Αυτόματη Εξαγωγή Παραλληλισμού από Αναδρομικές Συναρτήσεις Βασισμένη σε Οδηγίες

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από τη Γενική Συνέλευση Ειδικής Σύνθεσης

Εξεταστική Επιτροπή

του Τμήματος Μηχανικών Η/Υ & Πληροφορικής

από τον

Αριστείδη Μάστορα

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Πανεπιστήμιο Ιωαννίνων

Ιούλιος 2013

Αυτόματη Εξαγωγή Παραλληλισμού από Αναδρομικές Συναρτήσεις Βασισμένη σε Οδηγίες

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από τη Γενική Συνέλευση Ειδικής Σύνθεσης

Εξεταστική Επιτροπή

του Τμήματος Μηχανικών Η/Υ & Πληροφορικής

από τον

Αριστείδη Μάστορα

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Πανεπιστήμιο Ιωαννίνων
Ιούλιος 2013

Στην αδερφή μου Τάνια Μαρία,
και στους γονείς μου Αγγελική και Μανώλη.

ΕΥΧΑΡΙΣΤΙΕΣ

Αρχικά θέλω να ευχαριστήσω τον επιβλέποντα της μεταπτυχιακής εργασίας, *Επίκουρο Καθηγητή* κ. Γιώργο Μανή. Η καθοδήγησή του αποτελεί για εμένα ανεκτίμητη προσφορά. Τον ευχαριστώ ιδιαιτέρως για τις πολύτιμες συμβουλές του καθώς και για την υπομονή που επέδειξε σε κάθε δυσκολία που παρουσιάστηκε. Χωρίς τη δική του συμβολή, θα ήταν αδύνατη η ολοκλήρωση της εργασίας. Επιπλέον, θέλω να ευχαριστήσω τα μέλη της τριμελούς εξεταστικής επιτροπής, *Αναπληρωτή Καθηγητή* κ. Βασίλειο Δημακόπουλο και *Επίκουρο Καθηγητή* κ. Απόστολο Ζάρρα για τις επισημάνσεις τους και για τον χρόνο που διέθεσαν για την αξιολόγησή της. Ευχαριστώ επίσης τον υποψήφιο διδάκτορα Δημήτρη Σαούγκο για την προθυμία του κάθε φορά που ζήτησα τη βοήθειά του.

Θέλω να απευθύνω τις ιδιαίτερες ευχαριστίες μου στο *Ίδρυμα Κρατικών Υποτροφιών*, στο *Κοινωφελές Ίδρυμα Αλέξανδρος Σ. Ωνάσης* και στο *Κοινωφελές Ίδρυμα Ιωάννης Σ. Λάτσης* τόσο για την ηθική όσο και για την οικονομική υποστήριξη που μου παρείχαν, επιβραβεύοντας τις επιδόσεις μου στις μεταπτυχιακές σπουδές.

Τέλος, ευχαριστώ την οικογένειά μου για την αμέριστη συμπαράσταση και την ψυχολογική υποστήριξη σε όλα τα χρόνια των σπουδών μου, και τους φίλους μου που πάντα βρίσκουν τον τρόπο να με στηρίζουν.

Η ολοκλήρωση της εργασίας αυτής έγινε στο πλαίσιο της υλοποίησης του μεταπτυχιακού προγράμματος το οποίο συγχρηματοδοτήθηκε μέσω της πράξης «Πρόγραμμα χορήγησης υποτροφιών Ι.Κ.Υ. με διαδικασία εξατομικευμένης αξιολόγησης ακαδ. Έτους 2011-2012» από πόρους του Ε.Π. «Εκπαίδευση και Δια Βίου Μάθηση» του Ευρωπαϊκού Κοινωνικού Ταμείου (ΕΚΤ) και του ΕΣΠΑ, του 2007-2013.

ΠΕΡΙΕΧΟΜΕΝΑ

Ευρετηριο Σχηματων

Ευρετηριο Πινακων

Ευρετηριο Πηγαιου Κωδικα

Περιληψη

Αριστείδης Μάστορας
Τμήμα Μηχανικών Η/Υ & Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2013
Αυτόματη Εξαγωγή Παραλληλισμού από Αναδρομικές Συναρτήσεις Βασισμένη σε Οδηγίες
Επιβλέπων: *Επίκουρος Καθηγητής* Γιώργος Μανής

Η ραγδαία εξέλιξη της τεχνολογίας έχει ως συνέπεια την ευρέως διαδεδομένη χρήση των πολυπύρηνων επεξεργαστών. Ωστόσο, η αξιοποίησή τους δε γίνεται στον επιθυμητό βαθμό καθώς τα προγράμματα θα πρέπει να υλοποιούνται με τέτοιο τρόπο ώστε να εκμεταλλεύονται την ύπαρξή τους. Η παραλληλοποίηση του κώδικα αποτελεί μία αρκετά επίπονη διαδικασία. όμως μπορεί να επιτευχθεί αυτόματα από έναν μεταφραστή απαιτώντας ελάχιστη ή και καθόλου βοήθεια από τον προγραμματιστή.

Στα πλαίσια της εργασίας υλοποιείται ο μεταφραστής *Ariadne* ο οποίος πραγματοποιεί την αυτόματη παραλληλοποίηση των αναδρομικών συναρτήσεων ενός προγράμματος, προϋποθέτοντας για κάθε συνάρτηση, μία απλή και σύντομη οδηγία που εισάγει ο προγραμματιστής. Ο μεταφραστής *Ariadne* παρέχει τρία περάσματα για τον μετασχηματισμό του κώδικα, κάθε ένα από τα οποία είναι κατάλληλο για εφαρμογή σε μία συγκεκριμένη μορφή αναδρομικών συναρτήσεων.

Το πέρασμα *elimination* πραγματοποιεί εξάλειψη της αναδρομής, μετασχηματίζοντας τη συνάρτηση από αναδρομική σε επαναληπτική, ενώ το πέρασμα *parallel-reduction* κάνει απαλοιφή της αναδρομής και κατανέμει τον φόρτο εργασίας σε έναν αριθμό νημάτων που μπορούν να εκτελεστούν παράλληλα. Το πέρασμα *thread-safe* παραλληλοποιεί τις συναρτήσεις που περιλαμβάνουν τουλάχιστον δύο ανεξάρτητες αναδρομικές κλήσεις, αντικαθιστώντας κάθε τέτοια κλήση με ένα νήμα. Η δημιουργία των νημάτων είναι επιτρεπτή μέχρι ένα ορισμένο επίπεδο επειδή η διαχείριση μεγάλου αριθμού νημάτων θα είχε ως συνέπεια τη μείωση της απόδοσης. Ο κώδικας των συναρτήσεων απεικονίζεται είτε στο μοντέλο SVP, ο προγραμματισμός του οποίου γίνεται με τη γλώσσα SL, είτε στο πρότυπο νημάτων POSIX.

Τα αποτελέσματα των πειραμάτων που διεξήχθησαν, αποδεικνύουν τη σημαντική βελτίωση που παρατηρείται στην απόδοση του κώδικα που παράγεται από την εφαρμογή των περασμάτων. σε σχέση με τον αρχικό. Γίνεται επίσης αντιληπτή η πρωτότυπη μορφή παραλληλισμού που εξάγεται με την απεικόνιση του κώδικα στο μοντέλο SVP.

ABSTRACT

Aristeidis Mastoras
Department of Computer Science & Engineering, University of Ioannina, July 2013
Directive based Parallelism Extraction from Recursive Function Calls
Thesis Supervisor: *Assistant Professor* George Manis

The rapid evolution of technology has led to the wide use of multicore CPUs. However, this technology is not exploited to the desired extent, since parallel program development is a tedious and complicated task. The parallelization of existing code is also a fairly demanding process. Fortunately, this can be achieved automatically by the compiler, requiring little or no assistance on the programmer's part.

In this thesis, the *Ariadne* compiler is described. *Ariadne* focuses on the parallelization of recursive function calls. It requires a simple and short directive given by the programmer for each function. *Ariadne* supports three passes for the transformation of the code, each one of which is appropriate for a specific kind of recursive functions.

The *elimination* pass transforms the recursive function into an iterative one, whilst the *parallel-reduction* pass eliminates the recursion and distributes the workload into a number of threads executed in parallel. The *thread-safe* pass parallelizes the recursive functions which include two independent recursive calls at least by replacing each one call with a thread. The creation of threads is allowed up to a certain level, because the management of a large number of threads could result in the reduction of performance. The code of the functions is mapped either onto the SVP model, the programming of which is made with the SL language, or onto the POSIX threads standard.

The results of the conducted experiments demonstrate the significant improvement in the performance of the code which is generated by the application of the passes, in comparison with the original code. We also exploit the special characteristics of SVP, something that allow us to achieve finer grain parallelism among threads.

Κεφαλαιο 1

Εισαγωγη

1.1 Αντικείμενο της Διατριβής

Η παραλληλοποίηση του κώδικα από τον προγραμματιστή αποτελεί μία αρκετά επίπονη και απαιτητική διαδικασία, δεν ισχύει όμως το ίδιο όταν αυτή επιτυγχάνεται με αυτόματο τρόπο από τον μεταφραστή (compiler). Οι αναδρομικές συναρτήσεις αποτελούν ένα ιδιαίτερα ενδιαφέρον κομμάτι, καθώς κρύβουν πολύ παραλληλισμό τον οποίο μπορούμε να εκμεταλλευτούμε και μεγάλες καθυστερήσεις τις οποίες μπορούμε να αποφύγουμε.

Αυτός ο παραλληλισμός είναι δύσκολο να εξαχθεί σε γενική μορφή. Υπάρχουν περιπτώσεις στις οποίες μπορεί να γίνει απολύτως αυτόματα, ενώ υπάρχουν άλλες στις οποίες απαιτείται μία σύντομη οδηγία από τον προγραμματιστή. Η οδηγία αυτή είναι τόσο σημαντική για τον μεταφραστή και συγχρόνως τόσο απλή για τον προγραμματιστή.

Έτσι, ο προγραμματιστής έχει τη δυνατότητα να γράφει σειριακό κώδικα και προσθέτοντας μόνο μία σύντομη οδηγία να παίρνει αυτόματα το παράλληλο πρόγραμμα το οποίο μπορεί να εκτελεστεί ταχύτατα σε έναν επεξεργαστή πολυπύρηνης αρχιτεκτονικής. Με αυτόν τον τρόπο η απεικόνιση του σειριακού κώδικα σε παράλληλο, γίνεται αυτόματα με ελάχιστη έως και καθόλου βοήθεια από τον προγραμματιστή. Άρα τα ήδη υπάρχοντα προγράμματα δε χρειάζεται να υλοποιηθούν από την αρχή, μπορούν απλά να παραλληλοποιηθούν.

1.2 Στόχος της Διατριβής

Η αυτόματη παραλληλοποίηση του κώδικα αποτελεί ένα ενδιαφέρον και πολλά υποσχόμενο επιστημονικό πεδίο, καθώς δεν είναι ούτε γενικός ούτε και προφανής ο τρόπος με τον οποίο μπορεί να παραλληλοποιηθεί ένα πρόγραμμα. Υπάρχουν διαφορετικά είδη κώδικα τα οποία κρύβουν παραλληλισμό και θα πρέπει να αντιμετωπιστούν με ειδικό τρόπο από τον μεταφραστή.

Αν και έχει πραγματοποιηθεί πολλή έρευνα και έχουν ήδη δημιουργηθεί μεταφραστές που εστιάζουν σε βρόχους ή σε ανεξάρτητα τμήματα που παρουσιάζουν δυνατότητα παράλληλης εκτέλεσης, το κομμάτι των αναδρομικών συναρτήσεων δεν έχει ακόμα διερευνηθεί σε βάθος, παρόλο που παρουσιάζει σημεία από τα οποία μπορεί να εξαχθεί παραλληλισμός.

Η έρευνα που έχει πραγματοποιηθεί έχει εστιάσει κυρίως σε συγκεκριμένο είδος αναδρομικής συνάρτησης, όπως αυτό της μεθόδου διαίρει και βασίλευε (divide and conquer). Υπάρχουν όμως κι άλλες μορφές, οι οποίες μελετώνται στα πλαίσια της παρούσας διατριβής και κάθε μία από αυτές απαιτεί ειδική μεταχείριση.

Ένας προφανής διαχωρισμός των αναδρομικών συναρτήσεων μπορεί να γίνει σε αυτές που είναι από τη φύση τους αναδρομικές και σε αυτές οι οποίες μπορούν να υπολογιστούν και επαναληπτικά. Στην πρώτη περίπτωση ο παραλληλισμός μπορεί να εξαχθεί μόνο εφόσον υπάρχουν δύο ή περισσότερες ανεξάρτητες κλήσεις, ενώ στη δεύτερη, αρχικά μπορεί να γίνει εξάλειψη της αναδρομής και στη συνέχεια να εξεταστεί κατά πόσο οι υπολογισμοί αυτοί μπορούν να εκτελεστούν παράλληλα, κατανέμοντάς τους σε έναν αριθμό νημάτων.

Η επιτάχυνση της εκτέλεσης ενός προγράμματος εξαρτάται σε μεγάλο βαθμό και από το μοντέλο στο οποίο θα απεικονιστεί το σειριακό πρόγραμμα. Κάθε μοντέλο προσφέρει διαφορετικές δυνατότητες τις οποίες καλείται να εκμεταλλευτεί ο μεταφραστής για να πετύχει το βέλτιστο αποτέλεσμα. Για παράδειγμα, το πρότυπο νημάτων POSIX έχει ως βασικό στοιχείο τα νήματα (thread), η δημιουργία των οποίων έχει μικρό υπολογιστικό κόστος (overhead) και μπορούν να εκτελεστούν παράλληλα. Τα νήματα αποτελούν βασικό στοιχείο και του μοντέλου SVP (Self-adaptive Virtual Processor), όμως παρουσιάζουν ουσιαστικές διαφορές από αυτά του προτύπου POSIX. Στο SVP τα νήματα οργανώνονται σε οικογένειες (family of threads) και η επικοινωνία μεταξύ των νημάτων της ίδιας οικογένειας επιτυγχάνεται ταχύτατα μέσω της μνήμης συγχρονισμού (synchronizing memory) κάτι το οποίο αποτελεί σημαντικό πλεονέκτημα του μοντέλου SVP.

Η επιλογή λοιπόν του μοντέλου στο οποίο θα γίνει η απεικόνιση του κώδικα αποτελεί σημαντικό κομμάτι έρευνας για να επιτευχθεί η βέλτιστη επιτάχυνση μεταξύ σειριακού και παράλληλου κώδικα και εξαρτάται τόσο από το είδος του κώδικα που πρόκειται να παραλληλοποιηθεί όσο και από την αρχιτεκτονική του επεξεργαστή στον οποίο θα εκτελεστεί.

Στην παρούσα διατριβή γίνεται παρουσίαση του μεταφραστή *Ariadne* ο οποίος εξειδικεύεται στην αυτόματη παραλληλοποίηση αναδρομικών συναρτήσεων κάνοντας χρήση απλών οδηγιών από τον προγραμματιστή, και αντιμετωπίζει μία σειρά από διαφορετικές μορφές που παρουσιάζουν ιδιαίτερο ενδιαφέρον.

1.3 Ανάγκη Πρόσθετης Έρευνας

Οι πολυπύρηνοι επεξεργαστές είναι διαθέσιμοι με κάθε προσωπικό Η/Υ. όμως η αξιοποίησή τους δε γίνεται στον επιθυμητό βαθμό. Αυτό συμβαίνει επειδή το λογισμικό θα πρέπει να είναι υλοποιημένο για τη συγκεκριμένη τεχνολογία επεξεργαστών, κάτι το οποίο δεν είναι εύκολο για δύο βασικούς λόγους. Πρώτον, το λογισμικό θα πρέπει να υλοποιηθεί ειδικά για πολυπύρηνα συστήματα ώστε να εκμεταλλεύεται την ανεξαρτησία μεταξύ δύο τμημάτων τα οποία μπορούν να εκτελεστούν παράλληλα. Δεύτερον, το υπάρχον λογισμικό θα πρέπει να υλοποιηθεί από την αρχή για να μπορέσει να προσαρμοστεί στην τεχνολογία των νέων επεξεργαστών.

Η λύση στο πρόβλημα δίνεται μέσω της αυτόματης παραλληλοποίησης του κώδικα, την οποία πραγματοποιεί ο μεταφραστής διευκολύνοντας τον προγραμματιστή από μία τόσο επίπονη διαδικασία. Το OpenMP είναι ένα μοντέλο το οποίο χρησιμοποιεί απλές οδηγίες από τον προγραμματιστή και επιτυγχάνει την αυτόματη παραλληλοποίηση του κώδικα. Η Cilk είναι μία επέκτασης της C με την οποία μπορεί να εξαχθεί σημαντικός βαθμός παραλληλισμού από αναδρομικές συναρτήσεις που περιλαμβάνουν ανεξάρτητους υπολογισμούς, πραγματοποιώντας ελάχιστες αλλάγες συγκριτικά με τον αντίστοιχο κώδικα σε C. Υπάρχουν κι άλλα παρόμοια μοντέλα για αυτόματη παραλληλοποίηση, αλλά δεν εξειδικεύονται στην αυτόματη παραλληλοποίηση αναδρομικών συναρτήσεων.

1.4 Δομή της Διατριβής

Η παρούσα διατριβή αποτελείται από δέκα κεφάλαια. Στα επόμενα τέσσερα αναλύεται το θεωρητικό υπόβαθρο στο οποίο βασίζεται η έρευνα που διεξήχθη στα πλαίσια της εργασίας. Πιο συγκεκριμένα, στο δεύτερο κεφάλαιο παρουσιάζεται το μοντέλο SVP, ενώ στο τρίτο γίνεται αναλυτική περιγραφή της γλώσσας SL. Στο τέταρτο κεφάλαιο πραγματοποιείται μία σύντομη αναφορά στο προτύπο νημάτων POSIX, περιγράφοντας τις βασικότερες κλήσεις του. Η ολοκλήρωση του θεωρητικού υποβάθρου γίνεται με το πέμπτο κεφάλαιο όπου καταγράφονται εργασίες για μεταφραστές παραλληλοποίησης που υπάρχουν στη βιβλιογραφία.

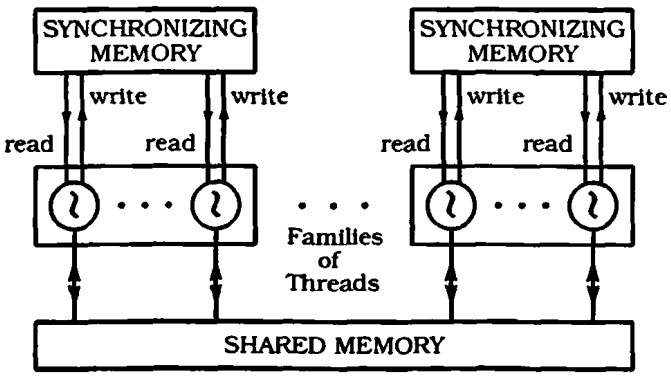
Εν συνεχεία, ακολουθούν δύο κεφάλαια τα οποία σχετίζονται με τον μεταφραστή *Ariadne* που υλοποιήθηκε στα πλαίσια της εργασίας. Έτσι, στο έκτο κεφάλαιο παρουσιάζονται λεπτομερώς θέματα για τη σχεδίαση του μεταφραστή, ενώ στο έβδομο παρουσιάζονται τεχνικές σχεδίασης που αφορούν την υλοποίησή του. Τέλος, στο όγδοο κεφάλαιο γίνεται παρουσίαση και αξιολόγηση των πειραματικών αποτελεσμάτων που λάβαμε, στο ένατο παρουσιάζονται πιθανές μελλοντικές επεκτάσεις για τον μεταφραστή *Ariadne*, ενώ στο δέκατο κεφάλαιο καταγράφονται τα συμπεράσματα από την έρευνα που πραγματοποιήθηκε.

ΚΕΦΑΛΑΙΟ 2

SVP

2.1 Το Μοντέλο SVP (Self - adaptive Virtual Processor)

Το SVP (Self - adaptive Virtual Processor) [1, 2, 3, 4] είναι μοντέλο πολυπύρηνης αρχιτεκτονικής (multi-core architecture) το οποίο υποστηρίζει τόσο την παράλληλη εκτέλεση μίας οικογένειας νημάτων (family of threads) αναθέτοντας τα νήματα (thread) σε πολλούς πυρήνες, όσο και την ταυτόχρονη εκτέλεση πολλών οικογενειών.



Σχήμα 2.1: Η αρχιτεκτονική του SVP.

Επειδή η διαχείριση και η επικοινωνία μεταξύ των νημάτων υλοποιείται από υλικό, θεωρούμε ελάχιστο το κόστος της δημιουργίας, της διαχείρισης, και γενικότερα της επικοινωνίας μεταξύ των νημάτων της ίδιας οικογένειας. Η επικοινωνία μεταξύ των νημάτων μίας οικογένειας πραγματοποιείται μέσω της γρήγορης μνήμης συγχρονισμού (synchronizing memory), ενώ υπάρχει και κοινόχρηστη μνήμη (shared memory).

2.2 Οικογένεια Νημάτων

Μία οικογένεια νημάτων είναι μία μονάδα εργασίας η οποία έχει συγκεκριμένα χαρακτηριστικά και ορίζεται ως ένα διατεταγμένο σύνολο από πανομοιότυπα νήματα. Η έννοια αυτή είναι που κάνει το μοντέλο SVP να διαφέρει από κάθε άλλο πολυνηματικό μοντέλο. Μία οικογένεια νημάτων δημιουργείται από ένα νήμα και έχει παραμέτρους μέσω των οποίων γίνεται η επικοινωνία με το νήμα αυτό. Κάθε νήμα μίας οικογένειας έχει τη δυνατότητα να δημιουργήσει τις δικές του οικογένειες νημάτων. Επομένως, γίνεται εύκολα αντιληπτό ότι το μοντέλο SVP υποστηρίζει τη δημιουργία μίας ιεραρχίας από οικογένειες νημάτων, σε πολλαπλά επίπεδα.

Κάθε νήμα μίας οικογένειας γνωρίζει τη θέση του μέσα από το μοναδικό αναγνωριστικό (id) που το χαρακτηρίζει. Η μοναδικότητα του αναγνωριστικού, του δίνει τη δυνατότητα να διαφοροποιηθεί από τα υπόλοιπα νήματα της ίδιας οικογένειας. Για παράδειγμα, στον κώδικα των νημάτων μπορεί να χρησιμοποιηθεί μία λογική συνθήκη η οποία βασιζόμενη στο αναγνωριστικό κάθε νήματος να ορίζει διαφορετικό τμήμα κώδικα προς εκτέλεση.

Κάθε νήμα περιλαμβάνει μία σειρά από λειτουργίες που είναι ορισμένες σε ένα σύνολο βαθμωτών μεταβλητών που δημιουργούνται με τη δημιουργία του νήματος, ενώ παύουν να υφίστανται με τον τερματισμό του. Με τις λειτουργίες αυτές γίνεται ο συγχρονισμός και η επικοινωνία μεταξύ των νημάτων μίας οικογένειας. Συγκεκριμένα, η επικοινωνία μεταξύ των νημάτων γίνεται μέσω των κοινόχρηστων μεταβλητών (shared variable) οι οποίες βρίσκονται στη μνήμη συγχρονισμού. Ένα νήμα μπορεί να επικοινωνεί μόνο με τα δύο γειτονικά του νήματα στο διατεταγμένο σύνολο. Δηλαδή με το προηγούμενο και το επόμενο. Από το προηγούμενο νήμα διαβάζει (read) δεδομένα, ενώ στο επόμενο γράφει (write). Καμία άλλη μορφή επικοινωνίας δεν μπορεί να υπάρξει ανάμεσα στα νήματα. Χαρακτηριστικά μίας οικογένειας νημάτων αποτελούν οι είσοδοι που λαμβάνει κατά τη δημιουργία της, αλλά και οι έξοδοι που επιστρέφει μετά την ολοκλήρωσή της. Οι είσοδοι είναι οι τιμές στις οποίες θα αρχικοποιηθούν οι κοινόχρηστες μεταβλητές του πρώτου νήματος, ενώ οι έξοδοι είναι οι τιμές που θα γράψει το τελευταίο νήμα σε αυτές.

2.3 Μνήμη Συγχρονισμού

Η μνήμη συγχρονισμού παρέχει έναν μηχανισμό που επιτρέπει τον συγχρονισμό και την επικοινωνία μεταξύ των νημάτων μίας οικογένειας με μεγάλη ταχύτητα, λόγω του ότι είναι κοντά στον επεξεργαστή. Κάθε νήμα δεσμεύει ένα πλαίσιο στη μνήμη συγχρονισμού, στο

οποίο βρίσκονται οι τοπικές μεταβλητές του για όση ώρα είναι ενεργό. Επειδή το μέγεθος της μνήμης συγχρονισμού είναι περιορισμένο, είναι πιθανόν κατά τη διάρκεια εκτέλεσης μίας οικογένειας, να υπάρχουν στη μνήμη οι μεταβλητές μόνο μερικών νημάτων. Τα υπόλοιπα νήματα θα αποκτήσουν τον δικό τους χώρο στη μνήμη, μόλις ελευθερωθεί κάποιο τμήμα της.

Όλες οι λειτουργίες στο μοντέλο SVP εκτελούνται πάνω σε δεδομένα που βρίσκονται στη μνήμη συγχρονισμού. Τα δεδομένα που βρίσκονται στην κοινόχρηστη μνήμη μεταφέρονται στη μνήμη συγχρονισμού πριν την εκτέλεση των λειτουργιών. Κάθε θέση στη μνήμη συγχρονισμού συνοδεύεται με αποκλεισμό κατά την ανάγνωση ο οποίος εξυπηρετεί τον συγχρονισμό των νημάτων.

Τα νήματα μίας οικογένειας θα πρέπει να έχουν εξαρτήσεις μόνο από το προηγούμενό τους νήμα. το οποίο σημαίνει ότι οι εξαρτήσεις της οικογένειας δημιουργούν ένα άκυκλο γράφημα (acyclic graph). Το γράφημα αυτό αρχικοποιείται με τη δημιουργία της οικογένειας και ο συγχρονισμός των νημάτων γίνεται με τις κοινόχρηστες μεταβλητές που υπάρχουν σε θέσεις της μνήμης συγχρονισμού. Οι εξαρτήσεις μεταξύ των νημάτων εξυπηρετούνται μέσω των κοινόχρηστων μεταβλητών με τις οποίες πραγματοποιείται η επικοινωνία. Τα δεδομένα του τελευταίου νήματος μπορούν να διαβαστούν από το νήμα που δημιούργησε την οικογένεια νημάτων, όταν αυτή ολοκληρωθεί.

2.4 Κοινόχρηστη Μνήμη

Για να αποθηκεύσουμε δεδομένα μεγάλου μεγέθους ή ακόμα και για να τα περάσουμε ως όρισμα, υπάρχει η ανάγκη ύπαρξης μίας επιπλέον μνήμης. Ονομάζεται κοινόχρηστη μνήμη, και τα νήματα όλων των οικογενειών έχουν τη δυνατότητα ανάγνωσης και εγγραφής από και προς αυτήν. Συνέπεια μνήμης υπάρχει μόνο αμέσως πριν και αμέσως μετά την ολοκλήρωση μίας οικογένειας νημάτων. Σε οποιοδήποτε άλλο χρονικό σημείο η κατάσταση της μνήμης δεν μπορεί να προσδιοριστεί με βεβαιότητα. Επομένως, κατά το πέρασμα παραμέτρων που βρίσκονται στην κοινόχρηστη μνήμη, η αρχικοποίηση των αντίστοιχων ορισμάτων θα πρέπει να γίνεται πριν τη δημιουργία της οικογένειας νημάτων. Το ίδιο ισχύει και για δεδομένα τα οποία ενδεχομένως να επιστρέφονται στο νήμα που δημιούργησε την οικογένεια, οπότε η ανάγνωσή τους μπορεί να γίνει μόνο εφόσον ολοκληρωθεί η εκτέλεση της οικογένειας. Αυτή η ασυνέπεια μνήμης οφείλεται σε πιθανές εγγραφές που πραγματοποιούν τα νήματα κατά την ταυτόχρονη εκτέλεσή τους.

ΚΕΦΑΛΑΙΟ 3

Η ΓΛΩΣΣΑ SL

3.1 Εισαγωγή

Η SL [5, 6] είναι μία επέκταση της γλώσσας προγραμματισμού ISO C 99/C 11 η οποία σχεδιάστηκε με στόχο να είναι μία διεπαφή γενικού σκοπού. Η SL επιτρέπει τον προγραμματισμό της αρχιτεκτονικής του μοντέλου SVP. Ως επέκταση της C, γίνεται εύκολα αντιληπτό ότι κάθε πρόγραμμα σε C είναι αποδεκτό και από τη γλώσσα SL. Στον Πηγαίο Κώδικα 3.1 φαίνεται ένα απλό πρόγραμμα σε SL το οποίο εμφανίζει το μήνυμα "Hello World!".

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World!\n");
6
7     return (0);
8 }
```

Πηγαίος Κώδικας 3.1: Πρόγραμμα Hello World στη γλώσσα SL.

Στις ενότητες που ακολουθούν γίνεται περιγραφή της γλώσσας SL. Όμως, στα πλαίσια της παρούσας διατριβής θα μας απασχολήσει μόνο ένα μέρος της γλώσσας το οποίο και παρουσιάζεται στον Πίνακα 3.1

sl_create()	Δημιουργία οικογένειας νημάτων.
sl_sync()	Συγχρονισμός οικογένειας νημάτων.
sl_def() { ... } sl_enddef	Ορισμός συνάρτησης νημάτων.
sl_*parm()	Δήλωση παραμέτρου σε συνάρτηση νημάτων, όπου * είναι sh για κοινόχρηστη ή gl για καθολική, ενώ σε περίπτωση τύπου κινητής υποδιαστολής, ακολουθεί το γράμμα f.
sl_*arg()	Δήλωση ορίσματος στην sl_create(), όπου * είναι sh για κοινόχρηστο ή gl για καθολικό, ενώ σε περίπτωση τύπου κινητής υποδιαστολής, ακολουθεί το γράμμα f.
sl_getp()	Ανάγνωση καθολικής ή κοινόχρηστης παραμέτρου εντός του σώματος της συνάρτησης νημάτων.
sl_setp()	Εγγραφή κοινόχρηστης παραμέτρου εντός του σώματος της συνάρτησης νημάτων.
sl_geta()	Ανάγνωση κοινόχρηστου ορίσματος μετά την ολοκλήρωση μίας οικογένειας νημάτων.
sl_index()	Δήλωση μεταβλητής και εκχώρηση αναγνωριστικού.
sl_def(t_main, void) { ... } sl_enddef	Ορισμός της συνάρτησης main.

Πίνακας 3.1: Αντικείμενα της SL που θα μας απασχολήσουν στα πλαίσια της παρούσας διατριβής.

3.2 Αντικείμενα της SL

3.2.1 Δημιουργία Οικογένειας Νημάτων

Η βασική ιδέα της SL είναι η δημιουργία οικογενειών νημάτων με στόχο την αξιοποίηση του μοντέλου SVP. Η δημιουργία μίας οικογένειας γίνεται με την κλήση sl_create(). Η οικογένεια νημάτων που θα δημιουργηθεί, θα αποτελείται από ένα σύνολο διατεταγμένων και πανομοιότυπων νημάτων, τα οποία γνωρίζουν τη θέση τους μέσω του αναγνωριστικού (id) που τα χαρακτηρίζει (είναι μοναδικό για κάθε νήμα μέσα στην οικογένεια). Η sl_create() παίρνει μία σειρά από ορίσματα, αλλά στα πλαίσια της παρούσας διατριβής θα μας απασχολήσουν μόνο μερικά από αυτά, και συγκεκριμένα τα start, limit, step, fexp και args.

sl_create(, place, start, limit, step, block, , fexp, args...);

Με τα start, limit και step, ορίζεται ο αριθμός των νημάτων και τα αναγνωριστικά τους. Το πρώτο νήμα που θα δημιουργηθεί θα έχει ως αναγνωριστικό το "start", το δεύτερο νήμα θα έχει το "start + step", το τρίτο νήμα θα έχει το "start + 2 * step" κ.ο.κ.. Ο αριθμός των νημάτων που δημιουργούνται εξαρτάται από το limit το οποίο λειτουργεί ως φράγμα. Οι τιμές των start, limit και step μπορεί να είναι είτε θετικές είτε αρνητικές, ως εκ τούτου τα αναγνωριστικά των νημάτων μπορεί να έχουν και αρνητική τιμή, αλλά πάντα ακέραια.

Το fexp είναι το όνομα της συνάρτησης που θα εκτελέσουν τα νήματα της οικογένειας. Δεν είναι μία κοινή συνάρτηση της C, ο ορισμός της γίνεται με ειδικό τρόπο. Τέλος, τα args είναι μία σειρά από ορίσματα που θα δοθούν στη συνάρτηση των νημάτων, fexp. Τα ορίσματα δίνονται και αυτά σε ειδική μορφή και μπορούν να είναι είτε κοινόχρηστες βαθμωτές (scalar) μεταβλητές τις οποίες χρησιμοποιούν τα νήματα μίας οικογένειας για τη μεταξύ τους επικοινωνία είτε καθολικές μεταβλητές που έχουν την έννοια των συνηθισμένων ορισμάτων της C. Οι τιμές των start, limit και step είναι προαιρετικές. Οι προεπιλεγμένες τους τιμές εφόσον δεν καθοριστούν, είναι 0, 1 και 1 αντίστοιχα.

3.2.2 Αναμονή Οικογένειας Νημάτων

Μετά από κάθε δημιουργία μίας οικογένειας νημάτων, δηλαδή μετά από κάθε κλήση sl_create() πρέπει να υπάρχει η κλήση sl_sync(), με την οποία γίνεται ο συγχρονισμός όλων των νημάτων της οικογένειας. Το μοναδικό όρισμά της είναι το expression το οποίο αντιστοιχεί στον κωδικό εξόδου της οικογένειας και πρέπει να είναι ακέραια τιμή. Μπορεί ακόμα και να παραλειφθεί.

sl_sync([expression]);

Με αυτόν τον τρόπο μπορούμε να περιμένουμε να ολοκληρωθεί η εκτέλεση ολόκληρης της οικογένειας πριν συνεχιστεί η εκτέλεση του υπόλοιπου κώδικα. Μεταξύ των sl_create() και sl_sync() μπορεί να υπάρχει άλλος κώδικας, αλλά και οι δύο κλήσεις θα πρέπει να βρίσκονται στο ίδιο επίπεδο ενός μπλοκ της C.

3.2.3 Αποδέσμευση Οικογένειας Νημάτων

Αν δε θέλουμε να περιμένουμε την εκτέλεση μίας οικογένειας νημάτων, τότε μπορούμε να το δηλώσουμε μέσω της κλήσης sl_detach(). Στα πλαίσια της παρούσας διατριβής πάντα υπάρχει η ανάγκη να ολοκληρωθεί μία οικογένεια νημάτων, ως εκ τούτου δε γίνεται χρήσης της sl_detach().

sl_detach();

3.2.4 Χειριστής Οικογένειας Νημάτων

Η δημιουργία μίας οικογένειας νημάτων μέσω της sl_create() δεν προσφέρει κάποιον τρόπο μέσω του οποίου να μπορούμε να διαχειριστούμε τη συγκεκριμένη οικογένεια. Για να επιτευχθεί αυτό, θα πρέπει αρχικά να δηλώσουμε έναν χειριστή (handler) για μία οικογένεια με την κλήση sl_spawndecl(). Το μοναδικό όρισμά της είναι το identifier, δηλαδή το όνομα του χειριστή που δηλώνουμε.

sl_spawndecl(identifier);

Στη συνέχεια δημιουργούμε την οικογένεια νημάτων με την κλήση sl_spawn(), και όχι με την sl_create(). Τα ορίσματα είναι ίδια με αυτά της sl_create(), εκτός από το πρώτο, το οποίο στην sl_spawn() είναι το όνομα του χειριστή.

sl_spawn(identifier, place, **start**, **limit**, **step**, block, , **fexp**, **args...**);

Ο συγχρονισμός όλων των νημάτων μίας οικογένειας που δημιουργήθηκε με την sl_spawn(), γίνεται με την κλήση sl_spawnsync() η οποία παίρνει ως όρισμα τον χειριστή της αντίστοιχης οικογένειας.

sl_spawnsync(identifier);

3.2.5 Ορισμός Συνάρτησης Νημάτων

Ο ορισμός της συνάρτησης που εκτελεί μία οικογένεια νημάτων αρχίζει με τη δεσμευμένη λέξη sl_def ενώ τελειώνει με την sl_enddef. Χαρακτηρίζεται από ένα όνομα fexp και τις παραμέτρους parms. Μεταξύ του ονόματος και των παραμέτρων μπορεί να υπάρχει η λέξη sl_static με την οποία υποδεικνύεται η στατική εμβέλεια της συνάρτησης η οποία ισοδυναμεί με τη static εμβέλεια μίας συνάρτησης στη C.

