Ανάλυση ροής δεδομένων στον μεταφραστή OMPi

Βαρθαλίτης Ηλίας

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Δημακόπουλος

Ιωάννινα, Ιούνιος 2025

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Περίληψη

Ηλίας Βαρθαλίτης, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2025.

Ανάλυση ροής δεδομένων στον μεταφραστή OMPi.

Επιβλέπων: Βασίλειος Β. Δημακόπουλος, Καθηγητής.

Η παρούσα εργασία επικεντρώνεται στην ανάλυση ροής δεδομένων στον μεταφραστή OMPi. Η ανάλυση ροής δεδομένων αποτελεί ένα από τα πιο βασικά ζητήματα στη σχεδίαση μεταφραστών, καθώς αποτελεί τη βάση για διάφορες βελτιστοποιήσεις. Ο OMPi είναι ένας μεταφραστής πηγαίου σε πηγαίο κώδικα (source-to-source) για προγράμματα που χρησιμοποιούν το OpenMP. Δεδομένου ότι ο OMPi βασίζεται στον απλό μεταφραστή του συστήματος για τη μετάφραση του τελικού παραγόμενου κώδικα, η βελτιστοποίηση του παραγόμενου κώδικα δεν αποτέλεσε το κίνητρο στην ανάπτυξή του. Ως εκ τούτου, λείπει η λειτουργικότητα της ανάλυσης ροής δεδομένων από τον OMPi. Στο πλαίσιο αυτής της διπλωματικής εργασίας, υλοποιούμε έναν πλήρως γενικό και επεκτάσιμο μηχανισμό ανάλυσης ροής δεδομένων που επιτρέπει την επίλυση οποιουδήποτε σχετικού προβλήματος. Σε αντίθεση με τη συνήθη πρακτική, ο μηχανισμός μας είναι διπλής φύσης, καθώς μπορεί να λειτουργήσει είτε πάνω στο CFG (Control Flow Graph) είτε πάνω στο AST (Abstract Syntax Tree) , ανάλογα με την προτίμηση του προγραμματιστή. Τέλος, εφαρμόζουμε τον μηχανισμό μας στο πολύ σημαντικό πρόβλημα της αυτόματης αναγνώρισης των ιδιοτήτων των μεταβλητών που χρησιμοποιούνται μέσα σε περιοχές κώδικα OpenMP. Πιο συγκεκριμένα, ο μηχανισμός αναλύει τη συμπεριφορά κάθε μεταβλητής, εντοπίζοντας τη φράση στην οποία ανήκει, και προειδοποιεί το προγραμματιστή OpenMP για τυχόν λανθασμένη επιλογή φράσεων κοινοχρησίας.

Abstract

Ilias Varthalitis, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, June 2025.

Data flow analysis in the OMPi compiler.

Advisor: Vassilios V. Dimakopoulos, Professor.

The present work focuses on data flow analysis in the OMPi compiler. Data-flow analysis constitutes one of the most fundamental issues in compiler design, as it forms the basis for various optimizations. OMPi is a source-to-source compiler for programs that use OpenMP. Given that OMPi relies on the system's base compiler for translating the final generated code, the optimization of the generated code was not a motive in its development. As a result, data flow analysis functionality is missing from OMPi. Within the framework of this diploma thesis, we implement a fully general and extensible data flow analysis mechanism that allows the solution of any related problem. Contrary to common practice, our mechanism is of a dual nature, as it can operate either on the CFG (Control Flow Graph) or on the AST (Abstract Syntax Tree), depending on the programmer's preference. Finally, we apply our mechanism to the very important problem of automatically recognizing the properties of the variables used inside OpenMP code regions. More specifically, the mechanism analyzes the behavior of each variable, locating the statement to which it belongs, and then proceeds to the next processing stage.

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος προγραμμάτων

Κεφάλαιο 1.

Εισαγωγή

1.1 Εξέλιξη της τεχνολογίας και προβλήματα

Τα τελευταία 20 χρονιά η τεχνολογία έχει αναπτυχθεί ραγδαία. Η συνεχής βελτίωση της οδηγεί σε προβλήματα τα οποία δεν είχαμε να αντιμετωπίσουμε στο παρελθόν. Συγκεκριμένα ο Αμερικάνος επιστήμονας και συνιδρυτής της Intel, Gordon Moore προέβλεψε ότι ο αριθμός των τρανζίστορ που μπορούν να χωρέσουν σε ένα ολοκληρωμένο κύκλωμα διπλασιάζεται περίπου κάθε δύο χρόνια, οδηγώντας σε μεγαλύτερη υπολογιστική ισχύ και μικρότερο κόστος ανά τρανζίστορ. Ένα χαρακτηριστικό παράδειγμα που βασίζεται στον νόμο του Moore [1] είναι η προσπάθεια για αύξηση των συχνοτήτων λειτουργίας και ταυτόχρονα η σμίκρυνση των φυσικών διαστάσεων των ηλεκτρονικών κυκλωμάτων. Στόχος είναι να διπλασιαστεί ο αριθμός των τρανζίστορ που χωράνε σε ένα ηλεκτρονικό κύκλωμα και αυτό γίνεται καθώς μικραίνει το μέγεθος τους. Αυτό φέρνει σαν αποτέλεσμα την αύξηση της ταχύτητας των κυκλωμάτων. Η τεχνική αυτή όμως δημιούργησε ένα βασικό πρόβλημα: την απότομη αύξηση της κατανάλωσης ενέργειας. Η δυναμική κατανάλωση ισχύος αυξάνεται ανάλογα με την συχνότητα. Έτσι, η αύξηση της συχνότητας προκάλεσε δραματική αύξηση στην κατανάλωση ενέργειας.

Για να επιλυθεί αυτό το πρόβλημα η ιδέα είναι απλή. Αντί να υπάρχει ένας ισχυρός επεξεργαστής ο οποίος εκτελεί σειριακά κάθε εντολή, να υπάρχουν πολλοί (όχι απαραίτητα το ίδιο ισχυροί) επεξεργαστές μαζί ώστε να εκτελούνται πολλές εντολές ταυτοχρονα. Εδώ εμφανίζονται τα παράλληλα συστήματα όπου ουσιαστικά πολλοί επεξεργαστές επικοινωνούν και συνεργάζονται για να λύσουν ένα πρόβλημα.

1.2 Παράλληλα Συστήματα

Τα παράλληλα συστήματα είναι υπολογιστικές μονάδες που εκμεταλεύονται την ταυτόχρονη εκτέλεση πολλαπλών διεργασιών με στόχο την αύξηση της ταχύτητας και αποδοτικότητας. Τα τελευταία χρόνια τα παράλληλα συστήματα βρίσκονται στο

επίκεντρο της τεχνολογίας κυρίως με την μορφή πολυπύρηνων επεξεργαστών. Χωρίζονται σε κυρίως δύο μεγάλες κατηγορίες:

- Στα συστήματα κοινής μνήμης
- Στα συστήματα κατανεμημένης μνήμης

1.2.1 Συστήματα Κοινής Μνήμης

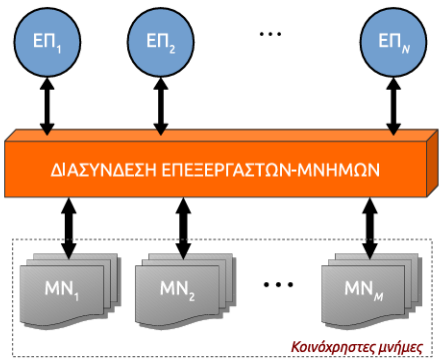
Η κεντρική ιδέα στα συστήματα κοινής μνήμης είναι ότι οι επεξεργαστές δεν έχουν ιδιωτικές μνήμες. Αντιθέτως, μοιράζονται μια συλλογή από όμοιες μνήμες 1.1 [1], η προ σπέλαση των οποίων απαιτεί τον ίδιο χρόνο από όποιον επεξεργαστή και να γίνει. Γι' αυτό [1] και τα συστήματα αυτά είναι επίσης γνωστά και ως συστήματα ομοιόμορφης προσπέλασης μνήμης (UMA—Uniform Memory Access).

Ο προγραμματισμός συστημάτων κοινής μνήμης ακολουθεί το μοντέλο κοινού χώρου διευθύνσεων, με χρήση νημάτων. Τα νήματα μοιράζονται στους επεξεργαστές ώστε να επιτευχθεί η παράλληλη εκτέλεση.

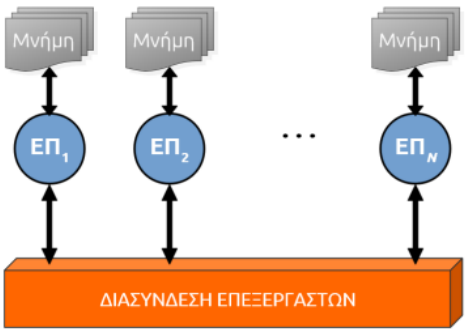
1.2.2 Συστήματα Κατανεμημένης Μνήμης

Στα συστήματα κατανεμημένης μνήμης κάθε επεξεργαστής διαθέτει δική του ιδιωτική μνήμη και συνδέονται μεταξύ τους μέσω ενός δικτύου διασύνδεσης 1.2 [1] . Η επικοινωνία μεταξύ των επεξεργαστών γίνεται μέσω ανταλλαγής μηνυμάτων επάνω στις γραμμές του δικτύου.

Η οργάνωση των υπολογιστών αυτών βασίζεται σχεδόν αποκλειστικά στο δίκτυο διασύνδεσης μεταξύ των επεξεργαστών, το οποίο θα πρέπει να είναι γρήγορο, έτσι ώστε να εισάγονται όσο το δυνατόν λιγότερες καθυστερήσεις κατά τις επικοινωνίες.



Σχήμα 1.1: Οργάνωση κοινόχρηστης μνήμης



Σχήμα 1.2: Οργάνωση κατανεμημένης μνήμης

1.3 Παράλληλος προγραμματισμός

Ο παράλληλος προγραμμάτισμος αναπτύχθηκε με πολύ μικρότερη ταχύτητα από τι τα παράλληλα συστήματα. Αυτό είναι λογικό καθώς ο προγραμματισμός σε σειριακά μοντέλα είναι πολύ διαφορετικός. Αυτό έπεται στο γεγονός ότι στον παράλληλο προγραμματισμό πρέπει να λάβουμε υπόψη περισσότερα πράγματα από τι στο σειριακό. Για παράδειγμα, η ανάπτυξη ενός παράλληλου προγράμματος εξαρτάται σημαντικά και από την αρχιτεκτονική του συστήματος όπως επίσης δεν υπάρχει και πλήρης συμβατότητα για διαφορετικά μηχανήματα. Ο παράλληλος προγραμματισμός χωρίζεται σε τρεις βασικές κατηγορίες:

- Μοντέλο παραλληλισμού δεδομένων
- Μοντέλο κοινόχρηστου χώρου διευθύνσεων
- Μοντέλο μεταβίβασης μηνυμάτων

Στο μοντέλο παραλληλισμού δεδομένων [1] όλοι οι επεξεργαστές κάθε χρονική στιγμή εκτελούν την ίδια εντολή. Υπάρχουν δομές που μοιράζουν τα δεδομένα σε κάθε επεξεργαστή.

