Πανεπιστημιο Ιωαννινων
Σχολη Θετικων Επιστημων
Τμημα Πληροφορικης

Πτυχιακή Εργασία

Υλοποίηση μεταφραστή C με επεκτάσεις OpenMP για παραλληλοποίηση

Ηλίας Λεοντιάδης
Α.Μ. 302

Γεώργιος Τζούμας
Α.Μ. 331

Επιβλέπων καθηγητής: Βασίλειος Δημακόπουλος

Ιουλιος 2002

Περίληψη

Το θέμα της πτυχιακής μας εργασίας είναι η υλοποίηση ενός μεταφραστή για τη γλώσσα C με επεκτάσεις *OpenMP*. Οι επεκτάσεις αυτές είναι ειδικές εντολές που περιγράφουν στον μεταφραστή πώς να παραλληλοποιήσει το πρόγραμμα. Ο μεταφραστής μας είναι γραμμένος εξολοκλήρου σε C, ενώ έχουν χρησιμοποιηθεί και τα εργαλεία Flex και Bison. Παράγεται κώδικας σε ANSI C ο οποίος χρησιμοποιεί τη βιβλιοθήκη POSIX threads.

Έχουμε επικεντρώσει το ενδιαφέρον μας στον τρόπο με τον οποίο μετασχηματίζεται ο κώδικας της μίας γλώσσας στην άλλη. Αυτό δεν είναι τόσο εύκολο, διότι πέρα από το ότι το τελικό πρόγραμμα θα πρέπει να είναι σημασιολογικά ισοδύναμο, θα πρέπει να είναι και αποδοτικό όσον αφορά την ταχύτητα εκτέλεσης και τη χρήση μνήμης.

Το τελικό αποτέλεσμα ήταν η δημιουργία ενός μεταφραστή που πληρεί το πρότυπο *OpenMP*. Βασικά πλεονεκτήματά του είναι ο μικρός χρόνος μετάφρασης και ο αποδοτικός παραγόμενος κώδικας. Τέλος, η υλοποίηση αυτή είναι μεταφέρσιμη σε οποιαδήποτε πλατφόρμα UNIX που υποστηρίζει POSIX threads.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλα συστήματα

Υπάρχει όλο και μεγαλύτερη ανάγκη για γρηγορότερους υπολογιστές. Η ανάγκη αυτή για μεγαλύτερη υπολογιστική ισχύ πηγάζει από δύο βασικά αίτια:

- Την γρηγορότερη επίλυση προβλημάτων.

- Την εκτέλεση κάποιων υπολογισμών με μεγαλύτερη ακρίβεια.

Έτσι, οι υπολογιστές γίνονται όλο και πιο γρήγοροι, συγκεκριμένα οι επιδόσεις τους διπλασιάζονται κάθε δεκαοχτώ μήνες. Πολλοί είναι αυτοί που υποστηρίζουν ότι αυτή η αύξηση των επιδόσεων δεν μπορεί να συνεχιστεί επ' άπειρο. Επιπλέον, υπάρχουν δύσκολα προβλήματα που απαιτούν άμεση λύση άλλα η ταχύτητα των σημερινών υπολογιστών δεν αρκεί.

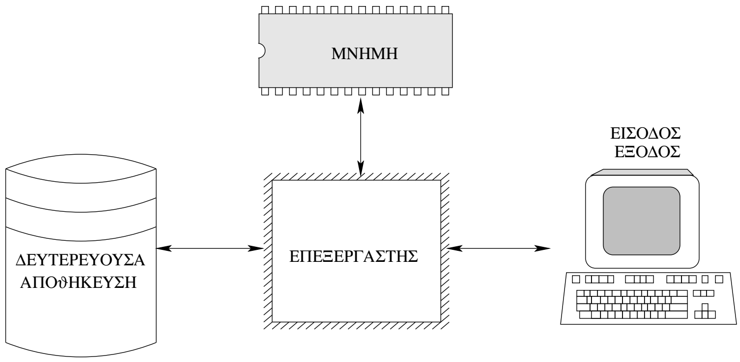
Ένα χαρακτηριστικό παράδειγμα είναι το πρόβλημα τις πρόβλεψης του αυριανού καιρού που απαιτεί την γρήγορη επίλυση του (μέσα σε ένα 24ωρο) με την μεγαλύτερη δυνατή ακρίβεια. Έτσι δημιουργήθηκε η ανάγκη για τη δημιουργία παράλληλων συστημάτων.

Αν το δούμε αφαιρετικά, ένα παράλληλο σύστημα ουσιαστικά σπάει το πρόβλημα σε μικρότερα προβλήματα, τα επιλύει ταυτόχρονα (παράλληλα) σε πολλούς υπολογιστές και έπειτα συγκεντρώνει τα αποτελέσματα και τα συνενώνει ώστε να εξάγει το τελικό. Αυτή η λύση είναι γνωστή ως *παράλληλος υπολογισμός* (*parallel computing*) ή *πολυεπεξεργασία* (*multiprocessing*).

1.2 Αρχιτεκτονικές

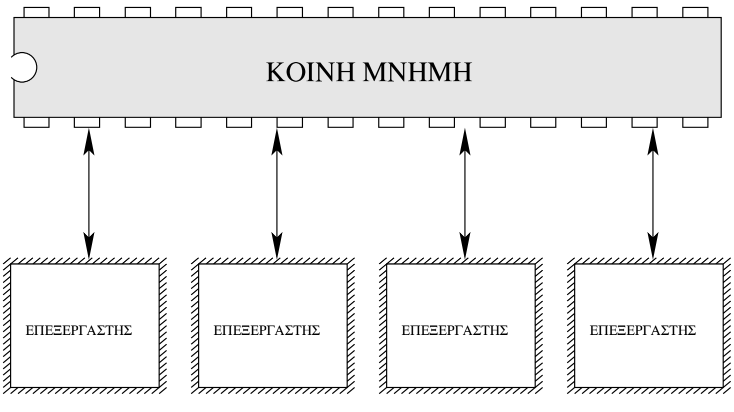
Πριν προχωρήσουμε παραπέρα στην ανάλυση των παράλληλων συστημάτων είναι αναγκαίο να γνωρίσουμε κάποια βασικά στοιχεία της αρχιτεκτονικής τους.

Όπως φαίνεται και στο σχήμα 1.1, ένα απλό μοντέλο υπολογιστή (μη-παράλληλου) αποτελείται από μια μονάδα επεξεργασίας (CPU) που συνδέεται σε κάποια μνήμη, κάποια μονάδα αποθήκευσης και κάποιες μονάδες εισόδου εξόδου — πληκτρολόγιο, ποντίκι, οθόνη, κάρτα δικτύου κτλ...



Σχήμα 1.1: Σειριακός υπολογιστής

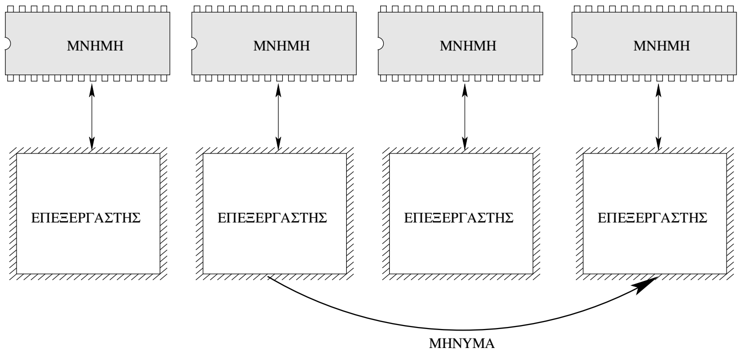
Τα παράλληλα συστήματα που έχουν αναπτυχθεί ακολουθούν δύο βασικές αρχιτεκτονικές: Τους πολυεπεξεργαστές κοινής μνήμης και τους πολυεπεξεργαστές (ή πολυυπολογιστές) κατανεμημένης μνήμης.



Σχήμα 1.2: Αρχιτεκτονική κοινής μνήμης

Στους πολυεπεξεργαστές κοινής μνήμης δεν υπάρχει σύνδεση μεταξύ των επεξεργαστών, παρά μόνο ανάμεσα στους επεξεργαστές και τη μνήμη η οποία είναι άμεσα προσπελάσιμη από όλους (σχήμα 1.2). Κάθε επεξεργαστής μπορεί να εκτελεί διαφορετική εντολή επάνω σε διαφορετικά κομμάτια τα οποία μπορούν να προσπελάσουν όλοι. Αυτό έχει σαν αποτέλεσμα ότι όλοι οι επεξεργαστές βλέπουν την ίδια μνήμη και με τον ίδιο τρόπο.

Στα συστήματα κατανεμημένης μνήμης, κάθε επεξεργαστής έχει τη δική του, ιδιωτική μνήμη, την οποία προσπελαύνει απευθείας μόνο αυτός (σχήμα 1.3). Η επικοινωνία είναι εφικτή λόγω ύπαρξης διασυνδετικού δικτύου διαμέσω του οποίου γίνεται ανταλλαγή μηνυμάτων.



Σχήμα 1.3: Αρχιτεκτονική κατανεμημένης μνήμης

1.3 Μοντέλα παράλληλου προγραμματισμού

Υπάρχουν λοιπόν συστήματα που είναι ικανά να επεξεργαστούν ταυτόχρονα πολλές εντολές. Είναι λοιπόν αναγκαία η υιοθέτηση ενός καλού προγραμματιστικού μοντέλου για την αξιοποίηση αυτών των συστημάτων. Αυτό που συμβαίνει συνήθως είναι ότι το πρόβλημα διαχωρίζεται σε πολλές ανεξάρτητες διεργασίες (processes) ή νήματα (threads) που εκτελούνται ταυτόχρονα μέσα σε έναν παράλληλο υπολογιστή.

Οι διεργασίες αυτές πρέπει με κάποιο τρόπο να επικοινωνούν, ώστε για παράδειγμα να μπορούν να ανταλλάσσουν δεδομένα, και να συγχρονίζονται. Έτσι, υπάρχουν τα παρακάτω μοντέλα που επηρεάζονται από την αρχιτεκτονική του παράλληλου υπολογιστή.

1.3.1 Μοντέλο κοινής μνήμης

Στο μοντέλο κοινού χώρου διευθύνσεων ο προγραμματιστής βλέπει τα προγράμματα σαν μια συλλογή διεργασιών οι οποίες προσπελαύνουν μια συλλογή από κοινές (καθολικές) μεταβλητές. Φυσιολογικά το μοντέλο αυτό ταιριάζει απόλυτα σε έναν πολυεπεξεργαστή κοινής μνήμης, όπου οι καθολικές μεταβλητές αποθηκεύονται στην κοινή μνήμη και προσπελαύνονται από όλους. Αυτός όμως είναι και ο λόγος για τον οποίο υπάρχει περίπτωση απροσδιορίστων καταστάσεων, όταν π.χ. παραπάνω από μία διεργασίες προσπαθούν να αλλάξουν ταυτόχρονα την τιμή μιας κοινής μεταβλητής. Επομένως, στο μοντέλο αυτό παρέχονται τρόποι και δομές που εμποδίζουν τις ταυτόχρονες προσπελάσεις μεταβλητών. Παράδειγμα τέτοιων δομών ειναι οι σημαφόροι. Προγραμματιστικά, ο τρόπος δεν διαφέρει πολύ από το γνώριμο σειριακό μοντέλο. Η επικοινωνία μεταξύ των διεργασιών γίνεται μέσω τροποποίησης κοινών μεταβλητών στη μνήμη.

Ο πιο διαδεδομένος τρόπος προγραμματισμού σε αυτό το μοντέλο είναι η χρήση νημάτων. Μια διεργασία μπορεί να περιέχει πολλά νήματα τα οποία μοιράζονται τον χώρο διευθύνσεών της. Τα νήματα αυτά ως μικροδιεργασίες εκτελούνται παράλληλα. Υπάρχουν αρκετές υλοποιήσεις νημάτων (με μορφή βιβλιοθήκης). Οι πιο δημοφιλείς είναι τα POSIX και Solaris threads.

1.3.2 Μοντέλο μεταβίβασης μηνυμάτων

Στο μοντέλο μεταβίβασης μηνυμάτων το πρόγραμμα είναι μια συλλογή αυτόνομων διεργασιών οι οποίες δεν έχουν καμία κοινή μεταβλητή και οι οποίες έχουν τη δυνατότητα να επικοινωνούν μεταξύ τους ανταλλάσσοντας δεδομένα. Επομένως υπάρχουν προγραμματιστικές δομές που καθιστούν δυνατή την αποστολή και λήψη μηνυμάτων μεταξύ των διεργασιών. Είναι λογικό ότι υπάρχει άμεση αντιστοιχία με τους πολυεπεξεργαστές κατανεμημένης μνήμης.

Για πολλές εφαρμογές το μοντέλο αυτό είναι αρκετά αποτελεσματικό. Όμως στα προβλήματα εκείνα που μεγάλος όγκος δεδομένων πρέπει να είναι προσβάσιμος σε πολλές (ή και όλες τις) διεργασίες υπάρχει πρόβλημα λόγω των πολλών μηνυμάτων που ανταλλάσσονται. Επιπρόσθετα, ο προγραμματισμός με αυτό το μοντέλο είναι αρκετά πολύπλοκος.

Δύο από τα πιο δημοφιλή πακέτα που υποστηρίζουν το μοντέλο μεταβίβασης μηνυμάτων είναι το LAM/MPI [6] και το PVM [7]. Το πρώτο είναι μια υλοποίηση του προτύπου MPI [5] το οποίο είναι και το ποιο διαδεδομένο πρότυπο για προγραμματισμό στο μοντέλο μεταβίβασης μηνυμάτων. Χρησιμοποιεί σταθμούς εργασίας ως επεξεργαστές και το δίκτυο στο οποίο συνδέονται για την επικοινωνία. Παρέχει ρουτίνες για αποστολή και λήψη μηνυμάτων. Το PVM είναι ένα περιβάλλον το οποίο επιτρέπει παράλληλο δικτυακό προγραμματισμό. Εκτός απο ρουτίνες αποστολής και λήψης μηνυμάτων διαθέτει και μηχανισμούς για διαχείριση διεργασιών.

1.4 Αντικείμενο της εργασίας

Και στα δύο μοντέλα προγραμματισμού που αναφέραμε παραπάνω, ο προγραμματιστής είναι υπεύθυνος για την παραλληλοποίηση του προγράμματος. Ο προγραμματιστής πρέπει να χωρίσει το πρόγραμμα του σε διεργασίες, να φροντίσει να αποδώσει στις διεργασίες αυτές από κάποιο μέρος του προγράμματος να επεξεργαστούν και στο τέλος να συγκεντρώσει τα αποτελέσματα. Αν συμπεριλάβουμε στα παραπάνω και τους έλεγχους για συγκρούσεις, προβλήματα συγχρονισμών και επικοινωνίας, τότε καταλαβαίνουμε ότι ο προγραμματιστής πρέπει να έχει αρκετές γνώσεις πάνω στα παράλληλα συστήματα ώστε να γράψει ένα αποδοτικό πρόγραμμα.

Η εργασία αυτή έχει ως στόχο την υλοποίηση ενός μεταφραστή (compiler) ο οποίος, θα κάνει πιο εύκολη τη μετατροπή ενός σειριακού προγράμματος σε παράλληλο και δε θα απαιτεί από τον χρήστη να έχει καλές γνώσεις πάνω στα παράλληλα συστήματα. Επιπλέον ο μεταφραστής αυτός θα πρέπει να είναι γρήγορος και να παράγει αποδοτικό παράλληλο κώδικα. Τέλος θα πρέπει ο μεταγλωττιστής να είναι μεταφέρσιμος σε πολλά λειτουργικά συστήματα.

Έχει γίνει αρκετή έρευνα πάνω σε αυτό το θέμα και έχουν προταθεί δύο λύσεις [13, 14]: Μεταφραστές που αναλύουν αυτόματα ολόκληρο τον κώδικα για σημεία που είναι δυνατόν να παραλληλοποιηθούν χωρίς προβλήματα και μεταφραστές που παραλληλοποιούν μέρη του προγράμματος με βάση οδηγίες που δίνει ο προγραμματιστής. Όσον αφορά την πρώτη λύση, λόγω της πολυπλοκότητας που έχει αυτή η διαδικασία, οι μεταφραστές που έχουν δημιουργηθεί είναι σχετικά αργοί, δεν παράγουν πάντα αποτελεσματικό κώδικα, και λειτουργούν για συγκεκριμένου είδους προγράμματα (κυρίως παραλληλοποίηση βρόχων for). Η δεύτερη λύση είναι λοιπόν αυτή που αφορά αυτή την εργασία.

Στην εργασία αυτή υλοποιούμε έναν μεταφραστή που ακολουθεί το πρότυπο *OpenMP* C/C++[3]. Κατά το πρότυπο αυτό, στην γλώσσα C/C++, προσθέτονται κάποιες επιπλέον εντολές στην γλώσσα C που υποδεικνύουν στον μεταφραστή πως να παραλληλοποιήσει το πρόγραμμα. Συγκεκριμένα, υπάρχουν εντολές σχετικά με τον χωρισμό σε διεργασίες, αμοιβαίο αποκλεισμό και συγχρονισμό κ.α.

Το πρότυπο αυτό είναι αρκετά διαδεδομένο. Πολλές εταιρίες (όπως οι Sun, Sillicon Graphics, IBM) έχουν αναπτύξει τέτοιους μεταγλωττιστές. Εκτός από αυτές τις εμπορικές υλοποιήσεις, υπάρχουν και δύο ελεύθερες· το OdinMP[8] και το OmniMP[9]. Η πρώτη υλοποίηση είναι γραμμένη σε Java και παράγει κώδικα C. Η δεύτερη είναι γραμμένη σε Java και C και παράγει κατευθείαν τελικό (εκτελέσιμο) κώδικα.

Η δική μας υλοποίηση —γραμμένη εξολοκλήρου σε C— δεν παράγει απ' ευθείας κώδικα μηχανής, αλλά μετασχηματίζει το αρχικό πρόγραμμα με τις εντολές *OpenMP* σε κώδικα ANSI C που χρησιμοποιεί τη βιβλιοθήκη POSIX threads. Έτσι, όταν μεταφραστεί το πρόγραμμα που δημιουργήθηκε με έναν απλό compiler της C θα εκτελείται παράλληλα σύμφωνα με τις *OpenMP* εντολές που είχε ο κώδικας του χρήστη.

1.5 Δομή της εργασίας

Η εργασία αυτή έχει οργανωθεί ως εξής:

Στο κεφάλαιο 2 δίνεται μία περιληπτική περιγραφή του προτύπου *OpenMP*. Παρουσιάζουμε βασικά πλεονεκτήματα της χρήσης του, και στη συνέχεια περιγράφουμε τις εντολές που παρέχει μαζί με μερικά παραδείγματα.

Στο κεφάλαιο 3 περιγράφουμε την υλοποίηση του δικού μας μεταφραστή. Αναλύουμε τα εργαλεία που χρησιμοποιήσαμε για τις διάφορες φάσεις της μετάφρασης καθώς και τα προβλήματα που αντιμετωπίσαμε.

Το κεφάλαιο 4 ολοκληρώνει την περιγραφή του μεταφραστή εξετάζοντας με λεπτομέρεια την παραγωγή του τελικού (μετασχηματισμένου) κώδικα. Για κάθε εντολή *OpenMP* περιγράφουμε τον τρόπο με τον οποίο αυτή μετασχηματίστηκε σε κώδικα C, κάνοντας χρήση των συναρτήσεων των POSIX threads.

