Πτυχιακή Εργασία

Μουρελής Άγγελος

Σεπτέμβριος 2013

Ανάλυση ροής δεδομένων και autoscoping στον παραλληλοποιητικό μεταφραστή OMPi

Επιβλέπων: Βασίλειος Δημακόπουλος

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Η νέα εποχή

Όταν το 1965 ο συνιδρυτής της εταιρείας κατασκευής μικροεπεξεργαστών Intel, Gordon Moore, προέβλεψε ότι ο αριθμός των τρανζίστορ σε ένα μικροεπεξεργαστή θα διπλασιάζεται κάθε 18 μήνες, ίσως ακόμη και ο ίδιος να μην περίμενε ότι η πρόβλεψή του αυτή, θα επαληθευόταν πρακτικά σε τέτοιο σημείο, ώστε να ονομαστεί 'Νόμος του Moore'.

Πράγματι, από την ημέρα διατύπωσής του και μετά, ο νόμος αυτός επαληθεύεται ομαλά μέχρι το 2004: τα τρανζίστορ ανά μικροεπεξεργαστή αυξάνονται σύμφωνα με την πρόβλεψη του Moore. Το 2004 όμως, η εταιρεία Intel, υποχρεώθηκε να πάρει μια απόφαση, την οποία ήδη είχαν πάρει άλλες εταιρείες κατεσκευής, και η οποία σηματοδότησε την μεταστροφή στην παραγωγή πλακετών με περισσότερους του ενός μικροεπεξεργαστές (multi-core chips): ανακοίνωσε ότι σταματάει την παραγωγή του μικροεπεξεργαστή Tejas, ο οποίος επρόκειτο να διαδεχθεί τον Pentium 4. Υποβίβασε δηλαδή, τις διαδικασίες ανάπτυξης και παραγωγής, μικροεπεξεργαστών με μεγαλύτερο αριθμό τρανζίστορ, και έφερε στο προσκήνιο, την ανάπτυξη επεξεργαστικών συστημάτων, τα οποία αποτελούνται από δύο ή περισσότερες ανεξάρτητες επεξεργαστικές μονάδες.

Η αιτία δεν είναι άλλη από τους φυσικούς περιορισμούς που υπεισέρχονται στην παραγωγική αυτή διαδικασία. Η τεχνολογική αυτή βελτίωση, απαιτεί σμίκρυνση των τρανζίστορ, επομένως αύξηση του αριθμού τους και κατ' επέκτασιν αύξηση της συχνότητας στην οποία λειτουργούν. Η συνεχής παραγωγή επεξεργαστών ικανών να εκτελούν όλο και περισσότερες εντολές ανά χρονική μονάδα, συναντάει δύο σημαντικούς φραγμούς:

Πρώτον, όπως ήταν αναμενόμενο, η μείωση του μεγέθους των τρανζίστορ δεν θα μπορούσε να συνεχιστεί επ' άπειρον, δεδομένου ότι το μέγεθος των επεξεργαστών, έχει την τάση να παραμένει σταθερό, για λόγους παραγωγικού κόστους, αλλά και των ολοένα μικρότερων ηλεκτρονικών συσκευών στα οποία πρέπει να ενσωματωθούν.

Δεύτερον, η αύξηση της συχνότητας, οδηγεί σε αύξηση της κατανάλωσης από τον επεξεργαστή, με αποτέλεσμα κάτι τέτοιο να καθιστά την παραγωγή επεξεργαστών με πολύ υψηλή συχνότητα ασύμφορο.

Οι δύο παραπάνω λόγοι, κάνουν φανερό ότι το να βασίζεται κανείς, πλέον, στην κατασκευή ισχυρότερων επεξεργαστών δεν αρκεί. Έτσι το βάρος πέφτει στην αρχιτεκτονική και την οργάνωση τους, με υπερισχύουσα τεχνική αυτήν της παραλληλοποίησης σε επίπεδο επεξεργαστών. Αντί λοιπόν για έναν πανίσχυρο επεξεργαστή, διαμοιράζουμε την επεξεργαστική ισχύ σε περισσότερους του ενός, οι οποίοι καλούνται να συνεργαστούν.

Κάτι τέτοιο, βέβαια, μαζί με τους ορίζοντες που επεκτείνει, δημιουργεί την απαίτηση για τον προγραμματισμό παραλληλοποιημένων προγραμμάτων, τομέας ο οποίος, εφ' όσον παρέμενε μέχρι τότε

αδρανής, βρίσκεται ακόμα στην ανάπτυξή του.

1.2 Παράλληλα συστήματα

Ο κύριος διαχωρισμός, με βάση την αρχιτεκτονική των παράλληλων συστημάτων, γίνεται μεταξύ συστημάτων κοινής και κατανεμημένης μνήμης. Στα παράλληλα προγράμματα, ο προγραμματιστής σχεδιάζει και υλοποιεί την λύση ενός προβλήματος, μεταβάλλοντας τις μεθόδους και τις τεχνικές του ανάλογα με το σύστημα πάνω στο οποίο θα εφαρμοστεί η υλοποίησή του. Το τελευταίο, καθιστά την μελέτη, την ανάλυση και την ανάπτυξη των παράλληλων αρχιτεκτονικών άρρηκτα συνδεδεμένες με την εξέλιξη του παράλληλου προγραμματισμού.

1.2.1 Συστήματα κοινόχρηστης μνήμης

Στα συστήματα κοινόχρηστης μνήμης, οι επεξεργαστικές μονάδες, συνήθως διασυνδεδεμένες μεταξύ τους με δίαυλο, μοιράζονται κοινό σύστημα μνήμης, το οποίο είναι προσβάσιμο σε όλους. Έτσι, οι μονάδες αυτές έχουν πρόσβαση στα ίδια δεδομένα, και ενδεχόμενες μεταβολές σε αυτά είναι καθολικά ορατές.

Τέτοια συστήματα, έχουν ως όριο αριθμού επεξεργαστών, το όριο εκείνο που επιτάσσει η χωρητικότητα του διαύλου, καθώς αυξημένη κίνηση στον δίαυλο θα είχε ως αποτέλεσμα τον κορεσμό της απόδοσης του συστήματος.

Στα πλεονεκτήματά του, εντάσσεται η απλότητά της δομής του, αλλά και το γεγονός, ότι υπό συνθήκες ομαλής λειτουργίας του συστήματος, οι επεξεργαστικές οντότητες δύνανται να έχουν απλή, αποτελεσματική και ενδεχομένως αποδοτική πρόσβαση στα κοινά δεδομένα.

1.2.2 Συστήματα κατανεμημένης μνήμης

Εδώ, η κάθε επεξεργαστική οντότητα διαθέτει μια ιδιωτική μνήμη στην οποία και έχει άμεση πρόσβαση. Οι οντότητες αυτές συνδέονται και επικοινωνούν μεταξύ τους, για ενδεχόμενες αποστολές και παραλαβές δεδομένων. Στην περίπτωση δηλαδή, στην οποία μια οντότητα Α χρειάζεται να προσπελάσει κάποιο δεδομένο το οποίο δεν βρίσκεται στην ιδιωτική της μνήμη, στέλνει κατάλληλη αίτηση στο δίκτυο και επικοινωνεί με την οντότητα Β, η οποία βρίσκει το δεδομένο αυτό στην δική της μνήμη. Έτσι, η οντότητα Α αποκτά έμμεση πρόσβαση σε αυτό.

Το βασικό πρόβλημα που αντιμετωπίζουν δίκτυα αυτής της μορφής, είναι ότι η απομακρυσμένη προσπέλαση των δεδομένων ενδέχεται να είναι αργή, και σε περιπτώσεις κατά τις οποίες αυτό

συμβαίνει συχνά, προκαλεί συμφόρηση στο δίκτυο.

Βέβαια, καταργείται το φράγμα των εν δυνάμει συνδεδεμένων μονάδων στο δίκτυο που αναφέρθηκε στα συστήματα κοινόχρηστης μνήμης, κάτι το οποίο μας δίνει τη δυνατότητα κατασκευής οσοδήποτε μεγάλων δικτύων επεξεργαστικών οντοτήτων.

Στην πράξη, αυτό που συναντάται κατά κόρον σήμερα είναι ένα συνονθύλευμα των δύο: συστήματα κοινής μνήμης, τα οποία αποτελούν τις επεξεργαστικές οντότητες για συστήματα κατανεμημένης μνήμης. Έτσι, οι μονάδες ομαδοποιούνται, με αποτέλεσμα οι ομάδες αυτές να διαθέτουν κοινή μνήμη στο εσωτερικό τους, αλλά δύνανται να επικοινωνήσουν με άλλες ομάδες για την ανταλλαγή των κατανεμημένα διαμοιρασμένων δεδομένων.

1.3 Παράλληλος προγραμματισμός

Για την κατασκευή προγραμμάτων, τα οποία θα αναλαμβάνουν τον διαχωρισμό και την κατανομή των επιμέρους εργασιών, την επικοινωνία και τον συγχρονισμό των (περισσοτέρων του ενός) παράλληλων εκτελεστικών οντοτήτων, αλλά και την ασφαλή προσπέλαση των δεδομένων από τις παράλληλες αυτές οντότητες, έχουν κατασκευαστεί διάφορες προγραμματιστικές διεπαφές, κάθε μια με τα δικά της χαρακτηριστικά. Βασικότερες εξ' αυτών είναι το μοντέλο νημάτων Pthreads, το MPI, το OpenMP και το OpenCL. Η επιλογή διεπαφής γίνεται με βάση το είδος του παράλληλου συστήματος το οποίο έχουμε στη διάθεσή μας.

Για τον προγραμματισμό σε συστήματα κοινής μνήμης, επιλέγονται συνήθως πλατφόρμες που κάνουν χρήση των νημάτων ως επεξεργαστικές οντότητες, καθώς αυτά μοιράζονται κοινό χώρο διευθύνσεων, υπό την προϋπόθεση ότι έχουν κοινή διεργασία ως δημιουργό. Άρα, μοντέλα διεπαφής όπως το OpenMP και τα Pthreads αποτελούν συνήθεις επιλογές.

Για τον προγραμματισμό συστημάτων κατανεμημένης μνήμης, χρησιμοποιείται κυρίως το μοντέλο ανταλλαγής μηνυμάτων MPI, το οποίο παρέχει διαδικασίες που ρυθμίζουν την αποτελεσματική και αποδοτική επικοινωνία που απαιτείται, για την ανταλλαγή δεδομένων αλλά και την χρονοδρομολόγηση των οντοτήτων.

1.4 Αντικείμενο πτυχιακής εργασίας

Στην εργασία αυτή, στόχος είναι η επέκταση των δυνατοτήτων του παραλληλοποιητικού μεταφραστή OMPi (Κεφάλαιο 3) και πιο συγκεκριμένα η ενσωμάτωση λειτουργικότητας για ανάλυση ροής δεδομένων (data flow analysis) (Κεφάλαιο 4) , και του autoscoping (Κεφάλαιο 5).

Ο OMPi, είναι ένας μεταφραστής source-to-source, ο οποίος δέχεται ως είσοδο ένα σειριακό πρόγραμμα, το οποίο περιέχει εντολές του μοντέλου OpenMP (Κεφάλαιο 2), και εξάγει ένα πρόγραμμα σε C, παραλληλοποιημένο αποδοτικά και έτοιμο να μεταφραστεί στο εκάστοτε σύστημα από τον τοπικό μεταφραστή.

Το autoscoping, είναι η δυνατότητα, να παρθεί η απόφαση, για το αν μια μεταβλητή ενδέχεται να προσπελαστεί ταυτοχρόνως από δύο ή περισσότερες επεξεργαστικές οντότητες, και για το πως αυτή θα πρέπει να αντιμετωπιστεί, από τον μεταφραστή και όχι από τον χρήστη. Είναι ένα εργαλείο στην λογική της φιλικής προς τον χρήστη παραλληλοποίησης, αλλά και ενδεχομένως ένα εργαλείο για την διευκόλυνση και επιτάχυνση της διαδικασίας κατά την οποία παραλληλοποιείται ένα πρόγραμμα.

Για να καταστεί το παραπάνω δυνατό, γίνεται η σύνδεση του autoscoping, με το πρόβλημα εγγραφής/ανάγνωσης, που επιλύεται μέσω της ανάλυσης ροής δεδομένων (Κεφάλαιο 4).

Στόχος λοιπόν, της συγκεκριμένης πτυχιακής εργασίας, είναι:

- Προσθήκη δυνατότητας ανάλυσης ροής δεδομένων (data flow analysis) στον OMPi
- Εφαρμογή της ανάλυσης ροής δεδομένων, για την προσθήκη δυνατότητας autoscoping στον OMPi.

1.5 Δομή πτυχιακής εργασίας

Συνοπτικά, τα κεφάλαια της εργασίας αυτής, παρουσιάζονται ως εξής:

- Κεφάλαιο 2: Παρουσιάζεται το μοντέλο προγραμματιστικής διεπαφής OpenMP, καθώς και αναλύονται οι εντολές που το διέπουν. Ιδιαίτερη έμφαση δίνεται στον τρόπο που το OpenMP αντιμετωπίζει τις μεταβλητές, που βρίσκονται σε παράλληλες περιοχές.
- Κεφάλαιο 3: Περιγράφεται ο παραλληλοποιητικός μεταφραστής OMPi και αναλύονται τα κομμάτια του, τα οποία συνδέονται με τη συγκεκριμένη εργασία.
- Κεφάλαιο 4: Προσεγγίζεται θεωρητικά η ανάλυση ροής δεδομένων, παρουσιάζεται το πρόβλημα εγγραφής/ανάγνωσης που επιλύεται δια μέσου της μεθόδου αυτής,καθώς και υλοποιείται στον OMPi ο γενικότερος μηχανισμός.
- Κεφάλαιο 5: Αναλύεται η ιδέα του autoscoping, η δυνατότητες και οι περιορισμοί του και παρουσιάζεται η υλοποίησή του στον OMPi.

Κεφάλαιο 2

Το πρότυπο OpenMP

2.1 Εισαγωγή στο OpenMP

Το OpenMP (Open Multi-Processing) , είναι μια προγραμματιστική διεπαφή, η οποία έχει ως σκοπό την απλοποίηση της διαδικασίας παραλληλοποίησης προγραμμάτων σε συστήματα κοινής μνήμης, παρέχοντας ένα ευέλικτο, εύχρηστο και μεταφέρσιμο μοντέλο στον χρήστη.

Ουσιαστικά, το OpenMP παρέχει την δυνατότητα μιας υψηλού επιπέδου παραλληλοποίησης, αποκρύπτοντας από τον χρήστη διαδικασίες, τις οποίες ενώ αυτός αγνοεί, αυτές εκτελούνται αποδοτικά και ενδεχομένως βέλτιστα.

Η παραλληλοποίηση ενός προγράμματος μέσω OpenMP αποτελείται ουσιαστικά από 2 στάδια:

- Ο χρήστης προσθέτει στο πρόγραμμά του τα λεγόμενα **pragmas** ή **directives**. Πρόκειται για γενικές οδηγίες, όπως τα κομμάτια τα οποία θα παραλληλοποιηθούν, και ενδεχομένως κάποιες πιο αναλυτικές οδηγίες, για το πως ο μεταφραστής θα πρέπει να χειριστεί τα εκάστοτε δεδομένα και άλλα γενικά χαρακτηριστικά της επικείμενης παραλληλοποίησης.

  Επίσης, προσθέτει (προαιρετικά), κλήσεις σε **συναρτήσεις βιβλιοθήκης του OpenMP**, με τις οποίες δύναται να δώσει κάποιες πιο αναλυτικές εντολές για την παραλληλοποίηση, όπως να μεταβάλλει κατά βούληση τον αριθμό των νημάτων, ή να τα συγχρονίζει χειροκίνητα.

- Ο μεταφραστής, με βάση τα παραπάνω, παραλληλοποιεί το πρόγραμμα, σύμφωνα με τις οδηγίες του χρήστη, αποκρύπτοντας, ενδεχομένως, από αυτόν αρκετές λεπτομέρειες της διαδικασίας.

Το μοντέλο αυτό, βέβαια, υπόκειται στους περιορισμούς της αυτόματης παραλληλοποίησης, καθώς, ένα πρόγραμμα που παραλληλοποιείται αυτόματα, δεν πετυχαίνει βέλτιστη απόδοση. Παρ' όλα αυτά, αποτελεί ένα εργαλείο ανάπτυξης στον τομέα του παράλληλου προγραμματισμού, με παροχή σύνθετων δυνατοτήτων όσον αφορά την χρονοδρομολόγηση και τον καταμερισμό των εργασιών.

2.2 Προγραμματισμός στο OpenMP

Όπως αναφέρθηκε παραπάνω, θεμελιώδεις εντολές για τον προγραμματισμό στο συγκεκριμένο

μοντέλο, αποτελούν τα pragmas ή directives, καθώς και οι κλήσεις βιβλιοθήκης.

```
int main ( ) {
        int tid;

        #pragma omp parallel
        {
                tid = omp_get_thread_num();
                printf("Hello World from thread = %d\n", tid);
        }
}
```

Στο παραπάνω παράδειγμα, διακρίνουμε το directive (ξεκινάνε με #), καθώς και την κλήση βιβλιοθήκης OpenMP `omp_get_thread_num`, η οποία επιστρέφει το μοναδικό αναγνωριστικό για κάθε νήμα.

