Υλοποίηση cancellation στον παραλληλοποιητικό μεταφραστή OMPi

από τον

Εμμανουήλ Φελουτζή

με Α.Μ: 1579

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΠΛΩΜΑΤΟΣ

ΤΟΥ ΤΜΗΜΑΤΟΣ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΑΡΤΙΟΣ 2015

Επιβλέπων: Β.Δημακόπουλος

Περιεχόμενα

1 Εισαγωγή **7**
- 1.1 Υπολογιστές και Εξέλιξη . . . 7
- 1.2 Παράλληλοι Υπολογιστές . . . 7
  - 1.2.1 Οργάνωση Κοινής Μνήμης (Shared Memory) . . . 7
  - 1.2.2 Σύστημα Κατανεμημένης Μνήμης (Distributed Memory) . . . 9
- 1.3 Προγραμματισμός Παράλληλων Υπολογιστών . . . 9
  - 1.3.1 Προγραμματισμός με μοντέλο κοινού χώρου διευθύνσεων . . . 10
  - 1.3.2 Προγραμματισμός σε μοντέλο μεταβίβασης μηνυμάτων . . . 10
- 1.4 Αντικείμενο Εργασίας . . . 10
- 1.5 Δομή της Εργασίας . . . 11

2 Το πρότυπο OpenMP **13**
- 2.1 Εισαγωγή . . . 13
- 2.2 Προγραμματιστικό μοντέλο OpenMP . . . 14
- 2.3 Οδηγίες OpenMP για C\C++ . . . 14
  - 2.3.1 Σύνταξη Οδηγιών OpenMP . . . 14
  - 2.3.2 Παράλληλες Περιοχές . . . 15
  - 2.3.3 Περιοχές Διαμοιρασμού Εργασίας (Workshare Regions) . . . 16
  - 2.3.4 Φράσεις Οδηγιών του OpenMP . . . 16
  - 2.3.5 Οδηγίες Συγχρονισμού για C\C++ . . . 18
- 2.4 Κλήσεις Βιβλιοθήκης Runtime . . . 18
- 2.5 Μεταβλητές Περιβάλλοντος . . . 20
- 2.6 Νέες λειτουργίες του OpenMP v.4.0 . . . 20

3 Ο μεταφραστής OMPi **23**
- 3.1 Δομή του OMPi . . . 23
- 3.2 Ο μετασχηματιστής πηγαίου σε πηγαίο κώδικα (S2S) . . . 23
- 3.3 Το σύστημα Runtime του OMPi . . . 25
  - 3.3.1 Διεπαφή με το EELIB . . . 25
  - 3.3.2 Είσοδος σε παράλληλη περιοχή . . . 27

4 Ακύρωση (Cancellation) στο OpenMP v.4.0 **29**
- 4.1 Cancellation σε ομάδες νημάτων . . . 29
  - 4.1.1 Σύνταξη . . . 29
  - 4.1.2 Περιγραφή . . . 30
  - 4.1.3 Περιορισμοί . . . 31
- 4.2 Οδηγία σημείου ακύρωσης (Cancellation Point) . . . 31
  - 4.2.1 Εισαγωγή . . . 31
  - 4.2.2 Σύνταξη . . . 33
  - 4.2.3 Περιγραφή . . . 33

4.2.4 Περιορισμοί 33
4.3 Cancellation σε εργασίες 34
4.3.1 Εργασίες-Tasks 34
4.3.2 Taskgroup 35

5 Υλοποίηση στον μεταφραστή OMPi **37**
5.1 Κεντρική Ιδέα για την Υλοποίηση του CANCEL 37
5.2 Cancellation σε parallel construct 39
5.3 Cancellation σε sections construct 40
5.4 Cancellation σε for construct 42
5.5 Cancellation σε taskgroup construct 48
5.6 Υλοποίηση των Cancellation Point στον OMPi 49

6 Πειράματα **57**
6.1 Πειράματα ορθότητας 57
6.2 EPCC Microbenchmarks για την υποστήριξη cancellation στον OMPi 58
6.2.1 Syncbench microbencmark 58
6.2.2 Taskbench microbencmark 60
6.3 Εφαρμογή για την χρήση του cancel 61
6.3.1 Σύγκριση του μεταφραστή OMPi με άλλους compilers για την συγκεκριμένη εφαρμογή 63

7 Συμπεράσματα και μελλοντική δουλειά **67**
7.1 Σύνοψη διπλωματικής εργασίας και εφαρμογές 67
7.2 Μελλοντική εργασία στον OMPi 68

Κατάλογος σχημάτων

1.1 Μοντέλο Κοινής Μνήμης 8
1.2 Σύστημα Κατανεμημένης Μνήμης 9

2.1 Ενα απλό παράδειγμα παράλληλης περιοχής. 15

3.1 Στάδια λειτουργίας του μεταφραστή OMPi 24
3.2 Ο παραγόμενος κώδικας του μεταφραστή OMPi για τον παράδειγμα. 24
3.3 Διεπαφή μεταξύ του τμήματος ORT και του τμήματος EELIB 26

4.1 Innermost enclosing region 30
4.2 Παράδειγμα για την ερμηνεία του CANCEL 32
4.3 Παράδειγμα για την λειτουργία του CANCEL σε taskgroup 36

5.1 Περιγραφή του cancel 38
5.2 Απλό παράδειγμα εφαρμογής του cancel σε μία οδηγία parallel. 39
5.3 Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση του cancel σε parallel 40
5.4 Παραγόμενος κώδικας του OMPi, με την υλοποίηση του cancel σε parallel . . 41
5.5 Ο επαναληπτικός μηχανισμός ανάθεσης των section στα νήματα στη βιβλιοθήκη runtime του OMPi με την προσθήκη για έλεγχο cancellation. 42
5.6 Παράδειγμα χρήσης του cancel σε sections 43
5.7 Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση του cancel σε sections 44
5.8 Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο sections 45
5.9 Εφαρμογή του cancellation σε for 46
5.10 Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση για το cancel στο for 47
5.11 Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο for . . 47
5.12 Επαναληπτικός μηχανισμός ανάθεσης των επαναλήψεων στα νήματα σε dynamic schedule 50
5.13 Εφαρμογή του Cancellation σε for 50
5.14 Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση για το cancel στο for με schedule dynamic 51
5.15 Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο for με schedule dynamic 51
5.16 Επαναληπτικός μηχανισμός ανάθεσης των επαναλήψεων στα νήματα guided Schedule 52
5.17 Η δομή ενός taskgroup. 53
5.18 Ένα απλό παράδειγμα για την εφαρμογή του cancel taskgroup 53
5.19 Ο παραγόμενος κώδικας του Κώδικα 5.18 55

6.1 Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον icc. Η εκτέλεση έγινε στο σύστημα PARAGUAY. 59

6.2 Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον gcc. Η εκτέλεση έγινε στο σύστημα PARAGON. 60
6.3 Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον icc. Η εκτέλεση έγινε στο σύστημα PARAGUAY. 61
6.4 Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον gcc. Η εκτέλεση έγινε στο σύστημα PARAGON. 61
6.5 Παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο. 62
6.6 Αποτελέσματα εκτέλεσης της εφαρμογής 63
6.7 Παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο με την υλοποίηση Cancel Taskgroup 64
6.8 Αποτελέσματα εκτέλεσης της εφαρμογής με την χρήση του Cancel Taskgroup στο σύστημα PARAGON 65

Κεφάλαιο 1

Εισαγωγή

1.1 Υπολογιστές και Εξέλιξη

Από την αρχαιότητα ακόμη, ο άνθρωπος αναζητούσε τρόπους ώστε να βελτιώσει και να απλουστεύσει την καθημερινότητα και την ζωή του. Όσο οι απαιτήσεις της καθημερινότητας αυτής αυξάνονταν, τόσο πιο έντονη ήταν η ανάγκη για την δημιουργία μηχανών υπολογισμού. Γύρω στο 17ο αιώνα, αρχίζει σταδιακά η δημιουργία τέτοιων μηχανών, γεγονός που σηματοδότησε την αρχή μιας νέας εποχής στο χώρο της επιστήμης.

Η πρώτη καταγεγραμένη εμφάνιση υπολογιστή εντοπίζεται στην δεκαετία του '40. Συγκεκριμένα, το 1946 κάνει την εμφάνιση του ο πρώτος υπολογιστής με το όνομα ENIAC. Ο ENIAC καταλάμβανε γύρω στα 63 τ.μ. χώρο, και μπορούσε να εκτελέσει 5000 προσθέσεις το δευτερόλεπτο. Ένας σημερινός υπολογιστής, με πολύ μικρότερο όγκο, μπορεί να εκτελέσει στον ίδιο χρόνο δισεκατομμύρια τέτοιες πράξεις.

Ο άνθρωπος, θέλοντας να κερδίσει όσο το δυνατόν περισσότερο από τον υπολογιστή, και να κάνει γρηγορότερους και περισσότερους υπολογισμούς, δημιουργούσε όλο και πιο γρήγορους επεξεργαστές. Όμως η εξέλιξη αυτή δεν ήταν πλέον αρκετή, διότι περιοριζόταν από τους νόμους της φύσης και τα όρια του υλικού. Επομένως, έπρεπε να βρεθούν νέες ιδέες και καινοτομίες ώστε να επιλυθούν αυτά τα προβλήματα, και η λύση αυτή δόθηκε από τα παράλληλα συστήματα. Από εκείνη την στιγμή, εμφανίστηκε στον κλάδο της Πληροφορικής το παράλληλο μοντέλο υπολογισμού και η παράλληλη επεξεργασία.

1.2 Παράλληλοι Υπολογιστές

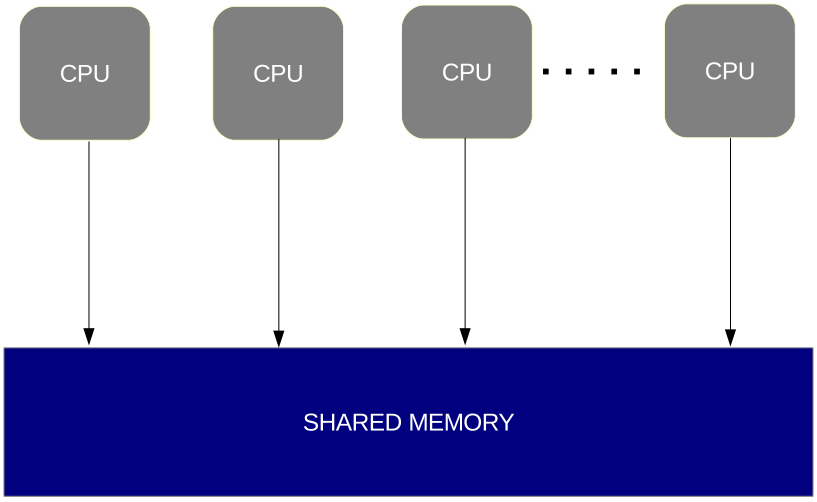
Παράλληλα συστήματα, είναι τα σύστημα με πολλούς επεξεργαστές(CPU's) οι οποίοι είναι σε θέση να συνεργάζονται, και να επικοινωνούν έτσι ώστε να παράγουν και να υπολογίζουν γρηγορότερα ένα αποτέλεσμα. Το πρόβλημα διαιρείται σε μικρότερα προβλήματα, τα οποία επιλύονται ταυτόχρονα και στη συνέχεια εξάγεται το τελικό αποτέλεσμα. Με αυτό τον τρόπο το πρόβλημα το οποίο καλείται να επιλύσει ο υπολογιστής θα εκτελεστεί γρηγορότερα αναλογικά ως προς τους επεξεργαστές τους οποίους διαθέτει. Υπάρχουν δύο κύριοι τρόποι οργάνωσης των παράλληλων συστημάτων, της κοινής μνήμης και της κατανεμημένης μνήμης.

1.2.1 Οργάνωση Κοινής Μνήμης (Shared Memory)

Στα συστήματα κοινής μνήμης, υπάρχει διασύνδεση μεταξύ των επεξεργαστών, μέσω της κύριας μνήμης η οποία είναι προσπελάσιμη άμεσα από όλους τους επεξεργαστές (Σχήμα

??). Οι τελευταίοι, έχουν πρόσβαση σε ένα κοινό χώρο διευθύνσεων, από όπου μπορούν να γνωρίζουν τυχόν αλλαγές σε μεταβλητές που επιθυμούν να χρησιμοποιήσουν. Επομένως, η επικοινωνία μεταξύ των επεξεργαστών πραγματοποιείται, μέσω της τροποποίησης κοινών μεταβλητών στη κοινόχρηστη μνήμη, που είναι ορατές σε όλους τους επεξεργαστές.

Η λειτουργία, στα συστήματα κοινής μνήμης, φαίνεται απλή θεωρητικά, τόσο στη δομή της όσο και στην αποτελεσματικότητα της, όμως δεν ισχύει κάτι τέτοιο. Η επικοινωνία σε ένα τέτοιο σύστημα επιτυγχάνεται συνήθως με τη χρήση ενός διαύλου. Στην περίπτωση που έχουμε μικρό αριθμό επεξεργαστών, το σύστημα μπορεί να είναι αποδοτικό. Άν όμως επιχειρήσουμε να αυξήσουμε τον αριθμό των επεξεργαστών, θα αυξηθεί και ο αριθμός των επικοινωνιών, ενώ η χωρητικότητα του διαύλου παραμένει ίδια. Επομένως, ένα μεγάλο μέρος του χρόνου της εκτέλεσης θα καταναλώνεται μόνο στην επικοινωνία με την μνήμη. Αυτό θα έχει ώς αποτέλεσμα, η απόδοση του συστήματος να μειώνεται με την αύξηση των επεξεργαστών. Έτσι δεν μπορούμε να έχουμε τέτοια συστήματα με μεγάλο αριθμό επεξεργαστών, με αποτέλεσμα να καταφύγουμε σε άλλου τύπου διασυνδέσεις, πλήν του διαύλου, όπως για παράδειγμα διακοπτικά δίκτυα πολλαπλών επιπέδων.

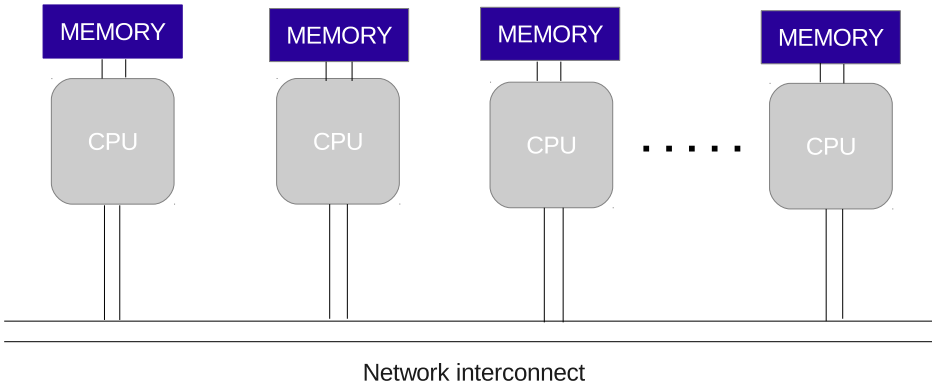


Σχήμα 1.1: Μοντέλο Κοινής Μνήμης

1.2.2 Σύστημα Κατανεμημένης Μνήμης (Distributed Memory)

Σε αντίθεση με τους πολυεπεξεργαστές κοινής μνήμης, στα συστήματα κατανεμημένης μνήμης, ο κάθε επεξεργαστής έχει την δική του ιδιωτική μνήμη και επικοινωνεί με τους άλλους επεξεργαστές μέσω ενός δικτύου (Σχήμα ??). Κάθε επεξεργαστής μπορεί άμεσα να προσπελάσει την μνήμη αυτή, να επεξεργαστεί τα δεδομένα της και να τα τροποποιήσει. Στην περίπτωση που ένας επεξεργαστής επιθυμεί να προσπελάσει δεδομένα που αντιστοιχούν σε μία ιδιωτική μνήμη ενός άλλου επεξεργαστή, τότε θα χρειαστεί να γίνει επικοινωνία μέσω μηνυμάτων, μεταξύ αυτών των δύο επεξεργαστών. Για παράδειγμα, ο επεξεργαστής 1 θέλει να προσπελάσει τα δεδομένα του επεξεργαστή 2. Ο επεξεργαστής 1 θα αποστείλει ένα μήνυμα στο δίκτυο, ζητώντας το/τα δεδομένο/να από την ιδιωτική μνήμη του επεξεργαστή 2. Ο τελευταίος με την σειρά του, μέσω μηνύματος, θα αποστείλει τα δεδομένα μέσα στο δίκτυο, που ζητήθηκαν από τον επεξεργαστη 1.

Ένα τέτοιο σύστημα, όπως είναι το σύστημα κατανεμημένης μνήμης, μπορεί να είναι κλιμακώσιμο σε σχέση με το μοντέλο κοινής μνήμης, με την έννοια ότι η εισαγωγή πολύ περισσότερων κόμβων και επεξεργαστών είναι εφικτή, ανάλογα με την τοπολογία του δικτύου διασύνδεσης. Το πρόβλημα που μπορεί να προκύψει, είναι ο φόρτος του δικτύου. Άν πολλά δεδομένα πρέπει να είναι προσπελάσιμα από πολλές διεργασίες τότε ο φόρτος επικοινωνίας είναι σημαντικός και έχει επιπτώσεις στο χρόνο εκτέλεσης του προγράμματος.



Σχήμα 1.2: Σύστημα Κατανεμημένης Μνήμης

1.3 Προγραμματισμός Παράλληλων Υπολογιστών

Ο παράλληλος προγραμματισμός θεωρείται αρκετά δυσκολότερος του κλασσικού σειριακού προγραμματισμού καθώς πρώτον, ο προγραμματιστής πρέπει να σπάσει σε κομμάτια το

πρόγραμμα και χειροκίνητα να αναθέσει φόρτο εργασίας στις υπολογιστικές οντότητες και δεύτερον, έχει σε μεγάλο βαθμό εξάρτηση από την φυσική οργάνωση του συστήματος.

1.3.1 Προγραμματισμός με μοντέλο κοινού χώρου διευθύνσεων

Ο πιο διαδεδομένος τρόπος προγραμματισμού συστημάτων κοινής μνήμης είναι η χρήση νημάτων POSIX ή διεργασιών. Στο συγκεκριμένο μοντέλο, χρησιμοποιούνται συχνότερα τα νήματα απ'ότι οι διεργασίες. Αυτό συμβαίνει, γιατί οι διεργασίες έχουν δικό τους ξεχωριστό χώρο διευθύνσεων για καθεμία, επομένως, η ασφάλεια πρόσβασης στην κοινή μνήμη πρέπει να εξασφαλίστει με κλήσεις συστήματος. Εν αντιθέσει, τα νήματα που έχουν δημιουργηθεί από την ίδια διεργασία, έχουν πρόσβαση στο χώρο διευθύνσεων της διεργασίας. Παρόλα αυτά ο προγραμματισμός με νήματα θεωρείται σχετικά δύσκολος και "χαμηλού" επιπέδου.

Το πρότυπο OpenMP [7] το οποίο εμφανίστηκε την τελευταία δεκαετία αποτελεί έναν από τους πιό δημοφιλής τρόπους προγραμματισμού για παράλληλα συστήματα κοινής μνήμης. Χαρακτηρίζεται τόσο για την ευκολία του, όσο και για την σταδιακή παραλληλοποίηση του προγράμματος, με την χρήση συγκεκριμένων εντολών. Το OpenMP, είναι μία διεπαφή εφαρμογών και προγραμμάτων (API) υλοποιημένο για τις γλώσσες προγραμματισμού C/C++ και Fortran και πλέον έχει επικρατήσει, ειδικά με την έλευση των πολυπύρηνων επεξεργαστών. Στο Κεφάλαιο 2 γίνεται εκτενής περιγραφή κάποιων βασικών λειτουργιών του προτύπου OpenMP.

1.3.2 Προγραμματισμός σε μοντέλο μεταβίβασης μηνυμάτων

Το πιό διαδεδομένο πρότυπο για προγραμματισμό στο μοντέλο μεταβίβασης μηνυμάτων (message passing), είναι το πρότυπο MPI [3]. Το MPI προσφέρει κάποιες έτοιμες ρουτίνες για την αποστολή και παραλαβή μηνυμάτων, καθώς και ρουτίνες που προσφέρουν πληθώρα λειτουργιών, χωρίς να αναγκάζει τον χρήστη να γνωρίζει τα πάντα για την αρχιτεκτονική του δικτύου που χρησιμοποιείται στο εσωτερικό του συστήματος. Το MPI μπορεί να προσφέρει σημαντικές επιδοσεις αλλά θεωρείται και αυτό σχετικά "χαμηλού" επιπέδου.

Λόγω της απουσίας της κοινόχρηστης μνήμης, η επικοινωνία μεταξύ των κόμβων γίνεται μέσω μηνυμάτων. Όμως, την τελευταία δεκαετία, αναπτύχθηκε η λογική της εικονικής κοινόχρηστης μνήμης σε αυτά τα συστήματα. Με την βοήθεια του λογισμικού δημιουργήθηκε η εικονική μνήμη, με βάση την κατανεμημένη μνήμη, που είναι μια ψευδαίσθηση ύπαρξης κοινής μνήμης για την επικοινωνία μεταξύ των επεξεργαστών. Το λογισμικό αυτό ονομάζεται sDSM (software Distributed Shared Memory) και αποτελεί έναν εναλλακτικό τρόπο για την πραγματοποίηση των προγραμμάτων χωρίς όμως να έχει πάντα τις καλύτερες επιδόσεις.

1.4 Αντικείμενο Εργασίας

Ο OMPi [8] είναι ένας μεταγλωττιστής για την γλώσσα C του προτύπου OpenMP, και υποστηρίζει εντολές του προτύπου OpenMP 3.1 και πρόσφατα έναν μεγάλο αριθμό των νέων γνωρισμάτων που προστέθηκαν και σχημάτισαν το πρότυπο OpenMP 4.0. Για το πρότυπο OpenMP θα αναφερθώ στο επόμενο κεφάλαιο (Κεφάλαιο 2). Πιο συγκεκριμένα, το αντικείμενο της διπλωματικής εργασίας είναι η υλοποίηση της υποδομής ακύρωσης (cancellation) στον μεταγλωττιστή OMPi όπως αυτό παρουσιάζεται από το πρότυπο OpenMP 4.0. Η εντολή ακύρωσης είναι μία αυτόνομη εντολή που επιβάλλει την ακύρωση ενός μπλόκ κώδικα που περικλύεται σε μία οδηγία. Το σημείο ακύρωσης είναι μία εντολή που την εισάγει

ο χρήστης, και ελέγχει για πιθανή ενεργοποιήση της ακυρωσης της οδηγίας που έχει οριστεί. Πιο αναλυτική αναφορά, σχετικά με την περιγραφη και την υλοποίηση για τις εντολές ακύρωσης στον μεταγλωττιστή OMPi, θα γίνει στα παρακάτω κεφάλαια.