Στο κεφάλαιο 5 παραθέτουμε κάποιες μετρήσεις σχετικά με τις επιδόσεις του μεταφραστή μας. Συγκεκριμένα εξετάζουμε τόσο το χρόνο μετάφρασης, όσο και την επιτάχυνση στην εκτέλεση. Επίσης γίνεται σύγκριση με τις δύο ελεύθερες υλοποιήσεις OdinMP και OmniMP.

Τέλος, στο κεφάλαιο 6 συνοψίζουμε κι εξετάζουμε μελλοντικές επεκτάσεις.

Κεφάλαιο 2

Εισαγωγή στο *OpenMP*

Το *OpenMP* είναι ένα πρότυπο που καθορίζει ένα σύνολο από εντολές για τον μεταφραστή, οι οποίες προστίθενται στον πηγαίο κώδικα ενός προγράμματος το οποίο έχει γραφεί έχοντας υπόψιν τη σειριακή εκτέλεση. Δηλαδή, ξεκινάμε από ένα σειριακό πρόγραμμα και σταδιακά μπορούμε να παραλληλοποιήσουμε κομμάτια του (incremental parallelization). Οι εντολές αυτές υποδεικνύουν στον μεταφραστή πώς να παραλληλοποιήσει τον κώδικα για εκτέλεση σε περιβάλλον πολυεπεξεργαστών κοινής μνήμης. Το πρότυπο *OpenMP* περιλαμβάνει επίσης ρουτίνες βιβλιοθήκης για τροποποίηση των παραμέτρων εκτέλεσης, όπως τον αριθμό νημάτων. Για πλήρη περιγραφή του προτύπου δείτε το [3].

2.1 Πλεονεκτήματα του *OpenMP*

Πριν την εμφάνιση του *OpenMP*, δεν υπήρχε τυποποιημένος τρόπος για να παραλληλοποιήσει κάποιος εύκολα ένα πρόγραμμα σε περιβάλλον πολυεπεξεργαστών κοινής μνήμης. Παρότι υπήρχε ένα πρότυπο για μεταβίβαση μηνυμάτων , οι προγραμματιστές που ήθελαν παραλληλισμό κοινής μνήμης έπρεπε είτε να χρησιμοποιήσουν μη στάνταρ και μη μεταφέρσιμα API είτε να γράψουν κώδικα για την υποκείμενη υλοποίηση πολυεπεξεργασίας, όπως τα νήματα POSIX. Αυτά τα προγράμματα ήταν γενικά μεταφέρσιμα, ωστόσο κατά την ανάπτυξή τους η προσοχή του προγραμματιστή αναλώνονταν στις λεπτομέρειες της υλοποίησης πολυεπεξεργασίας και όχι στο κυρίως πρόβλημα. Το *OpenMP* έρχεται να αλλάξει την κατάσταση αυτή:

- *Το* OpenMP *είναι ένα κοινά αποδεκτό πρότυπο.* Προγράμματα που αναπτύσσονται βάσει αυτού του προτύπου είναι μεταφέρσιμα σε μια μεγάλη ποικιλία συστημάτων πολυεπεξεργαστών κοινής μνήμης. Οι κατασκευαστές αυτών των συστημάτων δεν έχουν παρά να υλοποιήσουν ένα καλώς καθορισμένο API.

- *Το* OpenMP *δεν εξαναγκάζει* τον προγραμματιστή να αλλάξει το στυλ προγραμματισμού του. Τα προγράμματα μπορούν να γραφούν πρώτα για σωστή σειριακή εκτέλεση και στη συνέχεια να προστεθούν οι εντολές *OpenMP*, χωρίς να επηρεαστεί η εκτέλεση στη σειριακή περίπτωση. Έτσι, διαχωρίζεται η εργασία για την επίλυση του προβλήματος απο την παραλληλοποίησή του.

- *Η διαδικασία παραλληλοποίησης είναι καθοδηγούμενη από τον χρήστη.* Αυτό σημαίνει ότι ο μεταφραστής δε χρειάζεται να κάνει διεξοδική ανάλυση του κώδικα, αλλά αρκεί να βασιστεί στις πληροφορίες που του παρέχει ο χρήστης. Με αυτόν τον τρόπο ο χρήστης έχει πλήρη έλεγχο στο τι θα παραλληλιστεί και πώς, καθιστώντας ταυτόχρονα τον μεταφραστή απλούστερο στην υλοποίησή του.

2.2 Μοντέλο εκτέλεσης

Το *OpenMP* χρησιμοποιεί το μοντέλο παράλληλης εκτέλεσης fork-join. Το μοντέλο αυτό επιτρέπει να γράφονται προγράμματα που εκμεταλλεύονται πολλαπλά νήματα εκτέλεσης (threads).

Ένα πρόγραμμα που έχει γραφεί με το *OpenMP* C/C++ API ξεκινά την εκτέλεσή του σαν ένα μόνο νήμα, που ονομάζεται *νήμα-αφέντης (master thread)*. Το νήμα-αφέντης εκτελείται σειριακά μέχρι την πρώτη *παράλληλη περιοχή (parallel construct)*. Η εντολή `parallel` δηλώνει την παράλληλη περιοχή. Μόλις εκτελεστεί μία τέτοια εντολή, το νήμα-αφέντης δημιουργεί μία ομάδα από νήματα και κάθε νήμα ξεκινά την εκτέλεση του κώδικα της παράλληλης περιοχής. Δηλαδή, ο κώδικας αυτός εκτελείται παράλληλα από όλα τα νήματα. Υπάρχουν όμως και ειδικές εντολές που ορίζουν *περιοχές διαμοιρασμού εργασίας (work sharing constructs)*. Στις περιοχές αυτές, το κάθε νήμα της ομάδας αναλαμβάνει ένα *μέρος* της δουλειάς. Στο τέλος μιας παράλληλης περιοχής όλα τα νήματα σταματούν και μόνο το νήμα-αφέντης συνεχίζει να εκτελείται.

2.3 Εντολές (directives)

Το *OpenMP* χρησιμοποιεί τις δηλώσεις `#pragma` για την επέκταση του μεταφραστή της C. Μία εντολή *OpenMP* έχει την εξής μορφή:

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

Στις επόμενες ενότητες ακολουθεί μία περιληπτική περιγραφή των εντολών, όπως αυτές παρουσιάζονται στο πρότυπο. Για καλύτερη κατανόηση, όπου θεωρούμε ότι είναι απαραίτητο, παραθέτουμε κάποια παραδείγματα.

2.4 Η εντολή parallel

Η εντολή `parallel` δηλώνει μία *παράλληλη περιοχή*, η οποία εκτελείται από πολλαπλά νήματα. Αυτή είναι η βασική εντολή που εκκινεί την παράλληλη εκτέλεση δημιουργώντας τα νήματα. Η σύνταξη της εντολής είναι:

`#pragma omp parallel` [*clause*[*clause*] ...] *new-line*
    structured-block

Το *clause* μπορεί να είναι ένα από τα ακόλουθα:

`if` (*scalar-expression*)

`private` (*list*)

`firstprivate` (*list*)

`default (shared | none)`

`shared` (*list*)

`copyin` (*list*)

`reduction` (*operator* : *list*)

Μόλις ένα νήμα συναντήσει μία παράλληλη περιοχή, δημιουργείται μία ομάδα από νήματα στην περίπτωση που ισχύει μία από τις ακόλουθες περιπτώσεις:

- Δεν υπάρχει `if`.

- Η έκφραση του `if` έχει μη-μηδενική τιμή.

Το νήμα αυτό γίνεται ο αφέντης της ομάδας με αριθμό 0 και όλα τα νήματα μαζί με τον αφέντη εκτελούν παράλληλα την περιοχή.

Τα `private`, `firstprivate`, `default`, `shared`, `copyin` αναφέρονται σε μεταβλητές που θα είναι κοινές/ιδιωτικές σε κάθε νήμα, ενώ το `reduction` καθοδηγεί λειτουργίες υποβίβασης από τα δεδομένα των νημάτων. Περισσότερα για αυτά παρακάτω (παράγραφος 2.7).

Παράδειγμα:

```
#pragma omp parallel
    {
        printf("I am a thread!\n");
    }
```

Η εντολή `parallel` θα δημιουργήσει μια ομάδα απο νήματα και όλα τα μέλη της θα εκτελέσουν την εντολή printf.

2.5 Εντολές διαμοιρασμού εργασίας

Μία εντολή διαμοιρασμού εργασίας αναφέρεται σε μια περιοχή κώδικα. Η περιοχή αυτή θα κατανεμηθεί ανάμεσα στα νήματα της ομάδας που συναντούν την εντολή. Δηλαδή, δεν εκτελούν όλα τα νήματα όλες τις εντολές, αλλά καθένα εκτελεί συγκεκριμένα τμήματα. Οι εντολές αυτές δε δημιουργούν νέα νήματα.

2.5.1 for

Η εντολή `for` δηλώνει μία περιοχή η οποία περιλαμβάνει έναν βρόχο του οποίου οι εντολές εκτελούνται παράλληλα. Η σύνταξη της εντολής είναι:

`#pragma omp for` [*clause*[*clause*] ...] *new-line*
  for-loop

Το *clause* μπορεί να είναι ένα από τα ακόλουθα:

`private (`*list*`)`

`firstprivate` (*list*)

`lastprivate` (*list*)

`reduction` (*operator* : *list*)

`ordered`

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

`nowait`

Το `scedule` καθορίζει τον τρόπο με τον οποίο θα κατενεμηθούν οι επαναλήψεις του βρόχου στα νήματα. Το *OpenMP* καθορίζει τρεις τρόπους:

- *static*: Οι επαναλήψεις του βρόχου διαιρούνται σε σταθερά κομμάτια και ανατίθενται με προκαθορισμένο τρόπο στα νήματα.
- *dynamic*: Όπως και πριν, οι επαναλήψεις του βρόχου διαιρούνται σε ίσα κομμάτια μόνο που αυτή τη φορά, όταν ένα νήμα είναι ελεύθερο, ζητά το επόμενο κομμάτι.
- *guided*: Μοιάζει με το dynamic, αλλά το μέγεθος των κομματιών μειώνεται εκθετικά.
- *runtime*: Κατά την εκτέλεση του προγράμματος καθορίζεται ένας απο τους παραπάνω τρόπους.

Το `ordered` επιτρέπει τη σειριακή εκτέλεση ορισμένων εντολών του βρόχου. Οι εντολές αυτές καθορίζονται με την εντολή `ordered` (βλ. παράγραφο 2.6.6).

Το `lastprivate` καθορίζει κάποιες μεταβλητές των οποίων η τελική τιμή θα είναι ίδια με το σειριακό πρόγραμμα. Περισσότερες πληροφορίες θα βρείτε στην παράγραφο 2.7.

Όταν ένα νήμα τελειώσει την εκτέλεση των επαναλήψεων που του αναλογούν, περιμένει και τα υπόλοιπα νήματα πριν εκτελέσει τον κώδικα που ακολουθεί το `for`. Η παράμετρος `nowait` επιτρέπει στο νήμα να συνεχίσει την εκτέλεση του, χωρίς να περιμένει τα υπόλοιπα νήματα.

Παράδειγμα:

```
#pragma omp parallel
 #pragma omp for scedule(static, 10)
    for (i = 1; i < 100; i++)
      a[i] = 0;
```

Δημιουργείται μια ομάδα από νήματα και κάθε νήμα εκτελεί *ορισμένες* επαναλήψεις του βρόχου. Θα δημιουργηθούν κομμάτια των 10 επαναλήψεων, οι επαναλήψεις 1 ως 10 θα δωθούν στο πρώτο νήμα, οι 11–20 στο δεύτερο κ.ο.κ...

2.5.2 sections

Η εντολή `sections` περιγράφει ένα σύνολο από ενότητες (sections) οι οποίες θα διαμοιρασθούν ανάμεσα στα νήματα. Κάθε ενότητα (section) εκτελείται μόνο μία φορά από ένα (τυχαίο) νήμα. Η σύνταξη της εντολής είναι:

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

Το *clause* μπορεί να είναι ένα από τα ακόλουθα:

`private (`*list*`)`

`firstprivate (`*list*`)`

`lastprivate (`*list*`)`

`reduction (`*operator* : *list*`)`

`nowait`

Παράδειγμα:

```
#pragma omp parallel
  #pragma omp sections
    {
      #pragma omp section
        printf("first section\n");
      #pragma omp section
        printf("second section\n");
      #pragma omp section
        printf("third section\n");
    }
```

Έστω ότι έχουμε 2 νήματα. Το πρώτο θα πάρει το πρώτο section το επόμενο το δεύτερο και όποιο από τα δύο τελειώσει πρώτο, θα πάρει το τρίτο. Αν είχαμε περισσότερα απο 3 νήματα τότε τα τρία πρώτα θα έπερναν από ένα section και τα υπόλοιπα απλά θα περίμεναν.

2.5.3 single

Η εντολή `single` καθορίζει ότι η σχετική περιοχή εκτελείται μόνο από ένα νήμα της ομάδας (όχι απαραίτητα το νήμα-αφέντη). Η σύνταξη είναι η ακόλουθη:

`#pragma omp single` [*clause*[*clause*] ...] *new-line*
structured-block

Το *clause* μπορεί να είναι ένα από τα ακόλουθα:

`private (`*list*`)`

`firstprivate (`*list*`)`

`nowait`

Παράδειγμα:

```
#pragma omp parallel
  #pragma omp single
    printf("I am a single thread!\n");
```

Αν και δημιουργείται ομάδα από νήματα, λόγω της εντολής `parallel`, μόνο ένα νήμα θα τυπώσει το μήνυμα.

2.5.4 Συνδυασμός με parallel

Οι εντολές `for`, `sections` και `single` μπορούν να συνδυαστούν με μία παράλληλη περιοχή σε μία μόνο εντολή.

Για παράδειγμα η ακολουθία εντολών

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

είναι ισοδύναμη με

```
#pragma omp parallel for
```

2.6 Συγχρονισμός

Το *OpenMP* παρέχει κάποιες εντολές που αφορούν την ατομικότητα πράξεων, όπως επίσης και εντολές συγχρονισμού που αναγκάζουν κάποιο νήμα να περιμένει και τα υπόλοιπα να φτάσουν στο ίδιο σημείο με αυτό. Τέλος υπάρχουν εντολές που σειριοποιούν κάποια κομμάτια κώδικα.

2.6.1 master

Η εντολή `master` ορίζει μία περιοχή η οποία θα εκτελεστεί μόνο από το νήμα-αφέντη της ομάδας. Η σύνταξη είναι

`#pragma omp master` *new-line*
  structured-block

2.6.2 critical

Η εντολή `critical` ορίζει μία κρίσιμη περιοχή, δηλαδή μια περιοχή που εκτελείται *μόνο από ένα* νήμα κάθε φορά. Η εντολή συντάσσεται ως εξής

`#pragma omp critical` [(*name*)] *new-line*
  structured-block

Το *name* είναι προαιρετικό όρισμα και καθορίζει το όνομα της περιοχής. Ένα νήμα περιμένει κατά την είσοδο σε μία κρίσιμη περιοχή μέχρι να μην εκτελεί κανένα άλλο νήμα κάποια κρίσιμη περιοχή με το ίδιο όνομα. Περιοχές για τις οποίες δε δηλώνεται όνομα, θεωρείται ότι έχουν το ίδιο, «ακαθόριστο» όνομα.

Παράδειγμα:

```
x = 0;
#pragma omp parallel shared(x)
  #pragma omp critical(inc_x)
     x = x + 1;
```

Μετά την εκτέλεση του παραπάνω κώδικα το `x` θα ισούται με τον αριθμό των νημάτων, αφού όλοι θα το αυξήσουν ατομικά και αδιαίρετα.

2.6.3 barrier

Η εντολή `barrier` συγχρονίζει τα νήματα μίας ομάδας. Μόλις ένα νήμα συναντήσει την εντολή, περιμένει μέχρι και τα υπόλοιπα να φτάσουν στο ίδιο σημείο. Η εντολή συντάσσεται ως εξής

`#pragma omp barrier` *new-line*

Παράδειγμα:

```
#pragma omp parallel shared(x)
  x = 0;
  #pragma omp barrier
  #pragma omp critical
     x = x + 1;
```

Χωρίς το `barier` θα μπορούσε κάποιο νήμα να αυξήσει την τιμή του `x` ενώ κάποιο άλλο που έχει καθυστερήσει να τη μηδενίσει αμέσως μετά. Έτσι, στο τέλος του κώδικα η τιμή του `x` είναι απροσδιόριστη. Με το `barrier` είναι εγγυημένο ότι *πρώτα όλα* τα νήματα θα μηδενίσουν το `x` και *μετά* θα αρχίσουν να το αυξάνουν.

2.6.4 atomic

Η εντολή `atomic` βεβαιώνει ότι μία συγκεκριμένη θέση μνήμης ενημερώνεται ατομικά και αδιαίρετα, αντί να εκτίθεται σε ταυτόχρονη εγγραφή από πολλά νήματα. Η σύνταξη είναι

`#pragma omp atomic` *new-line*
expression-stmt

Το *expression-stmt* είναι μία έκφραση και παίρνει μία από τις ακόλουθες μορφές

$x\star = expr$

$x++$

$++x$

$x--$

$--x$

Παράδειγμα:

```
x = 0;
#pragma omp parallel shared(x)
  #pragma omp atomic
     x++;
```

2.6.5 flush

Η εντολή `flush` καθορίζει ένα «δια-νηματικό» σημείο εκτέλεσης, στο οποίο η υλοποίηση απαιτείται να διαβεβαιώσει ότι όλα τα νήματα της ομάδας έχουν συνεπή εικόνα συγκεκριμένων αντικειμένων στη μνήμη. Αυτό σημαίνει ότι προηγούμενοι υπολογισμοί εκφράσεων που αναφέρονται στα αντικείμενα έχουν ολοκληρωθεί και οι υπολογισμοί που έπονται δεν έχουν ξεκινήσει ακόμη. Η σύνταξη της εντολής είναι

`#pragma omp flush` [(*list*)] *new-line*
structured-block

Αν όλα τα αντικείμενα που απαιτούν συγχρονισμό μπορούν να καθοριστούν από μεταβλητές, τότε αυτές δηλώνονται στη λίστα *(list)*. Εαν απουσιάζει η λίστα, τότε η εντολή συγχρονίζει όλα τα κοινά αντικείμενα. Μία τέτοια εντολή (χωρίς λίστα) υπονοείται στις παρακάτω εντολές[1]

[1]Η εντολή δεν υπονοείται σε περίπτωση που έχει δηλωθεί η παράμετρος `nowait`.

- `barrier`
- είσοδος και έξοδος από `critical` και `ordered`
- έξοδος από `parallel`, `for`, `sections` και `single`

2.6.6 ordered

Η εντολή `ordered` ορίζει μια περιοχή η οποία όταν βρίσκεται μέσα σε `for` εκτελείται με τη σειρά που θα εκτελούνταν και στο σειριακό πρόγραμμα. Η σύνταξη είναι

`#pragma omp ordered` *new-line*
structured-block

Παράδειγμα:

```
#pragma omp parallel
  #pragma omp for ordered
    for(i = 0; i < 10; i++) {
      printf("A = %d\n", i);
      #pragma omp ordered
        printf("B = %d\n", i);
    }
```

Τα μηνύματα `A = i` δε θα τυπωθούν απαραίτητα με τη σειρά, ενώ τα μηνύματα `B = i` θα τυπωθούν εγγυημένα με τη σειρά.