Το συγκεκριμένο directive, που αποτελεί την βάση του OpenMP, δίνει την δυνατότητα στον χρήστη να ορίσει μια περιοχή, η οποία θα παραλληλοποιηθεί. Το συγκεκριμένο παράδειγμα, παρά την απλοϊκότητα του, είναι χαρακτηριστικό του πόσο εύκολη μπορεί να γίνει η διαδικασία παραλληλοποίησης.

Η λογική που το διέπει είναι η εξής: εισέρχεται στην κεντρική συνάρτηση το μοναδικό νήμα σειριακής εκτέλεσης, και συναντάει το directive (`#pragma omp parallel`). Τότε, δημιουργεί μια παράλληλη ομάδα. Εφ' όσον ο χρήστης δεν έχει ορίσει τον επιθυμητό αριθμό νημάτων, η απόφαση αυτή παίρνεται από τον μεταφραστή (συνήθης επιλογή είναι ο αριθμός των πυρήνων, εάν πρόκειται για κάποιο πολυπύρηνο σύστημα). Το νήμα αυτό, εντάσσει και τον εαυτό του ως μέλος της παράλληλης αυτής ομάδας, και εκτελεί τις εντολές που περιέχονται εντός της παράλληλης περιοχής. Αξίζει να σημειωθεί ότι μετά το πέρας της παράλληλης περιοχής, τα νήματα συγχρονίζονται: όλα τα νήματα σταματάνε στο σημείο αυτό, έως ότου το νήμα δημιουργός καταστρέψει την ομάδα που δημιούργησε.

2.3 Εντολές OpenMP

Στην ενότητα αυτή παρουσιάζεται μια πιο ενδελεχής ανάλυση τον εντολών που διέπουν το OpenMP, και συγκεκριμένα την έκδοση 3.0 [1], με σκοπό να δοθεί μια ακριβής εικόνα των δυνατοτήτων που εκείνο παρέχει. Οι εντολές που παρουσιάζονται στην εργασία αυτή αφορούν τις γλώσσες C και C++ (και όχι την Fortran), καθώς ο OMPi, πάνω στον οποίον δομήθηκε αυτή η εργασία, παραλληλοποιεί προγράμματα γραμμένα σε γλώσσα C. Παρ' όλα αυτά, κύριος στόχος είναι η εμβάθυνση στην λογική του OpenMP, εξάλλου οι συντακτικές διαφορές είναι μικρές και η μετάβαση από τη μία γλώσσα στην

άλλη, μάλλον τετριμμένη.

Ιδιαίτερη έμφαση δίνεται στον χειρισμό των μεταβλητών που προσπελάζονται εντός κάποιας παράλληλης περιοχής, το οποίο άλλωστε αποτελεί πυρηνικό θέμα της συγκεκριμένης εργασίας.

2.3.1 Γενικοί συντακτικοί κανόνες

Τα directives του OpenMP, ξεκινάνε με το διακριτικό `#pragma omp`, και δίνουν την δυνατότητα στον χρήστη να καθορίσει, τον βαθμό στον οποίον θα προσδιορίσει τις λεπτομέρειες της παραλληλοποίησης.

Το παραπάνω διακριτικό, ακολουθείται είτε από μια εντολή συγχρονισμού, είτε από μια εντολή παραλληλοποίησης.

Ακολουθούν οι συνθήκες που διέπουν την συγκεκριμένη εντολή, οι οποίες είναι εν γένει προαιρετικές. Έτσι, ο χρήστης, επιλέγοντας το ποιες οδηγίες θα παραμετροποιήσει χειροκίνητα και το ποιες θα αναθέσει στο ίδιο το OpenMP, καθορίζει στην ουσία το πόσο υψηλού επιπέδου παράλληλο προγραμματισμό θα χρησιμοποιήσει.

2.3.2 Παράλληλες περιοχές

Όπως είδαμε στην ενότητα 2.2, θεμέλιο της σύνταξης ενός προγράμματος με OpenMP, είναι οι παράλληλες περιοχές, οι οποίες ξεκινάνε με το διακριτικό `#pragma omp parallel` και περικλείονται μεταξύ δύο αγκυλών. Ένα νήμα το οποίο συναντά μια τέτοια περιοχή, δημιουργεί (το λεγόμενο νήμα master), μια παράλληλη ομάδα. Όπως αναφέρθηκε και στο παράδειγμα, μετά το πέρας μιας τέτοιας περιοχής υπονοείται συγχρονισμός μεταξύ των συμμετεχόντων σε αυτήν νημάτων.

Γενικεύοντας την παραπάνω συμπεριφορά, είναι δυνατή η χρήση εμφωλευμένων παράλληλων περιοχών, οι οποίες δίνουν την δυνατότητα στον χρήστη να χειριστεί παράλληλες υποομάδες νημάτων, με απλό συντακτικά τρόπο. Στην περίπτωση, εμφωλευμένης παράλληλης περιοχής 2ου επιπέδου, η συμπεριφορά των νημάτων σκιαγραφείται ως εξής: στην παράλληλη περιοχή 1ου επιπέδου, το κεντρικό νήμα δημιουργεί μια παράλληλη ομάδα και τίθεται master της. Κατά την προσπέλαση του εμφωλευμένου directive, κάθε ένα από τα νήματα της ομάδας αυτής, δημιουργεί μια υποομάδα νημάτων, της οποίας και τίθεται master.

Είναι εύκολο να δούμε ότι το βάθος φωλιάσματος δεν φράσσεται σε θεωρητικό επίπεδο. Στην πράξη όμως, η δημιουργία ομάδων μεγέθους M, με βάθος φωλιάσματος N, θα έχει ως αποτέλεσμα συνολικά M^N νήματα: το φράγμα επιτάσσεται από τις διαθέσιμες επεξεργαστικές οντότητες.

Η δυνατότητες παραμετροποίησης του συγκεκριμένου directive, είναι οι εξής:

- ***if(συνθήκη)***: ο χρήστης προσδιορίζει μια συνθήκη απόφασης για το αν η περιοχή θα είναι παράλληλη ή όχι.
- ***num_threads( )***: αν η παράμετρος αυτή υπάρχει, ο χρήστης καθορίζει τον αριθμό των νημάτων που θα δημιουργηθούν στην περιοχή αυτή. Διαφορετικά ο μεταφραστής παίρνει αυτήν την απόφαση, με συνηθισμένη, όπως προαναφέραμε, επιλογή, τον αριθμό των διαθέσιμων επεξεργαστικών οντοτήτων.
- ***Αντιμετώπιση μεταβλητών*:** Παρουσιάζεται αναλυτικά στην ενότητα 2.4.

2.3.3 Καταμερισμός εργασίας

Τα directives αυτού του τύπου, καθορίζουν τον τρόπο με τον οποίον θα διαμοιραστεί στα διαθέσιμα νήματα ο φόρτος και είναι τα εξής:

- ***#pragma omp for:*** Ακολουθείται αυστηρά από έναν βρόγχο τύπου for, και υποδηλώνει ότι οι επανάληψεις του βρόγχου, θα διαμοιραστούν από τον μεταφραστή στα διαθέσιμα νήματα. Η παραμετροποίηση του είναι δυνατή ως εξής:
  - *collapse:* Δίνει την δυνατότητα στον χρήστη να μοιράσει τις επαναλήψεις εμφωλευμένων βρόγχων for, χωρίς την χρήση εμφωλευμένης παραλληλοποίησης.
  - *Αντιμετώπιση μεταβλητών:* Αναλυτικά στην ενότητα 2.4.
  - *schedule:* Αναλυτικά στην ενότητα 2.3.5.
  - *nowait:* Η οδηγία for, υπονοεί συγχρονισμό τον νημάτων μετά το πέρας της. Με την παράμετρο αυτήν, ο συγχρονισμός παραλείπεται.
- ***#pragma omp sections:*** Καθορίζει εργασίες, οι οποίες διαμοιράζονται μεταξύ των διαθέσιμων νημάτων.
- ***#pragma omp task:*** Ορίζει μια εργασία ευέλικτου τύπου, η οποία δύναται να εκτελεστεί παράλληλα με κώδικα ο οποίος δεν αποτελεί task. Είναι δυνατή η εναλλαγή εκτέλεσης ενός task μεταξύ των νημάτων, κάτι το οποίο σημαίνει ότι ένα task μπορεί να εκτελεστεί τμηματικά από περισσότερα του ενός νήματα.

2.3.4 Σειριακές περιοχές

Το OpenMP, μας δίνει τη δυνατότητα να εισάγουμε σειριακές περιοχές εν μέσω παραλληλίας. Οι επιλογές μας είναι δύο: είτε να ορίσουμε μια σειριακή περιοχή η οποία θα εκτελεστεί από ένα τυχαίο νήμα, είτε να επιλέξουμε να εκτελεστεί από το νήμα δημιουργό της παράλληλης ομάδας(master thread).

- ***#pragma omp master*:** Το νήμα δημιουργός νήμα εκτελεί την προσδιορισμένη περιοχή.
- ***#pragma omp single*:** Το πρώτο διαθέσιμο νήμα εκτελεί την περιοχή αυτή. Το directive αυτού του τύπου, υπονοεί συγχρονισμό των νημάτων μετά το πέρας της περιοχής. Αυτό, αποφεύγεται αν γίνει χρήση της παραμέτρου **nowait.**

Η κομβική διαφορά των δύο μεθόδων, είναι ότι στο single, εν γένει, το ποιο νήμα θα εκτελέσει την περιοχή είναι απρόβλεπτο. Το master μας διασφαλίζει ότι δύο διαφορετικές σειριακές περιοχές αυτού του τύπου, θα εκτελεστούν από το ίδιο νήμα. Αυτό βέβαια, σημαίνει ότι διαθέσιμα νήματα, ενδεχομένως να παρακάμψουν την περιοχή αυτή.

2.3.5 Χρονοδρομολόγηση

Όπως είδαμε και στην εντολή pragma omp for, υπάρχουν περιπτώσεις κατά τις οποίες είναι δυνατή η διαμόρφωση του χρονοπρογράμματος (schedule), το οποίο θα ακολουθήσουν τα νήματα. Ο χρήστης προσδιορίζει τον τρόπο διαμόρφωσης του χρονοπρογράμματος, καθώς και το μέγεθος του κόκκου (chunk), το οποίο καθορίζει το μέγεθος των εργασιών, που θα διαμοιραστούν μεταξύ των νημάτων. Στον βρόγχο for, οι επαναλήψεις διαμερίζονται σε εργασίες στο μέγεθος του κόκκου.

Οι διαθέσιμοι τρόποι χρονισμού είναι οι εξής:

- ***static:*** Οι εργασίες διαμοιράζονται κυκλικά μεταξύ των νημάτων. Αν ο κόκκος δεν προσδιορίζεται από τον χρήστη, η προεπιλεγμένη επιλογή είναι [#εργασιών]/[#νημάτων].
- ***dynamic:*** Εδώ, το κάθε νήμα, αφού εκτελέσει την εργασία που του έχει ανατεθεί, αναλαμβάνει την επόμενη προς εκτέλεση εργασία δυναμικά.
- ***guided:*** Ισχύει η ίδια δυναμική μέθοδος χρονισμού με το dynamic, μόνο που το μέγεθος του κόκκου μειώνεται εκθετικά.
-

2.3.6 Συγχρονισμός

Το OpenMP παρέχει την δυνατότητα στον χρήστη να συγχρονίσει τα νήματα, έτσι ώστε να εξασφαλίσει συνέπεια ως προς τα δεδομένα με αποδοτικό τρόπο. Τέτοιες εντολές είναι:

- ***#pragma omp barrier:*** Σημείο συγχρονισμού των νημάτων. Όλα τα νήματα σταματάνε στο barrier εώς ότου προσέλθει και το τελευταίο.
- ***#pragma omp atomic:*** Προσδιορίζει μια θέση μνήμης η οποία θα πρέπει να ανανεωθεί ατομικά. Υποστηρίζει επομένως απλές εντολές, οι οποίες επηρεάζουν κοινόχρηστες μεταβλητές.
- ***#pragma omp critical:*** Καθορίζει μια ολόκληρη περιοχή, στην οποία, εξασφαλίζεται ότι οι εντολές που διενεργούν επάνω σε κοινόχρηστα δεδομένα, θα γίνουν ομαλά, ως προς την συνέπεια τους. Είναι πολύ πιο εύχρηστη από το atomic, έχοντας βέβαια μεγαλύτερη χρονική επιβάρυνση.
- ***#pragma omp flush:*** Κατά το flush, οι εκκρεμείς αναγνώσεις και εγγραφές ολοκληρώνονται. Εγγραφές ή αναγνώσεις οι οποίες ακολουθούν το flush, περιμένουν το πέρας του. Μας εξασφαλίζει δηλαδή, μια συνεπή εικόνα της μνήμης.
- ***#pragma omp ordered:*** Οι εντολές που βρίσκονται μέσα στο block πρέπει να εκτελεστούν ακολουθιακά, με την σειρά που συναντώνται κατά την ανάγνωση του κώδικα.

2.4 Μεταβλητές εντός παράλληλων περιοχών

Κεντρικό ζήτημα για την συγκεκριμένη εργασία, είναι το πως το OpenMP αντιμετωπίζει τις μεταβλητές που συναντώνται εντός παράλληλων περιοχών. Οι κατηγορίες στις οποίες αυτές χωρίζονται, με βάση την χρήση τους εντός της παράλληλης περιοχής είναι:

- ***shared:*** Η μεταβλητή είναι κοινόχρηστη μεταξύ των νημάτων που συμμετέχουν στην συγκεκριμένη παράλληλη περιοχή.
- ***private:*** Κάθε συμμετέχον νήμα έχει το δικό του αντίγραφο της μεταβλητής, η οποία είναι μη αρχικοποιημένη.
- ***firstprivate:*** Εδώ το κάθε νήμα έχει πάλι το δικό του αντίγραφο, μόνο που το αντίγραφο αυτό αρχικοποιείται στην τιμή την οποία είχε πριν την παράλληλη περιοχή.
- ***lastprivate:*** Ειδική περίπτωση που συνάδει με τις δομές for και sections. Όποιο νήμα εκτελέσει την τελευταία επανάληψη του βρόγχου (ή αντίστοιχα το τελευταίο section), δίνει την τιμή που έχει το αντίγραφό του, στην πρωτότυπη μεταβλητή.
- ***reduction:*** Ειδική περίπτωση κατά την οποία η συγκεκριμένη μεταβλητή υπόκειται σε μια συγκεκριμένη πράξη, στο τέλος μια παράλληλης περιοχής. Η πράξη αυτή περιγράφεται από

έναν εκ των τελεστών +, *, -, &, |, ^, &&, ||.

Αν κάποια μεταβλητή δεν δηλωθεί, το OpenMP της κατηγοριοποιεί σαν shared.

Ο προσδιορισμός της κατηγοριοποίησης (scoping) των μεταβλητών, αποτελεί σημαντικό προγραμματιστικό ζήτημα, καθώς ενδεχόμενο απερίσκεπτο scoping μπορεί να έχει σοβαρές επιδράσεις στο παραλληλοποιημένο πρόγραμμα.

Μία shared μεταβλητή, είναι ευάλωτη σε ασυνέπειες μνήμης, και είναι ευθύνη του προγραμματιστή να την προστατέψει, ενδεχομένως με κάποια εντολή συγχρονισμού. Από την άλλη, κατάχρηση του συγχρονισμού μπορεί να οδηγήσει με μεγάλη καθυστέρηση. Μία private μεταβλητή, όπως είδαμε, δεν είναι αρχικοποιημένη. Έτσι, αν κάποιο νήμα διαβάσει την τιμή της, πριν την καθορίσει το ίδιο, οδηγούμαστε σε απρόβλεπτα αποτελέσματα. Επίσης, μια firstprivate μεταβλητή, στην περίπτωση που η αρχικοποιημένη τιμή δεν διαβάζεται από κανένα νήμα, μας οδηγεί σε σπατάλη χώρου, αλλά και χρόνου.

Έτσι λοιπόν, διακρίνουμε ότι το scoping των μεταβλητών είναι ένα σύνθετο ζήτημα και πολλά από τα παραπάνω θέματα θα μας απασχολήσουν στο Κεφάλαιο 5, όπου θα αναλύσουμε την περίπτωση κατά την οποία ο μεταφραστής καλείται να αποφανθεί για την κατηγοριοποίηση των μεταβλητών (autoscoping).

2.5 Συναρτήσεις περιβάλλοντος και χρόνου εκτέλεσης

Οι συναρτήσεις αυτές μας επιτρέπουν να παραμετροποιήσουμε, ενδεχομένως δυναμικά, την παραλληλοποίηση, καθώς και να λάβουμε πληροφορίες για το προγραμματιστικό περιβάλλον.