Στο μοντέλο κοινόχρηστου χώρου διευθύνσεων όλο το πρόγραμμα αποτελείται από διεργασίες οι οποίες χρησιμοποιούν κοινόχρηστες μεταβλητές. Χρησιμοποιείται σε συστήματα κοινόχρηστης μνήμης όπου οι μεταβλητές είναι κοινές για όλες τις διεργασίες. Έτσι δημιουργούνται προβλήματα όπως η ταυτόχρονη πρόσβαση σε μια μεταβλητή από δύο διεργασίες. Λύσεις σε τέτοια προβλήματα δίνουν δομές όπως οι σημαφόροι ή οι κλειδαριές.

Στο μοντέλο μεταβίβασης μηνυμάτων όλο το πρόγραμμα αποτελείται από αυτόνομες διεργασίες οι οποίες δεν χρησιμοποιούν κοινές μεταβλητές. Το μοντέλο αυτό χρησιμοποιείται σε συστήματα κατανεμημένης μνήμης.

Εμάς μας ενδιαφέρει ο κοινός χώρος διευθύνσεων, δηλαδή ο παραλληλισμός εντός ενός κόμβου, όπου όλες οι διεργασίες εκτελούνται σε μία κοινή μνήμη, επιτρέποντας την ανταλλαγή δεδομένων μέσω κοινών μεταβλητών. Στο πλαίσιο του παραλληλισμού εντός ενός κόμβου, το OpenMP αποτελεί το δημοφιλέστερο πρότυπο προγραμματισμού για κοινόχρηστο χώρο διευθύνσεων. Παρέχει έναν απλό και ευέλικτο τρόπο διαχείρισης παραλληλισμού μέσω οδηγιών, επιτρέποντας τη δημιουργία και τον έλεγχο νημάτων σε πολυπύρηνα συστήματα.

1.4 Αντικείμενο Διπλωματικής

Στο πλαίσιο αυτής της διπλωματικής εργασίας, υλοποιείται ένας γενικός μηχανισμός ανάλυσης ροής δεδομένων στον παραλληλοποιητικό μεταφραστή OMPi (Κεφάλαιο 3). Ο OMPi είναι ένας ελαφρύς, ανοιχτού κώδικα μεταφραστής και σύστημα εκτέλεσης για OpenMP στη γλώσσα C. Διαθέτει μια αρχιτεκτονική που επιτρέπει τον πειραματισμό με διάφορες βιβλιοθήκες νημάτων, ενώ παράλληλα προσφέρει εξαιρετικά ανταγωνιστική απόδοση.

Αντικείμενο της διπλωματικής είναι η υλοποίηση ενός μηχανισμού ανάλυσης ροής δεδομένων (Data Flow Analysis) μέσα στον OMPi [5], ώστε μέσω αυτού να μπορούν να υλοποιούνται πληθώρα αναλύσεων του κώδικα και να ενεργοποιούνται βελτιστοποιήσεις στον παραγόμενο κώδικα. Ο μηχανισμός είναι γενικός, επιτρέποντας στον χρήστη να επιλέξει αν η ανάλυση θα πραγματοποιηθεί πάνω στο AST ή στο CFG. Ο μηχανισμός εφαρμόστηκε σε διάφορες περιπτώσεις που αφορούν συχνά λάθη ή παραλείψεις στη χρήση του OpenMP και είναι σε θέση να εντοπίσει προβληματικές καταστάσεις, όπως ακατάλληλες αναγνώσεις ή εγγραφές σε μεταβλητές. Περισσότερες πληροφορίες για τον OMPi και την ανάλυση ροής δεδομένων στα κεφάλαια 3 και 4 αντίστοιχα.

1.5 Δομή Διπλωματικής Εργασίας

Η παρούσα διπλωματική εργασία οργανώνεται ως εξής:

- Κεφάλαιο 2: Παρουσίαση και ανάλυση του προτύπου OpenMP. Επεξήγηση των δομών και οδηγιών που αποτελούν το πρότυπο.
- Κεφάλαιο 3: Παρουσίαση και περιγραφή του παραλληλοποιητικού μεταφραστή OMPi. Αναφορά στις λειτουργίες του μεταφραστή καθώς και στις δομές που χρησιμοποιεί, όπως το Abstract Syntax Tree (AST) και Control Flow Graph (CFG). Παραδείγματα χρήσης.
- Κεφάλαιο 4: Εισαγωγή στην ανάλυση ροής δεδομένων και περιγραφή του μηχανισμού που δημιουργήθηκε. Περιγραφή και ανάλυση των προβλημάτων που επιλύει ο μηχανισμός. Επεξήγηση του τρόπου χρήσης από τον μεταφραστή OMPi.
- Κεφάλαιο 5: Παρουσίαση και επεξήγηση παραδειγμάτων λειτουργίας του μηχανισμού ανάλυσης ροής δεδομένων.

- Κεφάλαιο 6: Σύνοψη διπλωματικής εργασίας, αναφορά σε μελλοντικές βελτιστοποιήσεις του μεταφραστή OMPi.

Κεφάλαιο 2.

Το πρότυπο OpenMP

2.1 Εισαγωγή στο OpenMP

Το OpenMP (open multi-programming) [2] είναι μια διεπαφή (API) που υποστηρίζει παράλληλο προγραμματισμό κοινόχρηστης μνήμης στις γλώσσες C, C++ και Fortran. Αποτελείται από οδηγίες προς το μεταφραστή, ρουτίνες βιβλιοθήκης και μεταβλητές περιβάλλοντος που επηρεάζουν τη συμπεριφορά χρόνου εκτέλεσης.

Οι πρώτες προδιαγραφές εκδόθηκαν τον Οκτώβρη του 1997 για Fortran και τον επόμενο χρόνο για C και C++. Το 2005 όλες αυτές οι προδιαγραφές ενοποιήθηκαν σε μία έκδοση, την 2.5, που κάλυπτε τις γλώσσες Fortran, C και C++. Μέχρι τότε, το OpenMP όριζε απλούς τρόπους παραλληλοποίησης συχνών βρόγχων στους οποίους το πλήθος των επαναλήψεων είναι γνωστό εκ των προτέρων. Αυτός ο περιορισμός οδήγησε σε μια προσπάθεια να εισαχθεί ο παραλληλισμός με tasks. Έτσι το 2008 έγινε διαθέσιμη η έκδοση 3.0 όπου εισήγαγε τεχνικές περα από την παραλληλοποίηση βασικών βρόγχων.

Τον Ιούλιο του 2013 παρουσιάστηκε η έκδοση 4.0 η οποία περιλαμβάνει κάποιες προσθήκες. Παρέχει υποστήριξη για επιταχυντές και νέες ιδιότητες όπως το thread affinity.

Η τρέχουσα έκδοση είναι η 6.0 και παρουσιάστηκε τον Νοέμβριο του 2024.

2.2 Προγραμματιστικό Μοντέλο OpenMP

Το πρότυπο του OpenMP αφορά την παραλληλία με χρήση νημάτων σε συστήματα κοινής μνήμης. Το προγραμματιστικό του μοντέλο βασίζεται στο γνωστό μοντέλο fork-join που συναντάται στις διεργασίες. Το πρόγραμμα εκτελείται σειριακά από το κύριο νήμα (master thread) και όταν συναντηθεί μια περιοχή παραλληλίας τότε δημιουργείται μια ομάδα νημάτων, το πλήθος των οποίων καθορίζεται από τον προγραμματιστή είτε

δυναμικά, είτε στατικά από μια μεταβλητή περιβάλλοντος. Η ομάδα νημάτων εκτελεί την περιοχή παραλληλίας, ενώ το κύριο νήμα περιμένει να τελειώσει η εκτέλεση για να συνεχίσει την εκτέλεση του σειριακού κώδικα. Η διαδικασία αυτή επαναλαμβάνεται όσες φορές χρειαστεί, για κάθε περιοχή παραλληλίας. Το πρότυπο του OpenMP προσφέρει αρκετές ιδιότητες για την κατάλληλη παραμετροποίηση του κώδικα μιας παράλληλης περιοχής. Στην επόμενη υποενότητα θα δούμε κάποιες από αυτές τις ιδιότητες.

2.3 Εντολές OpenMP

Στο παρόν κεφάλαιο περιγράφεται το συντακτικό του OpenMP. Θα δούμε τι είναι μια παράλληλη περιοχή καθώς επίσης και τεχνικές συγχρονισμού των νημάτων, με χρήση οδηγιών (directives) [7].

2.3.1 Οδηγίες OpenMP

Οι οδηγίες που θα δούμε είναι για τις γλώσσες προγραμματισμού C/C++.

Ένα παράδειγμα που μας δείχνει το συντακτικό μιας οδηγίας είναι:

```
➢ #pragma omp <όνομα οδηγίας> <φράση>… <newline>
```

Ισχύουν τα εξής:

- Μετά το #pragma omp είναι υποχρεωτικό το όνομα της οδηγίας
- Μετά την οδηγία τοποθετούνται, προαιρετικά, οι φράσεις οδηγιών οι οποίες ορίζουν τις συνθήκες υπό τις οποίες θα εκτελεστεί η οδηγία. Εάν δεν έχει τοποθετηθεί καμία οδηγία, τότε οι συνθήκες αυτές καθορίζονται στον χρόνο εκτέλεσης του προγράμματος.
- Μετά από την οδηγία OpenMP απαιτείται νέα γραμμή (newline) στο τέλος της.

2.3.2 Παράλληλες περιοχές

Με την έννοια παράλληλη περιοχή υποδηλώνουμε ένα μπλοκ κώδικα το οποίο θα εκτελεστεί από μια ομάδα νημάτων. Η δήλωση της παράλληλης περιοχής γίνεται με την οδηγία parallel όπως φαίνεται παρακάτω:

```
➢ #pragma omp parallel <φράση … >
          Τμήμα κώδικα
```

Το τμήμα κώδικα είναι ο κώδικας που θα τρέξει η ομάδα νημάτων. Καθώς τρέχει το κύριο νήμα (master thread) θα συναντήσει την οδηγία parallel και θα φτιάξει την ομάδα νημάτων στην οποία συμμετέχει ως γονέας. Το μέγεθος της ομάδας νημάτων καθορίζεται από τον χρήστη ή από το περιβάλλον εκτέλεσης.

Κατά την εκτέλεση είναι πολύ πιθανό ένα ή περισσότερα νήματα της ομάδας να ολοκληρώσουν την εκτέλεσή τους νωρίτερα. Αυτό οφείλεται στο γεγονός ότι ο κάθε επεξεργαστής για μια χρονική στιγμή μπορεί να εκτελεί διαφορετικού είδους εργασίες, η ύπαρξη των οποίων μεταβάλλει το πότε αυτός θα γίνει διαθέσιμος για το νήμα. Έτσι, για τον συγχρονισμό των νημάτων στο τέλος της παράλληλης περιοχής υπονοείται ένα φράγμα (barrier) το οποίο αναγκάζει κάθε νήμα να περιμένει τον τερματισμό και των υπολοίπων νημάτων. Όταν η εκτέλεση ολοκληρωθεί, το φράγμα λύνεται και η ομάδα νημάτων καταστρέφεται, ενώ το κύριο νήμα συνεχίζει την εκτέλεση του υπολοίπου προγράμματος.