1.5 Δομή της Εργασίας

Η διπλωματική εργασία είναι οργανωμένη ώς εξής :

- Κεφάλαιο 2 : Παρουσίαση και περιγραφή του προτύπου OpenMP για παράλληλο προγραμματισμό. Εξοικίωση με την χρήση εντολών, επεξήγηση αυτών και αναφορά σε κλήσεις συστήματος και μεταβλητών περιβάλλοντος. Μία εισαγωγή στα νέα γνωρίσματα που προστέθηκαν στο πρότυπο OpenMP 4.0 .
- Κεφάλαιο 3 : Παρουσίαση και περιγραφή του μεταφραστή OMPi . Σύντομη αναφορά στις λειτουργίες του μεταφραστή και στις δομές που περιλαμβάνει.
- Κεφάλαιο 4 : Παρουσίαση και περιγραφή των εργασιών [6] και της ομάδας εργασιών (*taskgroup*) των εντολών ακύρωσης (*Cancel Constructs*) καθώς και των σημείων ακύρωσης (*Cancellation point*). Αναλύεται ο τρόπος λειτουργίας για το καθένα και θέτονται οι περιορισμοί σύμφωνα με το πρότυπο OpenMP 4.0.
- Κεφάλαιο 5 : Υλοποίηση των εντολών ακύρωσης και σημείων ακύρωσης στον μεταγλωττιστή OMPi. Αναλυτική αναφορά στην υλοποίηση της κάθε δομής, παραδείγματα πάνω στην εφαρμογη των εντολών, και παράθεση του κώδικα με τις αλλαγές στον OMPi.
- Κεφάλαιο 6 : Περιγραφή και ανάλυση από την εκτέλεση διάφορων εφαρμογών (*epcc benchmarks*), για την μέτρηση της απόδοση στου OMPi μετά την εισαγωγή των νέων εντολών. Παρουσίαση εφαρμογής για την παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο και σχολιασμός αποτελεσμάτων.
- Κεφάλαιο 7 : Σύνοψη διπλωματικής εργασίας και προτάσεις για τον μέλλον που αφορούν την βελτιστοποίηση του μεταφραστή OMPi.

Κεφάλαιο 2

Το πρότυπο OpenMP

2.1 Εισαγωγή

Στην εισαγωγή της εργασίας, έγινε αναφορά για την ύπαρξη παράλληλων συστημάτων και λογισμικού που τα αξιοποιεί. Η ανάπτυξη τέτοιου λογισμικού, απαιτεί καλή διαχείριση περιβάλλοντος του προγράμματος (διαχείριση μεταβλητών, συναρτήσεων, κλειδαριών, κλπ.) από τον προγραμματιστή, έτσι ώστε να μπορούν αυτά, να είναι προσπελάσιμα από πολλαπλές οντότητες, με στόχο την διατήρηση της συνέπειας και της ακεραιότητας στις μεταβολές. Ο κώδικας που ειναι υπεύθυνος για τον συγχρονισμό των πολλών επεξεργαστών, αποτελείται συνήθως, από αρκετές εντολές και είναι δύσκολος στην αποσφαλμάτωση.

Αυτοί οι λόγοι οδήγησαν στην δημιουργία ενός νέου μοντέλου/προτύπου, του OpenMP για την συγγραφή παράλληλων προγραμμάτων, με γνώμονα την ευκολία στην χρήση του και την υψηλή απόδοση του.

Όπως επισημάνθηκε και στην προηγούμενη ενότητα, το OpenMP είναι μία διεπαφή εφαρμογών προγραμμάτων (Application Program Interface-API) για συστήματα κοινής μνήμης. Το API του OpenMP έχει οριστεί για τις γλώσσες προγραμματισμού C, C++, και Fortran, ενώ υποστηρίζεται κυρίως σε πλατφόρμες βασισμένες σε UNIX.

Τα OpenMP αποτελείται από:

- *Οδηγίες προς τον μεταγλωττιστή*. Απαιτούνται ορισμένες εντολές από τον χρήστη προς τον μεταφραστή, έτσι ώστε να μπορεί να κατανοήσει ο μεταγλωττιστής ποιά μέρη του προγράμματος επιθυμεί ο χρήστης να παραλληλοποιηθούν και με ποιόν τρόπο. Αυτές οι εντολές-κλειδιά είναι τα pragmas ή directives.
- *Κλήσεις Βιβλιοθήκης*. Είναι κλήσεις του OpenMP, με τις οποίες ο χρήστης μπορεί να πραγματοποιήσει μεταβολές ή να αποκτήσει πρόσβαση σε συγκεκριμένες πληροφορίες.
- *Μεταβλητές Περιβάλλοντος*. Είναι μεταβλητές που ρυθμίζουν τον τρόπο εκτέλεσης των παράλληλων περιοχών. Η μεταβολή στις μεταβλητές περιβάλλοντος μπορεί να γίνει είτε δυναμικά είτε στατικά.

Οι στόχοι του προτύπου OpenMP, είναι:

- Να αποτελέσει ένα πρότυπο που να είναι μεταφέρσιμο (*portable*), δηλαδή να υποστηρίζεται από πολλές αρχιτεκτονικές και πλατφόρμες, στα πλαίσια του μοντέλου της κοινής μνήμης.

- Να περιέχει ένα πεπερασμένο σύνολο οδηγιών. Δηλαδή το σύνολο των οδηγιών που παράγονται, πρέπει να είναι περιορισμένο αλλά να εμπλουτίζει με αρκετή παραλληλοποίηση το πρόγραμμα.
- Να είναι εύχρηστο και εύκολα κατανοητό για τον χρήστη.
- Να δίνει την δυνατότητα παραλληλοποίησης ενός προγράμματος σταδιακά με βάση το σειριακό πρόγραμμα χωρίς να το τροποποιήσει πλήρως.
- Να παρέχει την δυνατότητα τόσο χονδρού κόκκου, όσο και λεπτού κόκκου παραλληλίας.

2.2 Προγραμματιστικό μοντέλο OpenMP

Όπως ήδη αναφέρθηκε, το προγραμματιστικό μοντέλο του OpenMP στηρίζεται στην παραλληλία νημάτων με κοινή διαμοιραζόμενη μνήμη. Υλοποιεί το μοντέλο fork-join, που ο χρήστης μπορεί να αυξωμειώνει τον αριθμό των νημάτων που εκτελούν την εκάστοτε παράλληλη περιοχή.

Ένα πρόγραμμα που έχει συνταχθεί με βάση το πρότυπο OpenMP ξεκινά την εκτέλεσή του *σειριακά*, δηλαδή εκτελείται από ένα μόνο νήμα. Το νήμα αυτό ονομάζεται νήμα-αφέντης(*master thread*). Το νήμα-αφέντης συνεχίζει την εκτέλεσή του μέχρι να συναντήσει μία *περιοχή παραλληλίας*. Τότε δημιουργεί μία ομάδα νημάτων, που το μέγεθος αυτής είναι προκαθορισμένο από τον χρήστη, για να εκτελέσουν παράλληλα την συγκεκριμένη περιοχή. Μέσα στον παράλληλο κώδικα υπάρχει περίπτωση να συναντήσουμε ορισμένες εντολές, που ορίζουν τις λεγόμενες *περιοχές διαμοιρασμού εργασίας*. Στις περιοχές αυτές, τα νήματα της ομάδας, αναλαμβάνουν ένα μέρος εργασιών προς εκτέλεσή. Στο τέλος της παράλληλης περιοχής, όλα τα νήματα σταματούν την λειτουργία τους, εκτός από το νήμα-αφέντη που συνεχίζει σειριακά την εκτέλεσή του. Η παραπάνω διαδικασία, εκτελείται όσες φορές χρειαστεί.

2.3 Οδηγίες OpenMP για C\C++

Στο συγκεκριμένο κεφάλαιο, θα γίνει ένας πρόλογος για την βασική χρήση του OpenMP, την σύνταξη των εντολών που ακολουθείται στο πρότυπο, θα γίνει αναφορά για την λειτουργία και την κατανόηση των παράλληλων περιοχών και των περιοχών διαμοιρασμού εργασίας. Στη συνέχεια, θα γίνει αναφορά, για τον συγχρονισμό των νημάτων, τις μεταβλητές περιβάλλοντος, αλλά και μία συνοπτική περιγραφή των νέων χαρακτηριστικών που προστέθηκαν στο OpenMP 4.0.

2.3.1 Σύνταξη Οδηγιών OpenMP

Μία οδηγία στο OpenMP χωρίζεται σε 3 μέρη :

#pragma omp | Όνομα οδηγίας | [*φράση 1, . . .*]

- Το **#pragma omp** απαιτείται για όλες τις οδηγίες που συντάσσονται σε C\C++.
- Το Όνομα οδηγίας πρέπει να βρίσκεται ακριβώς μετά το **#pragma omp** . Υπάρχουν οδηγίες συγχρονισμού, οδηγίες παραλληλίας, αλλα και οδηγίες που ορίζουν περιοχές διαμοιρασμού εργασίας.

- Αμέσως μετά την οδηγία, τοποθετούνται οι φράσεις οδηγιών. Δεν υπάρχει κάποια συγκεκριμένη σειρά με την οποία θα τοποθετηθούν. Μπορεί να είναι παραπάνω από μια συνθήκη, αλλα μπορεί να μην οριστεί καμία. Στην τελευταία περίπτωση, οι συνθήκες εκτέλεσης των οδηγιών καθορίζονται από το runtime.

2.3.2 Παράλληλες Περιοχές

Μία παράλληλη περιοχή, ορίζεται ως ένα τμήμα κώδικα που είναι μέσα σε αγκύλες και στην αρχή του υπάρχει η εντολή:

#pragma omp parallel [*φράση 1, . . .*]

Όταν το αρχικό νήμα *(master-thread)* συναντήσει την παραπάνω εντολή, θα δημιουργήσει μία ομάδα νημάτων προκαθορισμένου μεγέθους. Κάθε ένα από αυτά τα νήματα θα εκτελέσει αυτούσιο τον κώδικα, που βρίσκεται μέσα στις αγκύλες. Μέρος της ομάδας είναι και το master-thread και έχει πάντα σαν αριθμό αναγνωριστικού το 0. Κάθε νήμα στην ομάδα μπορεί να φτάσει σε οποιοδήποτε σημείο στην παράλληλη περιοχή, σε ακαθόριστη χρονική στιγμή. Στο τέλος της περιοχής αυτής, πάντα υπονοείται ένα φράγμα, ώστε να συγχρονιστούν τα νήματα της ομάδας και να συνεχίσει την εκτέλεσή του μόνο το master-thread.

Η διεπαφή εφαρμογών προγραμμάτων επιτρέπει εμφωλευμένες οδηγίες. Δηλαδή, δίνεται η δυνατότητα δημιουργίας μίας παράλληλης ομάδας μέσα σε ήδη υπάρχουσα παράλληλη ομάδα. Στην συγκεκριμένη περίπτωση, κάθε νήμα δημιουργεί μία νέα ομάδα και σε αυτή, γίνεται το master-thread.

Είναι σημαντικό να αναφέρουμε ότι, ο αριθμός των νημάτων που δημιουργούνται κατά την είσοδο σε μία παράλληλη περιοχή, ελέγχεται από τον χρήστη με κλήσεις βιβλιοθήκης, μεταβλητές περιβάλλοντος και με συνθήκες οδηγιών που τοποθετούνται στην εκάστοτε οδηγία. Δηλαδή:

- με την χρήση της κλήσεις βιβλιοθήκης, *omp_set_num_threads()*, η οποία καλείται μέσα από το πρόγραμμα.
- με την αλλαγή μεταβλητής περιβάλλοντος *OMP_NUM_THREADS*, την οποία αλλάζουμε πρίν την εκτέλεση του προγράμματος.
- με την χρήση της συνθηκης οδηγίας, *num_threads()* που ακολουθεί μετά την οδηγία *parallel*.

Ένα παράδειγμα είναι το 2.1 , το οποίο θα δώσει την παρακάτω έξοδο.

```
1 int main()
2 {
3   int thread_id;
4   omp_set_num_threads(6);
5   #pragma omp parallel
6   {
7     thread_id=omp_get_thread_num();
8     printf("Hello world, thread %d \n",thread_id);
9   }
```

Κώδικας 2.1: Ενα απλό παράδειγμα παράλληλης περιοχής.

Σάν έξοδο θα έχουμε :

```
Hello world, thread 0
Hello world, thread 1
Hello world, thread 2
Hello world, thread 3
Hello world, thread 4
Hello world, thread 5
```

2.3.3 Περιοχές Διαμοιρασμού Εργασίας (Workshare Regions)

Μέσα σε μία παράλληλη περιοχή, μπορεί να υπάρχουν περιοχές διαμοιρασμού εργασίας. Δηλαδή, περιοχές στις οποίες, τα νήματα που ανήκουν στην παράλληλη ομάδα, μοιράζονται τον φόρτο εργασίας. Σε αντίθεση με τις περιοχές παραλληλίας, στις περιοχές διαμοιρασμού δεν δημιουργούνται νέα νήματα. Στο τέλος κάθε τέτοιας περιοχής, υπάρχει ένα φράγμα(*barrier*), ώστε να επιτυγχάνεται ο συγχρονισμός των νημάτων. Υπάρχει περίπτωση το φράγμα αυτό να παραλειφθεί, όταν ώς συνθήκη οδηγίας επιλέξουμε το *nowait*. Υπάρχουν τρείς τύποι περιοχών διαμοιρασμού εργασίας:

- **οδηγία for**: Αμέσως μετά από την οδηγία πρέπει να ακολουθεί μία δομή επανάληψης *for* η οποία και θα παραλληλοποιηθεί. Οι επαναλήψεις του βρόγχου διαμοιράζονται στα νήματα της ομάδας με τρόπο που θα περιγραφεί στην Ενότητα 2.3.4.

- **οδηγία sections**: Με την οδηγία *sections* η συνολική εργασία χωρίζεται σε διακριτά μέρη, που το καθένα εκτελείται από ένα νήμα της ομάδας. Ουσιαστικά, κάθε νήμα αναλαμβάνει την εκτέλεση ενός δομημένου κώδικα που περιέχεται μέσα στην περιοχή της οδηγίας sections, και ορίζεται με την εντολή *#pragma omp section*.

- **οδηγία single**: Με αυτή την οδηγία εξασφαλίζουμε ότι, μόνο το πρώτο νήμα που την συναντήσει, θα εκτελέσει τον κώδικα που ακολουθεί. Όλα τα υπόλοιπα νήματα, θα προσπεράσουν την οδηγία και θα περιμένουν στο τέλος αυτής, μέχρι το νήμα που εκτελεί τον κώδικα της οδηγίας single να τελειώσει την εκτέλεσή του. Μία παραλλαγή της οδηγίας αυτής είναι η οδηγία *master*, η οποία είναι όμοια με την λειτουργία της single, με τη διαφορά ότι μόνο το νήμα master πρέπει αυστηρά να εκτελέσει τον κώδικα που περικλύει.

2.3.4 Φράσεις Οδηγιών του OpenMP

Οι οδηγίες OpenMP, επιδέχονται ορισμένες συνθήκες που μπορούν να ρυθμίσουν την λειτουργία της οδηγίας. Αυτές ονομάζονται, φράσεις οδηγιών. Με αυτές καθορίζουμε τον τρόπο που θα παραλληλοποιηθεί ο κώδικας που ακολουθεί, τον τρόπο διαμοιρασμού των μεταβλητών, δηλαδή ποιές μεταβλητές θα είναι κοινόχρηστες μεταξύ των νημάτων, ποιές θα είναι ιδιωτικές μεταβλητές, καθώς και τον τρόπο συγχρονισμού ή μη των νημάτων. Οι συνθήκες οδηγίας, όπως είδαμε και σε προηγούμενη ενότητα, δηλώνονται ακριβώς μετά το όνομα οδηγίας.

Οι συνθήκες οδηγίας είναι οι εξής:

- **if(expression)**: Μόνο αν το expression μέσα στο if είναι αληθές, πραγματοποιείται η ενέργεια της οδηγίας.

- **shared(list of variables)**: Δήλωση της λίστας των κοινόχρηστων μεταβλητών που θα προσπελαθούν από τα νήματα που ανήκουν στην παράλληλη ομάδα. Δηλαδή τα νήματα προσπελαύνουν τις ίδιες θέσεις μνήμης.

- **private(list of variables)**: Δήλωση της λίστας των ιδιωτικών μεταβλητών. Κάθε νήμα που ανήκει στην παράλληλη ομάδα, προσπελαύνει και επεξεργάζεται τις δικές του ιδιωτικές μεταβλητές. Οι ιδιωτικές μεταβλητές των νημάτων δεν είναι αρχικοποιημένες.

- **firstprivate(list of variables)**: Δήλωση της λίστας των αντιγράφων ιδιωτικών μεταβλητών. Κάθε νήμα που ανήκει στην παράλληλη ομάδα, προσπελαύνει και επεξεργάζεται το δικό του αντίγραφο των αντίστοιχων μεταβλητών που έχει το master νήμα. Οι ιδιωτικές μεταβλητές των νημάτων αρχικοποιούνται με τις αντίστοιχες τιμές των μεταβλητών του master-thread.

- **lastprivate(list of variables)**: Δήλωση της λίστας των ιδιωτικών μεταβλητών. Κάθε νήμα που ανήκει στην παράλληλη ομάδα, προσπελαύνει και επεξεργάζεται τις ιδιωτικές μεταβλητές του. Στην τελευταία επανάληψη του βρόγχου for ή στο τελευταίο τμήμα της παράλληλης περιοχής που εκτελέστηκε, γίνεται η αντιγραφή των ιδιωτικών μεταβλητών στις αντίστοιχες μεταβλητές του master-thread.

- **nowait**: Η συγκεκριμένη συνθήκη οδηγίας, χρησιμοποιείται για να ακυρωθεί ο συγχρονισμός των νημάτων στο τέλος της αντίστοιχης οδηγίας.

- **num_threads(number)**: Με αυτή την συνθήκη οδηγίας, μπορούμε να καθορίσουμε τον αριθμό των νημάτων που θα δημιουργηθούν κατά την έναρξη μιας παράλληλης περιοχής, και στη συνέχεια θα την εκτελέσουν.

- **schedule(*type* [,*chunksize*])**: Χρησιμοποιείται για να καθορίσει τον τρόπο διαμοίρασης των επαναλήψεων ενός επαναληπτικού βρόγχου καθώς και το μέγεθος της εργασίας που θα εκτελέσει κάθε νήμα (*chunksize*). Οι τρόποι χρονοπρογραμματισμού είναι οι τέσσερις παρακάτω:

  - **static**: Οι επαναλήψεις χωρίζονται στατικά σε όλα τα νήματα της παράλληλης ομάδας που εκτελούν τον επαναληπτικό βρόγχο. Κάθε νήμα αναλαμβάνει έναν αριθμό επαναλήψεων ίσο με το chunksize που έχει οριστεί, όποτε αυτό είναι εφικτό.
  - **dynamic**: Οι επαναλήψεις χωρίζονται σε κομμάτια ίσα με το chunksize που έχει οριστεί, και μοιράζονται στα νήματα. Όταν ένα νήμα τελειώσει την εκτέλεση του δικού του κομματιού, τότε δυναμικά του αναθέτεται άλλο κομμάτι προς εκτέλεση, μέχρι να εκτελεστούν όλες οι επαναλήψεις. Άν το chunksize δεν έχει οριστεί, τότε θεωρείται ίσο με 1.
  - **guided**: Οι επαναλήψεις χωρίζονται σε κομμάτια ίσα με το chunksize που έχει οριστεί, και μοιράζονται στα νήματα. Όταν ένα νήμα τελειώσει την εκτέλεση του δικού του κομματιού, τότε δυναμικά του αναθέτεται άλλο εκθετικά μειωμένο κομμάτι, προς εκτέλεση.
  - **runtime**: Στον συγκεκριμένο τρόπο διαμοιρασμού, η επιλογή του αλγορίθμου δρομολόγησης εξαρτάται από τις αντίστοιχες μεταβλητές περιβάλλοντος.

- **reduction(operation: list of variables)**: Με την χρήση αυτής της συνθήκης, δημιουργείται για κάθε μεταβλητή της λίστας ένα τοπικό αντίγραφο για κάθε νήμα. Στο τέλος, πραγματοποιείται η πράξη που έχει οριστεί, μεταξύ των τοπικών αντιγράφων, ώστε να εξασφαλίζεται η συνέπεια και η ασφάλεια της τιμής στην αντίστοιχη μεταβλητή του master thread.

2.3.5 Οδηγίες Συγχρονισμού για C\C++

Το πρότυπο OpenMP παρέχει ορισμένες οδηγίες συγχρονισμού μεταξύ των νημάτων, οι οποίες είναι απαραίτητες για την ασφάλεια και την συνέπεια στις αλλαγές κοινόχρηστων μεταβλήτων και στην πρόσβαση σε αυτές κατά την διάρκεια εκτέλεσης του κώδικα. Οι οδηγίες συχρονισμού παραθέτονται και περιγράφονται παρακάτω:

- **Οδηγία Critical**: Η οδηγία αυτή, ορίζει περιοχές του κώδικα οι οποίες πρέπει να εκτελούνται αυστηρά από ένα νήμα σε κάθε χρονική στιγμή, γιατί σε αυτή περιέχονται κοινόχρηστες μεταβλητές και πρέπει να διασφαλίσουμε την ακεραιότητα αυτών. Οι περιοχές αυτές, ονομάζονται κρίσιμες περιοχές. Όσο ένα νήμα είναι σε κρίσιμη περιοχή, τα υπόλοιπα νήματα περιμένουν ώστε να να αποκτήσουν το δικαίωμα να έχουν πρόσβαση σε αυτή.

- **Οδηγία atomic**: Η οδηγία atomic καθορίζει οτι μία συγκεκριμένη κοινόχρηστη μεταβλητή θα ανανεωθεί ατομικά από ένα μόνο νήμα την φορά. Συνεπώς η χρήση της, προορίζεται για απλές εντολές ενημέρωσης και ανανέωσης της τιμής μιας κοινόχρηστης μεταβλητής, για την οποία πρέπει να εξασφαλιστεί ότι κάποιο άλλο νήμα δεν θα διακόψει ή θα αλλάξει την διαδικασία εγγραφής της.

- **οδηγία barrier**: Με την χρήση αυτής της οδηγίας, δίνουμε εντολή σε όλα τα νήματα της ομάδας να συγχρονιστούν σε ένα σημείο. Μόλις ένα νήμα συναντήσει την οδηγία barrier, σταματάει προσωρινά την λειτουργία του μέχρι την στιγμή που όλα τα υπόλοιπα νήματα φτάσουν στο ίδιο σημείο. Στη συνέχεια, όταν όλα τα νήματα συγχρονιστούν, συνεχίζουν την εκτέλεση του κώδικα που ακολουθεί. Η οδηγία barrier δεν μπορεί να εφαρμοστεί σε κάποιο υποσύνολο των νημάτων μιάς ομάδας.

- **Οδηγία flush**: Η flush οδηγία, αφορά στη συνέπεια της μνήμης σε μια δεδομένη χρονική στιγμή. Πιο συγκεκριμένα, καθορίζεται ένα σημείο συγχρονισμού, στο οποίο, για την εκτέλεση του κώδικα επιβάλλεται να υπάρχει πλήρης συνέπεια μνήμης, δηλαδή να μην εκρεμμούν εγγραφές σε αυτή. Επομένως, όταν συναντήσουμε αυτή την οδηγία, η τρέχουσα τιμή μιας κοινής μεταβλητής, γράφεται απευθείας στην μνήμη. Είναι η γνωστή διαδικασία *write back*.