Οι βασικότερες εξ' αυτών είναι:

- ***omp_set_num_threads(int):*** Η συνάρτηση αυτή καθορίζει τον αριθμό τον νημάτων τα οποία θα εκτελέσουν την επόμενη παράλληλη περιοχή. Απενεργοποιείται από ενδεχόμενη χρήση της παραμέτρου num_threads.
- ***omp_get_thread_num( ):*** Στο νήμα που την καλεί, επιστρέφει τον αριθμό των νημάτων που βρίσκονται στην παράλληλη ομάδα του.
- ***omp_set_nested( ):*** Με την συνάρτηση αυτή, ενεργοποιούμε ή απενεργοποιούμε την δυνατότητα χρήσης εμφωλευμένων παράλληλων περιοχών. Αν η δυνατότητα είναι απενεργοποιημένη, οι περιοχές αυτές εκτελούνται σειριακά.
- ***omp_in_parallel( ):*** Απαντά για το αν βρισκόμαστε εντός παράλληλης περιοχής ή όχι.
- ***omp_num_procs( ):*** Επιστρέφει τον αριθμό των επεξεργαστικών μονάδων του συστήματος.

2.6 Κλειδαριές

Πέρα από τα atomic και critical, για την διασφάλιση της ακεραιότητας των δεδομένων, το OpenMP μας παρέχει την δυνατότητα προγραμματισμού χαμηλότερου επιπέδου, με την χρήση κλειδαριών (locks). Οι κλειδαριές αυτές αποτελούν ξεχωριστό τύπο δεδομένων (omp_lock_t, omp_nest_lock_t) και ο χειρισμός τους γίνεται με τις παρακάτω συναρτήσεις:

- ***omp_init_lock( ):*** Αρχικοποίηση της κλειδαριάς.
- ***omp_set_lock( ):*** Το νημα αιτείται να πάρει το κλειδί. Αν είναι ελεύθερο το παίρνει, αν όχι, περιμένει αδρανές εώς ότου το πάρει.
- ***omp_unset_lock( ):*** Το νήμα ελευθερώνει το κλειδί. Αν υπάρχει κάποιο άλλο νήμα που το περιμένει, το κλειδί μεταβιβάζεται σ' εκείνο.
- ***omp_test_lock( ):*** Το νήμα αιτείται του κλειδιού, αλλά στην περίπτωση που αυτό δεν είναι διαθέσιμο, δεν αδρανοποιείται.

Αντίστοιχες συναρτήσεις υπάρχουν και για τα nest_locks, κλειδαριές δηλαδή που παρέχει το OpenMP για την διαφύλαξη των δεδομένων σε εμφωλευμένο παραλληλισμό.

Κεφάλαιο 3

Ο παραλληλοποιητικός μεταφραστής OMPi

3.1 Γενική περιγραφή του OMPi

Ο OMPi [2] είναι ένας μεταφραστής πηγαίου αρχείου σε πηγαίο αρχείο (source-to-source). Δέχεται ως είσοδο ένα πρόγραμμα γλώσσας C, εμπλουτισμένο με OpenMP directives, και εξάγει ένα πρόγραμμα σε C, έτοιμο να μεταφραστεί από τον ενεργό C μεταφραστή του φιλοξενούντος συστήματος.

Το εξαγόμενο πρόγραμμα, περιέχει κλήσεις στο περιβάλλον χρόνου εκτέλεσης (runtime) του OMPi. Ο μεταφραστής που θα αναλάβει να μεταφράσει το εξαγόμενο πρόγραμμα, θα συνδέσει τις κλήσεις αυτές με την κατάλληλη βιβλιοθήκη που έχει επιλεγεί.

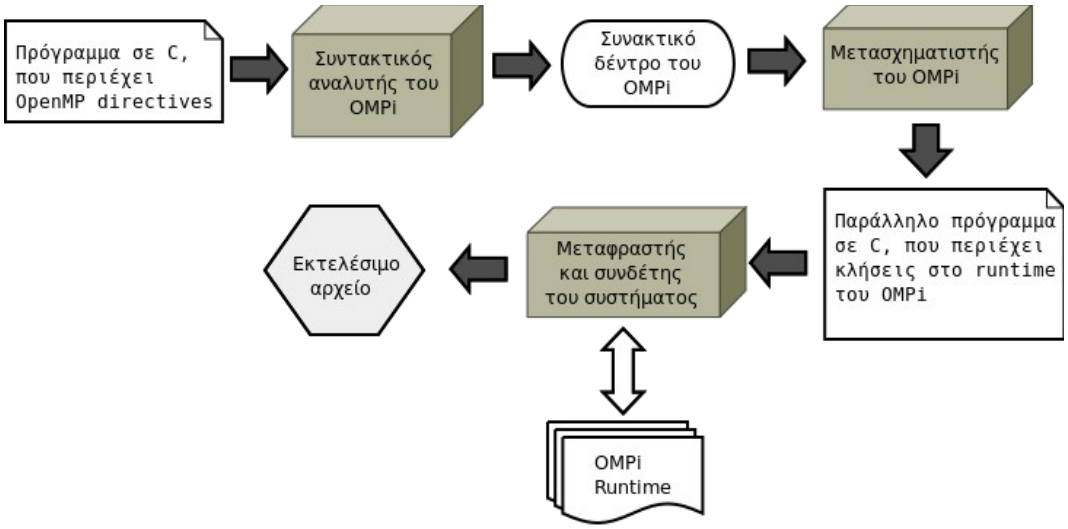
Ο OMPi κατασκευάζει αόριστες εκτελεστικές οντότητες (execution entities), την υλοποίηση των οποίων αναλαμβάνει το runtime (π.χ. νήματα POSIX επιπέδου πυρήνα), επιλέγοντας από τις διαθέσιμες βιβλιοθήκες που παρέχει ο OMPi.

3.2 Ροή μετάφρασης

Η ροή που ακολουθεί ο OMPi [3] για την εξαγωγή του παραλληλοποιημένου αρχείου και φείνεται στο σχήμα 3.1, συνοψίζεται σε 2 στάδια:

1. Το πρόγραμμα C που περιέχει τα OpenMP directives περνάει από τον συντακτικό αναλυτή (parser) του OMPi. Κατά την διαδικασία αυτή σχηματίζεται το συντακτικό δέντρο, του οποίου κάποιοι κόμβοι αποτελούνται από τα directives που ο parser συνάντησε.
2. Οι κόμβοι αυτοί, περνάνε από τις διαδικασίες μετασχηματισμού του OMPi, έτσι ώστε να αντιστοιχιστούν σε ισοδύναμο κώδικα C, ο οποίος περιέχει και κλήσεις στο runtime του.

Ο μεταφραστής γλώσσας C του συστήματος που φιλοξενεί τον OMPi, μεταφράζει το παραγόμενο αρχείο, συνδέοντας τις κλήσεις αυτές με τις βιβλιοθήκες runtime του. Ο OMPi παρέχει περισσότερες της μίας τέτοιες βιβλιοθήκες, οι οποίες καθορίζουν και την υλοποίηση των εκτελεστικών οντοτήτων (π.χ. Pthreads, Psthreads, διεργασίες). Τελικώς, παράγεται το παραλληλοποιημένο εκτελέσιμο πρόγραμμα.



Σχήμα 3.1*: Η ροή μετάφρασης του OMPi*

3.2.1 Παραγωγή συντακτικού δέντρου

Το συντακτικό δέντρο, αποτελείται από κόμβους 7 ειδών:

- Εκφράσεις (expressions): Μπορεί να περιγράφουν κάποιο αναγνωριστικό (identifier), αλφαριθμητικό, σταθερά, κλήση σε συνάρτηση ή κάποιον τελεστή. Ανάλογα με το είδος της έκφρασης, αποθηκεύονται στον κόμβο τα απαραίτητα χαρακτηριστικά του.
- Τμήμα προσδιορισμού μιας δήλωσης (declaration specifier): Οι κόμβοι αυτού του είδους περιέχουν το κομμάτι που προσδιορίζει για τί είδους δήλωση πρόκειται. (π.χ. unisgned int)
- Κυρίως τμήμα μιας δήλωσης: Πρόκειται για το αναγνωριστικό του υπό δήλωση δεδομένου (identifier), το μέγεθός του, αν πρόκειται για πίνακα, την πληροφορία για το αν πρόκειται για δείκτη, καθώς και ενδεχόμενη αρχικοποίηση.
- Προτάσεις (statements): Μπορεί να περιγράφουν επαναληπτικούς βρόγχους, άλματα συνθήκης, δηλώσεις συναρτήσεων ή μεταβλητών (οι οποίοι περιέχουν δείκτες στους δύο κόμβους που αναφέρθηκαν παραπάνω) ή προτάσεις που ανήκουν σε εντολές OpenMP.
- OpenMP directives: Περιέχουν τον τύπο του directive, καθώς και μια λίστα (ενδεχομένως να είναι κενή), με τις παραμέτρους που το συνοδεύουν.

- OpenMP παραμέτρους (clauses): Περιλαμβάνουν τον τύπο της παραμέτρου, το επόμενο στοιχείο στη λίστα του directive στο οποίο ανήκουν, καθώς και μια (ενδεχομένως κενή) λίστα μεταβλητών, τις οποίες η παράμετρος κατηγοριοποιεί.
- Μια OpenMP δομή, η οποία περιγράφεται από δείκτη στον κόμβο με το αντίστοιχο directive της, το σώμα της, ότι περικλείεται δηλαδή μεταξύ των αγκυλών, καθώς και δείκτη στην πρόταση OpenMP στην οποία ανήκει.

Η αρχική δομή δεδομένων του δέντρου, απαιτεί απλά καθοδικούς δείκτες από τη ρίζα προς τα φύλλα. Πάραυτα, κατά τον μετασχηματισμό, ενδέχεται να υπάρξει ανάγκη για ανοδικούς δείκτες. Έτσι οι δείκτες αυτοί αρχικοποιούνται σε NULL κατά το αρχικό συντακτικό δέντρο.

3.2.2 Πίνακας συμβόλων

Ο πίνακας συμβόλων του OMPi, διέπεται από την λογική της αποφυγής χειρισμού αλφαριθμητικών (strings), οι οποίοι κοστίζουν. Για τον λόγο αυτό, τα αλφαριθμητικά των υπό αποθήκευση δεδομένων στον πίνακα, μετατρέπονται σε *σύμβολα:* ένα σύμβολο δεν είναι πάρα ένας δείκτης στο αρχικό αλφαριθμητικό. Έτσι, δεδομένου ενός αλφαριθμητικού X, η συνάρτηση *Sybmol(*X*)*, χρησιμοποιώντας την συνάρτηση κατακερματισμού του πίνακα, επιστρέφει την θέση στον πίνακα όπου αυτό βρίσκεται. Εάν η αναζήτηση αποτύχει, δημιουργείται νέα καταχώρηση στον πίνακα, και αυτή επιστρέφεται.

Έτσι, εφ' όσον ο πίνακας αποτελείται από σύμβολα, το να αναζητήσουμε κάποιο απ' αυτά, συνεπάγεται ότι θα χρειαστεί να αναζητήσουμε το σύμβολο στον πίνακα: αρκεί να κατακερματίσουμε το σύμβολο, δηλαδή έναν δείκτη. Έτσι αποφεύγουμε δαπανηρούς κατακερματισμούς μεγάλων σε μέγεθος αλφαριθμητικών.

Οι βασικές πληροφορίες που κρατάει ο πίνακας για κάθε εγγραφή του είναι οι εξής:

- Το είδος του περιεχομένου:
  - Μεταβλητές
  - Τύποι ορισμένοι από τον χρήστη
  - Δομές ομαδοποίησης (struct ή union)
  - Enumerated τύποι
  - Ετικέτες (labels)
  - Συναρτήσεις

- Το επίπεδο εύρους (scoping level) του εκάστοτε συμβόλου. Στην ουσία πρόκειται για το επίπεδο φωλιάσματος στο οποίο ανήκει.
- Την πληροφορία αν πρόκειται για πίνακα ή threadprivate δεδομένα
- Τρεις δείκτες οι οποίοι κρατούν πληροφορίες για την δήλωση της μεταβλητής:
  - ***spec*:** Ο προσδιορισμός της δήλωσης, όπως αναφέρθηκε παραπάνω.
  - ***decl:*** Οι κυρίως πληροφορίες της δήλωσης.
  - ***idecl***: Πληροφορίες για πιθανή αρχικοποίηση της μεταβλητής. Αν αυτή δεν έχει αρχικοποιηθεί, ο δείκτης αυτός είναι NULL.

Οι θέσεις του πίνακα αποτελούνται από buckets τα οποία είναι της μορφής LIFO, έτσι ώστε στην περίπτωση που αναζητηθεί κάποιο σύμβολο που υπάρχει περισσότερες από μια φορές, η εγγραφή που θα επιστραφεί, να είναι αυτή που εμφανίστηκε τελευταία: αυτή με το μεγαλύτερο scope είναι και η υπερισχύουσα.

Όταν ένα scope κλείνει, ενδεχόμενες N γραμμικές αναζητήσεις για την αφαίρεση των N μεταβλητών που ανήκουν στο scope αυτό είναι ακριβές ($O(N^2)$), για τον λόγο αυτό τα στοιχεία του εκάστοτε scope παραμένουν συνδεδεμένα μεταξύ τους με λίστα, έτσι ώστε η διαγραφή τους να γίνεται ταχύτερα (O(N)).

Ο πίνακας συμβόλων είναι ο μόνος τρόπος για να συνδεθεί μια μεταβλητή, που εμφανίζεται σε κάποια έκφραση, με την δήλωσή της: το συντακτικό δέντρο δεν κρατάει αυτήν την πληροφορία. Ο πίνακας μεταβάλλεται δυναμικά κατά το parsing, επομένως μετά το πέρας αυτού, ο πίνακας είναι άδειος.

3.2.3 Μετασχηματισμός κόμβων με OpenMP directives

Όπως αναφέρθηκε παραπάνω, μετά το πέρας του parsing το συντακτικό δέντρο είναι άδειο. Έτσι, πριν τον μετασχηματισμό τον κόμβων που περιέχουν εντολές OpenMP, το δέντρο αυτό είναι απαραίτητο να επανασχηματιστεί.

Προτού γίνει η κλήση προς την κεντρική συνάρτηση μετασχηματισμού (*ast_xform)*, εισέρχονται στο δέντρο οι δηλώσεις μερικών απαραίτητων runtime συναρτήσεων, καθώς ο παραγόμενος κώδικας βασίζεται σ' αυτές.

Αφού ο υπό επεξεργασία κόμβος μετασχηματιστεί, έχει παραχθεί ένα υπόδεντρο, ισοδύναμο με το υπόδεντρο το οποίο έχει ρίζα τον συγκεκριμένο κόμβο. Το παραγόμενο αυτό υπόδεντρο, περιέχει κλήσεις σε σύνθετες διαδικασίες του runtime, δεν περιέχει όμως πλέον εντολές OpenMP. Έτσι, το παλιό υπόδεντρο αντικαθίσταται.

Μία από τις διαδικασίες βελτιστοποίησης του παραλληλοποιημένου προγράμματος που γίνεται από τον OMPi, είναι ο έλεγχος για περιττεύοντα barriers. Πολλές από τις εντολές του OpenMP υπονοούν barrier στο τέλος τους, έτσι στην περίπτωση εμφωλευμένων directives, η περίπτωση να υπάρχουν περισσότερα του ενός barrier, από τα οποία απαραίτητο είναι μόνο το ένα, είναι αρκετά πιθανό να εμφανιστεί. Ο OMPi μέσω της συνάρτησης *xform_implicit_barrier_is_needed* αποκλείει αυτό το ενδεχόμενο.

Κατά τον μετασχηματισμό, η κατηγοριοποίηση των μεταβλητών που αναφέραμε στην ενότητα 2.4 (shared, private κ.τ.λ.), ενδέχεται να προκαλέσει μεταβολές στο εύρος ορατότητας των μεταβλητών. Έτσι, μπορεί να χρειαστεί να δημιουργηθεί ένα καινούριο scope και οι μεταβλητές αυτές να επαναδηλωθούν καταλλήλως.

Μία από τις πολυπλοκότητες που δύνανται να προκύψουν, είναι κατά την αντιμετώπιση firstprivate μεταβλητών. Αν αυτές είναι απλές, τότε αρκεί στο νέο αρχείο να συμπεριληφθεί η δήλωσή τους. Αν όμως, πρόκειται για πίνακα, κάτι τέτοιο καθίσταται αδύνατο. Έτσι, στην αρχή του κυρίους μέρους του προγράμματος, προστίθενται κατάλληλες κλήσεις της συνάρτησης αντιγραφής μνήμης *memcpy*, για να επιτευχθεί η κατάλληλη αρχικοποίηση των firstprivate πινάκων.