2.7 Περιβάλλον δεδομένων

2.7.1 threadprivate

Η εντολή `threadprivate` δηλώνει ότι οι μεταβλητές που βρίσκονται στη λίστα *list* θα είναι (αυτόματα) ιδιωτικές σε κάθε νήμα Με αυτό τον τρόπο, ο χρήστης δε χρειάζεται να τις δηλώνει κάθε φορά. Σύνταξη:

`#pragma omp threadprivate` (*list*) *new-line*

2.7.2 Δηλώσεις ορατότητας δεδομένων

Μερικές εντολές επιδέχονται παραμέτρους που επιτρέπουν στο χρήστη να ελέγχει τις ιδιότητες ορατότητας των μεταβλητών σε μία περιοχή, δηλάδή ποιες θα είναι κοινές, ιδιωτικές κλπ.

private(*list*)

Η δήλωση `private` ορίζει ότι οι μεταβλητές που δηλώνονται στη λίστα που ακολουθεί είναι ιδιωτικές σε κάθε νήμα της ομάδας. Η αρχική τιμή των μεταβλητών είναι ακαθόριστη.

firstprivate(*list*)

Η δήλωση αυτή παρέχει ένα υπερσύνολο της λειτουργικότητας της `private`. Οι μεταβλητές είναι ιδιωτικές αλλά η αρχική τους τιμή είναι η τιμή που είχαν πριν την εντολή αυτή.

lastprivate(*list*)

Η δήλωση αυτή παρέχει ένα υπερσύνολο της λειτουργικότητας της `private`. Οι μεταβλητές είναι ιδιωτικές αλλά η τελική τους τιμή αντιγράφεται στην αρχική μεταβλητή. Ως τελική τιμή προσδιορίζεται η τιμή της τελευταίας επανάληψης του βρόχου είτε η τιμή του τελευταίου λεκτικά `section`.

shared(*list*)

Η δήλωση `private` ορίζει ότι οι μεταβλητές που δηλώνονται στη λίστα που ακολουθεί είναι κοινές για όλα τα νήματα της ομάδας.

default(shared | none)

Καθορίζει την εξ' ορισμού ορατότητα των μεταβλητών.

Η δήλωση `default(shared)` είναι ισοδύναμη με το να τοποθετήσουμε όλες τις ορατές μεταβλητές σε μια δήλωση `shared`.

Η δήλωση `default(none)` απαιτεί να δηλώσουμε ρητά τις ιδιότητες ορατότητας όλων των ορατών μεταβλητών .

reduction(*op* : *list*)

Η δήλωση αυτή ορίζει υποβίβαση στις μεταβλητές που εμφανίζονται στη λίστα. Οι μεταβλητές αυτές είναι κοινές και πάνω σε αυτές εφαρμόζεται ο τελεστής *op*. Ο τελεστής μπορεί να είναι ένας απο τους $+$, $*$, $-$, &, |, $\wedge$, &&, ||. Οι μεταβλητές της λίστας μετατρέπονται σε τοπικές, έτσι ώστε το κάθε νήμα να υπολογίζει μερικά αποτελέσματα. Στο τέλος της περιοχής διαμοιρασμού εργασίας, υπολογίζεται το τελικό αποτέλεσμα στην κοινή μεταβλητή ανάλογα με τον τελεστή.

Παράδειγμα:

```
x = 0;
#pragma omp parallel
 #pragma omp for reduction (+:x)
   for(i = 0; i < 10; i++)
    x = x + 2;
```

Η μεταβλητή `x` με την εντολή `reduction` ορίζεται ως τοπική για κάθε νήμα. Μετά το τέλος των επαναλήψεων που ενέλαβε το κάθε νήμα, η τοπική μεταβλητή `x` θα έχει το μερικό άθροισμα (εξαρτάται απο το πόσες επαναλήψεις εκτέλεσε το καθένα). Τελικά, οι τοπικές μεταβλητές `x` αθροίζονται πάνω στην κοινή μεταβλητή. Μετά το `for` η κοινή μεταβλητή `x` θα έχει το σωστό αποτέλσμα (20).

copyin(*list*)

Η δήλωση αυτή παρέχει έναν μηχανισμό με τον οποίο ανατίθεται η ίδια τιμή στις threadprivate μεταβλητές για κάθε νήμα που εκτελεί την παράλληλη περιοχή. Η τιμή του νήματος-αφέντη αντιγράφεται στα υπόλοιπα νήματα.

2.8 Τοποθέτηση εντολών

Ο τρόπος με τον οποίο οι εντολές τοποθετούνται στο πρόγραμμα πρέπει να υπακούει στους παρακάτω κανόνες:

- Οι εντολές `for`, `sections`, `single`, `master` και `barrier` αφορούν την παράλληλη περιοχή που τις περιέχει, εφόσον υπάρχει, διαφορετικά δεν έχουν κάποιο αποτέλεσμα.

- Η εντολή `ordered` σχετίζεται με το `for` που την περιέχει.

- Η εντολή `atomic` επιφέρει αποκλειστικότητα ανάμεσα σε όλα τα νήματα, όχι μόνο της τρέχουσας ομάδας.

- Η εντολή `critical` επιφέρει αποκλειστικότητα ανάμεσα σε όλες τις κρίσιμες περιοχές σε όλα τα νήματα, όχι μόνο της τρέχουσας ομάδας.

- Μία εντολή δεν συσχετίζεται ποτέ με μία άλλη εντολή εξωτερικά της παράλληλης περιοχής που περιέχει την πρώτη.

2.9 Εμφώλευση

Η εμφώλευση των εντολών πρέπει να υπακούει στους παρακάτω κανόνες:

- Με την εμφάνιση μιας εντολής `parallel` μέσα σε μια παράλληλη περιοχή, σχηματίζεται μία νέα ομάδα που αποτελείται μόνο από το τρέχον νήμα, εκτός και αν έχει ενεργοποιηθεί ο εμφωλευμένος παραλληλισμός.

- Τα `for`, `sections` και `single` που σχετίζονται με το ίδιο `parallel` δεν μπορούν να εμφωλιάζονται μεταξύ τους. Επίσης, δεν επιτρέπεται να εμφανίζονται στις περιοχές των εντολών `critical`, `ordered`, και `master`.

- Κρίσιμες (`critical`) περιοχές με το ίδιο όνομα δεν μπορούν να εμφωλιάζονται.

- `barrier` δεν επιτρέπεται στις περιοχές `for`, `ordered`, `sections`, `single`, `master` και `critical`.

- `master` δεν επιτρέπεται σε `for`, `sections` και `single`.

- `ordered` δεν μπορεί να βρίσκεται μέσα σε `critical`.

- Κάθε εντολή που μπορεί να εκτελεστεί σε παράλληλη περιοχή, μπορεί να εκτελεστεί και έξω από αυτή. Στη δεύτερη περίπτωση, θεωρούμε ότι έχουμε μία ομάδα αποτελούμενη μόνο από το νήμα του κυρίως προγράμματος.

2.10 Runtime βιβλιοθήκη

Το *OpenMP* καθορίζει ένα σύνολο από ρουτίνες βιβλιοθήκης για χρήση στα προγράμματα. Οι ρουτίνες αυτές κατά την εκτέλεση του προγράμματος καθορίζουν παραμέτρους σχετικά με τη δημιουργία νημάτων, επιστρέφουν πληροφορίες σχετικά με την τρέχουσα ομάδα, και γενικότερα διαχειρίζονται το παράλληλο περιβάλλον εκτέλεσης.

2.10.1 Συναρτήσεις παράλληλου περιβάλλοντος

void omp_set_num_threads(int num_threads);

Καθορίζει τον αριθμό νημάτων που θα χρησιμοποιηθούν στις παράλληλες περιοχές που ακολουθούν.

int omp_get_num_threads(void);

Επιστρέφει τον αριθμό νημάτων στην ομάδα που εκτελεί την παράλληλη περιοχή στην οποία βρίσκεται η συνάρτηση αυτή.

int omp_get_max_threads(void);

Επιστρέφει τη μέγιστη τιμή που μπορεί να επιστρέψει η omp_get_num_threads().

int omp_get_thread_num(void);

Επιστρέφει τον αριθμό του νήματος (μέσα στην ομάδα του) που εκτελεί τη συνάρτηση. Η τιμή επιστροφής βρίσκεται μεταξύ 0 και omp_get_num_threads()−1. Για το νήμα-αφέντη επιστρέφεται 0.

int omp_get_num_procs(void);

Επιστρέφει το μέγιστο αριθμό επεξεργαστών τους οποίους θα μπορούσε να χρησιμοποιήσει το πρόγραμμα.

int omp_in_parallel(void);

Επιστρέφει μια μη-μηδενική τιμή εάν κληθεί μέσα από μία παράλληλη περιοχή, διαφορετικά επιστρέφει 0.

void omp_set_dynamic(int dynamic_threads);

Ενεργοποιεί/απενεργοποιεί το δυναμικό καθορισμό του αριθμού νημάτων στις παράλληλες περιοχές.

int omp_get_dynamic(void);

Επιστρέφει μη-μηδενική τιμή εάν είναι ενεργοποιημένος ο δυναμικός καθορισμός του αριθμού νημάτων στις παράλληλες περιοχές, 0 διαφορετικά.

void omp_set_nested(int nested);

Εμεργοποιεί/απενεργοποιεί τον εμφωλιασμένο παραλληλισμό.

void omp_get_nested(void);

Επιστρέφει μη-μηδενική τιμή εάν είναι ενεργοποιημένος ο εμφωλιασμένος παραλληλισμός και 0 διαφορετικά.

2.10.2 Συναρτήσεις για κλειδαριές

Οι κλειδαριές είναι δομές με τις οποίες εξασφαλίζεται ατομικότητα. Το *OpenMP* ορίζει δύο τύπους κλειδαριών: τα locks και τα nestlocks.

- *locks:* Όταν μια κλειδαριά είναι κλειδωμένη, τότε όποιος επιχειρήσει να την ξανακλειδώσει, μπλοκάρει μέχρις ότου να ξεκλειδωθεί.
- *nestlocks:* Αυτού του τύπου οι κλειδαριές διαφέρουν στο ότι αν κάποιος επιχειρήσει να την ξανακλειδώσει, τότε μπλοκάρει μόνο αν δεν ήταν αυτός που την κλείδωσε.

Ακολουθούν οι συναρτήσεις που διαχειρίζονται αυτού του τύπου τις κλειδαριές.

void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

Αρχικοποιεί την κλειδαριά `lock`.

void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);

Απελευθερώνει τους πόρους που καταλαμβάνει η αρχικοποιημένη κλειδαριά `lock`.

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

Παγώνει την εκτέλεση του νήματος που καλεί τη συνάρτηση μέχρι η κλειδαριά να γίνει διαθέσιμη και έπειτα την κλειδώνει.

void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);

Αφαιρεί την κατοχή της κλειδαριάς από το νήμα.

void omp_test_lock(omp_lock_t *lock);
void omp_test_nest_lock(omp_nest_lock_t *lock);

Προσπαθεί να καταλάβει την κλειδαριά χωρίς να παγώσει την εκτέλεση του νήματος.

Κεφάλαιο 3

Ο μεταφραστής μας

Για να υλοποιήσουμε την εργασία αυτή, έπρεπε να ασχοληθούμε με δύο θέματα.

1. Εύρεση μιας αντιστοιχίας μεταξύ ενός προγράμματος *OpenMP* C κι ενός ANSI C, ώστε αυτά να είναι σημασιολογικά ισοδύναμα.

2. Υλοποίηση ενός προγράμματος (μεταφραστής) που αναγνωρίζει τη γλώσσα *OpenMP* C και επιτελεί τον παραπάνω μετασχηματισμό.

Σε αυτό το κεφάλαιο εξετάζουμε την υλοποίηση του προγράμματος μετάφρασης, ενώ η ανάλυση των μετασχηματισμών που επιτελεί γίνεται στο επόμενο.

Ο μεταφραστής που υλοποιήσαμε, πληρεί την έκδοση 1.0 του προτύπου [3] και αφορά τη γλώσσα ANSI C. [1]

Παίρνει ως είσοδο ένα πρόγραμμα γραμμένο σε *OpenMP* C, ερμηνεύει τις εντολές *OpenMP* και παράγει κώδικα σε ANSI C ο οποίος χρησιμοποιεί τη βιβλιοθήκη POSIX threads για τη διαχείριση των νημάτων. Ο κώδικάς του είναι γραμμένος εξολοκλήρου σε C, ενώ για τη λεκτική και συντακτική ανάλυση χρησιμοποιήθηκαν τα εργαλεία flex και bison.

3.1 Λεκτική ανάλυση

Η πρώτη φάση της μετάφρασης του προγράμματος είναι η λεκτική ανάλυση. Ο λεκτικός αναλυτής είναι μια υπορουτίνα η οποία παίρνει σαν είσοδο το πηγαίο πρόγραμμα και κάθε φορά

[1] Κατά την ανάπτυξη του μεταφραστή (Μάρτιος 2002), βγήκε η έκδοση 2.0 του προτύπου, και έχουν επιπρόσθετα υλοποιηθεί ορισμένα από τα νέα χαρακτηριστικά.

που καλείται επιστρέφει την επόμενη λεκτική μονάδα (token) μαζί με έναν κωδικό (αναγνωριστικό) που την χαρακτηρίζει. Για παράδειγμα, στη γραμμή

```
while (1);
```

ένας λεκτικός αναλυτής για την ANSI C θα επέστρεφε κάτι που μοιάζει με το παρακάτω:

token	token id
`while`	WHILE
`(`	LEFT_PAREN
`1`	CONSTANT
`)`	RIGHT_PAREN
`;`	SEMICOLON

Όπως είδαμε στην παράγραφο 2.3, οι εντολές *OpenMP* έχουν την ακόλουθη μορφή:

`#pragma omp` *directive-name* [*clause*[*clause*] ...] *newline*

Για τη λεκτική ανάλυση επομένως είναι απαραίτητη η εισαγωγή κάποιων νέων λεκτικών μονάδων, επιπρόσθετα αυτών που χρειάζονται για την ANSI C. Τα καινούρια token επιστρέφονται μόνο στις γραμμές που περιγράφουν εντολές *OpenMP*. Σε διαφορετική περίπτωση επιστρέφεται το token όπως καθορίζεται από την ANSI C. Υπάρχουν επίσης κάποιες λεκτικές μονάδες οι οποίες έχουν διαφορετική σημασία στην ANSI C και σε μια εντολή *OpenMP*, όπως το `if` και το `for`. Κατά συνέπεια επιστρέφεται διαφορετικό αναγνωριστικό ανάλογα με τη θέση της λεκτικής μονάδας στον κώδικα.

Για τη λεκτική ανάλυση χρησιμοποιήσαμε το flex[10]. Το flex είναι ένα εργαλείο που παίρνει ως είσοδο ένα αρχείο με κανόνες. Οι κανόνες αυτοί αποτελούνται από κανονικές εκφράσεις για την αναγνώριση της λεκτικής μονάδας και από κώδικα C που εκτελείται όταν αναγνωριστεί η μονάδα αυτή (συνήθως επιστρέφεται το αντίστοιχο αναγνωριστικό). Στο παράρτημα Α΄ παρατίθενται οι κανονικές εκφράσεις της υλοποίησής μας μαζί με τα αναγνωριστικά τους.

Ένα σημαντικό πρόβλημα που χρειάστηκε να αντιμετωπίσουμε είναι οι δηλώσεις νέων τύπων που ορίζει ο χρήστης. Όταν η λεκτική μονάδα είναι τύπος δεδομένων, θα πρέπει να επιστρέφεται το κατάλληλο αναγνωριστικό (TYPE_NAME). Με τους βασικούς τύπους της C (int, char, double, ...) δεν υπάρχει πρόβλημα μια και πρόκειται για δεσμευμένες λέξεις, επομένως όποτε συναντάμε μία απ' αυτές τις λέξεις απλά επιστρέφουμε TYPE_NAME. Οι τύποι του χρήστη όμως, δεν είναι γνωστοί εκ των προτέρων και επομένως, όταν συναντάμε μία μη δεσμευμένη λέξη δεν ξέρουμε αν θα επιστρέψουμε TYPE_NAME ή IDENTIFIER (αναγνωριστικό που αντιστοιχεί σε όλες τις μη δεσμευμένες λέξεις). Έτσι, κάθε φορά που συναντάμε δήλωση νέου τύπου, τη φυλάμε σε μία δομή ώστε ο λεκτικός αναλυτής να την συμβουλεύεται για να αποφασίσει αν πρόκειται για τύπο.

3.2 Συντακτική ανάλυση

Η δεύτερη φάση της μετάφρασης είναι η συντακτική ανάλυση. Κατά τη συντακτική ανάλυση οι λεκτικές μονάδες που επιστρέφει ο λεκτικός αναλυτής, εξετάζονται για τον αν σχηματίζουν συντακτικά ορθές φράσεις της γλώσσας. Ο συντακτικός αναλυτής υλοποιήθηκε χρησιμοποιώντας το εργαλείο bison[11]. Το bison παίρνει ως είσοδο ένα σύνολο απο κανόνες που αποτελούν την γραμματική της γλώσσας που αναλύουμε. Ως έξοδο ο bison παράγει ένα πρόγραμμα σε C το οποίο κάνει τη συντακτική ανάλυση που θέλουμε με τον εξής τρόπο: Κάθε φορά καλείται ο λεκτικός αναλυτής και ανάλογα με το ποια λεκτική μονάδα επέστρεψε, εκτελείται ο κατάλληλος κανόνας.

Ανάμεσα στους κανόνες της γραμματικής, σύμφωνα με το εργαλείο bison, μπορούμε να προσθέσουμε και δικό μας κώδικα σε C ο οποίος εκτελείται όταν επαληθευθεί κάποιος κανόνας. Ουσιαστικά, εδώ έχουμε γράψει τον κώδικα που επιτελεί τους απαραίτητους μετασχηματισμούς για την παραγωγή του τελικού κώδικα.

Για την υλοποίησή μας ξεκινήσαμε με την γραμματική της ANSI C όπως ορίζεται στο [1]. Αφού την τροποποιήσαμε και προσθέσαμε τους νέους κανόνες σύμφωνα με το πρότυπο, προέκυψε η γραμματική της *OpenMP* C. Η γραμματική αυτή παρατίθεται στο παράρτημα Β΄.

Σε αυτή τη φάση, τα προβλήματα που είχαμε να αντιμετωπίσουμε ήταν αρκετά.

Αριθμός περασμάτων

Επειδή θέλαμε ο μεταφραστής να είναι γρήγορος, έπρεπε ο τελικός κώδικας να παράγεται με όσο το δυνατόν λιγότερα περάσματα του αρχικού αρχείου.

Για παράδειγμα, κατά την παραγωγή τελικού κώδικα, χρειάζεται να γνωρίζουμε εκ των προτέρων ποιες μεταβλητές χρησιμοποιούνται σε κάθε block, έτσι ώστε πριν από αυτό να κάνουμε κάποιες δηλώσεις και αρχικοποιήσεις [2]. Ο προφανής τρόπος είναι να διατρέξουμε μία φορά το αρχείο κρατώντας τις πληροφορίες αυτές. Έτσι, όταν το ξανα-διατρέξουμε, θα γνωρίζουμε ποιες μεταβλητές χρησιμοποιούνται σε κάθε block *πριν* το επεξεργαστούμε.

[2] Το παράδειγμα αυτό αποτελεί μια απλουστευμένη εκδοχή των προβλημάτων που αντιμετωπίσαμε.

