ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΗΣΗ OpenMP

Διπλωματική Εργασία

του

Κοντογιάννη Γεώργιου

Θεσσαλονίκη, Οκτώβριος 2020

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΗΣΗ OpenMP

Κοντογιάννης Γεώργιος

Δίπλωμα Πολιτικού Μηχανικού, ΑΠΘ, 2016

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Μαργαρίτης Κωνσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ηη/μμ/εεεε

Ονοματεπώνυμο 1	Ονοματεπώνυμο 2	Ονοματεπώνυμο 3
.....	.....	.....

Κοντογιάννης Γεώργιος

.....

Η σύνταξη της παρούσας εργασίας έγινε στο LaTeX

Περίληψη

Αντικείμενο της παρούσας διπλωματικής εργασίας είναι η μελέτη του OpenMP, ενός πρότυπου παράλληλου προγραμματισμού, που δίνει στο χρήστη τη δυνατότητα αναπτύξης παράλληλων προγραμμάτων για συστήματα μοιραζόμενης μνήμης, τα οποία είναι ανεξάρτητα από τη συγκεκριμένη αρχιτεκτονική και έχουν μεγάλη ικανότητα κλιμάκωσης[12].

Σκοπός της εργασίας είναι η μελέτη και συνοπτική περιγραφή των κύριων χαρακτηριστικών του OpenMP 2.5 αλλά και των νεότερων εκδόσεων 3.0 και 4.5 και η υλοποίηση αλγορίθμων σειριακά και παράλληλα εκτελέσιμων, με σκοπό τη συγκριτική μελέτη της απόδοσής τους. Για την παράλληλη υλοποίηση θα γίνει χρήση της Διεπαφής Προγραμματισμού Εφαρμογών (Application Programming Interface - API) OpenMP, με χαρακτηριστικά που εισήχθησαν στις εκδόσεις OpenMP 3.0 που δημοσιεύθηκε το 2008 και OpenMP 4.5 που δημοσιεύθηκε 2015. Χρησιμοποιήθηκαν επίσης χαρακτηριστικά παλαιότερων εκδόσεων[16].

Τον Μαιο του 2008 κυκλοφόρησαν οι προδιαγραφές του OpenMP 3.0 με την εισαγωγή των διεργασιών (Tasking) αλλά και βελτιώσεις στη C++. Αυτή ήταν η πρώτη ενημέρωση από την έκδοση 2.5 με σημαντικές βελτιώσεις. Το 2011 κυκλοφόρησε το OpenMP 3.1 χωρίς καινούργιο χαρακτηριστικά. Νέα λειτουργικότητα υλοποιήθηκε στο OpenMP 4.0 που κυκλοφόρησε τον Ιούλιο του 2013, όπου έγινε υποστήριξη της αρχιτεκτονικής cc-NUMA, του ετερογενούς προγραμματισμού, της διαχείρισης σφαλμάτων στο μπλοκ παράλληλου κώδικα και της διανυσματικοποίησης μέσω SIMD. Τον Ιούλιο του 2015 σημαντική βελτίωση έγινε στα παραπάνω χαρακτηριστικά με την έκδοση OpenMP 4.5[17].

Τα προαναφερθέντα χαρακτηριστικά χρησιμοποιήθηκαν για την υλοποίηση των αλγορίθμων με διαφορετικές εναλλακτικές μεθόδους, με στόχο τη συγκριτική μελέτη τους για την εξαγωγή συμπερασμάτων αναφορικά με τη βελτίωση της απόδοσης σε σχέση με τη σειριακή υλοποίηση αλλά και τη μεταξύ τους σύγκριση καθώς επίσης, και αξιολόγηση της ευχρηστίας της υλοποίησής τους. Στόχος της έρευνας είναι να βρεθούν οι καλύτερες υλοποιήσεις των αλγορίθμων με την επίτευξη της μέγιστης αξιοποίησης της χρήσης CPU

και/ή GPU. Ακόμη, γίνεται καταγραφή και αναφορά των προβλημάτων που μπορεί να προκύψουν για κάποια υλοποίηση.

Για την παραλληλοποίηση κώδικα, απαιτείται η σχεδίαση με τέτοιο τρόπο ώστε να παράγεται ένας μεγάλος αριθμός παράλληλων λειτουργιών που εκτελούνται από διαφορετικούς επεξεργαστές. Οι αλγόριθμοι που χρησιμοποιήθηκαν στην παρούσα εργασία περιέχουν ένα μεγάλο αριθμών λειτουργιών, ικανών να εκτελεστούν παράλληλα.

Τα βασικότερα παραδείγματα που χρησιμοποιήθηκαν είναι:

- μετασχηματισμός Fourier,
- mergesort,
- υπολογισμός π,
- πολλαπλασιασμός πινάκων,
- απλή εξίσωση διάδοσης θερμότητας,
- παραγοντοποίηση cholensky.