Ας αναλύσουμε ένα τέτοιο παράδειγμα μετασχηματισμού. Έστω το εξής σειριακό πρόγραμμα:

```
void main( ){
	int x,y,z = 0;
	#pragma omp parallel shared(x) private(y) firstprivate(z)
	{
		x = y + z;
	}
return;
}
```

Μεταφράζοντάς το με τον OMPi, ενεργοποιώντας το flag *-k*, έτσι ώστε να μην σβηστεί το ενδιάμεσο αρχείο που αυτός παράγει, έχουμε:

```
int main(int argc, char **argv)
{
  ort_initialize(&argc, &argv);
  __original_main(argc, argv);
  ort_finalize(0);
  return (0);
```

```
}
```

Η κύρια συνάρτηση του ενδιάμεσου αυτού προγράμματος, περιέχει τις κλήσεις του συστήματος χρόνου εκτέλεσης (runtime), `ort_initialize` και `ort_finalize`, καθώς και την παραγόμενη συνάρτηση `__original_main` . Οι κλήσεις στο runtime δεν θα αναλυθούν βαθύτερα στην συγκεκριμένη εργασία. Ας δούμε λοιπόν την παραγόμενη συνάρτηση:

```
void __original_main(int _argc_ignored, char ** _argv_ignored)
{
  int x, y, z = 0;


  {
    /* (l10) #pragma omp parallel shared(x) private(y) firstprivate(z) */
    struct __shvt__ {
        int (* x);
        int (* z);
      } _shvars = {
        &x, &z
      };


    ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
  }
  return;
}
```

Παρατηρούμε 2 πράγματα:

- Την δομή *__shvt__*, η οποία κρατάει δείκτες στις μεταβλητές *x* και *z*, για τους εξής λόγους:
  - Η x είναι shared, άρα τα νήματα την μοιράζονται, συνεπώς αρκεί να κρατήσουμε έναν δείκτη στην αρχική μεταβλητή.
  - Η y είναι private, άρα αρκεί να δημιουργήσουμε ένα αντίγραφό της: δεν χρειαζόμαστε δείκτη στην αρχική μεταβλητή, άρα δεν υπάρχει στο struct.
  - Η z είναι firstprivate, άρα δημιουργούμε ένα αντίγραφό της, το οποίο όμως θα πάρει την

τιμή που έχει η αρχική μεταβλητή, γι' αυτό και χρειαζόμαστε δείκτη στην αρχική.

- Την κλήση runtime `ort_execute_parallel`, με ορίσματα την συνάρτηση `_thrFunc0_`, και το παραπάνω `struct`. Ουσιαστικά, η *thrFunc0* είναι η συνάρτηση που θα εκτελέσουν τα νήματα που θα δημιουργηθούν από τον OMPi.

Ας αναλύσουμε τη συνάρτηση αυτή:

```
static void * _thrFunc0_(void * __me)
{
  struct __shvt__ * _shvars = (struct __shvt__ *) ort_get_shared_vars(__me);
  int (* x) = _shvars->x;
  int z = *(_shvars->z);
  int y;

    (*x) = y + z;

  ort_taskwait(2);
  return ((void *) 0);
}
```

Αρχικά, η κλήση συστήματος `ort_get_shared_vars` , παίρνει την δομή *_shvars* που περάστηκε σ' αυτήν σαν όρισμα παραπάνω, και αρχικοποιεί μια ίδια δομή.

- Η τοπική *x* αρχικοποιείται ως δείκτης στην αρχική μεταβλητή *x*.
- Δημιουργείται αντίγραφο της *z*, το οποίο παίρνει την τρέχουσα τιμή της αρχικής *z*.
- Το αντίγραφο της *y*, δεν αρχικοποιείται, καθώς είναι private.

Έτσι, όταν εκτελεστεί η πράξη `(*x) = y + z` , το αποτέλεσμα εκχωρείται κατευθείαν στην αρχική μεταβλητή x. Ότι αλλαγές γίνουν σ' αυτήν, περνάνε κανονικά στην αρχική, ενώ ότι αλλαγές γίνουν στις y και z, x, χάνονται μετά το πέρας της συνάρτησης. Η κλήση runtime `ort_taskwait` υποδηλώνει τον συγχρονισμό που υπάρχει στο τέλος της παράλληλης περιοχής.

Κεφάλαιο 4

Ανάλυση ροής δεδομένων

4.1 Εισαγωγή στην ανάλυση ροής δεδομένων

Η ανάλυση ροής δεδομένων (data flow analysis), είναι στατική ανάλυση των δεδομένων ενός προγράμματος, η οποία εξετάζει την συμπεριφορά συγκεκριμένων μεταβλητών ή εκφράσεων, και συλλέγει πληροφορίες για τον τρόπο με τον οποίον αυτές χρησιμοποιούνται. Οι μεταφραστές οι οποίοι βασίζονται στην στατική ανάλυση, χρησιμοποιούν την ανάλυση αυτή για την ανάπτυξη βελτιστοποιήσεων.

Ανάλογα με το εκάστοτε προς επίλυση πρόβλημα, διαμορφώνονται κατάλληλες εξισώσεις συνόλων, οι οποίες καλούνται ***εξισώσεις ροής ανάλυσης*** (data flow equations). Το πρόγραμμα χωρίζεται σε **τμήματα** (blocks), η μορφή των οποίων μεταβάλλεται επίσης ανάλογα με τις ανάγκες του προβλήματος. Οι εξισώσεις αυτές, επιλύονται αναδρομικά σε κάθε block, έτσι ώστε όταν η αναδρομική στοίβα αδειάσει, να έχουμε διαμορφωμένα τα σύνολα τα οποία απαντούν στο πρόβλημά μας.

Χαρακτηριστικά προβλήματα που επιλύονται από μεταφραστές μέσω τέτοιου τύπου ανάλυση, είναι:

- ***Διαθέσιμες εκφράσεις:*** Εξετάζεται κατά πόσο η αποτίμηση μιας σύνθετης έκφρασης έχει προϋπολογιστεί, και βρίσκεται ήδη αποθηκευμένη σε κάποιον καταχωρητή.
- ***Ανάλυση ενεργών μεταβλητών (liveness analysis):*** Εξετάζεται κατά πόσο, σε ένα συγκεκριμένο σημείο του προγράμματος, έχει νόημα η αποθήκευση εκχωρημένης τιμής σε μια μεταβλητή (ή αν ακολουθεί εγγραφή πριν την επόμενη ανάγνωσή της).
- ***Χρήση μη αρχικοποιημένων μεταβλητών:*** Εντοπίζονται περιπτώσεις κατά τις οποίες γίνεται ανάγνωση μιας μεταβλητής στην οποία δεν έχει ανατεθεί κάποια τιμή.
- ***Απαλοιφή νεκρού κώδικα:*** Εντοπίζονται σημεία του προγράμματος τα οποία λόγω συνθηκών που αποτυγχάνουν πάντα, μένουν εκτός ροής εκτέλεσης. Ο νεκρός κώδικας απαλείφεται από το εκτελέσιμο αρχείο.

Παρακάτω, θα αναλύσουμε τα 2 πρώτα προβλήματα, παρουσιάζοντας ενδεικτικά παραδείγματα.

4.1.1 Ανάλυση liveness

Όπως αναφέρθηκε παραπάνω, η ανάλυση αυτή εξετάζει το κατά πόσον η τρέχουσα τιμή μιας μεταβλητής, πρόκειται να χρησιμοποιηθεί στο μέλλον, ή η αποθήκευση της τρέχουσας τιμής της είναι περιττή. Η εξισώσεις ροής δεδομένων, περιγράφουν επί της ουσίας την εξής ιδέα: Έστω μία μεταβλητή, της οποίας την τιμή κρατάμε αποθηκευμένη. Αν αυτή η μεταβλητή, πρόκειται να υποστεί γραφή, προτού υποστεί ανάγνωση, τότε κρατάμε χωρίς λόγο αποθηκευμένη την παλιά τιμή της στις ενδιάμεσες εντολές.

Έτσι, ο μεταφραστής μπορεί να αποφασίσει να αποδεσμεύσει καταχωρητές που κρατούν νεκρές μεταβλητές, είτε εξ' αρχής να αποφασίσει την μη αποθήκευσή τους.

Προτού εξετάσουμε ένα χαρακτηριστικό παράδειγμα τέτοιας ανάλυσης, είναι απαραίτητο να ορίσουμε τα σύνολα που θα χρησιμοποιήσουμε στις εξισώσεις ροής δεδομένων [4].

Έστω B ένα block εντολών, όπως ορίστηκε στην προηγούμενη ενότητα. Ορίζουμε τα σύνολα:

- $\boldsymbol{in}[\boldsymbol{B}]$ = { Μεταβλητές οι οποίες είναι ζωντανές (live) εισερχόμενοι στο B }
- $\boldsymbol{out}[\boldsymbol{B}]$ = { Μεταβλητές οι οποίες είναι ζωντανές (live) εξερχόμενοι από το B }
- $\boldsymbol{def}[\boldsymbol{B}]$ = { Μεταβλητές στις οποίες εκχωρείται τιμή μέσα στο B }
- $\boldsymbol{use}[\boldsymbol{B}]$ = { Μεταβλητές οι οποίες υπόκεινται ανάγνωση μέσα στο B }

Οι εξισώσεις ροής ανάλυσης, με βάση τα παραπάνω σύνολα, έχουν την μορφή:

Εξισώσεις 4.1

4.1.1. $\textit{Αρχικοποίηση}: \ in[B]=\varnothing \ , \forall \ block \ B \ του \ προγράμματος.$

4.1.2. $\text{in}[B]=use[B]\cup(out[B]\text{def}[B])$

4.1.3. $out[B]=\bigcup_{[S \ απόγονος \ του \ B]} \text{in}[S]$

Οι τρεις αυτές εξισώσεις, περιγράφουν στην ουσία τις εξής δύο απλές ιδέες:

- Εισερχόμενοι στο block B, ζωντανές μεταβλητές, είναι αυτές οι οποίες διαβάζονται στο B, αλλά και εκείνες οι οποίες είναι ζωντανές κατά την έξοδο από αυτό και δεν υπόκεινται εκχώρηση στο B.
- Οι μεταβλητές οι οποίες είναι ζωντανές κατά την έξοδο από το B, δεν είναι άλλες από εκείνες που είναι ζωντανές στα blocks που ακολουθούν το B.

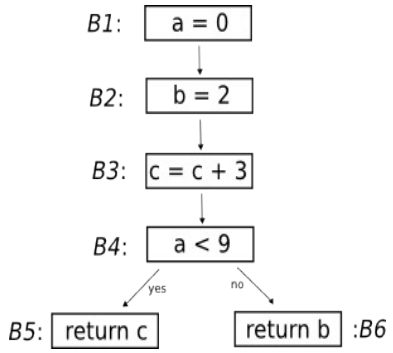
Εδώ, παρατηρούμε και την *λογική της ανάλυσης ροής δεδομένων*: απλές ιδέες, τυποποιούνται, έτσι ώστε να εκμεταλλευτούμε την μαθηματική αυστηρότητα της θεωρίας συνόλων, για να λύσουμε με ακρίβεια, προβλήματα που ενώ διαισθητικά μπορεί να φαίνονται απλά, μπορούν εύκολα να πολυπλοκοποιηθούν σε μεγάλα και σύνθετα προγράμματα.

Η παραπάνω ανάλυση, θα μας απαντήσει, για το κάθε block εντολών ενός προγράμματος, ποιές μεταβλητές είναι ζωντανές και ποιές όχι, κατά την είσοδο και έξοδο από αυτό. Ας εξετάσουμε ένα παράδειγμα, καθώς και την σταδιακή επίλυση των εξισώσεων:

Παράδειγμα 4.1

Έστω το παρακάτω πρόγραμμα, και το γράφημα ροής που του αντιστοιχεί, αν θεωρήσουμε ότι η κάθε μία εντολή αποτελεί και ένα block.

```
B1.    a = 0;
B2.    b = 2;
B3.    c = c + 3;
B4.    if( a < 9 )
B5.           return c;
       else
B6.           return b;
```



Επίλυση Εξισώσεων 4.1 για το Παράδειγμα 4.1:

Από την εξίσωση 4.1.3, συμπεραίνουμε ότι η επίλυση των εξισώσεων θα ξεκινήσει από τα φύλλα του δέντρου γραφήματος ροής: η εξίσωση απαιτεί τα σύνολα in των απογόνων, επομένως ξεκινάμε από τα blocks τα οποία δεν έχουν απογόνους.

1. Έχουμε λοιπόν στα φύλλα:

$$\begin{cases} 4.1.3 \Rightarrow [out[B5] = \varnothing \wedge out[B6] = \varnothing] & \\ [use[B5] = \{ c \} \wedge def[B5] = \varnothing] & (4.1.4) \\ [use[B6] = \{ b \} \wedge def[B6] = \varnothing] & (4.1.5) \\ [4.1.2, 4.1.4] \Rightarrow [in[B5] = \{ c \} \cup (\varnothing - \varnothing) = \{ c \}] & (4.1.6) \\ [4.1.2, 4.1.5] \Rightarrow [in[B6] = \{ b \} \cup (\varnothing - \varnothing) = \{ b \}] & (4.1.7) \end{cases}$$

2. Προχωράμε ανοδικά, και πλέον μπορούμε να επιλύσουμε της εξισώσεις για τον κόμβο B4:

$$\begin{cases} [4.1.3, 4.1.6, 4.1.7] \Rightarrow [out[B4] = in[B5] \cup in[B6] = \{ c \} \cup \{ b \} = \{ c, b \}] & (4.1.8) \\ [use[B4] = \{ a \} \wedge def[B4] = \varnothing] & (4.1.9) \\ [4.1.2, 4.1.8, 4.1.9] \Rightarrow [in[B4] = \{ a \} \cup (\{ b, c \} - \varnothing) = \{ a, b, c \}] & (4.1.10) \end{cases}$$

3. Στο επόμενο βήμα, υπολογίζουμε τα σύνολα για τον κόμβο B3:

$$\begin{cases} [4.1.3, 4.1.10] \Rightarrow [out[B3] = in[B4] = \{ a, b, c \}] & (4.1.11) \\ [use[B3] = \{ c \} \wedge def[B3] = \{ c \}] & (4.1.12) \\ [4.1.2, 4.1.11, 4.1.12] \Rightarrow [in[B3] = \{ c \} \cup (\{ a, b, c \} - \{ c \}) = \{ c \} \cup \{ a, b \} = \{ a, b, c \}] & (4.1.13) \end{cases}$$

4. Για τον B2:

$$\begin{cases} [4.1.3, 4.1.13] \Rightarrow [out[B2] = in[B3] = \{ a, b, c \}] & (4.1.14) \\ [use[B2] = \varnothing \wedge def[B2] = \{ b \}] & (4.1.15) \\ [4.1.2, 4.1.14, 4.1.15] \Rightarrow [in[B2] = \varnothing \cup (\{ a, b, c \} - \{ b \}) = \{ a, c \}] & (4.1.16) \end{cases}$$

5. Και τέλος για τον B1:

$$\begin{cases} [4.1.3, 4.1.16] \Rightarrow [out[B1] = in[B2] = \{ a, c \}] & (4.1.17) \\ [use[B1] = \varnothing \wedge def[B1] = \{ a \}] & (4.1.18) \\ [4.1.2, 4.1.17, 4.1.18] \Rightarrow [in[B1] = \varnothing \cup (\{ a, c \} - \{ a \}) = \{ c \}] & \end{cases}$$

□

Η παραπάνω μέθοδος επίλυσης του προβλήματος ζωντανών μεταβλητών, απέδειξε, με φορμαλιστικό τρόπο, αυτά που σε ένα τόσο απλό πρόγραμμα φαίνονται δια γυμνού οφθαλμού:

- Η πρόταση 4.1.19, μας δείχνει ότι εισερχόμενοι στην αρχή του προγράμματος, η μόνη ζωντανή μεταβλητή είναι η c, καθώς οι a και b υπόκεινται σε εγγραφές στους κόμβους B1 και B2 αντίστοιχα.
- Η 4.1.17, μας δείχνει ότι εξερχόμενοι από τον B1, έχουμε ζωντανή και την a: η εκχωρημένη τιμή της θα ζητηθεί στον κόμβο συνθήκης B4.

4.2.2 Διαθέσιμες εκφράσεις

Η επίλυση των εξισώσεων, σε αυτήν την περίπτωση, έχει ως στόχο την αποφυγή περιττών υπολογισμών σύνθετων εκφράσεων. Τα οφέλη που μπορεί να αποφέρει μια βελτιστοποίηση διαθέσιμων εκφράσεων σε έναν μεταφραστή είναι προφανή: αποφυγή δαπανηρών περιττών υπολογισμών.

Έστω block εντολών B. Θεωρούμε ότι η έκφραση *x+y* έχει ήδη υπολογιστεί, και βρίσκεται αποθηκευμένη σε κάποιον καταχωρητή. Λέμε ότι το block B *σκοτώνει (kills)* την έκφραση, εάν εκχωρεί τιμή στο x ή στο y, και δεν την επαναϋπολογίζει. Λέμε ότι το B *γεννά (generates)* την έκφραση, εάν την αποτιμά, χωρίς να ακολουθεί εκχώρηση στο x ή στο y.