2.3.3 Οδηγίες Διαμοιρασμού Εργασίας

Μια σημαντική δυνατότητα που προσφέρει το OpenMP είναι ο διαμοιρασμός εργασίας μεταξύ των νημάτων εντός μίας παράλληλης περιοχής. Έτσι, μέσα στην παράλληλη περιοχή ο χρήστης μπορεί να χρησιμοποιήσει τις εξής οδηγίες:

- **for.** Αφορά την κατανομή των επαναλήψεων ενός βρόγχου for στα νήματα. Ακριβώς μετά την εντολή αυτή, ακολουθεί ο βρόχος for που πρόκειται να παραλληλοποιηθεί.
  Η σύνταξη της οδηγίας είναι:

  ```
  #pragma omp for <φράση … >
  ```

- **sections**. Αφορά τον καταμερισμό των τμημάτων κώδικα που δηλώνονται, ώστε αυτά να διαμοιραστούν σε διαφορετικά νήματα. Η σύνταξη είναι η εξής:

  ```
  #pragma omp sections <φράση … > <νέα-γραμμή>
  {
     #pragma omp section
         Τμήμα κώδικα
         ...
  }
  ```

- **single/master.** Το πρώτο νήμα που θα συναντήσει την οδηγία single ή το κύριο νήμα στην περίπτωση της οδηγίας master εκτελεί το τμήμα κώδικα που ακολουθεί με τα υπόλοιπα να την προσπερνούν. Ακολουθεί η σύνταξη της οδηγίας single και master.

```
➢  #pragma omp [single | master]  <φράση … > <νέα-γραμμή>
              Τμήμα κώδικα
```

2.3.4 Οδηγίες συγχρονισμού

Οι οδηγίες αυτές χρησιμοποιούνται για τον συγχρονισμό των νημάτων κατά την παράλληλη εκτέλεση έτσι ώστε να διασφαλιστεί η συνέπεια των δεδομένων και η σωστή λειτουργία του προγράμματος.

- **atomic** : Χρησιμοποιείται για αδιαίρετη πράξη σε μια μεταβλητή. Εξασφαλίζει ότι στη μεταβλητή που πρόκειται να τροποποιηθεί δεν παρεμβάλλεται κάποιο άλλο νήμα.
- **barrier** : Ορίζει ένα φράγμα συγχρονισμού των νημάτων στο σημείο που τοποθετείται. Τα νήματα που καταφτάνουν στο σημείο αυτό περιμένουν τα υπόλοιπα της ομάδας, ώστε να συνεχιστεί η εκτέλεση με τις τιμές των κοινόχρηστων μεταβλητών ίδιες για όλα τα νήματα.
- **critical** : Παρόμοια με την οδηγία atomic αλλά για τμήμα κώδικα που πρέπει να εκτελεστεί από ένα νήμα τη φορά. Τα νήματα που φτάνουν στην κρίσιμη περιοχή περιμένουν την εκτέλεση από το νήμα που βρίσκεται ήδη σε αυτήν.

2.3.5 Φράσεις Οδηγιών

Οι φράσεις οδηγιών τοποθετούνται στη δήλωση μιας οδηγίας OpenMP μετά το όνομά της. Ο στόχος τους είναι να ορίσουν τις συνθήκες από τις οποίες θα εκτελεστεί η εκάστοτε οδηγία ή να καθορίσουν τη συμπεριφορά των νημάτων πριν, κατά τη διάρκεια και μετά την εκτέλεση του τμήματος κώδικα. Κάποιες βασικές φράσεις που χρησιμοποιούνται τις περισσότερες φορές είναι οι εξής:

- **if** (συνθήκη). Εξετάζεται αν ισχύει η συνθήκη και αν είναι αληθής τότε εκτελείται η οδηγία που περιέχει τη φράση.
- **shared** (λίστα μεταβλητών). Χρησιμοποιείται για να ορίσει τα αντικείμενα της λίστας μεταβλητών ως κοινόχρηστα μεταξύ των νημάτων της ομάδας.
- **private** (λίστα μεταβλητών). Χρησιμοποιείται για να ορίσει ως ιδιωτικά τα αντικείμενα της λίστας μεταβλητών για κάθε νήμα της ομάδας.
- **firstprivate** (λίστα μεταβλητών). Παρόμοια χρήση με το private. Η μόνη διαφορά είναι ότι υπάρχει και αρχικοποίηση των μεταβλητών της λίστας με την τιμή που έχει εκτός της παράλληλης περιοχής.

- **lastprivate** (λίστα μεταβλητών). Το lastprivate λειτουργεί όπως και το private, με την μόνη διαφορά ότι στο τέλος της περιοχής παραλληλίας οι τιμές των αντικείμενων της λίστας μεταβλητών ενημερώνονται.
- **nowait**. Χρησιμοποιείται έτσι ώστε τα νήματα να αγνοήσουν το φράγμα μετά την αντίστοιχη οδηγία.
- **num_threads** (πλήθος νημάτων). Χρησιμοποιείται για να ορίσει το πλήθος των νημάτων που θα συμμετάσχουν σε μια παράλληλη περιοχή.
- **schedule** (τύπος, chunk size). Το schedule χρησιμοποιείται σε συνδυασμό με την οδηγία for και ορίζουν την πολιτική διαμοιρασμού των επαναλήψεων του βρόγχου στα νήματα. Έτσι, υπάρχουν πολιτικές static, για στατικό διαμοιρασμό του βρόγχου σε τμήματα chunk size ή πλήθους νημάτων, dynamic για δυναμικό διαμοιρασμό, με την έννοια ότι το επόμενο τμήμα μεγέθους chunk size θα το αναλάβει όποιο νήμα είναι διαθέσιμο και guided, όπου ο διαμοιρασμός γίνεται παρόμοια με το dynamic αλλά το μέγεθος του κόκκου μειώνεται εκθετικά.
- **reduction** (πράξη : λίστα μεταβλητών). Χρησιμοποιείται για την εκτέλεση μιας πράξης σε μια κοινόχρηστη μεταβλητή και εξασφαλίζει τη συνέπεια των δεδομένων, χωρίς να απασχολείται με αυτό το κομμάτι ο προγραμματιστής.

2.3.6 Tasks

Η οδηγία task επιτρέπει στον προγραμματιστή να ορίσει μια περιοχή κώδικα που μπορεί να εκτελεστεί από οποιοδήποτε νήμα, οποιαδήποτε χρονική στιγμή. Αυτό είναι ιδιαίτερα σημαντικό, καθώς το συγκεκριμένο τμήμα θα εκτελεστεί μόλις ένας πυρήνας γίνει διαθέσιμος ή ακόμα και παράλληλα με άλλες παράλληλες περιοχές. Αυτός ο τρόπος παραλληλισμού είναι ιδανικός για εφαρμογές όπου η εξισορρόπηση φόρτου μεταξύ των νημάτων αποτελεί περιορισμό. Επιπλέον, τα tasks μπορούν να μεταφερθούν μεταξύ νημάτων, δηλαδή ένα νήμα μπορεί να ξεκινήσει την εκτέλεση και, σε κάποιο σημείο, να την αναλάβει ένα άλλο.

2.3.7 Συναρτήσεις Βιβλιοθήκης OpenMP

Το OpenMP προσφέρει ορισμένες συναρτήσεις που μπορεί να χρησιμοποιήσει ο προγραμματιστής από τη βιβλιοθήκη του χρόνου εκτέλεσης (omp.h). Αυτές χρησιμοποιούνται για να ορίσουν δυναμικά το πλήθος των νημάτων με βάση τους

διαθέσιμους πόρους, για συγχρονισμό των νημάτων με χρήση κλειδαριών, για χρονομετρήσεις κ.α. Κάποιες από αυτές είναι:

- **omp_get_num_threads:** Επιστρέφει το πλήθος των νημάτων που χρησιμοποιούνται.
- **omp_set_num_threads:** Ορίζει το πλήθος των νημάτων που θα εκτελέσουν μια παράλληλη περιοχή.
- **omp_get_thread_num:** Επιστρέφει τον αριθμό του τρέχοντος νήματος.
- **omp_get_num_procs:** Επιστρέφει το πλήθος των διαθέσιμων πυρήνων που βλέπει το σύστημα.
- **omp_init_lock:** Αρχικοποιεί μια απλή κλειδαριά.
- **omp_destroy_lock:** Αφαιρεί μια κλειδαριά.

2.3.8 Implicitly determined μεταβλητές

Ένα σημαντικό στοιχείο στο OpenMP είναι οι Implicitly determined μεταβλητές. Κάθε οδηγία πρέπει να γνωρίζει πώς να αναγνωρίζει τις μεταβλητές. Οι explicitly determined μεταβλητές βρίσκονται σε συγκεκριμένες φράσεις (όπως firstprivate), επιτρέποντας στην οδηγία να καθορίσει τη χρήση τους. Αντίθετα, οι implicitly determined μεταβλητές δεν ανήκουν σε κάποια φράση, αλλά η οδηγία εξακολουθεί να χρειάζεται έναν τρόπο να τις διαχειριστεί. Οι κανόνες για αυτές τις μεταβλητές διαφέρουν ανάλογα με την οδηγία. Για παράδειγμα, η οδηγία parallel τις δηλώνει ως shared. Στην περίπτωση της task, οι μεταβλητές που έχουν οριστεί πριν από αυτήν και δεν βρίσκονται ήδη σε κάποια φράση δηλώνονται ως firstprivate, αντίθετα με την parallel, η οποία τις θεωρεί shared. Περισσότερες πληροφορίες για την υλοποίηση και παραδείγματα λειτουργίας θα παρουσιαστούν στα κεφάλαια 4 και 5 αντίστοιχα.

Κεφάλαιο 3.