- **Οδηγία ordered**: Η οδηγία ordered εμφανίζεται κυρίως σε επαναληπτικούς βρόγχους. Ουσιαστικά, η χρήση της επιβάλλει στον συγκεκριμένο επαναληπτικό βρόγχο, ο τρόπος εκτέλεσης των επαναλήψεων να γίνει ακριβώς με την σειρά που θα εκτελούνταν άν είχαμε σειριακή εκτέλεση.

2.4 Κλήσεις Βιβλιοθήκης Runtime

Το OpenMP παρέχει και κάποιες κλήσεις βιβλιοθήκης που παρέχονται από το runtime. Είναι κλήσεις που υποστηρίζουν την ευελιξία στη εκτέλεση του προγράμματος, όπως το κλείδωμα περιοχών κώδικα, την μεταβολή (δυναμικά ή μή) στον αριθμό των νημάτων και την επιστροφή πληροφοριών που αφορούν την εκτέλεση του προγράμματος. Οι κυριότερες εξ' αυτών, είναι οι εξής:

- **void omp_set_num_threads (int num_threads)** : Με την κλήση αυτής της συνάρτησης, ορίζεται ο αριθμός των νημάτων που θα εκτελέσουν μία παράλληλη περιοχή. Μόνο το νήμα-αφέντης έχει το δικαίωμα να καλέσει την συγκεκριμένη συνάρτηση, δηλαδή

όταν έχουμε σειριακή εκτέλεση. Άν έχει ενεργοποιηθεί ο δυναμικός ορισμός των νημάτων, τότε η συνάρτηση επιστρέφει τον μέγιστο αριθμό νημάτων που έχει οριστεί (OMP_NUM_THREADS). Στην περίπτωση που δεν έχει ενεργοποιηθεί ο δυναμικός ορισμός, η συνάρτηση επιστρέφει τον συγκεκριμένο αριθμό νημάτων num_threads που έχουμε ορίσει.

- **int omp_get_num_threads (void)** : Η συνάρτηση αυτή, επιστρέφει τον αριθμό των νημάτων που εκτελούν την παράλληλη περιοχή που βρισκόμαστε. Στην περίπτωση που η συνάρτηση κληθεί εκτός παράλληλης περιοχής, τότε μας επιστρέφει 1, που σημάινει ότι η εκτέλεση της περιοχής που βρισκόμαστε, εκτελείται μόνο από το νήμα master.

- **void omp_get_thread_num (void)** ; Η κλήση αυτής της συνάρτησης επιστρέφει το αναγνωριστικό ενός νήματος που την κάλεσε. Το νήμα master έχει το αναγνωριστικό 0, ενώ τα άλλα νήματα παίρνουν τιμές από 1 έως OMP_NUM_THREADS - 1.

- **void omp_set_nested (int number)** : Η κλήση της συνάρτησης αυτής ενεργοποιεί ή απενεργοποιεί την δυνατότητα εμφωλευμένων παράλληλων περιοχών. Η τιμή του number μπορεί να είναι 1 για ενεργοποίηση και 0 για απενεργοποίηση.

- **void omp_get_nested (void)**: Η συνάρτηση αυτή επιστρέφει την κατάσταση για την δυνατότητα εμφωλευμένων παράλληλων περιοχών. Επιστρέφει 1 άν είναι ενεργή και 0 σε αντίθετη περίπτωση.

- **omp_set_dynamic(int number)** : Με την κλήση αυτής της συνάρτησης ενεργοποιείται ή απενεργοποιείται η δυνατότητα δυναμικής ανάθεσης αριθμού των νημάτων. Η τιμή του number μπορεί να είναι 1 για ενεργοποίηση και 0 για απενεργοποίηση.

- **omp_get_dynamic (void)** : Η συνάρτηση αυτή επιστρέφει την κατάσταση για την δυναμική ανάθεση στον αριθμό των νημάτων. Επιστρέφει 1 άν είναι ενεργή και 0 σε αντίθετη περίπτωση.

- **omp_in_parallel (void)** : Επιστρέφει 1 άν βρισκόμαστε σε παράλληλη περιοχή, και 0 σε διαφορετική περίπτωση.

- **omp_num_procs (void)** : Η συνάρτηση επιστρέφει τον αριθμό των επεξεργαστών που υπάρχουν στο τρέχον σύστημα.

- **void omp_init_lock (omp_lock_t *lock)** : Η κλήση της συνάρτησης, αρχικοποιεί μία κλειδαριά που δείχνει ο δείκτης lock. Η αρχική κατάσταση της κλειδαριάς είναι ¨unlock¨.

- **void omp_set_lock (omp_lock_t *lock)** : Η κλήση αυτής της συνάρτησης, Θέτει την κλειδαριά μή διαθέσιμη. Κάθε νήμα αναγκάζεται να περιμένει εκεί μέχρις ότου η κλειδαριά στην οποία δείχνει το lock να γίνει διαθέσιμη. Η συναρτηση αυτή, απαγορεύεται να κληθεί εάν το lock δεν είναι αρχικοποιημένο.

- **void omp_unset_lock (omp_lock_t *lock)** : Με την κλήση της συνάρτησης αυτής, αποδευσμεύεται η κλειδαριά από το νήμα που την χρησιμοποιούσε. Η συναρτηση αυτή, απαγορεύεται να κληθεί εάν το lock δεν είναι αρχικοποιημένο.

- **void omp_test_lock (omp_lock_t *lock)** : Η συνάρτηση αυτή προσπαθεί να αρχικοποιήσει μία κλειδαριά, με την διαφορά ότι τα νήματα που την συναντούν δεν μπλοκάρουν την εκτέλεσή τους. Επιστρέφει 1 άν η κλειδαριά που δείχνει ο δείκτης lock έχει αρχικοποιηθεί επιτυχώς και 0 σε διαφορετική περίπτωση.

2.5 Μεταβλητές Περιβάλλοντος

Στο OpenMP ο έλεγχος για τον τρόπο εκτέλεσης μιας παράλληλης περιοχής μπορεί να γίνει και μέσω της χρήσης μεταβλητών περιβάλλοντος. Όλες οι μεταβλητές αυτές, συντάσσονται με κεφαλαία γράμματα και δεν μεταβάλλουν την τιμή τους σε καμία περίπτωση, εκτός της χειροκίνητης μεταβολής αυτών, από τον χρήστη, μέσα απο τις συναρτήσεις του runtime που περιγράψαμε προηγουμένως. Οι βασικότερες μεταβλητές περιβάλλοντος είναι οι παρακάτω :

- **OMP_SCHEDULE** : Η τιμή της καθορίζει τον τρόπο διαμοίρασης των επαναλήψεων ενός βρόγχου στα διαθέσιμα νήματα. Χρησιμοποιηείται μόνο για τις οδηγίες parallel for και for και η χρήση του είναι ως εξής (υποθέτοντας το bash shell):

export OMP_SCHEDULE="schedule,chunksize"

- **OMP_NUM_THREADS** : Η τιμή αυτής της μεταβλητής, ορίζει τον αριθμό νημάτων που μπορεί να εκτελέσουν μία παράλληλη περιοχή. Η χρήση του είναι ως εξής:

export OMP_NUM_THREADS=number

- **OMP_DYNAMIC** : Με την μεταβλητή αυτή μπορούμε να ενεργοποιήσουμε ή να απενεργοποιήσουμε την δυναμική αυξομείωση στον αριθμό των νημάτων που είναι διαθέσιμα για την εκτέλεση των παράλληλων περιοχών. Η χρήση του είναι ως εξής:

export OMP_DYNAMIC=boolean

- **OMP_NESTED** : Με την μεταβλητή αυτή μπορούμε να ενεργοποιήσουμε ή να απενεργοποιήσουμε την δυνατότητα ύπαρξης εμφωλευμένων παράλληλων περιοχών. Η χρήση του είναι ως εξής:

export OMP_NESTED=boolean

- **OMP_CANCELLATION** :Με την μεταβλητή αυτή μπορούμε να ενεργοποιήσουμε ή να απενεργοποιήσουμε την δυνατότητα της ακύρωσης της περικλειόμενης δομής. Η χρήση του είναι ως εξής:

export OMP_CANCELLATION=boolean

2.6 Νέες λειτουργίες του OpenMP v.4.0

Το πρότυπο OpenMP εξελίσσεται συνεχώς, ώστε να προσφέρει περισσότερη ευελιξία στην χρήση του, αλλά και περισσότερες δυνατότητες στους χρήστες, με την προσθήκη νέων γνωρισμάτων που έγιναν γνωστά στην πρόσφατη έκδοση OpenMP 4.0. Συνοπτικά, θα γινεί παρακάτω, μία σύντομη αναφορά και περιγραφή στα βασικότερα καινούργια γνωρίσματα.

- *device constructs* : Για την λειτουργία αυτών των δομών θα πρέπει να δώσουμε κάποιους ορισμούς:

  - *device* : Είναι μία συσκευή η οποία μπορεί να εκτελεί κομμάτια κώδικα.
  - *host device* : Είναι η συσκευή που ένα πρόγραμμα σε OpenMP ξεκινά την εκτέλεσή του, συνήθως ο βασικός επεξεργαστής του συστήματος.
  - *target device* : Είναι η συσκευή στη οποία μεταφέρονται δεδομένα και κώδικας προς εκτέλεση, από τον host, για παράδειγμα μία κάρτα γραφικών (GPU).

  Ο host device μεταφέρει κώδικα και δεδομένα στα target devices. Ο κώδικας και τα δεδομένα που μεταφέρονται ονομάζεται περιοχή στόχου (target region). Ένα target region μπορεί να εκτελεστεί από την target device αλλά και από την host device. Η διεργασία του host που συναντάει μία δομή στόχου περιμένει στο τέλος αυτής μέχρι να ολοκληρωθεί η εκτέλεση του target region, και στην συνέχεια απαιτείται συγχρονισμός.

- *Cancellation Construct/Points* : Αποτελεί το αντικείμενο της τρέχουσας διπλωματικής εργασίας. Επιτρέπει την ακύρωση της περικλειόμενης περιοχής απο την οδηγία που έχει οριστεί. Πιο αναλυτική αναφορά θα γίνει στα Κεφάλαια 4 και 5.

- *Taskgroup Construct* : Είναι μία δομή η οποία επιτρέπει πιο βαθύ συγχρονισμό, μεταξύ των ιεραρχιών, των εργασιών. Πιο συγκεκριμένα, θα γίνει αναφορά στο Κεφάλαιο 4 στην ενότητα 3.4.

- *Task dependence* : Είναι ένας καινούργιος όρος για την δομή των εργασιών. Επιτρέπει και προσδιορίζει την εξάρτηση ή μη, μεταξύ των εργασιών. Όταν μία διεργασία Α εξαρτάται από μία άλλη διεργασία Β, δεν επιτρέπεται η εκτέλεσή της Α, μέχρι να τελειώσει την εκτέλεση της η διεργασία Β.

Κεφάλαιο 3

Ο μεταφραστής OMPi

3.1 Δομή του OMPi

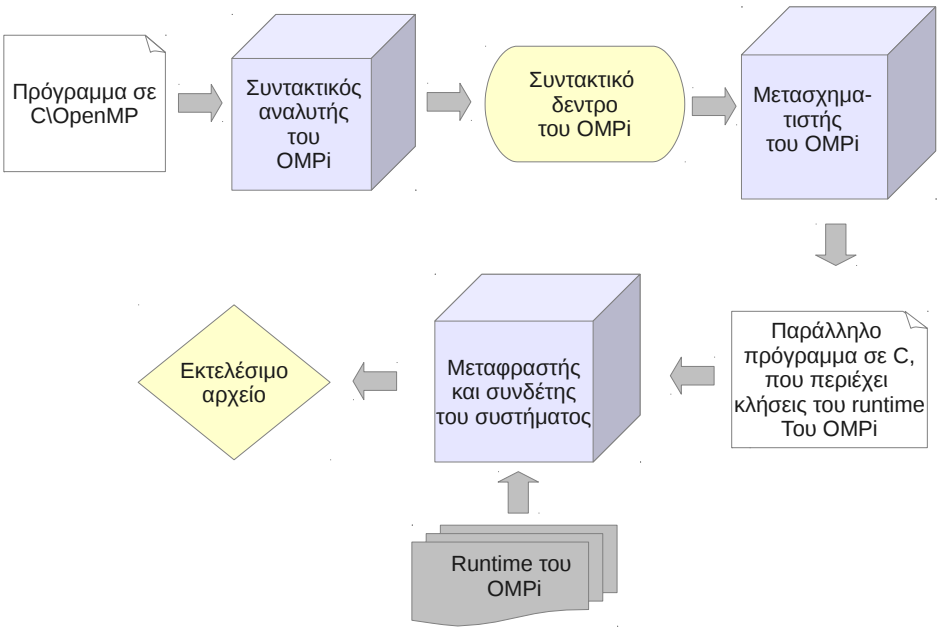
Ο OMPi είναι ένας μεταφραστής (compiler) προγραμμάτων OpenMP που συντάσσονται σε γλώσσα C. Η δομή του αποτελείται από 2 τμήματα, τον μεταγλωττιστή και το runtime. Αυτά τα δύο τμήματα συνεργάζονται ώστε να παράξουν ένα κώδικα που υποστηρίζει παράλληλη επεξεργασία όταν ζητηθεί. Ο μεταγλωττιστής αναλαμβάνει να μετατρέψει ένα πρόγραμμα σε C/OpenMP σε πολυνηματικό κώδικα C ο οποίος έχει εμπλουτιστεί με κλήσεις προς το runtime του OMPi.

Ο OMPi παρέχει ποικίλες επιλογές ώς προς τον τύπο των νημάτων που θα εκτελέσουν μια παράλληλη περιοχή καθώς δημιουργεί αόριστες εκτελεστικές οντότητες, την υλοποίηση των οποίων αναλαμβάνει το runtime (π.χ. νήματα POSIX επιπέδου πυρήνα) χρησιμοποιώντας τις διαθέσιμες βιβλιοθήκες που παρέχονται από τον OMPi.

3.2 Ο μετασχηματιστής πηγαίου σε πηγαίο κώδικα (S2S)

Ο μεταφραστής του OMPi παίρνει ως είσοδο το πρόγραμμα σε C, με δυνατότητα υποστήριξης εντολών σε OpenMP, και παράγει ένα νέο πηγαίο κώδικα, στο οποίο έχει ενσωματώσει κλήσεις προς το σύστημα runtime. Η διαδικασία της μεταγλώτισης, ολοκληρώνεται μετά από μία σειρά σταδίων, όπως φαίνεται στο Σχήμα 3.1. Το αρχικό στάδιο, όπως σε όλους τους μεταφραστές, είναι η συντακτική ανάλυση του κώδικα. Κατά την εκτέλεση του συντακτικού αναλυτή, το πρόγραμμα διασχίζεται(*parsing*) και δημιουργείται ένα αφηρημένο συντακτικό δέντρο που αναφέρεται στο πρόγραμμα. Έπειτα, το παραγώμενο δέντρο δίνεται ως είσοδο στο επόμενο στάδιο, του μετασχηματισμού. Στον μετασχηματισμό, διασχίζεται κάθε κόμβος του συντακτικού δέντρου και όπου υπάρχει μία οδηγία/δήλωση σε OpenMP τότε αυτός αντικαθίσταται με την αντίστοιχη κλήση runtime του OMPi.

Στην τρίτη φάση της μεταγλώτισης, γίνεται μία διάσχιση του πλέον μετασχηματισμένου δέντρου και παράγεται ο C κώδικας. Στη συνέχεια, ο παραγόμενος κώδικας θα μεταφραστεί από τον C μεταφραστή του συστήματος, ο οποίος θα αναλάβει να το συνδέσει και με τη βιβλιοθήκη runtime που έχει επιλεγεί για να υλοποιήσει τις υπολογιστικές οντότητες.



Σχήμα 3.1: Στάδια λειτουργίας του μεταφραστή OMPi

Ο μετασχηματισμός που απαιτεί τη μεγαλύτερη ανάλυση είναι αυτός της οδηγίας *parallel*. Ο παραγόμενος κώδικας του παραδείγματος 2.1 φαίνεται στον Παραγόμενος κώδικα 3.2

```
1 int __original_main(int _argc_ignored, char ** _argv_ignored)
2 # 5 "parallel.c"
3 {
4   int thread_id;
5
6   omp_set_num_threads(6);
7   {
8     struct __shvt__ {
9         int (* thread_id);
10      } _shvars;
11
12    _shvars.thread_id = &thread_id;
13    ort_execute_parallel(_thrFunc0_, (void *) &_shvars, -1, 0, 1);
14  }
15 }
```

Κώδικας 3.2: Ο παραγόμενος κώδικας του μεταφραστή OMPi για τον παράδειγμα.

Όταν κατα τη διάρκεια του parsing εμφανιστεί η οδηγία *parallel*, τότε ο κώδικας που βρίσκεται στο εσωτερικό της οδηγίας, μεταφέρεται σε μια συνάρτηση __thrFunc#__ , όπου '#' είναι ο αύξων αριθμός της συνάρτησης ξεκινώντας την αρίθμηση από το 0. Στην θέση της οδηγίας και του κώδικα αυτής, τοποθετεί μια κλήση του συστήματος runtime του OMPi, με το όνομα ort_execute_parallel(). Η συνάρτηση αυτή έχει 5 ορίσματα. Το πρώτο όρισμα είναι η συνάρτηση που θα εκτελέσουν τα νήματα της παράλληλης ομάδας (__thrFunc#__). Δεύτερο όρισμα είναι μία δομή με τις μεταβλητές που θα πρέπει να είναι κοινές μεταξύ των νημάτων. Το τρίτο όρισμα είναι ο αριθμός των νημάτων που θα εκτελέσουν την παράλληλη περιοχή.

Όταν δεν υπάρχει η φράση num_threads() που δηλώνει τον αριθμό των νημάτων μετά την οδηγία του *parallel*, τότε δίνεται η τιμή -1. Στην συγκεκριμένη περίπτωση, μέσω κλήσεων συστήματος του runtime, αντιλαμβάνεται τον αριθμό των επεξεργαστών που υπάρχουν στο σύστημα και δημιουργεί τόσα νήματα όσοι είναι οι επεξεργαστές.Το τέταρτο και το πέμπτο όρισμα, είναι κάποια flags.

3.3 Το σύστημα Runtime του OMPi

Το σύστημα runtime του OMPi [2], αναλαμβάνει την επεξεργασία των υπολογιστικών οντοτήτων που προορίζονται να εκτελέσουν τα τμήματα κώδικα που απαιτούν παραλληλοποίηση. Το σύστημα runtime, αποτελείται από δύο τμήματα. Το ένα ονομάζεται ORT και είναι υπεύθυνο να συνδυάζει και να χειρίζεται τις υπολογιστικές οντότητες του OMPi. Το δεύτερο τμήμα είναι το EELIB, το οποίο θα υλοποιεί τις υπολογιστικές οντότητες. Είναι σημαντικό να σημειωθεί ότι τα δύο τμήματα αυτά είναι ανεξάρτητα μεταξύ τους. Ο OMPi παρέχει μια πληθώρα υπολογιστικών οντοτήτων, με κύριο γνώμονα την τήρηση της ανεξαρτησίας αυτών των δύο τμημάτων.

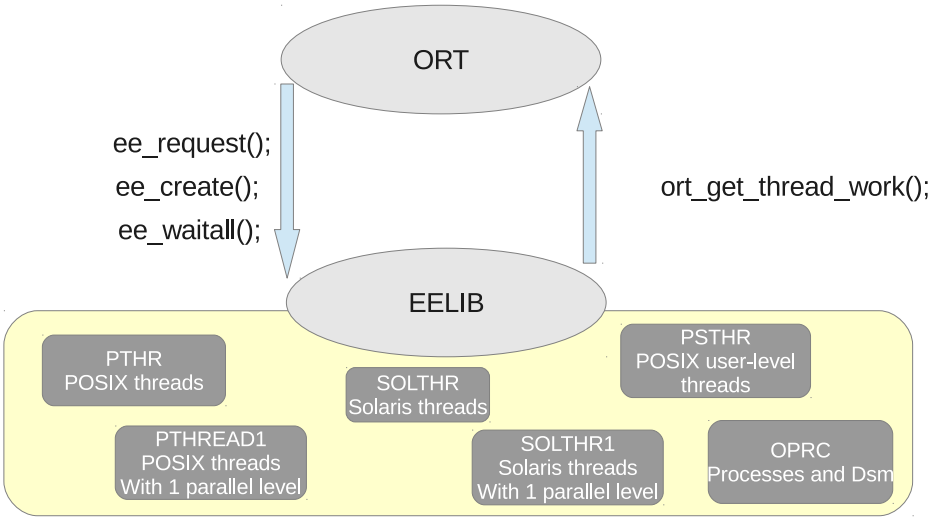
Στην εκκίνηση του προγράμματος, εισάγεται από τον μετασχηματιστή η κλήση προς το σύστημα runtime, ort_initialize(). Στην κλήση αυτής της συνάρτησης το νήμα master θα αναλάβει να αρχικοποιήσει κάποιες σημαντικές για την εκτέλεση παραμέτρους. Αρχικά, λαμβάνει τις τιμές των μεταβλητών περιβάλλοντος και ανάλογα με τις τιμές τους ξεκινά μια διαδικασία αρχικοποιήσεων. Έπειτα, καλείται η αντίστοιχη συνάρτηση αρχικοποίησης του τμήματος EELIB, η ee_initialize(). Σε αυτή την συνάρτηση, δηλώνεται ο αριθμός των υπολογιστικών οντοτήτων που θα χρειαστούν. Στον OMPi, κατα την αρχικοποίηση δημιουργούνται οι υπολογιστικές οντοτήτες. Όταν χρειαστούν, απλά τις ξυπνάει και τους αναθέτει κάποιες εργασίες. Η τελευταία και πιο σημαντική αρμοδιότητα της συνάρτησης αρχικοποίησης, είναι η δημιουργία της δομής ελέγχου (*eecb*) του νήματος master. Σε αυτή την δομή υπάρχουν όλες οι πληροφορίες που χρειάζεται το τμήμα ORT για να δρομολογήσει τις οντοτήτες που του παρέχει το τμήμα EELIB.

3.3.1 Διεπαφή με το EELIB

Όπως προαναφέρθηκε, τα δύο τμήματα του συστήματος runtime του OMPi, είναι ανεξάρτητα μεταξύ τους, αλλά έχουν στενή επικοινωνία για την ανταλλαγή πληροφοριών. Το τμήμα ORT δεν γνωρίζει τον τύπο και το είδος των υπολογιστικών οντοτήτων αλλά μπορεί να τις δρομολογεί και να τις χειρίζεται μέσω των συναρτήσεων που προσφέρει το EELIB. Αυτές οι συναρτήσεις αποτελούν την διεπαφή μεταξύ του τμήματος ORT και του τμήματος EELIB. Στο Σχήμα 3.3 φαίνεται πιο καθαρά η διεπαφή αυτών των δύο τμημάτων.