Ορίζουμε τα σύνολα [4]:

- $\boldsymbol{in}[\boldsymbol{B}]$ = { Εκφράσεις οι οποίες είναι διαθέσιμες εισερχόμενοι στο B }
- $\boldsymbol{out}[\boldsymbol{B}]$ = { Εκφράσεις οι οποίες είναι διαθέσιμες εξερχόμενοι από το B }
- $\boldsymbol{e}_{gen}[\boldsymbol{B}]$ = { Εκφράσεις οι οποίες γεννιούνται μέσα στο B }
- $\boldsymbol{e}_{kill}[\boldsymbol{B}]$ = { Εκφράσεις οι οποίες σκοτώνονται μέσα στο B }

Οι εξισώσεις ροής ανάλυσης, με βάση τα παραπάνω σύνολα, έχουν την μορφή:

Εξισώσεις 4.2

4.2.1. $in[B1]=\emptyset$, *όπου B1 η ρίζα του γραφήματος ροής του προγράμματος.*

4.2.2. $out[B]=e_{gen}[B]\cup(in[B]-e_{kill}[B])$

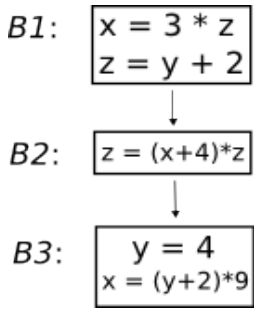
4.2.3. $in[B]=\underset{[S\ \text{πρόγονος του}\ B]}{\cap} out[S]$, *όπου* $B\neq B1$

Η παραπάνω ανάλυση θα μας απαντήσει για το ποιές εκφράσεις είναι διαθέσιμες σε οποιοδήποτε σημείο του προγράμματος. Παρατηρούμε την συμμετρία μεταξύ των εξισώσεων 4.1 και 4.2. Το σύνολο e_{gen} αντιστοιχεί στο σύνολο *use* της προηγούμενης ενότητας, και το e_{kill} στο *def*.

Παράδειγμα 4.2:

Θεωρούμε το παρακάτω κομμάτι κώδικα, και το γράφημα ροής που του αντιστοιχεί :

```
B1.   x = 3 * z;
      z = y + 2;
B2.   x = (x + 4)*z;
B3.   y = 4;
      x = (y + 2)*9;
```



Εδώ, εφόσον απαιτείται η γνώση του προγονικού σύνολου *out*, συμπεραίνουμε ότι η επίλυση των εξισώσεων θα είναι καθοδική ως προς το δέντρο ροής.

1. Ξεκινάμε λοιπόν από τη ρίζα του δέντρου ροής B1. Η μόνη έκφραση που γεννάται είναι η *y+2*, καθώς η έκφραση *3*z* δεν γεννάται, λόγω της ανάθεσης στο z που ακολουθεί:

$$\begin{cases} [4.2.1]\Rightarrow[in[B1]=\emptyset] & (4.2.4)\\ [e_{gen}[B1]=\{\ y+2\ \}\wedge e_{kill}[B1]=\emptyset] & (4.2.5)\\ [4.2.2,\ 4.2.4,\ 4.2.5]\Rightarrow[out[B1]=\{\ y+2\ \}\cup(\emptyset-\emptyset)=\{\ y+2\ \}] & (4.2.6)\end{cases}$$

2. Προχωράμε καθοδικά, εξετάζουμε τον B2, ο οποίος γεννά την έκφραση *x + 4*, αλλά όχι και την υπερέκφρασή της, *(x + 4)*z*, καθώς το αποτέλεσμα εκχωρείται στο z:

$$\begin{cases} [4.2.3,\ 4.2.6] \Rightarrow [in[B2] = out[B1] = \{\ y+2\ \}] & (4.2.7) \\ [e_{gen}[B2] = \{\ x+4\ \} \wedge e_{kill}[B2] = \emptyset] & (4.2.8) \\ [4.2.2,\ 4.2.7,\ 4.2.8] \Rightarrow [out[B2] = \{\ x+4\ \} \cup (\{\ y+2\ \} - \emptyset) = \{\ x+4,\ y+2\ \}] & (4.2.9) \end{cases}$$

3. Στο επόμενο βήμα, υπολογίζουμε τα σύνολα για τον κόμβο B3, ο οποίος δεν σκοτώνει την έκφραση *y + 2*, καθώς παρά την εκχώρηση στο y, ξανά αποτιμά την έκφραση, ως υποέκφραση της *(y + 2) * 9*, την οποία και γεννά. Η έκφραση που σκοτώνεται είναι η *x + 4* μέσω της εκχώρησης στο x:

$$\begin{cases} [4.2.3,\ 4.2.9] \Rightarrow [in[B3] = out[B2] = \{\ x+4,\ y+2\ \}] & (4.2.10) \\ [e_{gen}[B3] = \{\ (y+2)*9\ \} \wedge e_{kill}[B3] = \{\ x+4\ \}] & (4.2.11) \\ [4.2.2,\ 4.2.10,\ 4.2.11] \Rightarrow out[B3] = \{\ (y+2)*9\ \} \cup (\{\ x+4,\ y+2\ \} - \{\ x+4\ \}) = & \\ \qquad = \{\ (y+2)*9\ \} \cup \{\ y+2\ \} = \{\ y+2,\ (y+2)*9\ \} & \end{cases}$$

□

Αξίζει να σημειωθεί ότι οι μέθοδοι ανάλυσης ροής δεδομένων που παρουσιάστηκαν στις ενότητες 4.2.1 και 4.2.2 ανήκουν στις επαναληπτικές μεθόδους τέτοιου τύπου ανάλυσης, καθώς είναι δυνατό να εφαρμοστούν σε γραφήματα ροής με βρόγχους. Στην περίπτωση αυτή, τα παραπάνω βήματα εκτελούνται επαναληπτικά, έως ότου σημειωθούν δύο επαναλήψεις χωρίς μεταβολή συνόλων. Η παρουσίαση ενός τέτοιου παραδείγματος κρίνεται άσκοπη στην παρούσα εργασία, λόγω του επικείμενου μεγάλου όγκου πράξεων συνόλων, αλλά είναι ενδεικτικό της απλοποίησης πολύπλοκων προγραμμάτων, με ορθή χρήση του μαθηματικού φορμαλισμού.

4.3 Το πρόβλημα εγγραφής/ανάγνωσης

Το πρόβλημα συνοψίζεται ως εξής: έστω block εντολών B. Η ανάλυση καλείται να αποφανθεί για το ποιές μεταβλητές υπόκεινται εγγραφή, ποιές ανάγνωση, ποιές και τα δύο, και με ποιά σειρά, στο B.

Ορίζουμε τα εξής σύνολα:

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

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

***RW*(*B*)** = { Μεταβλητές που υφίστανται και τα δύο στο B, αλλά πρώτα ανάγνωση}

***WR*(*B*)** = { Μεταβλητές που υφίστανται και τα δύο στο B, αλλά πρώτα εγγραφή}

***unk*(*B*)** = { Μεταβλητές για των οποίων την χρήση στο B δεν είναι δυνατόν να αποφανθούμε}

Το σύνολο *unk*, υπάρχει διότι υπάρχουν περιπτώσεις στις οποίες η ανάλυση αδυνατεί να αποφανθεί για τη χρήση που υφίσταται κάποια μεταβλητή. Τέτοιες περιπτώσεις είναι η σύνθετη χρήση δεικτών, κατά την οποία ενδέχεται να προκύψουν εγγραφές και αναγνώσεις αδύνατο να εντοπιστούν στατικά . Επίσης αν η μεταβλητή αυτή περνάει ως όρισμα σε συνάρτηση, στης οποίας τον κώδικα δεν έχουμε πρόσβαση, τότε επίσης δεν μπορούμε να αποφανθούμε.

Η λύση, εκφρασμένη μέσω της θεωρίας συνόλων, είναι η εξής:

Έστω μεταβλητή x η οποία εμφανίζεται στο block B:

Εξετάζουμε σε τί είδους εντολή εμφανίζεται η x:

- *[εντολή ανάγνωσης]* ⇒
    - *αν x ∈ WR(B) ή x ∈ RW(B): τότε αγνόησε την εμφάνιση*
    - *αν x ∈ W(B): τότε x ∈ WR(B)*
    - *αλλιώς: x ∈ R(B)*

- *[εντολή εγγραφής]* ⇒
    - *αν x ∈ WR(B) ή x ∈ RW(B): τότε αγνόησε την εμφάνιση*
    - *αν x ∈ R(B): τότε x ∈ RW(B)*
    - *αλλιώς: x ∈ W(B)*

- *[δείκτης στην διεύθυνση του x]* ⇒ *x ∈ unk(B)*

Με την επίλυση του προβλήματος αυτού, έχουμε στην ουσία μια εικόνα για την ροή των δεδομένων στο πρόγραμμά μας. Στην παράλληλη επεξεργασία, όπου παράλληλες εκτελεστικές οντότητες ενδέχεται να προσπελάσουν συγχρόνως τα ίδια δεδομένα, η παραπάνω ανάλυση μπορεί να μας βοηθήσει να εντοπίσουμε και να αποτρέψουμε ανεπιθύμητες τέτοιες προσπελάσεις. Επίσης, μπορεί να μας βοηθήσει να αποτρέψουμε περιττές μεταφορές δεδομένων (π.χ. αν η ανάλυσή μας αποφασίσει ότι τα δεδομένα που μεταφέρουμε, δεν πρόκειται να προσπελαστούν), οι οποίες σε παράλληλα συστήματα, ενδέχεται να είναι πολύ δαπανηρές.

4.3.1 Διασυναρτησιακή ανάλυση

Η διασυναρτησιακή ανάλυση, μας επιτρέπει να ακολουθήσουμε την ροή του προγράμματος, στην περίπτωση που υπάρχει κλήση κάποιας συνάρτησης, στης οποίας τον κώδικα έχουμε πρόσβαση, βρίσκεται δηλαδή, σε κάποιο από τα υπό μετάφραση αρχεία. Έστω *B* block εντολών, και έστω ότι στο B καλείται η προσβάσιμη συνάρτηση *f*. Τότε, εφαρμόζουμε αρχικά την ανάλυση μας στην *f*, και εκτελούμε τις εξής πράξεις συνόλων:

- Αρχικά ενώνονται τα σύνολα εγγραφής και ανάγνωσης
  - $R(B) = R(B) \cup R(f)$
  - $W(B) = W(B) \cup W(f)$

- Μεταβλητές που γράφηκαν στην f, και είχαν ήδη διαβαστεί στο B ανήκουν στο RW, και αντίστροφα, στο WR.
  - $RW(B) = RW(f) \cup (R(B) \cap W(f))$
  - $WR(B) = WR(f) \cup (W(B) \cap R(f))$

- Τέλος, αφαιρούμε από τα σύνολα εγγραφής και ανάγνωσης, τα στοιχεία της προαναφερθείσας τομής
  - $W(B) = W(B) – (RW(B) \cup WR(B))$
  - $R(B) = R(B) – (RW(B) \cup WR(B))$

Έτσι, μετά το πέρας της ανάλυσης, θα έχουμε ακολουθήσει όλες τις πιθανές κλήσεις που συναντήσαμε, με αποτέλεσμα να έχουμε εικόνα για την ροή των δεδομένων σε όλη την ακολουθία κλήσεων, καθώς και μια εικόνα για τις εξαρτήσεις που τις διέπουν.

4.4 Υλοποίηση στον OMPi

Οι βελτιστοποιήσεις που προκύπτουν από το πρόβλημα liveness, καθώς και αυτό των διαθέσιμων εκφράσεων, πέρα από το ενδιαφέρον που παρουσιάζει η θεωρητική τους προσέγγιση, παρέχονται από την πλειοψηφία των μεταφραστών. Ο OMPi, πάνω στον οποίον δομήθηκε αυτή η εργασία, είναι ένας μεταφραστής source-to-source. Συνεπώς το εξαγόμενο από αυτόν πρόγραμμα, θα μεταφραστεί πιθανότατα από κάποιον μεταφραστή ο οποίος παρέχει τις προαναφερθείσες βελτιστοποιήσεις.

Ο OMPi όμως, είναι ένας παραλληλοποιητικός μεταφραστής, το οποίο συνεπάγεται ότι σημαντικές για την απόδοση των παραλληλοποιημένων προγραμμάτων του, είναι η μεταφορά, αποθήκευση και γενικότερα η διαχείριση δεδομένων. Έτσι, το πρόβλημα του οποίου τη λύση επιλέξαμε να υλοποιήσουμε στον OMPi, είναι αυτό της ***εγγραφής/ανάγνωσης***. Η επίλυση, ή ακόμα και η διατύπωση του προβλήματος αυτού δεν είναι συνήθης, καθώς σε σειριακούς μεταφραστές, δεν παρουσιάζονται προβλήματα που να επιλύονται μέσω αυτού. Η λύση του, αν και από θεωρητικής άποψης δεν είναι ιδιαίτερα πολύπλοκη, πρακτικά μας δίνει την δυνατότητα να εφαρμόσουμε την τεχνική του autoscoping (Κεφάλαιο 5), καθώς και άλλες βελτιστοποιήσεις.

Η λύση που παρουσιάστηκε στην προηγούμενη ενότητα, υλοποιήθηκε στον OMPi, μέσω της διαδικασίας κατασκευής του συντακτικού δέντρου, την οποία είδαμε στην υποενότητα 3.2.1. Το δέντρο προσπελάζεται, είτε πριν, είτε μετά τον μετασχηματισμό του, και κατά την προσπέλασή του αυτή, κρατούνται οι απαραίτητες πληροφορίες, έτσι όταν εντοπιστεί η εμφάνιση μιας μεταβλητής, να μπορούμε να εφαρμόσουμε τον αλγόριθμο της ενότητας 4.3.

4.4.1 Υλοποίηση συνόλων

Για την παραπάνω υλοποίηση, προστέθηκε στον OMPi μια δομή δεδομένων που υλοποιεί τα σύνολα ως μαθηματικές οντότητες, με τους κανόνες που τα διέπουν. Συνοδεύεται από τις βασικότερες συναρτήσεις χειρισμού συνόλων. Για την υλοποίηση αυτή χρησιμοποιήθηκε ως δομή ο πίνακας κατακερματισμού, λόγω της εύρεσης και προσπέλασης στοιχείου σε χρόνο O(1 + c), όπου *c ο αριθμός των συγκρούσεων στην συγκεκριμένη θέση του πίνακα*. Χρησιμοποιήθηκε για τον λόγο αυτόν, η δομή του πίνακα συμβόλων του OMPi. Επομένως το σύνολο, ως τύπος δεδομένων, δεν είναι παρά ένα περιτύλιγμα μετονομασίας της δομής *symtab*:

```
typedef symtab set;
```

Έτσι οι διαδικασίες που υλοποιήθηκαν, είναι οι εξής:

- ***set createSet( ):*** Η δημιουργία ενός συνόλου δεν είναι παρά η αρχικοποίηση ενός πίνακα συμβόλων:

```
set createSet(){
	set a = (set)Symtab();
	return a;
}
```

- ***addToSet(set a, symbol s)***: Η προσθήκη στοιχείου σε σύνολο, στην περίπτωση που δεν έχουμε μεγάλο αριθμό συγκρούσεων στον πίνακα συμβόλων εκτελείται σε χρόνο O(1 + c). Πρόκειται απλά για κλήσεις των ήδη υλοποιημένων διαδικασιών προσπέλασης και εισαγωγής, με έλεγχο για την ήδη ύπαρξη του στοιχείου στο σύνολο, καθώς ενώ ο πίνακας συμβόλων επιτρέπει ως δομή την ύπαρξη μεταβλητών με το ίδιο όνομα, το σύνολο όχι.

```
int addToSet(set a, symbol s){
	/* A set, by definition, can't contain a duplicate element */
	if( symtab_get( (symtab)a, s, IDNAME) == NULL)
		symtab_put( a, s, IDNAME);
}
```

- ***removeFromSet(set a, symbol s):*** Όμοια με την εισαγωγή, έχουμε O(1 + c), και πρόκειται απλά για ένα περιτύλιγμα της υλοποιημένης συνάρτησης αφαίρεσης από τον πίνακα συμβόλων:

```
int removeFromSet(set a, symbol s){
	symtab_remove( a, s, IDNAME);
}
```

- ***int isInSet(set a, symbol s):*** Η αναζήτηση είναι επίσης ένα περιτύλιγμα χρόνου O(1 + c).

```
int isInSet(set a, symbol s){
  return( symtab_get(a, s, IDNAME) != NULL);
}
```

- ***setEmpty(set a):*** Η εκκένωση του συνόλου είναι ένα περιτύλιγμα της συνάρτησης αδειάσματος του πίνακα συμβόλων και εκτελείται σε O(N).

```
void setEmpty( set a){
```

```
    symtab_drain( (symtab)a );
}
```

Οι πράξεις συνόλων που υλοποιήθηκαν είναι οι εξής:

- ***set setIntersection(set a, set b):*** Η τομή δύο συνόλων γίνεται σε χρόνο O(N+N*C), καθώς εκτελούνται N αναζητήσεις (όπου N η πληθικότητα του ενός συνόλου), σε χρόνο O(1+C) η κάθε μία.
- ***set setUnion(set a, set b):*** Η ένωση συνόλων εκτελείται σε χρόνο O((2N + M) * (1+C)), όπου M η πληθικότητα του έτερου συνόλου. Πρόκειται για M εισαγωγές σε χρόνο O(1 + C) για το ένα σύνολο, N για το άλλο, καθώς και N αναζητήσεις για να αποφύγουμε διπλότυπα στοιχεία στο σύνολο.
- ***set setSubtraction(set a, set b)***: Η αφαίρεση συνόλων περιλαμβάνει N αναζητήσεις και N εισαγωγές, συνεπώς εκτελείται σε χρόνο O(2N + 2N*C).

Αξίζει να σημειωθεί ότι με ποιοτική συνάρτηση κατακερματισμού, αλλά και πίνακα μεταβαλλόμενου μεγέθους, ο αριθμός συγκρούσεων c, σε πρακτικές εφαρμογές είναι μικρός.