Βρήκαμε, λοιπόν, έναν τρόπο, ώστε ο τελικός κώδικας να παράγεται με ένα πέρασμα, *ταυτόχρονα* με τη συντακτική ανάλυση. Επειδή γίνονται μετακινήσεις και μετασχηματισμοί κώδικα, το τελικό αποτέλεσμα αποτελείται από πολλά αρχεία, στα οποία γράφουμε ταυτόχρονα κατά τη διάρκεια της μετάφρασης.

Έτσι, την παραπάνω περίπτωση την αντιμετωπίζουμε ως εξής. Πριν την είσοδο στο block κάνουμε `#include` ένα αρχείο το οποίο θα το δημιουργήσουμε κατά την επεξεργασία του block που ακολουθεί. Με το που συναντάμε μία μεταβλητή, τοποθετούμε τη δήλωση/αρχικοποίησή της σε αυτό το βοηθητικό αρχείο. Όταν το τελικό αποτέλεσμα το περάσουμε από τον προεπεξεργαστή της C, οι δηλώσεις/αρχικοποιήσεις θα είναι στη σωστή θέση.

Αναφορά λαθών

Το επόμενο πρόβλημα αφορούσε την αναφορά λαθών. Επειδή το αρχικό πρόγραμμα έχει περάσει από τον προεπεξεργαστή της C, έχουν προστεθεί σε αυτό πολλές γραμμές κώδικα (εξαιτίας των αρχείων που γίνονται `#include`). Σε περίπτωση που θέλουμε να αναφέρουμε κάποιο λάθος, δε γνωρίζουμε ούτε σε ποια γραμμή, ούτε σε ποιο αρχείο βρισκόταν. Αυτό το αντιμετωπίζουμε με το να εκμεταλλευόμαστε κάποιες πληροφορίες που τοποθετεί στο προεπεξεργασμένο αρχείο ο προεπεξεργαστής της C. Συγκεκριμένα, ο προεπεξεργαστής πριν από κάθε αρχείο που συμπεριλαμβάνει, προσθέτει μια γραμμή της μορφής:

αριθμος_γραμμής όνομα_αρχείου

Έτσι, διαβάζοντας τα στοιχεία αυτά, μπορούμε να καθορίσουμε επακριβώς τη θέση του σφάλματος.

Ορατότητα μεταβλητών

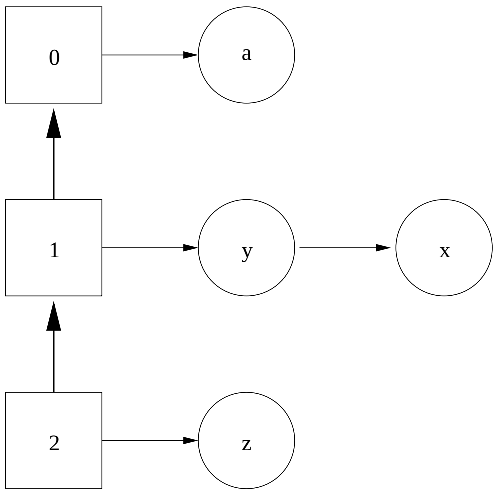
Ένα άλλο θέμα που μας απασχόλησε, ήταν η ορατότητα των μεταβλητών. Συγκεκριμένα, για κάθε μεταβλητή πρέπει να γνωρίζουμε πού έχει δηλωθεί (π.χ. καθολική ή τοπική). Επίσης, πρέπει να ξέρουμε τον τύπο της και το αν είναι κοινή/ιδιωτική ανάμεσα στα νήματα, ώστε να μπορούμε να κάνουμε επιπλέον δηλώσεις. Οι μεταβλητές που είναι δηλωμένες στο ίδιο block φυλάσσονται σε μία συνδεδεμένη λίστα. Οι λίστες αυτές είναι συνδεδεμένες ανάλογα με το πώς είναι εμφωλιασμένα τα block. Το σχήμα 3.1 παρουσιάζει τα περιεχόμενα των παραπάνω δομών για το παράδειγμα που ακολουθεί, όταν ο συντακτικός αναλυτής βρίσκεται στην εντολή `printf`.

```
int a;

int main()
{
    int x, y;

    if (x == 5) {
        int z;

        printf("hello, world!\n");
    }
}
```



Σχήμα 3.1: Ορατότητα μεταβλητών

Κεφάλαιο 4

Μετασχηματισμοί

4.1 Χειρισμός νημάτων με τη βιβλιοθήκη POSIX threads

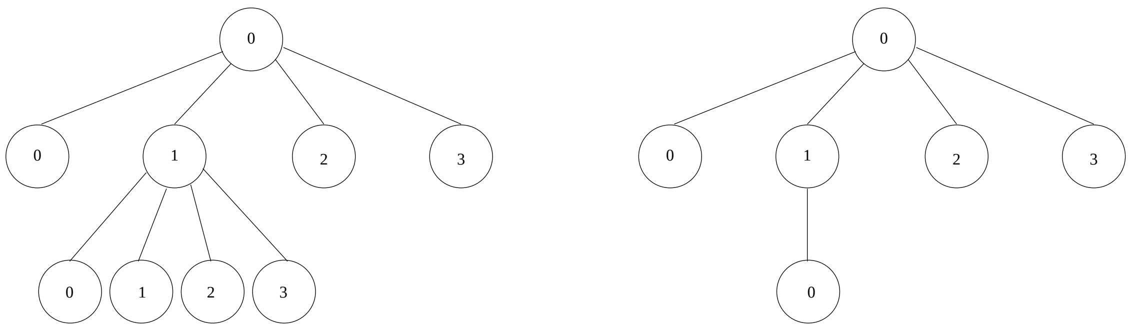
Ένα πρόγραμμα ξεκινά την εκτέλεσή του ως ένα μόνο νήμα (νήμα-αφέντης). Για να δημιουργήσουμε επιπλέον νήματα, θα πρέπει να χρησιμοποιήσουμε τη συνάρτηση `pthread_create()`. Ο κώδικας που εκτελεί το νήμα που δημιουργείται, θα πρέπει να βρίσκεται μέσα σε μια συνάρτηση την οποία περνάμε ως όρισμα στην `pthread_create()`. Προφανώς τα νήματα επικοινωνούν μεταξύ τους μέσω καθολικών μεταβλητών. Ένα νήμα τερματίζει όταν επιστρέψει η συνάρτηση που εκτελεί. Το νήμα-αφέντης μπορεί να περιμένει τον τερματισμό κάποιου άλλου νήματος εκτελώντας την συνάρτηση `pthread_join()`. Τέλος, στην υλοποίησή μας έχουμε χρησιμοποιήσει ρουτίνες από τη βιβλιοθήκη που διαχειρίζονται σημαφόρους και μεταβλητές συνθήκης (για συγχρονισμό).

4.2 parallel

Όπως είδαμε, με την εντολή parallel το νήμα-αφέντης δημιουργεί μια ομάδα από νήματα που εκτελούν όλα τον ίδιο κώδικα μαζί με αυτόν. Κάποια από τα νήματα ενδέχεται να ξανασυναντήσουν εντολή parallel. Σε αυτή την περίπτωση, αν υποστηρίζεται εμφωλιασμένος παραλληλισμός, τότε το νήμα που συνάντησε την εμφωλιασμένη εντολή parallel, ως αφέντης θα δημιουργήσει νέα ομάδα. Αν δεν υποστηρίζεται, τότε δε θα δημιουργηθούν νέα νήματα και τον κώδικα θα εκτελέσει μόνο το νήμα που συνάντησε την εντολή.

Για παράδειγμα θεωρείστε τον παρακάτω κώδικα:

```
#pragma omp parallel
{
    int myid = omp_get_thread_num();
    if (myid == 1) {
        #pragma omp parallel
        {
            ...
        }
    }
}
```

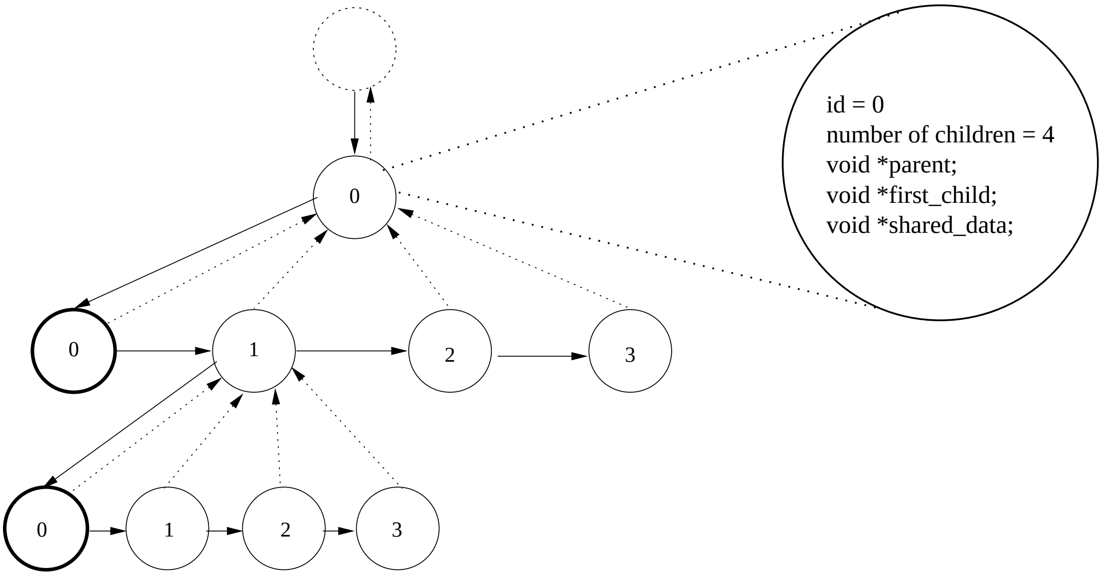


Σχήμα 4.1: Δένδρο νημάτων όταν υποστηρίζεται εμφωλισμένος παραλληλισμός (αριστερά) και όταν δεν υποστηρίζεται (δεξιά)

Όπως βλέπουμε στην εικόνα 4.1, η δημιουργία ομάδων σχηματίζει μια δενδρική δομή, με κάθε νήμα να αντιστοιχεί σε έναν κόμβο.

Ο κώδικας που παράγεται από τον μεταφραστή μας, διατηρεί και διαχειρίζεται την παραπάνω δομή με τη μορφή που παρουσιάζει το σχήμα 4.2. Όταν ένα νήμα δημιουργεί μία ομάδα, τότε ο κόμβος αυτός αντιγράφεται ως το πρώτο παιδί (id 0). Με έντονη γραμμή φαίνονται οι κόμβοι οι οποίοι είναι αντίγραφα του νήματος-αφέντη. Κάθε κόμβος αντιστοιχεί σε ένα νήμα, εκτός από τα νήματα που είναι αντίγραφα. Σε κάθε κόμβο αποθηκεύονται διάφορες πληροφορίες που περιλαμβάνουν την ταυτότητα (id) του νήματος, δείκτη στον πατέρα, δείκτες σε κοινές μεταβλητές καθώς και άλλα δεδομένα στα οποία θα αναφερθούμε παρακάτω.

Μόλις συναντήσουμε την εντολή parallel, ο κώδικας της παράλληλης περιοχής μεταφέρεται σε μία συνάρτηση έτσι ώστε να εκτελεστεί από τα νήματα που θα δημιουργηθούν. Στη θέση



Σχήμα 4.2: Δενδρική δομή όταν υποστηρίζεται εμφωλιασμένος παραλληλισμός

του κώδικα που μετακινήθηκε, παράγεται νέος κώδικας ο οποίος εκτελείται από το νήμα-αφέντη και κάνει τα εξής:

- Ενημερώνει την δενδρική δομή σχετικά με τα νέα νήματα.
- Δημιουργεί τα νήματα και εκτελεί τον κώδικά τους.
- Περιμένει τον τερματισμό τους.
- Επαναφέρει τη δενδρική δομή.

Έστω η παρακάτω συνάρτηση:

```
void parallel()
{
#pragma omp parallel
    {
        printf("I am a thread!\n");
    }
}
```

Ο μεταφραστής θα μετασχηματίσει τη συνάρτηση αυτή σε:

```
void parallel ()
{
/* #pragma omp parallel  */
  {
    /* declare a struct for shared vars */
    _OMP_PARALLEL_DECL_VARSTRUCT (parallel_parallel_0);
    /* create team of threads */
    _omp_create_team (_OMP_THREAD, parallel_parallel_0,
                      (void *) &parallel_parallel_0_var);
    _omp_destroy_team (_OMP_THREAD->parent);
  }
}
```

Η `_omp_create_team()` ενημερώνει τη δενδρική δομή και δημιουργεί τα νήματα της νέας ομάδας. Η `_omp_destroy_team()` περιμένει τα νήματα να τελειώσουν την εκτέλεσή τους. Το `_OMP_THREAD` είναι ένα macro που επιστρέφει δείκτη στον κόμβο της δομής που αφορά το νήμα που εκτελεί τον κώδικα αυτό.

Τα νήματα της ομάδας εκτελούν τον κώδικα της παράλληλης περιοχής ο οποίος έχει μετακινηθεί στη συνάρτηση `parallel_parallel_0()`. Τα ονόματα αυτών των συναρτήσεων παράγονται με τον εξής τρόπο: Πρώτα παραθέτουμε το όνομα της συνάρτησης στην οποία συναντήθηκε εντολή parallel. Στη συνέχεια παραθέτουμε τη λέξη parallel και τέλος, έναν αριθμό που χαρακτηρίζει μοναδικά την παράλληλη περιοχή μέσα στη συνάρτηση που την περιέχει. Έτσι, για το παραπάνω παράδειγμα έχουμε:

```
void *parallel_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  {
    printf ("I am a thread!\n");
  }
  return 0;
}
```

4.2.1 shared (κοινές μεταβλητές)

Όπως αναφέραμε, η παράμετρος shared ορίζει τις κοινές μεταβλητές ανάμεσα στα νήματα. Εάν πρόκειται για καθολικές μεταβλητές, τότε δεν υπάρχει πρόβλημα. Σε διαφορετική περίπτωση παράγεται μία δομή με δείκτες στις κοινές μεταβλητές. Η δομή αυτή φυλάσσεται στον πατέρα κάθε ομάδας και αρχικοποιείται πριν τη δημιουργία των νημάτων. Όταν ένα νήμα αναφέρεται σε μια κοινή μεταβλητή, η αναφορά αυτή γίνεται μέσω του δείκτη που υπάρχει στη δομή.

Έστω ο παρακάτω κώδικας:

```
int global_a;

void parallel_shared()
{
    int local_b;
#pragma omp parallel shared(global_a, local_b)
    {
        global_a++;
        local_b++;
    }
}
```

Επειδή η `local_b` είναι κοινή μεταβλητή αλλά όχι καθολική, παράγεται η εξής δομή:

```
typedef struct {
  int (*local_b);
} parallel_shared_parallel_0_vars;
```

Το όνομα της δομής παράγεται με τον ίδιο τρόπο που παράγονται και τα ονόματα συναρτήσεων μόνο που στον τέλος παρατίθεται η λέξη vars. Ο αρχικός κώδικας αντικαθίσταται με:

```
int global_a;

void parallel_shared ()
{
  int local_b;
/* #pragma omp parallel  shared(global_a, local_b) */
  {
    /* declare a struct for shared vars */
    _OMP_PARALLEL_DECL_VARSTRUCT (parallel_shared_parallel_0);
    _OMP_PARALLEL_INIT_VAR (parallel_shared_parallel_0, local_b);
    /* create team of threads */
    _omp_create_team (_OMP_THREAD, parallel_shared_parallel_0,
                      (void *) &parallel_shared_parallel_0_var);
    _omp_destroy_team (_OMP_THREAD->parent);
  }
}
```

Το macro _OMP_PARALLEL_DECL_VARSTRUCT() δηλώνει μία μεταβλητή για την παραπάνω δομή. Το _OMP_PARALLEL_INIT_VAR() αρχικοποιεί τον δείκτη που περιέχει η δομή, ώστε να δείχνει στην πραγματική μεταβλητή. Τέλος, η δομή αυτή με τις κοινές μεταβλητές περνιέται ως όρισμα στην _omp_create_team() η οποία με τη σειρά της θα την τοποθετήσει στη δενδρική δομή (στον κόμβο-πατέρα, πριν δημιουργήσει τα νήματα).

Ας έρθουμε τώρα στη συνάρτηση που παράγεται:

```
void *parallel_shared_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  int (*local_b) = &_OMP_VARREF (parallel_shared_parallel_0, local_b);
  {
    global_a++;
    (*(local_b))++;
  }
  return 0;
}
```

Η προσπέλαση των κοινών μεταβλητών γίνεται με τη μακροεντολή _OMP_VARREF(). Για λόγους ταχύτητας στην αρχή της συνάρτησης δηλώνονται τοπικοί δείκτες (με τα ίδια ονόματα) στις κοινές μεταβλητές και αρχικοποιούνται. Έτσι, η προσπέλαση γίνεται πιο απλά με (*<όνομα μεταβλητής>). Υπενθυμίζουμε ότι για τις καθολικές κοινές μεταβλητές δεν αλλάζει ο κώδικας.

4.2.2 private (ιδιωτικές μεταβλητές)

Κάθε νήμα πρέπει να διατηρεί ξεχωριστό αντίγραφο για τις ιδιωτικές μεταβλητές. Επομένως, αρκεί να τις επαναδηλώσουμε στην αρχή της συνάρτησης του νήματος.

Για τον κώδικα

```
void parallel_private()
{
    int x;
#pragma omp parallel private(x)
    {
```

```
        x++;
    }
}
```

η συνάρτηση νήματος που δημιουργείται είναι:

```
void *parallel_private_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  int x;
  {
    x++;
  }
  return 0;
}
```

Παρατηρείστε ότι η μεταβλητή x δηλώνεται πάλι μέσα στη συνάρτηση.

4.2.3 firstprivate

Δηλώνεται δείκτης στη δομή όπως στο shared, αλλά γίνεται και τοπική δήλωση όπως στο private. Για όσες από αυτές τις μεταβλητές είναι πίνακες, παράγεται κώδικας που τις αρχικοποιεί μέσω του δείκτη που κρατήσαμε, και τοποθετείται πριν την πρώτη εκτελέσιμη εντολή της παράλληλης περιοχής. Οι υπόλοιπες μεταβλητές αρχικοποιούνται κατά τη δήλωσή τους (πάλι μέσω του δείκτη).

Παράδειγμα:

```
void parallel_firstprivate()
{
    int x, a[5];
#pragma omp parallel firstprivate(x,a,global_a)
    {
        int foo;

        x++;
        global_a++;
        foo = a[1];
    }
}
```

Η δομή με τους δείκτες που δημιουργείται είναι:

```
typedef struct {
  int (*x);
  int (*a)[5];
} parallel_firstprivate_parallel_0_vars;
```

Και η συνάρτηση του νήματος:

```
void *parallel_firstprivate_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  int a[5];
  int x = _OMP_VARREF (parallel_firstprivate_parallel_0, x);

  int (*_omp_firstlastprivate_global_a) = &global_a;
  int global_a = *_omp_firstlastprivate_global_a;

  memcpy (a, &_OMP_VARREF (parallel_firstprivate_parallel_0, a), sizeof (a));
  {
    int foo;
    x++;
    global_a++;
    foo = a[1];
  }
  return 0;
}
```

Όπως βλέπουμε, για την απλή μεταβλητή `x` έγινε δήλωση με αρχικοποίηση. Για τον πίνακα `a` έγινε δήλωση και στην πρώτη εκτελέσιμη εντολή αρχικοποίηση (με την `memcpy()`). Σχετικά με την καθολική μεταβλητή `global_a` ερχόμαστε αντιμέτωποι με το εξής πρόβλημα. Μετά την τοπική δήλωση (με το ίδιο όνομα) δεν μπορούμε να έχουμε πρόσβαση στην καθολική μεταβλητή. Γι' αυτό κρατάμε έναν τοπικό δείκτη στην τελευταία, πριν την τοπική επαναδήλωση. Έτσι, η αρχικοποίηση γίνεται μέσω αυτού του δείκτη.