Η πρώτη διεπαφή με το τμήμα EELIB εμφανίζεται στην έναρξη εκτέλεσης του προγράμματος που καλείται η αντίστοιχη συνάρτηση αρχικοποίησης του κάθε τμήματος. Το EELIB ενημερώνει το ORT για τις δυνατότητες του, που αφορούν τον πολυεπίπεδο προγραμματισμό την δυνατότητα δυναμικού ορισμού των οντοτήτων αλλά και τον αριθμό αυτών που του παρέχει. Επίσης το EELIB παρέχει και άλλες τρείς συναρτήσεις(Σχήμα 3.3) για την διεπαφή του με το ORT:

- *ee_request()* : Με αυτήν την κλήση το ORT ζητάει ένα συγκεκριμένο αριθμό υπολογιστικών οντοτήτων από το EELIB. Το τελευταίο με την σειρά του, απαντάει με τον αριθμό που μπορεί να παρέχει στο ORT ο οποίος εξαρτάται από κάποιες παραμέτρους.
- *ee_create()* : Με αυτή την κλήση, το ORT ζητά από το EELIB να δημιουργήσει τον αριθμό των οντοτήτων που τελικά μπορεί και παρέχει το EELIB. Στις παραμέτρους



Σχήμα 3.3: Διεπαφή μεταξύ του τμήματος ORT και του τμήματος EELIB

της συνάρτησης περιλαμβάνονται οι απαραίτητες πληροφορίες όπως, την συνάρτηση που θα εκτελέσουν, το επίπεδο παραλληλίας που βρίσκονται καθώς και πληροφορίες για τον πατέρα της ομάδας.

- *ee_waitall()* : Η συνάρτηση αυτή, καλεί το αρχικό νήμα όταν ολοκληρώσει τον κώδικά του να συγχρονιστεί με όλες τις άλλες οντότητες της ομάδας,δηλαδή να περιμένει να τελειώσουν την δουλειά τους.

Το τμήμα EELIB του OMPi αποτελείται από έναν αριθμό διαφορετικών βιβλιοθηκών που προσφέρουν ποικίλες υπολογιστικές οντότητες. Ο OMPi δημιουργεί εκ των προτέρων N νήματα (ο αριθμός N καθορίζεται από τη μεταβλητή περιβάλλοντος, αλλίως ισούται με το πλήθος των επεξεργαστικών πυρήνων), τα οποία περιμένουν μέχρι να τους ανατεθεί δουλειά. Όταν το ORT ζητήσει έναν αριθμό νημάτων, το EELIB θα διαβάσει τον αριθμό των νημάτων που περιμένουν, ώστε να γίνει έλεγχος άν υπάρχει ο ζητούμενος αριθμός από αυτά, και στην συνέχεια τα αναθέτει στο ORT. Αν βρισκόμαστε σε βαθύτερο του πρώτου επιπέδου παραλληλισμού, το αν θα δοθεί ο απαιτούμενος αριθμός νημάτων εξαρτάται και από τις αντίστοιχες μεταβλητές περιβάλλοντος αλλά και από την πολιτική με την οποία έχει δομηθεί η βιβλιοθήκη. Πιο συγκεκριμένα ο OMPi προσφέρει 4 βιβλιοθήκες νημάτων πυρήνα και 1 νημάτων χρήστη όπως φαίνεται στο Σχήμα 3.3. Υπάρχουν δύο βιβλιοθήκες με νήματα POSIX που καλούνται PTHREADS. Υπάρχει διαφορά μεταξύ των PTHREADS και PTHREADS1 στην υποστήριξη πολυεπίπεδου παραλληλισμού γιατί η PTHREADS παρέχει περιορισμένη υποστήριξη παραλληλίας, ενώ η PTHREADS1 σε επίπεδα παραλληλίας μεγαλύτερα του 1, ο κώδικας εκτελείται σειριακά.

Εκτός από τα νήματα POSIX, ο OMPi παρέχει και νήματα Solaris. Και εδώ υπάρχουν δύο βιβλιοθήκες με τις ίδιες ιδιότητες όπως και στα POSIX. Για τις περιπτώσεις που απαι-

τείται αυστηρά πολυεπίπεδος παραλληλισμός, ο OMPi παρέχει μια βιβλιοθήκη με νήματα επιπέδου χρήστη (PSTHREADS) [5] η οποία επιτυγχάνει ιδιαίτερα καλές επιδόσεις. Τα νήματα χρήστη τρέχουν στους δικούς τους εικονικούς επεξεργαστές και η δρομολόγησή τους γίνεται από τη βιβλιοθήκη πάνω στους φυσικούς επεξεργαστές. Λόγω της διεπαφής που υπάρχει ανάμεσα στα δύο αυτά τμήματα του συστήματος runtime του OMPi αλλά και της ανεξαρτησίας μεταξύ τους, είναι εύκολο από κάποιον προγραμματιστή να δημιουργήσει και να προσαρτήσει στον OMPi τη δικιά του βιβλιοθήκη οντοτήτων.

3.3.2 Είσοδος σε παράλληλη περιοχή

Η οδηγία parallel του προτύπου OpenMP, αντικαθίσταται με την κλήση συστήματος runtime, ort_execute_parallel(). Με βάση τις παραμέτρους της συνάρτησης αυτής, ορίζεται ο αριθμός τον νημάτων που θα ξυπνήσουν και θα εκτελέσουν την παράλληλη περιοχή, ποιά συνάρτηση θα εκτελεστεί και πληροφορίες σχετικά με τις μεταβλητές που θα χρησιμοποιηθούν. Όλα τα παραπάνω, πρέπει να υλοποιηθούν με κλήσεις προς το τμήμα του EELIB. Πιο αναλύτικά, το τμήμα ORT αρχικά επικοινωνεί με το τμήμα EELIB και ζητά τον αριθμό των οντοτήτων που χρειάζεται. Τότε το EELIB, ξυπνάει τον αριθμό οντοτήτων που του ζητήθηκε, για να αποτελέσουν την ομάδα που θα εκτελέσει την παραλληλη περιοχή, αλλα χωρίς να γνωρίζει το είδος τους. Με την σειρά τους οι οντότητες αυτές θα πρέπει να πάρουν την δουλειά που πρέπει να εκτελέσουν. Για αυτό τον λόγο καλει την συνάρτηση ort_get_thread_work() ώστε να λάβουν την δουλειά από το τμήμα ORT, το οποίο παρέχει όλες τις απαραίτητες πληροφορίες για την δουλειά που πρέπει να εκτελέσουν, μέσω μίας δομής.

Οι πληροφορίες παρέχονται μέσα από την δομή του *eecb*. Είναι μια δομή ελέγχου πατέρανήματος της ομάδας που έχει δημιουργηθεί. Μέσα στη δομή eecb, δεσμεύονται πληροφορίες όπως, το αναγνωριστικό της οντότητας στην ομάδα, το επίπεδο παραλληλισμού, τον δείκτη προς το block του πατέρα, τον δείκτη προς την συνάρτηση που θα εκτελέσουν, το μέγεθος της ομάδας καθώς και μεταβλητές που εκφράζουν την κατάσταση της ακύρωσης της ομάδας. Για τον έλεγχο της κατάστασης της ακύρωσης μιας ομάδας υπάρχουν flags, που αναφέρονται στην οδηγία (parallel, for, taskgroup, sections) που υπόκειται σε ακύρωση. Το σύνολο αυτών των πληροφοριών, είναι απαραίτητο ώστε να ξεκινήσει η παράλληλη εκτέλεση από τις οντότητες.

Κεφάλαιο 4

Ακύρωση (Cancellation) στο OpenMP v.4.0

Το OpenMP εισήγαγε για πρώτη φορά την δυνατότητα να ακυρώνει ο προτγραμματιστής νήματα ή εργασίες που ήδη έχουν ξεκινήσει και εκτελούνται. Μέχρι και την έκδοση του OpenMP v.3.1 μία παράλληλη περιοχή ή μία περιοχή διαμοιρασμού των εργασιών δεν μπορούσε να ακυρώσει την λειτουργία της. Θα έπρεπε να εκτελεστεί πάντα μέχρι το τέλος, ακόμα και αν δεν χρειαζόταν. Εκεί ακριβώς βασίζεται η σύλληψη της ιδέας για την ακύρωση της τρέχουσας περιοχής. Άν έπρεπε να εκτελεστεί μία αναζήτηση σε ένα δένδρο ή σε μία βάση δεδομένων για την εύρεση ενός ζητούμενου, θα έπρεπε να συνεχίστεί η παράλληλη εκτέλεση ακόμα και άν είχε πραγματοποιηθεί η εύρεση του ζητούμενου. Έτσι λοιπόν, στην προσπάθεια να μειωθεί ακόμα περισσότερο ο χρόνος εκτέλεσης μιας τέτοιας εφαρμογής, ορίστηκε η οδηγία της ακύρωσης (cancel), της περιοχής όταν αυτο ζητηθεί. Η δυνατότητα αυτή θεωρείται ιδιαίτερα σημαντική καθώς μπορεί να αυξήσει κατακόρυφα τις επιδόσεις των εφαρμογών που αναφέρθηκαν. Συνολικά ορίζει 5 νέες οδηγίες προκειμένου να ακυρωθεί η λειτουργία νημάτων, δομών διαμοιρασμού εργασίας (worksharing) και εργασιών (tasks).

4.1 Cancellation σε ομάδες νημάτων

Η οδηγία cancel του OpenMP v.4.0 είναι μία αυτόνομη οδηγία και ενεργοποιεί την ακύρωση της περικλειόμενης περιοχής (*innermost enclosing region*) από τον τύπο που έχει οριστεί. Το Σχήμα 4.1 διευκρινίζει τον όρο *innermost enclosing region*.

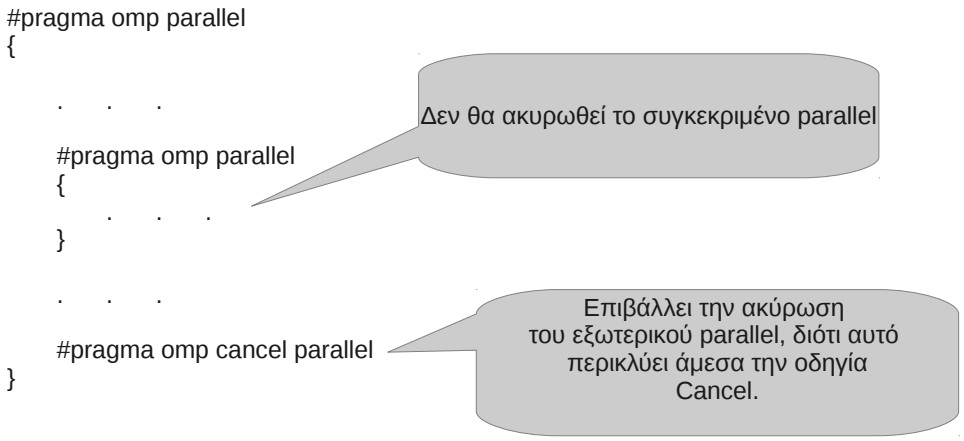
4.1.1 Σύνταξη

Η σύνταξη του cancel construct, στο OpenMP είναι η εξής:

 # pragma omp cancel *construct-type-clause[[,] if-clause] new-line*

όπου *construct-type-clause* είναι ένα από τα παρακάτω:

- **parallel**
- **sections**
- **for**
- **taskgroup**



Σχήμα 4.1: Innermost enclosing region

και *if-clause* είναι:

- **if** (*scalar-expression*)

4.1.2 Περιγραφή

Το σύνολο των νημάτων το οποίο επηρεάζει η οδηγία cancel είναι η τρέχουσα ομάδα. Το cancel ενεργοποιείται μόνο για την περικλειόμενη οδηγία (τυπου: parallel, sections, for, taskgroup) η οποία έχει οριστεί από την εντολή του cancel.

Το cancel ενεργοποιείται μόνο όταν η μεταβλητή περιβάλλοντος OMP_CANCELLATION (*cancel-var*) είναι ενεργοποιημένη (true). Σε αυτή την περίπτωση, το νήμα που ενεργοποίησε την ακύρωση της αντίστοιχης περιοχής, συνεχίζει την εκτελεσή του, ακριβώς μετά το τέλος της συγκεκριμένης οδηγίας. Σε άλλη περίπτωση, δηλαδή το *cancel-var* είναι false, τότε η οδηγία cancel αγνοείται.

Τα νήματα ελέγχουν για ενεργό cancel *μόνο* στα παρακάτω σημεία:

- *implicit barriers* : Είναι φράγματα (οδηγίες συγχρονισμού) τα οποία υπονοούνται στο τέλος μιας περιοχής διαμοιρασμού εργασίας, έτσι ώστε να συχρονιστούν οι υπολογιστικές οντότητες που εκτελούν την παράλληλη περιοχή.

- *barrier regions* : Είναι μία οδηγία συγχρονισμού που εισάγει ο χρήστης, ώστε να δηλώσει ρητά την χρήση συχρονισμού στην συγκεκριμένη περιοχή.

- *cancel regions* : Είναι η περιοχές στις οποίες συναντάμε τις εντολές ακύρωσης.

- *cancellation point regions* : Είναι οδηγίες που εισάγει ο χρήστης, μόνο για να ελέγχει την κατάσταση της ακύρωσης στο περικλειόμενο block της οδηγίας, και τις οποίες θα δούμε στην Ενότητα 4.2.

Άν ένα νήμα συναντήσει κάποιο από τα παραπάνω σημεία ακύρωσης και το *cancel-var* είναι true, τότε το νήμα ελέγχει άν το cancel είναι ενεργοποιημένο. Αν το cancel είναι ενεργό, τότε το νήμα συνεχίζει την εκτέλεσή του μετά το τέλος της ακυρωμένης περιοχής.

Όταν ο όρος if υπάρχει σε ένα cancel contruct, τότε ελέγχουμε πρώτα το if. Άν είναι false, τότε δεν ενεργοποιείται το cancel.

4.1.3 Περιορισμοί

Οι περιορισμοί για το cancel είναι οι εξής:

- Η συμπεριφορά, μιας ταυτόχρονης ακύρωσης μιας περιοχής και της εμφωλευμένης, σε αυτή, περιοχή, είναι ακαθόριστη.
- Το cancel construct πρέπει αυστηρά να περιλαμβάνεται λεκτικά μέσα σε περιοχή τύπου *construct-type-clause*.
- Ένα workshare construct που υπόκειται σε ακύρωση, δεν πρέπει να έχει την φράση nowait.
- Σε ένα βρόγχο που είναι υπόκειται σε ακύρωση, δέν πρέπει να υπάρχει ο όρος ordered.
- Ένα construct που υπόκειται σε ακύρωση, δεν πρέπει να συναντήσει ένα orphaned cancellation point. Orphaned είναι μία οδηγία που θα έπρεπε ρητά να ανήκει μέσα σε μία παράλληλη περιοχή, ενώ αυτή βρίσκεται έξω απο αυτήν (π.χ. σε μία συνάρτηση που κλήθηκε μέσα απο την περιοχή). Επομένως, ένα cancellation point πρέπει αυστηρά να περιέχεται λεκτικά μέσα στο μπλόκ κώδικα του συγκεκριμένου construct.

Στο παράδειγμα 4.2 διευκρινίζεται η λειτουργία του cancel όταν ακυρώνει την εκτέλεση των παράλληλων περιοχών. Όταν σε μία επανάληψη έχουμε exception ακυρώνουμε την περιοχη με την οδηγία for (γραμμή 15), αφού έχουμε κρατήσει την πληροφορία για το exception ώστε να το διαχειριστεί κατάλληλα αργότερα το master νήμα. Άν υπάρχει κάποιο exception τότε ακυρώνουμε και την περιοχή του parallel (γραμμή 20). Το barrier είναι ένα cancellation point. Επομένως η συνάρτηση phase_1() μπορεί να έχει εκτελεστεί κάμποσες φορές, αλλά η συνάρτηση phase_2() σίγουρα δεν θα εκτελεστεί ποτέ, άν έχουμε exception. Εναλλακτικά θα μπορουσαμε να εισάγουμε στην γραμμή 23 την εντολή, *#pragma omp cancellation point parallel*, για να ελέγχουμε το cancel στο parallel.

4.2 Οδηγία σημείου ακύρωσης (Cancellation Point)

4.2.1 Εισαγωγή

Όπως είπαμε τα νήματα ελέγχουν για πιθανή ενεργοποίηση του cancel σε περιοχές barrier, στην περιοχή ενεργοποίησης του cancel και στα σημεία ακύρωσης (cancellation points). Το σημείο ακύρωσης (cancellation point) είναι μία οδηγία που εισάγει ο χρήστης, για να ελέγξει ρητά άν έχει ενεργοποιηθεί η ακύρωση της περικλειόμενης περιοχής από την οδηγία που έχει οριστεί. Η οδηγία του σημείου ακύρωσης (cancellation point) είναι μία αυτόνομη εντολή.

```
1 void example() {
2   std::exception *ex = NULL;
3   #pragma omp parallel shared(ex)
4   {
5    #pragma omp for
6    for (int i = 0; i < N; i++) {
7      try {
8      causes_an_exception();
9      }
10     catch (const std::exception *e) {
11    // still must remember exception for later handling
12    #pragma omp atomic write
13      ex = e;
14    // cancel worksharing construct
15    #pragma omp cancel for
16     }
17   }
18 // if an exception has been raised, cancel parallel region
19   if (ex) {
20   #pragma omp cancel parallel
21    }
22    phase_1();
23    #pragma omp barrier
24    phase_2();
25   }
26   /* continue here if an exception
27    * has been thrown in the worksharing loop*/
28   if (ex) {
29   // handle exception stored in ex
30   }
31 }
```

Κώδικας 4.2: Παράδειγμα για την ερμηνεία του CANCEL

4.2.2 Σύνταξη

Η σύνταξη του cancellation point, στο OpenMP είναι η εξής:

 # pragma omp cancellation point *construct-type-clause new-line*

όπου *construct-type-clause* είναι ένα από τα παρακάτω:

- **parallel**
- **sections**
- **for**
- **taskgroup**

4.2.3 Περιγραφή

Η λειτουργία του cancellation point είναι να ελέγχει εάν έχει γίνει αίτημα ακύρωσης της περικλειόμενης περιοχής από μία οδηγία που έχει προσδιοριστεί από τον χρήστη. Αν όρθα έχει ανιχνευτεί η ακύρωση της οδηγίας αυτής, τότε πραγματοποιεί την διαδικασία της ακύρωσης. Δηλαδή όλα τα νήματα συνεχίζουν την εκτέλεση τους μετά το τέλος της ακυρωμένης περιοχής.

Είναι σημαντικό να αναφέρουμε, ότι σε ένα σημείο ακύρωσης το μόνο που πραγματοποιείται είναι ο έλεγχος για ενεργή ακύρωση της περικλειόμενης περιοχής της οδηγίας που έχει οριστεί, και η διαδικασία οδήγησης των νημάτων στο τέλος αυτής της περιοχής, σε περίπτωση που είναι ενεργοποιημένη η ακύρωση. Δεν υλοποιείται κάποιος αλγόριθμος συγχρονισμού των νημάτων κατά την έξοδο από την ακυρωμένη περιοχή.

Πιό αναλυτικά η λειτουργία του ελέγχου υλοποιεί τα παρακάτω βήματα. Όταν ένα νήμα συναντήσει ένα σημείο ακύρωσης με την μεταβλητή περιβάλλοντος (*cancel-var*) να είναι αληθή, τότε ελέγχει άν έχει ενεργοποιηθεί η ακύρωση της αντίστοιχης περιοχής που έχει ορίσει ο χρήστης. Στην περίπτωση που έχει ενεργοποιηθεί, το νήμα που συνάντησε το σημείο ακύρωσης (cancellation point) συνεχίζει την εκτέλεσή της, στο τέλος της ακυρωμένης δομής. Σε αντίθετη περίπτωση, η εντολή αγνοείται.

4.2.4 Περιορισμοί

- Η δομή ακύρωσης περιοχής και ο όρος construct type clause που έχει ορίσει ο χρήστης, πρέπει οπωσδήποτε να συμπεριλαμβάνεται μέσα σε μία περιοχή του συγκεκριμένου όρου. Στην περίπτωση που ο όρος είναι **taskgroup** τότε το σημείο ακύρωσης πρέπει να βρίσκεται *αυστηρά μέσα* σε μία εργασία (task), όπως θα δούμε στην επόμενη ενότητα .

- Ένα πρόγραμμα σε OpenMP με μία εντολή cancellation point που είναι ορφανή, δέν είναι ορθό.

4.3 Cancellation σε εργασίες

4.3.1 Εργασίες-Tasks

Για να δημιουργήσουμε μια εργασία στο πρότυπο OpenMP, θα πρέπει να χρησιμοποιήοουμε την παρακάτω εντολή

pragma omp task *[clause[[,] clause] ...] new-line*
structured-block

όπου *clause* είναι ένα απο:

- **if** (*scalar-expression*)
- **final** (*scalar-expression*)
- **untied**
- **default (shared | none)**
- **mergeable**
- **private** (*list*)
- **firstprivate** (*list*)
- **shared** (*list*)
- **depend** (*dependence type : list*)

Όταν ένα νήμα συναντήσει την παραπάνω εντολή, τότε δημιουργείται μία εργασία προς εκτέλεση με το συγκεκριμένο δομημένο κομμάτι κώδικα. Οι εργασίες έχουν δικό τους ξεχωριστό χώρο διευθύνσεων για κάθε μια, και το περιβάλλον των δεδομένων στο οποίο έχουν πρόσβαση, εξαρτάται απο:

- το όρο που ακολουθεί το *#pragma omp task*, και αφορά τον διαμοιρασμό της μνήμης.
- τόσο τις τιμές των μεταβλητών περιβάλλοντος, όσο και τις προκαθορισμένες τιμές του runtime.

Το νήμα που θα συναντήσει την εντολή #pragma omp task δημιουργεί την εργασία και μπορεί να την εκτελέσει κατευθείαν, αλλά μπορεί να την δημιουργήσει μόνο, και να την εκτελέσει κάποιο άλλο νήμα που ανήκει στην ομάδα, και την συγκεκριμένη στιγμή δεν έκτελεί καποια άλλη εργασία. Επίσης μία εργασία μπορεί να είναι εμφωλευμένη μέσα σε μία άλλη. 'Ομως, η εσωτερική εργασία δεν αποτελεί μέρος της εξωτερικής.

4.3.2 Taskgroup

Στο OpenMP v.4.0 έχει προστεθεί ένας νέος όρος στην διαχείριση των εργασιών tasks και ονομάζεται taskgroup. Η οδηγία taskgroup δηλώνει μία ομάδα εργασιών, η οποία, για την ολοκλήρωσή της, πρέπει να περιμένει την ολοκλήρωση της εκτέλεσης όλων των εργασιών που δημιουργήθηκαν μέσα στην ομάδα. Η σύνταξη της οδηγίας taskgroup, έχει την παρακάτω σύνταξη:

pragma omp taskgroup *new-line*
structured-block

Όταν ένα νήμα συναντήσει την οδηγία taskgroup, αρχίζει να εκτελεί την δεδομένη περιοχή κώδικα. Πάντα στο τέλος ενός taskgroup υπονοείται συγχρονισμός των εργασιών που ανήκουν στην ομάδα. Στο συγκεκριμένο σημείο, η τρέχουσα εργασία, περιμένει να ολοκληρωθούν όλες οι εργασίες που δημιούργησε, καθώς και όλες οι εργασίες που δημιουργήθηκαν από τις εργασίες των απογόνων της.