```
1  sl_def(fexp, void, parms...)
2  {
3
4  }
5  sl_enddef
6
7  int main()
8  {
9      sl_create(, , start, limit, step, , , fexp, args...);
10     sl_sync();
11
12     return (0);
13 }
```

Πηγαίος Κώδικας 3.2: Δημιουργία μίας οικογένειας νημάτων στη γλώσσα SL.

Όπως κατά την κλήση της sl_create() όπου τα ορίσματα δίνονται σε ειδική μορφή, έτσι και εδώ, οι παράμετροι βρίσκονται επίσης σε ειδική μορφή. Σε περίπτωση που δεν υπάρχουν

παράμετροι, τότε το αντίστοιχο τμήμα της δήλωσης παραμένει κενό. Στο σώμα της συνάρτησης δεν υπάρχουν διαφορές σε σχέση με μία συνάρτηση της C, όμως μπορεί να εμπεριέχεται οποιαδήποτε από τις δεσμευμένες λέξεις της SL που αναφέρονται στον Πίνακα 3.2.

Στο σημείο αυτό αξίζει να σημειωθεί ότι η παραπάνω μορφή ορισμού των συναρτήσεων, αφορά μόνο τις συναρτήσεις τις οποίες εκτελεί μία οικογένεια νημάτων της SL. Ο ορισμός μίας κοινής συνάρτησης σε C γίνεται όπως και στη C και έχει ακριβώς την ίδια έννοια. Η διαφορά έγκειται στο γεγονός ότι οι κοινές συναρτήσεις της C δεν μπορούν να κληθούν από μία οικογένεια νημάτων, αλλά και ότι μία συνάρτηση της SL δεν μπορεί να κληθεί όπως μία κοινή συνάρτηση της C.

3.2.6 Ορίσματα και Παράμετροι Συνάρτησης Νημάτων

Οι παράμετροι μίας συνάρτησης νημάτων είναι κοινόχρηστες (shared) ή καθολικές (global). Οι κοινόχρηστες χρησιμοποιούνται για τον συγχρονισμό και την επικοινωνία των νημάτων μίας οικογένειας και είναι μόνο βαθμωτά (scalar) μεγέθη. Από την άλλη, οι καθολικές χρησιμοποιούνται με την κλασική έννοια των παραμέτρων και μπορούν να είναι είτε βαθμωτά μεγέθη είτε πίνακες (array). Οι καθολικές βρίσκονται στην κοινόχρηστη μνήμη και προσφέρονται μόνο για ανάγνωση, ενώ οι κοινόχρηστες βρίσκονται στη μνήμη συγχρονισμού και επιδέχονται και εγγραφή.

Η δήλωση μίας καθολικής παραμέτρου γίνεται με τη λέξη sl_glparm, εκτός αν ο τύπος της παραμέτρου είναι αριθμός κινητής υποδιαστολής (floating-point) όπου η δήλωση γίνεται με τη λέξη sl_glfparm. Η δήλωση μίας κοινόχρηστης παραμέτρου γίνεται με τις λέξεις sl_shparm και sl_shfparm αντίστοιχα. Η γενική μορφή των δηλώσεων φαίνεται παρακάτω, όπου declaration-specifiers είναι ουσιαστικά ο τύπος της παραμέτρου και identifier το όνομά της. Στο declaration-specifiers δεν πρέπει να εμπεριέχεται καμία δεσμευμένη λέξη που ανήκει στην κατηγορία των class-specifier, δηλαδή καμία εκ των typedef, extern, static, auto και register. Ο ίδιος περιορισμός ισχύει και για τη δεσμευμένη λέξη volatile η οποία ανήκει στην κατηγορία των type-qualifier.

Η πρόσβαση σε κάποια από τις παραμέτρους δεν μπορεί να γίνει μέσω του ονόματος, αλλά απαιτεί μία απλή διαδικασία στην οποία θα αναφερθούμε στη συνέχεια.

sl_glparm(declaration-specifiers, identifier)
sl_glfparm(declaration-specifiers, identifier)
sl_shparm(declaration-specifiers, identifier)
sl_shfparm(declaration-specifiers, identifier)

Τα ορίσματα της συνάρτησης δίνονται μέσω της κλήσης sl_create() κατά τη δημιουργία της οικογένειας νημάτων. Η δήλωση των ορισμάτων κατά την κλήση sl_create() γίνεται μέσω των λέξεων sl_glarg, sl_glfarg, sl_sharg και sl_shfarg, η σημασία των οποίων είναι παρόμοια με αυτή για τη δήλωση των αντίστοιχων παραμέτρων που περιγράφεται παραπάνω. Κατά τη δήλωση ενός ορίσματος θα πρέπει επιπλέον να δοθεί μία αριθμητική έκφραση (expression) με την οποία θα αρχικοποιηθεί η αντίστοιχη παράμετρος. Επομένως, το identifier δεν έχει την έννοια του ορίσματος, αλλά μίας κωδικής λέξης μέσω της οποίας γίνεται το πέρασμα του

expression το οποίο αποτελεί το πραγματικό όρισμα. Το expression μπορεί να παραλειφθεί στην περίπτωση κοινόχρηστου ορίσματος, καθώς μπορεί να οριστεί και με διαφορετικό τρόπο στον οποίο θα αναφερθούμε στη συνέχεια. Η παράλειψη αυτή δεν μπορεί να γίνει όταν η οικογένεια νημάτων δημιουργείται με την κλήση sl_spawn(). Στα πλαίσια της παρούσας διατριβής η παράλειψη δεν πραγματοποιείται ποτέ.

sl_glarg(declaration-specifiers, identifier, expression)
sl_glfarg(declaration-specifiers, identifier, expression)
sl_sharg(declaration-specifiers, identifier[, expression])
sl_shfarg(declaration-specifiers, identifier[, expression])

Στον Πηγαίο Κώδικα 3.3 παρουσιάζεται ένα παράδειγμα προγράμματος σε SL στο οποίο δημιουργείται μία οικογένεια νημάτων που εκτελεί μία συνάρτηση με δύο παραμέτρους. Η πρώτη παράμετρος είναι καθολική, ενώ η δεύτερη κοινόχρηστη.

```
1  sl_def(fexp, void, sl_glparm(declaration-specifiers, identifier),
2                     sl_shparm(declaration-specifiers, identifier))
3  {
4
5  }
6  sl_enddef
7
8  int main()
9  {
10     sl_create(, , start, limit, step, , , fexp,
11                       sl_glarg(declaration-specifiers, identifier, expression),
12                       sl_sharg(declaration-specifiers, identifier, expression));
13     sl_sync();
14
15     return (0);
16 }
```

Πηγαίος Κώδικας 3.3: Δήλωση παραμέτρων και ορισμάτων στην SL.

3.2.7 Πρόσβαση στις Παραμέτρους Συνάρτησης Νημάτων

Η πρόσβαση τόσο στις κοινόχρηστες όσο και στις καθολικές παραμέτρους μίας συνάρτησης, απαιτεί μία διαδικασία ανάγνωσης πριν την πρώτη χρήση. Μία κοινόχρηστη παράμετρος χρησιμοποιείται για την επικοινωνία μεταξύ των νημάτων μίας οικογένειας. Κάθε νήμα λοιπόν έχει την ανάγκη να διαβάσει αλλά και να τροποποιήσει την τιμή μίας τέτοιας παραμέτρου. Η ανάγνωσή της γίνεται με την κλήση sl_getp() η οποία λειτουργεί με αποκλεισμό. Το μοναδικό όρισμα της sl_getp() είναι το όνομα της παραμέτρου.

sl_getp(identifier);

Όπως έχει ήδη αναφερθεί κατά την περιγραφή του μοντέλου SVP, κάθε νήμα έχει τη δυνατότητα να επικοινωνήσει μόνο με τους δύο γείτονές του. Από τον προηγούμενο διαβάζει, ενώ στον επόμενο γράφει. Άρα, θεωρείται αναγκαία η ύπαρξη αποκλεισμού ώστε να

επιτευχθεί ο συγχρονισμός μεταξύ των νημάτων. Η εγγραφή σε μία κοινόχρηστη παράμετρο γίνεται με την κλήση sl_setp(). Ουσιαστικά γράφει την τιμή την οποία θα διαβάσει το επόμενο νήμα στο διατεταγμένο σύνολο. Η sl_setp() έχει δύο ορίσματα, το πρώτο είναι το identifier το οποίο αντιστοιχεί στο όνομα της παραμέτρου στην οποία θα γίνει η εγγραφή και το δεύτερο είναι το expression το οποίο αντιστοιχεί στην αριθμητική έκφραση η οποία θα γραφτεί στην παράμετρο.

sl_setp(identifier, expression);

Ένας πολύ σημαντικός αλλά και λογικός περιορισμός για την κλήση sl_setp() είναι ότι δε θα πρέπει να κληθεί παραπάνω από μία φορά στον κώδικα που εκτελεί κάθε νήμα. Διαφορετικά δε θα μπορούσε να υπάρξει συγχρονισμός μεταξύ των νημάτων, καθώς η κλήση sl_getp() λειτουργεί με αποκλεισμό.

Το σημείο στο οποίο θα γίνει η κλήση των sl_setp() και sl_getp() απαιτεί μεγάλη προσοχή λόγω του αποκλεισμού που δημιουργείται κατά την κλήση της sl_getp(). Αν η κλήση της sl_getp() γίνει στην αρχή της συνάρτησης των νημάτων και η κλήση sl_setp() γίνει στο τέλος, τότε αυτό θα έχει ως αποτέλεσμα την εκτέλεση του κώδικα σειριακά και όχι παράλληλα. Είναι πολύ πιθανόν ένα νήμα να χρειαστεί να περιμένει το προηγούμενο του (στο διατεταγμένο σύνολο), ώστε να γράψει στην κοινόχρηστη παράμετρο, κάτι το οποίο είναι καταστροφικό για τον παραλληλισμό. Για να μειώσουμε αυτήν την πιθανότητα, θα πρέπει καταρχήν κάθε νήμα να εκτελέσει όσο το δυνατόν περισσότερο κώδικα πριν γίνει η κλήση της sl_getp(), ώστε να εκμεταλλευτεί τη δυνατότητα παράλληλης εκτέλεσης, και με την πρώτη ευκαιρία να καλέσει την sl_setp() ώστε να μειώσουμε την καθυστέρηση που θα έχει το επόμενο νήμα, περιμένοντας το τρέχον να καλέσει την sl_setp().

Όσον αφορά τις καθολικές μεταβλητές δεν υπάρχει η έννοια της εγγραφής αλλά μόνο της ανάγνωσης. Η ανάγνωση και σε αυτήν την περίπτωση γίνεται με την κλήση sl_getp(), με τη διαφορά ότι δεν υπάρχει αποκλεισμός και η κλήση της δεν επηρεάζει την επίτευξη της παραλληλίας. Άρα μπορεί να κληθεί σε οποιοδήποτε σημείο.

3.2.8 Πρόσβαση στα Ορίσματα Συνάρτησης Νημάτων

Μετά την ολοκλήρωση της εκτέλεσης μίας οικογένειας νημάτων υπάρχει η ανάγκη να διαβάσουμε την τιμή που έγραψε το τελευταίο νήμα σε μία κοινόχρηστη μεταβλητή. Την τιμή αυτή μπορούμε να την πάρουμε με την κλήση sl_geta().

sl_geta(identifier);

Όπως έχουμε ήδη αναφέρει κάθε καθολικό ή κοινόχρηστο όρισμα αρχικοποιείται κατά τη δημιουργία της οικογένειας νημάτων, μέσω της κλήσης sl_create(). Η τιμή με την οποία αρχικοποιείται ένα καθολικό όρισμα είναι αυτή που θα διαβάσει κάθε ένα από τα νήματα της οικογένειας. Στην περίπτωση του κοινόχρηστου ορίσματος, η τιμή με την οποία αρχικοποιείται, είναι αυτή που θα διαβάσει το πρώτο νήμα και μόνο, καθώς τα υπόλοιπα νήματα θα

διαβάσουν την παράμετρο, αφού πρώτα την τροποποιήσει το προηγούμενό τους νήμα. Επομένως, η κλήση sl_geta() ουσιαστικά θα μας δώσει την τιμή που έγραψε το τελευταίο νήμα της οικογένειας στην κοινόχρηστη παράμετρο. Αν δεν ορισθεί κάποια αρχική τιμή κατά την κλήση της sl_create() για ένα κοινόχρηστο όρισμα, μπορεί να ορισθεί στη συνέχεια, αλλά πριν την κλήση της sl_sync(), μέσω της sl_seta(), όπου identifier είναι το όνομα του ορίσματος και expression μία αριθμητική έκφραση με την οποία θα αρχικοποιηθεί. Η sl_seta() δεν μπορεί να χρησιμοποιηθεί όταν η δημιουργία της οικογένειας νημάτων έγινε με την sl_spawn().

sl_seta(identifier, expression);

Στον Πηγαίο Κώδικα 3.4 πραγματοποιείται πρόσβαση στις παραμέτρους της συνάρτησης f_sum. Η ανάγνωση της καθολικής παραμέτρου x γίνεται στην αρχή με την κλήση sl_getp(), ενώ η ανάγνωση και η εγγραφή της κοινόχρηστης παραμέτρου sum γίνεται με τέτοιο τρόπο ώστε ο υπολογισμός της συνάρτησης foo() να γίνει παράλληλα από όλα τα νήματα της οικογένειας. Η δυνατότητα αυτή αποτελεί ένα πρωτότυπο χαρακτηριστικό του μοντέλου SVP και της γλώσσας SL, το οποίο αξιοποιούμε στην παρούσα διατριβή.

```
1  #include <stdio.h>
2
3  extern int foo();
4
5  sl_def(f_sum, void, sl_glparm(int, x), sl_shparm(int, sum))
6  {
7      int x, z, sum;
8
9      x = sl_getp(x);
10     z = foo();
11     sum = sl_getp(sum);
12     sl_setp(sum, sum + x * z);
13 }
14 sl_enddef
15
16 int main()
17 {
18     sl_create(, , 0, 10, 1, , , f_sum, sl_glarg(int, x, 10), sl_sharg(int, sum, 0));
19     sl_sync();
20
21     printf("sum = %d\n", sl_geta(sum));
22
23     return (0);
24 }
```

Πηγαίος Κώδικας 3.4: Παράδειγμα ανάγνωσης και εγγραφής παραμέτρων στην SL.

3.2.9 Αναγνωριστικό Νημάτων

Στην SL, κάθε νήμα μίας οικογένειας χαρακτηρίζεται από ένα αναγνωριστικό που το ταυτοποιεί. Μέσα από το αναγνωριστικό αυτό γνωρίζει τη θέση του στο διατεταγμένο σύνολο, ενώ μπορεί να το μάθει με την κλήση sl_index(). Η sl_index() έχει ένα μοναδικό όρισμα το οποίο δε θα πρέπει να αντιστοιχεί σε όνομα κάποιας μεταβλητής, καθώς η δήλωση της

μεταβλητής και η εγγραφή του αναγνωριστικού σε αυτήν, πραγματοποιούνται κατά την κλήση της. Ο τύπος της μεταβλητής που δηλώνεται είναι ακέραιος, αλλά η τιμή της δεν είναι απαραίτητα θετική.

sl_index(identifier);

3.2.10 Διακοπή Εκτέλεσης

Αν υπάρχει return εντός της συνάρτησης νημάτων, τότε γίνεται τερματισμός του συγκεκριμένου νήματος μόλις φτάσει στο σημείο αυτό. Η χρήση του return θα πρέπει να αποφεύγεται μεταξύ των κλήσεων δημιουργίας και συγχρονισμού μίας οικογένειας νημάτων, καθώς το αποτέλεσμα είναι απροσδιόριστο.

Το sl_break χρησιμοποιείται εντός μίας συνάρτησης νημάτων και έχει ως αποτέλεσμα τη διακοπή δημιουργίας νέων νημάτων. Τα ήδη υπάρχοντα νήματα συνεχίζουν κανονικά την εκτέλεσή τους.

3.2.11 Ορισμός της Κύριας Συνάρτησης

Η συνάρτηση main() έχει τη δική της μορφή στη γλώσσα SL. Όπως στις συναρτήσεις που εκτελεί μία οικογένεια νημάτων, έτσι και σε αυτήν, ο ορισμός της ξεκινάει με τη λέξη sl_def και ολοκληρώνεται με τη λέξη sl_enddef. Συγκεκριμένα, για τη συνάρτηση main() η ύπαρξη παραμέτρων δεν είναι δυνατή και το πρωτότυπο κατά κανόνα είναι sl_def(t_main, void). Φυσικά, ο ορισμός της συνάρτησης main() όπως στη C, είναι αποδεκτός και από την SL. ο οποίος αποτελεί τον μοναδικό τρόπο για το πέρασμα παραμέτρων και την επιστροφή συνάρτησης μέσω return.

```
1 sl_def(t_main, void)
2 {
3
4 }
5 sl_enddef
```

Πηγαίος Κώδικας 3.5: Ορισμός της κύριας συνάρτησης στην SL.

3.3 Τα Ονόματα στην SL

Ένα αναγνωριστικό (identifier) στην SL έχει την ίδια έννοια με ένα αναγνωριστικό της C. Μπορεί να είναι συνάρτηση ή μεταβλητή, όνομα ή πεδίο, μίας δομής (struct) ή μίας ένωσης (union) ή ενός απαριθμητή (enum). Το ίδιο ισχύει και για το όνομα μίας ετικέτας (label) αλλά και για οτιδήποτε άλλο είναι επιτρεπτό από τη C. Είναι επίσης ορατό με τον ίδιο τρόπο, και η έννοια της εμβέλειας είναι ίδια με αυτή της C.

Οι λεκτικές μονάδες της SL είναι ίδιες με αυτές της γλώσσας C, όμως τηρεί κάποιους περιορισμούς όσον αφορά τα έγκυρα αναγνωριστικά και έχει επίσης περισσότερες δεσμευμένες λέξεις συγκριτικά με τη C. Επομένως, τα αναγνωριστικά που αρχίζουν με _sl_, _sl, _FILE_, _file_, _LINE_, _line_ θεωρούνται δεσμευμένα από τη γλώσσα SL και δεν πρέπει να χρησιμοποιούνται.

Ο Πίνακας 3.2 περιλαμβάνει τις λέξεις που θεωρούνται ως δεσμευμένες και δεν πρέπει να χρησιμοποιούνται με άλλη έννοια πέρα από αυτήν που ορίζει η γλώσσα SL.

sl_def	sl_enddef	sl_index	sl_break
sl_create	sl_sync	sl_detach	sl_kill
sl_spawndecl	sl_spawn	sl_spawnsync	
sl_shparm	sl_glparm	sl_shfparm	sl_glfparm
sl_sharg	sl_glarg	sl_shfarg	sl_glfarg
sl_seta	sl_geta	sl_setp	sl_getp
sl_decl	sl_decl_fptr	sl_typedef_fptr	
sl_end_thread	sl_lbr	sl_rbr	sl_glparm_mutable

Πίνακας 3.2: Οι δεσμευμένες λέξεις της SL.

Η δήλωση μίας παραμέτρου στις κλήσεις που ορίζονται αποκλειστικά στην SL και όχι στη C, απαιτεί μία ιδιαίτερη μορφή στην οποία ο τύπος της παραμέτρου είναι απόλυτα διαχωρισμένος από το όνομα της παραμέτρου. Έτσι, η δήλωση ενός πίνακα στην SL απαιτεί μία ιδιόρρυθμη διαδικασία. Θα πρέπει λοιπόν ο τύπος της παραμέτρου να γίνει typedef για κάθε διάσταση του πίνακα, και ως τύπος της, να δοθεί ο τύπος που ορίστηκε μέσω των typedef. Για παράδειγμα, αν επιθυμούμε να έχουμε ως παράμετρο τον πίνακα array, τριών διαστάσεων, με τύπο int, θα πρέπει να ορίσουμε τον τύπο t3_0.

```
int array[10][20][30];

typedef int t1_0[30];
typedef t1_0 t2_0[20];
typedef t2_0 *t3_0;
```

3.4 Χώρος Ονομάτων

Ο χώρος ονομάτων (name space) στην SL είναι παρόμοιος με αυτόν της C. Η διαφορά υπάρχει στις κλήσεις που ορίζονται αποκλειστικά από την SL. Τα αναγνωριστικά των παραμέτρων στις συναρτήσεις των νημάτων και αυτά των ορισμάτων, έχουν έναν ξεχωριστό χώρο ονομάτων από τα υπόλοιπα αναγνωριστικά που χρησιμοποιούνται και από τη C. Για παράδειγμα, στον Πηγαίο Κώδικα 3.6 η συνάρτηση foo έχει παράμετρο με όνομα x και έχει και τοπική μεταβλητή με το ίδιο όνομα. Υπάρχει επίσης δυνατότητα, το ίδιο όνομα να χρησιμοποιηθεί και ως όρισμα κατά τη δημιουργία μίας νέας οικογένειας νημάτων, χωρίς αυτό να δημιουργεί

κάποια σύγχυση, όπως συμβαίνει με τη συνάρτηση soo η οποία έχει και αυτή όρισμα με το όνομα x.

```
1  sl_def(foo, void, sl_glparm(int, x))
2  {
3      int x;
4
5      x = sl_getp(x);
6
7      sl_create(, , 0, 10, 1, , , soo, sl_glarg(int, x, 10));
8      sl_sync();
9  }
10 sl_enddef
```

Πηγαίος Κώδικας 3.6: Παράδειγμα αξιοποίησης του χώρου ονομάτων της SL.

3.5 Τύποι Ονομάτων

Οι τύποι στην SL είναι ακριβώς όπως στη C, αλλά η SL ορίζει μία επιπλέον κατηγορία τύπων, αυτή των συναρτήσεων νημάτων (thread function types), οι οποίοι είναι διαφορετικοί από τους τύπους των κοινών συναρτήσεων της C. Ένας τέτοιος τύπος χαρακτηρίζεται από το πλήθος, τον τύπο και την κατηγορία (κοινόχρηστη ή καθολική) των παραμέτρων της συνάρτησης. Οι τύποι της νέας αυτής κατηγορίας δεν είναι συμβατοί με τους τύπους μίας κοινής συνάρτησης. Δεν είναι επίσης συμβατοί με τον τύπο μίας συνάρτησης νημάτων με διαφορετικά χαρακτηριστικά, για παράδειγμα με διαφορετικό πλήθος παραμέτρων.

Η δήλωση μίας συνάρτησης νημάτων πραγματοποιείται με την κλήση sl_decl() και είναι παρόμοια με αυτήν του ορισμού της. Το πρώτο της όρισμα είναι το identifier το οποίο αντιστοιχεί στο όνομά της. Το δεύτερο είναι το attributes το οποίο αντιστοιχεί στα χαρακτηριστικά της και μπορεί να έχει την τιμή sl_static. Το τρίτο και τελευταίο είναι το thread-param-list, δηλαδή η λίστα παραμέτρων της, η οποία μπορεί να είναι κενή για να δηλώσει ότι η συνάρτηση δεν έχει παραμέτρους.

sl_decl(identifier, attributes [, thread-param-list]);

Η παραπάνω δήλωση μπορεί να γίνει μόνο σε καθολική εμβέλεια (global scope). Υπάρχει επίσης η δυνατότητα δήλωσης ενός δείκτη σε συνάρτηση νημάτων μέσω της κλήσης sl_decl_fptr() η οποία έχει τα ίδια ορίσματα με την sl_decl(). Σε αντίθεση με τη δήλωση συνάρτησης, η δήλωση ενός δείκτη μπορεί να γίνει οπουδήποτε.

sl_decl_fptr(identifier, attributes [, thread-param-list]);

Μπορούμε επίσης να ορίσουμε έναν τύπο που είναι δείκτης σε συνάρτηση νημάτων, έτσι ώστε να τον χρησιμοποιούμε δηλώνοντας δείκτες σε συναρτήσεις νημάτων με τον συνηθισμένο τρόπο δήλωσης αντικειμένων της C. Αυτό γίνεται μέσω της κλήσης sl_typedef_fptr(),

με την οποία ορίζεται ένας νέος τύπος που είναι δείκτης σε μία συγκεκριμένη συνάρτηση νημάτων, δηλαδή με συγκεκριμένο πλήθος, τύπο και κατηγορία παραμέτρων. Η χρήση του είναι επιτρεπτή εντός του σώματος συναρτήσεων.

sl_typedef_fptr(identifier, attributes [, thread-param-list]);

3.6 Υπολογισμός του Παραγοντικού στην SL

Στο παράδειγμα που ακολουθεί δημιουργούμε μία οικογένεια νημάτων για να υπολογίσουμε το παραγοντικό ενός αριθμού. Η οικογένεια νημάτων θα δημιουργηθεί μόνο όταν ισχύει n>0, αναθέτοντας σε κάθε νήμα την εκτέλεση ενός υπολογισμού με βάση το αποτέλεσμα που έχει λάβει από το προηγούμενο νήμα και προωθώντας το στο επόμενο. Για να το πετύχουμε αυτό χρειαζόμαστε μία κοινόχρηστη μεταβλητή με την οποία θα γίνεται η επικοινωνία αλλά και ο συγχρονισμός μεταξύ των νημάτων. Κάθε νήμα θα πρέπει να παίρνει την τιμή που έγραψε το προηγούμενο νήμα, να την πολλαπλασιάζει με το αναγνωριστικό του, και να γράφει το αποτέλεσμα πίσω στην κοινόχρηστη μεταβλητή ώστε να τη διαβάσει το επόμενο νήμα το οποίο και θα ακολουθήσει την ίδια διαδικασία.

```
1  #include <stdio.h>
2
3  sl_def(factorial, void, sl_shparm(int, fact))
4  {
5      sl_index(n);
6      sl_setp(fact, sl_getp(fact) * n);
7  }
8  sl_enddef
9
10 sl_def(t_main, void)
11 {
12     int n = 10;
13
14     if (n == 0)
15     {
16         printf("factorial(0) = 1\n");
17     }
18     else
19     {
20         sl_create(, , 1, n + 1, 1, , , factorial, sl_sharg(int, fact, 1));
21         sl_sync();
22
23         printf("factorial(%d) = %d\n", n, sl_geta(fact));
24     }
25 }
26 sl_enddef
```

Πηγαίος Κώδικας 3.7: Υπολογισμός του παραγοντικού στην SL.

Τα βήματα αυτά είναι ο κώδικας της συνάρτησης νημάτων factorial την οποία θα εκτελέσουν τα νήματα. Για να υπολογιστεί σωστά το αποτέλεσμα θα πρέπει να δημιουργήσουμε τον ακριβή αριθμό νημάτων και κάθε νήμα να έχει το κατάλληλο αναγνωριστικό το οποίο

εξαρτάται από το βήμα που ορίζεται στην κλήση της sl_create(). Στην περίπτωσή μας χρειάζονται n νήματα με αναγνωριστικά από 1 εώς και n (για το λόγο αυτό το τέταρτο όρισμα είναι n+1) με βήμα 1 τα οποία θα εκτελέσουν τον κώδικα της συνάρτησης νημάτων factorial. Η κοινόχρηστη μεταβλητή που χρειάζεται είναι η fact την οποία αρχικοποιούμε σε 1.

Το τελικό αποτέλεσμα θα υπολογιστεί μετά την ολοκλήρωση και του τελευταίου νήματος, το οποίο θα γράψει το αποτέλεσμα στην κοινόχρηστη μεταβλητή. Τέλος, με την κλήση sl_geta() παίρνουμε την τιμή της κοινόχρηστης μεταβλητής fact την οποία και εμφανίζουμε με την printf().

Κεφαλαιο 4

Το Προτυπο Νηματων POSIX

4.1 Εισαγωγή

Τα POSIX Threads ή αλλιώς Pthreads. είναι ένα πρότυπο νημάτων που ορίζει μία διεπαφή προγραμματισμού εφαρμογών (API - Application Programming Interface) για τη δημιουργία και τη διαχείριση νημάτων. Η διεπαφή αυτή ορίστηκε με το πρότυπο IEEE POSIX 1003.1c για τα συστήματα UNIX. έχοντας ως στόχο τη φορητότητα των πολυνηματικών εφαρμογών.

Το πρότυπο νημάτων POSIX αποτελείται από τύπους δεδομένων, σταθερές και συναρτήσεις υλοποιημένες στη γλώσσας C, και παρέχεται μέσω του αρχείου κεφαλίδων (header file) pthread.h. Οι συναρτήσεις του ομαδοποιούνται σε τρεις βασικές κατηγορίες. Η πρώτη περιλαμβάνει συναρτήσεις για τη δημιουργία και τη μετέπειτα διαχείριση των νημάτων, η δεύτερη έχει συναρτήσεις σχετικά με τον αμοιβαίο αποκλεισμό (mutual exclusion) και η τρίτη για τις μεταβλητές συνθήκης (condition variable).

Τα νήματα του προτύπου POSIX είναι κατάλληλα για τον προγραμματισμό παράλληλων εφαρμογών. Υπάρχουν διαφορετικές υλοποιήσεις του προτύπου POSIX, νήματα σε επίπεδο πυρήνα (kernel level) ή νήματα σε επίπεδο χρήστη (user level). Η διαφορά είναι στο αν το λειτουργικό σύστημα γνωρίζει την ύπαρξή τους. Οι λειτουργίες που προσφέρονται είναι

κλήσεις συστήματος, με αποτέλεσμα να είναι χρονοβόρες. Οι υλοποιήσεις του προτύπου POSIX χρησιμοποιούν κατά κανόνα νήματα επιπέδου πυρήνα.

Οι συναρτήσεις που προσφέρει το πρότυπο είναι περίπου εκατό, όμως στη συνέχεια του κεφαλαίου θα αναφερθούμε μόνο σε μερικές από αυτές, ενώ στα πλαίσια της παρούσας διατριβής θα χρησιμοποιήσουμε τις δύο που παρουσιάζονται στον Πίνακα 4.1.

pthread_t	Τύπος αναγνωριστικού νήματος.
pthread_create()	Δημιουργία νήματος.
pthread_join()	Συγχρονισμός νήματος.

Πίνακας 4.1: Αντικείμενα του προτύπου POSIX που θα μας απασχολήσουν στα πλαίσια της παρούσας διατριβής.

4.2 Δημιουργία Νήματος

Η δημιουργία ενός νήματος γίνεται με την κλήση pthread_create(). Το πρώτο όρισμα είναι το thread το οποίο είναι δείκτης τύπου pthread_t, και εκεί αποθηκεύεται το αναγνωριστικό του νήματος που δημιουργείται. Το δεύτερο όρισμα είναι το attr και αφορά τις ιδιότητες του νήματος (όπως η πολιτική δρομολόγησης, το μέγεθος της στοίβας κ.τ.λ.). Αν δεν ορίστεί ρητά, δηλαδή αν δοθεί ως όρισμα η τιμή NULL, γίνεται χρήση των προεπιλεγμένων τιμών. Η συνάρτηση που εκτελεί το νήμα που δημιουργείται, ορίζεται μέσω του τρίτου ορίσματος, start_routine, το οποίο είναι ένας δείκτης σε συνάρτηση. Το τελευταίο όρισμα της pthread_create() αναφέρεται στο όρισμα το οποίο λαμβάνει η συνάρτηση start_routine(). Είναι ένας δείκτης τύπου void, άρα αν θέλουμε να περάσουμε παραπάνω από ένα ορίσματα, θα πρέπει να ορίσουμε μία δομή (struct) με τα αντίστοιχα πεδία. Σε περίπτωση επιτυχίας, η κλήση επιστρέφει 0, διαφορετικά επιστρέφει έναν κωδικό σφάλματος. Συγκεκριμένα, επιστρέφει EAGAIN, αν υπάρχει έλλειψη διαθέσιμων πόρων ή αν έχει επιβληθεί όριο στον επιτρεπτό αριθμό των νημάτων που μπορούν να δημιουργηθούν. Όταν η τιμή που ορίζεται από το attr είναι λανθασμένη, επιστρέφει EINVAL. Τέλος, επιστρέφει EPERM όταν δεν έχει τα απαραίτητα δικαιώματα για τον ορισμό της συγκεκριμένης πολιτικής δρομολόγησης (scheduling policy) ή των παραμέτρων δρομολόγησης.

```
int pthread_create(pthread_t  *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

4.3 Αναμονή Νήματος

Η *αναμονή ολοκλήρωσης* ενός νήματος επιτυγχάνεται μέσω της *κλήσης* pthread_join(). Το πρώτο όρισμα είναι το αναγνωριστικό του νήματος, δηλαδή μία μεταβλητή τύπου pthread_t.

Μέσω του δεύτερου ορίσματος είναι δυνατόν να ληφθεί η τιμή που επέστρεψε το νήμα (μπορεί να είναι και NULL για να υποδείξουμε ότι δεν τη χρειαζόμαστε). Η κλήση pthread_join() αναστέλλει την εκτέλεση του νήματος που την καλεί μέχρι να ολοκληρωθεί το νήμα που περιμένει. Αν η κλήση είναι επιτυχής, επιστρέφει 0, αλλιώς επιστρέφει έναν κωδικό σφάλματος. Ο κωδικός EINVAL υποδεικνύει ότι το αναγνωριστικό του νήματος που δόθηκε ως όρισμα δεν αναφέρεται σε νήμα το οποίο μπορεί να αποδεσμευτεί, ενώ ο κωδικός ESRCH αντιστοιχεί σε μη έγκυρο αναγνωριστικό νήματος.

int pthread_join(pthread_t thread, void **value_ptr);