Ο Μεταφραστής OMPi

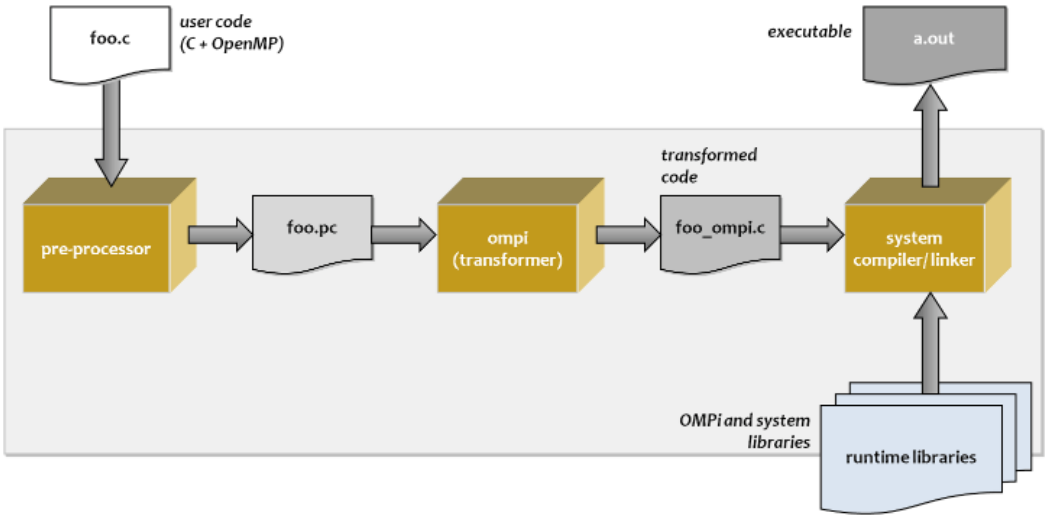
3.1 Η δομή του OMPi

Ο μεταφραστής OMPi [5] είναι ένας παραλληλοποιητικός μεταφραστής γλώσσας C για το πρότυπο OpenMP. Είναι διαχωρισμένος σε δύο βασικά τμήματα, αυτό του μεταγλωττιστή (compiler) και αυτό της υποστήριξης εκτέλεσης (runtime). Το πρώτο, χρησιμοποιείται για την μετατροπή του πηγαίου κώδικα, που περιλαμβάνει οδηγίες OpenMP, σε πολυνηματικό κώδικα C. Στον παραγόμενο κώδικα όλες οι οδηγίες αντικαθιστώνται από κλήσεις σε ρουτίνες του τμήματος υποστήριξης εκτέλεσης. Ο πολυνηματικός κώδικας μπορεί να χρησιμοποιεί διαφορετικό είδος νημάτων για παράλληλη εκτέλεση.

3.2 Ροή Μετάφρασης του OMPi

Το αρχικό βήμα της μεταγλώττισης περιλαμβάνει την διαδικασία της συντακτικής ανάλυσης, η οποία είναι κοινή μεταξύ όλων των μεταγλωττιστών. Η συντακτική ανάλυση περιλαμβάνει σε πρώτο στάδιο τη λεκτική ανάλυση του κώδικα και, έπειτα, τη διάσχισή του ώστε να δημιουργηθεί το συντακτικό δέντρο. Αφού παραχθεί το συντακτικό δέντρο, τότε αυτό διασχίζεται μία φορά ώστε να αντικατασταθεί κάθε οδηγία OpenMP με τις αντίστοιχες κλήσεις του συστήματος υποστήριξης εκτέλεσης. Το αποτέλεσμα των αντικαταστάσεων είναι ένας πηγαίος κώδικας σε C, ο οποίος βασίζεται μόνο σε βιβλιοθήκες υποστήριξης εκτέλεσης του μεταφραστή. Το τελευταίο βήμα της διαδικασίας της μεταγλώττισης είναι η μετάφραση του παραγόμενου πηγαίου κώδικα C και η σύνδεση των απαραίτητων βιβλιοθηκών υποστήριξης εκτέλεσης με αυτόν, ώστε να παραχθεί το εκτελέσιμο αρχείο.

Η διαδικασία απεικονίζεται στο Σχήμα 3.1.



Σχημα 3.1 ροή μετάφρασης στον OMPi

3.3 Abstract Syntax Tree (AST)

Το abstract syntax tree (AST) είναι μια δομή δεδομένων που χρησιμοποιείται κυρίως στους μεταφραστές (compilers) για να αναπαραστήσει τη συντακτική δομή ενός προγράμματος. Είναι μια αφηρημένη αναπαράσταση του πηγαίου κώδικα, που αφαιρεί λεπτομέρειες όπως παρενθέσεις ή μη απαραίτητα σύμβολα και εστιάζει στη λογική δομή του προγράμματος. Ουσιαστικά, το AST είναι ένα δέντρο όπου κάθε κόμβος αντιπροσωπεύει ένα στοιχείο για τον κώδικα, όπως δηλώσεις μεταβλητών ή εκφράσεις. Βασικά χαρακτηριστικά του AST είναι:

- **Αφαιρετικότητα**. Το AST δεν περιλαμβάνει όλους τους κόμβους της γραμματικής, αλλά μόνο τους ουσιώδεις που εκφράζουν τη λογική του προγράμματος.
- **Ιεραρχία.** Το AST είναι δομημένο ως δέντρο, όπου κάθε κόμβος αναπαριστά μια συντακτική δομή.
- **Δομή δέντρου**. Κάθε κόμβος του AST αντιπροσωπεύει ένα στοιχείο του κώδικα του προγράμματος και μπορεί να έχει παιδιά που αναπαριστούν υπο-στοιχεία αυτού του κώδικα. Για παράδειγμα, στο Σχήμα 3.2, ο κόμβος EXPRESSION έχει ως παιδί έναν κόμβο που αναπαριστά την ενέργεια της ανάθεσης τιμής σε μια μεταβλητή. Αυτός ο κόμβος, με τη σειρά του, έχει δύο παιδιά: το όνομα της μεταβλητής και την τιμή που της ανατίθεται.

- **Βελτιστοποίηση**. Το AST χρησιμοποιείται ευρέως στη βελτιστοποίηση και ανάλυση κώδικα από μεταγλωττιστές.

Χρησιμοποιώντας το AST μπορούμε να αντλήσουμε πληροφορίες για την ροή των δεδομένων ενός προγράμματος και να βγάλουμε χρήσιμα αποτελέσματα.

3.4 Control Flow Graph (CFG)

Το Control Flow Graph (CFG) είναι μια αναπαράσταση της ροής ελέγχου ενός προγράμματος [2]. Απεικονίζει τα διάφορα τμήματα του προγράμματος και τις πιθανές διαδρομές που μπορεί να ακολουθήσει η εκτέλεση, με βάση τις συνθήκες και τις δομές ελέγχου του κώδικα. Πρόκειται για ένα κατευθυνόμενο γράφημα με κόμβους και ακμές. Οι κόμβοι του, αντιπροσωπεύουν εντολές ή μπλοκ εντολών, ενώ οι ακμές μεταξύ αυτών, δείχνουν τις πιθανές διαδρομές εκτέλεσης του προγράμματος.

Βασικές διαφορές του CFG και AST είναι:

- **Αναπαράσταση**. Το CFG αναπαριστά τη ροή ελέγχου ενός προγράμματος, δηλαδή πώς οι εντολές εκτελούνται και ποια διαδρομή ακολουθεί ο έλεγχος, σε αντίθεση με το AST που αναπαριστά τη συντακτική δομή ενός προγράμματος, απεικονίζοντας την ιεραρχία των εντολών χωρίς να ασχολείται με τη ροή εκτέλεσης.
- **Δομή**. Το CFG αποτελείται από κόμβους για εντολές και δομές ελέγχου (όπως if,while) και ακμές για την ροή ελέγχου, ενώ το AST αποτελείται από κόμβους για συντακτικά στοιχεία του προγράμματος, όπως μεταβλητές, εκφράσεις και δηλώσεις.
- **Χρήση**. Το CFG χρησιμοποιείται κυρίως για ανάλυση και βελτιστοποίηση ροής ελέγχου, ανάλυση σφαλμάτων και στατική ανάλυση, σε αντίθεση με το AST που χρησιμοποιείται κυρίως για την αναπαράσταση του προγράμματος με σκοπό τη μεταγλώττιση ή ανάλυση της σύνταξης.

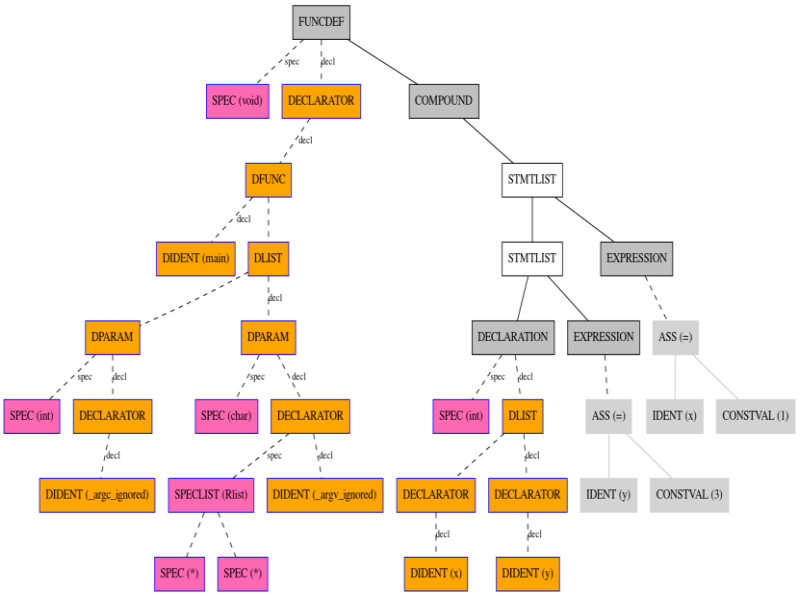
3.5 Παραδείγματα για AST και CFG

Για να κατανοήσουμε καλύτερα αυτές τις δομές, αλλά και ταυτόχρονα να αναδείξουμε τις διαφορές τους, θα εξετάσουμε τα προγράμματα 3.1 και 3.2, όπου θα απεικονίσουμε τις δύο δομές και θα εστιάσουμε στον τρόπο λειτουργίας τους.

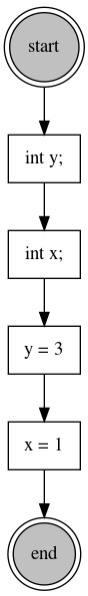
Πρόγραμμα 3.1: Απλό πρόγραμμα για τις δομές AST και CFG

```
1  void main( )
2  {
3    int x,y;
4    y=3;
5    x=1;
6    ...
7  }
```

Κοιτώντας τη δομή του AST στο Σχήμα 3.2 και του CFG στο Σχήμα 3.3, επιβεβαιώνουμε τις αρχικές παρατηρήσεις μας. Στο AST, ξεκινάμε με έναν κόμβο FUNCDEF, ο οποίος δηλώνει τον ορισμό της συνάρτησης, ενώ στο CFG, ξεκινάμε από έναν αρχικό κόμβο με την ονομασία start, ο οποίος δεν περιέχει τον ορισμό της συνάρτησης main(). Στο AST, συναντάμε τρεις βασικούς κόμβους: ο κόμβος SPEC κρατάει τον τύπο της συνάρτησης ή της μεταβλητής που δηλώνεται, ο DECLARATOR αφορά τις δηλώσεις μεταβλητών (αν και στην περίπτωση αυτή, δεν έχουμε ορίσματα στην main(), οπότε δεν περιέχει κάποια σημαντική πληροφορία), και ο COMPOUND που περιλαμβάνει όλη την πληροφορία μέσα στην main(). Αντίθετα, στο CFG δεν υπάρχουν αντίστοιχοι κόμβοι. Η πληροφορία που περιέχεται στο COMPOUND του AST βρίσκεται στο CFG μέσω των κόμβων DECLARATOR για τις δηλώσεις (int x, y;) και expression για τις εκφράσεις (y = 3; και x = 1;). Εδώ φαίνεται η πιο λεπτομερής και σύνθετη δομή του AST. Για παράδειγμα, για την έκφραση x = 1; το AST απαιτεί έναν κόμβο EXPRESSION, ο οποίος οδηγεί σε έναν κόμβο ASSIGNMENT (για να δείξει ότι πρόκειται για έκφραση ανάθεσης), και στη συνέχεια σπάει σε δύο κόμβους για το όνομα της μεταβλητής και την τιμή της. Αντίθετα, στο CFG η έκφραση x = 1; αποτυπώνεται απευθείας σε έναν κόμβο χωρίς επιπλέον λεπτομέρειες. Ωστόσο, αξίζει να σημειώσουμε ότι το CFG κρατάει πληροφορίες για τους κόμβους του AST από τους οποίους προήλθε, κάτι που θα μας φανεί χρήσιμο στην ανάλυση ροής δεδομένων, την οποία θα εξετάσουμε στο επόμενο κεφάλαιο.