Όταν ενεργοποιηθεί το cancel σε tasks, μέσω του cancel taskgroup, το περικλείον taskgroup θα πρέπει να ακυρωθεί. Το task που συνάντησε το cancel taskgroup θα πρέπει να συνεχίσει την εκτέλεση μετά το τέλος του περιοχής της εργασίας όπου και συνεπάγεται το τέλος αυτού του task. Κάθε άλλο task, που συμμετέχει στο συγκεκριμένο taskgroup, και έχει ήδη ξεκινήσει την εκτέλεση μιάς εργασίας, πρέπει να ολοκληρώσει την εκτελεσή του, μέχρι να συναντήσει cancellation point. Όταν ένα νήμα συναντήσει ένα cancellation point και το cancel-var είναι true, τότε το νήμα συνεχίζει την εκτελεσή του στο τέλος της περιοχής του taskgroup. Ένα task του οποίου η εκτέλεση δεν έχει ξεκινήσει, μπορεί να απορριφθεί.

- Όταν έχουμε taskgroup το cancel θα πρέπει να είναι λεκτικά μέσα σε ένα task.
- Άν έχουμε taskgroup και το cancel δεν είναι λεκτικά μέσα στο taskgroup, τότε η συμπεριφορά του προγράμματος δεν μπορεί να προσδιοριστεί.

Το Παράδειγμα 5.17 δείχνει πώς μπορεί να γίνει η ακύρωση σε μία παράλληλη αναζήτηση σε δυαδικό δέντρο όταν η τιμή που αναζητείται έχει βρεθεί. Ο κώδικας δημιουργεί μία εργασία για να κατέβει στα παιδιά του τρέχοντος κόμβου του δέντρου (γραμμές 8,18). Άν η τιμή που ψάχνουμε έχει βρεθεί, αποθηκεύουμε τον κόμβο στον οποίο βρέθηκε σε ένα δείκτη και έπειτα, σταματάμε όλες τις εργασίες αναζήτησης (γραμμές 15,25). Η συνάρτηση search_tree_parallel() ομαδοποιεί όλες τις εργασίες αναζήτησης σε μία ομάδα αναζήτησης έτσι ώστε να μπορεί να έχει αποτέλεσμα η οδηγία της ακύρωσης.

```
1 binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
2   binary_tree_t *found = NULL;
3   if (tree) {
4   if (tree->value == value) {
5   found = tree;
6   }
7   else {
8     #pragma omp task shared(found) if(level < 10)
9     {
10    binary_tree_t *found_left = NULL;
11    found_left = search_tree(tree->left, value, level + 1);
12    if (found_left) {
13      #pragma omp atomic write
14      found = found_left;
15      #pragma omp cancel taskgroup
16    }
17    }
18    #pragma omp task shared(found) if(level < 10)
19    {
20    binary_tree_t *found_right = NULL;
21    found_right = search_tree(tree->right, value, level + 1);
22    if (found_right) {
23      #pragma omp atomic write
24      found = found_right;
25      #pragma omp cancel taskgroup
26    }
27    }
28    #pragma omp taskwait
29  }
30  }
31  return found;
32 }
33 binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
34   binary_tree_t *found = NULL;
35   #pragma omp parallel shared(found, tree, value)
36   {
37   #pragma omp master
38   {
39     #pragma omp taskgroup
40     {
41     found = search_tree(tree, value, 0);
42     }
43   }
44   }
45   return found;
46 }
```

Κώδικας 4.3: Παράδειγμα για την λειτουργία του CANCEL σε taskgroup

Κεφάλαιο 5

Υλοποίηση στον μεταφραστή OMPi

Στο κεφάλαιο αυτό περιγράφουμε τις λεπτομέρειες της υλοποίησης που έγιναν ώστε να υπάρχει πλήρη υποστήριξη της οδηγίας cancel, στον παραλληλοποιητικό μεταφραστή OMPi. Η υλοποίηση καλύπτει όλες τις δομές ακύρωσης που περιγράψαμε στο Κεφάλαιο 4 και είναι πλήρως σύμφωνη με τις προδιαγραφές του προτύπου OpenMP 4.0. Η υλοποίηση στον OMPi για την ακύρωση περιγράφεται παρακάτω σε 4 ενότητες, για κάθε οδηγία για την οποία και επηρεάζει.

5.1 Κεντρική Ιδέα για την Υλοποίηση του CANCEL

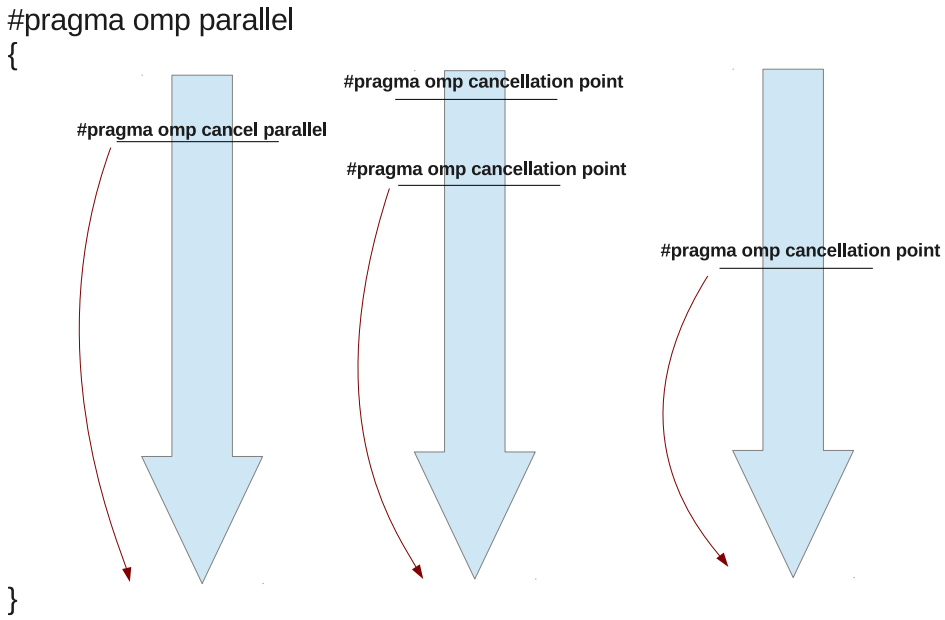
Στην δομή κάθε νήματος(EECB - Execution Entity Control Block), έχουν εισαχθεί νέες μεταβλητές (flags) που αναφέρονται στην ενεργοποίηση ή απενεργοποίηση του cancel. Πιο συγκεκριμένα, προστέθηκαν τα εξής flags:

- *cancel_parallel* : για έλεγχο της κατάστασης της ακύρωσης μίας παράλληλης περιοχής.
- *cancel_sections* : για έλεγχο της κατάστασης της ακύρωσης μίας παράλληλης περιοχής στην δομή sections. / *cancel_sections_me* : για έλεγχο της κατάστασης της ακύρωσης μίας περιοχής στην δομή sections όταν το πρόγραμμα εκτελείται σειριακά.
- *cancel_for* : για έλεγχο της κατάστασης της ακύρωσης της επαναληπτικής δομής for που εκτελείται παράλληλα./ *cancel_for_me* : για έλεγχο της κατάστασης της ακύρωσης της επαναληπτικής δομής for που εκτελείται σειριακά.

Για το taskgroup έχει υλοποιηθεί μια ξεχωριστή δομή η οποία δημιουργείται και αρχικοποιείται όταν συναντήσουμε την εντολή #pragma omp taskgroup.

Σε μία παράλληλη περιοχή όταν υπάρχει η εντολή **#pragma omp cancel** *construct-type-clause* τότε, το νήμα που συνάντησε το cancel (εφόσον η μεταβλητή περιβάλλοντος OMP_CANCELLATION είναι ενεργοποιημένη), ενεργοποιεί το αντίστοιχο πεδίο cancel το οποίο βρίσκεται αποθηκευμένο στο master-thread, και συνεχίζει την εκτέλεση του στο τέλος της ακυρωμένης περιοχής. Όλα τα υπόλοιπα νήματα που ανήκουν στην ομάδα, ελέγχουν για ενεργοποιήση του cancel, στο πεδίο του master-thread, και σε περίπτωση που συναντήσουν ένα από τα cancellation points, συνεχίζουν την εκτέλεση τους στο τέλος της ακυρωμένης περιοχής. Στην περίπτωση που έχουμε σειριακή εκτέλεση, το νήμα κάνει ακριβώς την ίδια διαδικασία, μόνο που ενημερώνει τα αντίστοιχα σειριακά πεδία του cancel, γιατί στην συγκεκριμένη περίπτωση δεν υπάρχει master-thread. Η παραπάνω διαδικασία φαίνεται και

στο Σχήμα 5.1. Όταν συναντήσουμε ένα cancellation point και το cancel δεν έχει ενεργοποιηθεί, τότε αγνοείται. Την στιγμή που το cancel θα ενεργοποιηθεί, μόνο τότε τα cancellation points θα οδηγήσουν όλα τα νήματα στο τέλος της περικλειόμενης περιοχής.



Σχήμα 5.1: Περιγραφή του cancel

Οι δύο κύριες συναρτήσεις για την ενεργοποίηση και απενεργοποίηση της ακύρωσης μιας περιοχής, βρίσκονται στο runtime και είναι οι εξής:

- int **ort_enable_cancel**(*type*)
- int **ort_check_cancel**(*type*)

όπου *type* είναι ένα από τα

- **parallel** με *type* = 0
- **taskgroup** με *type* = 1
- **for** με *type* = 2
- **sections** με *type* = 3

Πιο αναλυτικά θα αναφερθώ παρακάτω, σε μια ξεχωριστή ενότητα για το καθένα, από τα παραπάνω.

```
1 main()
2 {
3   #pragma omp parallel num_threads(6)
4   {
5     #pragma omp barrier
6     printf("After barrier,thread %d\n",omp_get_thread_num());
7     #pragma omp cancel parallel
8     printf("After cancellation,thread %d\n",omp_get_thread_num());
9     #pragma omp cancellation point parallel
10    printf("After cancellation point,thread %d\n",omp_get_thread_num());
11    }
12 }
```

Κώδικας 5.2: Απλό παράδειγμα εφαρμογής του cancel σε μία οδηγία parallel.

5.2 Cancellation σε parallel construct

Όταν συναντήσουμε #pragma omp cancel parallel (αυστηρά μέσα σε ένα parallel construct) τότε καλείται η συνάρτηση ort_enable_cancel(0). Επειδή η ενεργοποιήση του cancel, είναι ένα cancellation point, η ort_enable_cancel(0) , επιστρέφει 1 άν έχει ενεργοποιηθεί το cancel (δεδομένου ότι η μεταβλητή περιβάλλοντος *OMP_CANCELLATION* είναι ενεργοποιημένη) και 0 σε διαφορετική περίπτωση. Τα νήματα όταν συναντήσουν ένα cancellation point, αρχικά ελέγχουν άν έχει γίνει cancel καλώντας την ort_check_cancel(0). Μέσα στην συνάρτηση γίνεται έλεγχος στο πεδίο του πατέρα (cancel_parallel), για το αν έχει ενεργοποιηθεί το cancel . Η συνάρτηση επιστρέφει 1 άν έχει ενεργοποιηθεί και 0 σε διαφορετική περίπτωση.

Ο Κώδικας 5.2 είναι ένα παράδειγμα που θα χρησιμοποιήσουμε για την επεξήγηση της διαδικασίας. Στην γραμμή 7 γίνεται ενεργοποίηση του cancel, επομένως όλα τα νήματα που ανήκουν στην παράλληλη ομάδα θα πρέπει να οδηγηθούν στο τέλος αυτής (γραμμή 11). Υποθέτοντας ότι η μεταβλητή περιβάλλοντος OMP_CANCELLATION είναι true, η έξοδος θα είναι:

΄Εξοδος:

```
After barrier,thread 0
After barrier,thread 1
After barrier,thread 2
After barrier,thread 3
After barrier,thread 4
After barrier,thread 5
```

Για την υποστήριξη της οδηγίας cancel έπρεπε, εκτός από την εισαγωγή νέων συναρτήσεων στο runtime, να πραγματοποιηθούν και αλλαγές στον κώδικα που παράγεται από τον source-to-source μετασχηματιστή του OMPi.

Οι μεταβολές στον παραγώμενο κώδικα του μεταφραστή OMPi, έτσι ώστε να υποστηρίζεται η υλοποίηση του cancellation παραθέτονται παρακάτω. Παρακάτω φαίνεται ο παραγόμενος κώδικας για το Παράδειγμα 5.2, χωρίς την υλοποίηση για το cancel(Κώδικας 5.3), και στην συνέχεια ο παραγόμενος κώδικας που υποστηρίζει την προαναφερθείσα επέκταση (Κώδικας 5.4).

Όπως αναφέρθηκε και στην Ενότητα 3.2 η οδηγία parallel οδηγεί στην δημιουργία μιάς συνάρτησης(γραμμή 1, _thrFunc0_) που θα εκτελέσουν όλα τα νήματα που ανήκουν στην ομάδα. Αντίστοιχα, η οδηγία barrier στην γραμμή 4 του Κώδικα 5.2 μετασχηματίζεται στην

κλήση ort_barrier_me()(γραμμή 6 του Κώδικα 5.3) στην οποία όλα τα νήματα της παράλληλης ομάδας, περιμένουν να συγχρονιστούν.

Ο Κώδικας 5.4 είναι ίδιος με τον Κώδικα 5.3 μόνο που πλέον υπάρχουν οι εξής μετατροπές:

- Η κλήση ort_barrier_me() επιστρέφει 1 όταν έχει ανιχνευτεί ενεργό cancel και 0 σε διαφορετική περίπτωση.
- Χρειάστηκε να μπεί νέα ετικέτα (CANCEL_PARALLEL8), ώστε να μπορούμε να μεταπηδήσουμε στο τέλος της περιοχής.
- Το **#pragma omp cancel parallel** μετασχηματίζεται στην κλήση **ort_enable_cancel(0)**.

```
1  static void * _thrFunc0_(void * __me)
2  {
3    /* (l8) #pragma omp parallel num_threads(10) -- body moved below */
4    {
5      /* #pragma omp barrier  */
6      ort_barrier_me();
7      printf("After barrier,thread %d\n",omp_get_thread_num());
8      printf("After cancellation,thread %d\n",omp_get_thread_num());
9      printf("After cancellation point,thread %d\n",omp_get_thread_num());
10   }
11   ort_taskwait(2);
12   return ((void *) 0);
13 }
```

Κώδικας 5.3: Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση του cancel σε parallel

5.3 Cancellation σε sections construct

Στην περίπτωση των sections, υλοποιήθηκε μια μέθοδος ανίχνευσης για cancel, κατά την διάρκεια ανάθεσης των case_id στα νήματα. Το case_id είναι ένας αναγνωριστικός αριθμός που αντιστοιχεί σε ένα section. Αυτόν τον αριθμό δέχονται τα νήματα που ανήκουν στην παράλληλη ομάδα και εκτελούν το section που αντιστοιχεί στο συγκεκριμένο αναγνωριστικό. Στο runtime υπάρχει ένας βρογχος που αναθέτει τα case_id στα νήματα, μέχρι να μην υπάρχει αλλο διαθέσιμο section πρός εκτέλεση. Όταν ανιχνευτεί cancel διακόπτουμε τη λειτουργία αυτού του βρόγχου, και δέν αναθέτονται άλλα section στα νήματα.

Ο Κώδικας 5.5, αφορά τον μηχανισμό επαναληπτικής ανάθεσης των case_id των section στα νήματα της ομάδας όπως βρίσκεται στη βιβλιοθήκη runtime του OMPi. Στις γραμμές 14-15 και 21-26 υλοποιείται ο μηχανισμός ανίχνευσης για cancellation.

Το παράδειγμα του Κώδικα 5.6 στην περίπτωση της σειριακής εκτέλεσης δίνει έξοδο:

```
In 1st section,thread 0
In 2nd section,thread 0
In 3rd section,thread 0
```

Ενώ στην περίπτωση της παράλληλης εκτέλεσης δίνει έξοδο:

```
In 5th section,thread 3
```

```
1 static void * _thrFunc0_(void * __arg)
2 {
3   /* byresult variables */
4   /* (l8) #pragma omp parallel num_threads(10) -- body moved below */
5   {
6     /* #pragma omp barrier  */
7     if (ort_barrier_me())
8       goto CANCEL_PARALLEL8;
9     printf("After barrier,thread %d\n",omp_get_thread_num());
10    /* #pragma omp cancel parallel */
11    if (ort_enable_cancel(0))
12      goto CANCEL_PARALLEL8;
13    printf("After cancellation,thread %d\n",omp_get_thread_num());
14    /* #pragma omp barrier  */
15    if (ort_barrier_me())
16      goto CANCEL_PARALLEL8;
17    /* #pragma omp cancellation point parallel */
18    if (ort_check_cancel(0))
19      goto CANCEL_PARALLEL8;
20    printf("After cancellation point,thread %d\n",omp_get_thread_num());
21  }
22  CANCEL_PARALLEL8 :
23    ort_taskwait(2);
24  return ((void *) 0);
25 }
```

Κώδικας 5.4: Παραγόμενος κώδικας του OMPi, με την υλοποίηση του cancel σε parallel

Στο Κώδικα 5.6 το νήμα που συνάντησε το #pragma omp cancel sections, καλεί την ort_enable_cancel(3), ενημερώνει το αντίστοιχο πεδίο του πατέρα για το cancel (cancel_sections) και στην συνέχεια, συνεχίζει την λειτουργία του στο τέλος του section και κατ' επέκταση στο τέλος του sections. Λόγω της έγκυρης ανίχνευσης για cancellation , δεν θα ανατεθούν αλλα sections προς εκτέλεση στα υπόλοιπα νήματα.

Στην περίπτωση της σειριακής εκτέλεσης, το νήμα που θα εκτελέσει όλα τα section, όταν συναντήσει #pragma omp cancel sections ενεργοποιεί το "δικό" του πεδίο (cancel_sections_me) σε 1 και οδηγείται κατευθείαν στο τέλος του sections. Αμέσως μετά την ανίχνευση του cancellation, το πεδίο cancel_sections_me αρχικοποιείται ξανά στο 0, για μελλοντική χρήση.

Παρακάτω φαίνονται οι αλλαγές στον παραγόμενο κώδικα 5.7 (χωρίς την υλοποίηση του cancel) και στον παραγόμενο κώδικα 5.8 (με την υλοποίηση του cancel).

Στις γραμμές 8-17 του Κώδικα 5.8 φαίνεται ο μηχανισμός ανάθεσης των caseid_ στα νήματα. Στην παράλληλη εκτέλεση, τα νήματα που ανήκουν στην παράλληλη ομάδα αναλαμβάνουν να εκτελέσουν κάποιο απο τα διαθέσιμα section. Αυτή η διαδικασία ανάθεσης (γραμμή 10) περιέχει και συνεχόμενη ανίχνευση για ενεργό cancel. Όταν ενεργοποιηθεί το cancel (γραμμή 32), το νήμα που το ενεργοποίησε ενημερώνει το αντίστοιχο πεδίο του cancel_sections και βγαίνει απο τον βρόχγχο. Τα υπόλοιπα νήματα που συνεχίζουν την εκτέλεση των διαθέσιμων sections όταν ανιχνεύσουν ενεργό cancel, σταματάνε την λειτουργία ανάθεσης των section και κατ' επέκταση την εκτέλεση των υπολοίπων section. Στην σειριακή εκτέλεση, το νήμα που εκτελεί τα sections, όταν συναντήσει την ενεργοποίηση του cancel τότε απευθείας οδηγείται στο τέλος της περιοχής των section.

```
1 int ort_get_section()
2 {
3   wsregion_t *r;
4   int        s;
5   ort_eecb_t *me = __MYCB;
6
7   s = me->mynextNWregion - 1;  /* My region ('cause it is 1 ahead) */
8   r = (me->nowaitregion) ?
9       &(me->parent->workshare.REGION(s)) :
10      &(me->parent->workshare.blocking);
11
12  if (r->sectionsleft < 0) return (-1);
13 #if defined(HAVE_ATOMIC_FAA) && !defined(EE_TYPE_PROCESS)
14  if (me->parent->cancel_sections == CANCEL_ACTIVATED)
15    s = -1;
16  else
17    s = _faa(&(r->sectionsleft), -1) - 1;
18 #else
19  /* Check for active cancellation in order to terminate
20   * case_id assignment in the current team     */
21  if (me->parent->cancel_sections == CANCEL_ACTIVATED)
22  {
23    ee_set_lock(&r->reglock);
24    s = -1;
25    ee_unset_lock(&r->reglock);
26  }
27  else
28  {
29    ee_set_lock(&r->reglock);
30    s = --(r->sectionsleft);
31    ee_unset_lock(&r->reglock);
32  }
33 #endif
34
35  return (s);
36 }
```

Κώδικας 5.5: Ο επαναληπτικός μηχανισμός ανάθεσης των section στα νήματα στη βιβλιοθήκη runtime του OMPi με την προσθήκη για έλεγχο cancellation.

5.4 Cancellation σε for construct

Μία αντίστοιχη τεχνική, δυναμικής ανάθεσης των επαναλήψεων στα νήματα, έχει υλοποιηθεί και στην περίπτωση της ακύρωσης ενός for construct. Ανάλογα με τον δοθέντα χρονοπρογραμματισμό (static | dynamic | guided | runtime), η υλοποίηση του cancel γίνεται είτε στο runtime είτε στον παραγόμενο κώδικα. Πιο συγκεκριμένα:

- **Schedule(static)**

  Όταν επιλεχθεί το static ως schedule, τότε τα νήματα παίρνουν εξ αρχής ένα συγκεκριμένο μέρος επαναλήψεων προς εκτέλεση. Επομένως, όταν ένα νήμα συναντήσει #pragma omp cancel for ενεργοποιεί το cancel καλώντας την συνάρτηση ort_enable_cancel(2). Έπειτα σταματάει την εκτέλεση των υπόλοιπων επαναλήψεων που έχει αναλάβει και οδηγείται αμέσως μετά το τέλος του βρόγχου.

  Με την τρέχουσα υλοποίηση τα άλλα νήματα που παραμένουν στο βρόγχο, αγνοούν

```
1 void main(){
2  #pragma omp (parallel) sections (num_threads(6))
3  {
4   #pragma omp section
5     printf("In 1st section,thread %d\n",omp_get_thread_num());
6   #pragma omp section
7     printf("In 2nd section,thread %d\n",omp_get_thread_num());
8   #pragma omp section
9     printf("In 3rd section,thread %d\n",omp_get_thread_num());
10  #pragma omp section
11  {
12    #pragma omp cancel sections
13    printf("In 4th section,thread %d\n",omp_get_thread_num());
14  }
15  #pragma omp section
16  printf("In 5th section,thread %d\n",omp_get_thread_num());
17 }
```

Κώδικας 5.6: Παράδειγμα χρήσης του cancel σε sections

την ενεργοποίηση του cancel στο for, και συνεχίζουν κανονικά την λειτουργία τους μέσα στον επαναληπτικό βρόγχο.