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 10
5
6  void *start_routine(void *arg)
7  {
8      int tid = *((int *) arg);
9
10     printf("Thread(%d): Hello World!\n", tid);
11
12     return (NULL);
13 }
14
15 int main()
16 {
17     int i, args[NUM_THREADS];
18     pthread_t tid[NUM_THREADS];
19
20     for (i = 0; i < NUM_THREADS; i++)
21     {
22         args[i] = i;
23         pthread_create(&tid[i], NULL, start_routine, (void *) &args[i]);
24     }
25
26     for (i = 0; i < NUM_THREADS; i++)
27     {
28         pthread_join(tid[i], NULL);
29     }
30
31     return (0);
32 }
```

Πηγαίος Κώδικας 4.1: Παράδειγμα δημιουργίας νημάτων με το πρότυπο νημάτων POSIX.

4.4 Αποδέσμευση Νήματος

Αν δε θέλουμε να περιμένουμε την ολοκλήρωση ενός νήματος, δηλαδή αν δεν πρόκειται να καλέσουμε ποτέ την pthread_join() για το συγκεκριμένο νήμα, μπορούμε να το δηλώσουμε με την pthread_detach(). Έτσι, το λειτουργικό σύστημα ενημερώνεται ότι μπορεί να επαναχρησιμοποιήσει τους πόρους του νήματος μόλις αυτό τερματίσει. Το μοναδικό όρισμά της.

είναι το αναγνωριστικό του νήματος προς αποδέσμευση. Σε περίπτωση επιτυχίας επιστρέφεται η τιμή 0, διαφορετικά ένας κωδικός σφάλματος (είναι ίδιοι με αυτούς για την κλήση pthread_join()).

int pthread_detach(pthread_t thread);

4.5 Έξοδος Νήματος

Ο τερματισμός ενός νήματος γίνεται όταν ολοκληρώσει την εκτέλεσή του ή καλώντας την κλήση pthread_exit(). Το μοναδικό όρισμά της είναι ένας δείκτης στη μεταβλητή που επιστρέφει το νήμα. Δεν επιστρέφει κάποια τιμή, λόγω του ότι είναι τύπου void.

void pthread_exit(void *value_ptr);

4.6 Αναγνωριστικό Νήματος

Κάθε νήμα μπορεί να μάθει το αναγνωριστικό του με την κλήση **pthread_self()**. Καλείται από το ίδιο το νήμα και δεν παίρνει κανένα όρισμα.

pthread_t pthread_self(void);

Κεφαλαιο 5

Σχετικες Εργασιες στη Βιβλιογραφια

5.1 Παραλληλοποίηση Αναδρομικών Συναρτήσεων

Ένα σύνηθες φαινόμενο για τον προγραμματιστή είναι να γράφει κώδικα που είναι ευανάγνωστος και εύκολος στην κατανόηση, χωρίς να επικεντρώνεται στην αποτελεσματικότητά του. Ένα καλό παράδειγμα τέτοιου κώδικα είναι οι αναδρομικές συναρτήσεις. Η αναδρομή είναι εύκολη στη χρήσης της αλλά μειώνει σημαντικά την απόδοση των προγραμμάτων. Υπάρχουν συναρτήσεις που δεν μπορούν να υλοποιηθούν χωρίς αναδρομή, ενώ υπάρχουν και αυτές που επιδέχονται απαλοιφή. Όμως, οι περισσότερες αναδρομικές συναρτήσεις προσφέρουν τη δυνατότητα εξαγωγής παραλληλισμού τον οποίο καλούμαστε να εκμεταλλευτούμε. Οι μεταφραστές συχνά εφαρμόζουν τεχνικές βελτιστοποίησης με στόχο τη βελτίωση της απόδοσης. Από την άλλη έχουν γίνει προσπάθειες για αυτόματη παραλληλοποίηση του κώδικα, αλλά δεν έχει υλοποιηθεί κάποιος μεταφραστής που να εξειδικεύεται στην παραλληλοποίηση αναδρομικών συναρτήσεων διαφορετικών μορφών.

Μία από τις πιο κοινές μορφές αναδρομικών συναρτήσεων είναι αυτή της μεθόδου διαίρει και βασίλευε. Χαρακτηρίζεται από δύο ή περισσότερες ανεξάρτητες αναδρομικές κλήσεις που αναλαμβάνουν την επίλυση των υποπροβλημάτων που προκύπτουν από το αρχικό. Στην περίπτωση αυτή υπάρχει έμφυτος παραλληλισμός που μπορεί να εξαχθεί από την παράλληλη εκτέλεση όλων των αναδρομικών κλήσεων. Στο [7] περιγράφεται ένας μεταφραστής που πραγματοποιεί αυτόματη παραλληλοποίηση αναδρομικών συναρτήσεων αυτής της μορφής, για προγράμματα σε C. Για να αναγνωρίσει πότε οι αναδρομικές κλήσεις είναι ανεξάρτητες και μπορούν να εκτελεστούν ταυτόχρονα με ασφάλεια, εφαρμόζει μία σειρά από αναλύσεις όπως η διαδικαστική ανάλυση συμβόλων σε πίνακες και η εκτίμηση των τιμών που λαμβάνουν όλες

οι εμπλεκόμενες αριθμητικές εκφράσεις κατά τον χρόνο εκτέλεσης. Αν δεν καταφέρει να αναγνωρίσει ανεξάρτητες αναδρομικές κλήσεις κατά την ώρα της μετάφρασης, χρησιμοποιεί έναν συνδυασμό τεχνικών μετάφρασης και χρόνου εκτέλεσης για την εξαγωγή της παραλληλίας.

Ένας παρόμοιος μεταφραστής για την παραλληλοποίηση αναδρομικών συναρτήσεων που εντάσσονται στην κατηγορία των αλγορίθμων διαίρει και βασίλευε έχει υλοποιηθεί από τους Rugina και Rinard, και περιγράφεται στο [8]. Στόχος τους είναι η υποστήριξη των αναδρομικών συναρτήσεων στις οποίες εμπλέκονται δείκτες, κάτι που επιτυγχάνουν με την εφαρμογή διαφορετικών τεχνικών ανάλυσης του κώδικα όπως είναι η διαδικαστική ανάλυση των δεικτών και ο έλεγχος εξαρτήσεων που αφορά τους δείκτες. Ο μεταφραστής έχει υλοποιηθεί χρησιμοποιώντας τον SUIF και παράγει παράλληλο κώδικα για τη γλώσσα Cilk.

Το εργαλείο Huckleberry που παρουσιάζεται στο [9] είναι ένας ακόμα μεταφραστής για την παραλληλοποίηση αναδρομικών συναρτήσεων των αλγορίθμων διαίρει και βασίλευε. Η γλώσσα εισόδου είναι η C, ενώ ο κώδικας που παράγει προορίζεται για πολυπύρηνες αρχιτεκτονικές κατανεμημένης μνήμης.

Στο [10] γίνεται η περιγραφή μιας διαφορετικής προσέγγισης που αφορά την παραλληλοποίηση αναδρομικών συναρτήσεων η οποία βασίζεται στην τεχνική της εξάλειψης. Αρχικά διασπά τους υπολογισμούς της συνάρτησης και δημιουργεί αναζάρτητα τμήματα κώδικα τα οποία μπουρούν να εκτελεστούν παράλληλα, και στη συνέχεια υπολογίζει το τελικό αποτέλεσμα από τον συνδυασμό των επιμέρους αποτελεσμάτων. Η όλη διαδικασία μπορεί να επιτευχθεί κατά αυτόματο τρόπο, χωρίς την ανάγκη υποστήριξης από την πλευρά του προγραμματιστή.

Μία ακόμα τεχνική αυτόματης παραλληλοποίησης κώδικα έχει υλοποιηθεί από τον μεταφραστή που παρουσιάζεται στο [11]. Ο μεταφραστής αυτός πραγματοποιεί απαλοιφή αναδρομικών συναρτήσεων σε Lisp, μετασχηματίζοντας τον κώδικα σε επαναληπτικό. Έπειτα εφαρμόζει μετασχηματισμούς παραλληλοποίησης για βρόχους και παράγει κώδικα που προορίζεται για πολυεπεξεργαστές κοινής μνήμης.

Τέλος, στα [12, 13] παρουσιάζεται μία αρχική προσπάθεια που επιχειρήσαμε για την απεικόνιση αναδρομικών συναρτήσεων στο μοντέλο SVP. Ο μεταφραστής αυτός παρέχει τη δυνατότητα αυτόματης παραλληλοποίησης των αναδρομικών συναρτήσεων που χαρακτηρίζονται από την ύπαρξη ενός δείκτη. Ο μετασχηματισμός είναι ίδιος με αυτόν που εφαρμόζει το πέρασμα *elimination* του μεταφραστή *Ariadne*. Η διαφορά έγκειται στο γεγονός ότι ο μεταφραστής αυτός δεν απαιτεί καμία οδηγία από τον προγραμματιστή, σε αντίθεση με τον μεταφραστή *Ariadne*, η φιλοσοφία του οποίου στηρίζεται στην παραλληλοποίηση με οδηγίες. Ως εκ τούτου, η απεικόνιση του κώδικα γίνεται με αποκλειστική ευθύνη του μεταφραστή.

Στη συνέχεια της ενότητας περιγράφουμε το μοντέλο OpenMP και τη γλώσσα Cilk, που παρέχουν στον προγραμματιστή έναν εύκολο και αποτελεσματικό τρόπο για την εξαγωγή παραλληλισμού από συναρτήσεις που έχουν τουλάχιστον δύο ανεξάρτητες αναδρομικές κλήσεις.

5.1.1 OpenMP

Το μοντέλο OpenMP [14] ορίζεται ως μία διεπαφή για τον παράλληλο προγραμματισμό εφαρμογών κοινής μνήμης. Αποτελείται από ένα σύνολο οδηγιών, μεταβλητές περιβάλλοντος και συναρτήσεις βιβλιοθήκης. Το OpenMP υποστηρίζεται από τα περισσότερα λειτουργικά συστήματα και τις περισσότερες αρχιτεκτονικές επεξεργαστών και είναι διαθέσιμο για τις γλώσσες C, C++ και Fortran. Η παραλληλοποίηση ενός προγράμματος επιτυγχάνεται με απλό τρόπο μέσω των οδηγιών που καθορίζουν τη μορφή του παραλληλισμού που επιθυμεί ο προγραμματιστής. Ο μεταφραστής δεν πραγματοποιεί κανένα είδος ελέγχου, όπως είναι οι εξαρτήσεις μεταξύ των δεδομένων, οι κρίσιμες περιοχές (critical section) και τα αδιέξοδα (deadlock). Ο κώδικας που παράγει αποτελεί αποκλειστική ευθύνη του προγραμματιστή, καθώς βασίζεται στις οδηγίες που εκείνος εισάγει. Δεν είναι δυνατή η αυτόματη παραλληλοποίηση του κώδικα χωρίς τη βοήθεια του προγραμματιστή.

Οι οδηγίες του OpenMP είναι ευέλικτες και έχουν ως στόχο να είναι απλές. Κάθε οδηγία έχει μία αυστηρά καθορισμένη σημασία η οποία επιτρέπει στον προγραμματιστή να περιγράψει τις εξαρτήσεις που χαρακτηρίζουν ένα τμήμα του κώδικα και τον παραλληλισμό που μπορεί να εξαχθεί από αυτό.

Μία από τις πιο διαδεδομένες οδηγίες του είναι η reduction, η οποία στην περίπτωση της C και της C++ χρησιμοποιείται στους βρόχους for. Με την οδηγία αυτή ο προγραμματιστής χαρακτηρίζει μία μεταβλητή ως reduction, έτσι ώστε ο μεταφραστής να είναι ενήμερος για το είδος της εξάρτησης που δημιουργεί και να τη διαχειριστεί κατάλληλα. Ο μεταφραστής *Ariadne* που υλοποιείται στα πλαίσια της παρούσας διατριβής, αναγνωρίζει μία παρόμοια οδηγία, την parallel-reduction. Η διαφορά μεταξύ των δύο οδηγιών έγκειται στο γεγονός ότι η οδηγία του μοντέλου OpenMP σχετίζεται με την κατανομή του φόρτου εργασίας ενός βρόχου σε μία σειρά από νήματα, σε αντίθεση με την οδηγία του μεταφραστή *Ariadne*, με την οποία γίνεται η κατανομή του φόρτου εργασίας μίας ολόκληρης αναδρομικής συνάρτησης. Παρ' όλα αυτά, οι οδηγίες αυτές βασίζονται στην ίδια ιδέα και η ύπαρξή τους οφείλεται στον ίδιο λόγο, δηλαδή στον χαρακτηρισμό της εξάρτησης που δημιουργείται.

Το OpenMP υποστηρίζει επίσης την παραλληλοποίηση αναδρομικών συναρτήσεων που έχουν τουλάχιστον δύο ανεξάρτητες αναδρομικές κλήσεις με τη χρήση της οδηγίας task. Σε αντίθεση με τον μεταφραστή *Ariadne* που επιτυγχάνει την παραπάνω μορφή παραλληλισμού με ακριβώς μία οδηγία πριν τον ορισμό της συνάρτησης, το μοντέλο OpenMP απαιτεί τουλάχιστον δύο οδηγίες (μία για κάθε κλήση), τις οποίες θα πρέπει να εισάγει ο προγραμματιστής σε διαφορετικά σημεία της συνάρτησης αναλόγως την περίπτωση.

5.1.2 Cilk

Η γλώσσα Cilk [15, 16] είναι επέκταση της C και περιέχει τρεις επιπλέον λέξεις που υποδεικνύουν συγχρονισμό. Είναι σχεδιασμένη για τον παράλληλο προγραμματισμό γενικού σκοπού, αλλά είναι ιδιαίτερα αποτελεσματική στην εκμετάλλευση δυναμικού και ασύγχρονου παραλληλισμού. Προορίζεται για συστήματα όπως το UNIX τα οποία υποστηρίζουν το πρότυπο νημάτων POSIX, όμως μπορεί να εκτελεστεί και σε άλλα συστήματα αρκεί να είναι

διαθέσιμα ο gcc, το πρότυπο νημάτων POSIX και το GNU make.

Σε αντίθεση με άλλα πολυνηματικά προγραμματιστικά συστήματα, η Cilk είναι αλγοριθμική δεδομένου ότι το σύστημα χρόνου εκτέλεσης χρησιμοποιεί ένα χρονοδιάγραμμα το οποίο επιτρέπει να εκτιμηθεί με ακρίβεια η επίδοση των προγραμμάτων, βασιζόμενο σε αφηρημένες μετρικές πολυπλοκότητας. Επομένως, η φιλοσοφία της βασίζεται στο να επικεντρωθεί ο προγραμματιστής στη δομή του κώδικα με στόχο την παραλληλοποίησή του, αφήνοντας στο σύστημα χρόνου εκτέλεσης (runtime system) της γλώσσας, την αποτελεσματική χρονοδρομολόγηση (scheduling) των νημάτων. Το σύστημα χρόνου εκτέλεσης φροντίζει λεπτομέρειες όπως η εξισορρόπηση φορτίου, ο συγχρονισμός και τα πρωτόκολλα επικοινωνίας.

Όταν ένα πρόγραμμα σε Cilk τρέχει σε έναν επεξεργαστή, έχει την ίδια σημασιολογία με το αντίστοιχο πρόγραμμα σε C αν αφαιρέσουμε τις λέξεις της Cilk. Το πρόγραμμα αυτό ονομάζεται serial elision ή C elision του προγράμματος σε Cilk. Η λέξη cilk προσδιορίζει μία Cilk διαδικασία η οποία είναι η παράλληλη έκδοση μίας διαδικασίας σε C. Ο περισσότερος κώδικας μίας Cilk διαδικασίας εκτελείται σειριακά όπως και στη C, ενώ ο παραλληλισμός δημιουργείται όταν δηλώνεται ρητά με τη λέξη spawn.

Σε αντίθεση με μία συνάρτηση σε C όπου ο πατέρας δε συνεχίζει την εκτέλεση μέχρι τον τερματισμό του παιδιού, στη Cilk συνεχίζει να εκτελείται παράλληλα με το παιδί. Έτσι ο πατέρας μπορεί να καλέσει στη συνέχεια και άλλες συναρτήσεις, δημιουργώντας υψηλό βαθμό παραλληλίας. Ο χρονοδρομολογητής (scheduler) της Cilk έχει την ευθύνη για τη χρονοδρομολόγηση των διαδικασιών που δημιουργήθηκαν με spawn, στους επεξεργαστές του παράλληλου υπολογιστή.

Η Cilk μπορεί να υποστηρίξει την παραλληλοποίηση αναδρομικών συναρτήσεων με τη λέξη spawn. Για να γίνει όμως αυτό, θα πρέπει να υπάρχουν ανεξάρτητες αναδρομικές κλήσεις και κάθε μία από αυτές να γίνει spawn. Συνεπώς, η γλώσσα Cilk υποστηρίζει μία συγκεκριμένη κατηγορία αναδρομικών συναρτήσεων και απαιτεί τον ρητό χαρακτηρισμό κάθε τέτοιας κλήσης, ως ανεξάρτητη. Η κατηγορία αυτή αντιστοιχεί στις αναδρομικές συναρτήσεις που υποστηρίζονται από τον μεταφραστή *Ariadne* μέσω του περάσματος *thread-safe*.

Η εκτέλεση ενός προγράμματος σε Cilk μπορεί να απεικονιστεί σε ένα κατευθυνόμενο άκυκλο γράφημα (DAG - Directed Acyclic Graph). Κάθε διαδικασία αναπαρίσταται από ένα ορθογώνιο και διαιρείται σε ακολουθίες νημάτων που αναπαρίστανται ως κύκλοι. Μία ακμή προς τα κάτω υποδεικνύει μία υποδιαδικασία που δημιουργήθηκε με spawn. Η οριζόντια ακμή υποδεικνύει τη συνέχεια του κώδικα και η προς τα πάνω την επιστροφή τιμής στον πατέρα. Όλες οι ακμές παριστάνουν εξαρτήσεις οι οποίες θα πρέπει να τηρηθούν και υποδεικνύουν τη σειρά με την οποία θα εκτελεστούν τα νήματα.

Μέχρι σήμερα, έχουν αναπτυχθεί σε Cilk πολλές πρωτότυπες εφαρμογές, όμως η μεγαλύτερη προσπάθεια ανάπτυξης εφαρμογών είναι μία σειρά από προγράμματα σκάκι τα οποία είναι παγκόσμιας εμβέλειας (Socrates, StarTech και Cilkchess).

5.2 Περισσότεροι Μεταφραστές Παραλληλοποίησης

Στην ενότητα αυτή γίνεται μία σύντομη περιγραφή μερικών από τους σημαντικότερους μεταφραστές παραλληλοποίησης που υπάρχουν στη βιβλιογραφία.

5.2.1 Polaris

Ο μεταφραστής Polaris [17, 18] είναι ένα εξαιρετικό εργαλείο για την πραγματοποίηση μετασχηματισμών και την ανάλυση πολύπλοκου κώδικα, αποσκοπώντας στην αποτελεσματική εκτέλεσή του σε έναν παράλληλο υπολογιστή. Είναι υλοποιημένος σε C++ και αποτελείται από 170.000 γραμμές κώδικα. Τα αντικειμενοστρεφή στοιχεία της γλώσσας υποστηρίζουν με ιδανικό τρόπο τη φιλοσοφία του συστήματος, το οποίο είναι οργανωμένο σε μία ιεραρχία κλάσεων. Παρ' όλα αυτά, η επιλογή της γλώσσας προγραμματισμού, αρχικά έγινε λόγω της δημοτικότητας και της ευελιξίας που χαρακτηρίζουν τη C++.

Η γλώσσα εισόδου είναι μία επέκταση της Fortran 77, ενώ παράγει κώδικα για μία από τις παράλληλες διαλέκτους της Fortran. Το πρόγραμμα εξόδου του προορίζεται για παράλληλους υπολογιστές υψηλών επιδόσεων με καθολικό χώρο διευθύνσεων (global address space), όπως αυτοί που διαθέτουν πολυπύρηνους επεξεργαστές κοινής μνήμης. Το πρόγραμμα εισόδου μπορεί να περιλαμβάνει οδηγίες από τον προγραμματιστή, μέσω των οποίων καθορίζεται ρητά το είδος των μετασχηματισμών που θα εφαρμοστούν. Ο μεταφραστής Polaris πέρα από τα συνηθισμένα περάσματα με τα οποία πραγματοποιεί τους μετασχηματισμούς, παρέχει διαδικασίες για την αναγνώριση των ιδιωτικών πινάκων (array privatization), τον έλεγχο εξάρτησης δεδομένων (data dependence testing), την αναγνώριση επαγωγικών μεταβλητών (induction variable recognition), τη διαδικαστική ανάλυση (interprocedural analysis) και την ανάλυση συμβόλων (symbolic program analysis).

Η εσωτερική του αναπαράσταση αποτελείται από ένα βασικό αφηρημένο συντακτικό δέντρο, στην κορυφή του οποίου υπάρχουν πολλά επίπεδα λειτουργικότητας. Αυτή η λειτουργικότητα δίνει τη δυνατότητα για περίπλοκες λειτουργίες πάνω στις δομές δεδομένων που διατηρεί, όπως επίσης και να μιμηθεί άλλες εσωτερικές αναπαραστάσεις. Επιπλέον, η εσωτερική αναπαράσταση έχει σχεδιαστεί για να ενισχύει τη συνοχή της κατάστασης της εσωτερικής δομής, τόσο από την άποψη της ορθότητας των δομών δεδομένων όσο και από την ορθότητα του κώδικα που θα χειριστεί. Οι λειτουργίες που αφορούν την εσωτερική αναπαράσταση, έχουν ως αποτέλεσμα την αυτόματη ενημέρωση των δομών δεδομένων που επηρεάζονται, όπως είναι η ροή πληροφοριών.

5.2.2 SUIF

Ο μεταφραστής SUIF [19] (Stanford University Intermediate Format) έχει σχεδιαστεί για να παραλληλοποιεί επιστημονικά προγράμματα για κλιμακούμενες παράλληλες μηχανές (scalable parallel machines). Ένας από τους κύριους στόχους του SUIF είναι η αυτόματη παραλληλοποίηση κώδικα που περιλαμβάνει σειριακούς υπολογισμούς πυκνών πινάκων. Πολλές από τις τεχνικές μετάφρασης που απαιτούνται για τη δημιουργία ορθού και αποτελεσματικού

κώδικα είναι κοινές σε όλες τις κλιμακούμενες μηχανές ανεξάρτητα από το αν ο χώρος διευθύνσεων (address space) είναι κοινόχρηστος ή κατανεμημένος. Έπομένως, ο κώδικας που παράγεται προορίζεται τόσο για μηχανές κοινόχρηστου χώρου διευθύνσεων (SAS - Shared Address Space) όσο και για κατανεμημένου (DAS - Distributed Address Space).

Οι γλώσσες εισόδου για τον μεταφραστή είναι η C και η Fortran 77. Ο κώδικας αρχικά μεταφράζεται στην ενδιάμεση αναπαράσταση του SUIF, ενώ στη συνέχεια ένα σύνολο από περάσματα ανάλυσης και βελτιστοποίησης που έχουν υλοποιηθεί ως ανεξάρτητα προγράμματα, εφαρμόζονται σε αυτήν. Τα περάσματα αυτά διαιρούνται σε τέσσερις κατηγορίες οι οποίες είναι η ανάλυση συμβόλων (symbolic analysis), η ανάλυση παραλληλισμού και τοπικότητας (parallelism and locality analysis), η ανάλυση επικοινωνίας και συγχρονισμού (communication and synchronization analysis) και η παραγωγή κώδικα (code generation). Στο τελικό στάδιο, η ενδιάμεση αναπαράστη μετατρέπεται σε κώδικα C ο οποίος μεταφράζεται από τον μεταγλωττιστή της C.

Κατά το πέρασμα της ανάλυσης συμβόλων, ο μεταφραστής αρχικά εξάγει όλη την απαραίτητη πληροφορία για τα περάσματα παραλληλισμού και βελτιστοποίησης που θα ακολουθήσουν. Συγκεκριμένα πραγματοποιεί ανάλυση των βαθμωτών μεταβλητών, κάνοντας διάδοση των σταθερών. αναγνώριση των επαγωγικών μεταβλητών, πρόσθια διάδοση (forward propagation), έλεγχο εξάρτησης των δεδομένων και ανάλυση ροής δεδομένων σε πίνακες. Το πέρασμα της ανάλυσης παραλληλισμού και τοπικότητας αναγνωρίζει και βελτιστοποιεί το επίπεδο παραλληλίας των βρόχων του προγράμματος. Στη συνέχεια ο μεταφραστής απεικονίζει τους υπολογισμούς στους επεξεργαστές καθορίζοντας μία διάταξη για τους πίνακες, με στόχο τη μεγιστοποίηση του παραλληλισμού και την ελαχιστοποίηση της επικοινωνίας. Το πέρασμα για την ανάλυση της επικοινωνίας και του συγχρονισμού χρησιμοποιεί τις πληροφορίες για την απεικόνιση, για να αναγνωρίσει τις προσβάσεις σε μη τοπικά δεδομένα. Η πληροφορία αυτή χρησιμοποιείται για να δημιουργήσει τα μηνύματα αποστολής (send) και λήψης (receive) για την πρόσβαση στα δεδομένα κατά την εκτέλεση στις μηχανές DAS, και για να μειώσει το κόστος επικοινωνίας κατά την εκτέλεση στις μηχανές SAS.

Η πραγματοποίηση των μετασχηματισμών που απαιτούν τα παραπάνω περάσματα γίνεται από το πέρασμα παραγωγής κώδικα. Αρχικά κάνει τη χρονοδρομολόγηση (scheduling) των παράλληλων βρόχων ώστε κάθε επεξεργαστής να εκτελεί τις δικές του επαναλήψεις και στη συνέχεια εισάγει τον απαραίτητο κώδικα για τον συγχρονισμό και την επικοινωνία. Τροποποιεί επίσης τις προσβάσεις στους πίνακες σύμφωνα με το προηγούμενο πέρασμα. Τα προηγούμενα περάσματα αποσκοπούν κυρίως στην ανάλυση του κώδικα και όχι στην τροποποίησή του, που θα είχε ως συνέπεια την επιρροή των επακόλουθων περασμάτων. Η πληροφορία μεταξύ των περασμάτων διαδίδεται μέσω σχολίων που προστίθενται στην ενδιάμεση αναπαράσταση του κώδικα.

5.2.3 Cetus

Το Cetus [20, 21, 22] είναι ένας source-to-source μεταφραστής παραλληλοποίησης που στην τρέχουσα έκδοση έχει ως γλώσσα εισόδου την ANSI C. Αν και ο μεταφραστής Polaris θεωρείται ο προκάτοχος του Cetus, στόχος του δεν είναι να γίνει για τη C ότι ο μεταφραστής

Polaris για τη Fortran. Η φιλοσοφία που ακολουθεί είναι αυτή του SUIF, θέτωντας ως στόχο να ενταχθεί στην κατηγορία των επεκτάσιμων μεταφραστών. Παράλληλα όμως δεν αποφεύγει την υιοθέτηση των θετικών χαρακτηριστικών του μεταφραστή Polaris. Ο μεταφραστής Cetus βρίσκεται σε συνεχή εξέλιξη για περίπου μία δεκαετία και αποτελεί τη βάση άλλων ερευνητικών εργασιών.

Είναι υλοποιημένος σε Java και αποτελείται από περισσότερες από 80.000 γραμμές κώδικα. Η επιλογή της Java ως γλώσσα υλοποίησης έγινε λόγω του ότι προσφέρει αρκετά χαρακτηριστικά που ευνοούν την καλή τεχνολογία λογισμικού. Παρέχει καλή υποστήριξη αποσφαλμάτωσης, υψηλή φορητότητα, συλλογή σκουπιδιών (garbage collection) η οποία συμβάλλει στην ευκολία συγγραφής περασμάτων, καθώς και το δικό της αυτόματο σύστημα τεκμηρίωσης. Χρησιμοποιεί το πρόγραμμα συντακτικής ανάλυσης Antlr [23] το οποίο είναι επανεισερχόμενο (reentrant) με την έννοια ότι μπορεί να δεχθεί νέα είσοδο χωρίς να έχει ολοκληρώσει την προηγούμενη. Έτσι, ο μεταφραστής Cetus δημιουργεί νήματα της Java για την ανάλυση και τη δημιουργία της ενδιάμεσης αναπαράστασης όλων των αρχείων παράλληλα.

Η ενδιάμεση αναπαράσταση του Cetus μπορεί να αναπαραστήσει εκφράσεις και εντολές σε γλώσσες όπως η C++ και η Java χωρίς ιδιαίτερες αλλαγές. επεκτείνοντας μόνο ένα τμήμα για την αναπαράσταση των εξαιρέσεων (exception) που δεν υποστηρίζονται από τη C. Η υλοποίησή του έχει τη μορφή ιεραρχίας κλάσεων της Java. Η υψηλού επιπέδου αναπαράσταση παρέχει στα περάσματα μία συντακτική όψη του κώδικα, κάνοντας εύκολη την κατανόηση, την πρόσβαση, αλλά και τους μετασχηματισμούς του προγράμματος εισόδου. Υπάρχει πλήρης αφαίρεση και κάθε πέρασμα διαχειρίζεται την ενδιάμεση αναπαράσταση μέσω συναρτήσεων. Όλα τα αντικείμενα ενδιάμεσης αναπαράστασης του Cetus παράγονται από μία βασική κλάση, η οποία παρέχει τη δυνατότητα διάσχισης όλων των αντικειμένων ενδιάμεσης αναπαράστασης με γενικό τρόπο, χρησιμοποιώντας την τεχνική του πολυμορφισμού.

Το Cetus προσφέρει μία σειρά από επιλογές τόσο για την ανάλυση όσο και για τον μετασχηματισμό του κώδικα. Μερικά από τα περάσματα ανάλυσης που παρέχει είναι για την εξάρτηση των δεδομένων (data dependence analysis), την αναγνώριση του εύρους τιμών που λαμβάνει μία μεταβλητή (range analysis) και των θέσεων μνήμης που δείχνει ένας δείκτης (pointer alias analysis). Κάποιοι από τους μετασχηματισμούς που πραγματοποιεί είναι η κανονικοποίηση των βρόχων, η αναγνώριση ιδιωτικών και επαγωγικών μεταβλητών, καθώς και η αναγνώριση των reduction μεταβλητών.

5.2.4 OSCAR

Ο OSCAR [24, 25] αποσκοπεί στην παραλληλοποίηση προγραμμάτων που είναι γραμμένα σε Fortran και ενδεχομένως να περιέχουν οδηγίες για το μοντέλο OpenMP, ενώ παρέχει μία σειρά από επιλογές για την απεικόνιση του κώδικα. Η πρώτη επιλογή είναι η παραγωγή του κώδικα για κάποια επέκταση της Fortran και συγκεκριμένα για OpenMP, STA MPI ή VPP. Η δεύτερη επιλογή που προσφέρει είναι η παραγωγή κώδικα μηχανής για κάποιο παράλληλο σύστημα όπως ο πολυεπεξεργαστής (multiprocessor) OSCAR.

Ο μεταφραστής OSCAR εξάγει διαφορετικές μορφές παραλληλισμού μέσω μίας ιεραρχίας. Ο παραλληλισμός υψηλού επιπέδου (coarse grain parallelism) εξάγεται μεταξύ των βρόχων,

των υποπρογραμμάτων και των βασικών μπλοκ. Ο παραλληλισμός βρόχων (loop parallelism) εξάγεται μεταξύ των επαναλήψεων ενός βρόχου, ενώ ο παραλληλισμός χαμηλού επιπέδου (fine grain parallelism) μεταξύ των εντολών του κώδικα. Κατά τη διάρκεια της μετάφρασης, το πρόγραμμα εισόδου αποσυντίθεται σε παράλληλα τμήματα κώδικα τα οποία ονομάζονται macrotasks και χαρακτηρίζονται από παραλληλισμό υψηλού επιπέδου. Είναι επίσης επιτρεπτή η δημιουργία μίας ιεραρχίας από macrotasks εντός των βρόχων και των υποπρογραμμάτων.

Στη συνέχεια πραγματοποιείται ανάλυση των εξαρτήσεων ελέγχου και των εξαρτήσεων μεταξύ των δεδομένων, και γίνεται η χρονοδρομολόγηση των macrotasks στους διαθέσιμους επεξεργαστές. Όταν δεν υπάρχουν εξαρτήσεις μεταξύ των macrotasks, η χρονοδρομολόγηση γίνεται στατικά κατά τη διάρκεια της μετάφρασης του κώδικα. Σε διαφορετική περίπτωση ο μεταφραστής OSCAR παράγει κώδικα ο οποίος θα είναι αρμόδιος για τη χρονοδρομολόγηση των macrotasks σε χρόνο εκτέλεσης. Έτσι, η ανάθεση των macrotasks στους επεξεργαστές του συστήματος, πραγματοποιείται δυναμικά ώστε να αντιμετωπιστούν οι αβεβαιότητες. Με τον όρο αβεβαιότητες αναφερόμαστε στα άλματα υπό συνθήκη μεταξύ των macrotasks και στον χρόνο εκτέλεσής τους.

5.2.5 PLUTO

Ο PLUTO [26, 27] είναι ένα εργαλείο που βασίζεται στο πολυεδρικό μοντέλο και εφαρμόζει τεχνικές βελτιστοποίησης με στόχο την καλή τοπικότητα της κρυφής μνήμης (data locality optimization), σε συνδυασμό με μετασχηματισμούς παραλληλοποίησης εμφωλευμένων βρόχων. Το πολυεδρικό μοντέλο είναι μία γεωμετρική αναπαράσταση για τα προγράμματα, και χρησιμοποιεί στοιχεία από τη γραμμική άλγεβρα και τον γραμμικό προγραμματισμό για την ανάλυση μετασχηματισμών υψηλού επιπέδου.

Οι περισσότερες εφαρμογές που πραγματοποιούν έναν σημαντικό αριθμό υπολογισμών, αφιερώνουν τον περισσότερο χρόνο της εκτέλεσής τους σε εμφωλευμένους βρόχους. Αυτό συμβαίνει κυρίως σε επιστημονικές ή τεχνολογικές εφαρμογές. Μέσα από το πολυεδρικό μοντέλο παρέχεται η δυνατότητα να αποφασίσει το εργαλείο κατά πόσο είναι ορθός ένας περίπλοκος μετασχηματισμός βρόχων, με τη βοήθεια της γραμμικής άλγεβρας και του γραμμικού προγραμματισμού.

Το εργαλείο PLUTO αποσκοπεί στην αναζήτηση μετασχηματισμών παραλληλοποίησης για αποτελεσματική εφαρμογή των τεχνικών loop tiling και loop fusion, χωρίς όμως να περιορίζεται σε αυτές. Το loop tiling αποτελεί έναν μετασχηματισμό βελτιστοποίησης καθώς βελτιώνει την τοπικότητα των δεδομένων, με συνέπεια να αυξάνεται η απόδοση της κρυφής μνήμης. Παράλληλα είναι και μετασχηματισμός παραλληλοποίησης λόγω του ότι μπορεί να δημιουργήσει ανεξάρτητους υπολογισμούς που μπορούν να εκτελεστούν ταυτόχρονα σε διαφορετικούς επεξεργαστές. Έτσι, βασικός στόχος του PLUTO είναι η εύρεση τρόπων για την αποδοτική εφαρμογή του loop tiling.

Ο PLUTO δέχεται προγράμματα σε C και παράγει κώδικα σε C που περιέχει οδηγίες του μοντέλου OpenMP. Παρ' όλα αυτά, είναι πολύ εύκολο να τροποποιηθεί η δομή του συστήματος, ώστε να δέχεται κώδικα από οποιαδήποτε γλώσσα υψηλού επιπέδου αρκεί αυτή να μπορεί να αναπαρασταθεί και να αναλυθεί με βάση το πολυεδρικό μοντέλο.

Το PLUTO υλοποιείται χρησιμοποιώντας άλλα εργαλεία στα οποία βασίζεται για να εφαρμόσει τους μετασχηματισμούς του. Η λεκτική και η συνταχτική ανάλυση καθώς και ο έλεγχος εξαρτήσεων γίνονται με τη βοήθεια του LooPo [28] που πραγματοποιεί πολυεδρικούς μετασχηματισμούς κώδικα, περιλαμβάνοντας υλοποιήσεις ποικίλων πολυεδρικών αναλύσεων και μετασχηματισμών που υπάρχουν στη βιβλιογραφία. Χρησιμοποιεί επίσης το PipLib 1.3.3 [29] για την επίλυση των σχέσεων γραμμικού προγραμματισμού (ILP - Integer Linear Programming) και το CLooG 0.14.1 [30] για την παραγωγή του κώδικα.

5.2.6 NANOS

Με τον όρο NANOS [31, 32] απευθυνόμαστε σε ένα σύνολο συστατικών. Στο περιβάλλον ανάπτυξης εφαρμογών που αποτελείται από ένα εργαλείο οπτικοποίησης της δομής και των μετασχηματισμών μίας ερφαρμογής, σε έναν μεταφραστή για μία επέκταση του μοντέλου OpenMP, στη βιβλιοθήκη χρόνου εκτέλεσης για νήματα επιπέδου χρήστη Nthlib (NANOS Threads Library), σε ένα εργαλείο για την οπτικοποίηση της απόδοσης και της ανάλυσης του κώδικα, σε έναν διαχειριστή του επεξεργαστή και σε ένα εργαλείο για την οπτικοποίηση της δραστηριότητας του συστήματος. Το περιβάλλον NANOS ενσωματώνει τεχνικές μετάφρασης, συστήματος χρόνου εκτέλεσης και λειτουργικού συστήματος, με στόχο την επίτευξη υψηλής απόδοσης στις παράλληλες εφαρμογές σε πολυπρογραμματιστικά περιβάλλοντα με πολυεπεξεργαστές κοινής μνήμης.

Ο μεταφραστής NANOS είναι υλοποιημένος στην κορυφή του Parafrase-2 και αποσκοπεί στην παραλληλοποίηση προγραμμάτων σε Fortran 77 που περιλαμβάνουν οδηγίες του μοντέλου OpenMP με κάποιες επεκτάσεις. Ο κώδικας που παράγει είναι σε Fortran και περιλαμβάνει κλήσεις σε νήματα επιπέδου χρήστη της βιβλιοθήκης Nthlib, καθώς επίσης και στη βιβλιοθήκη εργαλείων για την πρόβλεψη της απόδοσης χρησιμοποιώντας το εργαλείο DIMEMAS και τον χαρακτηρισμό της παραλληλίας και της ανάλυσης της απόδοσης με το εργαλείο PARAVER. Αν το πρόγραμμα εισόδου περιλαμβάνει οδηγίες για το OpenMP, τότε ο μεταφραστής παραλληλοποιεί τον κώδικα με βάση τις οδηγίες και πραγματοποιεί ανάλυση του κώδικα για να εξάγει επιπλέον παραλληλισμό που δεν περιγράφεται με αυτές. Υπάρχει επίσης η δυνατότητα να μη στηριχτεί στις οδηγίες του προγραμματιστή, αλλά να εξάγει τον παραλληλισμό μόνο από την ανάλυση που ο ίδιος πραγματοποιεί.

Είναι σε θέση να αναγνωρίσει παραλληλισμό πολλαπλών επιπέδων, βασίζοντας την εξαγωγή του σε βρόχους και κλήσεις συναρτήσεων. Η αναπαράσταση κάθε μορφής παραλληλισμού γίνεται μέσω του γραφήματος ιεραρχικών εργασιών (HTG - Hierarchical Task Graph) που αποτελεί την ενδιάμεση αναπαράσταση του κώδικα. Με τον όρο εργασία, ο μεταφραστής NANOS αναφέρεται σε ένα τμήμα κώδικα το οποίο οριοθετείται από φυσικά όρια, όπως είναι μία εντολή, ένας βρόχος, μία κλήση συνάρτησης ή ένα βασικό μπλοκ. Η προτεραιότητα εκτέλεσης των διεργασιών υπολογίζεται σύμφωνα με τον έλεγχο και την ανάλυση των εξαρτήσεων που πραγματοποιείται στα δεδομένα. Ο μεταφραστής είναι υπεύθυνος για την κατανομή του φόρτου εργασίας μεταξύ των νημάτων, τη δημιουργία των προτεραιοτήτων εκτέλεσης και την επιλογή του μηχανισμού παράλληλης εκτέλεσης.

Ο μεταφραστής NANOS προσφέρει στον προγραμματιστή ένα πρωτότυπο εργαλείο γρα-

φικής διεπαφής μέσω του οποίου μπορεί να εισάγει οδηγίες για το μοντέλο OpenMP κατά την ώρα της μετάφρασης, συμβουλεύομενος το γράφημα ιεραρχικών εργασιών. Έτσι ο προγραμματιστής μπορεί να κατευθύνει τη διαδικασία μετάφρασης, είτε επιτρέποντας στον μεταφραστή να εφαρμόσει τους μετασχηματισμούς παραλληλοποίησης που είναι εφικτοί με βάση την ανάλυση που ο ίδιος πραγματοποιεί, είτε καθορίζοντας εκείνος τη μορφή παραλληλισμού που επιθυμεί, τροποποιώντας τα τμήματα κώδικα που ορίζουν μία εργασία.

5.2.7 PROMIS

Ο μεταφραστής PROMIS [33] είναι ο διάδοχος των Parafrase-2 και EVE. Εξάγει στατικό και δυναμικό παραλληλισμό πολλαπλών επιπέδων, τόσο σε υψηλό επίπεδο όσο και σε επίπεδο εντολών (instruction-level), ενώ πραγματοποιεί και βελτιστοποίηση. Υποστηρίζει μία σειρά από γλώσσες προγραμματισμού όπως η C, η C++, η Java και η Fortran. Ο κώδικας που παράγει προορίζεται για διαφορετικές αρχιτεκτονικές συνόλου εντολών (instruction-set architectures) χωρίς να αλλάζει η αρχιτεκτονική του μεταφραστή. Συγκεκριμένα παράγει κώδικα για RISC, CISC και DSP. Βελτιώνει σημαντικά την απόδοση των προγραμμάτων καθώς εξάγει παραλληλισμό διαφορετικών επιπέδων και μπορεί πολύ εύκολα να προσαρμοστεί σε διαφορετικά συστήματα.

Η ενδιάμεση αναπαράσταση του κώδικα έχει ιεραρχική μορφή, είναι ενιαία για όλες τις γλώσσες εισόδου και αποτελεί τον πυρήνα του μεταφραστή. Το εμπρόσθιο (front-end) και το οπίσθιο τμήμα (back-end) ενσωματώνονται μέσω της ενδιάμεσης αναπαράστασης, με αποτέλεσμα η πληροφορία να διαδίδεται από πάνω προς τα κάτω. Όπως οι περισσότεροι μεταφραστές παραλληλοποίησης, έτσι και ο PROMIS αποτελείται από ένα σύνολο παραδοσιακών τεχνικών ανάλυσης και μετασχηματισμού που είναι απαραίτητες για τη μετάφραση. Όμως, στον μεταφραστή PROMIS αυτές οι τεχνικές είναι υλοποιημένες εντός του πλαισίου της ανάλυσης συμβόλων, κάτι που επιτρέπει την επίτευξη του καλύτερου δυνατού αποτελέσματος στις τεχνικές βελτιστοποίησης. Μερικές από τις τεχνικές αυτές είναι η απαλοιφή των επαγωγικών μεταβλητών, η ανάλυση εξαρτήσεων των συμβόλων, η αναγνώριση ιδιωτικών πινάκων και η στατική ανάλυση της απόδοσης. Ο μεταφραστής PROMIS πραγματοποιεί επίσης ανάλυση δεικτών και θέτει ως βασικό στόχο τη δημιουργία μίας ποσοτικής μετρικής για την αξιολόγηση της συνεργασίας μεταξύ της ανάλυσης συμβόλων και της ανάλυσης δεικτών.

5.2.8 PARADIGM

Ο PARADIGM [34, 35] δέχεται ως είσοδο ένα σειριακό πρόγραμμα σε Fortran 77 ή στην επέκταση HPF (High Performance Fortran) και παράγει κώδικα που εκτελείται αποτελεσματικά σε ένα σύνολο υπολογιστών (multicomputers) κατανεμημένης μνήμης (distributed-memory). Το όνομά του αποτελεί και περιγραφή του ρόλου του, καθώς προέρχεται από το PARAllelizing compiler for DIstributed memory General purpose Multicomputers.

Τα συστήματα υπολογιστών κατανεμημένης μνήμης όπως ο Intel iPSC/860, ο Intel Paragon, ο IBM SP-1 και ο CM-5 προσφέρουν σημαντικά πλεονεκτήματα σε σχέση με τους υπολογιστές πολλαπλών επεξεργαστών κοινής μνήμης. Δυστυχώς, η αξιοποίηση των

υπολογιστών αυτών απαιτεί και αποτελεσματικό κώδικα από την πλευρά του προγραμματιστή, κάτι που μπορεί να χαρακτηριστεί ως μία αρκετά κοπιαστική διαδικασία. Ένας από τους βασικούς λόγους που είναι δύσκολη η δουλειά του προγραμματιστή, είναι η απουσία ενός καθολικού χώρου διευθύνσεων, με αποτέλεσμα ο προγραμματιστής να πρέπει να αναλάβει την κατανομή του κώδικα και των δεδομένων ανάμεσα στους επεξεργαστές, και να διαχειριστεί ρητά τη μεταξύ τους επικοινωνία.

Τη λύση στο πρόβλημα καλείται να δώσει ο μεταφραστής PARADIGM, οποίος παρέχει ένα αυτοματοποιημένο μέσο για την παραλληλοποίηση και τη βελτιστοποίηση των προγραμμάτων. Πέρα από την εφαρμογή των παραδοσιακών βελτιστοποιήσεων που πραγματοποιούν οι περισσότεροι μεταφραστές, ο PARADIGM πρωτοτυπεί σε θέματα όπως είναι η αυτόματη κατανομή των υπολογισμών που ορίζονται από τον προγραμματιστή μέσω οδηγιών, η σύνθεση επικοινωνίας υψηλού επιπέδου, η υποστήριξη των υπολογισμών που δεν είναι γνωστοί κατά τον χρόνο μετάφρασης (irregular computations) χρησιμοποιώντας έναν συνδυασμό ανάλυσης σε χρόνο μετάφρασης και υποστήριξης σε χρόνο εκτέλεσης, η εκμετάλλευση του παραλληλισμού μεταξύ των συναρτήσεων αλλά και μεταξύ των δεδομένων μίας συνάρτησης, και η παραγωγή πολυνηματικού κώδικα για την ανοχή των καθυστερήσεων που παρουσιάζονται κατά την επικοινωνία.

Ο PARADIGM χρησιμοποιεί τον Parafrase-2 για την προεπεξεργασία του κώδικα έτσι ώστε να παράγει την ενδιάμεση αναπαράσταση μαζί με τα γραφήματα ροής, εξαρτήσεων και κλήσεων. Μία σειρά από περάσματα όπως η διάδοση σταθερών και η απαλοιφή επαγωγικών μεταβλητών πραγματοποιούνται στο στάδιο αυτό, ενώ οι υπόλοιποι μετασχηματισμοί και τεχνικές βελτιστοποίησης εκτελούνται σε μετεγενέστερα.

5.2.9 C2μTC/SL

Ο μεταφραστής παραλληλοποίησης C2μTC/SL [36, 37, 38, 39] αποσκοπεί στην αυτόματη εξαγωγή παραλληλίας από βρόχους. Δέχεται ως είσοδο προγράμματα σε C, ενώ απεικονίζει τον κώδικα στο μοντέλο SVP.

Σε αντίθεση με τους περισσότερους μεταφραστές παραλληλοποίησης, ο C2μTC/SL πραγματοποιεί τη χρονοδρομολόγηση των οικογενειών νημάτων σε χρόνο εκτέλεσης και όχι σε χρόνο μετάφρασης. Αυτό το επιτυγχάνει μέσω του κατάλληλου κώδικα που προστίθεται στο αρχείο που παράγει και ο οποίος έχει τις αρμοδιότητες του χρονοδρομολογητή σε χρόνο εκτέλεσης. Η διαφοροποίηση αυτή οφείλεται στο γεγονός ότι η πληροφορία που είναι διαθέσιμη σε χρόνο εκτέλεσης είναι περισσότερη από αυτήν σε χρόνο μετάφρασης, κάτι που εκμεταλλεύεται για τη βελτίωση της απόδοσης. Φυσικά, η υλοποίηση του χρονοδρομολογητή έχει και τα μειονεκτήματά της καθώς δημιουργεί επιπλέον υπολογισμούς για το πρόγραμμα που εκτελείται, όμως τα πειράματα αποδεικνύουν ότι τις περισσότερες φορές το μέγεθος των υπολογισμών είναι αμελητέο σε σχέση με το συνολικό μέγεθος των υπολογισμών ολόκληρου του προγράμματος.

Κεφαλαιο 6

Σχεδιαση του Μεταφραστη Ariadne

6.1 Εισαγωγή

Η *Ariadne* είναι ένας μεταφραστής παραλληλοποίησης με εξειδίκευση στις αναδρομικές συναρτήσεις. Παρέχει τρία περάσματα μετασχηματισμού κώδικα, κάθε ένα από τα οποία αναλαμβάνει την παραλληλοποίηση διαφορετικής μορφής αναδρομικής συνάρτησης. Η παραλληλοποίηση πραγματοποιείται αυτόματα με τη βοήθεια σύντομων οδηγιών από τον προγραμματιστή. Η γλώσσα εισόδου είναι η ANSI C, ενώ η γλώσσας εξόδου είναι είτε η γλώσσα SL, είτε η ANSI C περιλαμβάνοντας κλήσεις για νήματα του προτύπου POSIX. Προφανώς, μπορεί να είναι και ο συνδυασμός τους, το οποίο πολλές φορές είναι απαραίτητο για την επίτευξη της βέλτιστης απόδοσης, ανάλογα με τη φύση κάθε αναδρομικής συνάρτησης.

Οι αναδρομικές συναρτήσεις που υποστηρίζει διακρίνονται σε δύο βασικές κατηγορίες. Σε αυτές που μπορούν να εκτελεστούν και επαναληπτικά, και σε αυτές που είναι από τη φύση τους αναδρομικές. Στις αναδρομικές συναρτήσεις της πρώτης κατηγορίας δεν είναι προφανής ο τρόπος με τον οποίο μπορεί να εξαχθεί ο παραλληλισμός. Για καλύτερη κατανόηση, στο σημείο αυτό θα ορίσουμε τρεις υποκατηγορίες. Στην πρώτη ανήκουν οι συναρτήσεις οι οποίες χαρακτηρίζονται από την ύπαρξη ενός δείκτη αλλά δεν παρουσιάζουν κάποια συνηθισμένη μορφή παραλληλίας. Στη δεύτερη ανήκουν οι συναρτήσεις οι οποίες χαρακτηρίζονται

από την ύπαρξη ενός δείκτη και μπορούν να υπολογιστούν παράλληλα, κατανέμοντας τον φόρτο εργασίας σε μία σειρά από νήματα. Ενώ στην τρίτη, δεν υπάρχει δείκτης και δεν υποστηρίζονται από τον μεταφραστή *Ariadne*.

Οι αναδρομικές συναρτήσεις που συναντάμε, έχουν συνήθως μία κύρια δομή ελέγχου στην παρακάτω μορφή. Για να αναγνωριστούν από τον μεταφραστή *Ariadne*, ο ορισμός τους θα πρέπει να βασίζεται στη δομή αυτή.