Σχήμα 3.2: Απεικόνιση του AST



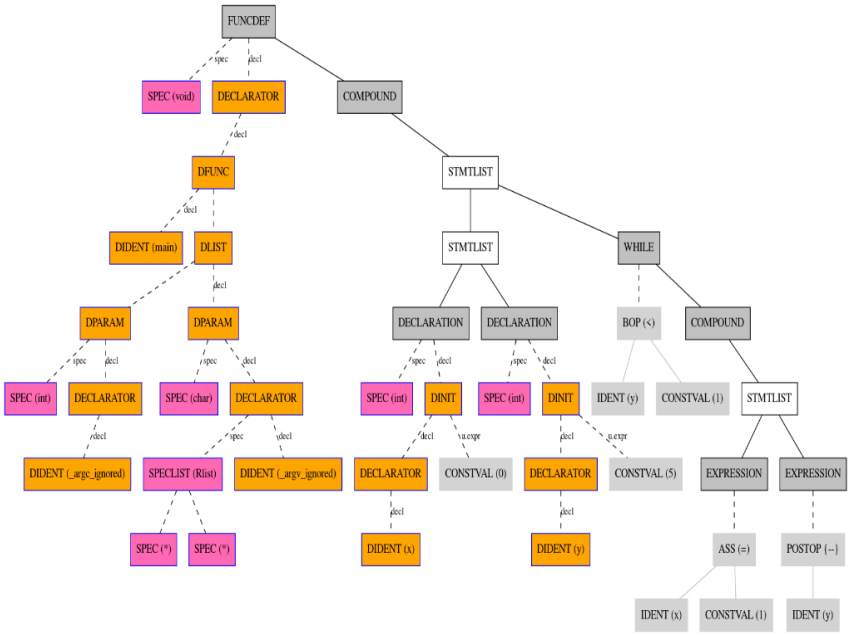
Σχήμα 3.3: Απεικόνιση του CFG

Στο επόμενο πρόγραμμα 3.2 θα δούμε έναν πιο σύνθετο κώδικα που θα μας βοηθήσει να κατανοήσουμε καλύτερα τη βασική διαφορά μεταξύ των δομών AST και CFG.

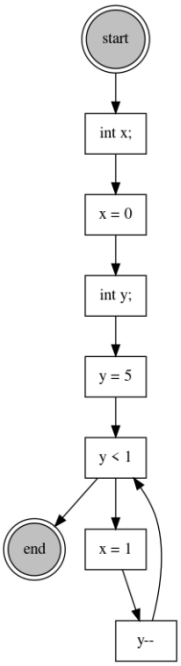
Πρόγραμμα 3.2: Σύνθετο πρόγραμμα για τις δομές AST και CFG

```
1  void main( )
2  {
3    int x=0;
4    int y=5;
5    while(y<1)
6    {
7      x=1;
8      y--;
9      ...
10    }
11 }
```

Σε αυτό το παράδειγμα, υπάρχει ένας βρόχος while με δύο πιθανές διαδρομές εκτέλεσης, ανάλογα με το αν η συνθήκη είναι αληθής ή όχι. Στο Σχήμα 3.4, εστιάζουμε στον κόμβο while, καθώς οι υπόλοιποι κόμβοι έχουν ήδη παρουσιαστεί στο προηγούμενο παράδειγμα. Παρατηρούμε ότι το AST είναι μια στατική δομή που ασχολείται μόνο με τη σύνταξη του κώδικα, χωρίς να υπάρχει ροή ελέγχου, ενώ το CFG στο Σχήμα 3.5 εστιάζει στη ροή εκτέλεσης του κώδικα. Στο Σχήμα 3.5 και στον κόμβο y<1 υπάρχουν δύο εξερχόμενα βέλη που δείχνουν τις δύο διαδρομές του προγράμματος. Στον κόμβο y--, ο βρόχος τελειώνει και επιστρέφει για νέο έλεγχο. Η βασική διαφορά των AST και CFG που αναφέραμε προηγουμένως φαίνεται εδώ, όπου το CFG παρέχει περισσότερες πληροφορίες για τη ροή δεδομένων, όπως πού και για πόσο χρόνο χρησιμοποιείται μια μεταβλητή, κάτι που είναι κρίσιμο για την ανάλυση.



Σχήμα 3.4: Απεικόνιση του AST



Σχήμα 3.5: Απεικόνιση του CFG

Κεφάλαιο 4.

Ανάλυση Ροής Δεδομένων

4.1 Εισαγωγή στην ανάλυση ροής δεδομένων

Η ανάλυση ροής δεδομένων [2] είναι ένα σύνολο από τεχνικές που χρησιμοποιούνται στους μεταφραστές και όχι μόνο για την εξαγωγή πληροφοριών σχετικά με τη χρήση των μεταβλητών και των εκφράσεων ενός προγράμματος. Στόχος της ανάλυσης ροής δεδομένων είναι να επιτευχθούν κάποιες βελτιστοποιήσεις ή ακόμα και ο έλεγχος σφαλμάτων.

Χαρακτηριστικά προβλήματα που λύνονται με την ανάλυση ροής δεδομένων είναι:

- Liveness Analysis
- Απαλοιφή νεκρού κώδικα
- Διαθέσιμες εκφράσεις
- Πρόβλημα Εγγραφής/Ανάγνωσης
- Πρόβλημα Firstprivate/Write
- Πρόβλημα Private/Read

Τα περισσότερα θα τα δούμε αναλυτικά παρακάτω.

4.2 Περιγραφή του μηχανισμού

Στο πλαίσιο της παρούσας διπλωματικής εργασίας αναπτύχθηκε ένας γενικός μηχανισμός ανάλυσης ροής δεδομένων, ο οποίος μπορεί να λειτουργήσει είτε με βάση το Abstract Syntax Tree (AST), είτε με βάση το Control Flow Graph (CFG), ανάλογα με τις απαιτήσεις του εκάστοτε προβλήματος. Παραδοσιακά, η ανάλυση ροής δεδομένων

υλοποιείται πάνω στο CFG, καθώς αυτό το επίπεδο αναπαράστασης αποτυπώνει με ακρίβεια τη ροή εκτέλεσης ενός προγράμματος. Στο πλαίσιο αυτής της εργασίας, ωστόσο, έγινε το επιπλέον βήμα να υποστηρίζεται και η ανάλυση σε επίπεδο AST, δίνοντας έτσι μεγαλύτερη ευελιξία και κάλυψη σε περιπτώσεις όπου η επεξεργασία μπορεί να γίνει αποτελεσματικότερα σε αυτό το επίπεδο.

Η ανάλυση ροής δεδομένων βασίζεται στην κατασκευή και επεξεργασία συνόλων για μεταβλητές που είναι σημαντικές για το πρόβλημα που επιδιώκουμε να λύσουμε. Ο τύπος των μεταβλητών αυτών καθώς και ο τρόπος που επεξεργάζονται διαφέρει ανάλογα με τις ανάγκες της ανάλυσης και τον ορισμό του προβλήματος.

Η διαδικασία ανάλυσης ξεκινά από τη ρίζα του AST ή τον αρχικό κόμβο του CFG και επισκέπτεται αναδρομικά όλους τους υπόλοιπους κόμβους. Κατά τη διάρκεια αυτής της αναδρομικής επίσκεψης, όταν εντοπίζεται ένας κόμβος που αντιστοιχεί σε expression, declaration ή statement, καλούνται ειδικές συναρτήσεις ανάκλησης (callback functions), οι οποίες επιτελούν την απαραίτητη επεξεργασία. Σκοπός των συναρτήσεων αυτών είναι η συναρμολόγηση των συνόλων που απαιτούνται για την επίλυση του εκάστοτε προβλήματος.

Στις επόμενες ενότητες θα αναλυθεί διεξοδικότερα ο μηχανισμός εφαρμόζοντάς τον σε συγκεκριμένα προβλήματα.

4.3 Προβλήματα που μας απασχόλησαν

Αρχικά, υλοποιήθηκαν μηχανισμοί για το πρόβλημα εγγραφής /ανάγνωσης, δηλαδή για το πως χρησιμοποιούνται οι μεταβλητές κατά την εκτέλεση(μόνο εγγραφή, μόνο ανάγνωσή ή και τα δυο μαζί). Επιπλέον υλοποιήθηκαν δύο επιπρόσθετοι μηχανισμοί: ο πρώτος αφορά τον εντοπισμό των μεταβλητών που έχουν δηλωθεί ως firstprivate (υποενότητα 2.3.5) και στη συνέχεια υφίστανται εγγραφή, ενώ ο δεύτερος αφορά τις μεταβλητές που έχουν δηλωθεί ως private (υποενότητα 2.3.5) και στη συνέχεια υφίστανται ανάγνωση.

4.3.1 Το Πρόβλημα Εγγραφής/Ανάγνωσης

Έστω ένα μπλοκ κώδικα B και μια μεταβλητή x. Θέλουμε να ανακαλύψουμε ποια από τις παρακάτω 4 περιπτώσεις ισχύει:

i. Η μεταβλητή x έχει υποστεί μόνο ανάγνωση.

ii. Η μεταβλητή x έχει υποστεί μόνο εγγραφή.

iii. Η μεταβλητή x δέχεται πρώτα εγγραφή και έπειτα ανάγνωση.

iv. Η μεταβλητή x δέχεται πρώτα ανάγνωση και έπειτα εγγραφή.

Θα χρειαστούν τα εξής σύνολα:

R(B) = { Μεταβλητές που υφίστανται ανάγνωση στο B }

W(B) = { Μεταβλητές που υφίστανται εγγραφή στο B }

F(B) = { Σύνολο που κρατάει το τελικό αποτέλεσμα }

Λύνουμε ως εξής:

i. Στην πρώτη περίπτωση ψάχνουμε τις μεταβλητές που έχουν υποστεί μόνο ανάγνωση. Εφόσον έχουμε τα σύνολα R(B) και W(B) αρκεί να κάνουμε την πράξη: F(B) = R(B) – W(B) η οποία θα μας επιστρέψει το ζητούμενο.

ii. Αντίστοιχα για την δεύτερη περίπτωση αρκεί να κάνουμε το αντίθετο, δηλαδή: F(B) = W(B) – R(B) η οποία θα μας επιστρέψει το ζητούμενο.

iii. Στην συγκεκριμένη περίπτωση εξετάζουμε μια μεταβλητή η οποία την δεδομένη στιγμή διαβάζεται. Έστω λοιπόν ότι η x την δεδομένη στιγμή διαβάζεται, ελέγχουμε αν $x \in W(B)$ και ταυτοχρόνως $x \notin R(B)$.

iv. Στην συγκεκριμένη περίπτωση εξετάζουμε μια μεταβλητή η οποία την δεδομένη στιγμή γράφεται. Έστω λοιπόν ότι η x την δεδομένη στιγμή γράφεται, ελέγχουμε αν $x \in R(B)$ και ταυτοχρόνως $x \notin W(B)$.

