοδο. Λόγω των συνεχών αναθέσεων, το κόστος δρομολόγησης είναι αυξημένο αλλά τα νήματα λαμβάνουν πιο ισορροπημένο φορτίο ανάλογο και με τις ταχύτητές τους. Ο χρόνος που κάποια νήματα παραμένουν αδρανή είναι ελάχιστος στις περισσότερες περιπτώσεις.

Στην περίπτωση που επιλεχθεί το `guided schedule`, ισχύει η ίδια λογική με το `dynamic`, αλλά με chunks που μειώνονται σε μέγεθος. Αν υποθέσουμε ότι σε μια

δεδομένη χρονική στιγμή μένουν να δρομολογηθούν Ri επαναλήψεις και P είναι το πλήθος των νημάτων στην παράλληλη ομάδα, τότε το επόμενο chunk που θα δώσει σε κάποιο νήμα η μέθοδος guided είναι ίσο με $\frac{Ri}{P}$ επαναλήψεις. Δηλαδή κάθε φορά ανατίθεται ένα chunk μεγέθους ίσο με τις εναπομείνασες επαναλήψεις δια τον αριθμό των νημάτων. Η μέθοδος αυτή επιχειρεί να εκμεταλλευτεί τα θετικά και των δύο παραπάνω μεθόδων. Δηλαδή ανάθεση λιγότερων chunks αλλά προσπαθώντας να διατηρηθεί ισορροπία ανάμεσα στα νήματα. Όμως αυτό σημαίνει ότι ο υπολογισμός του επόμενου μεγέθους chunk χρειάζεται περισσότερο χρόνο.

3.1.2 Μεταβλητή περιβάλλοντος OMP_SCHEDULE

Όταν δηλωθεί η επιλογή `runtime`, σημαίνει ότι η μέθοδος θα επιλεχθεί από μία μεταβλητή περιβάλλοντος. Κατά την έναρξη του προγράμματος διαβάζονται όλες οι μεταβλητές που μπορεί να χρειαστεί το πρόγραμμα και αποθηκεύεται η σχετική πληροφορία. Στη συνέχεια όταν εμφανιστεί κάποιο σημείο που χρειάζεται αυτή την πληροφορία, διαβάζει τα δεδομένα που αποθήκευσε και εκτελεί τις απαραίτητες ενέργειες.

Η μεταβλητή περιβάλλοντος `OMP_SCHEDULE` ακολουθεί την ίδια σύνταξη που έχει και η φράση `schedule`. Για τον καθορισμό μίας μεταβλητής περιβάλλοντος, καλούμε την ακόλουθη εντολή:

```
export OMP_SCHEDULE="dynamic,5"
```

Στην περίπτωση που η τιμή της μεταβλητής είναι κάτι απροσδιόριστο ή δεν υπάρχει η μεταβλητή, επιλέγεται η περίπτωση `auto`.

3.2 Χειρισμός από τον OMPi

Ο OMPi ακολουθεί την ίδια σύνταξη που περιγράφηκε και για την φράση `schedule` και για την μεταβλητή περιβάλλοντος `OMP_SCHEDULE`, αλλά δεν υποστηρίζει τους schedule modifiers. Έτσι αν βρει έναν θα τον διαβάσει και θα τον αγνοήσει. Ως προεπιλεγμένη μέθοδος αν δεν υπάρχει το `schedule clause` επιλέγεται η `static`. Στην περίπτωση που δηλωθεί η επιλογή `runtime` αλλά δεν βρεθεί κάποια άλλη μέθοδος στην μεταβλητή περιβάλλοντος, επιλέγεται το `auto`. Αυτό με τη σειρά του οδηγεί στην μέθοδο `static`.

Κώδικας 3.1: Απόσπασμα μετασχηματισμένου προγράμματος με μέθοδο static

```
1  niters_ = (long) ((((10) + 1) >= (0)) ? (((10) + 1) - (0)) : 0);
2  ort_entering_for(0, 0);
3  if (ort_get_static_default_chunk(niters_, &fiter_, &liter_))
4  {
5    for(iter_ = fiter_, i = (0) + fiter_ * 1; iter_ < liter_; iter_++, i+=1)
6      ...
```

Κώδικας 3.2: Απόσπασμα μετασχηματισμένου προγράμματος με μέθοδο guided

```
1  niters_ = (long) ((((10) + 1) >= (0)) ? (((10) + 1) - (0)) : 0);
2  ort_entering_for(0, 0);
3  while (ort_get_guided_chunk(niters_, 1, 1, &fiter_, &liter_, (int *) 0))
4  {
5    for(iter_ = fiter_, i = (0) + fiter_ * 1; iter_ < liter_; iter_++, i+=1)
6      ...
```

Ο κώδικας που δημιουργείται πριν τον βρόχο έχει τρεις διαφορετικές μορφές ανάλογα με την επιλογή μεθόδου. Στην περίπτωση της επιλογής `static`, πριν τον βρόχο υπάρχει μία συνθήκη `if` στην οποία ελέγχεται αν δόθηκε κάποιο chunk για εκτέλεση ή όχι. Αν η μέθοδος είναι `dynamic` ή `guided`, πρέπει η ανάθεση των chunks και ο έλεγχος να γίνονται επαναληπτικά, οπότε η `if` μετατρέπεται σε `while`. Στην περίπτωση του `runtime` η συνθήκη της `while` περιέχει έναν δείκτη σε μία συνάρτηση. Στον δείκτη έχει δοθεί τιμή μέσω μία συνάρτησης η οποία ελέγχει την επιλεγμένη μέθοδο για το `runtime` και δίνει την κατάλληλη συνάρτηση ως τιμή στον δείκτη. Οι πρώτες δύο περιπτώσεις παρουσιάζονται στα αποσπάσματα κώδικα 3.1 και 3.2 αντίστοιχα. Η λογική της τρίτης περίπτωση εμφανίζεται στις γραμμές 55-64 του παραρτήματος Α με τη διαφορά ότι η επιλογή στο παράρτημα είναι auto.

Η επιλογή `auto` διαμορφώθηκε έτσι ώστε να έχει παρόμοια λειτουργία με την επιλογή `runtime`. Η κυριότερη διαφορά τους είναι ότι η μεταβλητή περιβάλλοντος του `auto` έχει διαφορετική σύνταξη για να υποστηρίζει τις νέες μεθόδους. Αντί να περιμένει να βρει τη μέθοδο με μία προαιρετική τιμή το chunksize, περιέχει μέσα σε παρενθέσεις μία σειρά από εισόδους που προσδιορίζονται με ένα γράμμα ως ετικέτα της κάθε μίας.

3.2.1 Διαμοιρασμός επαναλήψεων

Η βασική διαδικασία που πρέπει να κάνει κάθε μέθοδος είναι να διανέμει στα νήματα τις επαναλήψεις που πρέπει να εκτελέσουν. Για τη σωστή λειτουργία του προγράμματος, είναι απαραίτητο να εκτελεστούν όλες οι επαναλήψεις και να μην υπάρχει καμία που θα εκτελεστεί πάνω από μία φορά.

Για τον σκοπό αυτό ο OMPi αποθηκεύει μία μεταβλητή που δηλώνει την επόμενη επανάληψη που πρέπει να εκτελεστεί. Όταν ένα νήμα καλεί τη συνάρτηση που θα του αναθέσει το επόμενο chunk, δίνει ως όρισμα δύο δείκτες. Ο πρώτος αποτελεί την πρώτη επανάληψη και ο δεύτερος την τελευταία. Η συνάρτηση που κλήθηκε πρέπει να αναθέσει τιμές σε αυτούς τους δείκτες αναλαμβάνοντας τον αμοιβαίο αποκλεισμό των νημάτων. Αυτό μπορεί να επιτευχθεί με δύο τρόπους. Είτε με τη χρήση κλειδαριών, είτε με τη χρήση των atomics αν υπάρχει η δυνατότητα.

Η χρήση κλειδαριών είναι απλή. Το σημείο στο οποίο ανατίθενται επαναλήψεις σε ένα νήμα περικλείεται με κλειδαριές. Στον δείκτη για την αρχή του chunk δίνεται η τιμή που αποθηκεύει την επόμενη επανάληψη. Σε αυτή προστίθεται το μέγεθος του chunk και η κλειδαριά ξεκλειδώνει. Η ανάθεση τιμής στον δεύτερο δείκτη δεν είναι απαραίτητο να δίνει σε περιβάλλον αμοιβαίου αποκλεισμού. Το πρόβλημα με τη χρήση των κλειδαριών είναι ότι εισάγουν καθυστέρηση για τον συντονισμό των νημάτων.

Έτσι στις περιπτώσεις που μπορούν να χρησιμοποιηθούν τα atomics στο σύστημα που εκτελείται ο κώδικας, αποτελούν καλύτερη επιλογή αφού δεν υπάρχει η ανάγκη συντονισμού των νημάτων. Τα atomics, αποτελούν εντολές που πραγματοποιούνται με μία εντολή assembly. Συνεπώς δεν υπάρχει η πιθανότητα να έχουμε συγκρούσεις ή ταυτόχρονη προσπέλαση ανάμεσα στα νήματα. Στον κώδικα για τον διαμοιρασμό των επαναλήψεων χρησιμοποιούνται δύο εντολές. Η πρώτη ονομάζεται *fetch and add* και η δεύτερη *compare and swap*. Όπως δηλώνει και το όνομά της, η πρώτη εντολή επιστρέφει την τιμή μίας μεταβλητής και μετά της προσθέτει την τιμή που της δίνουμε. Η χρήση της είναι κατάλληλη για μεθόδους που έχουμε ένα σταθερό μέγεθος chunk και δεν χρειάζεται να το υπολογίσουμε με βάση τις επαναλήψεις που απομένουν.

Σε κάποιες μεθόδους όμως, το μέγεθος του chunk εξαρτάται από τις επαναλήψεις που απομένουν. Στην περίπτωση που δύο νήματα υπολογίσουν ταυτόχρονα το μέγεθος και καλέσουν την `fetch and add` στη συνέχεια -αν και δεν θα υπάρχουν συ-

γκρούσεις- ένα από τα δύο θα έχει υπολογίσει λάθος τιμή, καθώς μετά την εκτέλεση της πρώτης κλήσης, έχουν απομείνει λιγότερες επαναλήψεις. Η μέθοδος `compare and swap` δίνει λύση σε αυτό το πρόβλημα. Η μέθοδος συγκρίνει την τιμή μία μεταβλητής με μία άλλη τιμή και αν είναι ίσες αλλάζει την τιμή της μεταβλητής με μία άλλη που δίνεται σαν όρισμα. Σε συνδυασμό με έναν βρόχο `do-while` σε περίπτωση που η τιμή της αρχικής επανάληψης έχει αλλάξει μέχρι να κληθεί η `compare and swap`, οι υπολογισμοί γίνονται και πάλι από την αρχή.

3.3 Υλοποίηση νέων μεθόδων

Όπως είπαμε και στην αρχή, υπάρχουν πολλές μέθοδοι δρομολόγησης που έχουν προταθεί στη βιβλιογραφία και όχι μόνο οι 3 που προδιαγράφονται από το OpenMP. Προκειμένου να εντάξουμε στο OpenMP επιπλέον μεθόδους δρομολόγησης θα πρέπει να βρεθεί ένας τρόπος ώστε να μην παραβαίνουμε τη σύνταξη και τους κανόνες του OpenMP. Αφού οι προδιαγραφές του OpenMP δεν προσδιορίζουν την συμπεριφορά της επιλογής `auto` μπορεί να διαμορφωθεί ώστε να χρησιμοποιεί τις νέες μεθόδους.

Για τη χρήση των νέων μεθόδων, δημιουργήσαμε μία νέα μεταβλητή περιβάλλοντος η οποία σχετίζεται με την επιλογή `auto`. Έτσι για να χρησιμοποιηθεί η τιμή της πρέπει να φτάσουμε σε μία κατάσταση που έχει επιλεχθεί το `schedule auto`. Στη συνέχεια, αν υπάρχει η μεταβλητή αυτή καθορίζει ποια μέθοδος θα χρησιμοποιηθεί είτε από τις προϋπάρχουσες είτε από τις νέες μεθόδους.