```
if (...)
{
        return ...;
}
else if (...)
{
        return ...;
}
else
{
        return ...;
}
```

Το πρώτο πέρασμα είναι το *elimination*, αναφέρεται στις συναρτήσεις της πρώτης υποκατηγορίας και είναι αρμόδιο για την εξάλειψη της αναδρομής, μετατρέποντας μία συνάρτηση από αναδρομική σε επαναληπτική, όταν αυτό είναι εφικτό. Το δεύτερο πέρασμα είναι το *parallel-reduction*, αναφέρεται στις συναρτήσεις της δεύτερης υποκατηγορίας και πέρα από την εξάλειψη της αναδρομής, κατανέμει τον φόρτο εργασίας σε έναν αριθμό νημάτων με στόχο την εξαγωγή παραλληλίας. Το *thread-safe* είναι το τρίτο και τελευταίο πέρασμα το οποίο παραλληλοποιεί αναδρομικές συναρτήσεις που έχουν περισσότερες από μία ανεξάρτητες αναδρομικές κλήσεις και είναι από τη φύση τους αναδρομικές.

Στα περάσματα *elimination* και *parallel-reduction*, η τιμή του δείκτη αποτελεί κριτήριο τερματισμού της αναδρομής και σημείο αναφοράς για τον μετασχηματισμό της συνάρτησης, από αναδρομική σε επαναληπτική. Ο δείκτης αυτός είναι μία ακέραια τιμή, η οποία σε κάθε κλήση αυξάνεται ή μειώνεται κατά μία ακέραια θετική σταθερά, την οποία καλούμε εξάρτηση της αναδρομικής συνάρτησης.

Η διάκριση μεταξύ των δύο αυτών περασμάτων γίνεται με βάση το πλήθος των αναδρομικών κλήσεων με διαφορετική εξάρτηση, καθώς κάθε αναδρομική κλήση αποτελεί επιπλέον εξάρτηση στην προσπάθεια για εξαγωγή παραλληλισμού.

Η δυνατότητα παραλληλίας μέσω του περάσματος *elimination* δεν είναι προφανής, αλλά εξαρτάται από το μοντέλο στο οποίο θα απεικονιστεί ο κώδικας. Η φύση του μοντέλου SVP προσφέρει τη δυνατότητα εξαγωγής παραλληλίας από συναρτήσεις αυτής της μορφής, δεν κάνει όμως το ίδιο το πρότυπο νημάτων POSIX. Αυτός είναι ένας από τους λόγους επιλογής του SVP, ως μοντέλο για την απεικόνιση του κώδικα εξόδου.

Το πέρασμα *elimination* είναι κατάλληλο για αναδρομικές συναρτήσεις όπως ο υπολογισμός της ακολουθίας Fibonacci, καθώς δεν είναι δυνατόν να κατανεμηθεί ο φόρτος εργασίας σε περισσότερα από ένα νήματα (λόγω εξαρτήσεων), αλλά μπορεί να γίνει απαλοιφή της αναδρομής. Σε μία αναδρομική συνάρτηση για τον υπολογισμό του παραγοντικού, μπορεί να εφαρμοστεί το πέρασμα *parallel-reduction*, έτσι ώστε να κατανεμηθούν οι επαναλήψεις σε έναν αριθμό νημάτων. Αναδρομικές συναρτήσεις που υλοποιούν αλγορίθμους ταξινόμησης, όπως ο Quick-Sort και ο Merge-Sort, έχουν παραπάνω από μία ανεξάρτητες αναδρομικές κλήσεις, οπότε η εκτέλεση του περάσματος *thread-safe* παρουσιάζει πολύ καλά αποτελέσματα. Γενικότερα, το πέρασμα *thread-safe* είναι κατάλληλο για την παραλληλοποίηση συναρτήσεων που ακολουθούν τη μέθοδο διαίρει και βασίλευε (divide and conquer).

6.2 Περιγραφή Οδηγιών

Η παραλληλοποίηση των αναδρομικών συναρτήσεων από τον μεταφραστή *Ariadne* απαιτεί μία απλή οδηγία από τον προγραμματιστή. Η οδηγία αυτή αφορά μία και μοναδική αναδρομική συνάρτηση και δίνεται αμέσως πριν τον ορισμό της. Η εφαρμογή ενός περάσματος από τον μεταφραστή, αποτελεί αποκλειστική ευθύνη του προγραμματιστή, την οποία αναλαμβάνει μέσω της οδηγίας που δίνει. Έτσι, κάθε πέρασμα πραγματοποιείται αν και μόνο αν υπάρχει ρητή οδηγία από τον προγραμματιστή. Αν και η ευθύνη ανήκει στον προγραμματιστή, ο μεταφραστής *Ariadne* πραγματοποιεί μία σειρά από ελέγχους για να μειώσει την πιθανότητα σφάλματος κατά την παραλληλοποίηση των αναδρομικών συναρτήσεων.

Για κάθε πέρασμα υπάρχει μία παραμετροποιημένη οδηγία, η οποία έχει το όνομα του περάσματος. Η εφαρμογή του περάσματος *elimination* απαιτεί την παρακάτω οδηγία, όπου model είναι το μοντέλο στο οποίο θα απεικονιστεί ο κώδικας εξόδου. Για απεικόνιση της αναδρομικής συνάρτησης στο μοντέλο SVP, θα πρέπει κατά την οδηγία να οριστεί ρητά, μέσω της λέξης svp. Στο συγκεκριμένο πέρασμα, ο μεταφραστής *Ariadne* δεν προσφέρει τη δυνατότητα απεικόνισης του κώδικα στο πρότυπο νημάτων POSIX γιατί δεν υπάρχει τρόπος για εξαγωγή παραλληλίας. Αν δεν οριστεί ρητά το model, τότε ο κώδικας εξόδου θα είναι κώδικας σε C, δηλαδή θα γίνει απλή εξάλειψη της αναδρομής.

```
#pragma ariadne elimination [model]
```

Το πέρασμα *parallel-reduction* εκτελείται μέσω παρόμοιας οδηγίας, με τη διαφορά ότι διαθέτει περισσότερες παραμέτρους. Στο πέρασμα αυτό, ο φόρτος εργασίας κατανέμεται σε έναν αριθμό νημάτων ή οικογενειών νημάτων ο οποίος ορίζεται υποχρεωτικά μέσω μίας ακέραιας θετικής σταθεράς N. Το τελικό αποτέλεσμα υπολογίζεται μετά από συνδυασμό των επιμέρους αποτελεσμάτων όλων των νημάτων. Η πράξη που θα εφαρμοστεί για τον υπολογισμό του τελικού αποτελέσματος, υποδεικνύεται μέσω του op. Ο μεταφραστής *Ariadne* υποστηρίζει την πράξη της πρόσθεσης, της αφαίρεσης, του πολλαπλασιασμού και της διαίρεσης. Επομένως, το op αντιστοιχεί σε έναν από τους αριθμητικούς τελεστές '+', '-', '*' ή '/'. Οι τελεστές '-' και '/' προαιρετικά μπορούν να ακολουθούνται από τον τελεστή '~' (tilde). Η σημασία του θα εξηγηθεί λεπτομερώς στη συνέχεια του κεφαλαίου. Η παράμετρος model

αναφέρεται και σε αυτήν την περίπτωση, στο μοντέλο που θα απεικονιστεί ο κώδικας εξόδου. Ο καθορισμός του μοντέλου είναι υποχρεωτικός και υπάρχουν δύο επιλογές. Για απεικόνιση στο μοντέλο SVP απαιτείται η λέξη svp, ενώ για απεικόνιση σε C με χρήση νημάτων του προτύπου POSIX, απαιτείται η λέξη posix.

#pragma ariadne parallel-reduction(op[~]) model threads(N)

Η οδηγία *thread-safe* είναι αρμόδια για την παραλληλοποίηση αναδρομικών συναρτήσεων με δύο ή παραπάνω ανεξάρτητες κλήσεις στον εαυτό της. Εκτός από τον καθορισμό του μοντέλου που είναι υποχρεωτικός και γίνεται μέσω του model, ακριβώς όπως και στην οδηγία *parallel-reduction*, υπάρχει και η παράμετρος level. Το level αναφέρεται στο επίπεδο, μέχρι το οποίο θα δημιουργούνται νήματα, ενώ για μεγαλύτερο επίπεδο οι κλήσεις θα γίνονται αναδρομικά. Το N πρέπει να είναι ακέραια θετική σταθερά. Για παράδειγμα, αν οριστεί στην τιμή 1, θα έχει ως συνέπεια, την πρώτη φορά να δημιουργηθεί ένα νήμα για κάθε ανεξάρτητη κλήση. Οι επόμενες κλήσεις, δηλαδή αυτές που θα πραγματοποιήσει κάθε ένα από τα νήματα αυτά, θα γίνουν αναδρομικά.

#pragma ariadne thread-safe model level(N)

6.3 Το Πέρασμα elimination

Το πέρασμα *elimination* αποσκοπεί στην εξάλειψη της αναδρομής, μετασχηματίζοντας μία συνάρτηση από αναδρομική σε επαναληπτική. Η εφαρμογή του περάσματος σε μία συνάρτηση προϋποθέτει μία απλή οδηγία από τον προγραμματιστή αμέσως πριν τον ορισμό της. Η ύπαρξη οδηγίας για το πέρασμα *elimination*, δε συνεπάγεται και την εκτέλεση του μετασχηματισμού από τον μεταφραστή *Ariadne*.

Για προστασία του προγραμματιστή, ο μεταφραστής αναλαμβάνει μία σειρά από ελέγχους. Όμως, οι έλεγχοι αυτοί δεν αφορούν γενικές εξαρτήσεις, όπως για παράδειγμα ενδεχόμενες κλήσεις της συνάρτησης προς άλλες συναρτήσεις οι οποίες επηρεάζουν την εκτέλεσή της. Για να εφαρμοστεί το πέρασμα θα πρέπει να εξάγει όλη την απαραίτητη πληροφορία, αφού πρώτα εξασφαλίσει ότι η συνάρτηση είναι αναδρομική και τηρούνται όλες οι προϋποθέσεις.

Συγκεκριμένα, θα πρέπει να εξετάσει αν η συνάρτηση έχει τουλάχιστον μία παράμετρο και αν ακριβώς μία από αυτές έχει το ρόλο του δείκτη. Κάθε μπλοκ της κύριας δομής ελέγχου θα πρέπει να ανήκει είτε στα υποψήφια μπλοκ αρχικών τιμών είτε στα μπλοκ αναδρομής. Να υπάρχει τουλάχιστον ένα μπλοκ κάθε κατηγορίας και τα μπλοκ αναδρομής να είναι το πολύ δύο. Εφόσον είναι δύο, στο ένα από αυτά πρέπει να γίνεται αύξηση του δείκτη σε κάθε κλήση και στο άλλο μείωση. Στη συνέχεια θα πρέπει να αναγνωρίσει τις τιμές του δείκτη για τις οποίες γίνονται οι αναδρομικές κλήσεις και τις αρχικές τιμές, δηλαδή αυτές με βάση τις οποίες υπολογίζεται το αποτέλεσμα της τελευταίας κλήσης. Τέλος, θα πρέπει να αναγνωρίσει τη μέγιστη εξάρτηση της αναδρομικής συνάρτησης.

Ο τύπος της συνάρτησης θα πρέπει να είναι ένας από τους βασικούς τύπους, και όχι δομή (struct), ένωση (union), απαριθμητής (enum) ή δείκτης (pointer).

6.3.1 Αναγνώριση Δείκτη Αναδρομικής Συνάρτησης

Για να είναι δείκτης μία παράμετρος, θα πρέπει να εμφανίζεται σε κάθε συνθήκη ελέγχου της κύριας δομής και η τιμή της να αποτελεί κριτήριο τερματισμού της αναδρομής. Θα πρέπει επίσης σε κάθε αναδρομική κλήση, να αυξάνεται ή να μειώνεται κατά μία ακέραια θετική σταθερά, την οποία καλούμε εξάρτηση. Ο τύπος της πρέπει να είναι ένας από τους βασικούς ακέραιους τύπους της C.

6.3.2 Διαχωρισμός των Μπλοκ της Κύριας Δομής Ελέγχου

Κάθε μπλοκ (block) της κύριας δομής ελέγχου στο οποίο δε γίνεται αναδρομική κλήση, αλλά επιστρέφεται η τιμή της συνάρτησης, χαρακτηρίζεται από τον μεταφραστή *Ariadne* ως υποψήφιο μπλοκ αρχικών τιμών, ενώ κάθε μπλοκ στο οποίο υπάρχει τουλάχιστον μία αναδρομική κλήση, χαρακτηρίζεται ως μπλοκ αναδρομής. Κάθε μπλοκ της κύριας δομής ελέγχου θα πρέπει να χαρακτηριστεί από τον μεταφραστή είτε ως υποψήφιο μπλοκ αρχικών τιμών είτε ως μπλοκ αναδρομής. Σε κάθε άλλη περίπτωση είναι αδύνατη η εφαρμογή του περάσματος. Τα μπλοκ αναδρομής είναι αυτά στα οποία θα πραγματοποιηθεί η απαλοιφή των αναδρομικών κλήσεων και θα αντικατασταθούν από μεταβλητές οι οποίες θα είναι αρμόδιες για τον υπολογισμό του αποτελέσματος. Τα υποψήφια μπλοκ αρχικών τιμών είναι αυτά από τα οποία θα εξαχθούν οι τιμές με τις οποίες θα αρχικοποιηθούν οι μεταβλητές. Ο όρος υποψήφια υποδεικνύει την ανάγκη για επιπλέον έλεγχο όσον αφορά το πότε ένα τέτοιο μπλοκ περιέχει τις τιμές στις οποίες θα αρχικοποιηθούν οι μεταβλητές. Αυτό συμβαίνει γιατί σε μία αναδρομική συνάρτηση, ενδεχομένως να υπάρχουν περισσότερα από ένα υποψήφια μπλοκ αρχικών τιμών, τα οποία όμως έχουν άλλο ρόλο κατά την εκτέλεσή της.

6.3.3 Εξαγωγή Αρχικών Τιμών

Η εξαγωγή των αρχικών τιμών αποτελεί ένα ενδιαφέρον κομμάτι των αναδρομικών συναρτήσεων. Για να επιτευχθεί η αναγνώριση θα πρέπει να ορίσουμε δύο λίστες. Ονομάζουμε την πρώτη plus και τη δεύτερη minus. Στις δύο αυτές λίστες διατηρούμε τριάδες στις οποίες το πρώτο μέρος είναι ένας εκ των τριών συγκριτικών τελεστών '<', '==' και '>' που αντιστοιχεί σε συγκριτικό τελεστή λογικής συνθήκης ενός υποψήφιου μπλοκ αρχικών τιμών. Το δεύτερο μέρος είναι η τιμή με την οποία συγκρίνεται ο δείκτης στη λογική συνθήκη, ενώ το τρίτο μέρος είναι η τιμή που επιστρέφει η αναδρομική συνάρτηση μέσα από το αντίστοιχο υποψήφιο μπλοκ αρχικών τιμών. Στο μπλοκ αρχικών τιμών που ακολουθεί, ο δείκτης της συνάρτησης είναι το x και η τριάδα που διατηρείται στη λίστα είναι [<, 6, 10].

```
if (x <6) { return (10); }
```

Συγκεκριμένα, στη λίστα plus διατηρούμε τις τριάδες που αντιστοιχούν σε λογικές συνθήκες των υποψήφιων μπλοκ αρχικών τιμών όταν ο δείκτης της συνάρτησης αυξάνεται σε κάθε κλήση, ενώ στη λίστα minus όταν ο δείκτης μειώνεται.

Γίνεται εύκολα αντιληπτό, ότι οι δύο αυτές λίστες δεν έχουν απαραίτητα τις ίδιες τριάδες, και αυτό συμβαίνει διότι μία λογική συνθήκη υπάρχει περίπτωση να μην εξεταστεί ποτέ σε

χρόνο εκτέλεσης, ανάλογα με το αν ο δείκτης της αυξάνεται ή μειώνεται. Έτσι, υπάρχουν υποψήφια μπλοκ αρχικών τιμών από τα οποία σίγουρα δε θα εξαχθούν οι αρχικές τιμές της αναδρομικής συνάρτησης.

Για απλοποίηση των διαφορετικών περιπτώσεων που προκύπτουν, οι τριάδες με συγκριτικό τελεστή (>) και (<) μετατρέπονται σε (>=) και (<=) αντίστοιχα, μειώνοντας ή αυξάνοντας την αντίστοιχη τιμή της τριάδας. Στη συνέχεια όταν αναφερόμαστε σε τριάδες με τον τελεστή (>) θα εννοούμε (>=), και όταν αναφερόμαστε σε τριάδες με τον τελεστή (<) θα εννοούμε (<=).

Ας εξετάσουμε λοιπόν την περίπτωση όπου ο δείκτης της αναδρομικής συνάρτησης μειώνεται σε κάθε κλήση. Τα υποψήφια μπλοκ αρχικών τιμών με λογικές συνθήκες της μορφής (index > value) δεν μπορούν να αποτελούν μπλοκ αρχικών τιμών εφόσον η συνθήκη προηγείται του αντίστοιχου μπλοκ της κύριας δομής ελέγχου στο οποίο πραγματοποιείται η αναδρομική κλήση. Αυτό ισχύει επειδή αν η τιμή value είναι μικρότερη της τιμής για την οποία εκτελείται τελευταία φορά η αναδρομική κλήση, τότε ο κώδικας είναι λανθασμένος, καθώς το μπλοκ που περιέχει την αναδρομική κλήση δε θα εκτελεστεί ποτέ. Σε περίπτωση που η τιμή value είναι μεγαλύτερη, τότε ο δείκτης ποτέ δε θα φτάσει σε αυτήν την τιμή επειδή συνέχεια θα μειώνεται. Από την άλλη, οποιαδήποτε λογική συνθήκη της μορφής (index < value) ή (index == value), καθώς και οποιαδήποτε λογική συνθήκη έπεται του αντίστοιχου μπλοκ στο οποίο πραγματοποιείται η αναδρομική κλήση και βρίσκεται στη μορφή (index > value), αποτελεί λογική συνθήκη υποψήφιου μπλοκ αρχικών τιμών.

Όλες οι πιθανές τριάδες συλλέγονται στην αντίστοιχη λίστα και ταξινομούνται με βάση την τιμή τους. Στο ένα άκρο της λίστας συγκεντρώνονται οι τριάδες (>), στο άλλο άκρο οι (<), και ανάμεσά τους υπάρχουν οι (==). Σε κάθε τριάδα η τιμή με την οποία συγκρίνεται ο δείκτης της συνάρτησης, υποδεικνύει το πλήθος των αρχικών τιμών που μπορούν να εξαχθούν από αυτή. Όταν ζητηθεί μία αρχική τιμή, η αναζήτησή της γίνεται σε ένα από τα δύο άκρα της λίστας ανάλογα με το αν αυξάνεται ή μειώνεται ο δείκτης. Στην περίπτωση που ο δείκτης μειώνεται, η εξαγωγή των αρχικών τιμών γίνεται ξεκινώντας από τις μεγαλύτερες τιμές και συνεχίζοντας προς τις μικρότερες. Στην πραγματικότητα οι μεγαλύτερες τιμές αντιστοιχούν στις συνθήκες (>) και οι μικρότερες στις συνθήκες (<), ενώ ανάμεσά τους βρίσκονται οι συνθήκες (==). Αν δεν ισχύει αυτό, τότε υπάρχει τομή στο διάστημα των αρχικών τιμών και η αναδρομική συνάρτηση δεν μπορεί να υποστηριχθεί από τον μεταφραστή *Ariadne*. Η παρουσία κενών διαστημάτων στις αρχικές τιμές δεν υποστηρίζεται επίσης από τον μεταφραστή *Ariadne*. Αν βρισκόμαστε στην περίπτωση όπου ο δείκτης αυξάνεται σε κάθε κλήση, τότε ισχύουν τα συμμετρικά όλων των παραπάνω.

6.3.4 Καθορισμός Ορίων και Βήματος

Η τιμή του δείκτη μαζί με τις αρχικές τιμές καθώς και το μέγεθος της μέγιστης εξάρτησης σε συνδυασμό με το πλήθος των αναδρομικών κλήσεων σε κάθε μπλοκ αναδρομής, είναι αυτά που καθορίζουν τα όρια και το βήμα του βρόχου που θα αντικαταστήσει την αναδρομή στην περίπτωση απλής εξάλειψης ή της οικογένειας νημάτων στην περίπτωση απεικόνισης του κώδικα στο μοντέλο SVP. Στη γενική περίπτωση για να απαλειφθεί μία αναδρομική

συνάρτηση, κάθε αναδρομική κλήση θα πρέπει να αντικατασταθεί από μία μεταβλητή. Το σύνολο των μεταβλητών είναι ίσο με τη μέγιστη εξάρτηση και αρχικοποιείται με τις τιμές που εξάγονται από τα μπλοκ αρχικών τιμών. Στην αναδρομική συνάρτηση, η πρώτη αναδρομική κλήση είναι η τελευταία που θα τερματίσει καθώς το αποτέλεσμά της εξαρτάται από το αποτέλεσμα του εαυτού της για μία μικρότερη ή μεγαλύτερη τιμή του δείκτη. Έτσι, πρώτα θα τερματίσει η τελευταία αναδρομική κλήση και στη συνέχεια όλες οι άλλες.

Το βήμα ορίζεται με τέτοιο τρόπο ώστε ο δείκτης του βρόχου (ή το αναγνωριστικό της οικογένειας νημάτων αντίστοιχα) να λάβει όλες τις τιμές που λαμβάνει και ο δείκτης της συνάρτησης κατά τη διάρκεια των υπολογισμών, δηλαδή σε 1. Τα όρια είναι από την τιμή του δείκτη της προτελευταίας κλήσης κατά την αναδρομική εκτέλεση, μέχρι και την τιμή του δείκτη κατά την πρώτη κλήση της αναδρομικής συνάρτησης. Η προτελευταία κλήση προκύπτει από το γεγονός, ότι στην τελευταία κλήση θα γίνει επιστροφή της αρχικής τιμής, ένα βήμα το οποίο θα πρέπει να προσπεράσουμε καθώς η αρχικοποίηση των μεταβλητών γίνεται πριν την εκτέλεση του βρόχου (ή της δημιουργία της οικογένειας νημάτων αντίστοιχα).

Αν ένα μπλοκ αναδρομής έχει ακριβώς μία αναδρομική κλήση, τότε χρησιμοποιούμε μόνο μία μεταβλητή, ασχέτως την εξάρτηση. Διαφορά παρουσιάζεται και στο βήμα, το οποίο σε αυτήν την περίπτωση θα ισούται με την εξάρτηση της αναδρομικής κλήσης.

Η βασική ιδέα λοιπόν είναι ότι προσομοιώνουμε την εκτέλεση της αναδρομικής συνάρτησης. Το αποτέλεσμα διαδίδεται από τη μία επανάληψη στην άλλη μέσω των μεταβλητών οι οποίες πάντα είναι ενημερωμένες με τις τιμές που επιστρέφει η αναδρομική συνάρτηση για τον συγκεκριμένο δείκτη του βρόχου, λαμβάνοντας υπ' όψιν την εξάρτηση της εκάστοτε κλήσης.

6.3.5 Εξάλειψη Αναδρομικής Συνάρτησης

Η εξάλειψη αναδρομικής συνάρτησης δεν αποτελεί μετασχηματισμό παραλληλοποίησης, όμως ο μεταφραστής *Ariadne* προσφέρει τη δυνατότητα αυτή, καθώς ο μετασχηματισμός μίας συνάρτησης από αναδρομική σε επαναληπτική, έχει ως συνέπεια σημαντική βελτίωση στην απόδοση. Για την εφαρμογή του περάσματος *elimination* και την παραγωγή κώδικα εξόδου σε C είναι απαραίτητη η οδηγία που ακολουθεί.

#pragma ariadne elimination

Στον Πηγαίο Κώδικα 6.1 βρίσκεται μια αναδρομική συνάρτηση για τον υπολογισμό της ακολουθίας Fibonacci, ενώ στον Πηγαίο Κώδικα 6.2 βρίσκεται η έξοδος του μεταφραστή *Ariadne* μετά την εκτέλεση του περάσματος *elimination* με την παραπάνω οδηγία. Στη συνέχεια περιγράφουμε πως μπορεί η συνάρτηση του παραδείγματος να μετασχηματιστεί από αναδρομική σε επαναληπτική.

Η κύρια δομή ελέγχου αποτελείται από τέσσερα μπλοκ. Ο μεταφραστής *Ariadne* αρχικά θα χαρακτηρίσει τα τρία πρώτα μπλοκ ως υποψήφια μπλοκ αρχικών τιμών επειδή στον κώδικά τους γίνεται επιστροφή τιμής και δεν πραγματοποιείται αναδρομική κλήση. Το τέταρτο μπλοκ θα το χαρακτηρίσει ως μπλοκ αναδρομής επειδή εντός αυτού, πραγματοποιείται αναδρομική κλήση. Επομένως, η μορφή της κύριας δομής ελέγχου είναι αποδεκτή.

```
1  long long int fibonacci(int n)
2  {
3      if (n < 0)
4      {
5          return (-1);
6      }
7      else if (n > 40)
8      {
9          return (-2);
10     }
11     else if (n < 2)
12     {
13         return (n);
14     }
15     else
16     {
17         return (fibonacci(n - 1) + fibonacci(n - 2));
18     }
19 }
```

Πηγαίος Κώδικας 6.1: Αναδρομική συνάρτηση υπολογισμού της ακολουθίας Fibonacci.

```
1  long long int fibonacci(int n)
2  {
3      if (n < 0)
4      {
5          return (-1);
6      }
7      else if (n > 40)
8      {
9          return (-2);
10     }
11     else if (n < 2)
12     {
13         return (n);
14     }
15     else
16     {
17         int i;
18         long long int shv, sh1, sh2;
19
20         sh1 = 1;
21         sh2 = 0;
22
23         for (i = 2; i <= n; i += 1)
24         {
25             shv = sh1 + sh2;
26             sh2 = sh1;
27             sh1 = shv;
28         }
29
30         return (shv);
31     }
32 }
```

Πηγαίος Κώδικας 6.2: Συνάρτηση υπολογισμού της ακολουθίας Fibonacci μετά την εφαρμογή του περάσματος *elimination* για απεικόνιση σε κώδικα C.

Στη συνέχεια θα αντικαταστήσει τις δύο αναδρομικές κλήσεις του μπλοκ αναδρομής, με δύο τοπικές μεταβλητές, τις sh1 και sh2 και θα προβεί σε περαιτέρω ελέγχους με τους οποίους θα αναγνωρίσει ότι οι τιμές με τις οποίες θα αρχικοποιηθούν οι μεταβλητές sh1 και sh2 είναι οι 1 και 0 αντίστοιχα. Αυτό συμβαίνει γιατί τα δύο πρώτα μπλοκ της κύριας δομής ελέγχου δεν επηρεάζουν τον υπολογισμό του αποτελέσματος όταν ο δείκτης παίρνει τιμές στο διάστημα [0, 40], με βάση τα όσα αναφέραμε στην ενότητα για την αναγνώριση των αρχικών τιμών. Σε περίπτωση που ο δείκτης έχει τιμή εκτός του διαστήματος αυτού, τότε εξ' ορισμού δε γίνεται αναδρομική κλήση, άρα δεν επηρεάζεται το αποτέλεσμα της συνάρτησης. Το τρίτο μπλοκ της κύριας δομής ελέγχου αποτελεί το μπλοκ αρχικών τιμών. Για n <2 γίνεται επιστροφή του δείκτη, άρα οι δύο αρχικές τιμές είναι το 0, για n = 0, και το 1, για n = 1, και η τιμή του δείκτη κατά την προτελευταία κλήση είναι 2. Άρα το βήμα του βρόχου είναι 1 και τα όρια είναι από 2 έως και n, ενώ χρησιμοποιούμε τόσες μεταβλητές όσες και η μέγιστη εξάρτηση, δηλαδή 2. Η ενημέρωση των μεταβλητών γίνεται στο τέλος κάθε επανάληψης, ώστε να έχουν πάντα τη σωστή τιμή στην αρχή της επόμενης.

6.3.6 Απεικόνιση στο Μοντέλο SVP

Το πέρασμα *elimination* δεν είναι το καταλληλότερο για εξαγωγή παραλληλίας. Όμως, η απεικόνιση του κώδικα στο μοντέλο SVP προσφέρει αυτή τη δυνατότητα, σε αντίθεση με άλλα παρόμοια μοντέλα. Για την εφαρμογή του περάσματος *elimination* και την παραγωγή κώδικα για το μοντέλο SVP είναι απαραίτητη η οδηγία που ακολουθεί.

#pragma ariadne elimination svp

Στον Πηγαίο Κώδικα 6.3 βρίσκεται μια αναδρομική συνάρτηση για τον υπολογισμό της μαθηματικής συνάρτησης $f(x)$, ενώ στον Πηγαίο Κώδικα 6.4 βρίσκεται η έξοδος του μεταφραστή *Ariadne* μετά την εκτέλεση του περάσματος *elimination* με την παραπάνω οδηγία.

```
1  int g(int x);
2
3  int f(int x)
4  {
5      if (x == 0)
6      {
7          return (1);
8      }
9      else
10     {
11         return (2 * f(x - 1) + g(x));
12     }
13 }
```

Πηγαίος Κώδικας 6.3: Αναδρομική συνάρτηση για τον υπολογισμό της μαθηματικής συνάρτησης $f(x) = 2\,f(x-1) + g(x)$, όπου $f(0) = 1$.

Η κατανομή του φόρτου εργασίας της συγκεκριμένης συνάρτησης σε έναν αριθμό νημάτων με στόχο την παράλληλη εκτέλεσή τους, είναι αδύνατη λόγω εξαρτήσεων που την

χαρακτηρίζουν. Στη συνέχεια περιγράφουμε πως μπορεί η συνάρτηση αυτή να απεικονιστεί στο μοντέλο SVP και να εξαχθεί παραλληλισμός χάρη στη φύση του μοντέλου SVP.

```
1  int g(int x);
2
3  sl_def(f_t_n , void, sl_shparm(int, sh1))
4  {
5      int t1;
6      int t0;
7      int shv, sh1;
8
9      sl_index(x);
10
11     t0 = g(x);
12     sh1 = sl_getp(sh1);
13     t1 = 2 * sh1;
14     shv = t0 + t1;
15     sl_setp(sh1, shv);
16 }
17 sl_enddef
18
19 int f(int x)
20 {
21     if (x == 0)
22     {
23         return (1);
24     }
25     else
26     {
27         int shv;
28
29         {
30             sl_create(, , 1, x + 1, 1, , , f_t_n , sl_sharg(int, sh1, 1));
31             sl_sync();
32
33             shv = sl_geta(sh1);
34         }
35
36         return (shv);
37     }
38 }
```

Πηγαίος Κώδικας 6.4: Παραγόμενος κώδικας εξόδου για τον υπολογισμό της μαθηματικής συνάρτησης $f(x) = 2f(x-1) + g(x)$, όπου $f(0) = 1$, μετά την εφαρμογή του περάσματος *elimination* για απεικόνιση του κώδικα στο μοντέλο SVP.

Τα βήματα που ακολουθεί ο μεταφραστής *Ariadne* είναι ακριβώς τα ίδια με αυτά για την απεικόνιση του κώδικα σε C. Για το λόγο αυτό, θα εστιάσουμε στις μικρές διαφορές που παρουσιάζουν οι δύο αυτοί μετασχηματισμοί.

Η βασική τους διαφορά είναι ότι κατά την απεικόνιση στο μοντέλο SVP η αναδρομή αντικαθίσταται από μία οικογένεια νημάτων και όχι από έναν βρόχο. Παρ' όλα αυτά, τα όρια και το βήμα είναι ίδια με αυτά του βρόχου. Επίσης, οι μεταβλητές που αντικαθιστούν τις αναδρομικές κλήσεις, δεν είναι τοπικές μεταβλητές, αλλά κοινόχρηστες παράμετροι της γλώσσας SL μέσω των οποίων επικοινωνούν τα γειτονικά νήματα.

Η παραλληλία που εξάγει ο μεταφραστής *Ariadne* είναι κατά την κλήση της συνάρτησης g() η οποία μπορεί να γίνει ταυτόχρονα από όλα τα νήματα της οικογένειας. Για να επιτευχθεί αυτό, θα πρέπει ο υπολογισμός της g() να γίνει πριν την κλήση της sl_getp() και να εκχωρηθεί σε μία τοπική μεταβλητή. Έτσι, κάθε νήμα θα υπολογίσει το δικό του αποτέλεσμα πριν φθάσει στο σημείο που παρουσιάζεται η εξάρτηση από το προηγούμενό του νήμα. Το κέρδος από την παράλληλη εκτέλεση της συνάρτησης g(). εξαρτάται από το μέγεθος των υπολογισμών της.

Ο διαχωρισμός της αριθμητικής έκφρασης (expression) 2 * f(x - 1) + g(x), όπως και η εισαγωγή των κλήσεων sl_getp() και sl_setp() γίνεται αυτόματα από τον μεταφραστή, και περιγράφονται στις υποενότητες που ακολουθούν.

6.3.7 Διάσπαση Αριθμητικών Εκφράσεων

Ο προγραμματιστής γράφει κώδικα με τρόπο που τον διευκολύνει. αλλά αυτό δημιουργεί προβλήματα στην παραλληλία που μπορεί να εξαχθεί από ένα τμήμα κώδικα. Επομένως, ο μεταφραστής θα πρέπει να αναλάβει την ευθύνη να τροποποιήσει τον κώδικα έτσι ώστε να επιτύχει το δικό του σκοπό. Ο μεταφραστής *Ariadne* διασπά τις αριθμητικές εκφράσεις ενός προγράμματος όταν η διάσπαση αυτή θα έχει ως συνέπεια να βελτιωθεί η απόδοση του κώδικα εξόδου που θα παράγει. Αυτό συμβαίνει όταν ο κώδικας απεικονίζεται στο μοντέλο SVP, όπου υπάρχει ανάγκη για εκτέλεση όσο το δυνατόν περισσότερου κώδικα πριν από μία κλήση sl_getp().

Ο μεταφραστής *Ariadne* εξετάζει την αριθμητική έκφραση σε κάθε εντολή (statement) εκχώρησης τιμής και σε κάθε return. Στόχος του είναι να απομονώσει τις κοινόχρηστες παραμέτρους που βρίσκονται σε μία αριθμητική έκφραση, υπολογίζοντας οποιοδήποτε τμήμα της, που δεν περιέχει κοινόχρηστη παράμετρο, πριν από τη χρήση τους. Έτσι, στον κώδικα που παράγει πραγματοποιούνται όσο το δυνατόν περισσότεροι ανεξάρτητοι υπολογισμοί κατά την παράλληλη εκτέλεση των νημάτων μίας οικογένειας. δηλαδή πριν από μία κλήση sl_getp().

Σε κάθε αριθμητική έκφραση που περιλαμβάνει κοινόχρηστη παράμετρο, εφαρμόζει τα ακόλουθα βήματα. Αρχικά τη χωρίζει σε όρους. το πλήθος των οποίων είναι κομβικό για το διαχωρισμό που θα εφαρμοστεί. Αν η αριθμητική έκφραση αποτελείται από τρεις ή περισσότερους όρους, τότε το αποτέλεσμά της θα υπολογιστεί σε τόσες πράξεις εκχώρησης όσες και το πλήθος των όρων (αν υπάρχει παραπάνω από ένας όρος που δεν περιλαμβάνει καμία κοινόχρηστη παράμετρο, τότε τους θεωρούμε όλους ως έναν ενιαίο όρο, επειδή δεν υπάρχει λόγος να διασπαστούν). Η πρώτη θα έχει τον τελεστή εκχώρησης '=' και θα εκχωρεί τον πρώτο όρο, ενώ οι επόμενες τον '+=' ή '-=', και θα προσθέτουν ή θα αφαιρούν τους επόμενους. Το αποτέλεσμα εκχωρείται σε μία προσωρινή μεταβλητή που χρησιμοποιείται αποκλειστικά για το σκοπό αυτό και είναι η ίδια σε όλες τις εκχωρήσεις.