Λύνοντας αυτό το πρόβλημα, έχουμε μια συνολική εικόνα για την ροή των δεδομένων μας στο πρόγραμμα. Το κύριο ζήτημα προκύπτει όταν έχουμε συγχρονισμένη πρόσβαση (προσπέλαση των ίδιων δεδομένων) στα δεδομένα από διεργασίες ή νήματα. Η ανάλυση αυτή, μας βοηθάει να εντοπίσουμε και να αποτρέψουμε τέτοιες ανεπιθύμητες προσπελάσεις, βελτιώνοντας την αξιοπιστία και την απόδοση του προγράμματος. Επιπλέον, η ανάλυση συμβάλλει στον εντοπισμό νεκρών μεταβλητών (dead variables). Αν μια μεταβλητή χρησιμοποιείται μόνο για εγγραφή (write-only), μπορούμε να την αφαιρέσουμε καθώς δεν διαβάζεται από κανένα νήμα και δεν χρησιμοποιείται πουθενά. Ένα ακόμη σημαντικό πρόβλημα που λύνεται είναι όταν εντοπίσουμε μεταβλητές που εγγράφονται πριν διαβαστούν. Τότε σε εκτέλεση κώδικα σε συνδεδεμένες συσκευές, που συνήθως εμπεριέχει χρονοβόρες

μεταφορές δεδομένων, οι αρχικές τιμές των μεταβλητών αυτών δεν χρειάζεται να μεταφερθούν προς την συσκευή. Μια τέτοια βελτιστοποίηση έχει ακόμα μεγαλύτερα οφέλη αν οι συσκευές είναι απομακρυσμένες [3].

4.3.2 Το πρόβλημα του firstprivate clause

Το πρόβλημα διατυπώνεται ως εξής :

Σε αυτήν την περίπτωση θέλουμε να ελέγξουμε ποιες μεταβλητές έχουν οριστεί ως firstprivate σε κάποιο construct και στην συνέχεια γράφονται πριν διαβαστούν, όπως φαίνεται στο πρόγραμμα 4.1.

Πρόγραμμα 4.1: Το πρόβλημα του firstprivate clause

```
1  #pragma omp parallel firstprivate(x)
2  {
3    x=1;
4    ...
5  }
```

Το firstprivate clause δημιουργεί ένα τοπικό αντίγραφο της μεταβλητής το οποίο αρχικοποιείται. Εφόσον η πρώτη χρήση της μεταβλητής είναι εγγραφή δεν υπάρχει λόγος αρχικοποίησης της. Έτσι είναι προτιμότερο να γίνει private και να αποφευχθεί η χρονική καθυστέρηση της αρχικοποίησης.

Θα χρειαστούν τα εξής σύνολα:

FP(B) = { Μεταβλητές που βρίσκονται σε firstprivate clause }

W(B) = { Μεταβλητές που έχουν υποστεί μόνο εγγραφή }

WR(B) = { Μεταβλητές που έχουν υποστεί πρώτα εγγραφή και μετά ανάγνωση }

Για κάθε μεταβλητή του block εντολών B ελέγχουμε αν ανήκει στο σύνολο FP(B). Αν ανήκει, τότε ελέγχουμε αν βρίσκεται σε τουλάχιστον ένα από τα σύνολα W(B) ή WR(B), αν ναι τότε η μεταβλητή τοποθετείται στο τελικό σύνολο που είναι και το τελικό αποτέλεσμα.

Η συγκεκριμένη ανάλυση όπως και η αντίστοιχη στο 4.3.3 αποτελεί πολύτιμο βοήθημα για προγραμματιστές OpenMP βοηθώντας τους να αποφύγουν συχνά λάθη που έχουν να κάνουν με τον χαρακτηρισμό των μεταβλητών [4]. Σε συνδυασμό με τη λειτουργία autoscoping που προσφέρει ο ίδιος μεταφραστής, δημιουργεί ένα πολύ ισχυρό σύστημα ανάλυσης και βελτιστοποίησης. Συγκεκριμένα για το πρόβλημα

firstprivate/write η ανάλυση είναι εξαιρετικά σημαντική, καθώς μπορεί να εντοπίσει λανθασμένη διαχείριση ροής δεδομένων μεταξύ των νημάτων. Επιπλέον συμβάλλει στην βελτιστοποίηση του κώδικα εντοπίζοντας περιττές εγγραφές.

4.3.3 Το πρόβλημα του private clause

Σε αυτήν την περίπτωση θέλουμε να ελέγξουμε ποιες μεταβλητές έχουν οριστεί ως private σε κάποιο construct και στην συνέχεια διαβάζονται πριν υποστούν εγγραφή, όπως φαίνεται στο πρόγραμμα 4.2.

Πρόγραμμα 4.2: Το πρόβλημα του private clause

```
1  int z,x;
2  #pragma omp parallel private(x)
3  {
4    z=x;
5    ...
6  }
```

Το private clause δημιουργεί ένα τοπικό αντίγραφο της μεταβλητής αλλά δεν αρχικοποιείται. Έτσι, αν γίνει ανάγνωση της μεταβλητής τότε το αποτέλεσμα θα είναι απροσδιόριστο. Θα έπρεπε πιθανώς να χρησιμοποιηθεί κάποιο άλλο clause όπως το firstprivate.

Θα χρειαστούν τα εξής σύνολα:

P(B) = { Μεταβλητές που βρίσκονται σε private clause }

R(B) = { Μεταβλητές που έχουν υποστεί μόνο ανάγνωση }

RW(B) = { Μεταβλητές που έχουν υποστεί πρώτα ανάγνωση και μετά εγγραφή }

Για κάθε μεταβλητή του block εντολών B ελέγχουμε αν ανήκει στο σύνολο P(B). Αν ανήκει, τότε ελέγχουμε αν βρίσκεται σε τουλάχιστον ένα από τα σύνολα R(B) ή RW(B), αν ναι τότε η μεταβλητή τοποθετείται στο τελικό σύνολο που είναι και το τελικό αποτέλεσμα. Η διαδικασία ακολουθεί την ίδια λογική που περιγράψαμε στην υπόενοτητα 4.3.2 για το firstprivate clause.

Για το πρόβλημα private/read η ανάλυση είναι πολύ σημαντική, καθώς μπορούμε να αναγνωρίσουμε ποιες μεταβλητές χρησιμοποιούνται χωρίς αρχικοποίηση και να αποφύγουμε απροσδιόριστες καταστάσεις.

4.4 Υλοποίηση στον OMPi

4.4.1 Υλοποίηση κατάλληλων δομών

Για την υλοποίηση του μηχανισμού ανάλυσης ροής δεδομένων δημιουργήθηκαν κάποιες καινούριες δομές. Αρχικά δημιουργήθηκε ένα struct, που περιλαμβάνει όλες τις μεταβλητές και τα σύνολα που απαιτούνται για την ανάλυση. Αυτό το struct είναι προσβάσιμο σε όλο το πρόγραμμα, αλλά για κάθε διαφορετικό πρόβλημα ανάλυσης ροής δεδομένων, δημιουργείται τοπικά ένα ξεχωριστό αντικείμενο του. Με αυτόν τον τρόπο, αποφεύγεται η χρήση καθολικών μεταβλητών, καθιστώντας τον κώδικα πιο οργανωμένο και ευέλικτο. Επιπλέον, για κάθε πρόβλημα ανάλυσης ροής δεδομένων υλοποιήθηκε ένα αντίστοιχο wrapper function, το οποίο θα δούμε στη συνέχεια.

4.4.2 Υλοποίηση συνόλων

Όπως αναφέρθηκε παραπάνω το struct που δημιουργήθηκε αποθηκεύει όλες τις μεταβλητές και τα σύνολα που χρησιμοποιούνται στην ανάλυση. Για την δημιουργία των συνόλων χρησιμοποιήθηκε έτοιμη δομή του OMPi η οποία παρέχει λειτουργίες όπως αρχικοποίηση, διάσχιση και βασικές πράξεις συνόλων, όπως ένωση (set_union), αφαίρεση(set_subtract), διευκολύνοντας έτσι τη διαχείρισή τους.

4.4.3 Υλοποίηση Συναρτήσεων

Όπως αναφέρθηκε στην υποενότητα 4.4.1 δημιουργήθηκαν wrapper functions για κάθε πρόβλημα ανάλυσης ροής δεδομένων. Κάθε μία από αυτές τις συναρτήσεις αναλαμβάνει τη δημιουργία και αρχικοποίηση των αντικειμένων του struct, καλεί τις απαραίτητες συναρτήσεις και επιστρέφει τα τελικά αποτελέσματα. Αυτή η προσέγγιση προσφέρει μεγάλη ευελιξία και δυνατότητες επέκτασης, καθώς τα wrapper functions μπορούν να κληθούν οπουδήποτε εντός του OMPi. Επίσης, αν προκύψει η ανάγκη για προσθήκη νέας λειτουργίας, η διαδικασία γίνεται ιδιαίτερα εύκολη, αρκεί να δημιουργηθεί ένα νέο wrapper function και να προστεθούν οι απαραίτητες μεταβλητές στο struct, χωρίς να χρειάζεται να τροποποιηθεί εκτενώς ο υπάρχων κώδικας.

4.4.4 Υλοποίηση των Implicitly determined μεταβλητών

Ένα σημαντικό χαρακτηριστικό του μηχανισμού, είναι η διαχείριση των Implicitly determined μεταβλητών (υποενότητα 2.3.8). Συγκεκριμένα, το σύστημα μπορεί να εντοπίσει μεταβλητές που δεν δηλώνονται ρητά σε κάποια φράση και να τις προσδιορίσει αυτόματα. Η λειτουργικότητα αυτή εξασφαλίζει μεγαλύτερη ακρίβεια και ευελιξία στην ανάλυση. Παραδείγματα αυτής της υλοποίησης θα παρουσιαστούν στο κεφάλαιο 5.

4.4.5 Μηνύματα του OMPi (Warnings)

Με βάση τον μηχανισμό που φτιάξαμε αποφασίσαμε να προσθέσουμε στον OMPi την δυνατότητα να ενημερώνει τους προγραμματιστές για πιθανά λάθη τους όσον αφορά τον χαρακτηρισμό των μεταβλητών, καθώς και να προτείνει ποια είναι η ορθότερη χρήση. Προκειμένου να γίνει αυτό, εντάξαμε τον μηχανισμό ανάλυσης ροής δεδομένων στον μετασχηματισμό όλων των σχετικών οδηγιών του OpenMP. Συγκεκριμένα, λύνουμε όλα τα προβλήματα που αναφέρθηκαν στην Ενότητα 4.3 κατά το μετασχηματισμό των οδηγιών:

- parallel
- teams
- for
- single
- task
- sections.