4.4.2 Γράφημα κληθέντων συναρτήσεων

Στα πλαίσια του ευρωπαϊκού προγράμματος ***SMECY*** (Smart Multicore Embeded Systems) [5], ο OMPi υποστήριξε την δυνατότητα χρήσης του, στον ενσωματωμένο επιταχυντή επίδοσης (embeded performance accelerator) ***STHORM*** [6], ο οποίος αναπτύχθηκε από την εταιρεία STMicroelectronics.

Στόχος του προγράμματος SMECY, είναι η ανάπτυξη νέων προγραμματιστικών τεχνικών, με σκοπό την ανάπτυξη και την εκμετάλλευση αρχιτεκτονικών με πολλούς πυρήνες (many core architectures).

Η συγκεκριμένη πλακέτα , αποτελείται από πολλαπλές συστάδες (clusters) , η κάθε μία εκ των οποίων περιέχει 16 μικροεπεξεργαστές, οι οποίοι μοιράζονται μνήμες πρώτου επιπέδου. Στα πλεονεκτήματά της, εντάσσονται η κλιμακωσιμότητα και η ενεργειακή αποδοτικότητα που παρέχει.

Η ενσωμάτωση του OMPi στον STHORM, είχε ως αποτέλεσμα την δυνατότητα υποστήριξης του μοντέλου OpenMP από τον επιταχυντή. Υλοποιήθηκε ένα σύστημα χρόνου εκτέλεσης (runtime), προσαρμοσμένο στην πλακέτα: αξιοποιήθηκε αποδοτικά το υλικό (hardware) της, που παρέχει δυνατότητες όπως η αύξηση και μείωση ατομικών μετρητών (atomic counters) [6].

Κατά τον μετασχηματισμό του αρχικού προγράμματος, παράγονται δύο αρχεία: το ένα πρόκειται να εκτελεστεί από το φιλοξενόν σύστημα, και το άλλο από τον επιταχυντή. Υπήρξε η ανάγκη οι

συναρτήσεις οι οποίες καλούνται στο κομμάτι του επιταχυντή, να ενσωματωθούν στο πρόγραμμα που αυτός πρόκειται να εκτελέσει, έτσι ώστε να αποφύγουμε περιττή επικοινωνία του με το φιλοξενόν σύστημα. Για την ενσωμάτωση των συναρτήσεων, χρησιμοποιήθηκε ένα ***γράφημα κληθέντων συναρτήσεων*** (call graph), το οποίο κατασκευάστηκε με χρήση της διασυναρτησιακής ανάλυσης ροής δεδομένων. Έτσι, αντί ο χρήστης να επιλέγει τις συναρτήσεις οι οποίες θα ενσωματοθούν, ο μεταφραστής κάνει αυτή την επιλογή μέσω του γραφήματος.

Ορισμός [7]: Το ***γράφημα κληθέντων συναρτήσεων*** ενός προγράμματος, είναι ένα γράφημα $G=(V,E)$, όπου V το σύνολο κορυφών του, οι οποίες αναπαριστούν διαδικασίες του προγράμματος, και E το σύνολο ακμών του, οι οποίες αναπαριστούν πιθανές κλήσεις μεταξύ τους.

Για τον σχηματισμό του *G*, το αρχικό πρόγραμμα υπέστη ανάλυση ροής δεδομένων αξιοποιώντας τις πληροφορίες που παρέχει η διασυναρτησιακή ανάλυση, εφαρμόζοντας την εξής διαδικασία:

- Ορίζεται ως ρίζα του *G* η κεντρική συνάρτηση που εκτελείται από την πλακέτα.
- Έστω κόμβος $v \in G$: Για κάθε συνάρτηση f με πιθανή κλήση στον v, ορίζουμε τον κόμβο u που αντιστοιχεί στην f , φέρουμε την ακμή $E(v,u)$ και εκτελούμε διασυναρτησιακή ανάλυση στον *u*.

Η ανάλυση εκτελείται αναδρομικά σε όλες τις κληθείσες, μέσω του προγράμματος του επιταχυντή, συναρτήσεις, και έτσι μας δίνει μια πλήρη εικόνα για τις εξαρτήσεις μεταξύ αυτών. Το αποτέλεσμα είναι, να ενσωματώνονται μόνο οι απαραίτητες συναρτήσεις. Έτσι, αποφεύγουμε την περιττή μεταφορά του κώδικα συναρτήσεων, αλλά και την ενδεχόμενη επικοινωνία πλακέτας-συστήματος σε περίπτωση απώλειας του κώδικα.

Η παραπάνω χρήση της ανάλυσης εγγραφής/ανάγνωσης, είναι ενδεικτική όσον αφορά την εφαρμογή της σε προβλήματα που αφορούν παράλληλα συστήματα. Εν προκειμένου, όταν μιλάμε για ενσωματωμένα (embeded) συστήματα, η επικοινωνία τους με το σύστημα το οποίο τα φιλοξενεί είναι εξ' ορισμού δαπανηρή και ζητάται να ελαχιστοποιηθεί. Προκύπτουν επομένως προβλήματα βελτιστοποίησης επικοινωνιών, προβλήματα στα οποία η ανάλυση εγγραφής/ανάγνωσης ενδέχεται να μπορεί να απαντήσει.

Κεφάλαιο 5

Autoscoping

5.1 Η έννοια του autoscoping

Το autoscoping στο OpenMP, παρουσιάστηκε για πρώτη φορά στο συνέδριο WOMPAT (*Workshop on Open MP Applications and Tools)* το 2004, από τους Yuan Lin, Christian Terboven, Dieter an Mey, Nawal Copty [8] και ενσωματώθηκε αργότερα στον μεταφραστή της εταιρείας Sun.

Ορισμός: Η δυνατότητα του μεταφραστή να αποφασίσει μόνος του, για την κατάλληλη κατηγοριοποίηση μιας μεταβλητής (scoping), η οποία συναντάται μέσα σε μια παράλληλη περιοχή, καλείται ***autoscoping.***

Όπως είδαμε και στην ενότητα 2.4, η κατηγοριοποίηση των μεταβλητών είναι ένα πολύ σύνθετο ζήτημα, και μπορεί να καθορίσει την ποιότητα παραλληλοποίησης που πετυχαίνουμε. Στόχος του autoscoping δεν είναι να πάρει τις κομβικές για τον προγραμματιστή αποφάσεις: είναι να του παρέχει ένα εργαλείο ανάλυσης και ελέγχου του συγκεκριμένου ζητήματος και να βοηθήσει στην ήδη φιλική προς τον χρήστη δυνατότητα παραλληλοποίησης προγραμμάτων που παρέχει το OpenMP.

Εργαλεία όπως το autoscoping, τα οποία εντάσσονται στον τομέα της αυτόματης παραλληλοποίησης, έχουν ως στόχο να πετύχουν βέλτιστη απόδοση [11] και γι' αυτό, δεν θα πρέπει να αντιμετωπίζονται ως ανεξάρτητες οντότητες, αλλά ως βοηθήματα στα χέρια του προγραμματιστή.

Ορισμός: Μία μεταβλητή σε παράλληλη περιοχή, λέμε ότι υπόκειται σε ***ανταγωνισμό δεδομένων (data race)***, αν είναι δυνατή η προσπέλασή της από περισσότερα του ενός νήματα, εκ των οποίων τουλάχιστον ένα εκτελεί εγγραφή σε αυτήν.

Στα πλαίσια της κατηγοριοποίησης μεταβλητών στο OpenMP, ένα δεδομένο για το οποίο υπάρχει ανταγωνισμός, δεν μπορεί να οριστεί ως shared: το πρόγραμμά μας θα έχει απρόβλεπτα αποτελέσματα. Ας εξετάσουμε ένα παράδειγμα τέτοιου ενδεχομένου:

```
#pragma omp parallel shared(x)
{
	x = omp_get_thread_num( );
	print( x );
}
```

Προφανώς, στην παραπάνω παράλληλη περιοχή δημιουργείται ανταγωνισμός για το x: τα νήματα της παράλληλης ομάδας που δημιουργούνται, εκχωρούν τιμή σ' αυτήν ασυγχρόνιστα. Για παράδειγμα, το νήμα με αναγνωριστικό 1, εκχωρεί το αναγνωριστικό του στην x, αλλά πριν προλάβει να το τυπώσει, το νήμα με αναγνωριστικό 2 εκτελεί την δικιά του εκχώρηση. Έτσι θα τυπωθεί δύο φορές το αναγνωριστικό του δεύτερου νήματος. Ομοίως, είναι πιθανό να τυπωθεί οποιοσδήποτε συνδυασμός αριθμών, κάτι που καθιστά το πρόγραμμά μας ασταθές.

Άλλο ενδεχόμενο:

```
#pragma omp parallel firstprivate(x)
{
	x = omp_get_thread_num( );
	print( x );
}
```

Εδώ, με την κατηγοριοποίηση ως *firstprivate*, εξασφαλίζουμε την συνέπεια των δεδομένων μας, αλλά κάνουμε σπατάλη μεταφοράς δεδομένων: η τρέχουσα τιμή του x θα εκχωρηθεί στο αντίγραφο του κάθε νήματος, όμως κανένα δεν θα την χρησιμοποιήσει.

Έτσι, καταλήγουμε ότι η σωστή κατηγοριοποίηση είναι ως *private*. Με τη λογική αυτή, ο μεταφραστής χρησιμοποιεί την ανάλυση ροής για να αποφασίσει το σωστό scoping της μεταβλητής. Η απόφαση αυτή δεν είναι σωστή ή λάθος: δεν αποκλείεται π.χ. για πειραματικούς λόγους να θέλουμε το scoping του πρώτου παραδείγματος. Η απόφαση, είναι με βάση την λογική χρήση των δεδομένων μας.

5.2 Υλοποίηση της Sun

Η Sun [8], θέσπισε 3 απλούς κανόνες για την λήψη των αποφάσεων κατηγοριοποίησης μεταβλητών που συναντώνται σε παράλληλη περιοχή, και των οποίων το scope δεν έχει καθοριστεί. Οι κανόνες ελέγχονται με τη σειρά, και αν η μεταβλητή υπάγεται σε κάποιον από αυτούς, λαμβάνει και την κατάλληλη κατηγοριοποίηση (οι εναπομείναντες κανόνες δεν ελέγχονται).

1. Αν η μεταβλητή αυτή δεν υπόκειται σε ανταγωνισμό δεδομένων στην παράλληλη περιοχή, από τα νήματα που συμμετέχουν σε αυτήν, τότε ορίζεται ως ***shared***.
2. Αν όλα τα νήματα που συμμετέχον στην περιοχή, της εκχωρούν τιμή σε αυτήν, προτού την διαβάσουν, η μεταβλητή ορίζεται ως ***private***.
3. Αν η μεταβλητή, χρησιμοποιείται σε μία πράξη τύπου reduction, όπως αυτή περιγράφεται στην ενότητα 2.4. , τότε ορίζεται ως ***reduction***.

Εξαιρέσεις: Ο μετρητής ενός παραλληλοποιημένου for βρόγχου είναι εξ' ορισμού private.

Η Sun προσφέρει τα εξής αναγνωριστικά για την χρήση του autoscoping:

- ***_auto(****λίστα μεταβλητών* ***):*** Προσδιορίζονται από τον χρήστη οι μεταβλητές, των οποίων το scope θα καθοριστεί με βάση την απόφαση του autoscoping.
- ***default****(_auto):* Το autoscoping τίθεται ως προεπιλογή. Όσες μεταβλητές στερούνται scoping από τον χρήστη, υπόκεινται αυτόματα στο autoscoping.

Παράδειγμα χρήσης του autoscoping της Sun [8]:

```
1.  void main( ){
2.
3.          float X[10], Y[10];
4.          float W = 0., MM, M, T;
5.
6.          #pragma omp parallel default(__auto)
7.          {
8.                  #pragma omp single
9.                  {
10.                         M = 0.;
11.                 }
12.                 MM = 0.;
```

```
13.
14.                 #pragma omp for
15.                 for( int i=0; i < 10; i++)
16.                 {
17.                         T = X[i];
18.                         Y[i] = T;
19.
20.                         if( MM > T ){
21.                                 W = W + T;
22.                                 MM = T;
23.                         }
24.                 }
25.
26.                 #pragma omp critical
27.                 {
28.                         if( MM > M )
29.                                 M = MM;
30.                 }
31.         }
32.
33.         return;
34.  }
```

Τα αποτελέσματα του autoscoping του μεταφραστή της Sun είναι τα εξής:

```
Variables autoscoped as SHARED in R1: M, X , Y
Variables autoscoped as PRIVATE in R1: MM, i, T
Variables autoscoped as REDUCTION of operator + in R1: W
```

Παρακάτω, θα εξηγήσουμε αναλυτικά τα αποτελέσματα τις επιλογές του μεταφραστή, με βάση τους

κανόνες που περιγράψαμε παραπάνω.

- Ο ακέραιος i ορίστηκε private, ως μετρητής παραλληλοποιημένου for loop.
- Η εντολή στη γραμμή 12, προφανώς θέτει τον δεκαδικό ΜΜ σε ανταγωνισμό δεδομένων, και εφόσον πρόκειται για εντολή εγγραφής την οποία εκτελούν όλα τα νήματα, η ΜΜ είναι private σύμφωνα με τον κανόνα 2.
- Η εκχώρηση στην μεταβλητή Τ της γραμμής 17, δημιουργεί ανταγωνισμό δεδομένων με τις γραμμές 20, 21, 22 και όμοια με την ΜΜ ορίζεται private.
- Η γραμμή 21 θέτει την μεταβλητή W σε ανταγωνισμό δεδομένων, όμως η μόνη πράξη που τελικά εκτελείται σ' αυτήν είναι μια προσαύξηση. Άρα ορίζεται σε reduction(+:W).
- Ο πίνακας Χ δεν υπόκειται πουθενά σε εγγραφή, και άρα σύμφωνα με τον κανόνα 1 ορίζεται ως shared.
- Οι εγγραφές της γραμμής 18 στον πίνακα Υ, θα διαμοιραστούν μεταξύ των νημάτων, αφού οι εγγραφές έχουν για index τον μετρητή i. Έτσι ο Υ μπορεί να οριστεί ως shared, σύμφωνα με τον κανόνα 1, αφού δεν δημιουργείται ανταγωνισμός δεδομένων.
- Η μεταβλητή Μ γράφεται από ένα μοναδικό νήμα στην γραμμή 10. Η εντολή αυτή όμως είναι αδύνατο να προκαλέσει data race, καθώς υπάρχει υπονοούμενο barrier στο τέλος του single. Οι γραμμές 28 και 29 βρίσκονται μέσα σε κρίσιμη περιοχή, και έτσι ούτε εδώ έχουμε δεδομένων. Πάλι σύμφωνα με τον κανόνα 1 επομένως, η Μ είναι shared.

5.3 Υλοποίηση autoscoping στον OMPi

Η υλοποίηση του autoscoping στον OMPi, έγινε με πρότυπο την υλοποίηση της Sun, με την διαφορά ότι στους 3 κανόνες της Sun προστέθηκε και ένας τέταρτος:

4. Αν όλα τα νήματα που συμμετέχον στην παράλληλη περιοχή, διαβάζουν την μεταβλητή, προτού εκχωρήσουν τιμή σε αυτήν, η μεταβλητή ορίζεται ως ***firstprivate***.

Ο κανόνας αυτός έχει προταθεί, στην εφαρμογή του autoscoping σε OpenMP tasks [9], στον μεταφραστή Mercurium [10].

Η λογική πίσω από τον κανόνα είναι απλή: εφ' όσον περάσαμε από τον κανόνα 1 υπάρχει data race, άρα δεν μπορεί να θεωρηθεί shared. Εφ' όσον όλα τα νήματα διαβάζουν αρχικά την τιμή της, πρέπει να διαβάσουν την τρέχουσα τιμή της μεταβλητής, αυτό που εκφράζει και το firstprivate.

Παρατηρούμε ότι οι απαιτήσεις των 4 κανόνων, καλύπτονται σε μεγάλο βαθμό από την λύση του προβλήματος εγγραφής/ανάγνωσης που περιγράφεται στην ενότητα 4.3. , με μια σημαντική διαφορά: δεν μας αρκεί για να αποφανθούμε αν μια μεταβλητή υπόκειται σε ανταγωνισμό δεδομένων ή όχι. Έτσι, στην υλοποίηση προστέθηκε ένας μηχανισμός ο οποίος είναι σε θέση να εντοπίσει τον ανταγωνισμό δεδομένων. Ο μηχανισμός αυτός διέπεται από την εξής λογική:

Έστω μεταβλητή x. Ορίζουμε τα εξής:

- Μια εκχώρηση της x, μέσα σε παράλληλη περιοχή, θεωρείται ***προστατευμένη***, αν περικλείεται σε μια critical, atomic, single ή master περιοχή.
- Δύο εμφανίσεις της x, θεωρούνται ***κοινώς προστατευμένες***, αν περιέχονται και οι δύο στην ίδια critical, atomic, single ή master περιοχή.
- Μια εμφάνιση της x σε παράλληλη περιοχή, θεωρείται ***ύποπτη για ανταγωνισμό***, αν από το σημείο στο οποίο προσπελάστηκε, δεν έχει μεσολαβήσει σημείο συγχρονισμού (barrier).