```
x = sh1 + a + 10;

t0 = a + 10;
t0 += sh1;
x = t0;
```

Αν η αριθμητική έκφραση έχει το πολύ δύο όρους, τότε δε διασπάται. Όμως, σε αυτήν την περίπτωση ο μεταφραστής *Ariadne* εξετάζει κάθε όρο ξεχωριστά, και αν κάποιος αποτελείται από δύο ή περισσότερους όρους ή παράγοντες, τότε αντικαθίσταται από μία προσωρινή μεταβλητή, στην οποία εκχωρείται ο όρος.

```
x = e * 6 + sh1;

t0 = e * 6;
x = t0 + sh1;
```

Αν ένας όρος είναι συγχρόνως και παράγοντας, δηλαδή δε διασπάται ούτε σε όρους ούτε σε παράγοντες, τότε δεν υπάρχει λόγος να αντικατασταθεί από μία προσωρινή μεταβλητή, καθώς η αντικατάστασή του δε θα προσφέρει επιπλέον παραλληλία στους υπολογισμούς.

```
x = sh1 + 10;
```

Αν όμως είναι κλήση συνάρτησης, τότε το αν θα προσφέρει παραλληλία εξαρτάται από το μέγεθος των υπολογισμών της συνάρτησης. Παρ' όλα αυτά, γίνεται πάντα αντικατάσταση του όρου με μία προσωρινή μεταβλητή γιατί κατά κανόνα υπάρχει όφελος από την αντικατάσταση μίας συνάρτησης.

```
x = foo(i) + sh1;

t0 = foo(i);
x = t0 + sh1;
```

Η παραπάνω διαδικασία εκτελείται αναδρομικά, ξεκινώντας από την αρχική αριθμητική έκφραση και συνεχίζοντας με τους όρους που προκύπτουν μέχρι να καταλήξει σε έναν παράγοντα (δηλαδή να μη διασπάται). Κάθε όρος διασπάται επίσης σε παράγοντες, ακολουθώντας αντίστοιχη διαδικασία, σύμφωνα με την προτεραιότητα των αριθμητικών τελεστών. Ο μεταφραστής *Ariadne* υποστηρίζει τη διάσπαση αριθμητικών εκφράσεων για τους αριθμητικούς τελεστές '+', '-', '*' και '/'. Όταν ένα όρος αποτελείται από μία σειρά παραγόντων και τουλάχιστον δύο από αυτούς, συνδέονται με την πράξη της διαίρεσης, η διάσπαση του όρου γίνεται μόνο αν η διαίρεση δεν είναι ακέραια. Η εξακρίβωση γίνεται ελέγχοντας τους τύπους του διαιρέτη και του διαιρετέου.

Ο τύπος των προσωρινών μεταβλητών που χρησιμοποιούνται κατά τη διάσπαση, εξαρτάται από τον τύπο των μεταβλητών, των συναρτήσεων και των σταθερών που εμπλέκονται στην αριθμητική έκφραση που υπολογίζουν.

6.3.8 Εισαγωγή των Κλήσεων Πρόσβασης στις Παραμέτρους

Οι κλήσεις sl_getp() και sl_setp() της γλώσσας SL, υποδεικνύουν εξαρτήσεις μεταξύ των νημάτων μίας οικογένειας. Υπάρχει ανάγκη επικοινωνίας μεταξύ των νημάτων της οικογένειας και θα πρέπει να πραγματοποιηθεί με τέτοιο τρόπο ώστε να μεγιστοποιήσουμε το

βαθμό παραλληλισμού. Οι κλήσεις αυτές εμφανίζονται σε ζεύγη, με την έννοια ότι για κάθε κοινόχρηστη παράμετρο, πρώτα καλείται η sl_getp() και στη συνέχεια η sl_setp(). Για να μεγιστοποιήσουμε το βαθμό παραλληλισμού, θα πρέπει να μειώσουμε τα τμήματα κώδικα που παρουσιάζουν εξαρτήσεις. Άρα, το τμήμα κώδικα που βρίσκεται ανάμεσα σε ένα ζεύγος τέτοιων κλήσεων, θα πρέπει να είναι όσο το δυνατόν μικρότερο.

Ο μεταφραστής *Ariadne* εισάγει μέσα από αυτόματη διαδικασία τις κλήσεις sl_getp() και sl_setp() για κάθε κοινόχρηστη παράμετρο μίας αναδρομικής συνάρτησης που μετασχηματίζει. Η εισαγωγή γίνεται έτσι ώστε να μειώσει τα τμήματα κώδικα που παρουσιάζουν εξαρτήσεις, κάτι το οποίο προϋποθέτει την εύρεση της βέλτιστης λύσης.

Στο σημείο αυτό αξίζει να σημειώσουμε ότι κάθε αναδρομική κλήση αντικαθίσταται από μία κοινόχρηστη παράμετρο. Ως εκ τούτου, μία κοινόχρηστη παράμετρος σε κάθε χρήση της επιδέχεται πάντα ανάγνωση, και ποτέ εγγραφή. Η παρατήρηση αυτή απλοποιεί σημαντικά τη διαδικασία εισαγωγής των κλήσεων όπως θα δούμε στη συνέχεια. Να σημειώσουμε επίσης, ότι μία κοινόχρηστη παράμετρος ενδέχεται να μη χρησιμοποιείται στον κώδικα της συνάρτησης, αλλά η ύπαρξή της να οφείλεται μόνο και μόνο στην ανάγκη για διάδοση του αποτελέσματος μέσω των κοινόχρηστων παραμέτρων.

Ο μεταφραστής *Ariadne* ξεκινάει με την εισαγωγή των κλήσεων της τελευταίας κοινόχρηστης παραμέτρου (αυτή με τον μεγαλύτερο δείκτη), επειδή υπάρχει πάντα, λόγω του ότι αντιστοιχεί στην αναδρομική κλήση με τη μέγιστη εξάρτηση. Εντοπίζει την πρώτη αναφορά της παραμέτρου στον κώδικα της αναδρομικής συνάρτησης και εισάγει την κλήση sl_getp() ακριβώς πριν από αυτήν. Στη συνέχεια, για όλες τις παραμέτρους σε φθίνουσα σειρά, εκτός από την πρώτη, δηλαδή την sh1, εκτελεί τα *ακόλουθα βήματα*.

Έστω sh[i] η τρέχουσα παράμετρος και sh[i-1] η αμέσως προηγούμενη. Εντοπίζει την πρώτη αναφορά για κάθε μία από αυτές και εισάγει τις κλήσεις sl_setp(sh[i], sh[i-1]) και sh[i-1] = sl_getp(sh[i-1]). Αν η αναφορά στην sh[i-1] είναι πριν από αυτήν στην sh[i], τότε εισάγει την κλήση sl_setp() πριν την αναφορά στην sh[i] και την κλήση sl_getp() πριν την αναφορά στην sh[i-1].

```
sh[i-1] = sl_getp(sh[i-1]);
/* reference to sh[i-1] */
sh[i] = sl_getp(sh[i]);
sl_setp(sh[i], sh[i-1]);
/* reference to sh[i] */
```

Αλλιώς, εισάγει την κλήση sl_setp() και ακριβώς μετά την κλήση sl_getp(), αμέσως πριν την αναφορά στην sh[i-1].

```
sh[i] = sl_getp(sh[i]);
/* reference to sh[i] */
sh[i-1] = sl_getp(sh[i-1]);
sl_setp(sh[i], sh[i-1]);
/* reference to sh[i-1] */
```

Όταν μία κοινόχρηστη παράμετρος λείπει, εντοπίζει την πρώτη αναφορά στην αμέσως προηγούμενή της που υπάρχει και ακολουθεί την ίδια διαδικασία.

Η εισαγωγή της κλήσης sl_setp() για την κοινόχρηστη παράμετρο sh1 αποτελεί ειδική περίπτωση. Στην περίπτωση αυτή, ο μεταφραστής εντοπίζει την πρώτη αναφορά στην sh1 και την shv (τοπική μεταβλητή που αντιστοιχεί στο return της αναδρομικής συνάρτησης). Αν η αναφορά στην shv είναι πριν από αυτήν στην sh1, τότε εισάγει την κλήση sl_setp(shv, sh1) πριν την πρώτη αναφορά στην sh1, αλλιώς την εισάγει αμέσως μετά την αναφορά στην shv. Αναφορά στην sh1 θα υπάρχει σίγουρα καθώς στο προηγούμενο βήμα θα έχει εισαχθεί η κλήση sh1 = sl_getp(sh1).

Όταν η αναφορά που αναζητάμε για κάποια από τις κοινόχρηστες παραμέτρους βρίσκεται εντός δομής που εκτελείται υπό συνθήκη, τότε η εισαγωγή κλήσης γίνεται αμέσως πριν ή αμέσως μετά τη συγκεκριμένη δομή, ανάλογα με το αν επρόκειτο για την sl_getp() ή την sl_setp() αντίστοιχα.

6.4 Το Πέρασμα parallel-reduction

Το πέρασμα *parallel-reduction* αποτελεί ειδική περίπτωση του *elimination* και αφορά την εξάλειψη της αναδρομής, κατανέμοντας τον φόρτο εργασίας σε έναν αριθμό νημάτων τα οποία εκτελούνται παράλληλα. Γίνεται εύκολα αντιληπτό πως όταν έχουμε αριθμούς κινητής υποδιαστολής (floating point) το αποτέλεσμα μπορεί να διαφέρει σε σχέση με αυτό που θα λαμβάναμε από την αναδρομική εκτέλεση των υπολογισμών επειδή αλλάζει η σειρά των πράξεων. Η εφαρμογή του περάσματος σε μία συνάρτηση πραγματοποιείται με την εισαγωγή της αντίστοιχης οδηγίας από τον προγραμματιστή αμέσως πριν τον ορισμό της συνάρτησης.

Όπως στο πέρασμα *elimination*, έτσι και στο *parallel-reduction*, η ύπαρξη οδηγίας από τον προγραμματιστή, δε συνεπάγεται την εφαρμογή του μετασχηματισμού από τον μεταφραστή *Ariadne*. Λόγω του ότι αποτελεί ειδική περίπτωση του *elimination*, για το *parallel-reduction* ισχύουν ακριβώς τα ίδια πράγματα. Στην παρούσα ενότητα δε θα αναφερθούμε στη μορφή που πρέπει να έχει η αναδρομική συνάρτηση, στη διαδικασία αναγνώρισης του δείκτη, στο διαχωρισμό των μπλοκ, στην εξαγωγή αρχικών τιμών και στον καθορισμό των ορίων και του βήματος, καθώς περιγράφονται αναλυτικά στην ενότητα για το πέρασμα *elimination*.

Σε αντίθεση όμως με το *elimination* το οποίο εφαρμόζεται σε οποιαδήποτε αναδρομική συνάρτηση έχει δείκτη, το πέρασμα *parallel-reduction* μπορεί να μετασχηματίσει μόνο όσες συναρτήσεις έχουν ακριβώς μία αναδρομική κλήση σε κάθε μπλοκ αναδρομής. Αυτό συμβαίνει επειδή κάθε επιπλέον κλήση δημιουργεί επιπρόσθετες εξαρτήσεις οι οποίες δεν είναι δυνατόν να αντιμετωπιστούν.

Το πέρασμα *parallel-reduction* πραγματοποιεί απαλοιφή της αναδρομής, υπολογίζοντας επαναληπτικά το αποτέλεσμα και κατανέμει τον φόρτο εργασίας σε νήματα. Κάθε νήμα υπολογίζει το δικό του αποτέλεσμα, ενώ το τελικό αποτέλεσμα υπολογίζεται αφού ολοκληρωθεί η εκτέλεση όλων των νημάτων, εφαρμόζοντας την κατάλληλη πράξη στα επιμέρους αποτελέσματα. Η reduction αριθμητική έκφραση στη γενική περίπτωση έχει τη μορφή της Σχέσης

6.1.

$$f(x) = expr1(x)\ op1\ f(x-k)\ op2\ expr2(x), k \in \mathbb{Z}^* \tag{6.1}$$

Το x είναι ο δείκτης της αναδρομικής συνάρτησης, και τα $op1$ και $op2$ είναι ένας από τους αριθμητικούς τελεστές '+', '-', '*' ή '/'. Τα $expr1(x)$ και $expr2(x)$ είναι αριθμητικές εκφράσεις οποιασδήποτε μορφής, αλλά ο προγραμματιστής οφείλει να εξασφαλίσει ότι δεν παρουσιάζουν εξαρτήσεις που δεν μπορούν να αντιμετωπιστούν με reduction. Ουσιαστικά, ο μεταφραστής *Ariadne* υποστηρίζει τη γενική μορφή όταν αυτή μπορεί να αναχθεί είτε στη μορφή της Σχέσης 6.2 είτε σε αυτήν της Σχέσης 6.3.

$$f(x) = f(x-k)\ op\ expr(x), k \in \mathbb{Z}^* \tag{6.2}$$

$$f(x) = expr(x)\ op\ f(x-k), k \in \mathbb{Z}^* \tag{6.3}$$

Ο μεταφραστής *Ariadne* υποστηρίζει τους τελεστές '+', '-', '*', '/' καθώς είναι αυτοί που παρουσιάζουν περισσότερο ενδιαφέρον και εμφανίζονται συχνότερα σε ένα πραγματικό πρόβλημα. Το μοντέλο OpenMP υποστηρίζει τους τελεστές '+' και '*' μέσω της οδηγίας reduction, καθώς και μία μορφή του τελεστή '-', αλλά δεν υποστηρίζει καθόλου τον τελεστή '/'. Για να εφαρμοστεί το πέρασμα *parallel-reduction* για την πράξη της πρόσθεσης και του πολλαπλασιασμού, αρκεί η δήλωση του αντίστοιχου τελεστή, μέσω της οδηγίας. Οι πράξεις της αφαίρεσης και της διαίρεσης παρουσιάζουν ιδιαιτερότητες, επειδή δεν τηρούν την αντιμεταθετική (commutative) και την προσεταιριστική (associative) ιδιότητα. Έτσι, στην περίπτωση εφαρμογής του περάσματος *parallel-reduction* για την πράξη της αφαίρεσης ή της διαίρεσης, απαιτείται επιπλέον πληροφορία προς τον μεταφραστή η οποία θα του υποδεικνύει σε ποια μορφή βρίσκεται η reduction αριθμητική έκφραση. Η υπόδειξη αυτή γίνεται εύκολα προσθέτοντας τον τελεστή '~' (tilde) αμέσως μετά τον αντίστοιχο αριθμητικό τελεστή. Όταν η αριθμητική έκφραση έχει τη μορφή της Σχέσης 6.3 απαιτείται ο τελεστής, ενώ όταν έχει τη μορφή της Σχέσης 6.2 δεν απαιτείται.

Κάθε νήμα υπολογίζει το δικό του αποτέλεσμα σε μία μεταβλητή, με βάση τις επαναλήψεις που του έχουν ανατεθεί. Η μεταβλητή αυτή αρχικοποιείται σε διαφορετική τιμή ανάλογα με τον τελεστή που καθορίζεται μέσω της οδηγίας *parallel-reduction*. Για την πράξη της πρόσθεσης και της αφαίρεσης, η μεταβλητή αρχικοποιείται σε 0. ενώ για την πράξη του πολλαπλασιασμού και της διαίρεσης σε 1. Το τελικό αποτέλεσμα λαμβάνει υπόψιν του τα επιμέρους αποτελέσματα όλων των νημάτων και υπολογίζεται με την πράξη της πρόσθεσης όταν ο αριθμητικός τελεστής είναι '+' ή '-' και με την πράξη του πολλαπλασιασμού όταν ο αριθμητικός τελεστής είναι '*' ή '/'.

Στην περίπτωση της αφαίρεσης και της διαίρεσης όπου η reduction αριθμητική έκφραση είναι στη μορφή της Σχέσης 6.3, ο υπολογισμός του τελικού αποτελέσματος παρουσιάζει ιδιαίτερο ενδιαφέρον, καθώς δεν είναι πάντα ο ίδιος, αλλά εξαρτάται από τις επαναλήψεις που κάθε νήμα εκτελεί. Αυτός είναι και ο λόγος που ο μεταφραστής *Ariadne* χρειάζεται υπόδειξη για τη μορφή στην οποία βρίσκεται η reduction αριθμητική έκφραση.

6.4.1 Ο Αριθμητικός Τελεστής της Αφαίρεσης

Στην υποενότητα αυτή θα περιγράψουμε λεπτομερώς πως υπολογίζεται το τελικό αποτέλεσμα μίας reduction αριθμητικής έκφρασης όταν βρίσκεται σε κάθε μία από τις δύο μορφές που αναφέραμε προηγουμένως. Στο σημείο αυτό αξίζει να υπενθυμίσουμε ότι το x είναι ο δείκτης της αναδρομικής συνάρτησης και λαμβάνει μόνο ακέραιες τιμές, ενώ η σταθερά $k \in \mathbb{Z}^+$ είναι η εξάρτηση της αναδρομικής κλήσης. Για λόγους διευκόλυνσης αλλά χωρίς βλάβη της γενικότητας, θα θεωρήσουμε ότι $k = 1$. Όταν η reduction αριθμητική έκφραση εντάσσεται στη μορφή της Σχέσης 6.2, το τελικό αποτέλεσμα μπορεί να γραφεί σύμφωνα με τον μη αναδρομικό τύπο της Σχέσης 6.4.

$$\begin{aligned}
f(x) &= f(x-1) - expr(x) \\
&= (f(x-2) - expr(x-1)) - expr(x) \\
&= \dots \\
&= ((\dots((\dots((f(0) - expr(1)) - expr(2)) - \dots \\
&\quad \cdots - expr(x-s+1)) - expr(x-s+2)) - \dots \\
&\quad \cdots - expr(x-2)) - expr(x-1)) - expr(x) \\
&= f(0) - expr(1) - expr(2) - \dots \\
&\quad \cdots - expr(x-s+1) - expr(x-s+2) - \dots \\
&\quad \cdots - expr(x-2) - expr(x-1) - expr(x) \\
&= f(0) - \sum_{i=1}^{x} expr(i) \qquad (6.4)
\end{aligned}$$

Σύμφωνα με τη Σχέση 6.4 το τελικό αποτέλεσμα προκύπτει πάντα προσθέτοντας αρχικά τα επιμέρους αποτελέσματα όλων των νημάτων και αφαιρώντας τα στη συνέχεια από την αρχική τιμή.

$$\begin{aligned}
f(x) &= expr(x) - f(x-1) \\
&= expr(x) - (expr(x-1) - f(x-2)) \\
&= \dots \\
&= expr(x) - (expr(x-1) - (expr(x-2) - \dots \\
&\quad \cdots - (expr(x-s+2) - (expr(x-s+1) - \dots \\
&\quad \cdots - (expr(2) - (expr(1) - f(0)))\dots))\dots)) \\
&= expr(x) - expr(x-1) + expr(x-2) - \dots \\
&\quad \cdots + (-1)^{s-2} expr(x-s+2) + (-1)^{s-1} expr(x-s+1) + \dots \\
&\quad \cdots + (-1)^{x-2} expr(2) + (-1)^{x-1} expr(1) + (-1)^x f(0) \\
&= (-1)^x f(0) + \sum_{i=1}^{x} (-1)^{x-i} expr(i) \qquad (6.5)
\end{aligned}$$

Στη Σχέση 6.5 παρουσιάζεται ο μη αναδρομικός τύπος για τον υπολογισμό του τελικού αποτελέσματος, όταν η reduction αριθμητική έκφραση βρίσκεται στη μορφή της Σχέσης 6.3. Σε αντίθεση με τη Σχέση 6.4, παρατηρούμε ότι σε αυτήν την περίπτωση το τελικό αποτέλεσμα δεν υπολογίζεται πάντα με τον ίδιο τρόπο, αλλά εξαρτάται από το πλήθος των επαναλήψεων που ανατέθηκαν σε κάθε νήμα. Πιο συγκεκριμένα, το αποτέλεσμα του τελευταίου νήματος προστίθεται πάντα. Το αποτέλεσμα του προτελευταίου νήματος, προστίθεται όταν ο αριθμός επαναλήψεων που ανέλαβε το τελευταίο νήμα είναι άρτιος, ενώ αφαιρείται όταν είναι περιττός. Το ίδιο ισχύει και για τα άλλα νήματα, όπου για παράδειγμα το αποτέλεσμα του τρίτου νήματος από το τέλος προστίθεται όταν το άθροισμα των επαναλήψεων των δύο επόμενών του, είναι άρτιος αριθμός, ενώ αφαιρείται όταν είναι περιττός. Μπορούμε λοιπόν να κατανέμουμε τον φόρτο εργασίας μεταξύ ενός αριθμού νημάτων, αλλά θα πρέπει να γνωρίζουμε τον αριθμό επαναλήψεων που ανέλαβε κάθε ένα από αυτά.

Στη συνέχεια δίνεται μία πιο τυπική περιγραφή όσων αναφέρθηκαν παραπάνω, με τον ορισμό δύο συνόλων δεικτών. Το πρώτο σύνολο είναι το $I = (I_1, ... I_t)$, όπου t είναι ο αριθμός των νημάτων και το I_i ισούται με 0 όταν οι επαναλήψεις που εκτελεί το νήμα i είναι άρτιος αριθμός, και με 1 όταν είναι περιττός αριθμός. Το δεύτερο σύνολο είναι το $R = (R_1, ..., R_t)$, όπου το R_i ισούται με 0 όταν το επιμέρους αποτέλεσμα του νήματος i πρέπει να προστεθεί στο τελικό αποτέλεσμα και με 1 όταν πρέπει να αφαιρεθεί. Σύμφωνα με τη Σχέση 6.5 το επιμέρους αποτέλεσμα του τελευταίου νήματος προστίθεται πάντα, άρα $R_t = 0$. Οι υπόλοιπες τιμές του συνόλου δεικτών R υπολογίζονται από τη Σχέση 6.6.

$$R_i = (R_{i+1} + I_{i+1}) \ mod \ 2, 0 \leq i < t \tag{6.6}$$

Παράδειγμα

Στη συνέχεια γίνεται παρουσίαση δύο τρόπων για τον υπολογισμό της Σχέσης 6.7, κατανέμοντας τον φόρτο εργασίας με διαφορετικό τρόπο μεταξύ των νημάτων.

$$f(x) = x \ - \ f(x-1), \text{ όπου } f(0) = 0 \tag{6.7}$$

Συγκεκριμένα υπολογίζει την τιμή $f(7)$. Ο αναδρομικός υπολογισμός φαίνεται στο πλαίσιο που ακολουθεί.

f(1) = 1 - f(0) = 1 - 0 = 1
f(2) = 2 - f(1) = 2 - 1 = 1
f(3) = 3 - f(2) = 3 - 1 = 2
f(4) = 4 - f(3) = 4 - 2 = 2
f(5) = 5 - f(4) = 5 - 2 = 3
f(6) = 6 - f(5) = 6 - 3 = 3
f(7) = 7 - f(6) = 7 - 3 = 4

Όταν ο υπολογισμός της τιμής γίνεται επαναληπτικά κατανέμοντας τις επαναλήψεις σε τρία νήματα, όπου το πρώτο αναλαμβάνει τις δύο πρώτες επαναλήψεις, το δεύτερο τις τρεις

επόμενες και το τρίτο τις δύο τελευταίες, τότε η διαδικασία είναι αυτή που περιγράφεται παρακάτω.

Πρώτο Νήμα (2 επαναλήψεις)

- Επανάληψη: 1, $f(1) = 1 - f(0) = 1 - 0 = 1$
- Επανάληψη: 2, $f(2) = 2 - f(1) = 2 - 1 = 1$

Δεύτερο Νήμα (3 επαναλήψεις)

- Επανάληψη: 3, $f(3) = 3 - 0 = 3 - 0 = 3$
- Επανάληψη: 4, $f(4) = 4 - f(3) = 4 - 3 = 1$
- Επανάληψη: 5, $f(5) = 5 - f(4) = 5 - 1 = 4$

Τρίτο Νήμα (2 επαναλήψεις)

- Επανάληψη: 6, $f(6) = 6 - 0 = 6 - 0 = 6$
- Επανάληψη: 7, $f(7) = 7 - f(6) = 7 - 6 = 1$

Εφόσον κάθε νήμα έχει υπολογίσει το δικό του αποτέλεσμα, μπορούμε να υπολογίσουμε τα σύνολα δεικτών I και R, ώστε να υπολογίσουμε το τελικό αποτέλεσμα. Ο υπολογισμός των συνόλων γίνεται σύμφωνα με τον αριθμό των επαναλήψεων που πραγματοποίησε κάθε νήμα και τη Σχέση 6.6.

$$\text{Αριθμός Επαναλήψεων} = (2, 3, 2)$$

$$I = (0, 1, 0)$$

$$R = (1, 0, 0)$$

$$\text{Τελικό Αποτέλεσμα} = -1 + 4 + 1 = 4$$

Όταν χρησιμοποιούμε τον ίδιο αριθμό νημάτων, αλλά κατανέμουμε τον φόρτο εργασίας με διαφορετικό τρόπο, και συγκεκριμένα αναθέτοντας τις ίδιες επαναλήψεις στο πρώτο νήμα, αλλά αντιστρέφοντας τις επαναλήψεις μεταξύ του δεύτερου και του τρίτου νήματος, ο τρόπος υπολογισμού του τελικού αποτελέσματος διαφέρει καθώς το πλήθος επαναλήψεων των δύο αυτών νημάτων μεταβάλλεται από άρτιο σε περιττό και αντίστροφα. Η διαδικασία περιγράφεται στη συνέχεια.

Πρώτο Νήμα (2 επαναλήψεις)

- Επανάληψη: 1, $f(1) = 1 - f(0) = 1 - 0 = 1$
- Επανάληψη: 2, $f(2) = 2 - f(1) = 2 - 1 = 1$

Δεύτερο Νήμα (2 επαναλήψεις)

- Επανάληψη: 3, $f(3) = 3 - 0 = 3 - 0 = 3$
- Επανάληψη: 4, $f(4) = 4 - f(3) = 4 - 3 = 1$

Τρίτο Νήμα (3 επαναλήψεις)

- Επανάληψη: 5, $f(5) = 5 - 0 = 5 - 0 = 5$
- Επανάληψη: 6, $f(6) = 6 - f(5) = 6 - 5 = 1$
- Επανάληψη: 7, $f(7) = 7 - f(6) = 7 - 1 = 6$

Σε αυτήν την περίπτωση η διαφορά έγκειται στο γεγονός ότι προσθέτουμε το αποτέλεσμα του τρίτου νήματος και αφαιρούμε αυτό του δεύτερου, σε αντίθεση με πριν όπου κάναμε το ακριβώς ανάποδο. Αυτό συμβαίνει επειδή κάναμε εναλλαγή στο πλήθος επαναλήψεων μεταξύ του δεύτερου και του τρίτου νήματος. Από αυτήν την εναλλαγή δεν επηρεάζεται η πράξη που εφαρμόζουμε στο αποτέλεσμα του πρώτου νήματος γιατί εξαρτάται από το συνολικό πλήθος επαναλήψεων του δεύτερου και του τρίτου νήματος, το οποίο παρέμεινε σταθερό.

$$\text{Αριθμός Επαναλήψεων} = (2, 2, 3)$$
$$I = (0, 0, 1)$$
$$R = (1, 1, 0)$$
$$\text{Τελικό Αποτέλεσμα} = -1 - 1 + 6 = 4$$

6.4.2 Ο Αριθμητικός Τελεστής της Διαίρεσης

Ο υπολογισμός του τελικού αποτελέσματος μίας reduction αριθμητικής έκφρασης για την πράξη της διαίρεσης, ακολουθεί την ίδια ιδέα με αυτόν της αφαίρεσης που έχουμε ήδη περιγράψει.

$$\begin{aligned}
f(x) &= f(x-1) \;/\; expr(x) \\
&= (f(x-2) \;/\; expr(x-1)) \;/\; expr(x) \\
&= \ldots \\
&= ((\ldots((\ldots((f(0) \;/\; expr(1)) \;/\; expr(2)) \;/\; \ldots \\
&\quad \ldots \;/\; expr(x-s+1)) \;/\; expr(x-s+2)) \;/\; \ldots \\
&\quad \ldots \;/\; expr(x-2)) \;/\; expr(x-1)) \;/\; expr(x) \\
&= f(0) \;/\; expr(1) \;/\; expr(2) \;/\; \ldots \\
&\quad \ldots \;/\; expr(x-s+1) \;/\; expr(x-s+2) \;/\; \ldots \\
&\quad \ldots \;/\; expr(x-2) \;/\; expr(x-1) \;/\; expr(x) \\
&= f(0) \;/\; \prod_{i=1}^{x} expr(i) \qquad (6.8)
\end{aligned}$$

Σύμφωνα με τη Σχέση 6.8 το τελικό αποτέλεσμα προκύπτει πάντα πολλαπλασιάζοντας αρχικά τα επιμέρους αποτελέσματα όλων των νημάτων και διαιρώντας στη συνέχεια την αρχική τιμή με το γινόμενό τους.

$$
\begin{aligned}
f(x) &= expr(x) \;/\; f(x-1) \\
&= expr(x) \;/\; (expr(x-1) \;/\; f(x-2)) \\
&= \ldots \\
&= expr(x) \;/\; (expr(x-1) \;/\; (expr(x-2) \;/\; \ldots \\
&\quad \ldots \;/\; (expr(x-s+2) \;/\; (expr(x-s+1) \;/\; \ldots \\
&\quad \ldots \;/\; (expr(2) \;/\; (expr(1) \;/\; f(0))) \ldots)) \ldots)) \\
&= \frac{\prod_{i=1,x \bmod 2=0}^{1} f(0) \prod_{i=1,(x-i) \bmod 2=0}^{x} expr(i)}{\prod_{i=1,x \bmod 2=1}^{1} f(0) \prod_{i=1,(x-i) \bmod 2=1}^{x} expr(i)}
\end{aligned}
\tag{6.9}
$$

Σύμφωνα με τη Σχέση 6.9 στην οποία παρουσιάζεται ο μη αναδρομικός τύπος, όταν η reduction αριθμητική έκφραση βρίσκεται στη μορφή της Σχέσης 6.3. ο τρόπος υπολογισμού του τελικού αποτελέσματος εξαρτάται από το πλήθος των επαναλήψεων που ανατέθηκαν σε κάθε νήμα. Το αποτέλεσμα του τελευταίου νήματος πολλαπλασιάζεται πάντα ενώ το αποτέλεσμα των άλλων νημάτων είτε πολλαπλασιάζεται είτε διαιρείται ανάλογα με το αν το πλήθος των επαναλήψεων που ανατέθηκαν στα επόμενά του νήματα είναι άρτιο ή περιττό αντίστοιχα. Άρα, όπως για την πράξη της αφαίρεσης, έτσι και για την πράξη της διαίρεσης μπορούμε να κατανέμουμε τον φόρτο εργασίας με την προϋπόθεση ότι γνωρίζουμε τον αριθμό επαναλήψεων που ανέλαβε κάθε νήμα. Είναι απαραίτητα λοιπόν και στην περίπτωση της διαίρεσης τα σύνολα δεικτών I και R τα οποία αναλύσαμε κατά την περιγραφή της αφαίρεσης.

6.4.3 Απεικόνιση στο Μοντέλο SVP

Η παραλληλοποίηση μίας αναδρομικής συνάρτησης απαλείφοντας την αναδρομή και κατανέμοντας τον φόρτο εργασίας σε μία σειρά από νήματα συμβάλλει σημαντικά στη βελτίωση της απόδοσης. Ο μεταφραστής *Ariadne* παρέχει στον προγραμματιστή τη δυνατότητα αυτή μέσω του περάσματος *parallel-reduction*. Η εφαρμογή του περάσματος *parallel-reduction* και η απεικόνιση του κώδικα στο μοντέλο SVP παράγοντας κώδικα για τη γλώσσα SL, γίνεται με την εισαγωγή της παρακάτω οδηγίας πριν τον ορισμό της αντίστοιχης συνάρτησης.

#pragma ariadne parallel-reduction(op[~]) svp threads(N)

Τα βήματα που ακολουθεί ο μεταφραστής *Ariadne* κατά την εφαρμογή του περάσματος *parallel-reduction* είναι παρόμοια με αυτά για το πέρασμα *elimination*. Επομένως, θα αναφερθούμε μόνο στις διαφορές των δύο περασμάτων.

Στη συνέχεια θα περιγράψουμε πως πραγματοποιείται η απεικόνιση μίας συγκεκριμένης αναδρομικής συνάρτησης στο μοντέλο SVP. Θα χρησιμοποιήσουμε τη συνάρτηση από τον Πηγαίο Κώδικα 6.5 η οποία προσεγγίζει το ημίτονο του x. Προφανώς η συνάρτηση αυτή

προσεγγίζει το ημίτονο υπολογίζοντας ένα πεπερασμένο πλήθος όρων το οποίο αποτελεί παράμετρο της συνάρτησης. Η έξοδος του μεταφραστή *Ariadne* φαίνεται στους Πηγαίους Κώδικες 6.6, 6.7 και 6.8. Η μαθηματική σχέση για τον υπολογισμό του ημιτόνου αποτυπώνεται με τη Σχέση 6.10.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, \forall x \tag{6.10}$$

Στην οδηγία που δίνεται πριν τον ορισμό της συνάρτησης, καθορίζεται ο αριθμητικός τελεστής της πρόσθεσης '+' και ο αριθμός των νημάτων σε έξι.

```
#pragma ariadne parallel-reduction(+) svp threads(6)
```

```
1  double sin_series(double x, int terms)
2  {
3      if (terms == 0)
4      {
5          return (x);
6      }
7      else
8      {
9          return (sin_series(x, terms - 1) + power(-1, terms)
10                                          * power(x, 2 * terms + 1)
11                                          / (double) factorial(2 * terms + 1));
12     }
13 }
```

Πηγαίος Κώδικας 6.5: Αναδρομική συνάρτηση για την προσέγγιση του $\sin(x)$.

```
1  sl_def(sin_series_t_n , void, sl_glfparm(double, x), sl_shfparm(double, shl))
2  {
3      double t0;
4      double x;
5      double shv, shl;
6
7      sl_index(terms);
8
9      x = sl_getp(x);
10
11     t0 = power(-1, terms) * power(x, 2 * terms + 1) / (double) factorial(2 * terms + 1);
12     shl = sl_getp(shl);
13     shv = t0 + shl;
14     sl_setp(shl, shv);
15 }
16 sl_enddef
```

Πηγαίος Κώδικας 6.6: Συνάρτηση που εκτελείται από τις οικογένειες νημάτων που δημιουργούνται κατά την απεικόνιση του κώδικα στο μοντέλο SVP μέσω του περάσματος *parallel-reduction*.

```
1  sl_def(sin_series_m_n, void, sl_glfparm(double, x), sl_glparm(double *, sh),
2                               sl_glparm(int, terms), sl_glparm(int, pos),
3                               sl_glparm(int, thread_iter), sl_glparm(int, total_threads))
4  {
5      double x;
6      double *sh;
7      int lower, upper, terms, pos;
8      int thread_iter, total_threads;
9
10     sl_index(i);
11
12     x = sl_getp(x);
13     sh = sl_getp(sh);
14     terms = sl_getp(terms);
15     pos = sl_getp(pos);
16     thread_iter = sl_getp(thread_iter);
17     total_threads = sl_getp(total_threads);
18
19     lower = i * 1 * thread_iter + pos;
20     upper = (i != total_threads - 1) ? lower + 1 * thread_iter : terms + 1;
21
22     {
23         sl_create(, , lower, upper, 1, , , sin_series_t_n, sl_glfarg(double, x, x),
24                                                            sl_shfarg(double, sh1, 0));
25         sl_sync();
26
27         sh[i] = sl_geta(sh1);
28     }
29 }
30 sl_enddef
```

Πηγαίος Κώδικας 6.7: Συνάρτηση που εκτελείται από μία οικογένεια νημάτων για να δημιουργήσει τις οικογένειες στις οποίες κατανέμεται ο φόρτος εργασίας κατά την απεικόνιση του κώδικα στο μοντέλο SVP μέσω του περάσματος *parallel-reduction*.

Το πέρασμα *parallel-reduction* καταργεί τις αναδρομικές κλήσεις της συνάρτησης και δημιουργεί τόσες οικογένειες νημάτων όσες ορίζονται από τον προγραμματιστή μέσω της οδηγίας. Κάθε οικογένεια νημάτων αναλαμβάνει ένα τμήμα του φόρτου εργασίας, ενώ κάθε επανάληψη εκτελείται από ένα νήμα της οικογένειας.

Στο μπλοκ της κύριας δομής ελέγχου στο οποίο βρίσκονται οι αναδρομικές κλήσεις, αφού τις καταργήσει, εισάγει κώδικα ο οποίος υπολογίζει τα όρια των έξι οικογενειών που θα δημιουργηθούν. Στη συνέχεια δημιουργεί μία οικογένεια νημάτων με έξι νήματα η οποία είναι υπεύθυνη για τη δημιουργία των έξι οικογενειών που θα αναλάβουν τον φόρτο εργασίας. Αμέσως μετά την ολοκλήρωση της οικογένειας, υπολογίζει το τελικό αποτέλεσμα, το οποίο και επιστρέφει. Το παραπάνω τμήμα κώδικα παρουσιάζεται στον Πηγαίο Κώδικα 6.8.

Στον Πηγαίο Κώδικα 6.7, η συνάρτηση νημάτων είναι αρμόδια για τη δημιουργία μίας οικογένειας νημάτων. Οι υπολογισμοί που πραγματοποιεί περιορίζονται στην εύρεση των ορίων της οικογένειας που θα δημιουργήσει. Μετά την ολοκλήρωση της οικογένειας, αποθηκεύει το επιμέρους αποτέλεσμα στον πίνακα sh. Η θέση στον πίνακα καθορίζεται από το αναγνωριστικό κάθε νήματος. Οι παράμετροι της συνάρτησης αποτελούνται από τις βασικές παραμέτρους της αναδρομικής συνάρτησης, με τη διαφορά ότι υπάρχει ένα σύνολο παραμέ-

τρων μέσω των οποίων υπολογίζει τα όρια lower και upper της οικογένειας που δημιουργεί. Μία επιπλέον παράμετρο αποτελεί ο πίναχας sh, ενώ όλες οι παράμετροι είναι καθολικές.