Μία βασική διαφορά που έχουν οι νέες μέθοδοι είναι ότι πολλές από αυτές δέχονται αρκετές εισόδους, ενώ στις προϋπάρχουσες το chunksize ήταν αρκετό. Έτσι αντί να έχουμε μία σειρά από αριθμούς χωρισμένους με κόμματα, για κάθε είσοδο δίνεται και ένα γράμμα ως ετικέτα. Με αυτόν τον τρόπο αρχικά οι είσοδοι μπορούν να δοθούν με οποιαδήποτε σειρά και έτσι διευκολύνεται η δουλειά του προγραμματιστή. Επιπλέον, σε μεθόδους που υπάρχουν κάποιες είσοδοι που είναι προαιρετικές, με τη χρήση ετικετών για κάθε μία είναι δυνατό να δοθούν όσες και όποιες από τις προαιρετικές εισόδους επιθυμεί ο προγραμματιστής.

Συνεπώς όταν επιλεχθεί μία νέα μέθοδος χρειάζεται να αποθηκευτούν και οι είσοδοί της ώστε να χρησιμοποιηθούν στη συνέχεια για τους υπολογισμούς που είναι απαραίτητοι. Επίσης, υπάρχει η περίπτωση σε κάποιον βρόχο να επιλεχθεί `schedule`

Πίνακας 3.1: Συχνά Σύμβολα

Σύμβολο	Σημασία
P	Ο συνολικός αριθμός των νημάτων
N	Οι συνολικές επαναλήψεις ενός βρόχου
R	Οι εναπομείνασες επαναλήψεις ενός βρόχου
h	Το κόστος διαμοιρασμού των επαναλήψεων (overhead)
μ	Η μέση τιμή των χρόνων εκτέλεσης των επαναλήψεων σε μs
σ	Η τυπική απόκλιση των χρόνων εκτέλεσης των επαναλήψεων σε μs
Cs	Μέγεθος κομματιού που θα ανατεθεί (chunk size)

`runtime` ενώ σε έναν άλλο `auto`. Πρέπει λοιπόν να υπάρχει μία δομή που αποθηκεύει τα δεδομένα για τη δρομολόγηση των βρόχων που έχουν την επιλογή `auto`. Για αυτό τον σκοπό δημιουργήθηκε ένα νέο `struct` το οποίο αποθηκεύει της πληροφορίες της μεταβλητής περιβάλλοντος που θα διαβαστεί στην αρχή του προγράμματος.

Στις μεθόδους που ακολουθούν, παρουσιάζονται κάποιοι μαθηματικοί τύποι που περιγράφουν τον τρόπο με τον οποίο υπολογίζεται το μέγεθος των chunks που ανατθέτουν. Στον πίνακα 3.1 εμφανίζονται τα σύμβολα που εμπλέκονται και η σημασία τους.

3.3.1 Trapezoid Self Scheduling

Η πρώτη καινούρια μέθοδος που υλοποιήθηκε ονομάζεται Trapezoid Self Scheduling [3]. Έχει την ίδια λογική με τη μέθοδο guided, δηλαδή αναθέτει chunks με μέγεθος που μειώνεται. Ο στόχος της είναι να αξιοποιήσει τα πλεονεκτήματα της μεθόδου guided αλλά χρησιμοποιώντας γραμμική συνάρτηση για τη μείωση του πλήθους των επαναλήψεων που αναθέτει κάθε φορά είναι απλούστερος και ταχύτερος ο υπολογισμός του. Συνεπώς επιχειρεί να πετύχει καλή ισορροπία φόρτου ανάμεσα στα νήματα, με μικρό κόστος δρομολόγησης. Το κάθε chunk υπολογίζεται με βάση τις ακόλουθες εξισώσεις:

$$A = \left\lceil \frac{2N}{f+l} \right\rceil$$

$$\delta = \frac{f-l}{A-1}$$

$$Cs(1) = f$$

$$Cs(t) = Cs(t-1) - \delta$$

Ως εισόδους δέχεται δύο αριθμούς f και l (αρχικά των first και last) που αντιστοιχούν στο μέγεθος του πρώτου και του τελευταίου chunk αντίστοιχα. Σε περίπτωση που δεν δοθεί η είσοδος f, ως προεπιλογή υπολογίζεται η τιμή $f = \frac{N}{2P}$ η οποία προτείνεται σύμφωνα με το [3], ενώ αν δεν δοθεί η τιμή l θεωρούμε ότι είναι 1.

Στην υλοποίηση της μεθόδου στον OMPi, αρχικά ελέγχεται αν έχει δοθεί η τιμή f. Αν δεν έχει δοθεί υπολογίζεται η προεπιλεγμένη τιμή. Μετά υπολογίζονται οι τιμές A και δ σύμφωνα με τους παραπάνω τύπους. Για τον υπολογισμό του επόμενου μεγέθους, αρκεί να κρατήσουμε έναν μετρητή i για τα πόσα chunks έχουν ήδη ανατεθεί. Με βάση αυτόν υπολογίζουμε το μέγεθος ως $Cs = f - i \times \delta$. Στη συνέχεια η τιμή του i αυξάνεται για τον υπολογισμό του επόμενου μεγέθους.

3.3.2 Taper

Μία ακόμα μέθοδος παρόμοια με την guided είναι η Taper [4]. Αυτή επιχειρεί να δημιουργήσει καλή ισορροπία φόρτου χρησιμοποιώντας όσο το δυνατόν μεγαλύτερο μέγεθος chunk. Συνεπώς μειώνονται οι αναθέσεις και δημιουργείται χαμηλό κόστος δρομολόγησης. Για τον υπολογισμό του μεγέθους που χρησιμοποιεί είναι απαραίτητη η μέση τιμή και η τυπική απόκλιση των χρόνων που απαιτούν οι επαναλήψεις για να εκτελεστούν. Με βάση αυτές, το μέγεθος ενός chunk υπολογίζεται από τους ακόλουθους τύπους:

$$Cs_i = max\left\{Cs_{min}, \left\lceil T_i + \frac{u_a^2}{2} - u_a\sqrt{2T_i + \frac{u_a^2}{4}} \right\rceil\right\}$$

$$T_i = \frac{Ri}{P}$$

$$u_a = \frac{\alpha\sigma}{\mu}$$

Δύο ακόμα είσοδοι που είναι προαιρετικές για την μέθοδο είναι το Cs_{min} που λειτουργεί σαν κάτω όριο μεγέθους ενός chunk και το α, που αποτελεί παράγοντα κλίμακας του συντελεστή διακύμανσης (*coefficient of variation* $CV = \frac{\sigma}{\mu}$) που βρίσκεται εμπειρικά με βάση το [4]. Οι παράγοντες που το επηρεάζουν είναι το κόστος δρομολόγησης και ο λόγος των επαναλήψεων διά τον αριθμό των νημάτων. Η προεπιλεγμένη τιμή που χρησιμοποιήθηκε στη συνέχεια για τις μετρήσεις είναι το 1.

Στις εξισώσεις της μεθόδου, παρατηρείται ότι η τιμή T_i είναι ίδια με τον υπολογισμό μεγέθους chunk της μεθόδου guided. Όταν η τυπική απόκλιση είναι μηδέν, η μέθοδος ταυτίζεται με την guided. Έτσι και εδώ χρησιμοποιείται ο ίδιος κώδικας με την μέθοδο guided, υπολογίζοντας όμως το μέγεθος με τον παραπάνω τύπο.

3.3.3 Fixed Size Chunking

Η επόμενη μέθοδος είναι η Fixed Size Chunking (FSC) [5], η οποία είναι παρόμοια με τη δυναμική δρομολόγηση. Η κύρια διαφορά τους είναι ότι δεν αναθέτει μία επανάληψη την φορά αλλά chunks μεγέθους που υπολογίζεται με τον παρακάτω τύπο:

$$Cs = \left(\frac{\sqrt{2}Nh}{\sigma P \sqrt{logP}} \right)^{\frac{2}{3}}$$

Ο τύπος αυτός δημιουργήθηκε μέσω μαθηματικής ανάλυσης με στόχο να βρεθεί το ιδανικό μέγεθος για να μειωθεί το κόστος δρομολόγησης διατηρώντας όσο είναι δυνατό, την καλή ισορροπία φορτίου της δυναμικής δρομολόγησης. Ως εισόδους δέχεται την τυπική απόκλιση των χρόνων εκτέλεσης των επαναλήψεων του βρόχου και το κόστος δρομολόγησης της μεθόδου.

Όπως αναφέρθηκε και προηγουμένως, είναι δυνατό στη δυναμική δρομολόγηση να δοθεί ως είσοδος ένα μέγεθος ώστε να αναθέτονται πολλές επαναλήψεις την φορά. Όμως, ανάλογα με το πρόβλημα ο προγραμματιστής πρέπει να υπολογίσει και να δοκιμάσει διάφορα μεγέθη ώστε να έχει τα καλύτερα αποτελέσματα. Αντίθετα στη μέθοδο FSC αρκεί να δώσει τις εισόδους που χρειάζονται και το μέγεθος υπολογίζεται αυτόματα.

Αφού η μέθοδος παρουσιάζει ομοιότητες με τη δυναμική δρομολόγηση, ο κώδικας βασίστηκε πάνω σε αυτή. Η διαφορά είναι ότι η δυναμική δρομολόγηση δέχεται ως όρισμα το μέγεθος των chunks ή χρησιμοποιείται η τιμή 1 από προεπιλογή. Στην FSC το μέγεθος υπολογίζεται σύμφωνα με τον παραπάνω τύπο με εισόδους το κόστος δρομολόγησης και την τυπική απόκλιση των χρόνων εκτέλεσης των επαναλήψεων του βρόχου.

3.3.4 Factoring

Μία προσπάθεια βελτίωσης της FSC είναι η μέθοδος Factoring [6]. Δεν δρομολογεί ένα chunk την φορά, αλλά δημιουργεί τεμάχια από αυτά ίσα με τον αριθμό των νημάτων που διαθέτονται για την εκτέλεση των επαναλήψεων. Δηλαδή με το ίδιο μέγεθος υπολογίζονται πολλά chunks τα οποία εντάσσονται σε μία ουρά από όπου τα νήματα παίρνουν τις επόμενες επαναλήψεις που θα εκτελέσουν όταν χρειαστεί. Αν δεν υπάρχει άλλο chunk στην ουρά, υπολογίζεται το επόμενο τεμάχιο μέχρι να

εκτελεστούν όλες οι επαναλήψεις. Το μέγεθος των chunks παρουσιάζεται στους ακόλουθους τύπους, όπου j ο αριθμός του τεμαχίου που υπολογίζεται.

$$Cs_j = \left\lceil \frac{R_j}{x_j P} \right\rceil$$

$$R_0 = N, R_{j+1} = R_j - PCs_j$$

$$b_j = \frac{P}{2\sqrt{R_j}} \frac{\sigma}{\mu}$$

$$x_0 = 1 + b_0^2 + b_0\sqrt{b_0^2 + 2}$$

$$x_j = 2 + b_j^2 + b_j\sqrt{b_j^2 + 4}, j > 0$$

Για την υλοποίηση της μεθόδου δεν είναι απαραίτητη η χρήση πραγματικής ουράς, αλλά γίνεται προσομοίωσή της. Χρειάζεται να αποθηκευτούν δύο μεταβλητές, το μέγεθος των chunks ενός τεμαχίου και ένας μετρητής που δηλώνει τον αριθμό των στοιχείων που περιέχει η ουρά. Όταν η ουρά είναι άδεια, δηλαδή ο μετρητής είναι μηδέν, υπολογίζεται το μέγεθος για το επόμενο τεμάχιο και η τιμή του μετρητή γίνεται ίση με τον αριθμό των νημάτων. Μόλις ένα νήμα πάρει ένα chunk για να το εκτελέσει, μειώνει κατά ένα τον μετρητή.