Στο Κώδικα 5.9, στην περίπτωση που το πρόγραμμα εκτελείται σειριακά, το j έχει σαν αποτέλεσμα την τιμή 10 καθώς όταν ενεργοποιηθεί το cancel αυτόματα βγαίνει εκτός βρόγχου. Όταν όμως ο κώδικας εκτελείται παράλληλα, τότε το j θα έχει την τιμή 25. Αυτό συμβαίνει γιατί στον στατικό χρονοπρογραμματισμό, κάθε νήμα λαμβάνει συγκεκριμένο πλήθος επαναλήψεων. Για το συγκεκριμένο παράδειγμα του Κώδικα 5.9, το κάθε νήμα αναλαμβάνει από 5 επαναλήψεις. Μόνο το νήμα που έχει αναλάβει τις επαναλήψεις 10-15, θα βγεί εκτός του βρόγχου(γιατί για i=10 ενεργοποιείται το cancel), με αποτέλεσμα η μεταβλητή j να μην προσαυξηθεί κατά 5.

Οι διαφορές που παρουσιάζονται στον παραγόμενο κώδικα, ώστε να επιτευχθεί η υλοποίηση για το cancellation στο for φαίνονται παρακάτω και είναι οι εξής:

- Χρειάστηκε να μπεί νέα ετικέτα (CANCEL_FOR16) για τον παραγόμενο κώδικα 5.11, ώστε να μπορούμε να μεταπηδήσουμε στο τέλος της περιοχής.
- Το **#pragma omp cancel for** μετασχηματίζεται στην κλήση **ort_enable_cancel(2)**.

Η ενεργοποίηση του cancel στο for γίνεται στην γραμμή 15 του Κώδικα 5.11. Όπως προαναφέρθηκε η ενεργοποίηση του cancel είναι και ένα σημειο ελέγχου ακύρωσης, επομένως θα πρέπει να ελέγχουμε άν κάποιο απο τα νήματα της ομάδας ενεργοποίησε το cancel_for και αυτη η διαδικασία γίνεται με την κλήση της ort_check_cancel(2) στην γραμμή 18. Στον static χρονοπρογραμματισμό δεν θα μας απασχολήσει αυτή η ιδιότητα, όσο στους dynamic και guided χρονοπρογραμματισμούς, γιατί τα νήματα που ανήκουν στην παράλληλη ομάδα εκτελούν το πλήθος επαναλήψεων που έχουν αναλάβει, αγνοώντας την ενεργοποίηση του cancel.

- **Schedule(dynamic)**

Στην περίπτωση που επιλεχθεί το dynamic schedule, η ανάθεση των επαναλήψεων στο βρόγχο πραγματοποιείται δυναμικά. Σε κάθε νήμα ανατίθεται ένα συγκεκριμένο πλήθος επαναλήψεων ίσο με το chunk_size που έχει οριστεί (η default τιμή του chunk_size

```
1     /* #pragma omp sections  */
2     int caseid_ = -1, inpar_;
3
4     if ((inpar_ = (omp_in_parallel() && omp_get_num_threads() > 1)) != 0)
5       ort_entering_sections(0, 5);
6     for (;;)
7     {
8       if (inpar_)
9         {
10          if ((caseid_ = ort_get_section()) < 0)
11            break;
12        }
13      else
14        {
15          if ((++caseid_) >= 5)
16            break;
17        }
18      switch (caseid_)
19      {
20        case 0 :
21          printf("In 1st section,thread %d\n", omp_get_thread_num());
22          break;
23        case 1 :
24          printf("In 2nd section,thread %d\n", omp_get_thread_num());
25          break;
26        case 2 :
27          printf("In 3rd section,thread %d\n", omp_get_thread_num());
28          break;
29        case 3 :
30          {
31            printf("In 4th section,thread %d\n", omp_get_thread_num());
32          }
33          break;
34        case 4 :
35          printf("In 5th section,thread %d\n", omp_get_thread_num());
36          break;
37      }
38    }
39    if (inpar_)
40      ort_leaving_sections();
41    ort_barrier_me();
```

Κώδικας 5.7: Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση του cancel σε sections

```
1     /* #pragma omp sections  */
2     int caseid_ = -1, inpar_;
3
4     if ((inpar_ = (omp_in_parallel() && omp_get_num_threads() > 1)) != 0)
5       ort_entering_sections(0, 5);
6     for (;;)
7     {
8       if (inpar_)
9         {
10          if ((caseid_ = ort_get_section()) < 0)
11            break;
12        }
13      else
14        {
15          if ((++caseid_) >= 5)
16            break;
17        }
18      switch (caseid_)
19      {
20        case 0 :
21          printf("In 1st section,thread %d\n", omp_get_thread_num());
22          break;
23        case 1 :
24          printf("In 2nd section,thread %d\n", omp_get_thread_num());
25          break;
26        case 2 :
27          printf("In 3rd section,thread %d\n", omp_get_thread_num());
28          break;
29        case 3 :
30        {
31            /* #pragma omp cancel sections */
32            if (ort_enable_cancel(3))
33              goto CANCEL_SECTIONS25;
34            printf("In 4th section,thread %d\n", omp_get_thread_num());
35          }
36          break;
37        case 4 :
38          printf("In 5th section,thread %d\n", omp_get_thread_num());
39          break;
40      }
41    }
42    CANCEL_SECTIONS25 :
43      if (inpar_)
44        ort_leaving_sections();
45    ort_barrier_me();
```

Κώδικας 5.8: Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο sections

```
1 main(){
2  int i,j=0;
3   #pragma omp (parallel) for schedule(static) (num_threads(6))
4   for(i=0;i<30;i++){
5     #pragma omp cancel for if(i==10)
6       j++;
7    }
8   printf("%d \n",j);
9 }
```

Κώδικας 5.9: Εφαρμογή του cancellation σε for

είναι 1). Όταν ένα νήμα τελειώσει τις επαναλήψεις που του έχουν ανατεθεί, τότε ανακτεί δυναμικά μία νέα ομάδα επαναλήψεων προς εκτέλεση.Όλη αυτή η διαδικασία επαναλαμβάνεται μέχρι να τελειώσουν όλες οι επαναλήψεις. Στον Κώδικα 5.12 περιγράφεται ο μηχανισμός ανάθεσης των επαναλήψεων στα νήματα της παράλληλης ομάδας, καθώς και ο μηχανισμός ανίχνευσης για ενεργό cancel σε for(γραμμές 8-9).

Για την υλοποίηση του cancel, χρειάστηκε να προστεθεί μια εντολή ελέγχου για cancel, μέσα στον βρόγχο που χρησιμοποιείται για την ανάθεση των επαναλήψεων σε κάθε νήμα. Το νήμα που θα συναντήσει το #pragma omp cancel for ενεργοποιεί το πεδίο cancel_for στον πατέρα του και συνεχίζει την εκτέλεση του εκτός του βρόγχου. Μέσα στον βρόγχο για τις αναθέσεις των επαναλήψεων, γίνεται συνεχής έλεγχος για πιθανή ενεργοποίηση του cancel_for του πατέρα. Όταν είναι ενεργό, τότε τερματίζεται ο βρόγχος ανάθεσης επαναλήψεων, και τα νήματα δέν παίρνουν άλλες επαναλήψεις.

Σημαντικό! - Τα υπόλοιπα νήματα θα σταματήσουν την εκτέλεση των επαναλήψεων, όταν κάποιο νήμα ενεργοποιήσει το cancel_for, και ανιχνευθεί έγκαιρα.

Στο Παράδειγμα 5.13 το j θα έχει τελική τιμή ίση με 10. Υπάρχει περίπτωση να μην ανιχνευθεί κατευθείαν το cancel, με αποτέλεσμα να προλάβουν να εκτελεστούν κάποιες επαναλήψεις παραπάνω. Αυτό συμβαίνει για δύο λόγους. Είτε το νήμα που συναντά το #pragma omp cancel for δεν προλαβαίνει να γράψει την τιμή στο πεδίο του πατέρα cancel_for, με αποτέλεσμα στα υπόλοιπα νήματα να ανατεθούν κι άλλες επαναλήψεις, είτε η συγκεκριμένη επανάληψη που περιέχει την εντολή ακύρωσης, να έχει προγραμματιστεί να εκτελεστεί εκτός σειράς.

Οι μεταβολές στον παραγώμενο κώδικα για την υλοποίηση του cancel στο for με dynamic schedulling φαίνονται στους Κώδικες 5.14, 5.15.

- **Schedule(guided)** Η υλοποίηση της οδηγίας cancel για το guided schedule έγινε με τρόπο παρόμοιο με αυτόν της υλοποίησης του dynamic schedule. Γίνεται παρέμβαση στη δυναμική ανάθεση των επαναλήψεων ώστε να ανιχνευτεί το cancel. Στον Κώδικα 5.16 παραθέτεται ο μηχανισμός ανάθεσης των επαναλήψεων στα νήματα και στις γραμμές 7-9 και 34-35 φαίνεται ο μηχανισμός ανίχνευσης του cancel. Η εφαρμογή

του cancel σε επαναληπτικό βρόγχο με Schedule(guided) καθώς και οι αλλαγές στον παραγώμενο κώδικα που εξάγει ο μεταφραστής OMPi, είναι ακριβώς ίδιοι, μόνο που στη συγκεκριμένη περίπτωση, καλείται η ort_get_guided_chunk().

```
1
2 /* #pragma omp for schedule(static ) */
3 int i;
4 struct _ort_gdopt_ gdopt_;
5 int niters_ = 0, iter_ = -1, fiter_, liter_ = -2;
6
7 ort_entering_for(0, 0, &gdopt_);
8 niters_ = ort_num_iters(1, (long) (30 - (0)), (long) 1, (int *) 0);
9 if (ort_get_static_default_chunk(niters_, &fiter_, &liter_))
10  {
11    for (iter_ = fiter_, i = 0 + fiter_ * 1;
12      iter_ < liter_; iter_++, i += 1)
13    {
14      j++;
15    }
16
17 }
18 ort_leaving_for();
19 ort_barrier_me();
20 }
```

Κώδικας 5.10: Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση για το cancel στο for

```
1 /* #pragma omp for schedule(static ) */
2 int i;
3 struct _ort_gdopt_ gdopt_;
4 int niters_ = 0, iter_ = -1, fiter_, liter_ = -2;
5
6 ort_entering_for(0, 0, &gdopt_);
7 niters_ = ort_num_iters(1, (long) (30 - (0)), (long) 1, (int *) 0);
8 if (ort_get_static_default_chunk(niters_, &fiter_, &liter_))
9 {
10   for (iter_ = fiter_, i = 0 + fiter_ * 1;
11     iter_ < liter_; iter_++, i += 1)
12   {
13   /* #pragma omp cancel for if(i == 10) */
14   if (i == 10)
15   if (ort_enable_cancel(2))
16     goto CANCEL_FOR16;
17   else
18     if (ort_check_cancel(2))
19     goto CANCEL_FOR16;
20     j++;
21   }
22 }
23 CANCEL_FOR16 :
24   ;
25 ort_leaving_for();
26 ort_barrier_me();
```

Κώδικας 5.11: Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο for

5.5 Cancellation σε taskgroup construct

Για την υλοποίηση του taskgroup, δημιουργήθηκε η δομή του Κώδικα 5.17 στο runtime του OMPi.

Το πεδίο next χρησιμοποιείται για την υλοποίηση ανακύκλωσης της μνήμης(recycler - γρηγορότερη υλοποίηση, αντί για malloc() και free()), το parent αποθηκεύει την πληροφορία ποιός δημιούργησε το taskgroup και το is_canceled είναι ένα flag που σηματοδοτεί την ενεργοποίηση του cancel.

Στο taskgroup, υπάρχει ο περιορισμός ότι το cancel construct πρέπει να είναι αυστηρά μέσα σε ένα task . Όταν δημιουργείται ένα task, δημιουργείται μία ειδική δομή που αποθηκεύει όλες τις απαραίτητες πληροφορίες που αφορούν την εκτέλεσή του. Στην δομή αυτή χρειάστηκε να προστεθεί ακόμα ένα πεδίο τύπου taskgroup_ t ώστε να κρατάω την πληροφορία, σε ποιό taskgroup ανήκει κάποιο task.

Η λειτουργία του taskgroup

Ένα taskgroup είναι μία ομάδα από tasks. Το χαρακτηριστικό τους γνώρισμα, είναι οτι όσα tasks έχουν δημιουργηθεί μέσα σε αυτή την ομάδα, θα πρέπει να τελειώσουν την εκτέλεσή τους, προτού η ροή του προγράμματος συνεχιστεί έξω από το μπλόκ κώδικα που ορίζει το taskgroup. Δηλαδή, ένα task που δημιουργεί ένα άλλο task, θα πρέπει να περιμένει να τελειώσει αυτό, αλλα και τα παιδιά των παιδιών του.

Υλοποίηση του cancel σε taskgroup

Όταν ένα task(εργασία), που είναι μέσα σε ένα taskgroup, συναντήσει την εντολή #pragma omp cancel taskgroup, τότε καλείται η συνάρτηση ort_enable_cancel(1), που ενεργοποιεί το πεδίο is_canceled του αντίστοιχου taskgroup. Το νήμα που ενεργοποίησε το cancel οδηγείται στο τέλος του task . Στον αλγόριθμο για την εκτέλεση των tasks, γίνεται έλεγχος για το άν κάποιο task ανήκει σε taskgroup και αν για το συγκεκριμένο taskgroup έχει ενεργοποιηθεί το cancel. Σε περίπτωση, που έχει ενεργοποιηθεί το cancel, το συγκεκριμένο task ποθ προορίζοταν για εκτέλεση αγνοείται και ο αλγόριθμος εκτέλεσης συνεχίζει με το επόμενο task.

Γενικότερα, κάθε νήμα έχει μία ουρά που αποθηκεύει τα tasks που έχουν δημιουργηθεί και είναι προς εκτέλεση. Στο Κώδικα 5.18, όλα τα νήματα που εκτελούν την παράλληλη περιοχή δημιουργούν tasks που ανήκουν στο ίδιο taskgroup. Στο συγκεκριμένο παράδειγμα, τα νήματα θα δημιουργήσουν όλα τα tasks αλλά θα εκτελέσουν μόνο το task 1, γιατί άν ένα νήμα προχωρήσει στην εκτέλεση του δεύτερου task θα ενεργοποιήσει το cancel taskgoup και κανένα άλλο task που ανήκει στο taskgroup, δεν θα εκτελεστεί.

Έξοδος:

```
In task1, thread : 0
In task1, thread : 1
In task1, thread : 2
In task1, thread : 3
In task1, thread : 4
In task1, thread : 5
```

Έχουν υλοποιηθεί κάποιες βελτιστοποιήσεις, όπως το κλέψιμο εργασιών και η γρήγορη εκκίνηση εκτέλεσης ενός task. Ο OMPi παράγει δύο φορές τον κώδικα των tasks όπως φαίνεται στον παραγόμενο κώδικα 5.19. Υπάρχουν κάποιες βελτιώσεις στην λειτουργία

τους που αφορά τον τρόπο εκτέλεσης των εργασιών. Όταν δημιουργείται ένα task μπορεί να εκτελεστεί την ίδια στιγμή από το ίδιο νήμα ή και από διαφορετικό. Τα νήματα μπορούν να κλέψουν τις εργασίες ενός άλλου task και να τις εκτελέσουν αντ' αυτού. Με αυτό τον τρόπο ο χρόνος εκτέλεσης μειώνεται σημαντικά. Η γρήγορη εκκίνηση ενός task που φαίνεται στις γραμμές 15-21, 32-41, 52-59, γίνεται όταν η ουρά των εργασιών ενός νήματος έχει γεμίσει και δεν μπορούν να εισέλθουν άλλες εργασίες. Τότε κάποια άλλα νήματα εκτελούν κατευθείαν τον κώδικα που είναι περικλυώμενος μέσα στο task.

5.6 Υλοποίηση των Cancellation Point στον OMPi

Όπως αναφέρθηκε και σε προηγούμενη ενότητα, τα σημεία ακύρωσης εισάγωνται από τον χρήστη. Σε ένα cancellation point, καλείται η συνάρτηση ort_check_cancel(type), όπου το type θα πρέπει να είναι το αναγνωριστικό που αντιστοιχεί στην οδηγία που είναι άμεσα περικλυώμενο το σημείο ακύρωσης.

Ένα ακόμα σημείο ακύρωσης είναι το φράγμα (barrier). Για την υλοποίηση του cancel χρειάστηκε να γίνουν αρκετές αλλαγές στον κώδικά του. Προστέθηκαν έλεγχοι για cancel την στιγμή που εισέρχονται τα νήματα, κατά την αναμονή τους στο φράγμα και αμέσως μετά από αυτήν. Στο barrier γίνεται έλεγχος μόνο για ενεργοποίηση της ακύρωσης σε μία παράλληλη περιοχή και σε taskgroup, καθώς δεν επιτρέπεται η ύπαρξη barrier μεσα σε for ή sections.

Επίσης, άν η μεταβλητή περιβάλλοντος OMP_CANCELLATION δεν ήταν ενεργή, δεν ήταν απαραίτητο να γίνουν όλοι αυτοί οι έλεγχοι. Γι αυτό, στην κλήση της συνάρτησης του barrier στο runtime, εκτελείται πρώτα έλεγχος για το άν έχει ενεργοποιηθεί η μεταβλητή περιβάλλοντος για το cancel. Στην περίπτωση που είναι ενεργοποιημένη, τότε εκτελούνται συναρτήσεις με όλους του απαραίτητους ελέγχους για πιθανή ενεργοποίηση του cancel. Σε αντίθετη περίπτωση, καλούνται οι συναρτήσεις του barrier χωρίς τους επιπρόσθετους ελέγχους για cancel, πράγμα που αφαιρεί σημαντική επιβάρυνση στο χρόνο εκτέλεσης των προγραμμάτων.

```
1 int ort_get_dynamic_chunk(int niters, int chunksize, int *fiter, int *liter,int *
      ignored, ort_gdopt_t *t)
2 {
3 ort_eecb_t          *me = __MYCB;
4
5 if (chunksize <= 0)
6 ort_error(1, "fatal: dynamic chunksize <= 0 requested!\n");
7 if (me->parent != NULL)
8   if (me->parent->cancel_for == CANCEL_ACTIVATED)
9     return (0);
10
11 if (t->nth == 1)
12 {
13   /* t->data used specially here */
14   if (t->data == NULL) return (0);
15   *fiter = 0;      /* Get just 1 chunk: all iterations */
16   *liter = niters;
17   t->data = NULL;
18   return (1);
19 }
20   else
21   {
22   /* t->data shall hold the next iter to give away */
23   if (*(t->data) >= niters) {
24   *fiter = niters + 1; return (0);
25   } /* done */
26
27
28   #if defined(HAVE_ATOMIC_FAA) && !defined(EE_TYPE_PROCESS)
29   *fiter = _faa(t->data, chunksize);
30
31   #else
32    ee_set_lock((ee_lock_t *) t->lock);
33   *fiter = *(t->data);
34   (*(t->data)) += chunksize;
35    ee_unset_lock((ee_lock_t *) t->lock);
36
37  #endif
38   *liter = *fiter + chunksize;
39     if (*liter > niters)
40     *liter = niters;
41   }
42 return (1);
43 }
```

Κώδικας 5.12: Επαναληπτικός μηχανισμός ανάθεσης των επαναλήψεων στα νήματα σε dynamic schedule

```
1 main(){
2  int i,j=0;
3   #pragma omp parallel for schedule(dynamic) num_threads(6)
4   for(i=0;i<30;i++){
5     #pragma omp cancel for if(i==10)
6       j++;
7    }
8   printf("%d \n",j);
9 }
```

Κώδικας 5.13: Εφαρμογή του Cancellation σε for

```
1   /* (l12) #pragma omp parallel num_threads(6) -- body moved below */
2 # 12 "injected_code"
3   {
4     /* #pragma omp for schedule(dynamic ) */
5     int i;
6     struct _ort_gdopt_ gdopt_;
7     int niters_ = 0, iter_ = -1, fiter_, liter_ = -2;
8
9     ort_entering_for(0, 0, &gdopt_);
10    niters_ = ort_num_iters(1, (long) (30 - (0)), (long) 1, (int *) 0);
11    while (ort_get_dynamic_chunk(niters_, 1, &fiter_, &liter_, (int *) 0, &gdopt_))
12      {
13        for (iter_ = fiter_, i = 0 + fiter_ * 1; iter_ < liter_; iter_++, i += 1)
14          {
15            (*j)++;
16          }
17      }
18  }
19  ort_taskwait(2);
20  return ((void *) 0);
```

Κώδικας 5.14: Παραγόμενος κώδικας του OMPi, χωρίς την υλοποίηση για το cancel στο for με schedule dynamic

```
1     /* #pragma omp for schedule(dynamic ) */
2     int i;
3     struct _ort_gdopt_ gdopt_;
4     int niters_ = 0, iter_ = -1, fiter_, liter_ = -2;
5
6     ort_entering_for(0, 0, &gdopt_);
7     niters_ = ort_num_iters(1, (long) (30 - (0)), (long) 1, (int *) 0);
8     while (ort_get_dynamic_chunk(niters_, 1, &fiter_, &liter_, (int *) 0, &gdopt_))
9       {
10        for (iter_ = fiter_, i = 0 + fiter_ * 1; iter_ < liter_; iter_++, i += 1)
11          {
12            /* #pragma omp cancel for if(i == 10) */
13            if (i == 10)
14              if (ort_enable_cancel(2))
15                goto CANCEL_FOR12;
16            else
17              if (ort_check_cancel(2))
18                goto CANCEL_FOR12;
19            (*j)++;
20          }
21      }
22    CANCEL_FOR12 :
23      ;
24  }
25  CANCEL_PARALLEL17 :
26    ort_taskwait(2);
27  return ((void *) 0);
```

Κώδικας 5.15: Παραγόμενος κώδικας του OMPi, με την υλοποίηση για το cancel στο for με schedule dynamic

```
1 int ort_get_guided_chunk(int niters, int chunksize, int *fiter, int *liter,int *
     ignored, ort_gdopt_t *t)
2 {
3   int   ch;
4   ort_eecb_t         *me = __MYCB;
5  if (chunksize <= 0) ort_error(1, "fatal: guided chunksize <= 0 requested!\n");
6
7   if (me->parent != NULL)
8     if (me->parent->cancel_for == CANCEL_ACTIVATED)
9     return (0);
10
11 if (t->nth == 1)
12 {
13   if (t->data == NULL) return (0);      /* t->data used specially here */
14   *fiter = 0;                    /* Get just 1 chunk: all iterations */
15   *liter = niters;
16   t->data = NULL;
17   return (1);
18 }
19
20  if (*(t->data) >= niters) { *fiter = niters + 1; return (0); } /* done */
21
22   #if defined(HAVE_ATOMIC_CAS) && !defined(EE_TYPE_PROCESS)
23   do
24   {
25     *fiter = *(t->data);
26     ch = niters - *fiter;
27     if (ch > chunksize)
28     {
29     ch = (ch + t->nth - 1) / t->nth;
30     if (ch < chunksize)
31     ch = chunksize;
32     }
33
34      if (me->parent->cancel_for == CANCEL_ACTIVATED){
35         return (0);
36       }
37   }while (!_cas(t->data, (*fiter), (*fiter) + ch));
38 #else
39   ee_set_lock((ee_lock_t *) t->lock);
40   *fiter = *(t->data);
41   ch = niters - *fiter;
42   if (ch > chunksize)
43   {
44     ch = (ch + t->nth - 1) / t->nth;
45     if (ch < chunksize)
46   ch = chunksize;
47   }
48   (*(t->data)) += ch;
49   ee_unset_lock((ee_lock_t *) t->lock);
50 #endif
51
52   *liter = *fiter + ch;
53   return (ch != 0);
54 }
```