4.2.4 reduction

Στην αρχή του κώδικα της παράλληλης περιοχής δηλώνονται οι μεταβλητές που καθορίζονται με την παράμετρο αυτή και αρχικοποιούνται στην κατάλληλη τιμή ανάλογα με τον τελεστή υποβίβασης. Πριν το τέλος της παράλληλης περιοχής, παράγεται κώδικας που εφαρμόζει τον τελεστή ανάμεσα στην κοινή μεταβλητή και το τοπικό αντίγραφο. Ο κώδικας αυτός προστατεύεται με έναν δυαδικό σημαφόρο για λόγους ανταγωνισμού. Στην πραγματικότητα, δηλώνεται ένας πίνακας τέτοιων σημαφόρων. Κάθε στοιχείο του πίνακα αντιστοιχεί σε μία εντολή που περιέχει την παράμετρο `reduction`.

Παράδειγμα:

```
void parallel_reduction()
{
    int z;
#pragma omp parallel reduction(+: z)
    {
        z = 1;
    }
}
```

Η συνάρτηση του νήματος που παράγεται είναι:

```
void *parallel_reduction_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  int z = 0;
  {
    z = 1;
  }
  pthread_mutex_lock (&_omp_reduction_lock[0]);
  _OMP_VARREF (parallel_reduction_parallel_0, z) += z;
  pthread_mutex_unlock (&_omp_reduction_lock[0]);
  return 0;
}
```

4.2.5 if

Παράγεται κώδικας που υπολογίζει την συνθήκη του `if`. Αν η συνθήκη είναι αληθής εκτελείται ο κώδικας που δημιουργεί τα νήματα όπως περιγράψαμε παραπάνω. Διαφορετικά καλείται η συνάρτηση του νήματος χωρίς να δημιουργηθούν επιπλέον νήματα, οπότε ουσιαστικά τον κώδικα τον εκτελεί μόνο ο γονέας.

Παράδειγμα:

```
void parallel_if()
{
    int k;
#pragma omp parallel if (k > 0)
    {
        k = 0;
    }
}
```

Ο παραπάνω κώδικας αντικαθίσταται με:

```
void parallel_if ()
{
  int k;
/* #pragma omp parallel  if (k > 0) */
  {
    _OMP_PARALLEL_DECL_VARSTRUCT (parallel_if_parallel_0);
    _OMP_PARALLEL_INIT_VAR (parallel_if_parallel_0, k);
    if (k > 0) {
      _omp_create_team (_OMP_THREAD, parallel_if_parallel_0,
          (void *) &parallel_if_parallel_0_var);
      _omp_destroy_team (_OMP_THREAD->parent);
    } else {
      void *tmp = _OMP_THREAD->sdn;
      _OMP_THREAD->shared_data = (void *) &parallel_if_parallel_0_var;
      _OMP_THREAD->sdn = _OMP_THREAD;
      parallel_if_parallel_0 ((void *) _OMP_THREAD);
      _OMP_THREAD->sdn = tmp;
    }
  }
}
```

Αυτό που πρέπει επισημάνουμε είναι ότι όταν δεν ισχύει η συνθήκη, *δεν* δημιουργούνται νήματα και απλά καλείται η συνάρτηση που δημιουργήθηκε. Όμως για να γίνει σωστά η κλήση, μια και δεν αλλάζει η δενδρική δομή, τα κοινά δεδομένα πρέπει να είναι τα σωστά. Έτσι, πριν την κλήση, τροποποιείται ο κόμβος που αντιστοιχεί στο νήμα και επαναφέρεται μετά.

4.2.6 copyin

Επειδή το `copyin` σχετίζεται άμεσα με τις threadprivate μεταβλητές, η ανάλυσή του γίνεται στην παράγραφο 4.12.1.

4.2.7 Εμφωλιασμένα parallel

Η υλοποίησή μας δεν υποστηρίζει εμφωλιασμένο παραλληλισμό. Σε περίπτωση εμφωλιασμένης παράλληλης περιοχής δημιουργείται ομάδα που αποτελείται μόνο από ένα νήμα. [1] Έτσι, δεν υπάρχει λόγος να γίνει μετακίνηση κώδικα στις *λεκτικά* εμφωλιασμένες παράλληλες περιοχές. Κατά συνέπεια δε χρειάζεται να κρατάμε δείκτες στις κοινές μεταβλητές μια και η αναφορά σε αυτές δεν τροποποιείται.

Παράδειγμα:

```
void parallel_nested()
{
#pragma omp parallel
    {
        int x;
#pragma omp parallel
        {
            x++;
        }
    }
}
```

Η συνάρτηση που παράγεται εξαιτίας της *πρώτης* παράλληλης περιοχής είναι:

[1] Και αυτή η περίπτωση είναι σύμφωνη με το πρότυπο *OpenMP*.

```
void *parallel_nested_parallel_0 (void *_omp_thread_data)
{
  int _omp_dummy = _omp_assign_key (_omp_thread_data);
  {
    int x;
/* #pragma omp parallel  */
    _omp_create_team (_OMP_THREAD, 0, _OMP_THREAD->sdn->shared_data);
    {
      {
    x++;
      }
    }
    _omp_destroy_team (_OMP_THREAD->parent);
  }
  return 0;
}
```

Παρατηρείστε ότι στην εμφωλιασμένη παράλληλη περιοχή δε μετακινείται ο κώδικας.

4.3 single

Όπως είδαμε, η εντολή `single` ορίζει μία περιοχή η οποία εκτελείται μόνο από ένα νήμα της ομάδας. Στη δική μας υλοποίηση, η περιοχή αυτή εκτελείται από το πρώτο νήμα που θα τη συναντήσει. Η παρακάτω περιοχή `single`

```
#pragma omp single
    printf("I am a single thread!\n");
```

μετασχηματίζεται σε

```
/* #pragma omp single  */
  {
/* the first thread executes the code */
    if (_omp_run_single (0)) {
      printf ("I am a single thread!\n");
    }
    _omp_barrier_wait (&_OMP_THREAD->parent->barrier);
    _omp_flush_all ();      /* implied flush */
  }
```

Η ρουτίνα `_omp_run_single()` εξετάζει αν το συγκεκριμένο νήμα μπορεί να εκτελέσει την περιοχή. Όμως ο έλεγχος αυτός δεν είναι και τόσο απλός, γιατί έχουμε να αντιμετωπίσουμε τα εξής προβλήματα:

- Ο κώδικας της περιοχής μπορεί να εκτελείται από διαφορετικές ομάδες. Ένα απλό παράδειγμα τέτοιας περίπτωσης είναι το παρακάτω:

```
#pragma omp parallel
 #pragma omp parallel
   #pragma omp single
     printf("Hello\n");
```

  Στην περίπτωση αυτή, το πρώτο parallel δημιουργεί μια ομάδα των n νημάτων. Το δεύτερο parallel (επειδή είναι εμφωλιασμένο) δημιουργεί n *ανεξάρτητες* ομάδες. Κάθε μία από αυτές πρέπει να εκτελέσει από μία φορά το `printf`. Έτσι συνολικά, το "Hello" πρέπει να τυπωθεί n φορές. Επομένως, η πληροφορία σχετικά με το αν θα εκτελεστεί η περιοχή πρέπει να σχετίζεται με την ομάδα (νήμα-αφέντης) και όχι με την ίδια την περιοχή.

- Η ίδια περιοχή `single` μπορεί να συναντηθεί πολλές φορές από την ίδια ομάδα, όπως δείχνει το παρακάτω παράδειγμα, όπου ο κώδικας της περιοχής βρίσκεται μέσα σε συνάρτηση.

```
void print_message()
{
#pragma omp single nowait
    printf("Hello\n");
}

void main()
{
  #pragma omp parallel
    {
        /* A */
```

```
        print_message();
        /* B */
        print_message();
        print_message();
        /* C */
    }
}
```

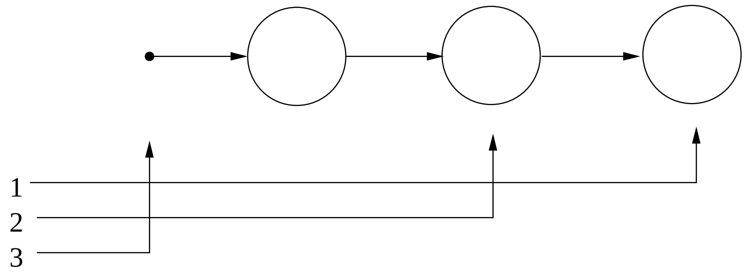
Ο κώδικας αυτός θα πρέπει να τυπώσει 3 φορές το μήνυμα. Το πρόβλημα είναι ότι δεν μπορούμε να αποθηκεύσουμε, για παράδειγμα, μία boolean μεταβλητή στο νήμα αφέντη, ώστε να γνωρίζουμε αν μία ομάδα έχει εκτελέσει ή όχι την περιοχή. Θεωρήστε την παρακάτω περίπτωση:

Έστω ότι κάποιο νήμα έχει καθυστερήσει και βρίσκεται στο σημείο A, ενώ τα υπόλοιπα βρίσκονται στο C. Όταν καλέσει και αυτό την `print_message()`, θα πρέπει να γνωρίζει ότι και τις 3 φορές η περιοχή του `single` έχει εκτελεστεί. Αυτό θα μπορούσε να επιτευχθεί αν κρατούσαμε τρεις boolean μεταβλητές. Έτσι ο αριθμός των μεταβλητών σχετίζεται με το πόσες φορές συναντάμε το συγκεκριμένο single κατά την εκτέλεση. Προτιμήσαμε λοιπόν μια δυναμική δομή για να κρατάμε αυτές τις πληροφορίες.

Στην υλοποίησή μας αντιμετωπίζουμε τα παραπάνω προβλήματα με τον εξής τρόπο. Το νήμα αφέντης διατηρεί μία ουρά (FIFO) στον κόμβο του. Κάθε κόμβος της ουράς αυτής αναπαριστά μια εκτέλεση/συνάντηση της εντολής. Το πρώτο νήμα της ομάδας που συναντά (και εκτελεί) την περιοχή, προσθέτει έναν κόμβο στην ουρά (όταν ξανασυναντήσει την ίδια περιοχή προσθέτει και δεύτερο κόμβο κ.ο.κ.). Επειδή η πληροφορία για μια συνάντηση δε χρειάζεται πλέον, όταν η εντολή έχει συναντηθεί από όλα τα νήματα της ομάδας, το τελευταίο νήμα που συναντά την περιοχή σβήνει τον κόμβο που είχε προστεθεί. Με αυτόν τον τρόπο διατηρείται μία λίστα με τις «ανοιχτές εκτελέσεις» [2] της περιοχής. Τα νήματα της ομάδας γνωρίζουν σε ποια εκτέλεση βρίσκονται διατηρώντας έναν δείκτη στον αντίστοιχο κόμβο.

Στο παραπάνω παράδειγμα, έστω ότι η ομάδα αποτελείται από 3 νήματα, με το πρώτο να βρίσκεται στο σημείο C, το δεύτερο στο B και το τρίτο στο A. Τότε στην ουρά θα υπάρχουν 3 κόμβοι. Το πρώτο νήμα θα δείχνει στον τρίτο, το δεύτερο στον δεύτερο και το τρίτο πουθενά, όπως φαίνεται και στο σχήμα 4.3.

[2] Με τον όρο αυτό εννοούμε το ότι η συγκεκριμένη εκτέλεση της εντολής (π.χ. single) δεν έχει ολοκληρωθεί από όλα τα νήματα της ομάδας.



Σχήμα 4.3: Ουρά «ανοιχτών εκτελέσεων»

Μόλις το τρίτο (τελευταίο) νήμα φτάσει στο σημείο B, θα αφαιρέσει τον πρώτο κόμβο, αφού αυτός δεν είναι πια απαραίτητος. Όταν όλα τα νήματα έχουν φτάσει στο σημείο C, η ουρά θα είναι κενή. Τέλος, πρέπει να σημειώσουμε ότι μια τέτοια ουρά διατηρείται για όλες τις περιοχές `single` που υπάρχουν στο πρόγραμμα.

4.3.1 private, firstprivate

Οι παράμετροι αυτοί έχουν υλοποιηθεί παρόμοια με του parallel.

4.3.2 nowait

Απλά δεν παράγεται κώδικας barrier (βλ. παράγραφο 4.10).

4.4 sections

Κάθε ενότητα (section) απαριθμείται και μετακινείται μεσα σε ένα `switch-case`. Κάθε νήμα ζητά την επόμενη δουλειά και εκτελεί το κατάλληλο `case`, μέχρι να εκτελεστούν όλες οι ενότητες. Για παράδειγμα ο παρακάτω κώδικας

```
#pragma omp sections
    {
#pragma omp section
        printf("first section\n");
#pragma omp section
        printf("second section\n");
#pragma omp section
        printf("third section\n");
    }
```

μετασχηματίζεται σε

```
/* #pragma omp sections  */
  {
    {
      int _omp_section_job;

      _omp_init_sections (0, _omp_num_section[0]);
      while (1) {
            _omp_section_job = _omp_get_next_section (0);
            if (_omp_section_job < 0) break;
            switch (_omp_section_job) {
            case 0:
              printf ("first section\n");
              break;
            case 1:      /* section */
              printf ("second section\n");
              break;
            case 2:      /* section */
              printf ("third section\n");
              break;
            }            /* switch */
      }             /* while */
    }
    _omp_barrier_wait (&_OMP_THREAD->parent->barrier);
    _omp_flush_all ();      /* implied flush */
  }
```

Και σε αυτή την περίπτωση, υφίστανται τα ίδια προβλήματα με το `single` σχετικά με τις πολλαπλές εκτελέσεις της περιοχής. Τα προβλήματα αυτά αντιμετωπίζονται με τον ίδιο τρόπο (ουρές) μόνο που επιπρόσθετα, κάθε κόμβος της ουράς περιέχει πληροφορία σχετικά με τις εναπομείναντες ενότητες. Η ρουτίνα `_omp_init_sections()` αρχικοποιεί και διαχειρίζεται τις ουρές (προσθέτει/διαγράφει κόμβους είτε προχωρά τους δείκτες). Η ρουτίνα `_omp_get_next_section()` επιστρέφει την επόμενη ενότητα απ' τον κατάλληλο κόμβο (μειώνοντας την τιμή του κατά ένα).

4.4.1 private, firstprivate, reduction

Η υλοποίηση των παραμέτρων αυτών έχει γίνει όπως και στο `parallel`.

4.4.2 lastprivate

Το νήμα που ανέλαβε την τελευταία (λεκτικά) περιοχή, ενημερώνει τη lastprivate μεταβλητή.

Παράδειγμα:

```
    int i;
#pragma omp sections lastprivate(i)
    {
#pragma omp section
        i = 1;
#pragma omp section
        i = 2;
#pragma omp section
        i = 3;
    }
```

Μετασχηματισμός:

```
  int i;
/* #pragma omp sections  lastprivate(i) */
  {
    int (*_omp_firstlastprivate_i) = &i;
    int i;
    {
      int _omp_section_job;

      _omp_init_sections (1, _omp_num_section[1]);
      while (1) {
            _omp_section_job = _omp_get_next_section (1);
            if (_omp_section_job < 0) break;
            switch (_omp_section_job) {
            case 0:
              i = 1;
              break;
            case 1:     /* section */
              i = 2;
              break;
            case 2:     /* section */
              i = 3;
              break;
            }           /* switch */
            if (_omp_section_job == 2) {
              *_omp_firstlastprivate_i = i;
            }
```

```
    }            /* while */
   }
   _omp_barrier_wait (&_OMP_THREAD->parent->barrier);
   _omp_flush_all ();      /* implied flush */
 }
```

4.4.3 nowait

Απλά δεν παράγεται κώδικας barrier (όπως και στο `single`).

4.5 for

Η υλοποίηση του `for` ακολουθεί παρόμοια λογική με το `sections`. Υπάρχει ένα `while` σε κάθε επανάληψη του οποίου ένα νήμα ζητά την επόμενη δουλειά (η οποία είναι ένα μέρος του βρόχου for) και την εκτελεί. Για παράδειγμα ο κώδικας

```
    int i;
#pragma omp for schedule(dynamic,2)
    for (i = 0; i < 100; i += 3)
        printf("%d\n", i);
```

μετατρέπεται στον

```
  int i;
/* #pragma omp for  schedule(dynamic, 2) */
  {
    int i;
    int _omp_lb, _omp_ub, _omp_incr;
    int _omp_num_job = 0, _omp_last_iter = 0;
    int _omp_schedule, _omp_chunksize = -1;

    _omp_schedule = _OMP_DYNAMIC;
    _omp_chunksize = 2;
    if (_omp_chunksize < 0)
      _omp_chunksize = _omp_get_default_chunksize (_omp_schedule, 100, 0, 3);
    _omp_incr = (3);
    _omp_init_directive (_OMP_FOR, 0, 0, _omp_incr, 0);
    _omp_incr *= _omp_chunksize;
```

```
  while (1) {
        /* get next job */
        _omp_lb = _omp_get_next_lb (_omp_schedule, 0, _omp_incr, &_omp_num_job);
        /* if none left, break */
        if (!(_omp_lb - _omp_incr < 100)) break;
        _omp_ub = _omp_lb + _omp_incr;
        if (100 <= _omp_ub) { /* check if out of bounds */
          _omp_ub = 100;
          _omp_last_iter = 1;
        }
        _omp_push_for_data (0, _omp_lb);

        for (i = _omp_lb; i < _omp_ub; i += 3)
              printf ("%d\n", i);     /* original code */

        _omp_pop_for_data ();
  }              /* while */
  if (_omp_last_iter) {   /* lastprivate assignments */
  }
  _omp_barrier_wait (&_OMP_THREAD->parent->barrier);
  _omp_flush_all ();      /* implied flush */
}
```

Καταρχήν, η μεταβλητή του `for` (`i`) ξαναδηλώνεται, αφού πρέπει να είναι τοπική για κάθε νήμα. Τα προβλήματα σχετικά με τις πολλαπλές εκτελέσεις της περιοχής αντιμετωπίζονται κι εδώ με τον ίδιο τρόπο (ουρές). Στη συνέχεια, μέσα στο `while`, κάθε νήμα, αφού πάρει δουλειά (με τη συνάρτηση `_omp_get_next_lb()`), υπολογίζει το κάτω και άνω όριο για τη μεταβλητή του βρόχου (`i`) και εκτελεί τον αυθεντικό κώδικα.

4.5.1 private, firstprivate, reduction

Η υλοποίηση των παραμέτρων αυτών έχει γίνει όπως και στο `parallel`.

4.5.2 lastprivate

Το νήμα που ανέλαβε την τελευταία επανάληψη του βρόχου, ενημερώνει τη lastprivate μεταβλητή.

4.5.3 nowait

Απλά δεν παράγεται κώδικας barrier.

4.5.4 schedule

Το default schedule είναι το *static*. Η συνάρτηση `_omp_get_next_lb()`, δέχεται ως όρισμα τον τύπο του schedule και επιστρέφει τις κατάλληλες εργασίες. Στην περίπτωση που το schedule είναι *dynamic*, στους κόμβους της ουράς αποθηκεύεται πληροφορία για την επόμενη δουλειά.