Με αυτόν τον τρόπο δεν είναι απαραίτητη η χρήση ουράς κάτι που θα δημιουργούσε παραπάνω κόστος κατά τη δρομολόγηση. Όμως πρέπει να διασφαλιστεί ότι μόνο ένα νήμα θα υπολογίσει το νέο μέγεθος και ότι όσο ο μετρητής είναι μηδέν κανένα νήμα δεν θα λάβει chunk με το προηγούμενο μέγεθος. Συνεπώς πρέπει να χρησιμοποιηθούν κλειδαριές για τον συντονισμό των νημάτων. Μία άλλη λύση είναι κάθε νήμα να υπολογίζει μόνο του το μέγεθος του επόμενου chunk με βάση έναν μετρητή για το πόσα έχουν ήδη εκτελεστεί. Αυτό βέβαια περιλαμβάνει παραπάνω υπολογισμούς και κόστος δρομολόγησης.

3.3.5 Επιλογή μεθόδου

Για την επιλογή μία νέας μεθόδου σε έναν βρόχο αξιοποιήθηκε η λειτουργία `auto`. Σύμφωνα με το πρότυπο η δρομολόγηση όταν χρησιμοποιείται το `auto` επαφίεται υλοποίηση. Έτσι δημιουργήθηκε μία νέα μεταβλητή περιβάλλοντος για την επιλογή της μεθόδου όταν εμφανιστεί το `auto`, με όμοιο τρόπο με την μεταβλητή `OMP_SCHEDULE` που χρησιμοποιείται όταν το `schedule` έχει καθοριστεί να είναι `runtime`.

Η νέα μεταβλητή περιβάλλοντος ονομάστηκε `OMPI_SCHED_AUTO`. Για τη χρήση των νέων μεθόδων υπάρχουν οι ακόλουθοι τρόποι:

1. βρόχος με `schedule auto` και επιλογή νέας μεθόδου μέσω κατάλληλης τιμής της μεταβλητής `OMPI_SCHED_AUTO`.

2. βρόχος με `shedule runtime` και επιλογή νέας μεθόδου μέσω της συνάρτησης `omp_set_schedule`.

3. βρόχος με `schedule runtime`, και επιλογή `auto` είτε με χρήση της συνάρτησης `omp_set_schedule` είτε με την μεταβλητή περιβάλλοντος `OMP_SCHEDULE`. Στη συνέχεια επιλογή νέας μεθόδου μέσω της μεταβλητής `OMPI_SCHED_AUTO`.

Κάποια παραδείγματα αποδεκτών τιμών για την μεταβλητή παρουσιάζονται στη συνέχεια:

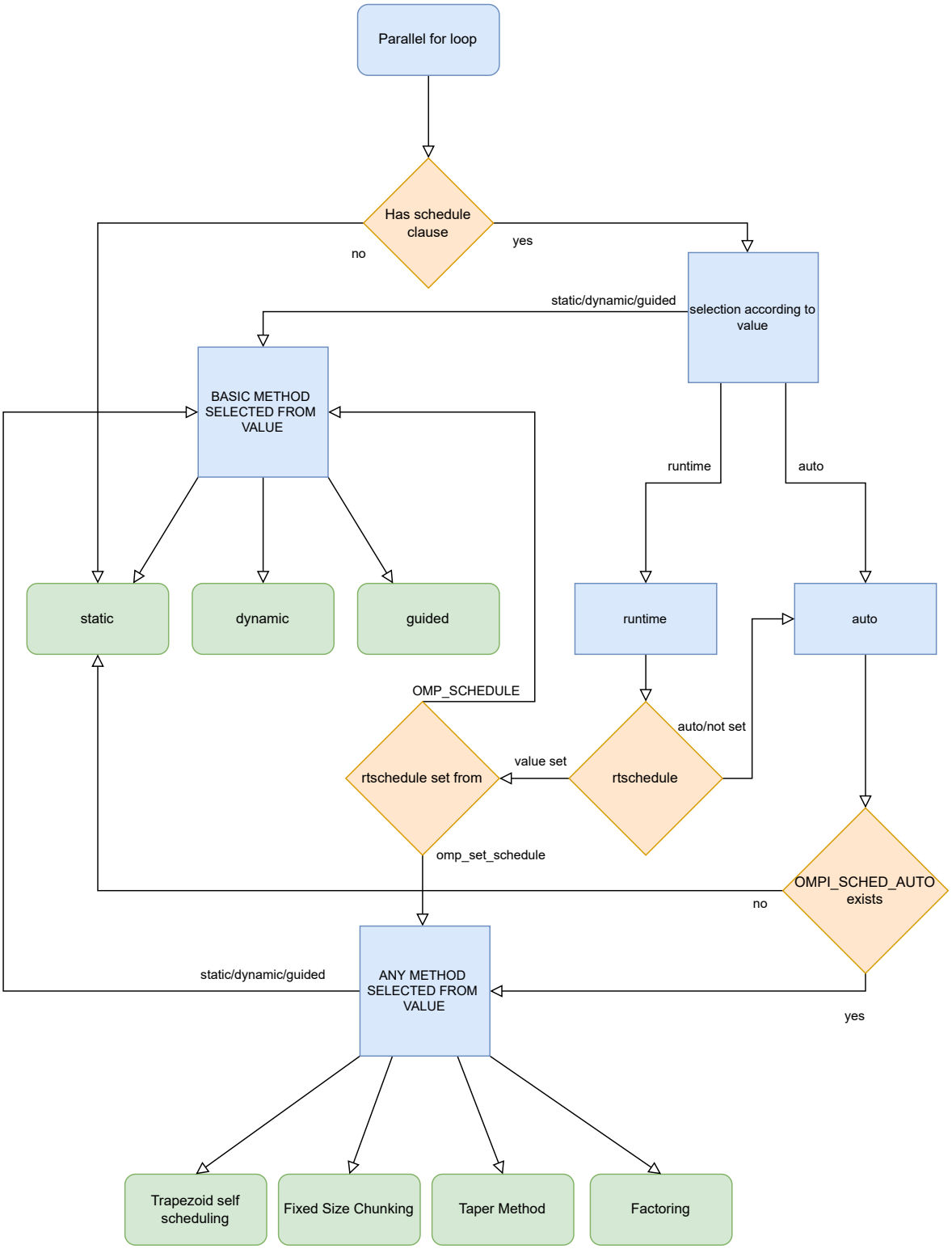
```
export OMPI_SCHED_AUTO="Trapezoid"
export OMPI_SCHED_AUTO="FSC(m= 200, h=100)"
export OMPI_SCHED_AUTO="static()"
export OMPI_SCHED_AUTO="dynamic(c=5)"
```

Δεδομένων όλων των πολιτικών δρομολόγησης, των φράσεων και των μεταβλητών περιβάλλοντος ο καθορισμός της επιθυμητής μεθόδου είναι πλέον πιο πολύπλοκος και γίνεται ως εξής:

- Αν δεν υπάρχει η φράση `schedule` η μέθοδος που επιλέγεται είναι η static.
- Αν υπάρχει στην οδηγία `for` η φράση `schedule` δίνεται μία από τις 5 επιλογές που προδιαγράφει το OpenMP.
  - Αν δοθεί η επιλογή `runtime`, ελέγχεται η τιμή της μεταβλητής `rtschedule`.
    - Αν δεν έχει αλλάξει η τιμή του `rtschedule`, είναι προεπιλεγμένη η τιμή `auto`.
    - Αν η τιμή έχει δοθεί μέσω της μεταβλητής περιβάλλοντος `OMP_SCHEDULE`, μπορεί να επιλεχθεί οποιαδήποτε από τις 3 βασικές μεθόδους του OpenMP ή η επιλογή `auto`.
    - Αν η τιμή έχει δοθεί μέσω της μεθόδου `omp_set_schedule` μπορεί να επιλεχθεί οποιαδήποτε από τις 7 μεθόδους ή η επιλογή `auto`
  - Αν επιλεχθεί η τιμή `auto`, η μέθοδος καθορίζεται από την μεταβλητή περιβάλλοντος `OMPI_SCHED_AUTO`. Μπορεί να επιλεχθεί οποιαδήποτε από τις 7 μεθόδους δρομολόγησης.

* Αν δεν υπάρχει η μεταβλητή `OMPI_SCHED_AUTO` επιλέγεται η μέθοδος static.

Στο σχήμα 3.1 παρουσιάζεται γραφικά η παραπάνω λογική.



Σχήμα 3.1: Διάγραμμα ροής για την επιλογή δρομολόγησης βρόχων στον OMPi

ΚΕΦΑΛΑΙΟ 4

Ετικέτες στο OpenMP

4.1 Περιορισμοί κατά την εκτέλεση

Ένα βασικό πλεονέκτημα της φράσης `schedule(runtime)` είναι ότι η επιλογή της μεθόδου γίνεται εκτός του προγράμματος. Δεν είναι απαραίτητη η τροποποίηση και μετάφραση εκ νέου του κώδικα, αλλά μόνο η αλλαγή μίας μεταβλητής περιβάλλοντος. Το ίδιο συμβαίνει πλέον και με την επιλογή `auto`. Όμως και στις δύο επιλογές εμφανίζεται ένα σημαντικό μειονέκτημα. Όλοι ο βρόχοι που χρησιμοποιούν μία από τις δύο επιλογές, ελέγχονται από μία και μόνο μεταβλητή περιβάλλοντος για την κάθε μία. Ως αποτέλεσμα, για όλους τους βρόχους αναγκαστικά ορίζεται η ίδια πολιτική δρομολόγησης. Ακόμη στο `runtime` δεν μπορούν να επιλεχθούν οι νέες μέθοδοι μέσω της μεταβλητής περιβάλλοντος καθώς κάτι τέτοιο θα απαιτούσε αλλαγές στη σύνταξή της.

Σε ένα πρόγραμμα με πολλούς παραλληλοποιημένους βρόχους διαφορετικής μορφής αναμένεται η χρήση διαφορετικών μεθόδων ανάμεσά τους να δίνει τα καλύτερα αποτελέσματα. Όμως δεν είναι πάντα προφανής η μέθοδος που θα πετύχει τις καλύτερες επιδόσεις, και για να βρεθεί ο καλύτερος συνδυασμός απαιτούνται δοκιμές. Συνεπώς είναι επιθυμητό να υποστηρίζονται διαφορετικοί τρόποι ελέγχου

της μεθόδου σε κάθε βρόχο.

Βέβαια υπάρχει η δυνατότητα χρήσης της επιλογής `runtime` σε συνδυασμό με την μέθοδο `omp_set_schedule`. Με αυτόν τον τρόπο σε κάθε βρόχο μπορεί να επιλεχθεί οποιαδήποτε μέθοδος. Όμως αυτό δεν εξυπηρετεί το σκοπό μας μιας και προϋποθέτει ότι για κάθε βρόχο έχει καθοριστεί μία μέθοδος από τη μετάφραση του προγράμματος. Ως αποτέλεσμα χάνεται η ευελιξία της επιλογής `runtime`.

4.2 Δημιουργία ετικετών

Καθώς το πρόβλημα που περιγράφεται παραπάνω σχετίζεται με τον έλεγχο της συμπεριφοράς του προγράμματος κατά την εκτέλεση, θα πρέπει να σχεδιαστεί ένας νέος μηχανισμός που να δίνει στον χρήστη τις δυνατότητες που επιθυμούμε για τον έλεγχο των διαφορετικών βρόχων. Ο μηχανισμός αυτός θα πρέπει να πληροί τα εξής κριτήρια:

- χρήση μεταβλητών περιβάλλοντος
- μοναδικός τρόπος αναφοράς σε κάθε βρόχο
- καμία παρεμβολή με τις υπάρχουσες λειτουργίες του OpenMP

Προκειμένου να λύσουμε το πρόβλημα σχεδιάστηκε εξ αρχής ένας νέος μηχανισμός ο οποίος πληροί όλα τα παραπάνω και δίνει ιδιαίτερη ευελιξία στον χρήστη. Συγκεκριμένα, εισάγαμε την έννοια της *ετικέτας* (tag) η οποία ουσιαστικά επιτρέπει να ονοματίσει ο προγραμματιστής οποιαδήποτε δομή του OpenMP. Στη συνέχεια, μεταβλητές περιβάλλοντος μπορούν να χρησιμοποιηθούν από τον χρήστη της εφαρμογής για να παραμετροποιήσουν τις ονοματισμένες δομές, χρησιμοποιώντας απλά την ετικέτα τους.