Κώδικας 5.16: Επαναληπτικός μηχανισμός ανάθεσης των επαναλήψεων στα νήματα guided Schedule

```
1 typedef struct taskgroup_s
2 {
3     struct taskgroup_s *next;     /* For use in garbage collector */
4     struct taskgroup_s *parent;
5     volatile int       is_canceled;
6 }taskgroup_t;
```

Κώδικας 5.17: Η δομή ενός taskgroup.

```
1  void main(){
2   #pragma omp parallel num_threads(6)
3   {
4     #pragma omp taskgroup
5     {
6       #pragma omp task
7           printf("In task1, thread :\t%d\n",omp_get_thread_num());
8       #pragma omp task
9       {
10        #pragma omp cancel taskgroup
11        printf("In task2, thread :\t%d\n",omp_get_thread_num());
12      }
13      #pragma omp task
14        printf("In task3, thread :\t%d\n",omp_get_thread_num());
15      }
16  }
17 }
```

Κώδικας 5.18: Ένα απλό παράδειγμα για την εφαρμογή του cancel taskgroup

```
1  static void * _thrFunc0_(void * __arg)
2 {
3   /* byresult variables */
4   /* (l6) #pragma omp parallel  -- body moved below */
5   {
6     /* #pragma omp taskgroup  */
7
8     ort_entering_taskgroup();
9
10    {
11      if (omp_in_final() || ort_task_throttling())
12        {
13          void * _itn;
14
15          _itn = ort_task_immediate_start(0);
16          {
17            printf("In task1, thread :\t%d\n", omp_get_thread_num());
18            CANCEL_task_10 :
19              ;
20          }
21          ort_task_immediate_end(_itn);
22        }
23      else
24        /* (l10) #pragma omp task  */
25        {
26          ort_new_task(_taskFunc0_, (void *) 0, 0, 0);
27        }
28      if (omp_in_final() || ort_task_throttling())
29        {
30          void * _itn;
31
32          _itn = ort_task_immediate_start(0);
33          {
34            /* #pragma omp cancel taskgroup */
35            if (ort_enable_cancel(1))
36              goto CANCEL_task_12;
37            printf("In task2, thread :\t%d\n", omp_get_thread_num());
38            CANCEL_task_12 :
39              ;
40          }
41          ort_task_immediate_end(_itn);
42        }
43      else
44        /* (l12) #pragma omp task  */
45        {
46          ort_new_task(_taskFunc1_, (void *) 0, 0, 0);
47        }
48      if (omp_in_final() || ort_task_throttling())
49        {
50          void * _itn;
51
52          _itn = ort_task_immediate_start(0);
53          {
54
55            printf("In task3, thread:\t%d\n", omp_get_thread_num());
56            CANCEL_task_17 :
57              ;
58          }
59          ort_task_immediate_end(_itn);
60        }
```

```
1       else
2
3         /* (l17) #pragma omp task  */
4         {
5           ort_new_task(_taskFunc2_, (void *) 0, 0, 0);
6         }
7     }
8     ort_leaving_taskgroup();
9
10    }
11   CANCEL_parallel_6 :
12     ort_taskwait(2);
13   return ((void *) 0);
14 }
15
16 /* Outlined code for (l17) #pragma omp task  */
17
18 static void * _taskFunc2_(void * __arg)
19 {
20   /* byresult variables */
21   /* (l17) #pragma omp task  -- body moved below */
22
23   {
24
25   printf("In task3, thread :\t%d\n", omp_get_thread_num());
26     CANCEL_task_17 :
27       ;
28   }
29   return ((void *) 0);
30 }
31 /* Outlined code for (l12) #pragma omp task  */
32 static void * _taskFunc1_(void * __arg)
33 {
34   /* byresult variables */
35   /* (l12) #pragma omp task  -- body moved below */
36   {
37     /* #pragma omp cancel taskgroup */
38     if (ort_enable_cancel(1))
39       goto CANCEL_task_12;
40       printf("In task2, thread :\t%d\n", omp_get_thread_num());
41     CANCEL_task_12 :
42       ;
43   }
44   return ((void *) 0);
45 }
46
47 /* Outlined code for (l10) #pragma omp task  */
48
49 static void * _taskFunc0_(void * __arg)
50 {
51   /* byresult variables */
52   /* (l10) #pragma omp task  -- body moved below */
53   {
54     printf("In task1, thread :\t%d\n", omp_get_thread_num());
55     CANCEL_task_10 :
56       ;
57   }
58   return ((void *) 0);
59 }
```

Κώδικας 5.19: Ο παραγόμενος κώδικας του Κώδικα 5.18

Κεφάλαιο 6

Πειράματα

Η υλοποίηση που περιγράψαμε στο Κεφάλαιο 5, ελέγχθηκε διεξοδικά σε πολλές διαφορετικές πλατφόρμες και με διαφορετικούς στόχους κάθε φορά.

Κατ' αρχήν όλα τα πειράματα που περιγράφουμε σε αυτό το κεφάλαιο εκτελέστηκαν στα παρακάτω παράλληλα συστήματα:

- PARADOX με 6 CPU's AMD Phenom™ II X6 1055T Processor(6 Cores 2.8GHz - Turbo 3.3GHz, 1 Core/CPU, 4.0 GB RAM, με λειτουργικό Debian 7.8 (wheezy) 64-bit.
- PARAGUAY με 4 x Intel Xeon 7040 Paxville MP (2 HT cores @ 3.00Ghz) 4 cpus / 8 cores / 16 thread, 8.0 GB RAM, με λειτουργικό Linux Debian Squeeze.
- PARAGON με 2 x AMD Opteron 6166 (12 cores, 1.8GHz),16 GB RAM, με λειτουργικό Linux Debian Squeeze.

Δεύτερον, εκτός από τον OMPi, είχαμε στην διαθεσή μας και τους εξής μεταφραστές:

- gcc version 4.7.2 (Debian 4.7.2-5)
- Intel icc Version 12.0.0

Τέλος, οι στόχοι μας ήταν τόσο για τον έλεγχο ορθότητας όσο και επιδόσεων.

6.1 Πειράματα ορθότητας

Για την υλοποίηση του cancellation στον OMPi, έγινε μία σειρά από πειράματα ορθότητας, ώστε να επαληθευτεί πως η υλοποίηση έχει την αναμενόμενη συμπεριφορά. Εκτός από κάποια πειράματα που δημιουργήσαμε, ώστε να γίνει έλεγχος όλων των οριακών καταστάσεων, χρησιμοποιήσαμε και μετροπρογράμματα του libgomp [1] για την έκδοση OpenMP v.4.0 που περιέχουν δοκιμές για την οδηγία cancel. Στα συγκεκριμένα πειράματα έγινε έλεγχος για την κλήση της συνάρτησης omp_get_cancellation(), όπου επιστρέφεται η τιμή της μεταβλητής περιβάλλοντος για το CANCELLATION, την χρήση του barrier ως σημείο ακύρωσης (cancellation point) και τον συντονισμό νημάτων ώστε επιτυχώς να πραγματοποιηθεί η ακύρωσης μιας παράλληλης ομάδας.

Όλα τα περιράματα εκτελέστηκαν και στις 3 πλατφόρμες με επιτυχία (PARADOX, PARAGON και PARAGUAY).

6.2 EPCC Microbenchmarks για την υποστήριξη cancellation στον OMPi

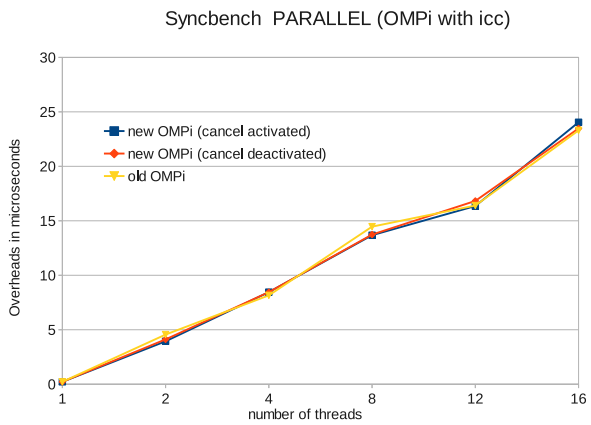
Η επόμενη κατηγορία πειραμάτων αφορά στις αποδόσεις της υλοποίησης και όχι μόνο. Στόχος μας είναι, να διαπιστώσουμε άν η προσθήκη της λογικής του cancel επιβαρύνει τα υπόλοιπα constructs του OpenMP μιάς και προσθέσαμε νέο κώδικα ο οποίος εκτελείται απο το σύνολο σχεδόν της βιβλιοθήκης runtime.

Προκειμένου να διαπιστώσουμε με ακρίβεια την ύπαρξη αλλά και την ποσότητα της επιβάρυνσης χρησιμοποιήσαμε τα EPCC Microbenchmarks [4]. Στόχος μας ήταν να μην επιβαρύνουμε σημαντικά την απόδοση του OMPi στη χρήση των οδηγιών του OpenMP έτσι ώστε να υποστηρίζει πλήρως την χρήση της οδηγίας cancel. Τα EPCC είναι μικρά προγράμματα που σκοπό έχουν να μετρήσουν την επιβάρυνση των οδηγιών OpenMP, δηλαδή το χρόνο που απαιτούν για να ολοκληρωθούν ανάλογα και με τα νήματα που συμμετέχουν σε αυτές. Τα προγράμματα αυτά χωρίζονται σε τρείς κατηγορίες, schedbench, taskbench και syncbench. Κάθε κατηγορία εξετάζει διαφορετικές λειτουργίες. Η κατηγορία schedbench αποτελείται από διάφορες δομές επανάληψης "for" οι οποίες έχουν παραλληλοποιηθεί με OpenMP και οι επαναλήψεις τους έχουν δρομολογηθεί με διάφορες παραμέτρους. Η κατηγορία taskbench αποτελείται από διάφορες λειτουργίες του OpenMP που αφορούν μόνο τις εργασίες-tasks, όπως είναι η παράλληλη εκτελεσή τους και ο συγχρονισμός τους. Η κατηγορία syncbench εξετάζει τις επιβαρύνσεις όλων των οδηγιών του OpenMP γενικότερα.

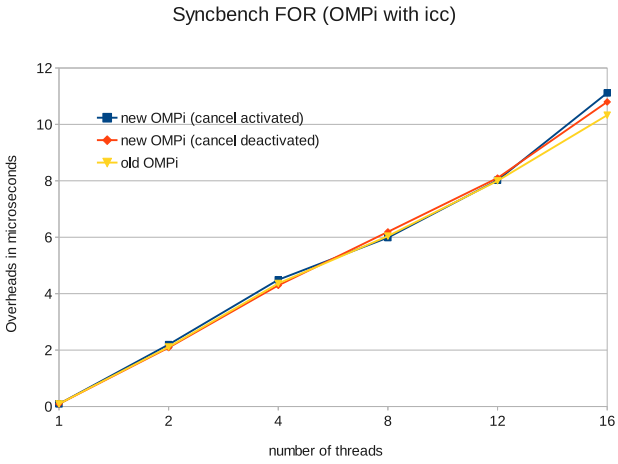
Για τα EPCC Microbenchmarks, πραγματοποιήθηκαν εκτελέσεις και των τριών κατηγοριών με αριθμό νημάτων 1, 2, 4, 8, 12, 16, 20 και 24. Έγιναν 3 μετρήσεις, μία για την νέα έκδοση του OMPi με την υλοποίηση του cancel και με ενεργοποιημένη την μεταβλητή περιβάλλοντος για το cancel, άλλη μία εκτέλεση με την νέα έκδοση του OMPi με την υλοποίηση του cancel, αλλά με την μεταβλητή περιβάλλοντος OMP_CANCELLATION false και μία ακόμα εκτέλεση της παλίας έκδοσης του OMPi δίχως την υλοποίηση του cancel, έτσι ώστε να γίνει αντιληπτό το μέγεθος των overheads (χρόνου επιβάρυνσης), που προστέθηκε ή μειώθηκε με την προσθήκη της εν λόγω υλοποίησης, καθώς δεν υπάρχει ειδικό μετροπρόγραμμα που χρησιμοποιεί την οδηγία του cancel.

6.2.1 Syncbench microbencmark

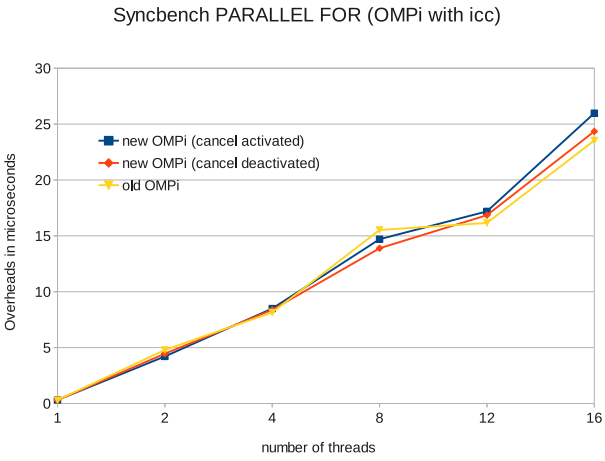
Η εκτέλεση των EPCC benchmarks έγινε μεταγλωτίζοντας τον OMPi με τον icc compiler και τον gcc. Στο Σχήμα 6.1 φαίνονται τα αποτελέσματα των EPCC benchmarks με την εκτέλεση του OMPi μεταγλωτισμένο με τον icc , ενώ στο Σχήμα 6.2 παρουσιάζονται τα αποτελέσματα με τον OMPi μεταγλωτισμένο με τον gcc compiler. Απο το syncbench παρουσιάζουμε ενδεικτικά τα benchmarks PARALLEL, FOR, PARALLEL FOR και BARRIER.



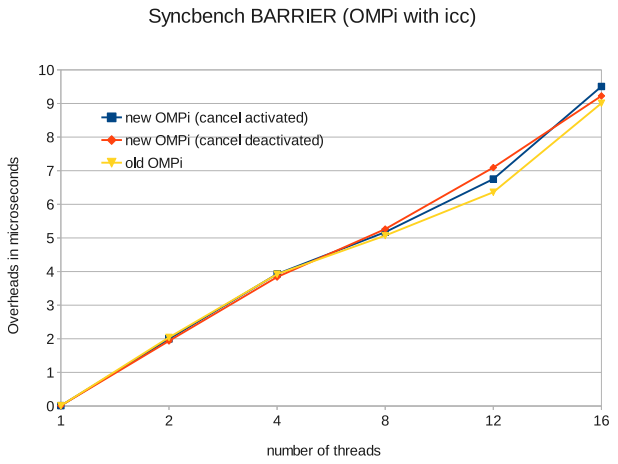
(α΄) Επιβάρυνση του PARALLEL



(β΄) Επιβάρυνση του FOR



(γ΄) Επιβάρυνση του PARALLEL FOR

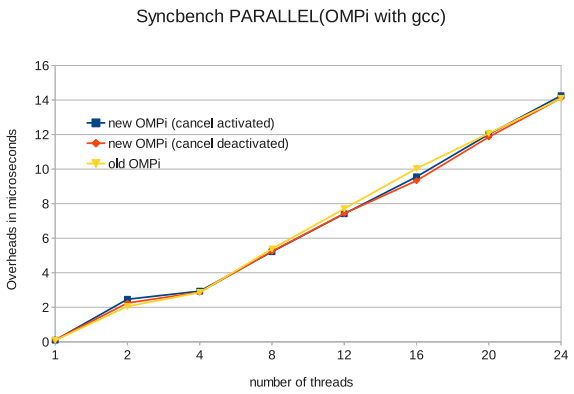


(δ΄) Επιβάρυνση του BARRIER

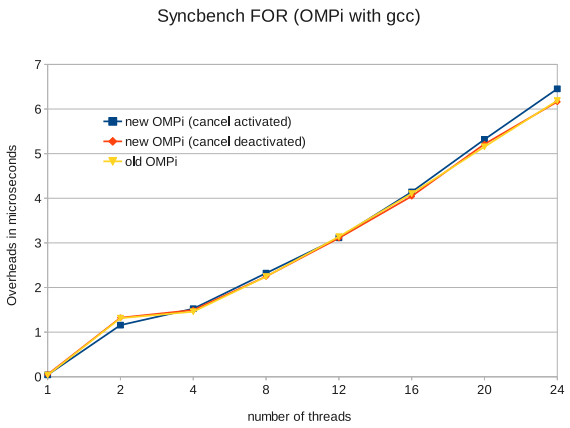
Σχήμα 6.1: Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον icc. Η εκτέλεση έγινε στο σύστημα PARAGUAY.

Ήταν αναμενόμενο πως η υποστήριξη για το cancel construct, θα προσέθετε κάποια επιβάρυνση λόγω των ελέγχων για ενεργό cancel. Κυρίως στο barrier αναμέναμε μια σημαντική επιβάρυνση, γιατί είναι σημείο ακύρωσης και πρέπει ανα πάσα στιγμή να ελέγχουμε για την ενεργοποίηση του cancel. Όμως, η σημαντική αυτή επιβάρυνση προστίθεται όταν η μεταβλητή περιβάλλοντος OMP_CANCELLATION είναι αληθής. Σε διαφορετική περίπτωση, ο barrier εκτελείται χωρίς τους ελέγχους για ενεργό cancel, με αποτέλεσμα να αποφύγουμε τις επιπρόσθετες επιβαρύνσεις με την υλοποίηση των 2 εκδοχών του barrier που αναφέραμε στην ενότητα 5.6.

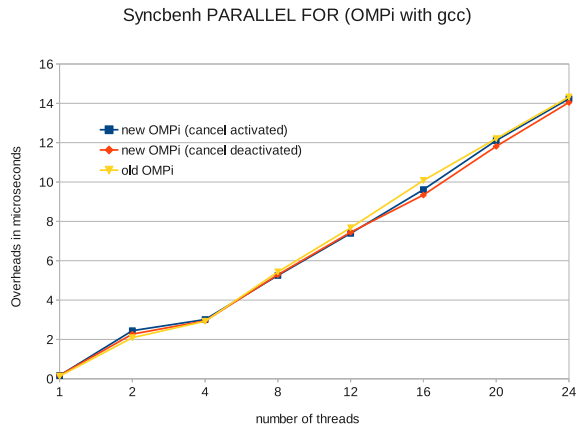
Σε γενικές γραμμές, για την υλοποίηση του cancellation δεν επιβαρύναμε αρκετά τον OMPi, παρά μόνο ένα ποσοστό μέχρι 3-4 %, και αυτό μόνο στην περίπτωση όπου η μεταβλητή περιβάλλοντος OMP_CANCELLATION είναι αληθής.



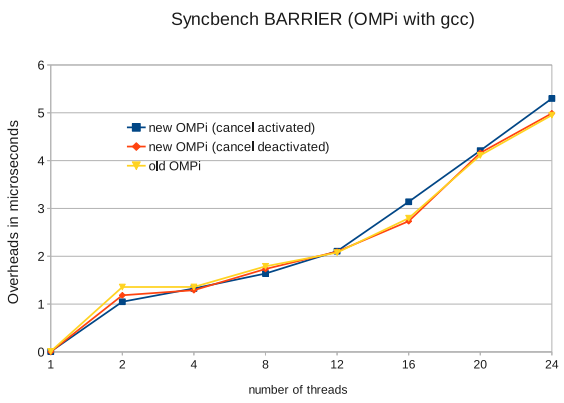
(α΄) Επιβάρυνση του PARALLEL



(β΄) Επιβάρυνση του FOR



(γ΄) Επιβάρυνση του PARALLEL FOR

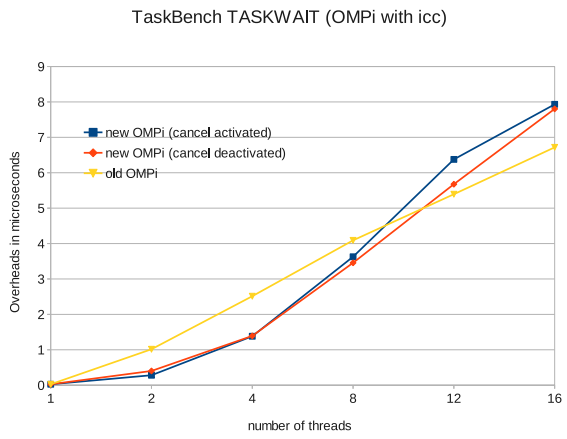


(δ΄) Επιβάρυνση του BARRIER

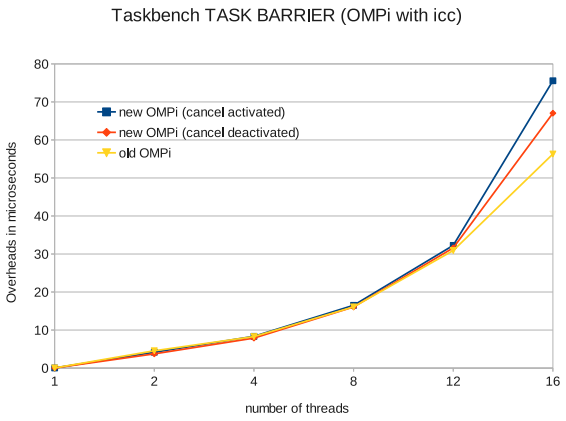
Σχήμα 6.2: Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον gcc. Η εκτέλεση έγινε στο σύστημα PARAGON.

6.2.2 Taskbench microbencmark

Ομοίως με την εκτέλεση για syncbench, έτσι και για το taskbench, εκτελέσαμε τα πειράματα με τον OMPi μεταγλωτισμένο με τον icc και τον gcc, ώστε να έχουμε μεγαλύτερη ακρίβεια στα αποτέλεσματα. Στο Σχήμα 6.3 παραθέτονται τα αποτελέσματα των EPCC Bencmarks του OMPi με τον icc, ενώ στο Σχήμα 6.4 παραθέτονται τα αποτελέσματα των EPCC Bencmarks του OMPi με τον gcc. Από το taskbench, παρουσιάζουμε ενδεικτικά το TASK WAIT και το TASK BARRIER benchmarks.

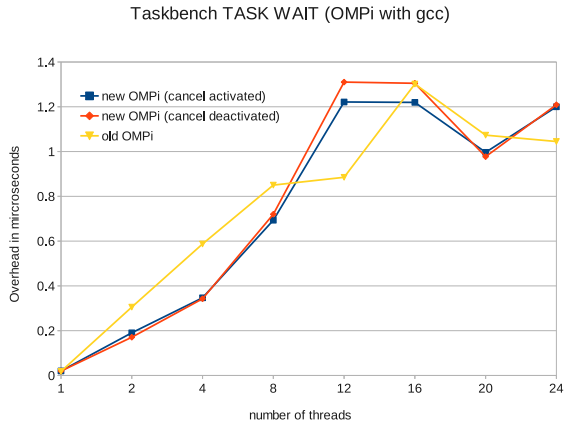


(α΄) Επιβάρυνση του PARALLEL

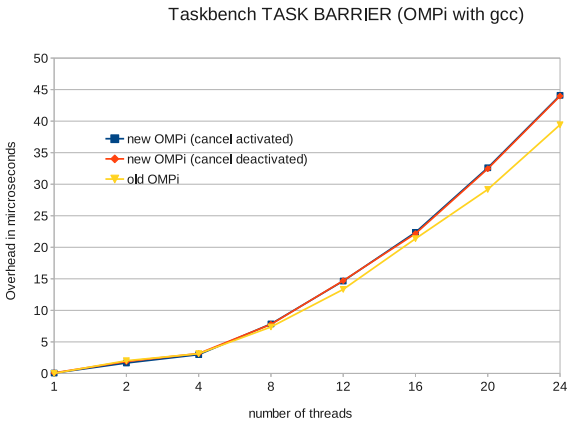


(β΄) Επιβάρυνση του FOR

Σχήμα 6.3: Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον icc. Η εκτέλεση έγινε στο σύστημα PARAGUAY.