```
1  double sin_series(double x, int terms)
2  {
3      if (terms == 0)
4      {
5          return (x);
6      }
7      else
8      {
9          int pos;
10         double retv, sh[6];
11         int i, total_iter, thread_iter, total_threads;
12
13         pos = 1;
14         total_iter = ((terms - pos) / 1 + 1);
15         total_threads = 6;
16
17         if (total_iter > total_threads)
18         {
19             thread_iter = total_iter / total_threads;
20         }
21         else
22         {
23             total_threads = total_iter;
24             thread_iter = 1;
25         }
26
27         {
28             sl_create(, , 0, total_threads, 1, , , sin_series_m_n,
29                       sl_glfarg(double, x, x), sl_glarg(double *, sh, sh),
30                       sl_glarg(int, terms, terms), sl_glarg(int, pos, pos),
31                       sl_glarg(int, thread_iter, thread_iter),
32                       sl_glarg(int, total_threads, total_threads));
33             sl_sync();
34         }
35
36         retv = sh[0];
37
38         for (i = 1; i < total_threads; i++)
39         {
40             retv += sh[i];
41         }
42
43         retv += x;
44
45         return (retv);
46     }
47 }
```

Πηγαίος Κώδικας 6.8: Συνάρτηση για την προσέγγιση του $sin(x)$ μετά την εφαρμογή του περάσματος *parallel-reduction* για απεικόνιση του κώδικα στο μοντέλο SVP.

Τέλος, στον Πηγαίο Κώδικα 6.6 παρουσιάζεται η συνάρτηση νημάτων που αναλαμβάνει την εκτέλεση του φόρτου εργασίας. Εκτός του παραλληλισμού που επιτυγχάνεται λόγω της παράλληλης εκτέλεσης των έξι οικογενειών, στο μοντέλο SVP επιτυγχάνεται μία άλλη μορφή παραλληλισμού, καθώς η ποσότητα που εκχωρείται στην τοπική μεταβλητή t0 μπορεί

να υπολογιστεί παράλληλα από όλα τα νήματα της ίδιας οικογένειας.

6.4.4 Απεικόνιση στο Πρότυπο Νημάτων POSIX

Η εφαρμογή του περάσματος *parallel-reduction* για απεικόνιση του κώδικα στο πρότυπο νημάτων POSIX, πραγματοποιείται με την εισαγωγή της οδηγίας που ακολουθεί, πριν τον ορισμό της αντίστοιχης συνάρτησης.

#pragma ariadne parallel-reduction(op[⁻]) posix threads(N)

Στους Πηγαίους Κώδικες 6.9 και 6.10 βρίσκεται η έξοδος της αναδρομικής συνάρτησης που προσεγγίζει το $\sin(x)$, που παρουσιάστηκε στον Πηγαίο Κώδικα 6.5. Στην οδηγία που δίνεται πριν τον ορισμό της συνάρτησης sin_series(), καθορίζεται ο αριθμητικός τελεστής της πρόσθεσης '+' και ο αριθμός των νημάτων σε έξι.

#pragma ariadne parallel-reduction(+) posix threads(6)

```
1  typedef struct sin_series_s_n
2  {
3      double x;
4      int id;
5      double sh;
6      int terms;
7      int pos;
8      int thread_iter;
9      int total_threads;
10 } sin_series_s_n;
11
12 void *sin_series_n(void *__)
13 {
14     sin_series_s_n *_ = (sin_series_s_n *) __;
15     int i, lower, upper;
16
17     lower = _->id * 1 * _->thread_iter + _->pos;
18     upper = (_->id != _->total_threads - 1) ? lower + 1 * _->thread_iter : _->terms + 1;
19     _->sh = 0;
20
21     for (i = lower; i < upper; i += 1)
22     {
23         _->sh = _->sh + power(-1, i) * power(_->x, 2 * i + 1)
24                                      / (double) factorial(2 * i + 1);
25     }
26
27     return (NULL);
28 }
```

Πηγαίος Κώδικας 6.9: Συνάρτηση που εκτελείται από τα νήματα κατά την απεικόνιση του κώδικα στο πρότυπο νημάτων POSIX μέσω του περάσματος *thread-safe* και ορισμός της δομής για τη μετάδοση της πληροφορίας.

Στον Πηγαίο Κώδικα 6.10, οι αναδρομικές κλήσεις έχουν αντικατασταθεί από κώδικα που δημιουργεί τα έξι νήματα που θα αναλάβουν τον φόρτο εργασίας. Πιο συγκεκριμένα, υπολογίζει τα όρια των έξι νημάτων και στη συνέχεια τα δημιουργεί. Αφού ολοκληρωθεί η εκτέλεσή τους, υπολογίζει το τελικό αποτέλεσμα, το οποίο και επιστρέφει. Ο κώδικας που εκτελεί κάθε νήμα είναι αυτός στον Πηγαίο Κώδικα 6.9. Κάθε νήμα εκτελεί ένα σύνολο επαναλήψεων, ανάλογα με τα όρια που του έχουν δοθεί.

```
1  double sin_series(double x, int terms)
2  {
3      if (terms == 0)
4      {
5          return (x);
6      }
7      else
8      {
9          int th[6];
10         sin_series_s_n data[6];
11         pthread_t tid[6];
12         int pos;
13         double retv;
14         int i, total_iter, thread_iter, total_threads;
15
16         pos = 1;
17         total_iter = ((terms - pos) / 1 + 1);
18         total_threads = 6;
19
20         if (total_iter > total_threads)
21         {
22             thread_iter = total_iter / total_threads;
23         }
24         else
25         {
26             total_threads = total_iter;
27             thread_iter = 1;
28         }
29
30         for (i = 0; i < total_threads; i++)
31         {
32             data[i].x = x;
33             data[i].id = i;
34             data[i].terms = terms;
35             data[i].pos = pos;
36             data[i].thread_iter = thread_iter;
37             data[i].total_threads = total_threads;
38             th[i] = pthread_create(&tid[i], NULL, sin_series_n, (void *) &data[i]);
39         }
40
41         for (i = 0; i < total_threads; i++)
42         {
43             if (th[i] != 0)
44             {
45                 sin_series_n((void *) &data[i]);
46             }
47         }
48
49         if (th[0] == 0)
50         {
51             pthread_join(tid[0], NULL);
```

```
52          }
53
54          retv = data[0].sh;
55
56          for (i = 1; i < total_threads; i++)
57          {
58              if (th[i] == 0)
59              {
60                  pthread_join(tid[i], NULL);
61              }
62
63              retv += data[i].sh;
64          }
65
66          retv += x;
67
68          return (retv);
69      }
70  }
```

Πηγαίος Κώδικας 6.10: Συνάρτηση για την προσέγγιση του $sin(x)$ μετά την εφαρμογή του περάσματος *parallel-reduction* για απεικόνιση του κώδικα στο πρότυπο νημάτων POSIX.

Επειδή το πρότυπο νημάτων POSIX δεν επιτρέπει το πέρασμα πολλών ορισμάτων σε μία συνάρτηση που θα εκτελεστεί από ένα νήμα, ορίζουμε μία δομή με τα απαραίτητα πεδία. Κάθε νήμα εκτός από τις βασικές παραμέτρους, θα πρέπει να γνωρίζει επιπλέον πληροφορίες για να μπορέσει να υπολογίσει τα όρια των επαναλήψεων που θα εκτελέσει. Θα πρέπει επίσης να υπάρχει ένα πεδίο μέσω του οποίου θα γνωρίζει το δικό του μοναδικό αναγνωριστικό και ένα ακόμα στο οποίο θα γράψει την τιμή του τελικού αποτελέσματος.

6.5 Το Πέρασμα thread-safe

Το τρίτο πέρασμα του μεταφραστή *Ariadne* είναι το *thread-safe*. Αποσκοπεί στην παραλληλοποίηση των αναδρομικών συναρτήσεων που είναι από τη φύση τους αναδρομικές και περιλαμβάνουν τουλάχιστον δύο ανεξάρτητες αναδρομικές κλήσεις οι οποίες μπορούν να εκτελεστούν ταυτόχρονα. Σε αντίθεση με τα δύο προηγούμενα περάσματα, δηλαδή το *elimination* και το *parallel-reduction*, δεν υπάρχει κάποια προϋπόθεση. όπως για παράδειγμα η ύπαρξη μίας παραμέτρου που έχει το ρόλο του δείκτη. Η ανεξαρτησία των κλήσεων αποτελεί ευθύνη του προγραμματιστή και υποδεικνύεται μέσω οδηγίας.

Ο μεταφραστής *Ariadne* εντοπίζει όλες τις ανεξάρτητες αναδρομικές κλήσεις της συνάρτησης, που βρίσκονται σε ίδιο μπλοκ της κύριας δομής ελέγχου και τις αντικαθιστά με μία τοπική μεταβλητή. Όλες οι κλήσεις εκτελούνται παράλληλα με τη δημιουργία νημάτων, και αμέσως μετά τον τερματισμό τους, γίνεται εκχώρηση του αποτελέσματος στην τοπική μεταβλητή που τις αντικατέστησε.

Η αντικατάσταση των αναδρομικών κλήσεων από νήματα δεν πραγματοποιείται πάντα. Τα νήματα δημιουργούνται μέχρι ένα συγκεκριμένο επίπεδο βάθους, ενώ μετά από αυτό το επίπεδο οι κλήσεις πραγματοποιούνται σειριακά. Αυτό συμβαίνει γιατί η συνεχής δημιουργία νημάτων όχι μόνο δε βελτιώνει, αλλά μειώνει αισθητά την απόδοση ενός παράλληλου

προγράμματος.

Το κατάλληλο επίπεδο βάθους ενδεχομένως να διαφέρει για δύο αναδρομικές συναρτήσεις, καθώς εξαρτάται από το πλήθος των υπολογισμών που πραγματοποιούνται κατά την εκτέλεσή τους. Ο μεταφραστής *Ariadne* δίνει στον προγραμματιστή τη δυνατότητα να επιλέξει ο ίδιος το επίπεδο βάθους μέχρι το οποίο επιθυμεί να δημιουργούνται τα νήματα. Ο καθορισμός γίνεται μέσω της οδηγίας με την παράμετρο level(N).

6.5.1 Απεικόνιση στο Μοντέλο SVP

Η παράλληλη εκτέλεση μίας σειράς ανεξάρτητων αναδρομικών κλήσεων μπορεί να γίνει εύκολα μέσω της αυτόματης παραλληλοποίησης που παρέχει ο μεταφραστής *Ariadne*. Η απεικόνιση του κώδικα στο μοντέλο SVP επιτυγχάνεται με την εισαγωγή της αντίστοιχης οδηγίας πριν τον ορισμό της αναδρομικής συνάρτησης.

#pragma ariadne thread-safe svp level(N)

Στη συνέχεια περιγράφουμε λεπτομερώς πως πραγματοποιείται η απεικόνιση της αναδρομικής συνάρτησης του Πηγαίου Κώδικα 6.11 στο μοντέλο SVP. Η συνάρτηση αυτή υπολογίζει προσεγγιστικά το ολοκλήρωμα του ημιτόνου στο διάστημα $[a, b]$ σύμφωνα με τη Σχέση 6.11. Οι παράμετροί της είναι τα a και b, καθώς και το d, η σημασία του οποίου φαίνεται από τη Σχέση 6.11. Ο κώδικας που παράγει ο μεταφραστής *Ariadne* μετά την εφαρμογή του περάσματος *thread-safe* παρουσιάζεται στους Πηγαίους Κώδικες 6.12 και 6.13.

$$\int_a^b \sin(x)\,\mathrm{d}x = \begin{cases} \dfrac{1}{2}\left(sin(a) + sin(b)\right)(b-a) & , \quad b-a \leq d \\ \displaystyle\int_a^{a+\frac{b-a}{2}} \sin(x)\,\mathrm{d}x + \int_{a+\frac{b-a}{2}}^b \sin(x)\,\mathrm{d}x & , \quad \text{διαφορετικά} \end{cases} \tag{6.11}$$

Στην οδηγία που εισάγεται πριν τον ορισμό της συνάρτησης, καθορίζεται το επίπεδο βάθους σε τέσσερα.

#pragma ariadne thread-safe svp level(4)

Ο κώδικας της αναδρομικής συνάρτησης αντικαθίσταται από τον Πηγαίο Κώδικα 6.13. όπου δημιουργείται μία οικογένεια νημάτων με ένα μόνο νήμα. Επειδή ο κώδικας της αναδρομικής συνάρτησης θα βρίσκεται πλέον στο σώμα μίας συνάρτησης νημάτων, θα πρέπει ακόμα και την πρώτη φορά να εκτελεστεί με τη δημιουργία μίας οικογένειας. Έτσι, ουσιαστικά επαναλαμβάνεται η πρώτη κλήση της συνάρτησης, ώστε να ξεκινήσει η εκτέλεση της συνάρτησης νημάτων. Αυτό συμβαίνει μόνο την πρώτη φορά για καθαρά προγραμματιστικούς λόγους που οφείλονται στη φύση της γλώσσας SL.

```
1  double integral(double a, double b, double d)
2  {
3      double k;
4
5      k = b - a;
6
7      if (k <= d)
8      {
9          return (((sin(a) + sin(b)) * k) / 2.0);
10     }
11     else
12     {
13         return (integral(a, a + k / 2.0, d) + integral(a + k / 2.0, b, d));
14     }
15 }
```

Πηγαίος Κώδικας 6.11: Αναδρομική συνάρτηση για την προσέγγιση του $\int_a^b \sin(x)\,dx$.

Η συνάρτηση νημάτων βρίσκεται στον Πηγαίο Κώδικα 6.12. Οι παράμετροι αποτελούνται από τις βασικές παραμέτρους της αναδρομικής συνάρτησης, έχοντας προσθέσει επιπλέον δύο παραμέτρους ειδικού σκοπού. Η παράμετρος level αντιστοιχεί στο τρέχον επίπεδο βάθους και αρχικοποιείται σε μηδέν κατά τη δημιουργία της οικογένειας νημάτων. Η άλλη παράμετρος είναι η result και είναι αρμόδια για την επιστροφή του τελικού αποτελέσματος. Οι βασικές παράμετροι της συνάρτησης αποτελούν καθολικές παραμέτρους της γλώσσας SL. Το ίδιο ισχύει και για την παράμετρο level, καθώς δε μας ενδιαφέρει η εγγραφή της παραμέτρου, παρά μόνο η ανάγνωσή της.

Η παράμετρος result αποτελεί καθολική παράμετρο, αλλά επειδή μας ενδιαφέρει η εγγραφή της, περνάμε δείκτη, έτσι ώστε το αποτέλεσμα να είναι ορατό και μετά την επιστροφή της συνάρτησης νημάτων. Μία εναλλακτική επιλογή για την παράμετρο result θα ήταν η δήλωσή της ως κοινόχρηστη παράμετρο, το οποίο θα μας παρείχε την εξ' ορισμού δυνατότητα εγγραφής σε αυτήν.

Στο μπλοκ της κύριας δομής ελέγχου στο οποίο υπήρχαν οι αναδρομικές κλήσεις έχει εισαχθεί μία δομή ελέγχου για να εξετάζει αν θα πρέπει να δημιουργηθούν νέες οικογένειες νημάτων ή αν οι κλήσεις θα πραγματοποιηθούν σειριακά. Ο έλεγχος γίνεται σύμφωνα με την τιμή της παραμέτρου level. Όταν η δημιουργία νέων οικογενειών είναι επιτρεπτή, τότε δημιουργούνται δύο νέες με ένα νήμα η κάθε μία. Και στις δύο οικογένειες νημάτων αρχικοποιούμε το όρισμα level στην τιμή που έχει η παράμετρος level του πατέρα, προσαυξημένη κατά ένα. Στη συνέχεια ακολουθούν οι κλήσεις sl_sync() που υποδεικνύουν συγχρονισμό, και εκτός της δομής ελέγχου ακολουθεί η εγγραφή του τελικού αποτελέσματος στην παράμετρο result. Ο συγχρονισμός γίνεται σε τέτοιο σημείο ώστε να μεγιστοποιείται ο βαθμός παραλληλίας. Ο μεταφραστής *Ariadne* εντοπίζει την πρώτη χρήση του αποτελέσματος και εισάγει την κλήση sl_sync() ακριβώς πριν από αυτήν.

Όταν το επίπεδο βάθους αυξηθεί σε σημείο που δεν είναι πλέον επιτρεπτή η παράλληλη εκτέλεση των αναδρομικών κλήσεων, τότε καλείται μία συνάρτηση της C η οποία είναι στην πραγματικότητα η αρχική αναδρομική συνάρτηση με διαφορετικό όνομα.

```
sl_def(integral_p, void, sl_glfparm(double, a), sl_glfparm(double, b),
                         sl_glfparm(double, d), sl_glparm(int, level),
                                                sl_glparm(double *, result))
{
    double a;
    double b;
    double d;
    double *result;
    double k;

    a = sl_getp(a);
    b = sl_getp(b);
    d = sl_getp(d);
    result = sl_getp(result);

    k = b - a;

    if (k <= d)
    {
        *result = ((sin(a) + sin(b)) * k) / 2.0;
    }
    else
    {
        int level;
        double result1;
        double result2;

        level = sl_getp(level);

        if (level < 4)
        {
            sl_create(, , 1, 2, 1, , , integral_p, sl_glfarg(double, a2, a + k / 2.0),
                                                   sl_glfarg(double, b2, b),
                                                   sl_glfarg(double, d2, d),
                                                   sl_glarg(int, level2, level + 1),
                                                   sl_glarg(double *, result2, &result2));
            sl_create(, , 1, 2, 1, , , integral_p, sl_glfarg(double, a1, a),
                                                   sl_glfarg(double, b1, a + k / 2.0),
                                                   sl_glfarg(double, d1, d),
                                                   sl_glarg(int, level1, level + 1),
                                                   sl_glarg(double *, result1, &result1));

            sl_sync();
            sl_sync();
        }
        else
        {
            result1 = integral_r(a, a + k / 2.0, d);
            result2 = integral_r(a + k / 2.0, b, d);
        }

        *result = result1 + result2;
    }
}
sl_enddef
```

Πηγαίος Κώδικας 6.12: Συνάρτηση νημάτων με το σώμα της αρχικής αναδρομικής συνάρτησης για την προσέγγιση του $\int_a^b \sin(x)\,dx$, μετά την απεικόνιση του κώδικα στο μοντέλο SVP με το πέρασμα *thread-safe*.

```
1  double integral(double a, double b, double d)
2  {
3      double result;
4
5      sl_create(, , 1, 2, 1, , , integral_p, sl_glfarg(double, a, a),
6                               sl_glfarg(double, b, b), sl_glfarg(double, d, d),
7                               sl_glarg(int, level, 0), sl_glarg(double *, result, &result));
8      sl_sync();
9
10     return (result);
11 }
```

Πηγαίος Κώδικας 6.13: Η αρχική αναδρομική συνάρτηση για την προσέγγιση του $\int_a^b \sin(x)\,dx$ μετά την αντικατάσταση του σώματός της, κατά την εφαρμογή του περάσματος *thread-safe* για απεικόνιση του κώδικα στο μοντέλο SVP.

6.5.2 Απεικόνιση στο Πρότυπο Νημάτων POSIX

Για την εφαρμογή του περάσματος *thread-safe* και την παραγωγή κώδικα για το πρότυπο νημάτων POSIX απαιτείται η παρακάτω οδηγία.

#pragma ariadne thread-safe posix level(N)

Στους Πηγαίους Κώδικες 6.14 και 6.15 βρίσκεται η έξοδος της αναδρομικής συνάρτησης που υπολογίζει προσεγγιστικά το ολοκλήρωμα του ημιτόνου $\int_a^b \sin(x)\,dx$ στο διάστημα $[a, b]$, που παρουσιάστηκε στον Πηγαίο Κώδικα 6.11. Στην οδηγία που δίνεται πριν τον ορισμό της συνάρτησης, καθορίζεται το επίπεδο βάθους σε τέσσερα.

#pragma ariadne thread-safe posix level(4)

Στον Πηγαίο Κώδικα 6.15, το σώμα της αρχικής αναδρομικής συνάρτησης έχει αντικατασταθεί από την κλήση της συνάρτησης που εκτελείται από τα νήματα του προτύπου POSIX. Πριν την κλήση γίνεται αρχικοποίηση των πεδίων της δομής που δίνεται ως όρισμα στη συνάρτηση, ενώ αμέσως μετά την κλήση γίνεται επιστροφή του πεδίου result στο οποίο έχει αποθηκευτεί το τελικό αποτέλεσμα.

Η δομή είναι απαραίτητη καθώς η συνάρτηση που εκτελεί ένα νήμα του προτύπου POSIX δεν μπορεί να λάβει περισσότερες από μία παραμέτρους. Επομένως, όλα τα απαραίτητα πεδία ομαδοποιούνται μέσω της δομής. Τα πεδία από τα οποία αποτελείται η δομή είναι οι βασικές παράμετροι της αναδρομικής συνάρτησης, μαζί με τα πεδία level και result ο ρόλος των οποίων έχει ήδη περιγραφεί προηγουμένως.

Στο σώμα της συνάρτησης που εκτελούν τα νήματα, δηλαδή στον Πηγαίο Κώδικα 6.14, οι αλλαγές είναι παρόμοιες με αυτές που εφαρμόζονται για την απεικόνιση του κώδικα στο μοντέλο SVP.

```
1  typedef struct integral_s
2  {
3      double a;
4      double b;
5      double d;
6      int level;
7      double result;
8  } integral_s;
9
10 void *integral_p(void *__)
11 {
12     integral_s *_ = (integral_s *) __;
13     double k;
14
15     k = _->b - _->a;
16
17     if (k <= _->d)
18     {
19         _->result = ((sin(_->a) + sin(_->b)) * k) / 2.0;
20         return (NULL);
21     }
22     else
23     {
24         integral_s call1, call2;
25         pthread_t tid1, tid2;
26         int th1, th2;
27
28         call1.a = _->a;
29         call1.b = _->a + k / 2.0;
30         call1.d = _->d;
31         call1.level = _->level + 1;
32
33         call2.a = _->a + k / 2.0;
34         call2.b = _->b;
35         call2.d = _->d;
36         call2.level = _->level + 1;
37
38         if (_->level < 4)
39         {
40             th1 = pthread_create(&tid1, NULL, integral_p, (void *) &call1);
41             th2 = pthread_create(&tid2, NULL, integral_p, (void *) &call2);
42
43             if (th1 != 0)
44             {
45                 integral_p((void *) &call1);
46             }
47
48             if (th2 != 0)
49             {
50                 integral_p((void *) &call2);
51             }
52         }
53         else
54         {
55             integral_p((void *) &call1);
56             integral_p((void *) &call2);
57         }
58
59         if (_->level < 4 && th1 == 0)
60         {
61             pthread_join(tid1, NULL);
62         }
```

```
63
64          if (_->level < 4 && th2 == 0)
65          {
66              pthread_join(tid2, NULL);
67          }
68
69          _->result = call1.result + call2.result;
70          return (NULL);
71      }
72  }
```

Πηγαίος Κώδικας 6.14: Συνάρτηση που εκτελείται από τα νήματα για την προσέγγιση του $\int_a^b \sin(x)\,dx$ κατά την απεικόνιση του κώδικα στο πρότυπο νημάτων POSIX μέσω του περάσματος *thread-safe* και ορισμός της δομής για τη μετάδοση της πληροφορίας.

Γίνεται εισαγωγή μίας δομής ελέγχου μέσω της οποία εξετάζεται κατά πόσο είναι επιτρεπτή η δημιουργία νέων νημάτων. Ο έλεγχος γίνεται σύμφωνα με την τιμή της μεταβλητής level που ανήκει στη δομή που ορίσαμε. Όταν επιτρέπεται η δημιουργία νημάτων, δημιουργούνται δύο νήματα, ένα για κάθε αναδρομική κλήση που αντικαθιστά. Στη συνέχεια γίνεται έλεγχος για το αν δημιουργήθηκαν όλα τα νήματα, καθώς σε περίπτωση αποτυχίας, η αντίστοιχη κλήση θα πραγματοποιηθεί σειριακά. Εκτός του μπλοκ της δομής ελέγχου γίνεται έλεγχος σχετικά με την επιτυχία δημιουργίας όλων των νημάτων, και για κάθε ένα από αυτά τηρείται αναμονή μέχρι να τερματίσει. Αυτό συμβαίνει ακριβώς πριν την πρώτη χρήση του αποτελέσματος που υπολογίζει, με στόχο τη μεγιστοποίηση του βαθμού παραλληλίας μεταξύ των ανεξάρτητων αναδρομικών κλήσεων. Αν η δημιουργία των νημάτων δεν επιτρέπεται, τότε η σειριακή εκτέλεση των αναδρομικών κλήσεων πραγματοποιείται εξ' αρχής. Τέλος, το τελικό αποτέλεσμα εγγράφεται στο πεδίο result της δομής.

```
1  double integral(double a, double b, double d)
2  {
3      integral_s _;
4
5      _.a = a;
6      _.b = b;
7      _.d = d;
8      _.level = 0;
9
10     integral_p((void *) &_);
11
12     return (_.result);
13 }
```

Πηγαίος Κώδικας 6.15: Η αρχική αναδρομική συνάρτηση για την προσέγγιση του $\int_a^b \sin(x)\,dx$ μετά την αντικατάσταση του σώματός της, κατά την εφαρμογή του περάσματος *thread-safe* για απεικόνιση του κώδικα στο πρότυπο νημάτων POSIX.

Κεφαλαιο 7

Υλοποιηση του Μεταφραστη Ariadne

7.1 Εισαγωγή

Ο μεταφραστής *Ariadne* είναι υλοποιημένος σε C++ και εκμεταλλεύεται τα αντικειμενοστρεφή στοιχεία της γλώσσας με στόχο την καλύτερη αναπαράσταση του πηγαίου κώδικα. Η οργάνωση του κώδικα πραγματοποιείται σε κλάσεις (class), η σημασία των οποίων σχετίζεται με μία συγκεκριμένη αρμοδιότητα κατά τη μετάφραση. Οι κλάσεις που απαρτίζουν τον μεταφραστή *Ariadne* παρουσιάζονται στον Πίνακα 7.1, μαζί με τα υπόλοιπα αρχεία του πηγαίου κώδικα. Κάθε κλάση βρίσκεται σε διαφορετικό ζεύγος αρχείου κεφαλίδων (.hpp) και πηγαίου κώδικα (.cpp). Σε ένα αρχείο κεφαλίδων βρίσκεται η εισαγωγή βιβλιοθηκών συστήματος και χρήστη, ο ορισμός της κλάσης και οι οδηγίες προς τον προεπεξεργαστή (preprocessor). Στο αντίστοιχο αρχείο πηγαίου κώδικα βρίσκεται το σώμα όλων των μεθόδων της κλάσης και η εισαγωγή βιβλιοθηκών. Η συνάρτηση main() του μεταφραστή *Ariadne* είναι στο αρχείο ariadne.cpp.

Η μεταγλώττιση του μεταφραστή *Ariadne* πραγματοποιήθηκε με τον μεταφραστή g++ (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 στο λειτουργικό σύστημα Ubuntu 12.04.2 LTS. Για

την υλοποίηση του λεκτικού αναλυτή χρησιμοποιήθηκε το εργαλείο Flex 2.5.35 (The Fast Lexical Analyzer), ενώ για την αυτόματη παραγωγή του συνταχτικού αναλυτή το εργαλείο bison (GNU Bison) 2.5.

ariadne.cpp	ariadne.ll	ariadne.yy
parserInterface.cpp	estring.cpp	
Line.cpp	TranslationUnit.cpp	
Alias.cpp	AliasAnalysis.cpp	
Entity.cpp	Scope.cpp	SymbolTable.cpp
Parameter.cpp	ParameterList.cpp	
Initiator.cpp	InitiatorList.cpp	
TypedefArrayType.cpp	TypedefArrayTypeList.cpp	
MainFunctionPass.cpp	RecursiveFunctionPass.cpp	RecursiveFunction.cpp
Elimination.cpp	ParallelReduction.cpp	ThreadSafe.cpp

Πίνακας 7.1: Τα αρχεία πηγαίου κώδικα του μεταφραστή *Ariadne*.

7.2 Λεκτική Ανάλυση

Ο λεκτικός αναλυτής του μεταφραστή *Ariadne* παράγεται αυτόματα με το εργαλείο Flex και η περιγραφή των λεκτικών μονάδων γίνεται στο αρχείο ariadne.ll. Οι λεκτικές μονάδες του μεταφραστή είναι αυτές της γλώσσας C κατά το πρότυπο ANSI, μαζί με μερικές ακόμα που κρίνονται απαραίτητες.

Τόσο τα include όσο και τα pragma αποτελούν οδηγίες προς τον προεπεξεργαστή και όχι προς τον μεταφραστή. Επομένως, η αναγνώρισή τους αποτελεί αρμοδιότητα του προεπεξεργαστή και μόνο. Επειδή στα πλαίσια της παρούσας διατριβής δεν υλοποιείται ο προεπεξεργαστής της C, ορίζουμε ως λεκτικές μονάδες τις λέξεις #include και #pragma οι οποίες ενδεχομένως να ακολουθούνται από οποιαδήποτε ακολουθία χαρακτήρων έως ότου βρεθεί ο χαρακτήρας αλλαγής γραμμής. Ο χαρακτήρας '#' στην αρχή των λεκτικών μονάδων είναι αναγκαίος καθώς χωρίς αυτόν, οποιαδήποτε χρήση των ονομάτων include και pragma θα θεωρούνταν ως οδηγία. Αυτό φυσικά δεν ισχύει, λόγω του ότι τα ονόματα αυτά αποτελούν έγκυρα αναγνωριστικά (identifier) της γλώσσας C. Τα token των λεκτικών μονάδων είναι τα INCLUDE_STRING και ARIADNE_PRAGMA. Στην πραγματικότητα το token ARIADNE_PRAGMA επιστρέφεται μόνο όταν μετά τη λέξη pragma ακολουθεί η λέξη ariadne, ενώ αγνοείται ολόκληρη η ακολουθία χαρακτήρων σε κάθε άλλη περίπτωση.

Ο ορισμός των δύο αυτών λεκτικών μονάδων γίνεται για να μπορεί ο μεταφραστής *Ariadne* να δεχθεί ως είσοδο ένα πρόγραμμα που περιλαμβάνει οδηγίες για την εισαγωγή βιβλιοθηκών, καθώς και για να δέχεται τις οδηγίες από τον προγραμματιστή, με βάση τις οποίες γίνεται η αυτόματη παραλληλοποίηση του κώδικα. Παρ' όλα αυτά, δεν είναι αποδεκτά τα προγράμματα που περιέχουν ορισμούς μέσω της οδηγίας define.

Όταν ο λεκτικός αναλυτής της C αναγνωρίζει μία λεκτική μονάδα που αποτελεί έγκυρο αναγνωριστικό της γλώσσας, θα πρέπει να εξετάσει αν το αναγνωριστικό αυτό έχει οριστεί ως εναλλακτικό όνομα για κάποιον τύπο δεδομένων της C, έτσι ώστε να επιστρέψει το token TYPE_NAME και όχι το IDENTIFIER. Το βήμα αυτό παραλείπεται από τον λεκτικό αναλυτή του μεταφραστή *Ariadne* επειδή δεν υποστηρίζει τον ορισμό εναλλακτικών ονομάτων μέσω typedef.

7.3 Συντακτική Ανάλυση

Ο συντακτικός αναλυτής του μεταφραστή *Ariadne* υλοποιείται αυτόματα με τη βοήθεια του εργαλείου Bison και η περιγραφή των κανόνων της γραμματικής είναι στο αρχείο ariadne.yy. Η γραμματική της γλώσσας αποτελεί υπερσύνολο της γραμματικής της γλώσσας ANSI C. Ο χαρακτηρισμός ως υπερσύνολο οφείλεται στους κανόνες που έχουν προστεθεί για να υποστηριχθεί η εισαγωγή των βιβλιοθηκών μέσω της οδηγίας include και η εισαγωγή οδηγιών pragma από τον προγραμματιστή.

Στον Πηγαίο Κώδικα 7.1 παρουσιάζεται τμήμα του αρχείου ariadne.yy με τις αλλαγές που πραγματοποιήθηκαν στη γραμματικής της ANSI C. Έχουμε ορίσει δύο νέους κανόνες, τους include_definition και ariadne_directive, οι οποίοι αποτελούν πλέον τμήμα του κανόνα external_declaration. Με τις προσθήκες αυτές, ο προγραμματιστής μπορεί να εισάγει βιβλιοθήκες και να προσθέσει οδηγίες για την παραλληλοποίηση των αναδρομικών συναρτήσεων, ακριβώς πριν τον ορισμό τους.

```
external_declaration
    : function_definition
    | declaration
    | include_definition
    | ariadne_directive
    ;

include_definition
    : INCLUDE_STRING
    ;

ariadne_directive
    : ARIADNE_PRAGMA
    ;
```

Πηγαίος Κώδικας 7.1: Οι αλλαγές στη γραμματική της ANSI C στο αρχείο ariadne.yy, για να υποστηριχθεί μέρος του προεπεξεργαστή της C.

Μία επιπλέον αλλαγή έχει πραγματοποιηθεί στη γραμματική της γλώσσας, έτσι ώστε να γίνεται προσθήκη των αγκίστρων που ορίζουν ένα μπλοκ της C σε όλες τις δομές όταν αυτές αποτελούνται από μία μόνο εντολή (statement). Η αλλαγή αυτή γίνεται καθαρά για λόγους ευκολίας κατά τη μετάφραση του προγράμματος. Για να επιτευχθεί αυτό ορίζουμε έναν νέο κανόνα που ονομάζεται statement_brackets και είναι πανομοιότυπος με τον κα-

νόνα statement, με τη διαφορά ότι προσθέτει τα άγκιστρα όταν δεν επιλέγεται ο κανόνας compound_statement. Ο κανόνας statement αντικαθίσταται από τον statement_brackets στους κανόνες labeled_statement, selection_statement και iteration_statement. Εξαίρεση αποτελεί το τμήμα ELSE του κανόνα selection_statement και το τμήμα IDENTIFIER του κανόνα labeled_statement, στα οποία η αντικατάσταση δε γίνεται επειδή δεν προσφέρει κάποιο κέρδος. Οι αλλαγές αυτές παρουσιάζονται στον Πηγαίο Κώδικα 7.2.

```
statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

statement_brackets
    : p28 labeled_statement      { statement_brackets1(); }
    | compound_statement
    | p28 expression_statement   { statement_brackets3(); }
    | p28 selection_statement    { statement_brackets4(); }
    | p28 iteration_statement    { statement_brackets5(); }
    | p28 jump_statement         { statement_brackets6(); }
    ;

labeled_statement
    : IDENTIFIER { labeled_statement1_1(*$1); } ':' p44 statement
    | CASE p31 constant_expression ':' p42 statement_brackets
    | DEFAULT p32 ':' p42 statement_brackets
    ;

selection_statement
    : IF p33 '(' p0 expression ')' p43 statement_brackets
    | IF p33 '(' p0 expression ')' p43 statement_brackets ELSE p34 statement
    | SWITCH p35 '(' p0 expression ')' p43 statement_brackets
    ;

iteration_statement
    : WHILE p36 '(' p0 expression ')' p43 statement_brackets
    | DO p37 statement_brackets WHILE p36 '(' p0 expression ')' ';'
                                    { iteration_statement2(); }
    | FOR p38 '(' p45 expression_statement expression_statement ')' p46
                                                        statement_brackets
    | FOR p38 '(' p45 expression_statement expression_statement expression ')' p46
                                                        statement_brackets
    ;
```

Πηγαίος Κώδικας 7.2: Οι αλλαγές στη γραμματική της ANSI C στο αρχείο ariadne.yy, για να εισαχθούν τα άγκιστρα που ορίζουν ένα μπλοκ σε κάθε δομή με μία μόνο εντολή.

Ο κανόνας p28 εισάγει το αριστερό άγκιστρο στον κώδικα, ενώ η κλήση των συναρτήσεων statement_brackets*() εισάγει το δεξί. Οι άλλοι κανόνες που το όνομά τους αρχίζει με το γράμμα p και ακολουθεί ένας ακέραιος αριθμός, αποτελούν βοηθητικούς κανόνες οι οποίοι αποθηκεύουν τον κώδικα σε μία δομή δεδομένων. Θα αναφερθούμε λεπτομερώς σε αυτούς στην επόμενη ενότητα. Το ίδιο ισχύει και για τις συναρτήσεις που καλούνται και

έχουν το όνομα του κανόνα ακολουθούμενο από έναν ακέραιο αριθμό.

7.4 Ενδιάμεση Αναπαράσταση

Κατά την υλοποίηση ενός μεταφραστή που παράγει κώδικα σε μία γλώσσα υψηλού επιπέδου, υπάρχει η ανάγκη για αποθήκευση του κώδικα εισόδου σε μία δομή δεδομένων. Η επιλογή της δομής δεδομένων αποτελεί μία από τις σημαντικότερες αλλά και δυσκολότερες αποφάσεις που πρέπει να ληφθούν κατά τη διάρκεια υλοποίησης του μεταφραστή, καθώς θα επηρεάσει όλη τη μετέπειτα διαδικασία.

Η αποθήκευση του κώδικα από τον μεταφραστή *Ariadne* γίνεται κατά τη συντακτική ανάλυση μέσω των βοηθητικών κανόνων και συναρτήσεων που καλούνται. Οι βοηθητικές συναρτήσεις είναι υλοποιημένες στο αρχείο πηγαίου κώδικα parserInterface.cpp και αποθηκεύουν τον κώδικα σε ένα αντικείμενο της κλάσης TranslationUnit, δίνοντάς του μία συγκεκριμένη μορφή που διευκολύνει την επεξεργασία του από τον μεταφραστή.

Η κλάση TranslationUnit αποτελεί το μέσο επικοινωνίας μεταξύ του μεταφραστή και της ενδιάμεσης αναπαράστασης του κώδικα. Το σημαντικότερο πεδίο της κλάσης είναι μία λίστα από αντικείμενα της κλάσης Line στα οποία αποθηκεύεται ο κώδικας εισόδου. Ένα αντικείμενο Line αναπαριστά μία γραμμή κώδικα σε C. Η μορφή της είναι αυστηρά καθορισμένη και έχει στην πραγματικότητα τη συνηθισμένη έννοια μίας γραμμής κώδικα. Κάτα τη συντακτική ανάλυση λοιπόν, ο κώδικας διασπάται σε γραμμές και στοιχίζεται με έναν συγκεκριμένο τρόπο. Σύμφωνα με τη στοίχιση του κώδικα πραγματοποιείται η επεξεργασία της ενδιάμεσης αναπαράστασης και εκτελούνται οι απαιτούμενοι μετασχηματισμοί. Υπάρχει δηλαδή η ανάγκη ύπαρξης μίας ενιαίας μορφής. Αυτός είναι και ο λόγος που γίνεται προσθήκη των αγκίστρων κατά τη συντακτική ανάλυση, όταν υπάρχει μόνο μία εντολή.

Τα βασικά πεδία της κλάσης Line είναι ένα αντικείμενο της κλάσης estring και μία λίστα από αντικείμενα της κλάσης Alias. Η κλάση estring είναι επέκταση της κλάσης string της C++, με την έννοια της κληρονομικότητας. Η ύπαρξη της κλάσης estring είναι ουσιαστική, καθώς επαναορίζει (overloading) τη μέθοδο find() για να αγνοεί τα πρότυπα (pattern) που υπάρχουν εντός των συμβολοσειρών μίας γραμμής κώδικα. Για παράδειγμα, στη γραμμή κώδικα του πλαισίου που ακολουθεί, το y είναι ένα αναγνωριστικό με πραγματική σημασία για τη διαδικασία μετάφρασης, ενώ το x αποτελεί απλά μερός της συμβολοσειράς με την οποία αρχικοποείται το y και δεν παρουσιάζει κανένα ενδιαφέρον για τη μετάφραση του κώδικα.