Η ανάλυση γίνεται ακριβώς πριν ξεκινήσει ο μετασχηματισμός καθώς αυτός εξαφανίζει τις οδηγίες OpenMP, αντικαθιστώντας τες με τον μετασχηματισμένο κώδικα. Η ανάλυσή μας γίνεται ακριβώς πριν ξεκινήσει ο μετασχηματισμός ώστε να έχουμε πρόσβαση στην αρχική μορφή του κώδικα που περιλαμβάνει τις οδηγίες OpenMP. Εφόσον η ανάλυση αποδείξει ότι γίνεται λανθασμένος χαρακτηρισμός σε κάποιες μεταβλητές, εκτυπώνεται σχετικό προειδοποιητικό μήνυμα προς τον χρήστη.

Κεφάλαιο 5.

Παραδείγματα Λειτουργίας

5.1 Παραδείγματα ορθότητας

Στην παρούσα ενότητα παρουσιάζουμε απλοποιημένα παραδείγματα, στα οποία εκτυπώνονται τα σύνολα που μας ενδιαφέρουν. Η λειτουργικότητα αυτή δεν απευθύνεται σε προγραμματιστές εφαρμογών, πρόκειται για εσωτερική λειτουργία που αναπτύχθηκε στο πλαίσιο της διπλωματικής εργασίας, με σκοπό τη διευκόλυνση της αποσφαλμάτωσης του κώδικα.

5.1.1 Read-only και Write-only Μεταβλητές

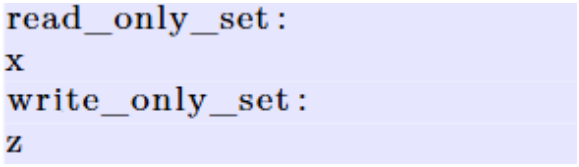
Αρχικά, θα εξετάσουμε το πρόγραμμα 5.1 για τις περιπτώσεις Write-only και Read-only. Συγκεκριμένα, θα αναλύσουμε ποιες μεταβλητές μέσα σε ένα μπλοκ κώδικα χρησιμοποιούνται αποκλειστικά για εγγραφή και ποιες αποκλειστικά για ανάγνωση. Σημειώνεται ότι τα παραδείγματα είναι ενδεικτικά και θα μπορούσαν να περιλαμβάνουν οποιαδήποτε OpenMP οδηγία, όπως parallel, for, teams, task, single ή sections. Για λόγους σαφήνειας και χρηστικότητας, στα παρακάτω παραδείγματα επιλέγεται η οδηγία parallel. Επιπλέον στα παραδείγματα επιλέξαμε ενδεικτικά να κάνουμε αναλύση με βάση το AST, αλλά προφανώς τα αποτελέσματα δεν αλλάζουν αν επιλεχθει το CFG.

Πρόγραμμα 5.1: Read-only και Write-only μεταβλητές

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel
5    {
6      z=x;
7      ...
8    }
9  }
```

Το πρόγραμμα 5.1 που εξετάζουμε βρίσκεται μέσα σε ένα parallel construct. Σε αυτό, ορίζονται τρεις μεταβλητές: x, y και z. Όπως παρατηρούμε, η x χρησιμοποιείται

αποκλειστικά για ανάγνωση, ενώ η z αποκλειστικά για εγγραφή. Ο μηχανισμός μας επιστρέφει δύο σύνολα αποτελεσμάτων 5.1. Όπως αναμενόταν, το read_only_set (το σύνολο που περιέχει τις μεταβλητές που χρησιμοποιούνται μόνο για ανάγνωση) περιλαμβάνει τη μεταβλητή x, ενώ το write_only_set (το σύνολο που περιέχει τις μεταβλητές που χρησιμοποιούνται μόνο για εγγραφή) περιλαμβάνει τη μεταβλητή z. Αυτό επιβεβαιώνει ότι ο μηχανισμός λειτουργεί σωστά και επιστρέφει το επιθυμητό αποτέλεσμα.



Σχήμα 5.1: Αποτελέσματα

5.1.2 Write/Read και Read/Write μεταβλητές

Σε αυτήν την υποενότητα θα εξετάσουμε το πρόγραμμα 5.2 και θα δούμε πως δουλεύει ο μηχανισμός για μεταβλητές που πρώτα υπόκεινται σε εγγραφή και στη συνέχεια σε ανάγνωση και μεταβλητές που πρώτα διαβάζονται και στη συνέχεια υπόκεινται σε εγγραφή.

Πρόγραμμα 5.2: Write/Read και Read/Write μεταβλητές

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel
5    {
6      z=x;
7      x=z;
8      ...
9    }
10 }
```

Στο πρόγραμμα 5.2, η μεταβλητή x πρώτα διαβάζεται και στη συνέχεια γράφεται, ενώ η μεταβλητή z πρώτα γράφεται και μετά διαβάζεται. Τα αποτελέσματα του σχήματος 5.2 επιβεβαιώνουν τα αποτελέσματα της ανάλυσης. Συγκεκριμένα:

- Το wr_set περιλαμβάνει τις μεταβλητές που πρώτα υφίστανται εγγραφή και στη συνέχεια ανάγνωση.
- Το rw_set περιλαμβάνει τις μεταβλητές που πρώτα διαβάζονται και έπειτα τροποποιούνται μέσω εγγραφής.

```
wr_set:
 z
rw_set:
 x
```

Σχήμα 5.2: Αποτελέσματα

5.1.3 Firstprivate και Write μεταβλητές

Όπως έχουμε ήδη αναφέρει, ένα από τα βασικά προβλήματα που επιδιώξαμε να λύσουμε είναι ο εντοπισμός των μεταβλητών που δηλώνονται ως firstprivate και στη συνέχεια υφίστανται άμεσα εγγραφή όπως φαίνεται στο πρόγραμμα 5.3.

Πρόγραμμα 5.3: Firstprivate και Write μεταβλητές χωρίς επαναρχικοποίηση

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel firstprivate(z)
5    {
6      z=3;
7      ...
8    }
9  }
```

Όπως παρατηρούμε στο πρόγραμμα 5.3, η μεταβλητή z έχει δηλωθεί ως firstprivate και στη συνέχεια υφίσταται εγγραφή. Συνεπώς, ο μηχανισμός μας θα πρέπει να την εντοπίσει και να την επιστρέψει ως αποτέλεσμα. Τα αποτελέσματα της ανάλυσης 5.3 επιβεβαιώνουν τη σωστή λειτουργία του μηχανισμού.

```
writen_firstprivate_vars_set:
z
```

Σχήμα 5.3: Αποτελέσματα

Είναι ιδιαίτερα σημαντικό να εξετάσουμε την περίπτωση όπου η μεταβλητή z επαναρχικοποιείται πριν υποστεί εγγραφή όπως φαίνεται στο πρόγραμμα 5.4.

Πρόγραμμα 5.4: Firstprivate και Write μεταβλητές με επαναρχικοποίηση

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel firstprivate(z)
5    {
6      int z;
7      z=3;
8      ...
9    }
10 }
```

Στο πρόγραμμα 5.4, παρατηρούμε ότι η εκχώρηση z = 3; αναφέρεται στη νέα μεταβλητή z, η οποία δηλώνεται εκ νέου πριν από την εκχώρηση και δεν ανήκει στην firstprivate φράση.

Ως αποτέλεσμα, το πρόγραμμά μας δεν θα επιστρέψει καμία μεταβλητή.

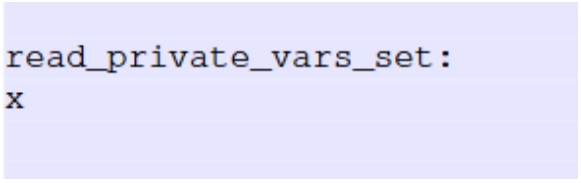
5.1.4 Private και Read μεταβλητές

Ένα ακόμη βασικό πρόβλημα που επιδιώξαμε να λύσουμε είναι ο εντοπισμός των μεταβλητών που δηλώνονται ως private και στη συνέχεια υφίστανται άμεσα ανάγνωση όπως φαίνεται και στο πρόγραμμα 5.5.

Πρόγραμμα 5.5: Private και Read μεταβλητές

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel private(x)
5    {
6       z=x;
7       ...
8    }
9  }
```

Όπως παρατηρούμε στο πρόγραμμα 5.5, η μεταβλητή x έχει δηλωθεί ως private και στη συνέχεια υφίσταται ανάγνωση. Συνεπώς, ο μηχανισμός μας θα πρέπει να την εντοπίσει και να την επιστρέψει ως αποτέλεσμα. Τα αποτελέσματα της ανάλυσης 5.4 επιβεβαιώνουν τη σωστή λειτουργία του μηχανισμού. Αν μια μεταβλητή x δηλωνόταν εκ νέου πριν την ανάγνωση της τότε θα ίσχυε ό,τι αναφέραμε παραπάνω στην υποενότητα 5.1.3.



Σχήμα 5.4: Αποτελέσματα

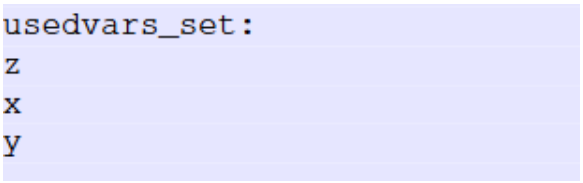
5.1.5 Αχρησιμοποίητες μεταβλητές

Στο πρόγραμμα 5.6 θα δούμε ένα απλό πρόβλημα, όπου μας ενδιαφέρει ποιες μεταβλητές που έχουν ήδη δηλωθεί στο πρόγραμμα χρησιμοποιούνται σε ένα μπλοκ κώδικα και ποιες όχι.

Πρόγραμμα 5.6: Αχρησιμοποίητες μεταβλητές

```
1  int main( )
2  {
3    int x,y,z,t;
4    #pragma omp parallel private(x)
5    {
6      z=3;
7      y=x;
8      ...
9    }
10 }
```

Στο πρόγραμμα 5.6 βλέπουμε ότι μέσα στο μπλοκ του parallel χρησιμοποιούνται οι μεταβλητές x, y, z, ενώ στην αρχή έχει οριστεί και η μεταβλητή t. Ο μηχανισμός θα επιστρέψει, όπως φαίνεται στα αποτελέσματα 5.5, μόνο τις x, y, z, επειδή μόνο αυτές χρησιμοποιούνται.



Σχήμα 5.5: Αποτελέσματα

5.1.6 Παράδειγμα με οδηγία Task

Όπως αναφέραμε στην υπόενοτητα 2.3.8, υπάρχουν οι implicitly determined μεταβλητές. Στην ειδική αυτή περίπτωση η οδηγία task χειρίζεται κάποιες μεταβλητές διαφορετικά από άλλες οδηγίες. Παρακάτω υπάρχουν δύο παραδείγματα κώδικα, που είναι ουσιαστικά ίδια, με την μόνη διαφορά ότι το πρόγραμμα 5.7 χρησιμοποιεί την

οδηγία parallel, ενώ το πρόγραμμα 5.8 χρησιμοποιεί την οδηγία task. Τα αποτελέσματα τους παρουσιάζονται στις εικόνες 5.6 και 5.7 αντίστοιχα.

Πρόγραμμα 5.7: Πρόγραμμα με οδηγία parallel

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp parallel firstprivate(z)
5    {
6      z=3;
7      x=3;
8      ...
9    }
10 }
```