4.5.5 ordered

Όταν εμφανίζεται αυτή η παράμετρος, στους κόμβους της ουράς αποθηκεύεται επιπλέον πληροφορία για το πόσες επαναλήψεις έχουν εκτελεστεί με τη σειρά. Στην περίπτωση που δε δηλώνεται αυτή η παράμετρος, οι εντολές `ordered` αγνοούνται.

4.6 ordered

Η εντολή αυτή έχει νόημα μόνο όταν εκτελεστεί μέσα από ένα βρόχο for που έχει την παράμετρο ordered. Η υλοποίησή της έγινε με χρήση μεταβλητών συνθήκης της βιβλιοθήκης POSIX threads. Κάθε φορά που ένα νήμα εισέρχεται σε μία τέτοια περιοχή, εξετάζει αν η περιοχή αυτή έχει εκτελεστεί από όλες τις προηγούμενες (σειριακά) επαναλήψεις του for. Εάν δε συμβαίνει αυτό, το νήμα αναστέλλεται μέχρι να έρθει η σειρά του. Όταν ένα νήμα εξέρχεται από μία τέτοια περιοχή, αφυπνίζει τα νήματα που περιμένουν, τα οποία με τη σειρά τους εξετάζουν και πάλι αν μπορούν να εισέλθουν στην περιοχή.

Παράδειγμα:

```
#pragma omp ordered
        printf("%d\n", i);
```

Μετασχηματισμός:

```
/* #pragma omp ordered */
    _omp_ordered_begin ();
    printf ("%d\n", i);
    _omp_ordered_end ();
```

Η συνάρτηση _omp_ordered_begin() διαχειρίζεται τον έλεγχο εισόδου και την αναμονή. Η συνάρτηση _omp_ordered_end() διαχειρίζεται την αφύπνιση.

4.7 master

Ο κώδικας της εντολής master, για παράδειγμα:

```
#pragma omp master
    printf("I am the master thread!\n");
```

μετασχηματίζεται ως εξής:

```
/* #pragma omp master */
  if (_OMP_THREAD->thread_num == 0) {
    printf ("I am the master thread!\n");
  }
```

Δηλαδή εξετάζεται η ταυτότητα του νήματος από την δενδρική δομή, για να διαπιστώσουμε αν πρόκειται για το νήμα-αφέντη.

4.8 critical

Απαριθμούνται όλες οι κρίσιμες περιοχές με διαφορετικά ονόματα και δηλώνεται ένας πίνακας δυαδικών σημαφόρων με αντίστοιχο μέγεθος, έτσι ώστε σε κάθε όνομα να αντιστοιχεί ένας σημαφόρος. Έτσι, ο παρακάτω κώδικας

```
    int x, y;
#pragma omp critical (alpha)
    x++;
#pragma omp critical (beta)
    y++;
#pragma omp critical (alpha)
    x*=2;
```

μετασχηματίζεται σε

```
  int x, y;
/* #pragma omp critical (alpha) */
  pthread_mutex_lock (&_omp_critical_lock[0]);
  _omp_flush_all ();        /* implied flush */
  x++;
  _omp_flush_all ();        /* implied flush */
  pthread_mutex_unlock (&_omp_critical_lock[0]);
/* #pragma omp critical (beta) */
  pthread_mutex_lock (&_omp_critical_lock[1]);
  _omp_flush_all ();        /* implied flush */
  y++;
  _omp_flush_all ();        /* implied flush */
  pthread_mutex_unlock (&_omp_critical_lock[1]);
/* #pragma omp critical (alpha) */
  pthread_mutex_lock (&_omp_critical_lock[0]);
  _omp_flush_all ();        /* implied flush */
  x *= 2;
  _omp_flush_all ();        /* implied flush */
  pthread_mutex_unlock (&_omp_critical_lock[0]);
```

4.9 atomic

Η έκφραση που δηλώνεται στην εντολή `atomic` τοποθετείται ανάμεσα σε εντολές που κλειδώνουν και ξεκλειδώνουν έναν δυαδικό σημαφόρο οποίος είναι καθολική μεταβλητή. Ο παρακάτω κώδικας

```
    int x;
#pragma omp atomic
    x++;
```

μετασχηματίζεται σε

```
  int x;
/* #pragma omp atomic */
  pthread_mutex_lock (&_omp_atomic_lock);
  x++;
  pthread_mutex_unlock (&_omp_atomic_lock);
```

4.10 barrier

Στο σημείο που συναντάμε την εντολή αυτή, καλείται η ρουτίνα `_omp_barrier_wait()` η οποία αναγκάζει κάθε νήμα να περιμένει μέχρι όλα τα νήματα της ομάδας να εκτελέσουν τη ρουτίνα αυτή. Έτσι, ο κώδικας

```
#pragma omp barrier
    i++;
```

μετασχηματίζεται σε

```
  _omp_barrier_wait (&_OMP_THREAD->parent->barrier);
  _omp_flush_all ();        /* implied flush */
  i++;
```

Οι μεταβλητές (σημαφόρος και μεταβλητή συνθήκης) που χρειάζονται για να υλοποιηθεί το `barrier` αποθηκεύονται στον κόμβο του πατέρα, ώστε να είναι κοινές για όλα τα παιδιά.

4.11 flush

Όταν συναντάμε την εντολή `flush` με λίστα μεταβλητών, καλείται η ρουτίνα `_omp_flush()` για κάθε μία από τις μεταβλητές. Εαν δεν υπάρχει λίστα μεταβλητών, καλείται η ρουτίνα `_omp_flush_all()`. Ο παρακάτω κώδικας

```
#pragma omp flush(global_a)
#pragma omp flush
```

μετασχηματίζεται σε

```
/* OMP FLUSH BEGIN */
  _omp_flush ((void *) &global_a);
/* #pragma omp flush (global_a) */
/* OMP FLUSH END */
/* OMP FLUSH BEGIN */
  _omp_flush_all ();
/* #pragma omp flush */
/* OMP FLUSH END */
```

Ο κώδικας των ρουτινών που αναφέραμε, εξαρτάται από την αρχιτεκτονική του υπολογιστή στον οποίο τρέχει ο μεταφραστής, και συνήθως αποτελείται από εντολές σε assembly.

4.12 threadprivate

Σύμφωνα με το πρότυπο *OpenMP*, threadprivate μεταβλητές μπορεί να είναι μόνο οι καθολικές. Ο μεταφραστής μας παράγει μία δομή που περιέχει όλες τις μεταβλητές αυτές. Επίσης, δηλώνεται ένας καθολικός πίνακας τέτοιων δομών. Κάθε θέση του πίνακα αντιστοιχεί σε ένα νήμα. Κατ' επέκταση, κάθε νήμα έχει το δικό του αντίγραφο των threadprivate μεταβλητών. Κάθε αναφορά σε μια τέτοια μεταβλητή από ένα νήμα, μετασχηματίζεται σε αναφορά στο αντίστοιχο πεδίο της δομής. Η αρχικοποίηση του πίνακα δομών γίνεται με την εκκίνηση του προγράμματος.

Παράδειγμα:

```
int global_b, global_c;
#pragma omp threadprivate(global_b,global_c)

void threadprivate()
{
    global_b = 1;
    global_c = 2;
}
```

Η δομή που δημιουργείται είναι:

```
struct _omp_threadprivate_s {
    int global_b;
    int global_c;
};
```

Η αρχικοποίηση γίνεται στην αρχή της `main()` καλώντας την παρακάτω συνάρτηση:

```
/* global threadprivate initialization */
void _omp_init_global_threadprivate ()
{
  int i;
  for (i = 0; i < _omp_max_threads; i++) {
    _omp_tp_vars[i].global_b = global_b;
    _omp_tp_vars[i].global_c = global_c;
  }
}
```

Τέλος, οι αναφορές μετασχηματίζονται ως εξής:

```
void threadprivate ()
{
  _omp_tp_vars[_OMP_THREAD->real_thread_num].global_b = 1;
  _omp_tp_vars[_OMP_THREAD->real_thread_num].global_c = 2;
}
```

4.12.1 copyin

Αν και πρόκειται για παράμετρο του `parallel`, σχετίζεται άμεσα με το threadprivate. Αυτό που γίνεται είναι ότι πριν τη δημιουργία των νημάτων αντιγράφονται οι μεταβλητές που δηλώνει η παράμετρος από τη θέση 0 του πίνακα δομών σε όλες τις υπόλοιπες.

Παράδειγμα:

```
int global_b;
#pragma omp threadprivate(global_b)

void parallel_copyin()
{
#pragma omp parallel copyin(global_b)
    global_b++;
}
```

Θα δημιουργηθεί μια συνάρτηση που το όνομά της σχετίζεται με την παράμετρο αυτή:

```
void _omp_copyin_0()
{
  int i;
  for (i = 1; i < _omp_max_threads; i++) {
    _omp_tp_vars[i].global_b = _omp_tp_vars[0].global_b;
  }
}
```

Η συνάρτηση αυτή καλείται λίγο πριν τη δημιουργία νημάτων:

```
void parallel_copyin ()
{
  _omp_copyin_0 ();
  /* initialize copyin variables */
/* #pragma omp parallel  copyin(global_b) */
  {
    _OMP_PARALLEL_DECL_VARSTRUCT (parallel_copyin_parallel_0);
    _omp_create_team (_OMP_THREAD, parallel_copyin_parallel_0,
                      (void *) &parallel_copyin_parallel_0_var);
    _omp_destroy_team (_OMP_THREAD->parent);
  }
}
```

4.13 Αρχιτεκτονική παραγόμενου κώδικα

Έστω ένα πρόγραμμα γραμμένο σε *OpenMP* C, με όνομα `name.c`. Κατά τη μετάφραση αυτού του προγράμματος θα δημιουργηθεί ο κατάλογος `omp_name`. Στον κατάλογο αυτόν θα δημιουργηθούν τα αρχεία `name_omp.c`, `name_defs.h`, `name_typedefs.h`, `name_threadfuncs.c`, `name_threadfuncs.h`, `name_threadp.h`, `name_copyin.c`, `Makefile`. Το μεταφρασμένο πρόγραμμα συμπληρώνεται από τα αρχεία `_omp.c`, `_omp.h` και `_omp_global.h`.

4.13.1 *_omp.c

Το αρχείο αυτό περιέχει τον αρχικό κώδικα περασμένο απ' τον προεπεξεργαστή της C. Επιπλέον, έχουν εφαρμοστεί οι μετασχηματισμοί που αναφέραμε στις προηγούμενες παραγράφους. Δεν περιέχει τις συναρτήσεις νημάτων που δημιουργήθηκαν, καθώς και τις δηλώσεις των νέων δομών.

Όλα τα παραπάνω αρχεία (εκτός του `Makefile`) γίνονται `#include` εδώ.

Η `main()` έχει τροποποιηθεί, ώστε να καλεί την `_omp_initialize()` η οποία εκτελεί απαραίτητες αρχικοποιήσεις.

4.13.2 *_defs.c

Το αρχείο αυτό περιέχει τις δηλώσεις των δομών με τους δείκτες στις κοινές μεταβλητές.

4.13.3 *_typedefs.h

Όλα τα typedef του αρχικού προγράμματος έχουν μετακινηθεί εδώ. Όσα δεν ήταν δηλωμένα στο global scope έχουν αλλαγμένα ονόματα. Η μετακίνηση αυτή γίνεται για να αντιμετωπιστεί το εξής πρόβλημα. Επειδή μετακινείται κώδικας από κάποιο scope μέσα σε συνάρτηση (στο global scope), πρέπει τα typedef να εξακολουθούν να είναι ορατά.

4.13.4 *_threadfuncs.c

Εδώ έχουν παραχθεί οι συναρτήσεις των νημάτων που περιέχουν τον μετακινημένο κώδικα.

4.13.5 *_threadfuncs.h

Στο αρχείο αυτό τοποθετούνται τα πρωτότυπα των παραπάνω συναρτήσεων.

4.13.6 *_threadp.h

Περιέχει τη δομή με τις threadprivate μεταβλητές.

4.13.7 *_copyin.c

Εδώ παράγονται οι ρουτίνες που αρχικοποιούν τις threaprivate-copyin μεταβλητές.

4.13.8 *_omp.c

Στο αρχείο αυτό περιέχονται ρουτίνες που καλεί ο κώδικας που παράγουμε.

4.13.9 *_omp.h

Στο αρχείο αυτό βρίσκονται τα πρωτότυπα των παραπάνω ρουτινών. Επίσης περιέχονται οι δηλώσεις των δομών που χρησιμοποιούνται κατά την εκτέλεση (π.χ. δένδρο νημάτων), καθώς και κάποιες μακροεντολές (όπως το `_OMP_VARREF()`).

4.13.10 *_omp_global.h

Εδώ δηλώνονται οι μεταβλητές των δομών (π.χ, δενδρική δομή, πίνακες σημαφόρων κλπ.) που χρησιμοποιούνται από τις ρουτίνες στο `_omp.c`.

4.14 Υλοποίηση βιβλιοθήκης

Η βιβλιοθήκη αποτελείται από τα αρχεία `omp.h` και `omp.c`. Στο πρώτο περιέχονται τα πρωτότυπα των συναρτήσεων και οι δηλώσεις των δομών που περιγράφονται στο πρότυπο. Στο δεύτερο υπάρχει ο κώδικας των συναρτήσεων.

Κεφάλαιο 5

Επιδόσεις

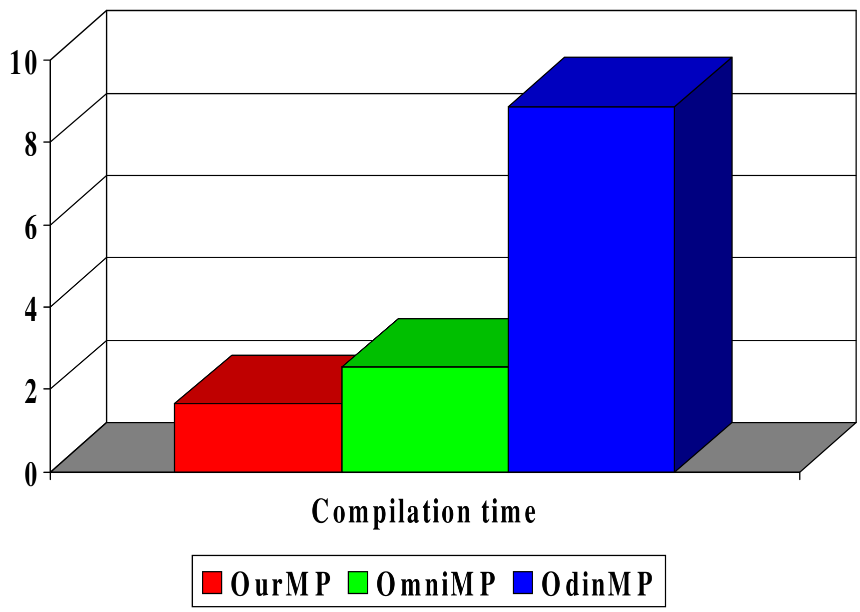
Στο κεφάλαιο αυτό παρουσιάζουμε κάποιες μετρήσεις που κάναμε για να διαπιστώσουμε την ταχύτητα μετάφρασης, αλλά *κυρίως* την ταχύτητα του παραγόμενου παράλληλου προγράμματος. Συγκεκριμένα μετράμε την επιτάχυνση, το χρόνο εκτέλεσης, αλλά και την επιβάρυνση (overhead) που προκύπτει από τον κώδικα που παράγουμε για να ερμηνεύσουμε τις εντολές *OpenMP*.

5.1 Χρόνος μετάφρασης

Στην εικόνα 5.1 φαίνεται ο χρόνος μετάφρασης για ένα απλό πρόγραμμα. Παρατηρούμε ότι επειδή ο δικός μας μεταφραστής είναι υλοποιημένος εξολοκλήρου σε C, είναι πολύ γρήγορος. Φαίνεται χαρακτηριστικά η διαφορά με το OdinMP το οποίο είναι γραμμένο σε Java. Το OmniMP είναι λίγο πιο αργό από το δικό μας μεταφραστή, επειδή λειτουργεί σε δύο φάσεις: αρχικά παράγει ενδιάμεσο κώδικα και ύστερα τον τελικό.

5.2 Τεστ

Για να μετρήσουμε την αποτελεσματικότητα του παραγόμενου κώδικα, όσον αφορά το χρόνο εκτέλεσης, χρησιμοποιήσαμε δύο προγράμματα: Το *pi* και το *molecular dynamics*. Τα προγράμματα αυτά τα πήραμε απ' το OdinMP.



Σχήμα 5.1:

5.2.1 Υπολογισμός του π

Το πρόγραμμα αυτό υπολογίζει το π χρησιμοποιώντας τον τύπο

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Το παραπάνω ολοκλήρωμα προσεγγίζεται με το άθροισμα

$$\sum_{i=0}^{N-1} \frac{4w}{1+[(i+0.5)w]^2}$$

Το άθροισμα αυτό υπολογίζεται με ένα βρόχο parallel for N βημάτων ($w = 1/N$ και όσο μεγαλύτερο το N τόσο μεγαλύτερη η ακρίβεια), κάνοντας υποβίβαση αθροίσματος. Ο κώδικας του προγράμματος βρίσκεται στο παράρτημα Δ΄.

5.2.2 Molecular Dynamics

Το πρόγραμμα αυτό υπολογίζει τις δυνάμεις που ασκούνται μεταξύ μορίων. Για περισσότερες πληροφορίες ανατρέξτε στο [12]. Και αυτό το πρόγραμμα χρησιμοποιεί παράλληλους βρόχους for. Ο κώδικας και αυτού του προγράμματος βρίσκεται στο παράρτημα Δ΄.

5.2.3 Περιβάλλον εκτέλεσης

Οι μετρήσεις έγιναν σε δύο διαφορετικές αρχιτεκτονικές. Ο παραγόμενος κώδικας μεταφράστηκε σε κώδικα μηχανής χωρίς τη χρήση optimization (–O, –O2, –O3).

- *SGI Origin 2000* με 16 CPU και 3.2GB μνήμη. Στον υπολογιστή αυτόν τρέξαμε το πρόγραμμα pi με $N = 100000000$ και το molecular dynamics με $nparts = 1024$.

- *SUN SparcServer* με 4 CPU και 512MB μνήμη. Εδώ τρέξαμε το πρόγραμμα pi με $N = 30000000$ και το molecular dynamics με $nparts = 512$.

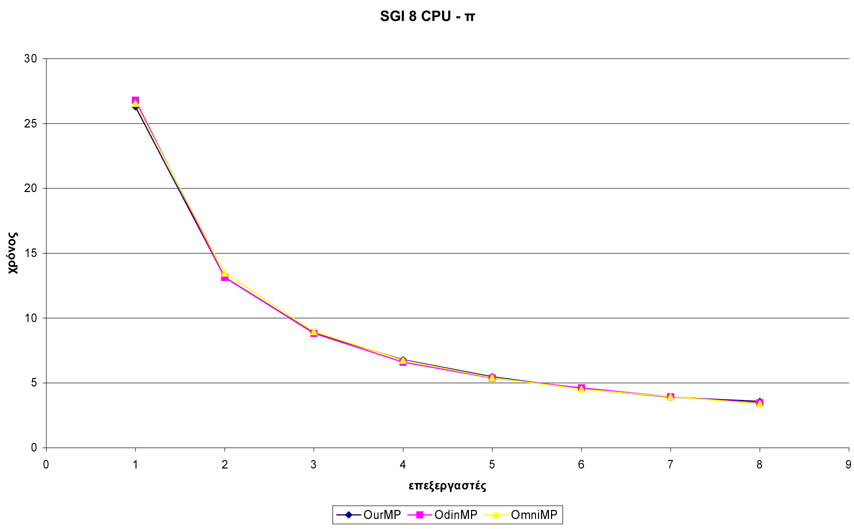
5.3 Χρόνος Εκτέλεσης

Όπως παρατηρούμε στα σχήματα 5.2 και 5.3, ο χρόνος εκτέλεσης είναι πολύ κοντά στον ιδανικό $1/N$ (όπου N ο αριθμός επεξεργαστών), επειδή το πρόγραμμα παραλληλοποιείται πλήρως. Παρατηρούμε ότι ο μεταφραστής μας παράγει αρκετά αποτελεσματικό κώδικα, σε σύγκριση με τις άλλες υλοποιήσεις.