Πιο συγκεκριμένα, ο προγραμματιστής μπορεί να δηλώσει ένα αλφαριθμητικό ως αναγνωριστικό μίας δομής με το οποίο μπορεί στη συνέχεια να αναφερθεί σε αυτήν. Αυτό γίνεται με τη χρήση μίας νέας οδηγίας που εισάγαμε στον OMPi και έχει την ακόλουθη σύνταξη:

```
#pragma ompix tag(<tag>)
    omp-constuct
```

Με βάση την ετικέτα του, κάθε κόμβος μπορεί να προσδιοριστεί μοναδικά. Ακόμη, για μεγαλύτερη ευελιξία στη δημιουργία ετικετών, υπάρχει η δυνατότητα προσθήκης μίας μεταβλητής τύπου `int` στο τέλος του αλφαριθμητικού. Με αυτόν τον τρόπο, αν για παράδειγμα υπάρχει ένας βρόχος εμφωλευμένος μέσα σε έναν άλλο, είναι δυνατό σε συγκεκριμένες επαναλήψεις να έχει διαφορετική ετικέτα από ότι σε άλλες.

Οι ετικέτες αποθηκεύονται σε μία στοίβα και διαγράφονται μετά τη δομή που περιγράφουν. Όταν κατά την εκτέλεση εμφανιστεί μία καινούρια ετικέτα, ελέγχεται αν υπάρχει η αντίστοιχη μεταβλητή περιβάλλοντος. Αν βρεθεί, τα δεδομένα της αποθηκεύονται μαζί με την ετικέτα.

4.3 Υλοποίηση

Το πρώτο μέρος της υλοποίησής, αφορά το μέρος του μεταφραστή. Αρχικά πρέπει κατά τη λεκτική ανάλυση, να αναγνωρίζεται η νέα λέξη κλειδί `tag`. Στη συνέχεια κατά τη συντακτική ανάλυση διασφαλίζεται ότι το `tag` θα εμφανιστεί μόνο πριν από κάποια δομή του OpenMP. Για ευκολότερη χρήση, προστέθηκε και η δυνατότητα να εμφανιστεί η λέξη `tag` κατευθείαν μέσα από την οδηγία `for` ή τον συνδυασμό `parallel for`, αντί να πρέπει να τις περικλείει. Για παράδειγμα αν θέλουμε να δώσουμε ετικέτα σε έναν βρόχο, αντί να είναι απαραίτητη η σύνταξη:

```
#pragma ompix tag("ForLoopTag")
#pragma omp for schedule(auto)
for (...)
    ...
```

το ίδιο αποτέλεσμα μπορεί να επιτευχθεί και ως εξής:

```
#pragma omp for schedule(auto) tag("ForLoopTag")
for (...)
    ...
```

Το ίδιο συμβαίνει και στην οδηγία `parallel for`. Ολοκληρώνοντας αυτές τις αλλαγές, ένα πρόγραμμα που περιέχει τη χρήση ετικετών θεωρείται συντακτικά σωστό και μπορεί να μεταφραστεί από τον OMPi, αλλά δεν έχει καμία λειτουργία.

Για την υλοποίηση της δημιουργίας και αποθήκευσης μία ετικέτας υπάρχουν δύο περιπτώσεις. Όταν ο προγραμματιστής δίνει μόνο ένα αλφαριθμητικό ως όρισμα, η ετικέτα είναι έτοιμη και μπορεί να αποθηκευτεί κατευθείαν. Έτσι σε αυτό το σημείο στο μετασχηματισμένο πρόγραμμα προστίθεται η κλήση μίας νέας συνάρτησης που δημιουργεί ένα νέο `tag` (τύπος ενός νέου `struct`) και το αποθηκεύει σε έναν δείκτη ενός νήματος. Ένα παράδειγμα για την κλήση της παραπάνω συνάρτησης εμφανίζεται στη γραμμή 16 του Κώδικα 4.2 Στην περίπτωση όμως που υπάρχει και δεύτερο όρισμα, μία μεταβλητή `int`, πρέπει να παραχθεί ένα ακόμη κομμάτι κώδικα. Σε αυτό, δίνεται η απαραίτητη μνήμη και μετά το αρχικό αλφαριθμητικό αντιγράφεται η τιμή της μεταβλητής που δόθηκε. Αφού πλέον η ετικέτα είναι πλήρως σχηματισμένη μπορεί να αποθηκευτεί στην στοίβα. Αυτό παρουσιάζεται στις γραμμές 1-9 του Κώδικα 4.2 που θα δούμε με λεπτομέρεια παρακάτω.

Όπως αναφέρθηκε και νωρίτερα, υπάρχουν περιπτώσεις που οι βρόχοι είναι εμφωλευμένοι και έτσι μία μεταβλητή για τις ετικέτες δεν είναι αρκετή. Το `struct` που δημιουργήθηκε για να κρατάει την απαραίτητη πληροφορία για μία ετικέτα, περιλαμβάνει έναν δείκτη προς ένα `struct` ίδιου τύπου. Έτσι αν έχουμε την προσθήκη νέας ετικέτα, αυτή αποθηκεύεται στην αρχή μίας συνδεδεμένης λίστας από ετικέτες. Αφού ολοκληρωθεί ο κώδικας της νέας ετικέτας, αυτή διαγράφεται και πλέον ισχύει η προηγούμενη ετικέτα αν υπάρχει. Ο τρόπος λειτουργίας της συνδεδεμένης λίστας είναι ίδιος με τη λειτουργία μίας στοίβας. Συνεπώς οι ετικέτες «κληροδοτούνται» στις εμφωλευμένες δομές. Με άλλα λόγια αν για παράδειγμα σε μία παράλληλη περιοχή έχει δοθεί μία ετικέτα `T`, όλες οι δομές OpenMP που υπάρχουν μέσα στην παράλληλη περιοχή θα διαθέτουν την ετικέτα `T`. Σε περίπτωση που μία από τις δομές αυτές έχει την δική της ετικέτα, η ετικέτα `T` δεν ισχύει για την δομή αυτή.

Η δημιουργία νέας ετικέτας μπορεί να εμφανιστεί ανεξάρτητα της ύπαρξης παράλληλης περιοχής ή όχι. Έτσι πρέπει να διασφαλιστεί ότι η αποθήκευσή της στη στοίβα θα εκτελεστεί σε κάθε περίπτωση μία μόνο φορά. Για τον λόγο αυτό η αποθήκευση εκτελείται μετά τον έλεγχο αν υπάρχει ένα μοναδικό νήμα, ή το νήμα έχει το αναγνωριστικό 0. Μετά την εκτέλεση του κώδικα από τη δομή που περιγράφει η ετικέτα, ο ίδιος έλεγχος επαναλαμβάνεται ώστε ένα νήμα να διαγράψει την ετικέτα.

Καθώς η επιθυμητή λειτουργία του προγράμματος μπορεί να εξαρτάται από κάποια ετικέτα, είναι απαραίτητο η αποθήκευση και διαγραφή να γίνουν πριν και μετά την εκτέλεση του κώδικα που περικλείει η ετικέτα αντίστοιχα. Σε διαφορετική περίπτωση κάποια νήματα στην αρχή ή το τέλος πιθανώς να μην είχαν τις

πληροφορίες της ετικέτας, αφού ακόμα δεν έχει δημιουργηθεί. Συνεπώς πρέπει να διασφαλιστεί ο συγχρονισμός των νημάτων με την προσθήκη φραγμάτων γύρω από τον κώδικα της εσωτερικής δομής.

Τέλος αφού, υπάρχει πλέον η λειτουργία των ετικετών, απομένει ο μηχανισμός για τη χρήση τους στην επιλογή της μεθόδου δρομολόγησης. Αυτό γίνεται με τη χρήση μεταβλητών περιβάλλοντος για κάθε ετικέτα. Έχοντας ξεχωριστές μεταβλητές για κάθε βρόχο που χρειάζεται, ο προγραμματιστής μπορεί να ελέγξει για κάθε έναν τη μέθοδο και τις εισόδους που επιθυμεί. Αυτό περιγράφεται στην επόμενη υποενότητα.

4.3.1 Επιλογή μεθόδου δρομολόγησης

Με τη χρήση των ετικετών, είναι δυνατή η επιλογή μεθόδου δρομολόγησης ενός βρόχου. Μετά τον κώδικα με τον οποίο αποθηκεύεται στη στοίβα μία ετικέτα έστω `<T>`, καλείται μία συνάρτηση που διαβάζει -εάν υπάρχει- τη μεταβλητή περιβάλλοντος `OMPI_SCHED_TAG_<T>`. Η σύνταξή της είναι ίδια με τη σύνταξη της μεταβλητής `OMPI_SCHED_AUTO`. Στην περίπτωση που εντοπιστεί και έχει μία αποδεκτή τιμή, τα δεδομένα της αποθηκεύονται μαζί με την ετικέτα.

Στη συνέχεια, όταν χρειαστεί να επιλεχθεί κάποια μέθοδος για έναν βρόχο, εάν υπάρχει ετικέτα θα χρησιμοποιηθεί η μέθοδος που έχει επιλεχθεί για αυτή. Καθώς οι ετικέτες παραμένουν στη στοίβα μέχρι να ολοκληρωθεί ο κώδικας που περιέχει η δομή που περικλείουν, σε εμφωλευμένους κόμβους, αν δεν έχουν τη δική τους ετικέτα, ισχύουν τα δεδομένα της ετικέτας του εξωτερικού κόμβου.

Στο σημείο αυτό πρέπει να αναφερθεί ότι αν υπάρχει κάποια άλλη μέθοδος επιλεγμένη για κάποιον κόμβο με τη χρήση της φράσης `schedule`, θα έχει προτεραιότητα και εκείνη θα είναι η μέθοδος που θα χρησιμοποιηθεί. Για να επιλεχθεί η μέθοδος που δηλώνει μία ετικέτα πρέπει να βρεθούμε σε μία από τις δύο συναρτήσεις που επιλέγουν μία μέθοδο κατά την εκτέλεση, είτε με την επιλογή `runtime`, είτε με την επιλογή `auto`.

Στο τέλος της δομής ένα νήμα αναλαμβάνει να διαγράψει την ετικέτα από τη στοίβα. Καθώς η δημιουργία και η διαγραφή της ετικέτας επηρεάζει την λειτουργία του προγράμματος είναι απαραίτητο να συντονίσουμε τα νήματα. Έτσι μετά την προσθήκη και πριν την αφαίρεση νέας ετικέτας καλείτε ένα φράγμα που εξασφαλίζει ότι όλα τα νήματα θα βρεθούν σε εκείνο το σημείο πριν συνεχίσουν τη λειτουργία

τους.

Ο χρήστης θα μπορούσε επομένως να χρησιμοποιήσει τις μεταβλητές περιβάλλοντος ως εξής προκειμένου να καθορίσει τη μέθοδο δρομολόγησης του βρόχου στη γραμμή 5 του Κώδικα 4.1. Αν το πρόγραμμα εκτελεστεί χωρίς να έχει δοθεί καμία τιμή στις μεταβλητές περιβάλλοντος, η μέθοδος που θα χρησιμοποιηθεί είναι η static ως προεπιλογή του `schedule(auto)`. Στη συνέχεια κάνοντας

```
export OMPI_SCHED_AUTO="Trapezoid(f=100, l=10)"
```

και εκτελώντας εκ νέου το πρόγραμμα, επιλέγεται η μέθοδος Trapezoid Self Scheduling, όπως περιγράφηκε και στο Κεφάλαιο 3. Ως ορίσματα δίνονται οι τιμές 100 για το μέγεθος του πρώτου chunk και 10 για το μέγεθος του τελευταίου. Στη συνέχεια μπορεί να ανατεθεί τιμή για την ετικέτα της γραμμής 3 με τον παρακάτω τρόπο.[1]

```
export OMPI_TAG_SCHED_withVar0="Factoring(m=100,s=22.45)"
```

Όμως, αν εκτελεστεί και πάλι το πρόγραμμα η μέθοδος που θα επιλεχθεί συνεχίζει να είναι η Trapezoid Self Scheduling. Αυτό συμβαίνει καθώς στην κορυφή της στοίβας των ετικετών βρίσκεται η ετικέτα «simple», για την οποία δεν υπάρχει ακόμη η αντίστοιχη μεταβλητή περιβάλλοντος. Καθώς δεν υπάρχουν πληροφορίες δρομολόγησης για την ετικέτα που προσδιορίζει τον βρόχο της γραμμής 5 η επιλογή γίνεται με βάση την αποθηκευμένη τιμή για τους βρόχους με `schedule(auto)`. Τέλος αν δοθεί τιμή για τη μεταβλητή της ετικέτας που δημιουργείται στην γραμμή 4 ως εξής:

```
export OMPI_TAG_SCHED_simple="Taper(m=100, s=45.35, c=10)"
```

στο βρόχο που ακολουθεί επιλέγεται ως μέθοδος δρομολόγησης η Taper. Ακόμη, χρησιμοποιούνται οι είσοδοι για τη μέση τιμή την τυπική απόκλιση και το ελάχιστο μέγεθος chunk στους υπολογισμού της μεθόδου που παρουσιάστηκαν στην υποενότητα 3.3.2.