Ο αλγόριθμος για τον εντοπισμό συνοψίζεται ως εξής:

- Αν εντοπιστεί απροστάτευτη εγγραφή μεταβλητής, τότε υπάρχει γι' αυτήν ανταγωνισμός δεδομένων.
- Αν εντοπιστεί απροστάτευτη ανάγνωση ή προστατευμένη εγγραφή της x, και υπάρχει ενεργή ύποπτη για ανταγωνισμό εμφάνισή της: αν οι δύο εμφανίσεις δεν είναι κοινώς προστατευμένες, ελέγχεται αν η σύγκρουση ύποπτων εμφανίσεων προκαλεί ανταγωνισμό. Ο έλεγχος γίνεται με βάση τον συμμετρικό πίνακα συγκρούσεων 5.1.
- Αν εμφανιστεί barrier, οι ύποπτες εμφανίσεις διαγράφονται.

	R	W	Single R	Single W	Master R	Master W	Critical R	Critical W	Atomic R	Atomic W
R		✓		✓		✓		✓		✓
W	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Single R		✓		✓		✓		✓		✓
Single W	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Master R		✓		✓				✓		✓
Master W	✓	✓	✓	✓			✓	✓	✓	✓
Critical R		✓		✓		✓				✓
Critical W	✓	✓	✓	✓	✓	✓			✓	✓
Atomic R		✓		✓		✓		✓		
Atomic W	✓	✓	✓	✓	✓	✓	✓	✓		

Πίνακας 5.1*: Ο πίνακας των συγκρούσεων που προκαλούν ανταγωνισμό δεδομένων.*

Το 'R' υποδηλώνει ανάγνωση, ενώ αντίστοιχα το 'W' εγγραφή. Π.χ. το 'Critical W' υποδηλώνει εγγραφή που παρουσιάζεται μέσα σε κρίσιμη περιοχή. Οι γεμάτες θέσεις του πίνακα, υποδεικνύουν ανταγωνισμό δεδομένων.

5.3.1Κατασκευή συνόλων autoscoping

Ορίζουμε την δομή:

```
typedef struct _autoScopeSets{
	set autoShared;
	set autoPrivate;
	set autoFirstPrivate;
	set autoUnk;
	set autoReduction;
} autoScopeSets;
```

Η κεντρική συνάρτηση της ανάλυσης ροής είναι η `autoScopeSets dfa_show(aststmt tree)`, η οποία γεμίζει τα σύνολα της παραπάνω δομής με τα αποτελέσματα του autoscoping, και παίρνει σαν όρισμα έναν οποιονδήποτε κόμβο του συντακτικού δέντρου, εν προκειμένω τη ρίζα του υπόδεντρου που περιγράφει μια παράλληλη περιοχή.

5.3.2 Το αναγνωριστικό auto

Για να δοθεί η δυνατότητα στον χρήστη να αιτηθεί autoscoping σε μία λίστα μεταβλητών, προστέθηκε, στις δυνατές επιλογές κατηγοριοποίησης μιας μεταβλητής, που συναντάται σε παράλληλη περιοχή, η επιλογή ***aut****o*(*λίστα μεταβλητών*)**,** ως παράμετρος μια παράλληλης περιοχής**.** Έτσι, ο χρήστης προσδιορίζει ένα σύνολο μεταβλητών, για την κατηγοριοποίηση των οποίων θα αποφασίσει ο OMPi.

Επίσης, ο χρήστης μπορεί να καθορίσει την αυτόματη κατηγοριοποίηση ως προεπιλογή, κάνοντας χρήση της παραμέτρου ***default(auto).*** Έτσι, όσες μεταβλητές δεν κατηγοριοποιούνται από τον χρήστη, υπόκεινται κατηγοριοποίηση με βάση την απόφαση του autoscoping που εκτελείται από τον OMPi.

5.3.3 Εφαρμογή του autoscoping

Ο OMPi μετασχηματίζει μια παράλληλη περιοχή (*#pragma omp parallel*). μέσω του αρχείου ***x_parallel.c* .** Εκεί, η συνάρτηση *store_dataclause_var,* αναλαμβάνει να χειριστεί τις μεταβλητές που κατηγοριοποιούνται στις παραμέτρους της παράλληλης περιοχής (αν αυτές υπάρχουν). Για κάθε μια παράμετρο που βρίσκει, καλεί την συνάρτηση *store_varlist_vars*, για να τις αποθηκεύσει κατάλληλα στον πίνακα συμβόλων *dataclause_vars*, έτσι ώστε να υποστούν τους κατάλληλους μετασχηματισμούς.

Επομένως, όταν κληθεί η *store_varlist_vars*, για να χειριστεί την παράμετρο (αν υπάρχει) ***auto,*** για κάθε μια μεταβλητή που συναντάει:

- Ελέγχει τα σύνολα τύπου *autoScopeSets*, τα οποία περιέχουν τα αποτελέσματα του autoscoping, και βλέπει ως τί έχει κατηγοριοποιηθούν η συγκεκριμένη μεταβλητή.
- Καταχωρεί την μεταβλητή στον πίνακα συμβόλων *dataclause_vars,* σύμφωνα με την παραπάνω κατηγοριοποίηση.

Έτσι, όταν ο μετασχηματισμός αρχίσει, οι μεταβλητές αυτές θα θεωρηθούν κατηγοριοποιημένες και ως τέτοιες θα μετασχηματιστούν. Για να αποσαφηνιστεί περαιτέρω η διαδικασία αυτή, παρουσιάζονται και αναλύονται κάποιες ενδεικτικές περιπτώσεις χρήσης του autoscoping στον OMPi.

Παράδειγμα 5.1

```
1.        int  x, y = 1, z, w, i;
2.
3.       #pragma omp parallel auto(x,y,z,w)
4.       {
5.              #pragma omp single nowait{
6.                     z = 0;
7.              }
8.
9.              x = y + w;
10.
11.             #pragma omp single
12.             {
13.                    w = y + z;
14.             }
15.      }
```

Τα αποτελέσματα του autoscoping είναι:

Shared:	*y*
Private:	*x*
Firstprivate:	w
Αδύνατη κατηγοριοποίηση:	z

Η εξήγηση των αποφάσεων είναι η εξής:

- Οι προσπελάσεις της y στις γραμμές 13 και 9 δεν είναι κοινώς προστατευμένες, όμως από την θέση [*R*][*Critical R*] του πίνακα, διαπιστώνουμε ότι δεν υπάρχει ανταγωνισμός, άρα σύμφωνα με τον ***κανόνα 1*** η y είναι shared.
- Η εκχώρηση της γραμμής 9 είναι απροστάτευτη και όλα τα νήματα εκτελούν εγγραφή επάνω στην x. Άρα, σύμφωνα με τον ***κανόνα 2***, ορίζεται ως private.
- Η z, θα υποστεί την εγγραφή της γραμμής 6 από ένα νήμα της ομάδας. Εφόσον υπάρχει η παράμετρος *nowait*, η εγγραφή αυτή έρχεται σε σύγκρουση με τις προσπελάσεις της γραμμής 13, παρά το γεγονός ότι αυτές μεταξύ τους είναι συγχρονισμένες από το *critical*. Έτσι, από τη θέση [*Single W*][*Critical R*] του πίνακα, υπάρχει ανταγωνισμός. Όμως, ένα νήμα εκτελεί πρώτα εγγραφή, ενώ τα υπόλοιπα ανάγνωση. Επομένως, η z δεν υπάγεται σε ***κανέναν*** από τους 4 κανόνες, η κατηγοριοποίηση κρίνεται αδύνατη, και ο χρήστης ειδοποιείται για την αιτία.

- Οι εγγραφές της γραμμής 13 στο w, ομοίως, έρχονται σε σύγκρουση με τις αναγνώσεις της γραμμής 9, το οποίο από τη θέση [*R*][Single *W*] βλέπουμε ότι προκαλεί ανταγωνισμό. Όλα τα νήματα εκτελούν αρχικά την ανάγνωση της γραμμής 9, οπότε σύμφωνα με τον ***κανόνα 4***, η w ορίζεται ως *firstprivate*. □

Στο παραπάνω παράδειγμα, πέραν του σημείου συγχρονισμού που υπάρχει στο τέλος της παράλληλης περιοχής, δεν υπάρχει κάποιο άλλο: δεν υπάρχει κάποιο directive τύπου barrier, και το υπονοούμενο barrier στο τέλος της σειριακής περιοχής single ακυρώνεται μέσω της παραμέτρου nowait. Επομένως, όλες οι εμφανίσεις μεταβλητών, παραμένουν ύποπτες για ανταγωνισμό, καθ' όλη τη διάρκεια της διαδικασίας που παρουσιάζεται.

5.3.4 Υποστήριξη εμφωλευμένου παραλληλισμού

Στον τομέα του εμφωλευμένου (nested) παραλληλισμού, η υλοποίησή μας, αν και βασίστηκε στην υλοποίηση της Sun [8], διαχωρίζεται από αυτήν, με σκοπό να την επεκτείνει. Με αναγωγή των περιπτώσεων εμφάνισης μιας μεταβλητής σε εμφωλευμένη παράλληλη περιοχή, σε απλούστερες μη εμφωλευμένες περιπτώσεις, ορίσαμε κάποιους νέους κανόνες, για τις περιπτώσεις φωλιάσματος, με σκοπό να διατηρηθούν οι θεμελιώδεις αρχικοί 4 κανόνες. Ας εξετάσουμε, αρχικά, τις απαντήσεις που δίνει ο μεταφραστής της Sun, ως προς το autoscoping, σε 2 χαρακτηριστικά παραδείγματα:

Παράδειγμα 5.2

```
1.          int x;
2.
3.    R1:   #pragma omp parallel __auto(x)
4.          {
5.    R2:           #pragma omp parallel __auto(x)
6.                  {
7.                          x = 1;
8.                  }
9.          }
```

Τα αποτελέσματα του autoscoping της Sun είναι:

```
Variables autoscoped as
PRIVATE in R1: x

Variables autoscoped as
PRIVATE in R2: x
```

Παρατηρούμε, ότι ο μεταφραστής της Sun όρισε τον ακέραιο x, ως private και στις δύο παράλληλες περιοχές R1 και R2.

- Όσον αφορά την περιοχή R2, η απόφαση συνάδει με τους κανόνες που περιγράψαμε στην ενότητα 5.2: η x δημιουργεί ανταγωνισμό, γράφεται απ' όλους και σύμφωνα με τον κανόνα 2, ορίζεται ως *private.*
- Ας εξετάσουμε την παράλληλη περιοχή R1: η μόνη εμφάνιση του x είναι μέσω της R2, μέσα στην οποία έχουμε δώσει στην x την κατηγοριοποίηση *private.* Αυτό σημαίνει όμως, ότι η αρχική μεταβλητή x μένει ανέπαφη: κάθε νήμα που ανήκει σε παράλληλη υποομάδα που δημιουργείται στο R2, θα έχει το δικό του αντίγραφο της x. Επομένως, ο ακέραιος x, ως προς την R1, δεν έχει ανταγωνισμό δεδομένων, και το να την ορίσουμε ως *private,* αντιβαίνει τον ***κανόνα 1:*** θα έπρεπε να οριστεί ως shared. □

Παράδειγμα 5.3

```
1.          int y;
2.
3.    R3.   #pragma omp parallel __auto(y)
4.          {
5.    R4.           #pragma omp parallel __auto(y)
6.                  {
7.                            #pragma omp single
8.                            {
9.                                    y = 2;
10.                           }
11.                 }
12.         }
```

Τα αποτελέσματα του autoscoping της Sun είναι:

```
Variables autoscoped as
SHARED in R3: y

Variables autoscoped as
SHARED in R4: y
```

Εδώ, η μεταβλητή y ορίζεται ως *shared,* τόσο στην παράλληλη περιοχή R3, όσο και στην R4.

- Στην περιοχή R4, η απόφαση συμβαδίζει και πάλι με τους κανόνες της ενότητας 5.2: η εγγραφή της y είναι προστατευμένη σε κρίσιμη περιοχή, άρα δεν υπάρχει ανταγωνισμός και σύμφωνα με τον κανόνα 1 η y ορίζεται ως *shared*.
- Στην R3, τα πράγματα είναι λίγο πιο σύνθετα: στην γραμμή 5, κάθε νήμα της R3 δημιουργεί μια παράλληλη υποομάδα και ενσωματώνεται σ' αυτήν. Η εγγραφή της γραμμής 9, γίνεται από ένα νήμα της κάθες υποομάδας, χωρίς όμως να υπάρχει κάποιος συγχρονισμός μεταξύ τους. Έτσι, τα αρχικά νήματα της R3 δημιουργούν ανταγωνισμό για την y. Το να την ορίσουμε ως *shared*, αντιβαίνει με τον ***κανόνα 1:*** θα έπρεπε να οριστεί ως *private.* □

Παρατηρούμε λοιπόν, ότι σε περιπτώσεις εμφωλευμένου autoscoping, η υλοποίηση της Sun δεν διαθέτει συνέπεια με τους κανόνες της. Παρακάτω, παροσιάζεται η ιδέα που υλοποιήθηκε στην εργασία αυτή, με σκοπό να διατηρήσουμε τους 4 κανόνες που ορίσαμε στις ενότητες 5.2 και 5.3 αναλλοίωτους σε περιπτώσεις εμφωλευμένου παραλληλισμού.

Ισχυρισμός: Έστω μεταβλητή X στην οποία εκτελείται εγγραφή (εκτός κρίσιμης περιοχής) σε παράλληλη περιοχή επιπέδου N+1, όπου η X έχει οριστεί ως *shared* στο επίπεδο N+1. Τότε, από την σκοπιά της παράλληλης περιοχής επιπέδου N, δημιουργείται ανταγωνισμός για την X.

Απόδειξη: Η X είναι *shared* στο επίπεδο N+1, και άρα εγγραφή σ' αυτό το επίπεδο, σημαίνει εγγραφή στο επίπεδο N. Ενόσω βρισκόμαστε στο επίπεδο N+1, και η εγγραφή γίνεται εκτός κρίσιμης περιοχής, γίνεται ασυχρόνιστα μεταξύ των νημάτων επιπέδου N. Επομένως πρόκειται για ασύγχρονη εκχώρηση δεδομένων στο επίπεδο N, άρα έχουμε ανταγωνισμό δεδομένων στο επίπεδο N.
□

Με βάση τον παραπάνω ισχυρισμό, ορίζουμε τους εξής κανόνες, για την περίπτωση του autoscoping σε εμφωλευμένες παράλληλες περιοχές:

Κανόνες 5.3.4

Έστω περιοχή επιπέδου *N+1*, και μεταβλητή *X*:

1. Αν η *X* είναι *private* στο επίπεδο *N+1*, τότε αυτό δεν επηρεάζει καθόλου την κατηγοριοποίηση της *X* στο επίπεδο *N*: για το *N+1* δημιουργούνται αντίγραφα της *X* τα οποία χάνονται μετά το πέρας το επιπέδου αυτού.
2. Αν η *X* είναι *firstprivate* στο επίπεδο *N+1*, τότε αυτό ισοδυναμεί με μια ανάγνωση επιπέδου *N*: η τιμή της *X* διαβάζεται μια φορά, για να αρχικοποίηθούν κατάλληλα τα αντίγραφά της.
3. Αν η *X* είναι *lastprivate* στο επίπεδο *N+1* (ισχύει αν πρόκειται για βρόγχο for ή sections), τότε

αυτό ισοδυναμεί με μία εγγραφή επιπέδου *N*: το νήμα που θα εκτελέσει την τελευταία επανάληψη, θα κάνει μία εγγραφή στο *N*.

4. Αν η *X* είναι *shared* στο επίπεδο *N+1*, τότε σύμφωνα με τον ισχυρισμό μας, ισχύουν τα εξής:

I. Αν γίνεται εγγραφή εκτός κρίσιμης περιοχής επιπέδου *N+1*, ισοδυναμεί με απροστάτευτη εγγραφή επιπέδου *N*.

II. Αν γίνεται ανάγνωση επιπέδου *N+1*, ισοδυναμεί με ανάγνωση επιπέδου *N*.

Οι παραπάνω κανόνες, μας επιτρέπουν, σε συνδυασμό με αναδρομική επίλυση του autoscoping στις εμφωλευμένες παράλληλες περιοχές, να εξασφαλίσουμε την συνέπεια των αρχικών μας κανόνων σε οσοδήποτε βάθος παραλληλισμού. Ας εξετάσουμε ένα παράδειγμα στην υλοποίηση του OMPi.

Παράδειγμα 5.4

```
1.          int x,y;
2.    R5.   #pragma omp parallel auto(x,y)
3.          {
4.
5.                #pragma omp atomic
6.                      x = 2 + y;
7.
8.    R6.             #pragma omp parallel auto(x,y)
9.                    {
10.                       #pragma omp single
11.                       {
12.                             y = x;
13.                       }
14.                       x = 0;
15.                   }
16.         }
```

Τα αποτελέσματα του autoscoping είναι:

For #parallel at line 2:

Private:	x
Firstprivate:	y

For #parallel at line 8:

Shared:	y
Firstprivate:	x

Για την περιοχή R6:

- Το y υπόκειται προστατευμένη εγγραφή στην γραμμή 12, και άρα ορίζεται ως *shared* από τον κανόνα 1.
- Το x υπόκειται απροστάτευτη εγγραφή στη γραμμή 14, και ορίζεται ως *private* από τον κανόνα 2.