Όπως παρατηρούμε στα σχήματα 5.4 και 5.5, ο δικός μας κώδικας είναι ελαφρά πιο αργός. Αυτό οφείλεται στο ότι στο δεύτερο πρόγραμμα (molecular dynamics) δημιουργούνται και καταστρέφονται νήματα. Οι υπόλοιπες υλοποιήσεις δεν καταστρέφουν τα νήματα, απλά τα απενεργοποιούν. Σε μελλοντική έκδοση σκοπεύουμε κι εμείς να κάνουμε τέτοιου είδους βελτιώσεις.

5.4 Επιτάχυνση (speedup)

Η επιτάχυνση ορίζεται ως ο λόγος του χρόνου εκτέλεσης με ένα νήμα προς το χρόνο εκτέλεσης με n νήματα. Προφανώς το ιδανικό είναι η επιτάχυνση να ισούται με τον αριθμό των επεξεργαστών. Όλες οι υλοποιήσεις, όπως φαίνεται στα σχήματα 5.6, 5.7, 5.8, 5.9 είναι πολύ



Σχήμα 5.2:

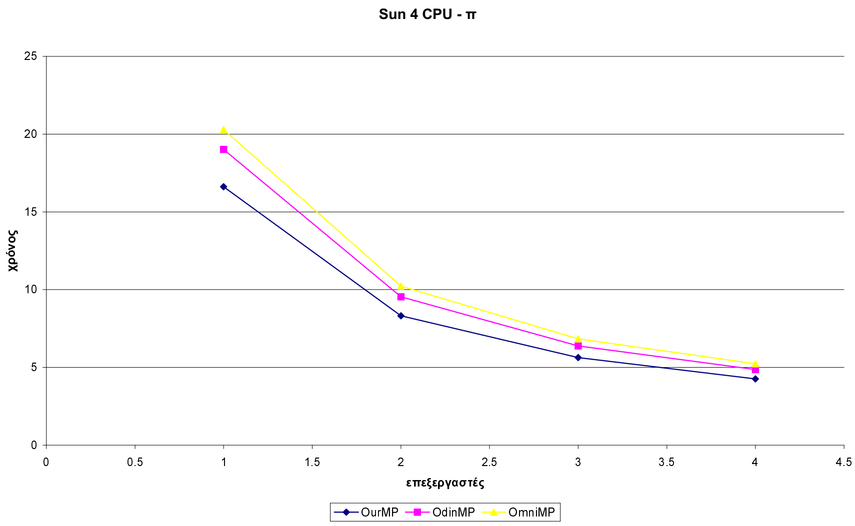
κοντά στα ιδανικά αποτελέσματα. Οι κυριότεροι λόγοι που στην πράξη δεν παίρνουμε ιδανικά αποτελέσματα είναι:

- Ποτέ δεν παραλληλοποιείται το 100% του προγράμματος. Πάντα υπάρχει ένα μέρος που εκτελείται από έναν μόνο επεξεργαστή.
- Υπάρχει ανισοκατανομή φόρτου ανάμεσα στους επεξεργαστές.

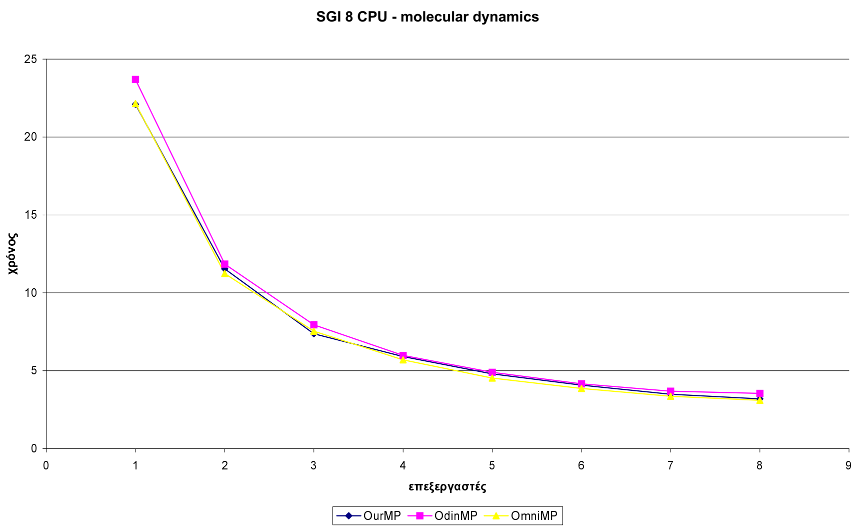
5.5 Επιβάρυνση (overhead)

Για να μετρήσουμε την επιβάρυνση που προκύπτει από ένα parallel for construct μετρήσαμε το χρόνο εκτέλεσης του παραγόμενου προγράμματος χρησιμοποιώντας μόνο έναν επεξεργαστή, αυξάνοντας τον αριθμό νημάτων. Στη συνέχεια, αφαιρέσαμε το χρόνο του σειριακού προγράμματος από τις προηγούμενες μετρήσεις.

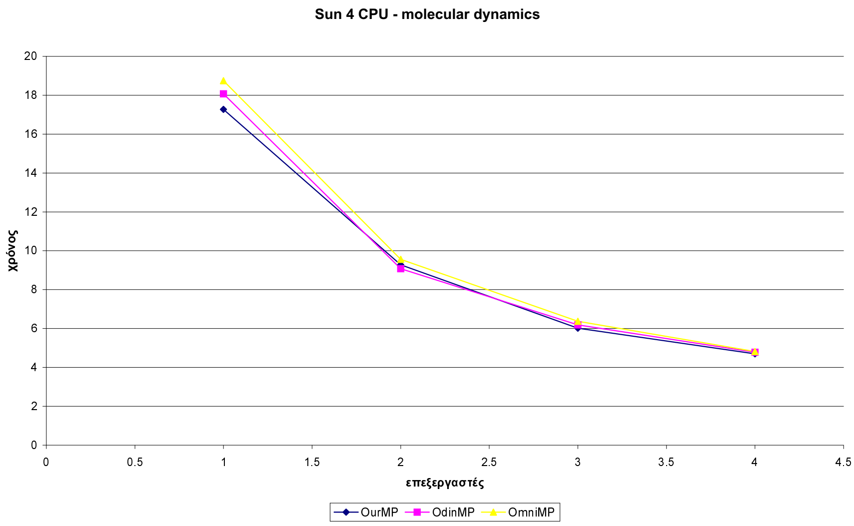
Όπως βλέπουμε στην εικόνα 5.10, τόσο η δική μας υλοποίηση, όσο και το OdinMP έχουν παρόμοια συμπεριφορά, μόνο που η δεύτερη έχει μία σταθερή επιπρόσθετη καθυστέρηση. Το OmniMP βλέπουμε ότι επιφέρει μεγάλη επιβάρυνση καθώς αυξάνεται ο αριθμός νημάτων. Μάλιστα, για μεγάλο αριθμό νημάτων το πρόγραμμα δεν έτρεχε!



Σχήμα 5.3:



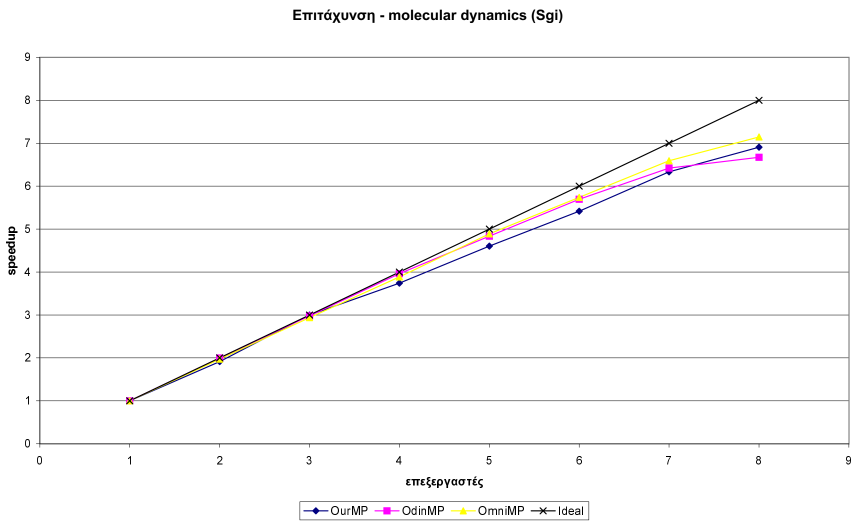
Σχήμα 5.4:



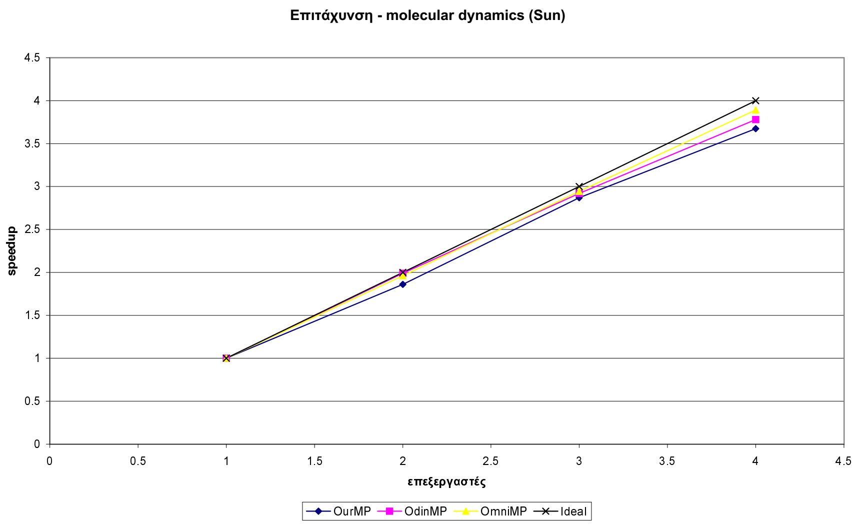
Σχήμα 5.5:

Σχήμα 5.6:

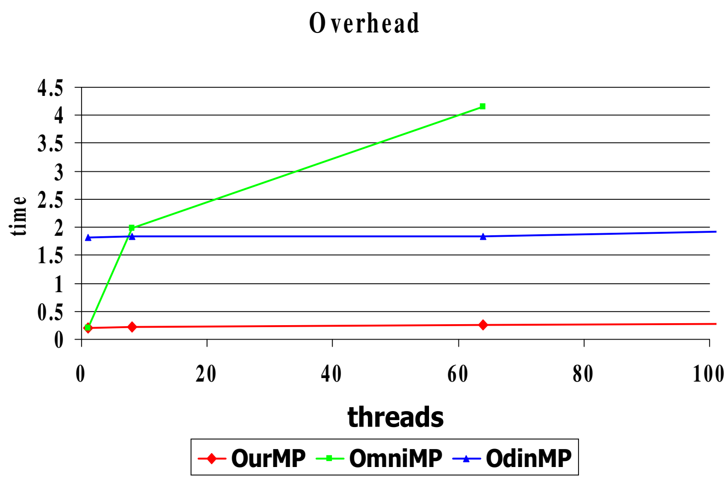
Σχήμα 5.7:



Σχήμα 5.8:



Σχήμα 5.9:



Σχήμα 5.10:

Κεφάλαιο 6

Συμπεράσματα

Ο στόχος της εργασίας αυτής ήταν η δημιουργία ενός μεταφραστή, ο οποίος δοθέντος ενός προγράμματος C με επεκτάσεις *OpenMP*, παράγει κώδικα ANSI C που χρησιμοποιεί τη βιβλιοθήκη POSIX threads για την επίτευξη του παραλληλισμού. Παράλληλα προσπαθήσαμε, ώστε τόσο ο κώδικας του μεταφραστή, όσο και ο παραγόμενος κώδικας να είναι μεταφέρσιμοι σε πολλές πλατφόρμες. Ένα άλλο κριτήριο που προσπαθήσαμε να ικανοποιήσουμε ήταν ο ικανοποιητικος χρόνος μετάφρασης κι εκτέλεσης του μεταγλωττισμένου αρχείου.

Το αποτέλεσμα ήταν η δημιουργία ενός μεταφραστή που πληρεί όλους τους παραπάνω στόχους. Ικανοποιεί πλήρως το πρότυπο *OpenMP*. Οι επιδόσεις του μεταφραστή μας, συναγωνίζονται και σε πολλές περιπτώσεις ξεπερνούν αυτές των άλλων ελεύθερων υλοποιήσεων, πλησιάζοντας αυτές των εμπορικών υλοποιήσεων.

Αν και ο μεταφραστής μας σχεδιάστηκε αρχικά ώστε να πληρεί την έκδοση 1.0 του προτύπου, τον Μάρτιο του 2002 βγήκε επίσημα η έκδοση 2.0 [4]. Έτσι, τροποποιήσαμε το σχεδιασμό μας, ώστε μελλοντικά να υποστηριχθεί η νέα έκδοση. Ήδη σε αυτή την υλοποίηση υποστηρίζονται ορισμένα από τα νέα χαρακτηριστικά. Συγκεκριμένα, υποστηρίζονται οι static threadprivate μεταβλητές και η επαναδήλωση μιας μεταβλητής ως private.

Τέλος, για να πάρετε μια ιδέα της έκτασης της εργασίας, αναφέρουμε ότι ο πηγαίος κώδικας που αναπτύχθηκε ξεπερνά τις 4200 γραμμές κώδικα ή αλλιώς τα 123kb. Αυτά χωρίς να συνυπολογίσουμε τα δεκάδες τεστ που γράψαμε για δοκιμές.

Μελλοντικές επεκτάσεις

Σε πρώτη φάση, σκεφτόμαστε να βελτιστοποιήσουμε την αποδοτικότητα του παραγόμενου κώδικα, έτσι ώστε να μειωθεί περαιτέρω ο χρόνος εκτέλεσης. Για παράδειγμα, ανάμεσα από δύο παράλληλες περιοχές, τα νήματα καταστρέφονται και ξαναδημιουργούνται. Αυτό είναι κάτι που προσθέτει αρκετές καθυστερήσεις και μπορεί να αποφευχθεί. Ένας τρόπος είναι να μην καταστρέφουμε τα νήματα, αλλά να τα διατηρούμε ανενεργά.

Επιπρόσθετα, πιστεύουμε ότι μπορεί σχετικά εύκολα να υλοποιηθεί ο εμφωλιασμένος παραλληλισμός μια και υπάρχει η βασική υποδομή στον κώδικά μας. Επίσης, μία πιθανή επέκταση είναι να υποστηριχθεί ο δυναμικός παραλληλισμός, δηλαδή κατά την εκτέλεση του προγράμματος να παίρνονται αυτόματα αποφάσεις σχετικά με τον αριθμό των νημάτων που θα δημιουργούνται σε κάθε παράλληλη περιοχή.

Τέλος, έχοντας υπόψιν την έκδοση 2.0 του προτύπου *OpenMP* η οποία ανακοινώθηκε πρόσφατα, και το ότι καμμία ελεύθερη υλοποίηση δεν την πληρεί για την ώρα, θα θέλαμε να είμαστε οι πρώτοι που θα το πετύχουμε!

Βιβλιογραφία

[1] Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language (Second Edition)*, Prentice Hall 1988

[2] Βασίλειος Β. Δημακόπουλος: *Παράλληλη Επεξεργασία*, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2001

[3] *OpenMP C and C++ API*, Version 1.0, October 1998, `http://www.openmp.org`

[4] *OpenMP C and C++ API*, Version 2.0, March 2002, `http://www.openmp.org`

[5] The Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard*, `http://www.mpi-forum.org`

[6] *LAM/MPI Parallel Computing*, `http://www.lam-mpi.org`

[7] *PVM: Parallel Virtual Machine*, `http://www.csm-ornl.gov/pvm/pvm_home.html`

[8] Christian Brunschen: *OdinMP/CCp — A portable Compiler for C with OpenMP to C with POSIX threads*, MSc. Thesis, Lund University, July 1999

[9] *OmniMP Compiler Project,* `http://phase.etl.go.jp/Omni/`

[10] Vern Paxson: *Flex, A fast scanner generator*, Edition 2.5, March 1995, `http://www.gnu.org`

[11] Charles Donelly, Richard Stallman: *Bison, The YACC-compatible Parser Generator*, November 1995, `http://www.gnu.org`

[12] Bill Magro: *A simple molecular dynamics simulation, using the velocity Verlet time integration scheme*, Kuck and Associates, Inc. (KAI), 1998

[13] *An Introduction to Parallel Computing on Clusters of Machines and on the SP*, http://www.apri.com/apr_forgex.html

[14] Pedro Diniz: *Commutativity Analysis*, http://www.isi.edu/~pedro/CA/commutativity.html

Παράρτημα Α΄

Λεκτική ανάλυση — **Flex**

Στους πίνακες που ακολουθούν, η πρώτη στήλη περιγράφει την κανονική έκφραση και οι υπόλοιπες τα επιστρεφόμενα αναγνωριστικά.