Στον Κώδικα 4.1 και 4.2 παραθέτουμε δύο αποσπάσματα κώδικα, για τη χρήση των ετικετών και τον κώδικα που παράγεται από τον OMPi[2]

[1] Η μεταβλητή `sum` αρχικοποιείται στην τιμή 0 σε προηγούμενο σημείο του προγράμματος.

[2] Ο παραγόμενος κώδικας είναι ελαφρώς τροποποιημένος για ευκολότερη ανάγνωση και έχει παραλειφθεί ο κώδικας του βρόχου.

Κώδικας 4.1: Χρήση ετικετών

```
1  #pragma omp parallel private(j,k,sum)
2  {
3    #pragma ompix tag("withVar",sum)
4    #pragma omp for tag("simple")
5    for (i = 0; i < N; i++)
6      for (j = 0; j < N; j++)
7      {
8        for (k = sum = 0; k < N; k++)
9        sum += A[i][k]*B[k][j];
10       C[i][j] = sum;
11     }
12 }
```

Κώδικας 4.2: Αποτέλεσμα μετατροπών του OMPi στις ετικέτες

```
1  char arg[ 11];
2
3  snprintf(arg, 11, "%d", sum);
4  int len = sizeof("withVar");
5  char * tag = ort_memalloc(len + sizeof(arg));
6
7  memcpy(tag, "withVar", len);
8  memcpy(tag + len - 1, arg, sizeof(arg));
9  ort_push_tag(tag);
10 ort_read_tag_sched();
11 ort_barrier_me();
12 {
13   int i;
14
15
16   ort_push_tag("simple");
17   ort_read_tag_sched();
18   ort_barrier_me();
19
20   ...
21
22   ort_leaving_for();
23   ort_barrier_me();
24   ort_pop_tag();
25 }
26 ort_barrier_me();
27 ort_pop_tag();
```

ΚΕΦΑΛΑΙΟ 5

Πειραματικά αποτελέσματα

Προκειμένου της αξιολόγησης της εργασίας μας, πειραματιστήκαμε σε τρεις διαφορετικούς άξονες. Ο πρώτος άξονας αφορά την αποτίμηση των καθυστερήσεων (overheads) των νέων μεθόδων δρομολόγησης που υλοποιήθηκαν. Ο δεύτερος αφορά την εφαρμογή των διαφορετικών μεθόδων σε πραγματικές εφαρμογές. Για το σκοπό αυτό επιλέχθηκε η σουίτα εφαρμογών NAS Parallel Benchmarks. Ο τρίτος αφορά τη χρήση των ετικετών σε εφαρμογές με πολλαπλούς βρόχους. Αυτό το είδαμε σε ένα υποσύνολο των εφαρμογών NAS.

Όλες οι μετρήσεις έγιναν στον υπολογιστή parade που διαθέτει η Ομάδα Παράλληλης Επεξεργασίας. Ο υπολογιστής διαθέτει 4 επεξεργαστές Intel Xeon Gold 6130, όπου ο καθένας περιέχει 16 πυρήνες που αντιστοιχούν σε 32 threads. Δηλαδή συνολικά το σύστημα έχει 64 πυρήνες και 128 threads. Με βάση αυτό τα πειράματα που ακολουθούν έγιναν με επιλογή 64 ή 128 νημάτων στο OpenMP.

5.1 Αποτίμηση κόστους δρομολόγησης

Μία από τις εισόδους που χρειάζεται η μέθοδος Fixed Size Chunking είναι ο χρόνος που απαιτείται για την ανάθεση των επαναλήψεων στα νήματα, δηλαδή το κόστος δρομολόγησης της μεθόδου. Για τη μέτρηση αυτή χρησιμοποιήθηκε η εφαρμογή schedbench των EPCC OpenMP Microbenchmarks [7]. Αυτή αρχικά εκτελεί έναν βρόχο σειριακά μετρώντας τον χρόνο που χρειάζεται ως χρόνο αναφοράς. Στη συνέχεια ο ίδιο βρόχος εκτελείται παράλληλα με διαφορετικές μεθόδους δρομολόγησης και μετριέται ο χρόνος που απαιτείται. Οι επαναλήψεις στους παράλληλους βρόχους πολλαπλασιάζονται με τον αριθμό των νημάτων ώστε σε κάθε ένα να εκτελεστούν όσες επαναλήψεις είχε η εκτέλεση αναφοράς. Η διαφορά των δύο χρόνων αποτελεί το κόστος δρομολόγησης της μεθόδου.

Ο κώδικας του schedbench προσαρμόστηκε ώστε να εκτελεστούν οι νέες μέθοδοι. Δημιουργήθηκε μία νέα συνάρτηση η οποία κάνει χρήση της επιλογής schedule runtime. Η μέθοδος αυτή καλείται σε συνδιασμό με τη συνάρτηση omp_set_schedule ώστε να επιλεχθούν και να γίνουν μετρήσεις σε όλες τις νέες μεθόδους. Όμως για να μετρηθεί σωστά το κόστος δρομολόγησης πρέπει στο πρόγραμμα να δοθούν οι απαραίτητες παράμετροι. Ως σημείο αναφοράς μπορούν να χρησιμοποιηθούν οι μετρήσεις της μεθόδου dynamic. Γνωρίζουμε ότι το κόστος της, θα μειώνεται περίπου στο μισό κάθε φορά που διπλασιάζεται το μέγεθος των chunks καθώς χρειάζονται οι μισές αναθέσεις. Για πιο ακριβείς μετρήσεις είναι καλό να έχουμε μεγάλο αριθμό επαναλήψεων και μικρή καθυστέρηση.

Ακόμη, οι μετρήσεις επηρεάζονται από τον αριθμό των νημάτων που τις εκτελούν. Είναι επιθυμητό να υπάρχουν νήματα ίσα με τον αριθμό των φυσικών πυρήνων που διαθέτει το σύστημα. Διαφορετικά τα λογικά νήματα μπορεί δημιουργήσουν καθυστέρηση από τον ανταγωνισμό μεταξύ τους. Για τον έλεγχο των νημάτων πρέπει να χρησιμοποιηθεί η μεταβλητή περιβάλλοντος του OpenMP. Επιλέχθηκαν 64 νήματα με βάση το σύστημα που έγιναν οι μετρήσεις. Για να εξασφαλιστεί ότι κάθε νήμα εκτελείται σε διαφορετικό πυρήνα δόθηκε στην μεταβλητή περιβάλλοντος OMP_PLACES η τιμή cores. Ακόμη μετά από λίγες δοκιμές χρησιμοποιήθηκαν οι ακόλουθες παράμετροι που προτείνονται στο [8].

- itersperthr = 8192
- –outer-repetitions 50

Πίνακας 5.1: Κόστος δρομολόγησης ανά μέθοδο

Μέθοδος	Κόστος δρομολόγησης (μs)
static	2.6876
dynamic	32817.2664
guided	504.7057
Trapezoid Self Scheduling	348.8038
Fixed Size Chunking	153.8179
Taper	453.0683
Factoring	891.3377

- –delay-time 0.01
- –test-time 2000

Οι μετρήσεις του κόστους της κάθε μεθόδου φαίνονται στον πίνακα 5.1.

Όπως αναμένεται, το μεγαλύτερο κόστος παρουσιάζεται στη δυναμική δρομολόγηση και το μικρότερο στη στατική. Ακόμη η μέθοδος Trapezoid φαίνεται ότι παρουσιάζει μειωμένο κόστος από την guided, το οποίο ήταν και ο στόχος της. Παρατηρούμε επίσης ότι το μικρότερο κόστος εμφανίζεται στην μέθοδο FSC καθώς αρκεί ο υπολογισμός του μεγέθους και δεν χρειάζεται κάποιος συντονισμός στα νήματα. Τέλος καθώς η υλοποίηση της μεθόδου Factoring περιλαμβάνει κλειδαριές, βλέπουμε ένα λίγο αυξημένο κόστος συγκριτικά με τις υπόλοιπες μεθόδους.

5.2 Μέση τιμή και τυπική απόκλιση

Κατά την παρουσίαση των νέων μεθόδων στο Κεφάλαιο 3, αρκετές φορές έγινε αναφορά στη μέση τιμή και την τυπική απόκλιση των χρόνων εκτέλεσης των επαναλήψεων. Οι δύο τιμές, όπου είναι απαραίτητες δίνονται ως είσοδοι στις νέες μεθόδους. Καθώς οι τιμές αυτές δεν σχετίζονται με τη μέθοδο αλλά με το πρόβλημα και τη μορφή του βρόχου που εκτελείται κάθε φορά, βρίσκονται με την πραγματοποίηση κάποιων μετρήσεων κατά την εκτέλεση του βρόχου.

Για τον λόγο αυτό προστέθηκε μία ακόμα επιλογή για τη δρομολόγηση ενός βρόχου η οποίο ονομάστηκε `profiling`. Η επιλογή της μπορεί να γίνει μέσω της μεταβλητής `OMPI_SCHED_AUTO`, δίνοντάς της την τιμή `profiling`. Η μέθοδος αυτή ανα-

θέτει μία επανάληψη τη φορά σε κάθε νήμα όπως η δυναμική δρομολόγηση, αλλά μετράει τον χρόνο που χρειάστηκε για να εκτελεστεί. Όλοι οι χρόνοι των επαναλήψεων αποθηκεύονται σε έναν πίνακα. Μόλις εκτελεστούν όλες οι επαναλήψεις ένα νήμα αναλαμβάνει να υπολογίσει τη μέση τιμή και την τυπική απόκλιση των χρόνων, τις οποίες στη συνέχεια τυπώνει στην οθόνη με την μορφή:

```
Mean = 155,  standard deviation = 14.798649, n = 200000
```

Ο προγραμματιστής έχει πλέον τη μέση τιμή και την τυπική απόκλιση των χρόνων εκτέλεσης των επαναλήψεων του βρόχου και μπορεί να τις δώσει ως είσοδο στις μεθόδους που είναι απαραίτητο, σε επόμενες εκτελέσεις της εφαρμογής.

5.3 NAS Parallel Benchmarks

Τα NAS Parallel Benchmarks [9] αποτελούν μία σειρά από προγράμματα με επιστημονικούς υπολογισμούς που είναι κατάλληλα για παραλληλοποίηση και για αυτό το λόγο χρησιμοποιούνται συχνά ως μετροπρογράμματα (benchmarks) σε παράλληλα συστήματα. Τα περισσότερα είναι γραμμένα σε Fortran οπότε για τις μετρήσεις στον OMPi χρησιμοποιήθηκε μία ανεπίσημη έκδοση που μεταφράστηκε σε C και παραλληλοποιήθηκε για το OpenMP από το Πανεπιστήμιο της Tsukuba. Εμείς χρησιμοποιήσαμε την έκδοση 3.0 η οποία είναι διαθέσιμη στο [10]. Στον πίνακα 5.2 παρουσιάζονται οι εφαρμογές που εκτελέστηκαν, οι κλάσεις μεγέθους τους και το πλήθος των βρόχων στους οποίους χρησιμοποιήθηκε η επιλογή auto.