```
char *y = "x = 1";
```

Επομένως, στο αντικείμενο της κλάσης estring αποθηκεύεται η γραμμή κώδικα, η οποία επιδέχεται κάθε μορφή επεξεργασίας, μέσα από τις βασικές μεθόδους της κλάσης string και τις μεθόδους που ορίζονται στην κλάση estring.

Η λίστα αντικειμένων της κλάσης Alias περιλαμβάνει τα ονόματα που χρησιμοποιούνται στη συγκεκριμένη γραμμή κώδικα και είναι διαθέσιμη κατά τη διαδικασία εκτέλεσης ενός περάσματος. Η πληροφορία που διατηρούν τα αντικείμενα της κλάσης, είναι το όνομα και ο τύπος δεδομένων, το επίπεδο εμβέλειας στο οποίο έχει δηλωθεί ή οριστεί, το είδος του

αντικειμένου, δηλαδή αν είναι μεταβλητή, πίνακας ή συνάρτηση, καθώς και η διάστασή τους όταν επρόκειτο για πίνακα. Διατηρείται επίσης η πληροφορία για το αν σε αυτήν τη γραμμή κώδικα επιδέχεται εγγραφή ή όχι.

Η δημιουργία της λίστας γίνεται στο αρχικό στάδιο εφαρμογής ενός περάσματος μετασχηματισμών. Η ανάλυση των ονομάτων που υπάρχουν σε ολόκληρο τον κώδικα γίνεται μέσω της κλάσης AliasAnalysis, η οποία χαρακτηρίζεται από δύο πεδία. Το πρώτο πεδίο είναι ένα αντικείμενο της κλάσης TranslationUnit και το δεύτερο ένα αντικείμενο της κλάσης SymbolTable που αναπαριστά τον πίνακα συμβόλων και θα περιγραφεί στην επόμενη ενότητα. Παρέχει επίσης τη μέθοδο start() η οποία είναι αρμόδια για την ανάλυση των ονομάτων που βρίσκονται στις γραμμές κώδικα με τη βοήθεια του πίνακα συμβόλων.

7.5 Πίνακας Συμβόλων

Ο πίνακας συμβόλων του μεταφραστή *Ariadne* αναπαρίσταται με την κλάση SymbolTable. Αν και το πρότυπο ANSI δεν επιτρέπει τον ορισμό εμφωλευμένων συναρτήσεων στη C, παρέχει τη δυνατότητα ορισμού εμφωλευμένων μπλοκ στο σώμα μίας συναρτησης. Τα μπλοκ αυτά αποτελούν διαφορετικά επίπεδα εμβέλειας (scope) τα οποία θα πρέπει να υποστηριχθούν από τον πίνακα συμβόλων. Έτσι, το μοναδικό πεδίο της κλάσης SymbolTable είναι μία λίστα από αντικείμενα της κλάσης Scope. Το Scope αναπαριστά ένα επίπεδο εμβέλειας, και αποτελείται από μία λίστα οντοτήτων, δηλαδή από αντικείμενα της κλάσης Entity.

Μία οντότητα είναι ένα αναγνωριστικό της C, δηλαδή μία μεταβλητή, ένας πίνακας, μία συνάρτηση ή ακόμα και μία σταθερά. Στα πλαίσια της παρούσας διατριβής υπάρχει ανάγκη να διατηρούμε στον πίνακα συμβόλων, τις μεταβλητές και τις σταθερές τις οποίες κατατάσσουμε σε μία ενιαία κατηγορία την οποία καλούμε SCALAR, τους πίνακες τους οποίους ονομάζουμε ARRAY και τις συναρτήσεις που ονομάζουμε FUNCTION.

Για κάθε ένα από αυτά αποθηκεύουμε το όνομα, τον τύπο δεδομένων και το επίπεδο εμβέλειας, και συγκεκριμένα αν είναι GLOBAL ή LOCAL. Αν είναι πίνακας, αποθηκεύουμε επίσης τη διάστασή του.

Η πρόσβαση στον πίνακα συμβόλων επιτυγχάνεται μόνο από την κλάση AliasAnalysis η οποία δημιουργεί τα αντικείμενα Alias.

7.6 Παραγωγή Κώδικα

Η παραγωγή του κώδικα εξόδου πραγματοποιείται με την εκτέλεση των τριών περασμάτων που υποστηρίζει ο μεταφραστής *Ariadne*. Κάθε πέρασμα υλοποιείται μέσα από μία κλάση που φέρει το όνομά του. Επομένως υπάρχουν τρεις κλάσεις, η Elimination, η ParallelReduction και η ThreadSafe. Και οι τρεις αποτελούν επέκταση της βασικής κλάσης RecursiveFunctionPass, με την έννοια της κληρονομικότητας.

Η κλάση RecursiveFunctionPass χαρακτηρίζεται από δύο πεδία. Το πρώτο είναι μία λίστα από αντικείμενα της κλάσης Line και αναπαριστά το σώμα της αναδρομικής συνάρτησης που

θα παραλληλοποιηθεί. Το δεύτερο είναι ένα αντικείμενο της κλάσης RecursiveFunction και αναπαριστά μία αναδρομική συνάρτηση. Τα βασικά χαρακτηριστικά της αναδρομικής συνάρτησης είναι το όνομα και ο τύπος της, ο δείκτης και ο τύπος του δείκτη, και τρία αντικείμενα της κλάσης InitiatorList.

Η κλάση InitiatorList αναπαριστά μία λίστα από αρχικές τιμές, δηλαδή είναι μία λίστα από αντικείμενα της κλάσης Initiator, που χρησιμοποιούνται στα περάσματα εξάλειψης της αναδρομής, Elimination και ParallelReduction.

Η εκτέλεση κάθε περάσματος ξεκινάει από τη συνάρτηση main(), όπου δημιουργείται ένα αντικείμενο της αντίστοιχης κλάσης και καλείται η μέθοδος run() πάνω σε αυτό. Στη μέθοδο run() κάθε κλάσης, γίνεται αναζήτηση της αντίστοιχης οδηγίας. Αν βρεθεί οδηγία για εφαρμογή του περάσματος, τότε ο μεταφραστής πραγματοποιεί ανάλυση της οδηγίας και εκτελεί μία σειρά από ελέγχους σχετικά με τη μορφή της αναδρομικής συνάρτησης, ώστε να εξασφαλίσει ότι ο μετασχηματισμός του κώδικα μπορεί να εφαρμοστεί με επιτυχία.

Για κάθε πέρασμα υπάρχουν δύο επιλογές όσον αφορά την απεικόνιση του κώδικα. Έτσι, η μέθοδος run() αφού πραγματοποιήσει τους απαραίτητους ελέγχους, καλεί μία από τις δύο μεθόδους που προσφέρει κάθε κλάση για την απεικόνιση του κώδικα. Η εφαρμογή του περάσματος συνεπάγεται και την αφαίρεση της οδηγίας από τον κώδικα εισόδου.

Η επίτευξη των μετασχηματισμών προϋποθέτει την αποθήκευση πληροφορίας που δε διατηρείται στις κλάσεις που έχουμε ήδη περιγράψει. Θα πρέπει λοιπόν να ορίσουμε τέσσερις επιπλέον κλάσεις. Οι δύο πρώτες είναι οι Parameter και ParameterList. Χρησιμοποιούνται για τη συλλογή των παραμέτρων της αναδρομικής συνάρτησης που θα οριστούν ως παράμετροι των νέων συναρτήσεων που δημιουργούνται μετά το μετασχηματισμό. Άρα ένα αντικείμενο της κλάσης Parameter είναι στην πραγματικότητα μία παράμετρος, ενώ ένα αντικείμενο της κλάσης ParameterList είναι μία λίστα από παραμέτρους.

Οι άλλες δύο κλάσεις είναι οι TypedefArrayType και TypedefArrayTypeList. Ο λόγος ύπαρξής τους συνδέεται με τον περιορισμό της γλώσσας SL, σύμφωνα με τον οποίο ο τύπος δεδομένων κατά το πέρασμα παραμέτρων σε μία συνάρτηση νημάτων, θα πρέπει να είναι ενιαίος. Ο μεταφραστής *Ariadne* υπακούει στον περιορισμό αυτό και ορίζει εναλλακτικά ονόματα για τους τύπους με typedef. Ο ορισμός γίνεται μία φορά για κάθε τύπο δεδομένων, και όχι για κάθε πίνακα, άρα θα πρέπει να υπάρχει μία δομή δεδομένων στην οποία θα αποθηκεύονται οι τύποι που έχουν ήδη οριστεί. Η δομή δεδομένων είναι μία λίστα, και συγκεκριμένα η κλάση TypedefArrayTypeList που χαρακτηρίζεται από δύο πεδία.

Το πρώτο πεδίο είναι ένας ακέραιος αριθμός με το ρόλο του μετρητή, για να υπολογίζει το πλήθος των τύπων που έχουν ήδη οριστεί και να αναθέτει ένα μοναδικό εναλλακτικό όνομα σε κάθε τύπο. Το δεύτερο πεδίο είναι μία λίστα με αντικείμενα της κλάσης TypedefArrayType. Η πληροφορία που διατηρεί η κλάση είναι ένα αναγνωριστικό που αντιστοιχεί στο εναλλακτικό όνομα του τύπου που ορίζεται μέσω αυτής, καθώς και ο τύπος δεδομένων στον οποίο ορίζεται το εναλλακτικό όνομα. Ο μεταφραστής *Ariadne* χρειάζεται μόνο ένα αντικείμενο της κλάσης TypedefArrayTypeList που θα είναι ορατό από όλα τα περάσματα. Η δήλωση του αντικειμένου πραγματοποιείται στο αρχείο ariadne.cpp ως καθολική μεταβλητή.

Όταν τουλάχιστον μία αναδρομική συνάρτηση του αρχείου εισόδου απεικονίζεται στο

μοντέλο SVP, ο μεταφραστής *Ariadne* μετασχηματίζει τη συνάρτηση main() σύμφωνα με τη γλώσσα SL, αρκεί να μην έχει παραμέτρους. Ο μετασχηματισμός γίνεται μέσω ενός διαφορετικού περάσματος που παρέχεται με την κλάση MainFunctionPass. Η εκτέλεση του περάσματος γίνεται όπως και στα άλλα περάσματα, δηλαδή με τη δημιουργία ενός αντικειμένου της κλάσης, στη συνάρτηση main(), και καλώντας τη μέθοδο run() πάνω στο αντικείμενο.

Τέλος, η απεικόνιση τουλάχιστον μίας αναδρομικής συνάρτησης στο πρότυπο νημάτων POSIX, έχει ως αποτέλεσμα την εισαγωγή του αρχείου κεφαλίδων pthread.h στο αρχείο εξόδου.

7.7 Περιορισμοί του Μεταφραστή

Η ευελιξία της γλώσσας C αποτελεί σημαντικό πλεονέκτημα για έναν προγραμματιστή. Από την άλλη όμως, δημιουργεί αρκετές δυσκολίες στην υλοποίηση του μεταφραστή. Υπάρχουν έλεγχοι περιπτώσεων που έχουν εσκεμμένα αγνοηθεί καθώς δεν έχουν να προσφέρουν τίποτα απολύτως στην αυτόματη παραλληλοποίηση κώδικα. Επομένως, ο μεταφραστής *Ariadne* συνοδεύεται από ένα σύνολο λογικών περιορισμών με στόχο τη διευκόλυνση της υλοποίησης, χωρίς όμως να χάνεται το νόημα των μετασχηματισμών του. Στη συνέχεια της ενότητας ακολουθεί εκτενής περιγραφή των περιορισμών.

Αρχείο Εισόδου

Το αρχείο εισόδου θα πρέπει να είναι σημασιολογικά ορθό, με την έννοια ότι έχει μεταφραστεί επιτυχώς από κάποιον μεταφραστή όπως ο gcc. Ο μεταφραστής *Ariadne* δεν πραγματοποιεί ελέγχους σχετικά με τη δήλωση δύο μεταβλητών με το ίδιο όνομα ή τη χρήση ονόματος με διαφορετικό τρόπο από αυτόν που επιτρέπει ο ορισμός του. Αυτοί και άλλοι παρόμοιοι έλεγχοι δεν πραγματοποιούνται για λόγους ευκολίας.

Εναλλακτικά Ονόματα Τύπων Δεδομένων

Αν το αρχείο εισόδου περιλαμβάνει ονόματα που έχουν ως τύπο δεδομένων ένα εναλλακτικό όνομα που έχει οριστεί μέσω typedef, τότε προκαλείται συνταχτικό λάθος. Αυτό οφείλεται στο γεγονός ότι ο λεκτικός αναλυτής όταν αναγνωρίζει μία λεκτική μονάδα που πληροί τις προϋποθέσεις του αναγνωριστικού, δεν εξετάζει αν αποτελεί εναλλακτικό όνομα τύπων, καθώς δε διατηρείται η πληροφορία αυτή σε κάποια δομή δεδομένων.

Δήλωση Μεταβλητών

Η δήλωση μεταβλητών σε μπλοκ το οποίο είναι εμφωλευμένο στο σώμα συνάρτησης υποστηρίζεται από τον πίνακα συμβόλων. Παρ' όλα αυτά, για απλοποίηση της διαδικασίας των μετασχηματισμών, οι συναρτήσεις που περιλαμβάνουν τέτοιες δηλώσεις δε μετασχηματίζονται από τον μεταφραστή.

Εντολή Άλματος

Η εντολή goto αντιβαίνει στην έννοια του δομημένου προγραμματισμού επειδή είναι ικανή να αλλάζει τη ροή εκτέλεσης ολόκληρου του προγράμματος. Ως εκ τούτου, δεν μπορεί να αντιμετωπιστεί από τον μεταφραστή *Ariadne*. Όταν περιλαμβάνεται στο αρχείο εισόδου, δεν εφαρμόζεται κανένα πέρασμα μετασχηματισμών.

Αριθμητικά Σύστηματα

Οι σταθερές που αναγνωρίζονται από τον μεταφραστή *Ariadne* είναι μόνο αυτές του δεκαδικού συστήματος. Ο περιορισμός αυτός ισχύει για τις σταθερές που επιδέχονται επεξεργασία και μόνο, όπως συμβαίνει στην εξαγωγή των αρχικών τιμών μίας αναδρομικής συνάρτησης. Η χρήση σταθερών τόσο στο οκταδικό όσο και στο δεκαεξαδικό σύστημα αριθμών δεν παραβαίνει κανέναν περιορισμό εφόσον δεν απαιτείται κάποια επεξεργασία από τον μεταφραστή. Σε διαφορετική περίπτωση ο κώδικας που θα παραχθεί, ενδεχομένως να μην είναι ορθός.

Δείκτης σε Συνάρτηση

Η χρήση δείκτη σε συνάρτηση δε θα πρέπει να παρουσιάζεται στις παραμέτρους μίας αναδρομικής συνάρτησης. η οποία επιθυμούμε να μετασχηματιστεί. Όμως, μπορεί να χρησιμοποιηθεί χωρίς κανέναν περιορισμό σε οποιοδήποτε άλλο σημείο του αρχείου εισόδου.

Τελεστής comma

Ο τελεστής ',' δεν πρέπει να χρησιμοποιείται από τον προγραμματιστή σε τμήματα κώδικα τα οποία επρόκειτο να δεχτούν επεξεργασία από τον μεταφραστή και αποτελούν ειδικές περιπτώσεις χρήσης του.

Κεφαλαιο 8

Πειραματικα Αποτελεσματα

8.1 Εκτέλεση Πειραμάτων

Τα πειράματα που παρουσιάζονται στις επόμενες ενότητες πραγματοποιήθηκαν για την αξιολόγηση όλων των περασμάτων του μεταφραστή *Ariadne*. Έγιναν μετρήσεις για αναδρομικές συναρτήσεις που απεικονίστηκαν στο μοντέλο SVP και στο πρότυπο νημάτων POSIX. καθώς και για συναρτήσεις στις οποίες εφαρμόστηκε απλή εξάλειψη αναδρομής παράγοντας κώδικα σε C. Σε όλα τα διαγράμματα γίνεται σύγκριση του αναδρομικού κώδικα, με αυτόν που προκύπτει από την εφαρμογή του περάσματος από τον μεταφραστή *Ariadne*.

8.1.1 Πειράματα για το Μοντέλο SVP

Τα πειράματα για τις συναρτήσεις που απεικονίστηκαν στο μοντέλο SVP παράγοντας κώδικα για τη γλώσσα SL, έχουν εκτελεστεί κάνοντας χρήση του μεταφραστή και του περιβάλλοντος προσομοίωσης, rbm256 της έκδοσης 3.1.154-r4008. Οι μετρήσεις πραγματοποιήθηκαν σε 4,

8, ή 64 πυρήνες, και η απόδοση της εκτέλεσης εκφράζεται σε κύκλους ρολογιού (clock cycles).

Στο πλαίσιο που ακολουθεί παρουσιάζουμε τον τρόπο με τον οποίο γίνεται η μετάφραση ενός προγράμματος της γλώσσας SL με τον μεταφραστή slc και η εκτέλεσή του στον προσομοιωτή slr.