Α΄.1 Νέα token (***OpenMP***)

`parallel`	OMP_PARALLEL	TYPE_NAME	IDENTIFIER
`sections`	OMP_SECTIONS	TYPE_NAME	IDENTIFIER
`nowait`	OMP_NOWAIT	TYPE_NAME	IDENTIFIER
`ordered`	OMP_ORDERED	TYPE_NAME	IDENTIFIER
`schedule`	OMP_SCHEDULE	TYPE_NAME	IDENTIFIER
`dynamic`	OMP_DYNAMIC	TYPE_NAME	IDENTIFIER
`guided`	OMP_GUIDED	TYPE_NAME	IDENTIFIER
`runtime`	OMP_RUNTIME	TYPE_NAME	IDENTIFIER
`section`	OMP_SECTION	TYPE_NAME	IDENTIFIER
`single`	OMP_SINGLE	TYPE_NAME	IDENTIFIER
`master`	OMP_MASTER	TYPE_NAME	IDENTIFIER
`critical`	OMP_CRITICAL	TYPE_NAME	IDENTIFIER
`barrier`	OMP_BARRIER	TYPE_NAME	IDENTIFIER
`atomic`	OMP_ATOMIC	TYPE_NAME	IDENTIFIER
`flush`	OMP_FLUSH	TYPE_NAME	IDENTIFIER
`threadprivate`	OMP_THREADPRIVATE	TYPE_NAME	IDENTIFIER
`private`	OMP_PRIVATE	TYPE_NAME	IDENTIFIER
`firstprivate`	OMP_FIRSTPRIVATE	TYPE_NAME	IDENTIFIER
`lastprivate`	OMP_LASTPRIVATE	TYPE_NAME	IDENTIFIER

shared	OMP_SHARED	TYPE_NAME	IDENTIFIER
none	OMP_NONE	TYPE_NAME	IDENTIFIER
reduction	OMP_REDUCTION	TYPE_NAME	IDENTIFIER
copyin	OMP_COPYIN	TYPE_NAME	IDENTIFIER
^"#"[\t]*"pragma"			
[\t]+"omp"[\t]+	PRAGMA_OMP		

A′.2 Κοινά tokens (ANSI C και *OpenMP* C)

default	DEFAULT	OMP_DEFAULT
for	FOR	OMP_FOR
if	IF	OMP_IF
static	STATIC	OMP_STATIC
&&	AND_OP	OMP_LAND
||	OR_OP	OMP_LOR
&	'&'	OMP_BAND
-	'-'	OMP_MINUS
+	'+'	OMP_PLUS
*	'*'	OMP_MULT
^	'^'	OMP_XOR
|	'|'	OMP_BOR

A′.3 ANSI C-only tokens

auto	AUTO
break	BREAK
case	CASE
char	CHAR
const	CONST
continue	CONTINUE
do	DO
double	DOUBLE
else	ELSE
enum	ENUM
extern	EXTERN
float	FLOAT
goto	GOTO
int	INT

long	LONG
register	REGISTER
return	RETURN
short	SHORT
signed	SIGNED
sizeof	SIZEOF
struct	STRUCT
switch	SWITCH
typedef	TYPEDEF
union	UNION
unsigned	UNSIGNED
void	VOID
volatile	VOLATILE
while	WHILE
>>=	RIGHT_ASSIGN
<<=	LEFT_ASSIGN
+=	ADD_ASSIGN
-=	SUB_ASSIGN
*=	MUL_ASSIGN
/=	DIV_ASSIGN
%=	MOD_ASSIGN
&=	AND_ASSIGN
^=	XOR_ASSIGN
|=	OR_ASSIGN
>>	RIGHT_OP
<<	LEFT_OP
++	INC_OP
--	DEC_OP
->	PTR_OP
<=	LE_OP
>=	GE_OP
==	EQ_OP
!=	NE_OP
=	'='
!	'!'
~	'~'

/	'/'
%	'%'
<	'<'
>	'>'
?	'?'
;	';'
{	'{'
}	'}'
,	','
:	':'
(	'('
)	')'
[	'['
]	']'
.	'.'
...	ELIPSIS

Α′.3.1 Σταθερές

Έχουμε ορίσει κάποιες ομάδες χαρακτήρων, ώστε να απλοποιηθούν ορισμένες κανονικές εκφράσεις.

```
D    [0-9]
L    [a-zA-Z\_]
H    [a-fA-F0-9]
E    [Ee][+-]?{D}+
FS   (f|F|l|L)
IS   (u|U|l|L)*
```

Όλες οι παρακάτω κανονικές εκφράσεις επιστρέφουν CONSTANT ως αναγνωριστικό.

```
{L}({L}|{D})*

0[xX]{H}+{IS}?
0{D}+{IS}?
{D}+{IS}?
'(\\.|[^\\'])+'

{D}+{E}{FS}?
{D}*"."{D}+({E})?{FS}?
{D}+"."{D}*({E})?{FS}?
```

Παράρτημα Β΄

Γραμματική *OpenMP* C

Στο παράρτημα αυτό παρουσιάζουμε τη γραμματική της *OpenMP* C σε μια μορφή παρόμοια της BNF, όπως απαιτεί ο bison.

```
string_literal_seq:
          STRING_LITERAL
        | STRING_LITERAL string_literal_seq
        ;


primary_expr:
          identifier
        | CONSTANT
        | string_literal_seq
        | '(' expr ')'
        ;


postfix_expr:
          primary_expr
        | postfix_expr '[' expr ']'
        | postfix_expr '(' ')'
        | postfix_expr '(' argument_expr_list ')'
        | postfix_expr '.' identifier
        | postfix_expr PTR_OP identifier
        | postfix_expr INC_OP
        | postfix_expr DEC_OP
        ;

argument_expr_list:
          assignment_expr
        | argument_expr_list ',' assignment_expr
        ;
```

```
unary_expr:
          postfix_expr
        | INC_OP unary_expr
        | DEC_OP unary_expr
        | unary_operator cast_expr
        | SIZEOF unary_expr
        | SIZEOF '(' type_name ')'
        ;

unary_operator:
          '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expr:
          unary_expr
        | '(' type_name ')' cast_expr
        ;

multiplicative_expr:
          cast_expr
        | multiplicative_expr '*' cast_expr
        | multiplicative_expr '/' cast_expr
        | multiplicative_expr '%' cast_expr
        ;

additive_expr:
          multiplicative_expr
        | additive_expr '+' multiplicative_expr
        | additive_expr '-' multiplicative_expr
        ;

shift_expr:
          additive_expr
        | shift_expr LEFT_OP additive_expr
        | shift_expr RIGHT_OP additive_expr
        ;

relational_expr:
          shift_expr
        | relational_expr '<' shift_expr
        | relational_expr '>' shift_expr
        | relational_expr LE_OP shift_expr
        | relational_expr GE_OP shift_expr
        ;
```

```
equality_expr:
          relational_expr
        | equality_expr EQ_OP relational_expr
        | equality_expr NE_OP relational_expr
        ;

and_expr:
          equality_expr
        | and_expr '&' equality_expr
        ;

exclusive_or_expr:
          and_expr
        | exclusive_or_expr '^' and_expr
        ;

inclusive_or_expr:
          exclusive_or_expr
        | inclusive_or_expr '|' exclusive_or_expr
        ;

logical_and_expr:
          inclusive_or_expr
        | logical_and_expr AND_OP inclusive_or_expr
        ;

logical_or_expr:
          logical_and_expr
        | logical_or_expr OR_OP logical_and_expr
        ;

conditional_expr:
          logical_or_expr
        | logical_or_expr '?' logical_or_expr ':' conditional_expr
        ;

assignment_expr:
          conditional_expr
        | unary_expr assignment_operator assignment_expr
        ;

assignment_operator:
          '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
```

```
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expr:
          assignment_expr
        | expr ',' assignment_expr
        ;

constant_expr:
          conditional_expr
        ;

declaration:
          declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        | threadprivate_directive
        ;

declaration_specifiers:
          storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        ;

init_declarator_list:
          init_declarator
        | init_declarator_list ','  init_declarator
        ;

init_declarator:
          declarator
        | declarator '='  initializer
        ;

storage_class_specifier:
          TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

type_specifier:
          CHAR
        | SHORT
        | INT
        | LONG
        | SIGNED
        | UNSIGNED
```

```
        | FLOAT
        | DOUBLE
        | CONST
        | VOLATILE
        | VOID
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;

struct_or_union_specifier:
        struct_or_union identifier '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union identifier
        ;

struct_or_union:
          STRUCT
        | UNION
        ;

struct_declaration_list:
          struct_declaration
        | struct_declaration_list struct_declaration
        ;

struct_declaration:
          type_specifier_list struct_declarator_list ';'
        ;

struct_declarator_list:
          struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;

struct_declarator:
          declarator
        | ':' constant_expr
        | declarator ':' constant_expr
        ;

enum_specifier:
          ENUM '{' enumerator_list '}'
        | ENUM identifier '{' enumerator_list '}'
        | ENUM identifier
        ;

enumerator_list:
          enumerator
        | enumerator_list ',' enumerator
        ;
```

```
enumerator:
          identifier
        | identifier '=' constant_expr
        ;

declarator:
          declarator2
        | pointer declarator2
        ;

declarator2:
          identifier
        | '(' declarator ')'
        | declarator2 '[' ']'
        | declarator2 '[' constant_expr ']'
        | declarator2 '(' ')'
        | declarator2 '(' parameter_type_list ')'
        | declarator2 '(' parameter_identifier_list ')'
        ;

pointer:
          '*'
        | '*' type_specifier_list
        | '*' pointer
        | '*' type_specifier_list pointer
        ;

type_specifier_list:
          type_specifier
        | type_specifier_list type_specifier
        ;

parameter_identifier_list:
          identifier_list
        | identifier_list ',' ELIPSIS
        ;

identifier_list:
          identifier
        | identifier_list ',' identifier
        ;

parameter_type_list:
          parameter_list
        | parameter_list ',' ELIPSIS
        ;

parameter_list:
          parameter_declaration
        | parameter_list ',' parameter_declaration
```

```
        ;

parameter_declaration:
          type_specifier_list declarator
        | type_name
        ;

type_name:
          type_specifier_list
        | type_specifier_list abstract_declarator
        ;

abstract_declarator:
          pointer
        | abstract_declarator2
        | pointer abstract_declarator2
        ;

abstract_declarator2:
          '(' abstract_declarator ')'
        | '[' ']'
        | '[' constant_expr ']'
        | abstract_declarator2 '[' ']'
        | abstract_declarator2 '[' constant_expr ']'
        | '(' ')'
        | '(' parameter_type_list ')'
        | abstract_declarator2 '(' ')'
        | abstract_declarator2 '(' parameter_type_list ')'
        ;

initializer:
          assignment_expr
        | '{' open_brace initializer_list '}'
        | '{' open_brace initializer_list ',' '}'
        ;

initializer_list:
          initializer
        | initializer_list ','  initializer
        ;

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

```
openmp_construct:
          parallel_construct
        | for_construct
        | sections_construct
        | single_construct
        | parallel_for_construct
        | parallel_sections_construct
        | master_construct
        | critical_construct
        | atomic_construct
        | ordered_construct
        ;

openmp_directive:
          barrier_directive
        | flush_directive
        ;

structured_block:
          statement
        ;

parallel_construct:
          parallel_directive structured_block
        ;

parallel_clause_optseq:
          /* empty */
        | parallel_clause_optseq parallel_clause
        ;

parallel_directive:
          PRAGMA_OMP OMP_PARALLEL parallel_clause_optseq '\n'
        ;

parallel_clause:
          unique_parallel_clause
        | data_clause
        ;

unique_parallel_clause:
          OMP_IF '(' expr ')'
        ;

for_construct:
          for_directive iteration_statement
        ;

for_clause_optseq:
          /* empty */
        | for_clause_optseq for_clause
```

```
        ;

for_directive:
          PRAGMA_OMP OMP_FOR for_clause_optseq '\n'
        ;

for_clause:
          unique_for_clause
        | data_clause
        | OMP_NOWAIT
        ;

unique_for_clause:
          OMP_ORDERED
        | OMP_SCHEDULE '(' schedule_kind ')'
        | OMP_SCHEDULE '(' schedule_kind ',' expr ')'
        ;

schedule_kind:
          OMP_STATIC
        | OMP_DYNAMIC
        | OMP_GUIDED
        | OMP_RUNTIME
        ;

sections_construct:
          sections_directive section_scope
        ;

sections_clause_optseq:
          /* empty */
        | sections_clause_optseq sections_clause
        ;

sections_directive:
          PRAGMA_OMP OMP_SECTIONS sections_clause_optseq '\n'
        ;

sections_clause:
          data_clause
        | OMP_NOWAIT
        ;

section_scope:
          '{'
          section_sequence
          '}'
        ;

section_sequence:
          structured_block
```

```
        | section_directive structured_block
        | section_sequence section_directive structured_block
        ;

section_directive:
          PRAGMA_OMP OMP_SECTION '\n'
        ;


single_construct:
          single_directive structured_block
        ;

single_clause_optseq:
          /* empty */
        | single_clause_optseq single_clause
        ;

single_directive:
          PRAGMA_OMP OMP_SINGLE single_clause_optseq '\n'
        ;

single_clause:
          data_clause
        | OMP_NOWAIT
        ;

parallel_for_construct:
          parallel_for_directive iteration_statement
        ;

parallel_for_clause_optseq:
          /* empty */
        | parallel_for_clause_optseq parallel_for_clause
        ;

parallel_for_directive:
          PRAGMA_OMP OMP_PARALLEL OMP_FOR parallel_for_clause_optseq '\n'
        ;

parallel_for_clause:
          unique_parallel_clause
        | unique_for_clause
        | data_clause
        ;

parallel_sections_construct:
          parallel_sections_directive section_scope
        ;

parallel_sections_clause_optseq:
```

```
          /* empty */
        | parallel_sections_clause_optseq parallel_sections_clause
        ;

parallel_sections_directive:
          PRAGMA_OMP OMP_PARALLEL OMP_SECTIONS parallel_sections_clause_optseq '\n'
        ;

parallel_sections_clause:
          unique_parallel_clause
        | data_clause
        ;

master_construct:
          master_directive structured_block
        ;

master_directive:
          PRAGMA_OMP OMP_MASTER '\n'
        ;

critical_construct:
          critical_directive structured_block
        ;

critical_directive:
          PRAGMA_OMP OMP_CRITICAL region_phrase_opt '\n'
        ;

region_phrase_opt:
          /* empty */
        | '(' identifier ')'
        ;

barrier_directive:
          PRAGMA_OMP OMP_BARRIER '\n'
        ;

atomic_construct:
          atomic_directive expression_statement
        ;

atomic_directive:
          PRAGMA_OMP OMP_ATOMIC '\n'
        ;

flush_directive:
          PRAGMA_OMP OMP_FLUSH flush_vars_opt '\n'
        ;

flush_vars_opt:
```

```
          /* empty */
        | '(' variable_list ')'
        ;

ordered_construct:
          ordered_directive structured_block
        ;

ordered_directive:
          PRAGMA_OMP OMP_ORDERED '\n'
        ;

threadprivate_directive:
          PRAGMA_OMP OMP_THREADPRIVATE '(' variable_list ')' '\n'
        ;

data_clause:
          OMP_PRIVATE '(' variable_list ')'
        | OMP_FIRSTPRIVATE '(' variable_list ')'
        | OMP_LASTPRIVATE '(' variable_list ')'
        | OMP_SHARED '(' variable_list ')'
        | OMP_DEFAULT '(' OMP_SHARED ')'
        | OMP_DEFAULT '(' OMP_NONE ')'
        | OMP_REDUCTION '(' reduction_operator ':' variable_list ')'
        | OMP_COPYIN '(' variable_list ')'
        ;

reduction_operator:
          OMP_PLUS
        | OMP_MULT
        | OMP_MINUS
        | OMP_BAND
        | OMP_XOR
        | OMP_BOR
        | OMP_LAND
        | OMP_LOR
        ;

variable_list:
          identifier
        | variable_list ',' identifier
        ;

labeled_statement:
          identifier ':'
          statement
        | CASE constant_expr ':'
          statement
        | DEFAULT ':'
          statement
        ;
```

```
open_brace:
          ;

compound_statement:
          '{'  '}'
        | '{'
          statement_list '}'
        | '{' open_brace declaration_list '}'
        | '{' open_brace declaration_list
          statement_list '}'
        ;

declaration_list:
          declaration
        | declaration_list declaration
        ;

statement_list:
          statement
        | openmp_directive
        | statement_list statement
        | statement_list openmp_directive
        ;

expression_statement:
          ';'
        | expr ';'
        ;

else_statement:
          /* empty */
        | ELSE
          statement
        ;

selection_statement:
          IF '(' expr ')'
          statement else_statement
        | SWITCH '(' expr ')'
          statement
        ;

iteration_statement:
          WHILE '(' expr ')'  statement
        | DO  statement WHILE '(' expr ')' ';'
        | FOR '(' ';' ';' ')'  statement
        | FOR '(' ';' ';' expr ')'  statement
        | FOR '(' ';' expr ';' ')'  statement
        | FOR '(' ';' expr ';' expr ')'  statement
        | FOR '(' expr ';' ';' ')'  statement
```

```
        | FOR '(' expr ';' ';' expr ')'  statement
        | FOR '(' expr ';' expr ';' ')'  statement
        | FOR '(' expr ';' expr ';' expr ')'  statement
        ;

jump_statement:
          GOTO identifier ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expr ';'
        ;

file:
          external_definition
        | file external_definition
        ;

external_definition:
          function_definition
        | declaration
        ;

function_definition:
          declarator  function_body
        | declaration_specifiers declarator  function_body
        ;

function_body:
          compound_statement
        | declaration_list  compound_statement
        ;

identifier:
          IDENTIFIER
        ;
```

Παράρτημα Γ′

Πηγαίος κώδικας

Εδώ παραθέτουμε τον πηγαίο κώδικα της υλοποίησής μας. Συγκεκριμένα παρατίθεται ο κώδικας των αρχείων:

scanner.l Λεκτικός αναλυτής (flex)

parser.y Συντακτικός αναλυτής (bison) — Παραγωγή τελικού κώδικα

_omp.h, _omp.c, _omp_global.c Επιπρόσθετος τελικός κώδικας (ανεξάρτητος της εισόδου)

omp.h, omp.c Runtime βιβλιοθήκη

Παράρτημα Δ΄

Κώδικας των τεστ (pi, molecular dynamics)

Παράρτημα Ε΄

Οδηγίες

Ε΄.1 Οδηγίες εγκατάστασης

Το πρόγραμμα είναι διαθέσιμο σε μορφή `ompc-X.Y.Z.tar.gz`, όπου X.Y.Z. ο αριθμός έκδοσης. Αφού αποσυμπιεστεί (με τις εντολές `gunzip` και `tar xvf`) δημιουργείται ο κατάλογος `ompc-X.Y.Z` που περιέχει τον πηγαίο κώδικα του μεταφραστή.

Για τη μετάφραση του μεταφραστή αρκεί να τρέξουμε την εντολή `make`. Μετά τη μετάφραση έχει παραχθεί το εκτελέσιμο του μεταφραστή, καθώς και η runtime βιβλιοθήκη.

Ε΄.2 Περιγραφές αρχείων

scanner.l Λεκτικός αναλυτής. Πρόκειται για την είσοδο του flex.

parser.y Συντακτικός αναλυτής. Το μεγαλύτερο μέρος της υλοποίησης βρίσκεται εδώ. Το αρχείο αυτό αποτελεί το αρχείο εισόδου του bison. Ο κώδικας αυτού του αρχείου κάνει συντακτική ανάλυση, έλεγχο λαθών και επιτελεί μετασχηματισμούς.

lib/_omp.h, lib/_omp.c, lib/_omp_global.c Στον παραγόμενο κώδικα χρησιμοποιούνται κάποιες δομές και συναρτήσεις. Αυτές βρίσκονται σε αυτά τα αρχεία τα οποία γίνονται `#inlcude` από τον παραγόμενο κώδικα.

lib/omp.h, lib/omp.c Runtime βιβλιοθήκη

ompc, ompcc Το `ompc` είναι το αποτέλεσμα της μετάφρασης του μεταφραστή. Καλείται εσωτερικά από το script `ompcc`.

tests/ Εδώ υπάρχει ένα σύνολο από προγράμματα *OpenMP*, για εξέταση της ορθής λειτουργίας του μεταφραστή.

Ε΄.3 Οδηγίες compilation (εκτέλεσης)

Καταρχήν, στο πρόγραμμα του χρήστη πρέπει να γίνεται `#include` είτε το `omp.h` είτε μαζί τα `pthread.h`, `stdlib.h`, `stdio.h`. Έστω ότι το *OpenMP* πρόγραμμά μας βρίσκεται στο αρχείο `name.c`. Για τη μετάφρασή του ακολουθούμε τα παρακάτω:

1. Εκτελούμε `ompcc name.c`. Θα δημιουργηθεί ο κατάλογος `omp_name`, που περιέχει τον μετασχηματισμένο κώδικα.

2. Μπαίνουμε στον παραπάνω κατάλογο κι εκτελούμε το `make`. Αυτό θα δημιουργήσει το τελικό εκτελέσιμο.