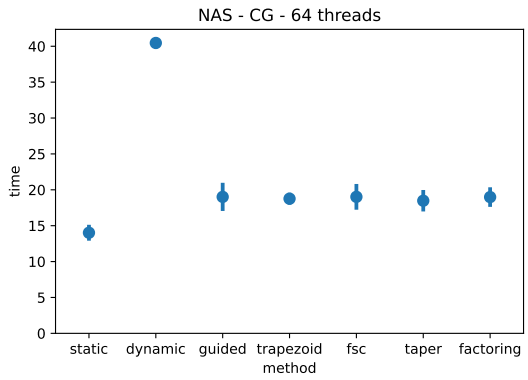
Για την εκτέλεση των προγραμμάτων στην έκδοση που χρησιμοποιήθηκε προσφέρονται πέντε επιλογές, οι S, W, A, B και C σε αύξουσα σειρά μεγέθους. Καθώς το μηχάνημα στο οποίο έγιναν οι μετρήσεις έχει αρκετή υπολογιστική δύναμη, στα περισσότερα προγράμματα χρησιμοποιήθηκε η επιλογή C. Κάποιες εφαρμογές παρουσιάζουν προβλήματα κατά την εκτέλεση ή τη μετάφρασή τους. Για να διασταυρωθεί ότι το πρόβλημα πηγάζει από αυτά δοκιμάστηκε η μετάφραση και εκτέλεσή τους και με άλλους μεταφραστές. Τα ίδια προβλήματα εμφανίστηκαν και στους μεταφραστές `gcc` (έκδοση 8.5.0) και `clang` (έκδοση 12.0.1).

Τα προγράμματα BT και FT δεν δουλεύουν στην κλάση C και έτσι επιλέχθηκε η προηγούμενη. Τέλος δεν χρησιμοποιήθηκαν δύο προγράμματα, τα IS και SP. Το πρώτο εκτελείται σωστά μόνο μέχρι την κλάση μεγέθους A, στην οποία ο χρόνος που απαιτείται είναι ελάχιστος για να παρατηρηθεί αλλαγή στις μετρήσεις. Το δεύτερο

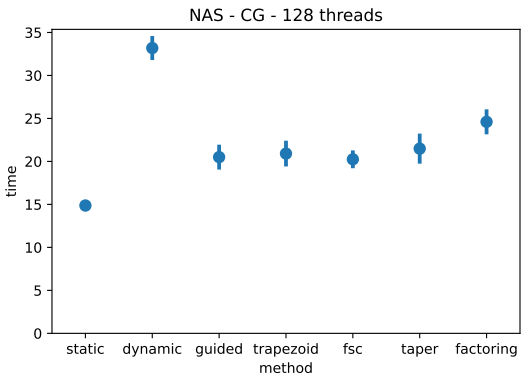
Πίνακας 5.2: Εφαρμογές που περιέχονται στα NAS

Εφαρμογή	Περιγραφή	Κλάση εκτέλεσης	Βρόχοι με schedule auto
BT	Επίλυση συνθετικού συστήματος μη γραμμικών μερικών διαφορικών εξισώσεων χρησιμοποιώντας τριγωνικά μπλοκ.	B	6
CG	Εκτίμηση της μικρότερης ιδιοτιμής ενός μεγάλου αραιού συμμετρικού θετικού καθορισμένου πίνακα με τη μέθοδο της αντίστροφης διάσχισης	C	2
EP	Παραγωγή ανεξάρτητων Γκαουσιανών μεταβλητών	C	1
FT	Επίλυση μερικής διαφορικής εξίσωσης τρίτου βαθμού χρησιμοποιώντας τον γρήγορο μετασχηματισμό Fourier	B	1
IS	Παράλληλη ταξινόμηση N μοναδικών ακεραίων που παράγονται από γεννήτρια ψευδοτυχαίων αριθμών	-	-
LU	Επίλυση συνθετικού συστήματος μη γραμμικών μερικών διαφορικών εξισώσεων χρησιμοποιώντας τη μέθοδο συμμετρικής διαδοχικής υπερχαλάρωσης	C	3
MG	Εφαρμογή της επαναληπτικής πολυπλεγματικής μεθόδου V-cycle σε τρισδιάστατο κυβικό πλέγμα για προσεγγιστική λύση σε μερική διαφορική εξίσωση Poisson	C	1
SP	Επίλυση συνθετικού συστήματος μη γραμμικών μερικών διαφορικών εξισώσεων χρησιμοποιώντας τριγωνικά μπλοκ.	-	-

περιέχει ένα μεγάλο πλήθος από βρόχους μικρού μεγέθους που παραλληλοποιούνται. Έτσι δεν υπάρχει κάποιος βασικός βρόχος ώστε να χρησιμοποιηθούν σε αυτόν

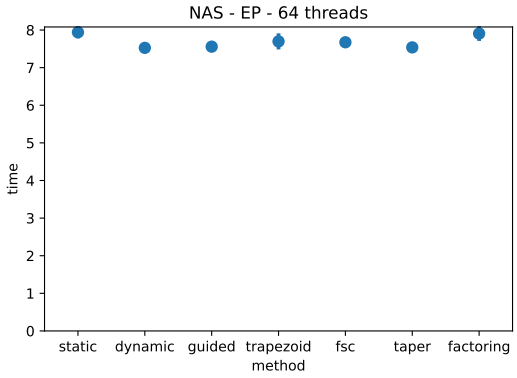


(α)

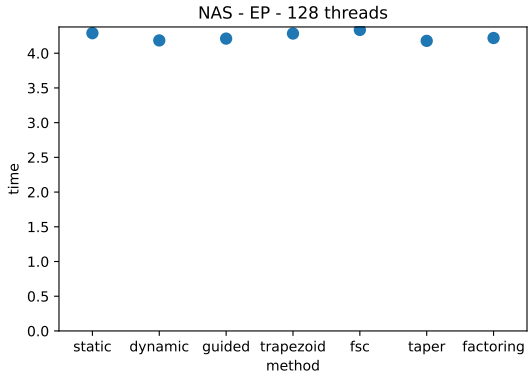


(β)

Σχήμα 5.1: Μετρήσεις για το benchmark CG



(α)



(β)

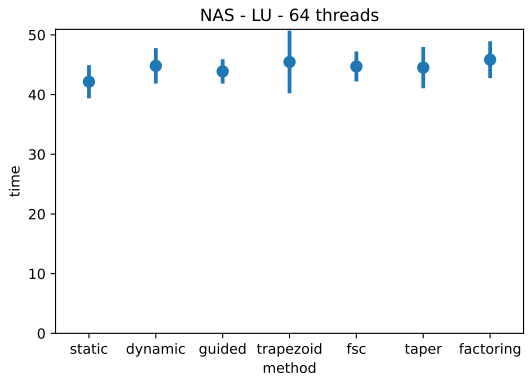
Σχήμα 5.2: Μετρήσεις tbια το benchmark EP

οι διαφορετικές μέθοδοι και να παρατηρηθεί κάποια διαφορά.

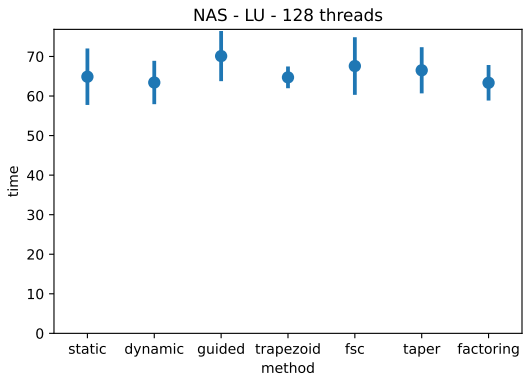
Στα σχήματα 5.1 - 5.6 παρουσιάζονται διαγράμματα με τις μετρήσεις που έγιναν για τις διαφορετικές μεθόδους για 64 ή 128 νήματα. Κάθε εφαρμογή εκτελέστηκε και χρονομετρήθηκε 10 φορές. Στα διαγράμματα εμφανίζεται η μέση τιμή των χρόνων εκτέλεσης, ενώ όπου είναι ορατές οι γραμμές απεικονίζουν την τυπική απόκλισή των χρόνων.

5.3.1 Αποτελέσματα και σχόλια

Αρχικά παρατηρώντας τα διαγράμματα για την εφαρμογή CG, βλέπουμε ότι η μέθοδος static πετυχαίνει τα καλύτερα αποτελέσματα. Όλες οι υπόλοιπες μέθοδοι έχουν παρόμοιους χρόνους, με εξαίρεση τη δυναμική δρομολόγηση, όπου έχουμε διπλά-

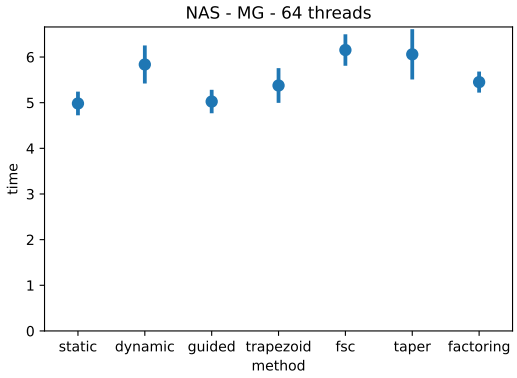


(α)

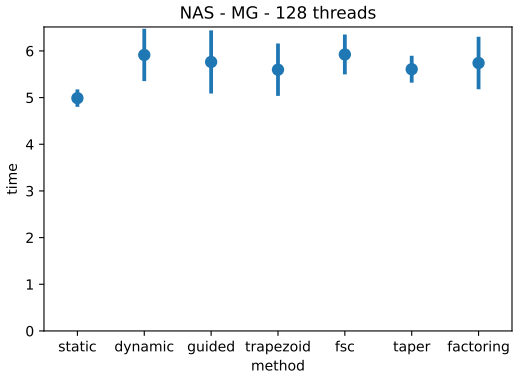


(β)

Σχήμα 5.3: Μετρήσεις για το benchmark LU



(α)

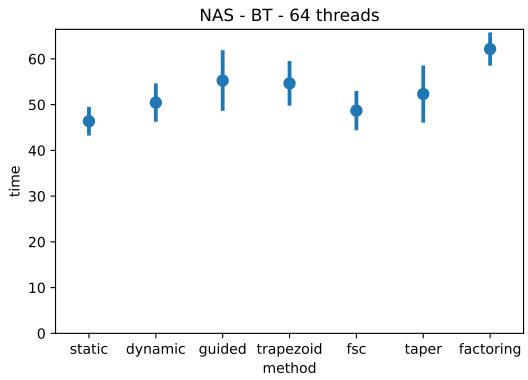


(β)

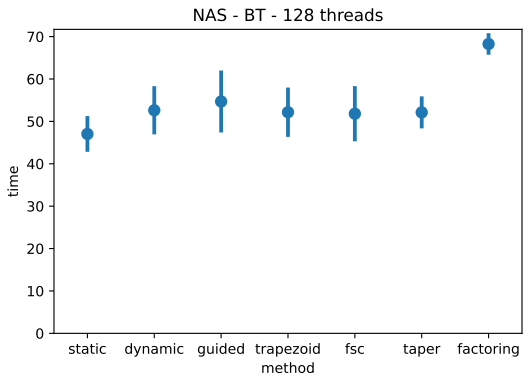
Σχήμα 5.4: Μετρήσεις για το benchmark MG

σιους χρόνους. Αυτό οφείλεται στο ότι ο βρόχος που εκτελούμε παράλληλα περιέχει μεγάλο αριθμό επαναλήψεων ισόποσου φόρτου και έτσι το πολύ μικρό κόστος δρομολόγησης της μεθόδου static υπερισχύει των άλλων. Ακόμη παρατηρούμε ότι όταν έχουμε τα διπλάσια νήματα, η μέθοδος factoring λόγω την κλειδαριών εμφανίζει ελαφρώς αυξημένους χρόνους, ενώ μόνο στη μέθοδο dynamic παρατηρείται βελτίωση λόγω τον παραπάνω νημάτων.