Πρόγραμμα 5.8: Πρόγραμμα με οδηγία Task

```
1  int main( )
2  {
3    int x,y,z;
4    #pragma omp task firstprivate(z)
5    {
6      z=3
7      x=3;
8      ...
9    }
10 }
```

Ο μηχανισμός θα επιστρέψει σωστά τα αποτελέσματα που φαίνονται στην εικόνα 5.7, διότι η οδηγία task αντιμετωπίζει τις μεταβλητές που δεν είναι μέσα σε κάποια φράση και έχουν οριστεί εξωτερικά πριν από αυτήν, ως firstprivate. Από την άλλη, η οδηγία parallel (και άλλες παρόμοιες) τις βλέπει ως shared μεταβλητές. Έτσι, για την οδηγία task, η μεταβλητή x θεωρείται firstprivate, και καθώς γίνεται τροποποίηση της, η τιμή της επιστρέφεται σωστά.

```
writen_firstprivate_vars_set:
z
```

Σχήμα 5.6: Αποτελέσματα

```
writen_firstprivate_vars_set:
z
x
```

Σχήμα 5.7: Αποτελέσματα

5.2 Υποστήριξη σε προγραμματιστές εφαρμογών

Τα προηγούμενα παραδείγματα ήταν τεχνητά τμήματα κώδικα προκειμένου να δειχθεί η ορθότητα του μηχανισμού. Τέτοια αποτελέσματα ανάλυσης ροής δεδομένων είναι διαθέσιμα μόνο στους προγραμματιστές που αναπτύσσουν τον μεταφραστή OMPi.

Σε αυτή την ενότητα, στρέφουμε το ενδιαφέρον μας στους χρήστες του OMPi, δηλαδή στους προγραμματιστές που αναπτύσσουν εφαρμογές OpenMP και θα δείξουμε πως ο μηχανισμός μας προσφέρει πολύτιμη βοήθεια, προειδοποιώντας για τυχόν λάθος χρήση των μεταβλητών μίας περιοχής. Θα δούμε μία ολοκληρωμένη εφαρμογή δίνοντας έμφαση στα μηνύματα (warnings) που επιστρέφει ο OMPi στους προγραμματιστές εφαρμογών. Συγκεκριμένα θα δούμε το πρόγραμμα 5.9 που υπολογίζει πρώτους αριθμούς. Η συνάρτηση openmp_primes(long int n) υπολογίζει όλους τους πρώτους αριθμούς μέχρι το n.

Πρόγραμμα 5.9: Ολοκληρωμένη εφαρμογή

```
1  void openmp_primes (long int n) {
2     long int i, num, divisor, quotient, remainder;
3
4     if (n < 2) return;
5     count = 1;
6     lastprime = 2;
7
8     #pragma omp parallel for private (num, divisor, quotient, remainder)
9     for (i = 0; i<(n - 1)/2; ++i) {
10         num = 2*i + 3;
11
12         divisor = 1;
13         do
14         {
15            divisor += 2;
16            quotient = num / divisor;
17            remainder = num % divisor;
18         } while (remainder && divisor <= quotient);
19
20
21         if (remainder || divisor == num)
22         {
23            count++;
24            lastprime = num;
25         }
26    }
27 }
```

Στο πρόγραμμα 5.9 εστιάζουμε στον κώδικα που βρίσκεται εντός της οδηγίας #pragma omp parallel for. Οι μεταβλητές num, divisor, quotient και remainder δηλώνονται ως private, ώστε κάθε νήμα να διαθέτει το δικό του αντίγραφο και να αποφεύγονται συγκρούσεις (race conditions). Δεν είναι απαραίτητο να χρησιμοποιηθεί firstprivate, καθώς οι συγκεκριμένες μεταβλητές αρχικοποιούνται τοπικά μέσα στον βρόχο και δεν εξαρτώνται από κάποια εξωτερική τιμή που να προέρχεται από το κύριο νήμα.

Για να αναδείξουμε τη λειτουργία του μηχανισμού μας και να κατανοήσουμε τη σημασία της σωστής χρήσης των OpenMP ιδιοτήτων, θα πειραματιστούμε θέτοντας ορισμένες από αυτές τις μεταβλητές ως firstprivate και θα παρατηρήσουμε τα αποτελέσματα στην εκτέλεση και στην ορθότητα του προγράμματος. Συγκεκριμένα, τροποποιούμε την οδηγία σε: #pragma omp parallel for firstprivate(num) private(divisor, quotient, remainder). Όπως βλέπουμε στην Εικόνα 5.8, ο μηχανισμός ανάλυσης ροής δεδομένων εντοπίζει και λύνει το πρόβλημα που είδαμε στην ενότητα 5.1.3 και εμφανίζει ένα ενημερωτικό μήνυμα προς τον προγραμματιστή της εφαρμογής, το οποίο εξηγεί με απλό και κατανοητό τρόπο το σφάλμα και προτείνει την κατάλληλη λύση.

```
In the for construct (test_dfa.c, line 16),
the following variables are firstprivate: num
However, they are written before they are read;
making them private is more appropriate.
```

Σχήμα 5.8: Αποτελέσματα

Κεφάλαιο 6.

Επίλογος

6.1 Σύνοψη Διπλωματικής Εργασίας

Σε αυτήν τη διπλωματική εργασία, ξεκινήσαμε εξετάζοντας την ανάπτυξη της τεχνολογίας τα τελευταία χρόνια και τον τρόπο που αυτή η ανάπτυξη συνέβαλε στην εξέλιξη των παράλληλων συστημάτων. Εστιάσαμε στη δύναμη και τη σημασία των παράλληλων συστημάτων τονίζοντας την τεράστια συμβολή τους στη σημερινή εποχή. Στη συνέχεια, κάναμε μια επισκόπηση του OpenMP, αναλύοντας τι είναι και πώς χρησιμοποιείται, εστιάζοντας κυρίως σε σημαντικές δομές (οδηγίες και φράσεις) που χρησιμοποιήθηκαν εκτενώς σε αυτήν τη διπλωματική εργασία. Επιπλέον, πραγματοποιήσαμε μια εισαγωγή στον OMPi και εστιάσαμε σε συγκεκριμένες δομές που είναι κρίσιμες για την εργασία μας, δεδομένου ότι το μέγεθος και οι δυνατότητές του είναι τεράστιες. Δημιουργήθηκε ένας εξ ολοκλήρου νέος μηχανισμός ανάλυσης ροής δεδομένων. Οποιοσδήποτε χρήστης του OMPi μπορεί, αν το επιθυμεί, να ενεργοποιήσει τον μηχανισμό με την κατάλληλη επιλογή (option: --dfa) και να εκτελέσει τον μηχανισμό μας, λαμβάνοντας λύσεις για συγκεκριμένα προβλήματα.

6.2 Μελλοντική Εργασία

Το πεδίο της ανάλυσης ροής δεδομένων είναι ευρύ και περιλαμβάνει ένα σύνολο τεχνικών που στοχεύουν τόσο στη βελτιστοποίηση όσο και στην ασφάλεια του κώδικα. Έχει ιδιαίτερα κρίσιμο ρόλο στους παραλληλοποιητικούς μεταφραστές, καθώς πρέπει να εξασφαλίσει τη σωστή παραλληλοποίηση, την αποδοτική εκτέλεση και τον σωστό συγχρονισμό. Σε συνδυασμό με το OpenMP, προκύπτουν αρκετά προβλήματα που θα ήταν ενδιαφέρον να λυθούν.

Στον OMPi, ο μηχανισμός ανάλυσης ροής δεδομένων που υλοποιήσαμε λειτουργεί σωστά και σχεδιάστηκε με τέτοιο τρόπο ώστε να είναι εύκολα επεκτάσιμος. Η δομή του

κώδικα βοηθά σημαντικά τους μελλοντικούς προγραμματιστές που θα θελήσουν να τον επεκτείνουν. Συγκεκριμένα, για κάθε νέο πρόβλημα ανάλυσης ρόης δεδομένων, το μόνο που απαιτείται είναι η προσθήκη μιας νέας συνάρτησης (wrapper function) και η εισαγωγή των αναγκαίων μεταβλητών στο struct, κάτι που καθιστά τον μηχανισμό μας ιδιαίτερα ευέλικτο και επεκτάσιμο.

Πέρα από την επίλυση πρόσθετων προβλημάτων ανάλυσης ροής δεδομένων, ο μηχανισμός μπορεί να αξιοποιηθεί σε πρακτικά σενάρια με στόχο τη μείωση άσκοπων μεταφορών δεδομένων. Ένα χαρακτηριστικό παράδειγμα αφορά προγράμματα που εκτελούνται σε συστήματα με συνδεδεμένες ή απομακρυσμένες συσκευές, όπου οι μεταφορές δεδομένων είναι ιδιαίτερα χρονοβόρες. Αν γνωρίζουμε ποιες μεταβλητές είναι πραγματικά απαραίτητες για την εκτέλεση του προγράμματος, μπορούμε να αποφύγουμε τη μεταφορά περιττών δεδομένων, κάτι που βοηθά στην εξοικονόμηση χρόνου και πόρων.

Επιπλέον, είναι σημαντικό να αναφέρουμε και ορισμένους περιορισμούς της υλοποίησης. Ένας βασικός περιορισμός προέρχεται από τη δομή του Control Flow Graph (CFG) του OMPi, ο οποίος προς το παρόν δεν υποστηρίζει scope-level πληροφορία. Αυτό σημαίνει ότι δεν μπορούμε εύκολα να ξεχωρίσουμε σε ποιο κομμάτι του κώδικα είναι «ορατή» κάθε μεταβλητή, κάτι που δυσκολεύει την ανάλυση σε πιο σύνθετες περιπτώσεις. Η αντιμετώπιση αυτού του περιορισμού αποτελεί μια σημαντική κατεύθυνση για μελλοντική επέκταση.

Βιβλιογραφία

[1] V.V. Dimakopoulos. “Parallel Systems and Programming”, March 2017.

[2] Keith D. Cooper, Linda Torczon. Engineering a Compiler, Second Edition.

[3] I.K. Kasmeridis, S. Mantelos, A. Piperis, V.V. Dimakopoulos, “Transparent remote OpenMP offloading based on MPI”, in Proc. Euro-Par 2023, 29th International European Conference on Parallel and Distributed Computing Workshops, Limassol, Cyprus, Aug. 2023, pp. 237—241.

[4] M. Süß and C. Leopold, Common mistakes in OpenMP and how to avoid them, in Proc. IWOMP 2006, 2nd International Workshop on OpenMP (Reims, France, 2006)312-323

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

[6] L. Dagum and R. Menon, “OpenMP: an industry standard API for sharedmemory programming,” IEEE computational science and engineering, vol. 5, no. 1, pp. 46–55, 1998.

[7] “OpenMP application programming interface , version 6.0,” https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf November 2024.