Για την περιοχή R5:

- Το x υπόκειται προστατευμένη εγγραφή στη γραμμή 6, η οποία έρχεται σε σύγκρουση με την εγγραφή που προκύπτει από την κατηγοριοποίηση του ως *firstprivate* στην γραμμή 8 (κανόνας 5.3.4.2). Άρα από τον κανόνα 2, η x ορίζεται ως *private*.
- Το y υπόκειται ανάγνωση από όλα τα νήματα στη γραμμή 6. Αφού το y ορίζεται ως *shared* στη γραμμή 8,α πό την εμφωλευμένη μη κρίσιμη εγγραφή της γραμμής 12, προκύπτει ανταγωνισμός (κανόνας 5.3.4.4.I). Άρα από τον κανόνα 4, η y ορίζεται ως *firstprivate*.

Κεφάλαιο 6

Πειραματικά αποτελέσματα

6.1 NAS Parallel Benchmarks

Τα NAS Parallel Benchmarks (NPB) [12], είναι σύνολο προγραμμάτων ανεπτυγμένων από την NASA Advanced Supercomputing Division, που έχουν ως στόχο την αξιολόγηση παράλληλων συστημάτων και υπερυπολογιστών. Στην περίπτωση που για την παραλληλοποίηση χρησιμοποιείται το μοντέλο OpenMP, τα NPB μπορούν να αποτελέσουν σημείο αναφοράς, για την αποδοτικότητα την οποία πετυχαίνει ο μεταφραστής κατά την παραλληλοποίηση.

Τα 8 προγράμματα αναφοράς είναι τα εξής:

- ***IS (Integer Sort)*** : Πρόκειται για την παράλληλη ταξινόμηση N μοναδικών ακεραίων. Οι ακέραιοι αυτοί παράγονται από μια γεννήτρια ψευδοτυχαίων αριθμών και διαμοιράζονται αρχικά ομοιόμορφα στην μνήμη, έτσι ώστε να αξιολογηθεί η τυχαία προσπέλαση της μνήμης (random memory access) καθώς και η απόδοση του συστήματος σε υπολογισμούς ακεραίων.

- ***EP (Embarrassingly Parallel)*** : Παράγονται ζεύγη αριθμών τα οποία ακολουθούν την κανονική (Γκαουσιανή) κατανομή, με μέση τιμή 0 και διακύμανση 1, δηλαδή με συνάρτηση πυκνότητας πιθανότητας:

$$f\ (x) = \frac{1}{\sqrt{2\pi}}\, \mathrm{e}^{-x^2/2}$$

  Τέλος, για $0 \leqslant i \leqslant 9,$ προσμετρούνται τα ζεύγη τέτοιων αριθμών $(X_\kappa, Y_\kappa),$ για τα οποία ισχύει $i \leqslant max\ (|X_k|\ ,|Y_k|) < i + 1.$ Οι μετρήσεις αυτές εξάγονται, καθώς και τα αθροίσματα $\sum X_\kappa, \sum Y_\kappa.$

  Η μόνη απαιτούμενη επικοινωνία των εκτελεστικών οντοτήτων, είναι ο συνδυασμός των επιμέρους μετρήσεων, για την εξαγωγή των 10 αποτελεσμάτων. Με την ελαχιστοποίηση των επικοινωνιών, η εφαρμογή αυτή μας παρέχει μια προσέγγιση για το άνω φράγμα των πράξεων κινητής υποδιαστολής που μπορεί να εκτελέσει το παράλληλο σύστημά μας ανά μονάδα χρόνου (FLOPS).

- ***CG (Conjugate Gradient)*** : Χρησιμοποιείται η επαναληπτική *μέθοδος των δυνάμεων*, σε έναν αραιό πίνακα A, με τυχαίο μοτίβο μη μηδενικών τιμών: παράγεται μια ακολουθία διανυσμάτων, η οποία συγκλίνει σε ιδιοδιάνυσμα που αντιστοιχεί στη μεγαλύτερη, κατ' απόλυτη τιμή, ιδιοτιμή του A [13].

  Η μέθοδος αυτή αξιολογεί την άτακτη προσπέλαση μνήμης (irregular memory access), καθώς εκτελούνται πράξεις πλέγματος χωρίς σχεδιασμό (unstructured grid computations).

- ***MG (Multi-Grid)***: Εφαρμόζεται η επαναληπτική πολυπλεγματική μέθοδος V-cycle, σε τρισδιάστατο κυβικό πλέγμα διάστασης 256, με οριακές συνθήκες, η οποία παράγει προσεγγιστική λύση *u*, στην διακριτή μερική διαφορική εξίσωση Poisson:

$$\nabla^2 u(x,y,z) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = v$$

  Η επίλυση της εξίσωσης αυτής, χρησιμοποιείται στην επεξεργασία γραφικών, για διαδικασίες όπως είναι η κλωνοποίηση χωρίς ραφές (seamless cloning) [14].

  Η εφαρμογή αυτή, είναι πολύ απαιτητική σε μνήμη, και αξιολογεί την απόδοση στις υψηλά σχεδιασμένες και δομημένες επικοινωνίες (highly structured communication), μεταξύ των εκτελεστικών οντοτήτων.

- ***FT (Fourier Transform)***: Θεωρούμε την μερική διαφορική εξίσωση:

$$\frac{\partial u(x,t)}{\partial t} = a\nabla^2 u(x,t)$$

  όπου το *x* εκφράζει σημείο στον τρισδιάστατο χώρο. Στην εφαρμογή αυτή, επιλύεται η διακριτή μορφή της παραπάνω εξίσωσης, με χρήση του τρισδιάστατου διακριτού *γρήγορου μετασχηματισμού Fourier (FFT)*.

  Εδώ, αξιολογείται η συλλογική επικοινωνία, για πράξεις πινάκων όπως είναι η αντιστροφή: ο πίνακας βρίσκεται διαμοιρασμένος κατά στήλες σε κάθε εκτελεστική οντότητα. Μετά την αντιστροφή, ο πίνακας θα πρέπει να έχει διαμοιραστεί κατά γραμμές, επομένως απαιτούνται συλλογική διασπορά (scatter) και συλλογή (gather) των δεδομένων του πίνακα.

Οι 3 τελευταίες εφαρμογές, έχουν ως στόχο να εξομοιώσουν προβλήματα που προκύπτουν συχνά, σε εφαρμογές στον τομέα της *υπολογιστικής ρευστοδυναμικής*. Έτσι, εξετάζεται η απόδοση στους υπολογισμούς και την μεταφορά των δεδομένων, σε ρεαλιστικές εφαρμογές state-of-the-art:

- ***BT (Block Tri-diagonal solver)***: Επιλύονται πολλαπλά, ανεξάρτητα μεταξύ τους γραμμικά συστήματα, το οποία διαθέτουν μη κυρίαρχη διαγώνιο, και είναι ανά 5x5 block τριδιαγώνια.
- ***SP (Scalar Penta-diagonal solver)***: Επιλύονται πολλαπλά, ανεξάρτητα μεταξύ τους

πενταδιαγώνια γραμμικά συστήματα, το οποία διαθέτουν μη κυρίαρχη διαγώνιο.

- ***LU (Lower-Upper Gauss-Seidel solver)***: Χρησιμοποιείται η μέθοδος της συμμετρικής διαδοχικής υπερχαλάρωσης (SSOR), η οποία αποτελεί μια ταχύτερης σύγκλισης παραλλαγή της μεθόδου Gauss-Seidel, για την επίλυση ενός μέτριας πυκνότητας γραμμικού συστήματος, το οποίο είναι ανά 5x5 block άνω-κάτω τριγωνικό.

6.1.1 Εφαρμογή του autoscoping στα NPB

Στην έκδοση NPB 2.3 από την NASA, χρησιμοποιήθηκε γλώσσα προγραμματισμού Fortran και περιλάμβανε την σειριακή έκδοση των 8 προγραμμάτων, καθώς και την παραλληλοποίηση τους μέσω του μοντέλου ανταλλαγής μηνυμάτων MPI. Οι σειριακές εκδόσεις μεταφράστηκαν σε γλώσσα προγραμματισμού C και παραλληλοποιήθηκαν με το μοντέλο OpenMP, στα πλαίσια της παρουσίασης του πηγαίου αρχείου σε πηγαίο αρχείο (source-to-source) μεταφραστή Omni [15]. Οι παραλληλοποιήσεις αυτές [16], χρησιμοποιήθηκαν για την αξιολόγηση της εφαρμογής του autoscoping στον OMPi, που αναπτύχθηκε στα πλαίσια αυτής της εργασίας.

Τα αποτελέσματα συνοψίζονται στον πίνακα 6.1.

ΕΦΑΡΜΟΓΗ	ΠΑΡΑΛΛΗΛΕΣ ΠΕΡΙΟΧΕΣ	ΚΑΤΗΓΟΡΙΟ-ΠΟΙΗΜΕΝΕΣ ΜΕΤΑΒΛΗΤΕΣ	ΡΗΤΑ ΚΑΤΗΓΟΡΙΟ-ΠΟΙΗΜΕΝΕΣ	ΕΠΙΤΥΧΗΜΕΝΟ AUTOSCOPING	ΠΟΣΟΣΤΟ ΕΠΙΤΥΧΙΑΣ
Integer Sort (IS)	2	11	0	11	100%
Embarrassingly Parallel (EP)	1	9	0	7	78%
Conjugate Gradient (CG)	5	45	8	42	93%
Multi-Grid (MG)	3	28	2	27	96%
Fourier Transform (FT)	2	30	0	24	80%
Block Tri-diagonal solver (BT)	2	117	4	108	92%
Scalar Penta-diagonal solver (SP)	2	48	2	48	100%
Lower-Upper solver (LU)	3	132	9	129	98%
ΣΥΝΟΛΟ	20	420	25	396	94%

Πίνακας 6.1: *Αποτελέσματα της εφαρμογής του autoscoping στα NAS Parallel Benchmarks*

Παρατηρούμε ότι το autoscoping ήταν ανεπιτυχές σε 24 από τις 420 περιπτώσεις. Οι λόγοι είναι οι εξής:

- Σε 3 περιπτώσεις, μία μεταβλητή η οποία υπόκειται μόνο ανάγνωση, έχει κατηγοριοποιηθεί στα benchmarks ως *firstprivate:* η επιλογή αυτή αντιβαίνει στον κανόνα 1, έτσι ο OMPi την κατηγοριοποιεί ως *shared*. Καμία από τις 2 επιλογές δεν είναι λάθος: η βέλτιστη επιλογή εξαρτάται από τα χαρακτηριστικά του εκάστοτε συστήματος. Κατά τον ορισμό ως *firstprivate*, δαπανούμε χώρο στην ιδιωτική μνήμη του κάθε νήματος, ενώ στον ορισμό ως *shared*, έχουμε κοινές προσπελάσεις μνήμης που ενδέχεται να είναι απομακρυσμένη.
- Σε 9 περιπτώσεις, το autoscoping είναι ανεπιτυχές αφού οι μεταβλητές περνούν με αναφορά σε κάποια συνάρτηση.
- Σε 11 περιπτώσεις, υπάρχουν προσπελάσεις σε πίνακες, κατά τις οποίες αν και το κάθε νήμα προσπελάσει ξεχωριστή θέση μνήμης, αυτό είναι αδύνατο να επαληθευτεί από την ανάλυση δεδομένων, έτσι υπάρχει λανθασμένη θετική απόκριση για ανταγωνισμό στον πίνακα.
- Σε 1 περίπτωση, μία μεταβλητή η οποία υπόκειται σε ανταγωνισμό, έχει κατηγοριοποιηθεί στα benchmarks ως *shared*, κάτι που αντιβαίνει με τον κανόνα 1.

Κατά την παραλληλοποίηση της εφαρμογής CG, σε μία παράλληλη περιοχή ορίζοντα ρητά ως private 4 μεταβλητές, από τις οποίες μόνο 2 εμφανίζονται στην περιοχή. Αυτό σημαίνει ότι με την αρχική κατηγοριοποίηση, ενδέχεται να δαπανήσουμε ιδιωτική μνήμη των νημάτων, για να δεσμεύσουμε χώρο για μεταβλητές οι οποίες δεν χρησιμοποιούνται. Κάτι τέτοιο δεν θα συνέβαινε εάν εφαρμόζαμε το autoscoping στην περιοχή αυτή.

Κεφάλαιο 7

Επίλογος

Στην εργασία αυτή, παρουσιάσαμε το autoscoping ως επέκταση του παραλληλοποιητικού μοντέλου OpenMP. Ο στόχος της επέκτασης αυτής, δεν είναι να αντικατασταθεί η ρητή κατηγοριοποίηση των μεταβλητών από τον χρήστη: είναι να διευκολυνθεί, να ελεγχθεί και να τελειοποιηθεί. Η εφαρμογή του autoscoping και της ανάλυσης ροής δεδομένων που το συνοδεύει, μας δίνει πλήρη εικόνα για το πώς οι παράλληλες εκτελεστικές οντότητες προσπελάσουν τα κοινά σ' αυτές δεδομένα, κάτι το οποίο αποτελεί πυρηνικό ζήτημα στην παραλληλοποίηση προγραμμάτων. Είτε για λόγους αλγοριθμικής πολυπλοκότητας, είτε για λόγους μεγάλου μεγέθους (σε γραμμές κώδικα) των προγραμμάτων, η διαδικασία αυτή μπορεί να γίνει επίπονη και επισφαλής. Το autoscoping, με σωστή χρήση του, μπορεί να αποτελέσει ένα πολύτιμο εργαλείο που απλοποιεί την προαναφερθείσα διαδικασία.

Αν και η υλοποίηση που παρουσιάστηκε περιορίζεται στο μοντέλο OpenMP, επικεντρώνεται σε ένα ζήτημα το οποίο εισήγαγε η κατάργηση της σειριακότητας των εντολών και απασχολεί τον τομέα του παράλληλου προγραμματισμού γενικότερα: τον *ανταγωνισμό δεδομένων*. Από την αδυναμία πρόβλεψής του, έως και την σπατάλη επικοινωνιών για την αποφυγή του, μπορεί να αποτελέσει αρνητικό παράγοντα για την ποιότητα της παραλληλοποίησης. Παρουσιάσαμε τρόπους για την ανάλυσή του και προσεγγίσαμε θεωρητικά και πρακτικά την ανάπτυξη εργαλείων για τον εντοπισμό του, με στόχο την βελτιστοποιημένη παραλληλοποίηση.

Βιβλιογραφία

[1] OpenMP Application Program Interface,

http://www.openmp.org/mp-documents/spec30.pdf

[2] V.V. Dimakopoulos, E. Leontiadis and G. Tzoumas: A Portable C Compiler for OpenMP V.2.0.
In Proc. EWOMP 2003, 5th European Workshop on OpenMP, Aachen, Germany, Sept. 2003

[3] V.V.Dimakopoulos: OMPi translator internals,

http://www.cs.uoi.gr/~ompi/docs/compiler/ompicompiler.pdf

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers: Principles, Techniques, and Tools

[5] SMECY, Smart multicore embedded systems,

http://www.smecy.eu/

[6] S.N. Agathos, V.V. Dimakopoulos, A. Mourelis, A Papadogiannakis: Deploying OpenMP on an Embedded Multicore Accelerator.

In Proc. SAMOS XIII, 13th Int'l Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation, Samos, Greece, July 2013

[7] Randy Allen, Ken Kennedy: Optimizing Compilers for Modern Architectures: A Dependence-based Approach.

[8] Yuan Lin, Christian Terboven, Dieter an Mey, and Nawal Copty: Automatic Scoping of Variables in Parallel Regions of an OpenMP Program.

In WOMPAT'2004: Workshop on OpenMP Applications and Tools, Houston, Texas, USA, May 2004.

[9] Sara Royuela, Alejandro Duran, Chunhua Liao, Daniel J. Quinlan: Auto-scoping for OpenMP Tasks.

8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012

[10] Programming Models, Barcelona Supercomputing Center*:* Mercurium compiler,

http://pm.bsc.es/mcxx

[11] Michael Voss, Eric Chiu, Patrick Man Yan Chow, Catherine Wong, Kevin Yuen: An Evaluation of autoscoping in OpenMP.

In WOMPAT'2004: Workshop on OpenMP Applications and Tools, Houston, Texas, USA, May 2004.

[12] NASA Advanced Supercomputing Division: NAS Parallel Benchmarks,

http://www.nas.nasa.gov/publications/npb.html

[13] Ταρουδάκη Βικτωρία: Αριθμητικές μέθοδοι για τον υπολογισμό ιδιοτιμών και ιδιοδιανυσμάτων.

Πτυχιακή εργασία, Ιούλιος 2008, Τμήμα Μαθηματικών, Πανεπιστήμιο Κρήτης.

[14] Βρίγκας Μιχαήλ: Συμπλήρωση Εικόνας με χρήση διαφορικών εξισώσεων με μερικές παραγώγους,

Πτυχιακή εργασία,2008, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.

[15] University of Tsukuba, Japan: Omni Compiler Project,

http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler

[16] Omni OpenMP benchmarks,

http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html