Στην εφαρμογή EP εμφανίζεται μία διαφορετική εικόνα. Η μέθοδος static παρουσιάζει τους χειρότερους χρόνους, με τη μέθοδο dynamic να έχει καλές επιδόσεις. Ακόμη, με την χρήση 128 νημάτων παρατηρούμε βελτίωση σε όλες τις μεθόδους, με μεγαλύτερη διαφορά στην μέθοδο factoring. Σε αυτή την εφαρμογή, μία καινούρια μέθοδος, η Taper, πετυχαίνει παρόμοιους χρόνους με τη δυναμική δρομολόγηση και για τα 128 νήματα έχει τις καλύτερες επιδόσεις, αν και είναι μικρή διαφορά.

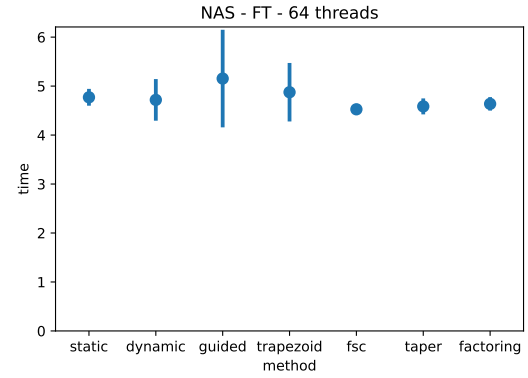


(α)

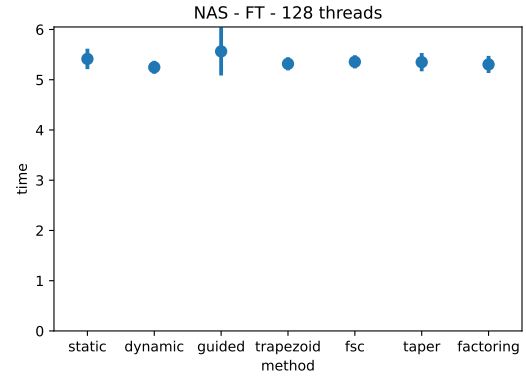


(β)

Σχήμα 5.5: Μετρήσεις για το benchmark BT



(α)



(β)

Σχήμα 5.6: Μετρήσεις για το benchmark FT

Στις εφαρμογές LU, MG και BT η στατική δρομολόγηση πετυχαίνει τις καλύτερες επιδόσεις. Εξαίρεση αποτελούν οι μετρήσεις της LU για 128 νήματα. Βέβαια οι χρόνοι για 128 νήματα είναι χειρότεροι από αυτούς με 64 για όλες τις μεθόδους. Ο καλύτερος χρόνος στην εφαρμογή LU με 128 νήματα είναι αυτός της μεθόδου Factoring.

Τέλος στην εφαρμογή FT, οι καινούριες μέθοδοι που υλοποιήθηκαν πετυχαίνουν τις καλύτερες επιδόσεις. Αρχικά στα 64 νήματα οι Fixed Size Chunking έχει τους καλύτερους χρόνους με την Taper και την Factoring να της ακολουθούν. Στα 128 νήματα, η σειρά τους αντιστρέφεται ενώ και η μέθοδος Trapezoid παρουσιάζει αντίστοιχους καλούς χρόνους. Σε αυτή την περίπτωση όμως η δυναμική δρομολόγηση έχει ελαφρώς καλύτερες επιδόσεις.

Συμπερασματικά μπορούμε να πούμε ότι οι 3 μέθοδοι του OpenMP είναι μάλλον

περιορισμένες. Υπήρξαν εφαρμογές όπου οι νέες μέθοδοι έδωσαν καλύτερα αποτελέσματα και επομένως πιστεύουμε ότι είναι σημαντική η μελέτη και έρευνα επάνω σε όσο δυνατόν περισσότερες μεθόδους δρομολόγησης.

5.4 Μετρήσεις για τη χρήση ετικετών

Μία από τις εφαρμογές των NAS που χρησιμοποιήθηκαν για τις μετρήσεις των νέων μεθόδων είναι η BT. Όπως αναφέρθηκε στον Πίνακα 5.2, χρησιμοποιήθηκε η επιλογή auto σε 6 διαφορετικούς βρόχους. Ακόμη οι βρόχοι μπορούσαν να χωριστούν σε δύο ομάδες των τριών καθώς παρουσίαζαν ίδια μορφή. Με την χρήση των ετικετών δοκιμάστηκαν όλοι οι δυνατοί συνδυασμοί μεθόδων ανάμεσα στις δύο ομάδες.

Σε πολλές περιπτώσεις ο συνδυασμό διαφορετικών μεθόδων έδωσε καλύτερα αποτελέσματα από τη χρήση μίας και για τους 6 βρόχους. Έτσι μπορούμε να παρατηρήσουμε ότι η χρήση τους μπορεί να οδηγήσει σε θετικά αποτελέσματα. Παρ' όλα αυτά από τις μετρήσεις της εφαρμογής είχε παρατηρηθεί ότι η μέθοδος static είναι η αποδοτικότερη για την εφαρμογή αυτή. Στις μετρήσεις που έγιναν κανένας συνδυασμός δεν παρουσίασε καλύτερους χρόνους από την μέθοδο static, με την εξαίρεση του συνδυασμού static - Taper που είχε παρόμοιες επιδώσεις.

Τα διαγράμματα παραθέτονται στο Παράρτημα Β αν και πέρα από τον σύντομο σχολιασμό που προηγήθηκε δεν παρουσιάζουν κάτι άξιο αναφοράς.

ΚΕΦΑΛΑΙΟ 6

Σύνοψη

6.1 Σύνοψη της εργασίας

Το OpenMP έχει εδραιωθεί ως ο βασικότερος τρόπος παράλληλου προγραμματισμού σε συστήματα κοινόχρηστης μνήμης. Χάρη στην απλότητά του, και τη χρήση οδηγιών προς τον μεταφραστή, διευκολύνει τον παράλληλο προγραμματισμό και αναλαμβάνει τη δημιουργία, τον συντονισμό των νημάτων και άλλες κλήσεις για τη διασφάλιση της ορθής λειτουργίας του προγράμματος. Έτσι το σειριακό πρόγραμμα μετατρέπεται σε παράλληλο χωρίς να χρειαστεί ιδιαίτερες αλλαγές ο κώδικας.

Στα σειριακά προγράμματα ένα από τα σημεία που μπορούν να παραλληλοποιηθούν εύκολα είναι οι βρόχοι. Σε αρκετά προβλήματα εμφανίζουν μεγάλο πλήθος κλήσεων και πράξεων, αφού εκτελούνται επαναληπτικά, και συνήθως οι υπολογισμοί είναι ανεξάρτητοι μεταξύ τους. Ως αποτέλεσμα αποτελούν ένα κομμάτι του προγράμματος που όταν είναι δυνατό παραλληλοποιείται.

Το OpenMP προσφέρει την οδηγία `for`, με την οποία έναν βρόχος `for` που ακολουθεί εκτελείται παράλληλα. Όμως δεν έχουν όλοι οι βρόχοι ίδια μορφή, και έτσι είναι σημαντικό να γίνει ο κατάλληλος διαμοιρασμός ανάμεσα στα νήματα. Το OpenMP προσφέρει τρεις μεθόδους για τον διαμοιρασμό που αν και με τις κατάλληλες πα-

ραμέτρους δίνουν αποδεκτά αποτελέσματα, δεν είναι πάντα οι καλύτερες δυνατές. Έτσι έχουν μελετηθεί και αναπτυχθεί περισσότεροι τρόποι για τον διαμοιρασμό των επαναλήψεων.

Στο πλαίσιο της εργασίας δημιουργήθηκε ένας γενικός μηχανισμός για την προσθήκη νέων μεθόδων δρομολόγησης και τη χρήση τους στον μεταφραστή OMPi. Ο μηχανισμός αξιοποίησε την επιλογή δρομολόγησης `auto` για τους βρόχους του OpenMP, η οποία διαμορφώθηκε ώστε να επιλέγει κάποια πολιτική δρομολόγησης με βάση μία μεταβλητή περιβάλλοντος. Με τον μηχανισμό αυτό, υλοποιήθηκαν τέσσερεις μέθοδοι οι οποίες δοκιμάστηκαν συγκριτικά με τις προϋπάρχουσες πάνω σε κάποια προγράμματα για τη μέτρηση των επιδόσεών του. Σε αρκετές περιπτώσεις παρατηρήθηκε η βελτίωση των χρόνων εκτέλεσης.

Στη συνέχεια, έχοντας πλέον επτά διαφορετικές μεθόδους δρομολόγησης παρατηρήθηκε μία αδυναμία στις δυνατότητες των επιλογών κατά την εκτέλεση του προγράμματος. Μέσω της μεταβλητής περιβάλλοντος `OMP_SCHEDULE` ήταν δυνατό να επιλεχθεί μία μόνο μέθοδος για όλους τους βρόχους του προγράμματος. Ως λύση σε αυτό αναπτύχθηκε η δυνατότητα προσθήκης ετικετών σε κάποια δομή του OpenMP. Δίνοντας ετικέτες στους βρόχους και με τη χρήση μεταβλητών περιβάλλοντος δόθηκε η δυνατότητα ελέγχου της μεθόδου κάθε βρόχου κατά την εκτέλεση ξεχωριστά.

6.2 Συμπεράσματα

Όπως είδαμε από τις μετρήσεις στις διαφορετικές εφαρμογές που περιλαμβάνονται στα NAS Benchmarks, η καλύτερη μέθοδος δρομολόγησης για κάθε βρόχο διαφέρει. Σε μία εφαρμογή η μέθοδος που πετυχαίνει τα καλύτερα αποτελέσματα, είναι η πιο αργή σε μία άλλη. Ακόμη, οι επιδόσεις της κάθε μεθόδου επηρεάζονται και από τον αριθμό των νημάτων που έχουν δημιουργηθεί.

Έτσι είναι σημαντικό ο προγραμματιστής να αναζητήσει και να επιλέξει την καταλληλότερη για κάθε πρόβλημα μέθοδο με τις σωστές παραμέτρους ώστε να πετύχει τους καλύτερους δυνατούς χρόνους. Η καλύτερη μέθοδος σε κάποιες περιπτώσεις δεν περιλαμβάνεται σε μία από τις βασικές μεθόδους που προδιαγράφει το OpenMP. Στις μετρήσεις εμφανίστηκαν περιπτώσεις που είναι προτιμότερες οι νέες μέθοδοι που υλοποιήθηκαν. Όμως και αυτές οι μέθοδοι δεν είναι σε όλες τις περιπτώσεις αποδοτικές και πρέπει να χρησιμοποιούνται με σωστές παραμέτρους.

6.3 Μελλοντική εργασία

Το πρώτο κομμάτι της εργασίας επικεντρώθηκε στην υλοποίηση νέων μεθόδων στον μεταφραστή OMPi. Ένα προφανές επόμενο βήμα ως συνέχεια της εργασίας είναι η υλοποίηση περισσότερων μεθόδων που έχουν αναλυθεί, καθώς σε κάποιες εφαρμογές πιθανώς να πετυχαίνουν καλύτερες επιδόσεις. Κάτι τέτοιο θα είναι πλέον αρκετά πιο εύκολο καθώς ο μηχανισμός που υλοποιήθηκε είναι γενικός και παρέχει όλες τις υποδομές που απαιτούνται για τη χρήση νέων μεθόδων, με την προσθήκη μίας νέας μεταβλητής περιβάλλοντος για την επιλογή της μεθόδου.

Όσον αφορά τη βελτίωση των επιδόσεων των νέων μεθόδων, πιθανώς να εμφανίζονταν επιτάχυνση με μία μέθοδο αρχικοποίησης τιμών κατά την εμφάνιση ενός παράλληλου βρόχου. Τιμές όπως τον μέγεθος των chunk στην μέθοδο FSC ή οι μεταβλητές A και δ στην μέθοδο Trapezoid Self Scheduling δεν μπορούν να υπολογιστούν πριν τον αντίστοιχο βρόχο καθώς απαιτούν γνώση των συνολικών επαναλήψεων ή του αριθμού των νημάτων. Για αυτό ο υπολογισμός τους γίνεται στις μεθόδους που αναθέτουν το επόμενο chunk σε ένα νήμα. Αν και η τιμή τους δεν μεταβάλλεται, υπολογίζονται κάθε φορά που ένα νήμα επιχειρεί να λάβει το επόμενο chunk επαναλήψεων. Αυτό δημιουργεί ένα κόστος δρομολόγησης που πιθανώς να μπορούσε να αποφευχθεί. Με μία μέθοδο αρχικοποίησης τιμών πριν την εκτέλεση των επαναλήψεων ενός βρόχου ίσως κάποιες τιμές να μπορούν να υπολογιστούν μία μόνο φορά και να αποθηκευτούν για χρήση από όλα τα νήματα. Βέβαια, μια τέτοια αλλαγή προϋποθέτει τον συντονισμό των νημάτων, κάτι που μπορεί να αντισταθμίσει τη μείωση του κόστους.