```
#compile
slc <file-name>  -o <bin-name>  -b mta

#run
slr <bin-name>  -n <num-of-cores>
```

8.1.2 Πειράματα για το Πρότυπο Νημάτων POSIX

Τα πειράματα των συναρτήσεων που παραλληλοποιήθηκαν με το πρότυπο νημάτων POSIX πραγματοποιήθηκαν σε επεξεργαστή Intel ® Core™2 Quad Processor Q8400 (4M Cache, 2.66 GHz, 1333 MHz FSB) και σε κύρια μνήμη 6GB. Μεταφράστηκαν με τον gcc (Debian 4.4.5-8) 4.4.5 και εκτελέστηκαν στο λειτουργικό σύστημα Debian GNU/Linux 6.0.1 (squeeze). Η απόδοση της εκτέλεσης στις μετρήσεις εκφράζεται σε μικροδευτερόλεπτα (microsecond) που υπολογίστηκαν με την κλήση gettimeofday() του αρχείου κεφαλίδων sys/time.h.

8.2 Υπολογισμός της Ακολουθίας Fibonacci

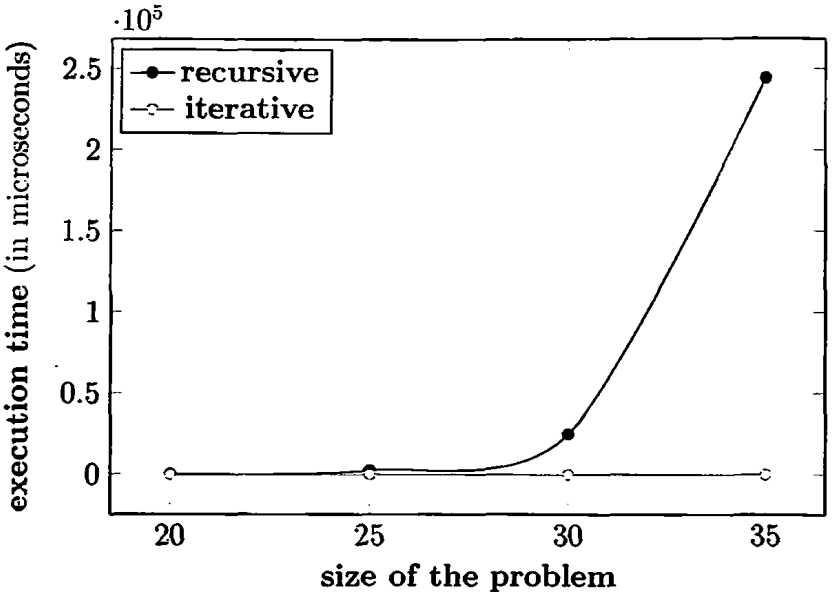
Στο Σχήμα 8.1 παρουσιάζονται τα πειραματικά αποτελέσματα από τη σύγκριση που έγινε μεταξύ της αναδρομικής συνάρτησης για τον υπολογισμό της ακολουθίας Fibonacci και της συνάρτησης που προκύπτει από την εξάλειψη της αναδρομής, παράγοντας κώδικα σε C. Παρατηρούμε ότι η διαφορά στην απόδοση είναι πολύ μεγάλη, κάτι που είναι απολύτως λογικό γιατί συγκρίνουμε μία αναδρομική συνάρτηση με μία επαναληπτική. Όταν το μέγεθος του προβλήματος είναι 20 ή 25, η διαφορά παραμένει πολύ μεγάλη, αλλά δεν μπορεί να αναπαρασταθεί στο σχήμα επειδή η διαφορά είναι πολύ πιο μικρή από αυτήν για μεγαλύτερο μέγεθος προβλήματος.

Παρόμοια αποτελέσματα παίρνουμε και από τη σύγκριση της ίδια συνάρτησης με αυτήν που παράγεται από την εξάλειψη της αναδρομής παράγοντας κώδικα σε SL. Τα αποτελέσματα παρουσιάζονται στο Σχήμα 8.2.

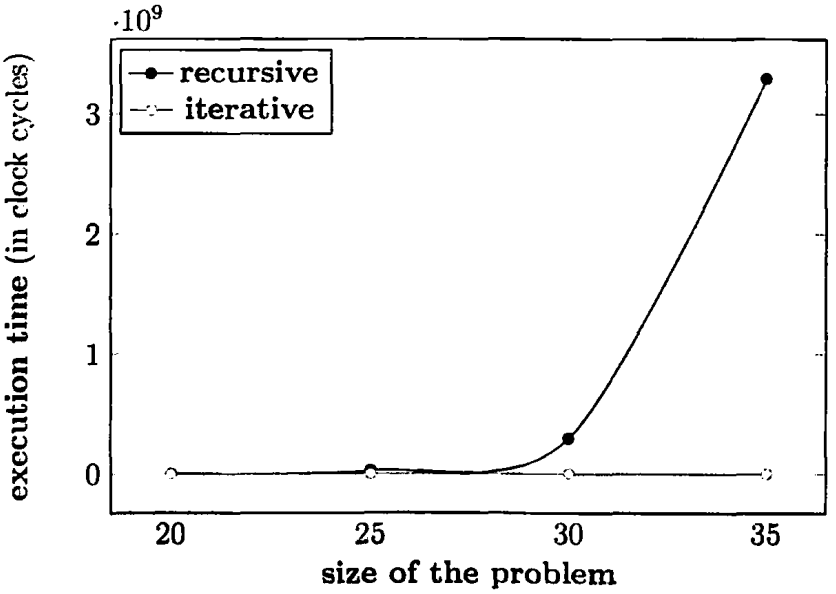
Στα Σχήματα 8.3 και 8.4 γίνεται παρουσίαση των μετρήσεων που λάβαμε από τη σύγκριση της ίδιας αναδρομικής συνάρτησης, με αυτήν που παράγεται κατά την εφαρμογή του περάσματος *thread-safe* για απεικόνιση του κώδικα στο μοντέλο SVP και στο πρότυπο νημάτων POSIX αντίστοιχα.

Να υπενθυμίσουμε ότι το πέρασμα *thread-safe* δεν κάνει απαλοιφή της αναδρομής, σε αντίθεση με το πέρασμα *elimination*. Αυτό έχει ως συνέπεια η βελτίωση της απόδοσης να μην είναι η ίδια, όπως φαίνεται και από τα πειραματικά αποτελέσματα. Η τάση των μη

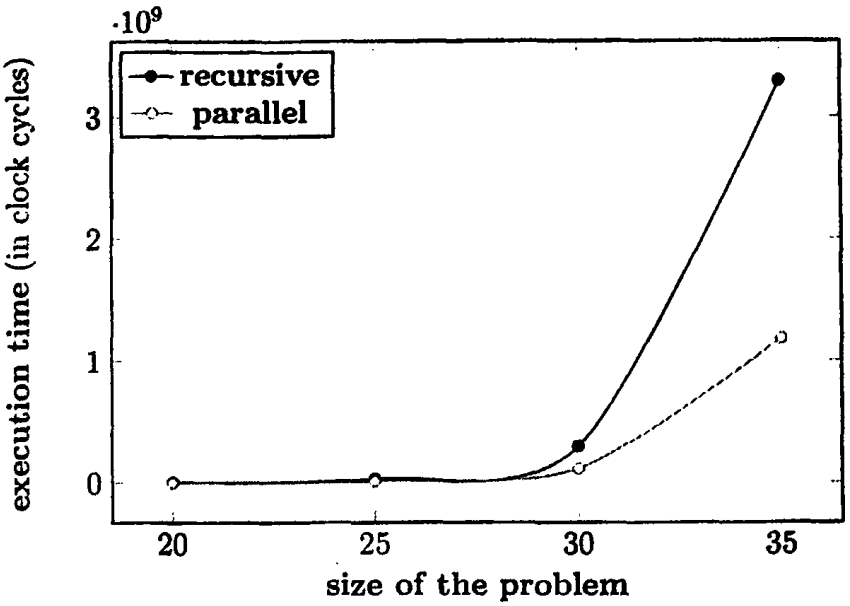
αναδρομικών μετρήσεων στα Σχήματα 8.1 και 8.2, σε σχέση με την τάση των μετρήσεων στα Σχήματα 8.3 και 8.4, μας δείχνουν ξεκάθαρα ότι το πέρασμα *elimination* είναι πιο αποδοτικό σε σχέση με το *thread-safe* για την αναδρομική συνάρτηση υπολογισμού της ακολουθίας Fibonacci.



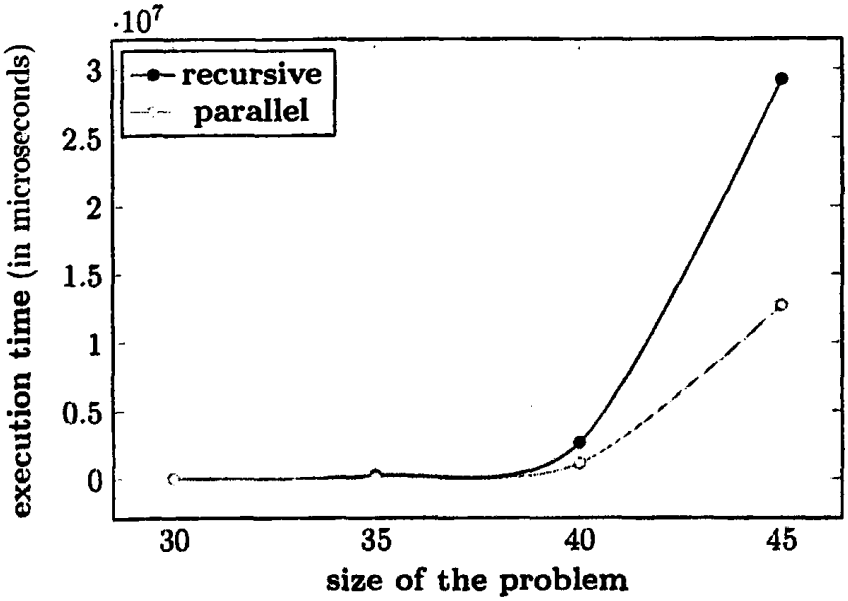
Σχήμα 8.1: Εφαρμογή του περάσματος *elimination* για παραγωγή κώδικα σε C. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης.



Σχήμα 8.2: Εφαρμογή του περάσματος *elimination* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης, χρησιμοποιώντας 4 πυρήνες.



Σχήμα 8.3: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης, επιτρέποντας τη δημιουργία νημάτων μέχρι 5 επίπεδα και χρησιμοποιώντας 4 πυρήνες.

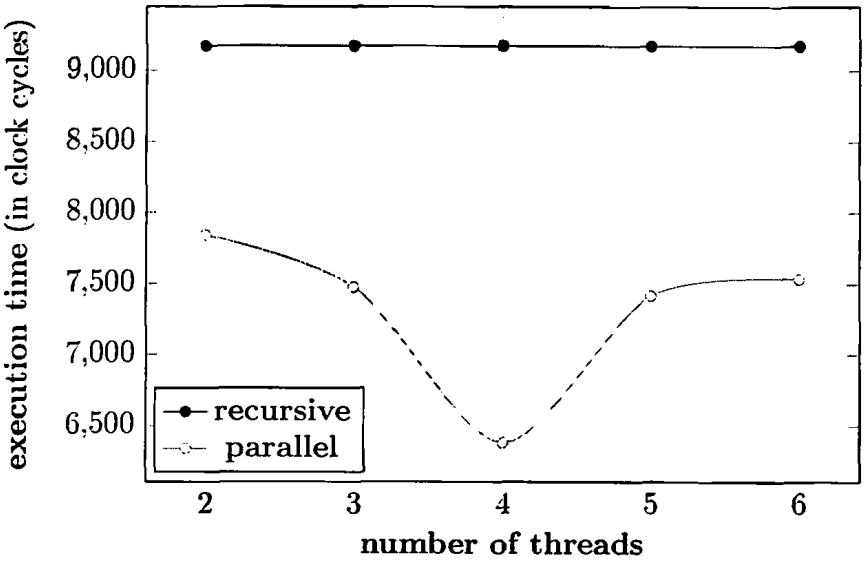


Σχήμα 8.4: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα για το πρότυπο νημάτων POSIX. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης, επιτρέποντας τη δημιουργία νημάτων μέχρι 5 επίπεδα και χρησιμοποιώντας 4 πυρήνες.

8.3 Υπολογισμός Δύναμης

Οι μετρήσεις που λάβαμε από τα πειραματικά αποτελέσματα για την αναδρομική συνάρτηση υπολογισμού της δύναμης ενός αριθμού, σε σχέση με τη συνάρτηση που παράγεται από την εφαρμογή του περάσματος *parallel-reduction* για απεικόνιση στο μοντέλο SVP, απεικονίζονται στο Σχήμα 8.5.

Παρατηρούμε ότι η χρήση νημάτων με στόχο την παράλληλη εκτέλεση των υπολογισμών παρουσιάζει σημαντική βελτίωση της απόδοσης. Πιο συγκεκριμένα, η απόδοση βελτιώνεται σημαντικά μέχρι και τη δημιουργία 4 νημάτων. Για περισσότερα νήματα υπάρχει κέρδος κατά την παράλληλη εκτέλεση, αλλά δεν είναι τόσο μεγάλο όσο όταν κατανέμουμε τους υπολογισμούς σε 4 νήματα. Αυτό συμβαίνει επειδή το μέγεθος των υπολογισμών δεν είναι αρκετά μεγάλο για να απαιτούνται περισσότερα από 4 νήματα.



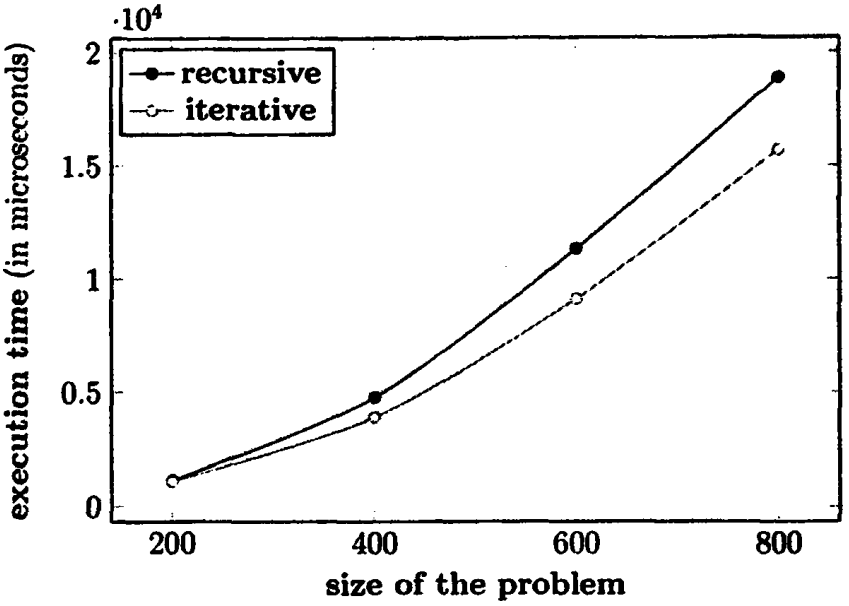
Σχήμα 8.5: Εφαρμογή του περάσματος *parallel-reduction* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για τον υπολογισμό του 2^{40} με διαφορετικό αριθμό νημάτων, χρησιμοποιώντας 8 πυρήνες.

8.4 Προσεγγιστικός Υπολογισμός του Ημιτόνου

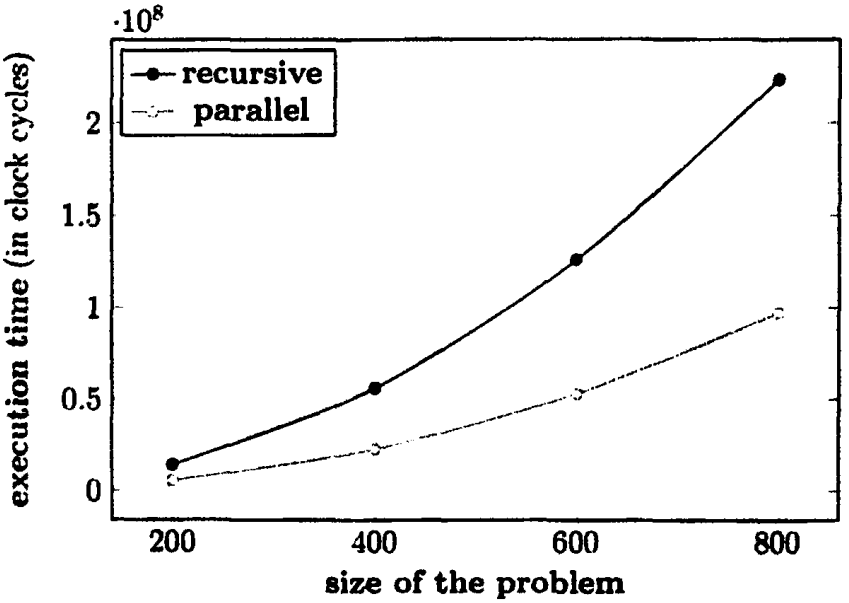
Τα πειραματικά αποτελέσματα που ακολουθούν έχουν ως στόχο να μας διδάξουν δύο βασικά πράγματα. Το πρώτο είναι η φύση του μοντέλου SVP, που μας δίνει τη δυνατότητα να επιτύχουμε μία πρωτότυπη μορφή παραλληλισμού και το δεύτερο είναι ο σημαντικός βαθμός παραλληλίας που εξάγεται με την κατανομή των υπολογισμών σε μία σειρά από νήματα.

Στο Σχήμα 8.6 παρουσιάζονται τα αποτελέσματα των πειραμάτων που λάβαμε από τη σύγκριση της απόδοσης της αναδρομικής συνάρτησης για την προσέγγιση του ημιτόνου. με αυτής που προκύπτει από την απαλοιφή της αναδρομής. Αντίστοιχα, τα αποτελέσματα για την απεικόνιση της συνάρτησης στο μοντέλο SVP μετά την εφαρμογή του περάσματος *elimination*. φαίνονται στο Σχήμα 8.7. Παρατηρώντας τα δύο αυτά Σχήματα, συμπεραίνου-

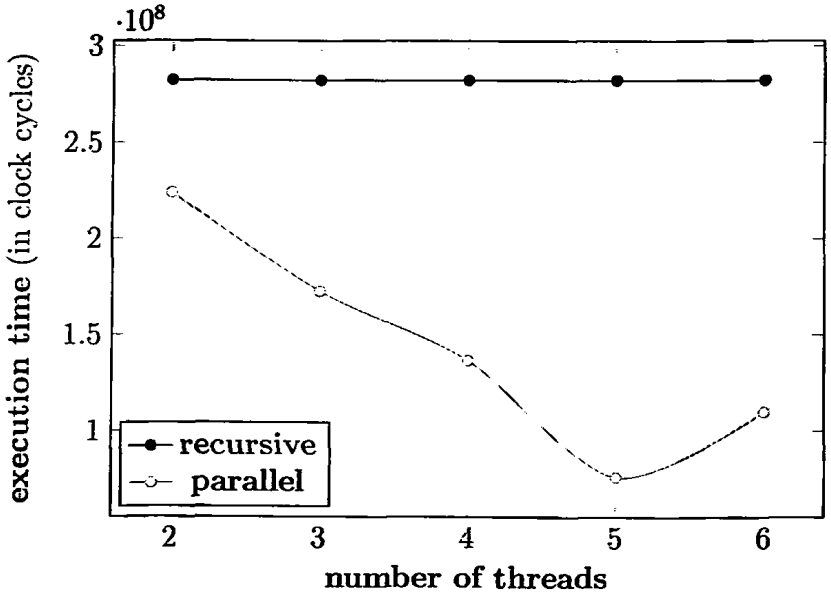
με ότι όταν πραγματοποιείται εξάλειψη της αναδρομής στο μοντέλο SVP, η διαφορά στην απόδοση είναι ιδιαίτερα σημαντική, σε σχέση με αυτήν που χαρακτηρίζει την απλή εξάλειψη αναδρομής. Συνεπώς, το μοντέλο SVP εκμεταλλεύεται τα χαρακτηριστικά της συγκεκριμένης αναδρομικής συνάρτησης και επιτυγχάνει την εξαγωγή παραλληλισμού.



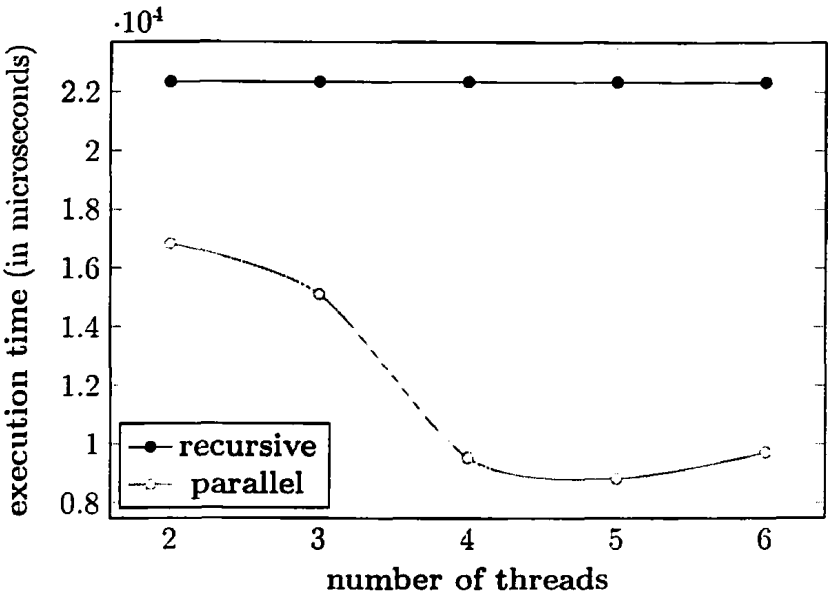
Σχήμα 8.6: Εφαρμογή του περάσματος *elimination* για παραγωγή κώδικα σε C. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης.



Σχήμα 8.7: Εφαρμογή του περάσματος *elimination* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για διαφορετική είσοδο της συνάρτησης, χρησιμοποιώντας 4 πυρήνες.



Σχήμα 8.8: Εφαρμογή του περάσματος *parallel-reduction* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για την προσέγγιση του sin(1.1) με 875 όρους, με διαφορετικό αριθμό νημάτων, χρησιμοποιώντας 8 πυρήνες.



Σχήμα 8.9: Εφαρμογή του περάσματος *parallel-reduction* για παραγωγή κώδικα του προτύπου νημάτων POSIX. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για την προσέγγιση του sin(1.1) με 875 όρους, με διαφορετικό αριθμό νημάτων, χρησιμοποιώντας 4 πυρήνες.

Στο Σχήμα 8.8 παρουσιάζουμε τα αποτελέσματα από τα πειράματα που διεξήχθησαν για τη σύγκριση της απόδοσης μεταξύ της ίδιας αναδρομικής συνάρτησης, και αυτής που προκύπτει από την εφαρμογή του περάσματος *parallel-reduction* για παραγωγή κώδικα σε SL.

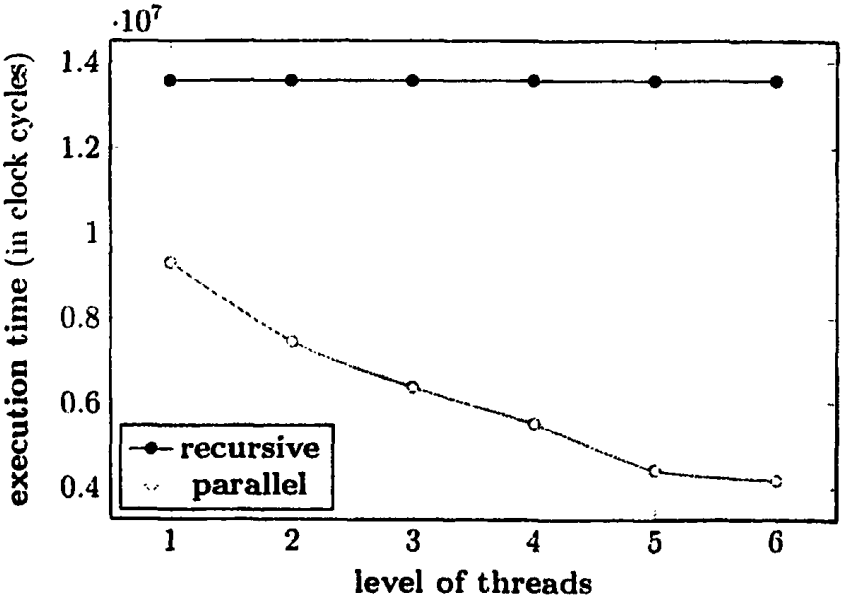
Η κατανομή του φόρτου σε περισσότερα από ένα νήματα γίνεται με μεγάλη επιτυχία, και συγκεκριμένα μέχρι τα 5 νήματα, καθώς οι υπολογισμοί δεν είναι τόσοι ώστε να απαιτούνται περισσότερα.

Στο Σχήμα 8.9 παρατηρούμε ότι τα αποτελέσματα που λάβαμε από τις μετρήσεις για την ίδια αναδρομική συνάρτηση σε σχέση με αυτήν που παράγεται από την εφαρμογή του περάσματος *parallel-reduction* για το πρότυπο νημάτων POSIX, είναι ελαφρώς διαφορετικά. Η βέλτιστη κατανομή των υπολογισμών επιτυγχάνεται και σε αυτήν την περίπτωση με τη χρήση 5 νημάτων κάτι που όμως οφείλεται σε ανακρίβεια των μετρήσεων, καθώς υπάρχουν μόνο 4 διαθέσιμοι πυρήνες.

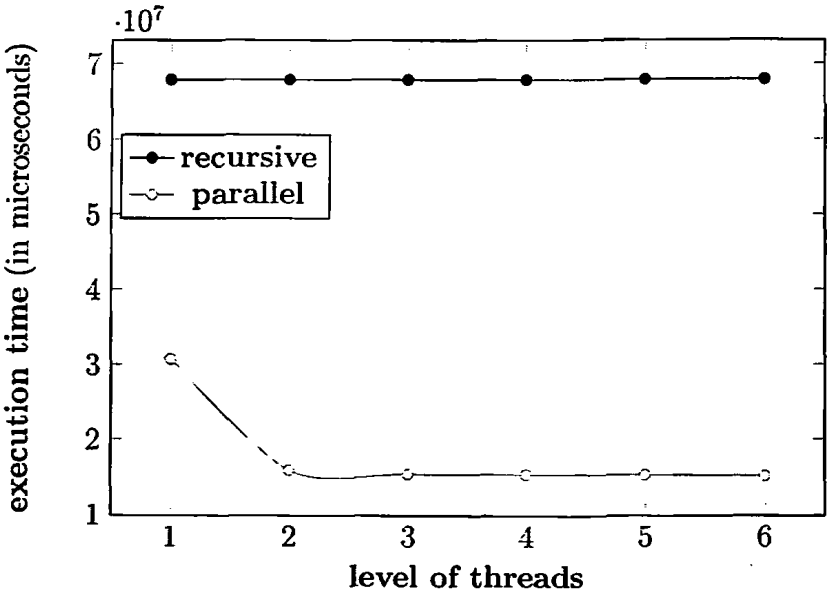
8.5 Προσέγγιση του Ολοκληρώματος του Ημιτόνου

Τα πειραματικά αποτελέσματα που ακολουθούν συλλέχθηκαν για την αξιολόγηση του περάσματος *thread-safe* για την προσέγγιση του ολοκληρώματος του ημιτόνου με τη μέθοδο διαίρει και βασίλευε. Στο Σχήμα 8.10 φαίνονται οι μετρήσεις που λάβαμε από την εφαρμογή του περάσματος για το μοντέλο SVP, ενώ στο Σχήμα 8.11 για το πρότυπο νημάτων POSIX.

Όσον αφορά το μοντέλο SVP παρατηρούμε ότι η απόδοση βελτιώνεται διαρκώς, όσο αυξάνουμε το επιτρεπτό επίπεδο για τη δημιουργία των νημάτων. Από την άλλη στο πρότυπο νημάτων POSIX, ενώ αρχικά υπάρχει σημαντική βελτίωση, στη συνέχεια βλέπουμε ότι η απόδοση παραμένει σταθερή. Αυτό εξηγείται από το γεγονός ότι χρησιμοποιούμε μόνο 4 πυρήνες. Επειδή η συνάρτηση έχει δύο αναδρομικές κλήσεις, στο πρώτο επίπεδο δημιουργούνται δύο νήματα, ενώ στο δεύτερο κάθε νήμα δημιουργεί άλλα δύο, άρα συνολικά έξι.



Σχήμα 8.10: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για την προσέγγιση του ολοκληρώματος στο διάστημα $[0, \pi]$ με ακρίβεια $d = 10^{-3}$, επιτρέποντας τη δημιουργία νημάτων μέχρι διαφορετικό επίπεδο και χρησιμοποιώντας 64 πυρήνες.



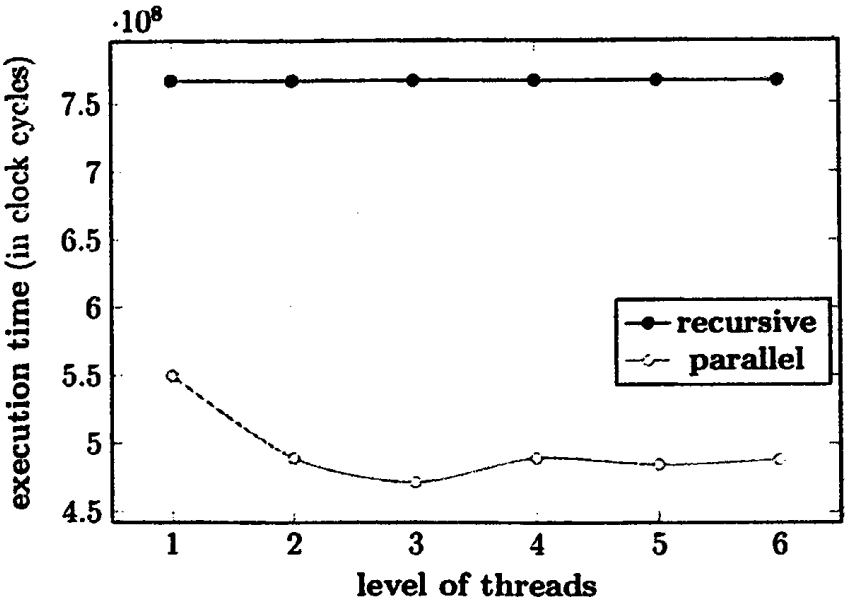
Σχήμα 8.11: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα του προτύπου νημάτων POSIX. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για την προσέγγιση του ολοκληρώματος στο διάστημα $[0, \pi]$ με ακρίβεια $d = 10^{-8}$, επιτρέποντας τη δημιουργία νημάτων μέχρι διαφορετικό επίπεδο και χρησιμοποιώντας 4 πυρήνες.

8.6 Ταξινόμηση με τον Αλγόριθμο Merge-Sort

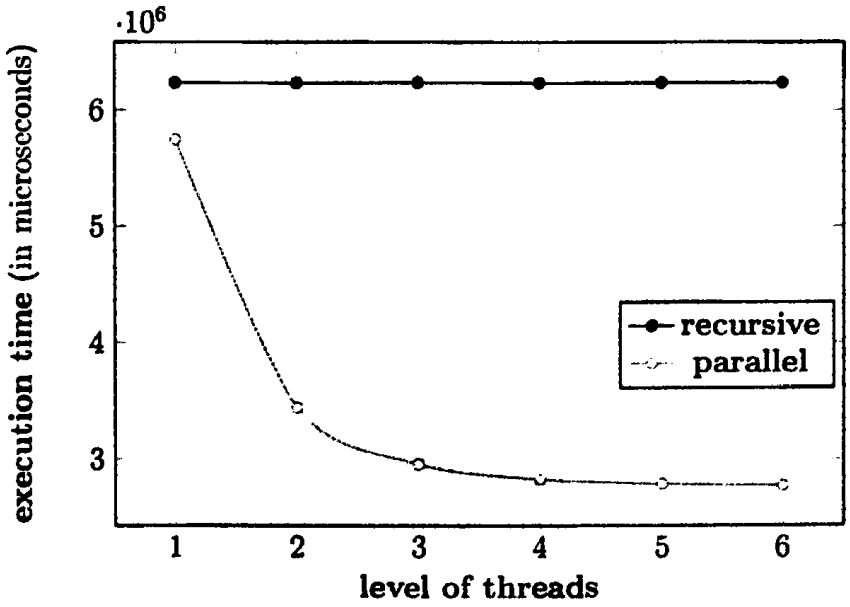
Η βελτίωση στην απόδοση της αναδρομικής συνάρτησης για ταξινόμηση σύμφωνα με τον αλγόριθμο Merge-Sort, από την εφαρμογή του περάσματος *thread-safe*. παρουσιάζεται στο Σχήμα 8.12 για το μοντέλο SVP και στο Σχήμα 8.13 για το πρότυπο νημάτων POSIX.

Στις μετρήσεις που πραγματοποιήθηκαν για το μοντέλο SVP, ταξινομήθηκαν 100.000 στοιχεία. Αρχικά παρατηρείται σημαντική βελτίωση στην απόδοση όσο δημιουργούνται νήματα, αλλά στη συνέχεια δείχνει να μειώνεται. Οι υπολογισμοί που απαιτούνται για την ταξινόμησή τους, αποδείχθηκε ότι δεν είναι αρκετοί για να απορροφήσουν τα νήματα που δημιουργούνται μετά τα τρία επίπεδα.

Για τα πειράματα του προτύπου νημάτων POSIX χρησιμοποιήσαμε 10.000.000 στοιχεία προς ταξινόμηση. Αυτό έχει αντίκτυπο στη βελτίωση της απόδοσης, καθώς απαιτούνται περισσότεροι υπολογισμοί, οι οποίοι εκμεταλλεύονται την παραλληλία που τους προσφέρεται. Όμως, και σε αυτήν την περίπτωση παρατηρούμε ότι η απόδοση βελτιώνεται αισθητά μέχρι και το τρίτο επίπεδο. Στη συνέχεια σταθεροποιείται επειδή δεν υπάρχουν διαθέσιμοι πόροι για να εκτελέσουν τα επιπλέον νήματα που δημιουργούνται.



Σχήμα 8.12: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα στο μοντέλο SVP. Οι μετρήσεις εκφράζονται σε κύκλους ρολογιού και πραγματοποιήθηκαν για την ταξινόμηση ενός πίνακα 100.000 στοιχείων, επιτρέποντας τη δημιουργία νημάτων μέχρι διαφορετικό επίπεδο και χρησιμοποιώντας 64 πυρήνες.



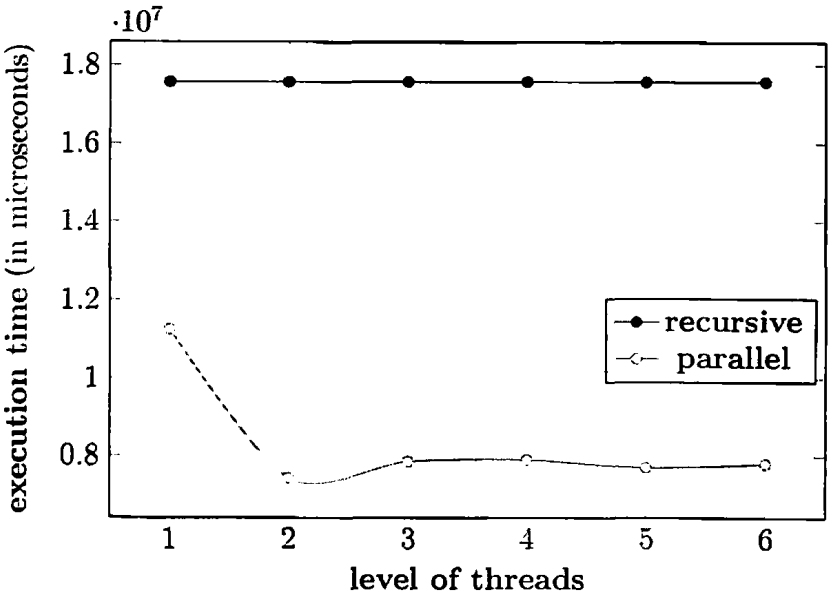
Σχήμα 8.13: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα του προτύπου νημάτων POSIX. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για την ταξινόμηση ενός πίνακα 10.000.000 στοιχείων, επιτρέποντας τη δημιουργία νημάτων μέχρι διαφορετικό επίπεδο και χρησιμοποιώντας 4 πυρήνες.

8.7 Εύρεση της Απόστασης των Πλησιέστερων Σημείων

Στο Σχήμα 8.14 παρουσιάζονται οι μετρήσεις που λάβαμε από τα πειράματα για την εύρεση της ελάχιστης απόστασης μεταξύ δύο σημείων, από ένα σύνολο 10.000.000 σημείων στο επίπεδο, που δημιουργήθηκαν τυχαία μέσω της rand(). Ο υπολογισμός πραγματοποιείται αναδρομικά με τη μέθοδο διαίρει και βασίλευε. Η σύγκριση γίνεται μεταξύ του αρχικού αναδρομικού κώδικα και του κώδικα που παράγεται από την εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα στο πρότυπο νημάτων POSIX.

Παρατηρούμε ότι υπάρχει σημαντική βελτίωση στην απόδοση όταν επιτρέπουμε να δημιουργηθούν νήματα μέχρι το πρώτο επίπεδο. Περαιτέρω βελτίωση σημειώνεται και κατά τη δημιουργία νημάτων μέχρι το δεύτερο επίπεδο, ενώ για περισσότερα επίπεδα δε μειώνεται ο χρόνος εκτέλεσης επειδή δεν υπάρχουν αρκετοί διαθέσιμοι πυρήνες για να αναλάβουν την εκτέλεση των νημάτων που δημιουργούνται.

Για το μοντέλο SVP δεν πραγματοποιήθηκαν πειράματα επειδή ο μεταφραστής slc δεν υποστηρίζει την κλήση rand() για την παραγωγή των τυχαίων σημείων.



Σχήμα 8.14: Εφαρμογή του περάσματος *thread-safe* για παραγωγή κώδικα του προτύπου νημάτων POSIX. Οι μετρήσεις εκφράζονται σε μικροδευτερόλεπτα και πραγματοποιήθηκαν για την εύρεση της απόσταση ανάμεσα σε 10.000.000 σημεία στο επίπεδο, επιτρέποντας τη δημιουργία νημάτων μέχρι διαφορετικό επίπεδο και χρησιμοποιώντας 4 πυρήνες.

Κεφαλαιο 9

Μελλοντικες Επεκτασεις

9.1 Μελλοντικές Επεκτάσεις

Ένα από τα θέματα μελλοντικής επέκτασης για τον μεταφραστή *Ariadne*, αποτελεί η αποδοτική απεικόνιση του κώδικα στο μοντέλο OpenMP, στη γλώσσα Cilk και στο πρότυπο MPI. Συγκεκριμένα, θα μπορούσαμε να προσφέρουμε στον προγραμματιστή τη δυνατότητα απεικόνισης του κώδικα στο μοντέλο OpenMP κατά την εφαρμογή του περάσματος *parallel-reduction*. Ο μεταφραστής *Ariadne* θα πραγματοποιεί απαλοιφή της αναδρομής μετασχηματίζοντας τη συνάρτηση από αναδρομική σε επαναληπτική, εφαρμόζοντας ουσιαστικά το πέρασμα *elimination* για παραγωγή κώδικα σε C. Στη συνέχεια θα προσθέτει πάνω από το βρόχο την οδηγία reduction του OpenMP για να γίνει η κατανομή των υπολογισμών σε μία σειρά από νήματα. Αυτό βέβαια δεν μπορεί να πραγματοποιηθεί για τον τελεστή της διαίρεσης και για τη μία από τις δύο μορφές του τελεστή της αφαίρεσης καθώς δεν υποστηρίζονται από το OpenMP. Το ίδιο μπορεί να γίνει και για το πέρασμα *thread-safe*, όπου σε κάθε ανεξάρτητη αναδρομική κλήση θα γίνεται προσθήκη της οδηγίας task, ώστε να εκτελούνται παράλληλα. Με αυτόν τον τρόπο ο μεταφραστής *Ariadne* απλά θα προσθέτει οδηγίες. αφήνοντας τη δημιουργία των νημάτων στο OpenMP.

Η δυνατότητα παραγωγής κώδικα για τη γλώσσα Cilk είναι εφικτή για το πέρασμα ***thread-safe***, και ο αρχικός κώδικας μπορεί να παραλληλοποιηθεί αποδοτικά πραγματοποιώντας ελάχιστες αλλαγές. Ο μεταφραστής *Ariadne* θα πρέπει να αναγνωρίζει τις αναδρομικές κλήσεις στον κώδικα της συνάρτησης και να εισάγει τη λέξη spawn, η οποία δηλώνει ότι μετά την αντίστοιχη κλήση, ο πατέρας μπορεί να συνεχίσει να εκτελεί τον κώδικά του, χωρίς να περιμένει να ολοκληρωθεί η αναδρομική κλήση που έγινε spawn. Όπως στο μοντέλο OpenMP, έτσι και στη γλώσσα Cilk, ο μεταφραστής *Ariadne* θα πρέπει να πραγματοποιήσει μικρές αλλαγές, χωρίς να εστιάσει στη δημιουργία και στη διαχείριση των νημάτων.

Η απεικόνιση του κώδικα στο πρότυπο MPI θα μπορούσε να πραγματοποιηθεί για το πέρασμα *parallel-reduction*, όπως και στην περίπτωση του μοντέλου OpenMP. Όμως, σε αντίθεση με το OpenMP, πέρα από την απαλοιφή της αναδρομής, ο μεταφραστής *Ariadne* θα πρέπει να αναλάβει την κατανομή των επαναλήψεων του βρόχου, στις διεργασίες που δημιουργούνται, και να υπολογίσει το τελικό αποτέλεσμα. Αυτά μπορούν να πραγματοποιηθούν με τη βοήθεια των κλήσεων MPI_Bcast() και MPI_Reduce(), αλλά η MPI_Reduce() δεν υποστηρίζει την πράξη της αφαίρεσης και της διαίρεσης, παρά μόνο αυτή της πρόσθεσης και του πολλαπλασιασμού. Παρόλο που το πρότυπο MPI δίνει στον προγραμματιστή τη δυνατότητα να ορίσει υπό περιορισμούς, δικές του πράξεις (user-defined operations) που θα υπολογίζουν το τελικό αποτέλεσμα, δεν μπορεί να υποστηριχτεί η πλήρης μορφή της αφαίρεσης και της διαίρεσης, καθώς δε μας προσφέρει την απαραίτητη πληροφορία για τον ακριβή αριθμό των επαναλήψεων που ανέλαβε κάθε διεργασία.

Ενδιαφέρον παρουσιάζει επίσης η υποστήριξη περισσότερων δυαδικών τελεστών για το πέρασμα *parallel-reduction*. Οι τελεστές αυτοί είναι οι '&&', '||', '&', '|' και '^', οι οποίοι υποστηρίζονται και από την οδηγία reduction του OpenMP κάτι που διευκολύνει την απεικόνιση του κώδικα στο μοντέλο OpenMP, σύμφωνα με τα όσα αναφέραμε προηγουμένως.

Κύριος στόχος του μεταφραστή *Ariadne* είναι η υποστήριξη διαφορετικών μορφών αναδρομικών συναρτήσεων, για κάθε μία από τις οποίες θα παρέχει μία οδηγία για να μπορεί ο προγραμματιστής να υποδείξει τη μορφή στην οποία ανήκει η αναδρομική συνάρτηση. Έτσι, η υποστήριξη περισσότερων μορφών μπορεί να αποτελέσει σημαντική επέκταση για τον μεταφραστή *Ariadne*. Μία τέτοια μορφή είναι αυτή των αναδρομικών συναρτήσεων που μπορούν να μετατραπούν σε επαναληπτικές, αλλά δε χαρακτηρίζονται από την ύπαρξη ενός δείκτη όπως στα περάσματα *elimination* και *parallel-reduction*. Ένα παράδειγμα αποτελεί η αναδρομική συνάρτηση για την ελευθέρωση μίας συνδεδεμένης λίστας, η οποία μπορεί να μετατραπεί σε επαναληπτική μέσω ενός βρόχου, αλλά δεν έχει κάποια παράμετρο που θα μπορούσε να χαρακτηριστεί ως δείκτης σύμφωνα με τις ιδιότητες που έχουμε ορίσει.

Τέλος, στην τρέχουσα έκδοση του μεταφραστή *Ariadne* υπάρχει ο περιορισμός για τη δομή της αναδρομικής συνάρτησης. Σύμφωνα με τον περιορισμό θα πρέπει ο κώδικας της αναδρομικής συνάρτησης να βασίζεται σε μία κύρια δομή ελέγχου. Ο περιορισμός οφείλεται σε θέματα ευκολίας υλοποίησης του μεταφραστή, μειώνοντας σημαντικά τις γραμμές κώδικα, χωρίς όμως να χάνεται η ουσία των περασμάτων που προσφέρει. Παρ' όλα αυτά, θα ήταν σημαντική βελτίωση η αφαίρεση του περιορισμού ώστε να υποστηριχθεί μία πιο ευέλικτη μορφή των συναρτήσεων. Το ίδιο ισχύει και για τους υπόλοιπους περιορισμούς του μεταφραστή *Ariadne*, η κατάργηση των οποίων θα έχει ως συνέπεια την ευελιξία του κώδικα εισόδου και όχι την αποτελεσματικότερη εξαγωγή παραλληλισμού.

Κεφαλαιο 10

Επιλογος

10.1 Επίλογος

Στην παρούσα διατριβή ασχοληθήκαμε με την αυτόματη παραλληλοποίηση του κώδικα. Υλοποιήσαμε τον μεταφραστή *Ariadne* ο οποίος παρέχει τρία περάσματα για την αυτόματη παραλληλοποίηση αναδρομικών συναρτήσεων, κάνοντας χρήση οδηγιών που δίνονται από τον προγραμματιστή.

Το θεωρητικό υπόβαθρο της εργασίας παρουσιάστηκε στο δεύτερο κεφάλαιο με την αναφορά στο μοντέλο SVP, στο τρίτο κεφάλαιο με την περιγραφή της γλώσσας SL, και στο τέταρτο κεφάλαιο με την αναφορά στο πρότυπο νημάτων POSIX. Μέσα από αυτά τα κεφάλαια έγινε μία προσπάθεια κατανόησης των πλεονεκτημάτων που προσφέρει το μοντέλο SVP μέσω των καινοτόμων χαρακτηριστικών του, σε σχέση με το πρότυπο νημάτων POSIX. Αναφέραμε επίσης παρόμοιες εργασίες που υπάρχουν στη βιβλιογραφία των μεταφραστών παραλληλοποίησης, στο πέμπτο κεφάλαιο.

Στο έκτο κεφάλαιο σχολιάσαμε λεπτομερώς τα θέματα που αφορούν τη σχεδίαση του μεταφραστή *Ariadne* παρουσιάζοντας αναλυτικά παραδείγματα για την καλύτερη κατανόηση των μετασχηματισμών του. Στο έβδομο κεφάλαιο αναφερθήκαμε στις τεχνικές αντικειμενοστρεφούς σχεδίασης που χρησιμοποιήθηκαν για την υλοποίηση του μεταφραστή, καθώς και σε λεπτομέρειες για την οργάνωση του πηγαίου κώδικα.

Από τα πειράματα που διεξήχθησαν, παρουσιάσαμε τα αποτελέσματα στο όγδοο κεφάλαιο. Γίνεται εύκολα αντιληπτή η σημαντική βελτίωση στην απόδοση ενός προγράμματος, όταν αυτό παραλληλοποιηθεί με τον μεταφραστή *Ariadne*. Η οδηγία που απαιτείται για την παραλληλοποίηση του κώδικα είναι απλή και σύντομη και δεν προϋποθέτει κάποια επίπονη διαδικασία από τον προγραμματιστή. Συνεπώς, ο μεταφραστής *Ariadne* αποτελεί έναν εύκολο τρόπο για την παραλληλοποίηση αναδρομικών συναρτήσεων, απεικονίζοντας τον κώδικα

είτε στο μοντέλο SVP παράγοντας κώδικα για τη γλώσσα SL, είτε στο πρότυπο νημάτων POSIX.

Τέλος, στο ένατο κεφάλαιο αναφέραμε μερικές μελλοντικές επεκτάσεις που θα μπορούσαν να πραγματοποιηθούν στον μεταφραστή *Ariadne*. Οι επεκτάσεις αυτές παρουσιάζουν ιδιαίτερο ενδιαφέρον καθώς μπορούν να προσφέρουν επιπλέον δυνατότητες στον προγραμματιστή, εξάγοντας παραλληλισμό από περισσότερες αναδρομικές συναρτήσεις.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol, and L. Zhang, "A General Model of Concurrency and its Implementation as Many-Core Dynamic RISC Processors," in *Proc. of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS '08)*, pp. 1–9, Samos, Greece, 2008

[2] K. Bousias, L. Guang, C. Jesshope, and M. Lankamp, "Implementation and Evaluation of a Microthread Architecture," *Journal of Systems Architecture - Embedded Systems Design*, vol. 55, no. 3, pp. 149–161, 2009

[3] C. Jesshope, M. Lankamp, and L. Zhang, "The implementation of an SVP many-core processor and the evaluation of its memory architecture," *SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 38–45, 2009

[4] *SVP model reference*, Computer Systems Architecture Group, University of Amsterdam, 2012

[5] *SL language reference*, Computer Systems Architecture Group, University of Amsterdam, 2011

[6] R. 'kena' Poss, "On the realizability of hardware microthreading," Ph.D. dissertation. University of Amsterdam, 2012

[7] M. Gupta. S. Mukhopadhyay, and N. Sinha, "Automatic Parallelization of Recursive Procedures," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (IEEE PACT '99)*, pp. 139–148, Newport Beach, California, USA, 1999

[8] R. Rugina and M. Rinard, "Automatic Parallelization of Divide and Conquer Algorithms," in *Proc. of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '99)*, pp. 72–83, Atlanta, Georgia, USA. 1999

[9] R. L. Collins. B. Vellore, and L. P. Carloni, "Recursion-Driven Parallel Code Generation for Multi-Core Platforms," in *Design, Automation and Test in Europe (DATE '10)*, pp. 190–195, Dresden, Germany, 2010

[10] A. Morihata and K. Matsuzaki, "Automatic Parallelization of Recursive Functions Using Quantifier Elimination," in *Proc. of the 10th International Conference on Functional and Logic Programming (FLOPS '10)*, Lecture Notes in Computer Science, vol. 6009, pp. 321–336, Sendai, Japan, 2010

[11] W. L. Harrison, "The Interprocedural Analysis and Automatic Parallelization of Scheme Programs," *LISP and Symbolic Computation*, vol. 2, no. 2, pp. 179–396, 1989

[12] D. Saougkos, A. Mastoras, and G. Manis, "Fine Grained Parallelism in Recursive Function Calls," in *Proc. of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM '11)*, Lecture Notes in Computer Science, vol. 7204, pp. 121–130, Torun, Poland, 2012

[13] A. Mastoras, "Automatic Code Parallelization with Cetus," Bachelor's thesis, Department of Computer Science, School of Sciences, University of Ioannina, 2011

[14] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1," 2011

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pp. 207–216, Santa Barbara, California, USA, 1995

[16] *Cilk 5.4.6 Reference Manual*, Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 2001

[17] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," *IEEE Parallel and Distributed Technology*, vol. 2, no. 3, pp. 37–47, 1994

[18] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen, "The Polaris Internal Representation," *International Journal of Parallel Programming*, vol. 22, no. 5, pp. 553–586, 1994

[19] S. P. Amarasinghe, J.-A. M. Anderson, M. S. Lam, and C.-W. Tseng, "An Overview of the SUIF Compiler for Scalable Parallel Machines," in *Proc. of the 7th SIAM Conference on Parallel Processing for Scientific Computing (PPSC '95)*, pp. 662–667, San Francisco, CA, 1995

[20] S. I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation," in *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, Lecture Notes in Computer Science, vol. 2958, pp. 539–553, College Station, TX, USA, 2004

[21] T. A. Johnson, S. I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. P. Midkiff, "Experiences in Using Cetus for Source-to-Source Transformations," in *Proc. of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, Lecture Notes in Computer Science, vol. 3602, pp. 1–14, West Lafayette, IN, USA, 2005

[22] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009

[23] T. J. Parr and R. W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," *Software Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995

[24] H. Kasahara, M. Obata, K. Ishizaka, K. Kimura, H. Kaminaga, H. Nakano, K. Nagasawa, A. Murai, H. Itagaki, and J. Shirako, "Multigrain Automatic Parallelization in Japanese Millennium Project IT21 Advanced Parallelizing Compiler," in *Proc. of the International Conference on Parallel Computing in Electrical Engineering (PARELEC '02)*, pp. 105–111, Warsaw, Poland, 2002

[25] H. Nakano, T. Kodaka, K. Kimura, and H. Kasahara, "Memory Management for Data Localization on OSCAR Chip Multiprocessor," in *Proc. of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA '04)*, pp. 82–88, Maui. HI, 2004

[26] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model," in *Proc. of the 17th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS CC '08)*, Lecture Notes in Computer Science, vol. 4959, pp. 132–146, Budapest, Hungary, 2008

[27] U. Bondhugula. A. Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Program Optimization System," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pp. 101–113, Tucson, AZ, USA, 2008

[28] C. Lengauer, "Loop Parallelization in the Polytope Model," in *Proc. of the 4th International Conference on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science, vol. 715, pp. 398–416, Hildesheim, Germany, 1993

[29] P. Feautrier, "Parametric Integer Programming," *RAIRO Recherche Op'erationnelle*, vol. 22, pp. 243–268, 1988

[30] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *Proc. of the 13th International Conference on Parallel Architectures and*

Compilation Techniques (IEEE PACT '04), pp. 7–16, Antibes Juan-les-Pins, France, 2004

[31] E. Ayguade, C. R. Calidonna, J. Corbalan, M. Giordano, M. Gonzalez. H. C. Hoppe, J. Labarta. M. M. Furnari, X. Martorell, N. Navarro, D. S. Nikolopoulos, J. Oliver, T. S. Papatheodorou, and E. D. Polychronopoulos, "NANOS: Effective Integration of Fine-grain Parallelism Exploitation and Multiprogramming," 1999

[32] Universitat Politecnica De Catalunya and P. Gmbh, "The NANOS Environment User Guide," 1999

[33] H. Saito, N. Stavrakos, S. Carroll, C. D. Polychronopoulos, and A. Nicolau, "The Design of the PROMIS Compiler," in *Proc. of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on Theory and Practice of Software, (ETAPS CC '99)*, Lecture Notes in Computer Science, vol. 1575, pp. 214–228, Amsterdam, The Netherlands, 1999

[34] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers," in *Proc. of the 1st International Workshop on Parallel Processing*, pp. 322–330, Bangalore, India, 1994

[35] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, vol. 28, no. 10, pp. 37–47, 1995

[36] D. Saougkos and G. Manis, "A parallelizing compiler for the microgrid: Exploiting concurrency from software continuity," in *the AppleCore Project Workshop: Making Multi-cores Mainstream, organized during High Performance Embedded Applications and Compilers (HiPEAC)*, Paris, France, 2012

[37] D. Saougkos, D. Evgenidou, and G. Manis, "Specifying Loop Transformations for C2uTC Source to Source Compiler," in *Proc. of the 14th Workshop on Compilers for Parallel Computing (CPC'09)*, Zurich, Switzerland, 2009

[38] D. Saougkos and G. Manis, "Run-Time Scheduling with the C2uTC/SL Parallelizing Compiler." in *Proc. of the Workshop on Parallel Programming and Run-Time Management Techniques for many-core Architectures, organized during the 24th International Conference on Architecture of Computing Systems (ARCS '11)*, pp. 151–157, Como, Italy, 2011

[39] D. Saougkos and G. Manis, "Self Adaptive Run Time Scheduling for the Automatic Parallelization of Loops with the C2uTC/SL Compiler," Parallel Computing, (to appear)

ΔΗΜΟΣΙΕΥΣΕΙΣ

Dimitris Saougkos, Aristeidis Mastoras, and George Manis, "Fine Grained Parallelism in Recursive Function Calls," in Proc. of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM '11), Lecture Notes in Computer Science, vol. 7204, pp. 121-130, Torun, Poland, 2012

ΒΙΟΓΡΑΦΙΚΟ

Ο Αριστείδης Μάστορας γεννήθηκε στο Αργυρόκαστρο την 1η Απριλίου 1990. Αποφοίτησε από το Γενικό Λύκειο Ζωσιμαίας Σχολής Ιωαννίνων με βαθμό «Λίαν Καλώς» 17.1/20 τον Ιούνιο του 2007. Τον Οκτώβριο του ίδιου έτους εισήχθη στο Τμήμα Πληροφορικής της Σχολής Θετικών Επιστημών του Πανεπιστημίου Ιωαννίνων όπου και πραγματοποίησε τις βασικές σπουδές, αποφοιτώντας με βαθμό «Άριστα» 9.25/10 τον Ιούνιο του 2011. Στη συνέχεια έγινε δεκτός στο Πρόγραμμα Μεταπτυχιακών Σπουδών του ίδιου Τμήματος, για την απόκτηση Μεταπτυχιακού Διπλώματος Ειδίκευσης στην Πληροφορική με εξειδίκευση στο Λογισμικό.

Στα τρία πρώτα έτη των προπτυχιακών σπουδών έλαβε υποτροφία και βραβείο επίδοσης από το *Ίδρυμα Κρατικών Υποτροφιών*, και στο τέταρτο έτος βραβεύτηκε από το ίδιο Ίδρυμα για την αποφοίτησή του ως πρώτος μεταξύ των συμφοιτητών του. Έλαβε επίσης υποτροφία επίδοσης από τη *Συνεταιριστική Τράπεζα Ηπείρου* για το δεύτερο έτος και βραβείο επίδοσης από το *Κοινωφελές Ίδρυμα Ιωάννης Σ. Λάτσης (ΙΑΟΑ)* για το τέταρτο έτος. Κατά τη διάρκεια των μεταπτυχιακών σπουδών διετέλεσε υπότροφος του *Ιδρύματος Κρατικών Υποτροφιών* και του *Κοινωφελούς Ιδρύματος Αλέξανδρος Σ. Ωνάσης*. Για την άριστη επίδοσή του βραβεύτηκε επίσης από το *Κοινωφελές Ίδρυμα Ιωάννης Σ. Λάτσης* στο πρώτο έτος των μεταπτυχιακών σπουδών.