(α΄) Επιβάρυνση του TASK WAIT



(β΄) Επιβάρυνση του TASK BARRIER

Σχήμα 6.4: Αποτελέσματα EPCC Bencmarks με τον OMPi μεταγλωτισμένο με τον gcc. Η εκτέλεση έγινε στο σύστημα PARAGON.

Για τον λόγο που εξηγήσαμε και στην ενότητα 6.2.1, αναμέναμε πως στο TASK BARRIER θα υπήρχε μία επιπλέον επιβάρυνση για τον έλεγχο του cancel. Στο TASK WAIT benchmark, φαίνεται να υπάρχει σημαντική αυξομείωση, αλλά δεν ισχύει κάτι τέτοιο γιατί η μέτρηση είναι σε microseconds και υπάρχει πολύ μικρή διαφορά μεταξύ των τιμών. Οπότε οι μετρήσεις που πέρνουμε είναι σχετικά σταθερές στα όρια του στατιστικού λάθους. Επομένως ούτε σε αυτή την περίπτωση, δεν επιβαρύναμε σημαντικά, τον OMPi (9-10% περισσότερη επιβάρυνση στην χειρότερη περίπτωση).

6.3 Εφαρμογή για την χρήση του cancel

Η τελευταία κατηγορία πειραμάτων ήταν η χρονομέτρηση πραγματικών εφαρμογών προκειμένου να διαπιστώσουμε τα πρακτικά οφέλη της χρήσης της οδηγίας cancel. Η εφαρμογή που δημιουργήθηκε είναι παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο με την βοήθεια της δημιουργίας των tasks. Για κάθε κόμβο του δέντρου έχουμε πληροφορίες για :

- το επίπεδο που βρίσκεται
- την τιμή που έχει και το αναγνωριστικό τους
- τα παιδιά του, αποθηκεύοντας τους σε 2 δείκτες right και left αντίστοιχα.

Όπως φαίνεται και στον Κώδικα 6.5 του παραδείγματος, κατά την εκτέλεση της αναζήτησης της ζητούμενης τιμής, σε περίπτωση που δεν έχει βρεθεί, δημιουργούνται νέες εργασίες για να ψάξουν στα παιδιά του τρέχοντος κόμβου. Επομένως, αναδρομικά θα δημιουργούνται συνεχόμενα εργασίες μέχρι να βρεθεί η ζητουμενη τιμή. Όταν βρεθεί σε κάποιο κόμβο, τότε στον πατέρα αυτού, επιστρέφεται το αναγνωριστικό του κόμβου που βρέθηκε και αναδρομικά η ζητούμενη τιμή ανεβαίνει στην κορυφή του δέντρου και τερματίζεται η λειτουργία της αναζήτησης.

```
1 binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
2   binary_tree_t *found = NULL;
3
4   if(tree){
5     if (tree->value == value)
6     {
7         found = tree;
8     }
9     else
10    {
11      #pragma omp task shared(found) if(level<11)
12      {
13        found = search_tree(tree->left, value, level + 1);
14      }
15      #pragma omp task (found) if(level<11)
16      {
17        found= search_tree(tree->right, value, level + 1);
18      }
19      #pragma omp taskwait
20    }
21  }
22 return found;
23 }
24
25 binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
26   binary_tree_t *found = NULL;
27   #pragma omp parallel shared(found, tree, value)
28       #pragma omp master
29       {
30           #pragma omp task
31           {
32               found = search_tree(tree, value, 0);
33           }
34       }
35 }
```

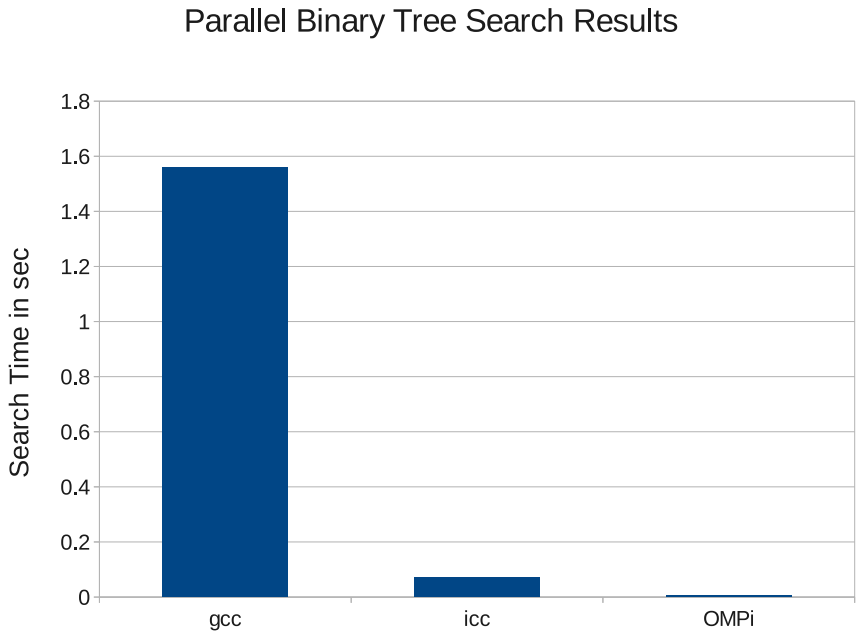
Κώδικας 6.5: Παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο.

Λόγω της τυχαιότητας που υπάρχει στην δημιουργία και την εκτέλεση των εργασιών, καλό θα ήταν να μπορέσουμε να ελέγξουμε την εκτέλεσή τους, τοποθετώντας τα σε μία ομάδα. Το taskgroup θα δημιουργείται από το master νήμα και θα περιέχει όλες τις εργασίες που δημιουργήθηκαν στην παράλληλη αναζήτηση. Με αυτό τον τρόπο θα μπορέσουμε να ανιχνεύσουμε την εύρεση της τιμής, και να σταματήσουμε την δημιουργία και την εκτέλεση

των περιττών εργασιών. Η παραπάνω λειτουργία χρησιμοποιείται για την υλοποίηση του cancel στο taskgroup.

6.3.1 Σύγκριση του μεταφραστή OMPi με άλλους compilers για την συγκεκριμένη εφαρμογή

Στο πείραμα αυτό, πήραμε αποτελέσματα(χρονομέτρηση) για την αναζήτηση σε πλήρες δυαδικό δέντρο που φαίνεται στον Κώδικα 6.5 από τους μεταφραστές gcc, icc και τον OMPi. Στο σχήμα 6.6 φαίνονται τα αποτελέσματα για την αναζήτηση χωρίς την χρήση του cancel, αναζητώντας στο ίδιο δέντρο τον ίδιο κόμβο. Το δέντρο είναι μεγέθους 15 επιπέδων(65.535 κόμβοι και εργασίες). Έχουν γίνει και μετρήσεις για δέντρο επιπέδου 20 (2.097.152 κόμβοι και εργασίες) και επιπέδου 25 (67.108.862 κόμβοι και εργασίες). Τα αποτελέσματα είναι εντελώς ανάλογα για κάθε μία από τις παραπάνω περιπτώσεις και παραθέτω μόνο τα αποτελέσματα για το δέντρο επιπέδου 15.



Σχήμα 6.6: Αποτελέσματα εκτέλεσης της εφαρμογής

Εύκολα θα παρατηρήσουμε ότι ο OMPi είναι πιο γρήγορος από τους άλλους δύο μεταφραστές, όσον αφορά την διαχείριση και την εκτέλεση των εργασίων. Αυτό συμβαίνει γιατί παράγει κώδικα που υλοποιεί δύο διαφορετικές εκτελέσεις αυτών. Πιο συγκεκριμένα, όταν δημιουργείται ένα task μπορεί να εκτελεστεί την ίδια στιγμή από το ίδιο νήμα ή και από διαφορετικό. Κάθε νήμα έχει μία ουρά, που αποθηκεύει τα task που έχουν δημιουργηθεί και είναι προς εκτέλεση. Έχουν υλοποιηθεί κάποιες βελτιστοποιήσεις, όπως το κλέψιμο εργασιών και η γρήγορη εκκίνηση εκτέλεσης ενός task. Τα νήματα μπορούν να κλέψουν τις εργασίες ενός άλλου task και να τις εκτελέσουν αντ' αυτού. Με αυτό τον τρόπο ο χρόνος εκτέλεσης μειώνεται σημαντικά. Η γρήγορη εκκίνηση ενός task γίνεται όταν η ουρά των εργασιών ενός νήματος έχει γεμίσει και δεν μπορούν να εισέλθουν άλλες εργασίες. Τότε

κάποια άλλα νήματα εκτελούν κατευθείαν τον κώδικα που είναι περικλυώμενος μέσα στο task.

Στον Κώδικα 6.7 έχει πραγματοποιηθεί η παράλληλη αναζήτηση με την χρήση του *#pragma omp cancel taskgroup*, γραμμές 15,22. Αρχικά φτιάχνουμε ένα taskgroup και όλα τα tasks που θα δημιουργηθούν θα ανήκουν σε αυτό. Ο τρόπος αυτός είναι πιο αποτελεσματικός για το λόγο ότι, μόλις βρεθεί η ζητούμενη τιμή το task στο οποίο βρέθηκε *ακυρώνει όλη την ομάδα* με αποτέλεσμα να μην δημιουργηθούν και εκτελεστούν άλλες περιττές εργασίες. Όπως αναφέρθηκε και στην αρχή της τρέχουσας ενότητας, η εκτέλεση των εργασιών

```
1 binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
2   binary_tree_t *found = NULL;
3
4   if(tree){
5     if (tree->value == value)
6     {
7         found = tree;
8     }
9     else
10    {
11      #pragma omp task shared(found) if(level<11)
12      {
13        found = search_tree(tree->left, value, level + 1);
14        if (found) {
15            #pragma omp cancel taskgroup
16          }
17      }
18      #pragma omp task shared(found) if(level<11)
19      {
20        found = search_tree(tree->right, value, level + 1);
21        if (found){
22          #pragma omp cancel taskgroup
23        }
24      }
25      #pragma omp taskwait
26    }
27  }
28 return found;
29 }
30
31 binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
32   binary_tree_t *found = NULL;
33   #pragma omp parallel shared(found, tree, value)
34       #pragma omp master
35       {
36           #pragma omp taskgroup
37           {
38               found = search_tree(tree, value, 0);
39           }
40       }
41 }
```

Κώδικας 6.7: Παράλληλη αναζήτηση σε πλήρες δυαδικό δέντρο με την υλοποίηση Cancel Taskgroup

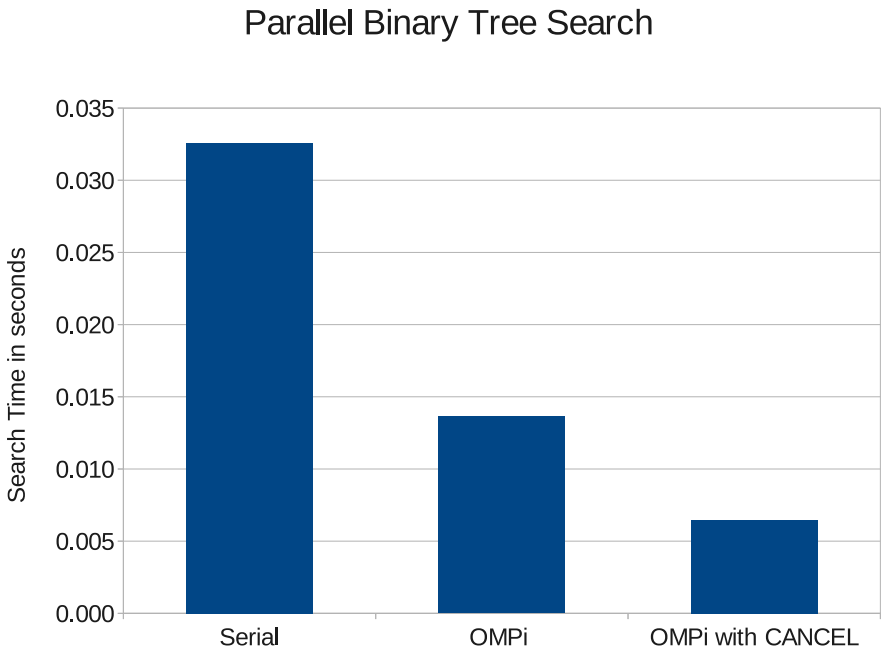
είναι τυχαία. Επομένως, οι διακυμάνσεις στο χρόνο αναζήτησης ήταν μεγάλες. Δηλαδή για τον ίδιο κόμβο, είχαμε σημαντικές αυξομειώσεις στον χρόνο αναζήτησης. Αυτό, είχε να κάνει με την χρονοδρομολόγηση των εργασιών.

Στην αντίστοιχη σειριακή αναζήτηση σε δυαδικό δέντρο, η διάσχιση θα ήταν DFS (Depth-

First-Search). Η αναζήτηση θα ξεκινούσε από την ρίζα. 'Αν δεν έβρισκε τη ζητούμενη τιμή, θα δίαλεγε το αριστερό ή το δεξί παιδί και η εκτέλεση θα συνέχιζε αναδρομικά μέχρι το τελευταίο επίπεδο, επιλέγοντας το ίδιο παιδί κάθε φορά. Επομένως, άν η ζητούμενη τιμή ήταν σε κόμβο από την πλευρά που θα γινόταν αρχικά η διάσχιση, ο χρόνος αναζήτησης θα ήταν πολύ μικρός. Άν όμως ήταν από την αντίθετη πλευρά, τότε ο χρόνος εύρεσης της ζητούμενης τιμής θα ήταν πολύ μεγάλος. Άρα στην σειριακή αναζήτηση ο χρόνος εκτέλεσης εξαρτάται από την θέση του κόμβου με την ζητούμενη τιμή.

Όμως, στην παράλληλη εκτέλεση με εργασίες ο χρόνος εκτέλεσης εξαρτάται μόνο από το βάθος στο οποίο είναι ο ζητούμενος κόμβος. Δεν γινόταν να αποδειχθεί αυτό, με μεμονωμένες προσπάθειες τοποθέτησης κόμβων σε διάφορα επιπεδα και μετρώντας τον χρόνο αναζήτησης για κάθε περίπτωση. Δεν θα ήταν εφικτό να γίνει ακριβής μέτρηση, γιατί υπάρχει ο παράγοντας τυχαιότητας στην εκτέλεση των εργασίων.

Για την απόδειξη αυτού, υλοποιήθηκε μία εφαρμογή που τοποθετούσε την ζητούμενη τιμή κάθε φορά σε διαφορετικό κόμβο του ίδιου επιπέδου, εκτελούσε την παράλληλη αναζήτηση και αποθήκευε τον χρόνο που απαιτήθηκε. Αυτή η διαδικασία επαναλαμβανόταν για όλους τους κόμβους στο συγκεκριμένο επίπεδο. Με αυτόν τον τρόπο, θα μπορούσαμε να υπολογίσουμε μια μέση τιμη του χρόνου αναζήτησης στην κάθε περίπτωση. Θέλοντας να ελέγξουμε την απόδοση του OMPi σε ένα δέντρο με μεγαλύτερο μέγεθος, στο Σχήμα 6.8, για δέντρο με 20 επίπεδα (2,097,152 κόμβους), φαίνεται με μεγάλη διαφορά πως με την χρήση του cancel taskgroup σημειώνεται πολύ σημαντικό κέρδος, της τάξης του 52%.



Σχήμα 6.8: Αποτελέσματα εκτέλεσης της εφαρμογής με την χρήση του Cancel Taskgroup στο σύστημα PARAGON

Είναι σημαντικό να επισημάνουμε ότι η μέτρηση έγινε με 24 νήματα. Βλέπουμε πως στην παράλληλη εκτέλεση χωρίς την υλοποίηση του cancel, δεν έχουμε σημαντική διαφορά με το σειριακό (περίπου 2,5 φορές γρηγορότερα) για το λόγο ότι σπαταλείται περισσότερος χρόνος για την δημιουργία του task, παρά για την εκτέλεσή του. Πάντως το μόνο σίγουρο είναι πως με την υλοποίηση για το cancel στο taskgroup σίγουρα ο χρόνος εκτέλεσης μειώ-

νεται σημαντικά και πόσο μάλλον σε ένα μεταφραστή, όπως είναι ο OMPi, που έχει πολύ γρήγορη υλοποίηση σε tasks, σε σχέση με άλλους μεταφραστές, όπως φαίνεται στο Σχήμα 6.6.

Κεφάλαιο 7

Συμπεράσματα και μελλοντική δουλειά

7.1 Σύνοψη διπλωματικής εργασίας και εφαρμογές

Ο OMPi είναι ένας μεταφραστής για την υποστήριξη OpenMP/C προγραμμάτων. Παρέχει ευελιξία στη χρήση του και μπορεί να προσαρμοστεί σε αρχιτεκτονικές συμβατές με το πρότυπο POSIX. Το δυνατό σημείο του OMPi, είναι η ανεξαρτησία των τμημάτων που τον απαρτίζουν, δηλαδή ο μεταφραστής και το τμήμα runtime. Η υλοποίηση για την παρούσα διπλωματική έγινε κυρίως στο τμήμα runtime, το οποίο αποτελείται από άλλα δύο ανεξάρτητα τμήματα, το ORT και το EELIB.

Το πρόβλημα στην παράλληλη εκτέλεση ήταν πως δέν γινόταν να διακοπεί η εκτέλεση μίας παράλληλης περιοχής. Επομένως, σε μία παράλληλη αναζήτηση ακόμα και αν είχε βρεθεί το ζητούμενο θα συνέχιζε η εκτέλεση μέχρι το τέλος της περιοχής αυτής. Για αυτόν ακριβώς το λόγο, δημιουργήθηκε από το OpenMP μίας νέα οδηγία που στόχο έχει να επιτελέσει αυτό το έργο.

Σε αυτό το σημείο είναι σημαντικό να τονίσουμε πως δεν υπάρχουν αρκετοί μεταφραστές που υποστηρίζουν πλήρως την χρήση της οδηγίας cancel. Επομένως, εισάγωντας στον παραλληλοποιητικό μεταφραστή OMPi την οδηγία cancel, μπορούμε να θεωρηθούμε πρωτοπόροι. Για αυτόν ακριβώς το λόγο, δεν ήταν δυνατό να συγκρίνουμε τις επιδόσεις της οδηγίας cancel του μεταφραστή OMPi με άλλους μεταφραστές.

Στην παρούσα διπλωματική, έγινε υλοποίηση κώδικα στον OMPi για την υποστήριξη του cancel construct του OpenMP v.4.0. Στο τμήμα runtime του μεταφραστή, και πιο συγκεκριμένα στο ORT, προστέθηκαν συναρτήσεις για την ενεργοποίηση και τον έλεγχο του cancel. Επίσης χρειάστηκαν να μεταβληθούν κάποια αρχεία στο υπάρχων τμήμα του ORT, για να πετύχουμε την δυναμική ανίχνευση του cancel σε αυτά τα σημεία. Υπήρξαν και μεταβολές στον compiler, για το πώς θα πρέπει να εξάγεται ο παραγόμενος κώδικας, ώστε να υποστηρίζεται το cancellation.

Μέσα από πειράματα και εφαρμογές που υλοποιήθηκαν, αποδείχθηκε ότι

- ο μεταφραστής OMPi δεν επιβαρύνθηκε σημαντικά με την πρόσθεση κώδικα, στο runtime, για την υλοποίηση της οδηγίας cancel.

- με την χρήση του cancel σε εφαρμογές για αναζήτηση σε δέντρα είναι δυνατό να κερδίσουμε σημαντικό χρόνο εκτέλεσης και να βρεθεί το ζητούμενο ταχύτερα.

7.2 Μελλοντική εργασία στον OMPi

Αποδείχθηκε ότι με την χρήση του cancellation σε παράλληλες περιοχές, μπορεί να επιτύχουμε σημαντικό κέρδος στο χρόνο εκτέλεσης. Ως μελλοντική δουλεία, είναι απαραίτητο να εφαρμοστεί η υποδομή του cancel και σε επιπλέον εφαρμογές που θα μπορέσουν να ωφεληθούν από αυτό. Τέτοιου είδους εφαρμογές είναι η αναζήτηση σε βάσεις δεδομένων, αλγόριθμους σύγκλισης σε κάποια λύση, σε μηχανές εκμάθησης τύπου perceptron, σε εφαρμογές αναζήτησης μονοπατιού σε λαβύρινθο και γενικότερα σε εφαρμογές τεχνητής νοημοσύνης. Μπορούμε να επεκτείνουμε αυτές τις εφαρμογές ώστε να υπολογίζουν τις επιβαρύνσεις και το μέσο κόστος σε κάθε περιοχή όπου χρησιμοποιήθηκε το cancellation.

Όσον αφορά την μελλοντική εργασία πάνω στον OMPi, ταυτόχρονα με την υλοποίηση του cancellation, πραγματοποιείται και η υλοποίηση του declare-target που είναι ένα καινούργιο γνώρισμα του OpenMP v.4.0. Στην νέα έκδοση του μεταφραστή OMPi, θα παρέχονται και οι προαναφερθείσες λειτουργίες. Στόχος είναι η πλήρης υλοποίηση του OpenMP v.4.0 και των νέων οδηγιών που παρέχει.

Βιβλιογραφία

[1] *GNU libgomp*. https://gcc.gnu.org/onlinedocs/libgomp/.

[2] V. Dimakopoulos G. Philos and P. Hadjidoukas. *A Runtime Architecture for Ubiquitous Support of OpenMP*. in Proc. ISPDC 2008, 7th International Symposium on Parallel and Distributed Computing. July 2008.

[3] Steven Huss-Lederman David Walker Jack Dongarra Marc Snir, Steve Otto. *MPI: The Complete Reference*. 1996.

[4] N. McDonnell Mark Bull, Fiona Reid. *EPCC OpenMP micro-benchmark suite*. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite.

[5] V.V. Dimakopoulos P.E. Hadjidoukas, G.Ch. Philos. *"Exploiting fine-grain thread parallelism on multicore architectures"*. Scientific Programming, Vol. 17, No. 4, Nov. 2009.

[6] Vassilios V. Dimakopoulos Spiros N. Agathos, Panagiotis E. Hadjidoukas. *Design and Implementation of OpenMP Tasks in the OMPi Compiler*. Septempber 2011. http://paragroup.cs.uoi.gr/Publications/139pci2011a.pdf.

[7] OpenMP Architecture Review Board V.4.0. July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[8] E. Leontiadis V.V. Dimakopoulos and G. Tzoumas. *A Portable C Compiler for OpenMP*. October 2014. http://www.cs.uoi.gr/~ompi/download.html.