Στο δεύτερο μέρος της εργασίας υλοποιήθηκε ο καινοτόμος μηχανισμός των ετικετών στο OpenMP. Σε συνέχεια του πρώτου μέρους οι ετικέτες χρησιμοποιήθηκαν στους βρόχους και τις μεθόδους δρομολόγησης τους. Συντακτικά όμως, μπορούν να χρησιμοποιηθούν πριν από οποιαδήποτε δομή υποστηρίζει ο OMPi. Συνεπώς θα μπορούσαν να αναζητηθούν και να υλοποιηθούν περισσότερες λειτουργίες και χρήσεις για τις ετικέτες. Όπως για παράδειγμα, αν υπάρχουν πολλές παράλληλες περιοχές με ετικέτες `<T>`, η μεταβλητή `OMP_NUM_THREADS_<T>` να καθορίζει το πλήθος των νημάτων. Κατά τη δημιουργία τους είναι δυνατό να γίνει έλεγχος για τον κώδικα που ακολουθεί και έτσι να κληθεί η κατάλληλη συνάρτηση ανάλογα με τον τρόπο χρήσης τους.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP v.2.0," in *Proc. EWOMP 2003, 5th European Workshop on OpenMP*, September 2003, pp. 5–11.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[3] T. Tzen and L. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, January 1993.

[4] S. Lucco, "A dynamic scheduling method for irregular parallel programs," *SIGPLAN Not.*, vol. 27, no. 7, p. 200–211, July 1992. [Online]. Available: https://doi.org/10.1145/143103.143134

[5] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, October 1985.

[6] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Commun. ACM*, vol. 35, no. 8, p. 90–101, August 1992. [Online]. Available: https://doi.org/10.1145/135226.135232

[7] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, vol. 8, September 1999, pp. 99–105. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.8780

[8] P. Langdal, "EPCC-OpenMP-micro-benchmarks," https://github.com/LangdalP/EPCC-OpenMP-micro-benchmarks, 2017.

[9] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The Nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, September 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[10] "NPB3.0-omp-C," https://github.com/benchmark-subsetting/NPB3.0-omp-C, November 2014.

ΠΑΡΑΡΤΗΜΑ Α

Μετασχηματισμένος κώδικας από τον OMPi

Κώδικας Α.1: Μετασχηματισμός του υπολογισμού μέσης τιμής από τον OMPi

```
1  int numbers[] = {
2    1, 8, 6, 8, 3, 7, 11, 4, 9, 10
3  };
4  static void * _thrFunc0_(void *);
5
6
7  void __original_main(int _argc_ignored, char ** _argv_ignored)
8  {
9    int i;
10   int sum = 0;
11
12   /* (l10) #pragma omp parallel reduction(+: sum) num_threads(4)
13   */
14   {
15     struct __shvt__ {
16       int (* sum);
17       unsigned long _sum_offset;
18     } _shvars;
19
20     /* reduction variables */
21     _shvars.sum = &sum;
22     ort_execute_parallel(_thrFunc0_, (void *) &_shvars, 4, 0, 0);
23     /* no other operations */
```

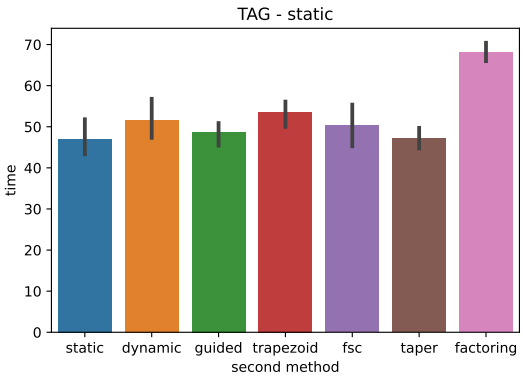
```
24    }
25    # 16 "mean.c"
26    printf("Mean = %.2f\n", (float) sum / 10);
27  }
28
29
30  static void * _thrFunc0_(void * __arg)
31  {
32    struct __shvt__ {
33      int (* sum);
34      unsigned long _sum_offset;
35    };
36    struct __shvt__ * _shvars = (struct __shvt__ *) __arg;
37
38    /* reduction variables */
39    int sum = 0;
40
41    /* no initializations */
42    /* (l10) #pragma omp parallel reduction(+: sum) num_threads(4)
43    -- body moved below */
44    # 10 "mean.c"
45    # 11 "mean.c"
46    {
47      {
48        /* (l12) #pragma omp for schedule(auto )
49        */
50        # 12 "mean.c"
51        int i;
52
53
54        unsigned long niters_ = 0, iter_ = 0, fiter_, liter_ = 0;
55        unsigned long chunksize_;
56        int (* get_chunk_)(unsigned long niters, unsigned long chunksize,
              int monotonic, unsigned long * fiter, unsigned long * liter, int
              * extra);
57        int staticextra_ = -1;
58
59        niters_ = (long) ((((10) + 1) >= (0)) ? (((10) + 1) - (0)) : 0);
60        ort_entering_for(0, 0);
61        ort_get_auto_schedule_stuff(&get_chunk_, &chunksize_);
62        while ((*get_chunk_)(niters_, chunksize_, 0, &fiter_, &liter_, &
```

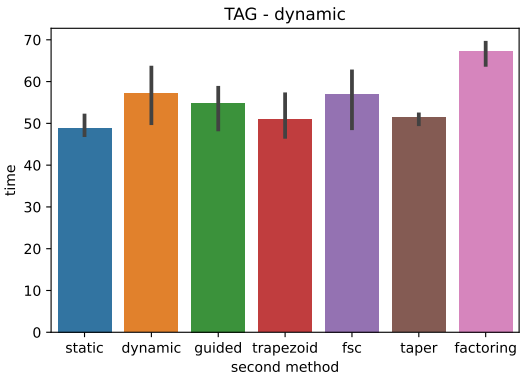
```
            staticextra_))
63        {
64          for (iter_ = fiter_, i = (0) + fiter_ * 1; iter_ < liter_; iter_
                ++, i += 1)
65          # 14 "mean.c"
66          sum += numbers[i];
67        }
68        CANCEL_for_14 :
69        ;
70        ort_leaving_for();
71        ort_fence();
72      }
73    }
74    /* reduce */
75    {
76      int * __tmp_lptr = &sum, * __tmp_gptr = (_shvars->sum);
77
78      __sync_fetch_and_add(__tmp_gptr, *__tmp_lptr);
79    }
80    CANCEL_parallel_10 :
81    ort_taskwait(2);
82    return ((void *) 0);
83  }
84
85
86
87  /* OMPi-generated main() */
88  int main(int argc, char **argv)
89  {
90    ort_initialize(&argc, &argv, 0, 1, "proc2");
91    __original_main(argc, argv);
92    ort_finalize(0);
93    return (0);
94  }
```

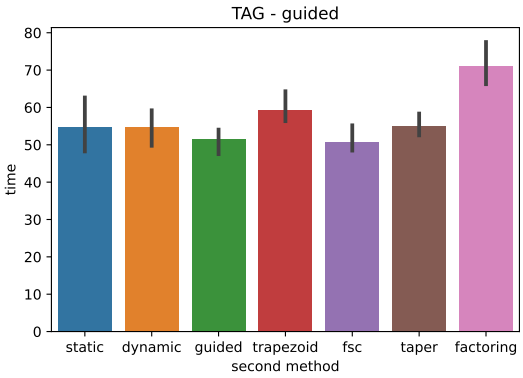
ΠΑΡΑΡΤΗΜΑ Β

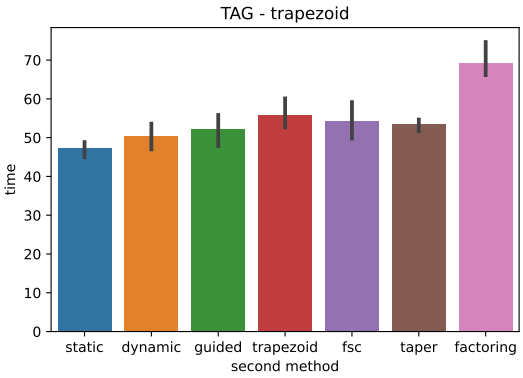
Διαγράμματα μετρήσεων της εφαρμογής BT με τη χρήση ετικετών

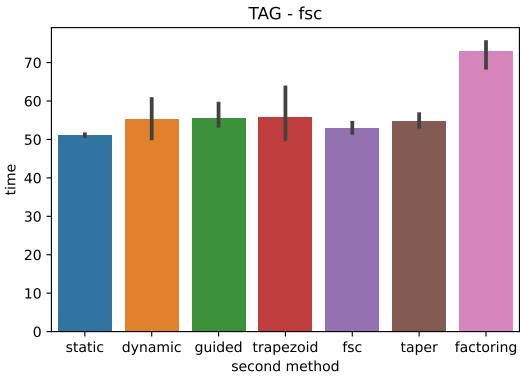
Τα διαγράμματα που ακολουθούν παραθέτονται για λόγους πληρότητας και δεν σχολιάζονται σε μεγάλο βαθμό. Οι μετρήσεις έγιναν πάνω στην εφαρμογή BT των NAS Parallel Benchmarks. Επικεντρωθήκαμε σε 6 παράλληλους βρόχους for, τους οποίους χωρίσαμε σε δύο ομάδες τριών βρόχων, καθώς είχαν όμοια μορφή. Με τη χρήση ετικετών δοκιμάστηκε κάθε δυνατός συνδυασμός μεθόδων δρομολόγησης. Κάθε διάγραμμα σχετίζεται με μία μέθοδο που χρησιμοποιήθηκε στην πρώτη ομάδα και αναφέρεται στον τίτλο του διαγράμματος. Στον οριζόντιο άξονα εμφανίζονται οι μέθοδοι που χρησιμοποιήθηκαν στην δεύτερη ομάδα βρόχων. Μία σύντομη περιγραφή των συμπερασμάτων που εξάγονται από τα διαγράμματα έγινε στην Ενότητα 5.4.

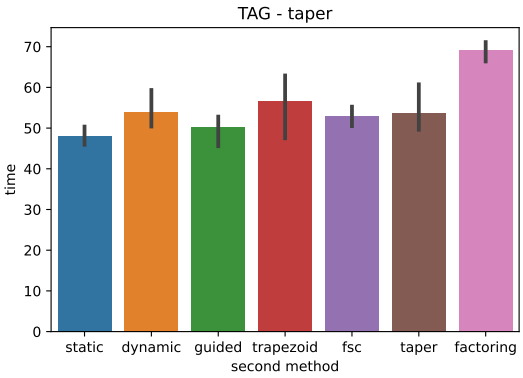

TAG - static
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring

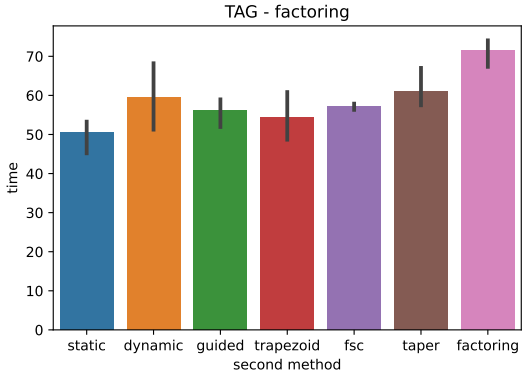

TAG - dynamic
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring


TAG - guided
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring


TAG - trapezoid
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring


TAG - fsc
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring


TAG - taper
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring


TAG - factoring
time
second method
static
dynamic
guided
trapezoid
fsc
taper
factoring