Για να υπάρχει άμεση σύγκριση των αποτελεσμάτων ο βασικός κορμός υλοποίησης είναι ο ίδιος για κάθε εφαρμογή, και οι χρονικές καταγραφές έγιναν σε συγκεκριμένα τμήματα του κώδικα. Οι παραλλαγές του κάθε αλγόριθμοι χρησιμοποιούν την CPU με απλή εκτέλεση χωρίς παραλληλοποίηση και με παράλληλη εκτέλεση. Οπου είναι εφικτό ο αλγόριθμους υλοποιείται για εκτέλεση στην GPU για την επίλυση του. Οι χρονικές καταγραφές συγκρίνονται μεταξύ τους για την εξαγωγή συμπερασμάτων. Ακόμη, γίνεται αξιολόγηση της ευχρηστίας για την υλοποίηση της κάθε παραλλαγής αλλά και προβλημάτων που προέκυψαν.

Λέξεις Κλειδιά: Παράλληλος Προγραμματισμός, Παραλληλοποίηση, OpenMP, accelerators, offloading, vectorization, SIMD, OpenMP4.5, UDRs

Abstract

[Enter abstract here.]

Keywords:

Ευχαριστίες

Εκφράζω τις θερμές μου ευχαριστίες στον επιβλέποντα καθηγητή κ. Κωνσταντίνο Μαργαρίτη, για την ουσιαστική του συνεισφορά στην εκπόνηση της παρούσας εργασίας.

Περιεχόμενα

Κατάλογος Εικόνων (αν υπάρχουν)

Κατάλογος Πινάκων (αν υπάρχουν)

Λίστα Συμβολισμών

1 Εισαγωγή

1.1 Συνοπτικά για το OpenMP

Το OpenMP είναι μια Διεπαφή Προγραμματισμού Εφαρμογών (API) για παραλληλοποίηση συστημάτων μοιραζόμενης μνήμης, σε γλώσσες C/C++ και Fortran. Η διεπαφή αποτελείται από τα παρακάτω σύνολα[17]:

- οδηγιών(directives) για τον μεταγλωττιστή που στόχο έχουν τον καθορισμό και τον έλεγχο της παραλληλοποίησης
- ενσωματωμένων συναρτήσεων της βιβλιοθήκης OpenMP
- μεταβλητών περιβάλλοντος.

Οι οδηγίες εφαρμόζονται στο μπλοκ κώδικα που ακολουθεί της οδηγίας. Κάθε κατασκευή ξεκινάει με #pragma omp και ακολουθούν οδηγίες μεταγλωττιστή για C/C++. Το δομημένο μπλοκ κώδικα αποτελείται από μια απλή εντολή ή ένα σύνολο απλών εντολών[5]. Οι εντολές που βρίσκονται εντός της περιοχής παράλληλου κώδικα, εκτελούνται από όλα τα νηματα. Η παράλληλη εκτέλεση ολοκληρώνεται με το πέρας της εκτέλεσης των εντολών εντος αυτής της περιοχής.

Συμβ. 1: Γραμματική σύνταξης οδηγίας OpenMP

```
#pragma omp (directive) [clause[[,] clause]...] new-line
```

Ετσι οι εφαρμογές εκμεταλλεύονται την ύπαρξη πολλαπλών επεξεργαστικών μονάδων σε έναν πολυεπεξεργαστή, με σκοπό να επιτύχουν αύξηση των υπολογιστικών επιδόσεων και μείωση του απαιτούμενου χρόνου εκτέλεσης της εφαρμογής. Ο παράλληλος προγραμματισμός μπορεί να ιδωθεί ως ειδική περίπτωση ταυτόχρονου προγραμματισμού, όπου η εκτέλεση γίνεται πραγματικά παράλληλα και όχι ψευδοπαράλληλα[21].

Συμβ. 2: Παράδειγμα παράλληλου κώδικα OpenMP

```
#include <omp.h>     // OpenMP include file
#include <stdio.h>   // Include input-output library

int main(void) {
        #pragma omp parallel
        {
                int id = omp_get_thread_num();
                std::cout << "Hello " << id;
                std::cout << "world " << std::endl;
        }
}
```

1.2 Σκοπός – Στόχοι

TODO

Σκόπος είναι η ανάλυση του ΟπενΜΠ και των νεων χαρακτηρίστικών του. Να δουμε τα οφέλει του ΑΠΙ και στόχος ειναι η σύγκριση των νεων εργαλείων και των παλιών και του σειριακου κώδικα για να δούμε τι κερδίζουμε και τι όχι.

1.3 Διάρθρωση της μελέτης

TODO

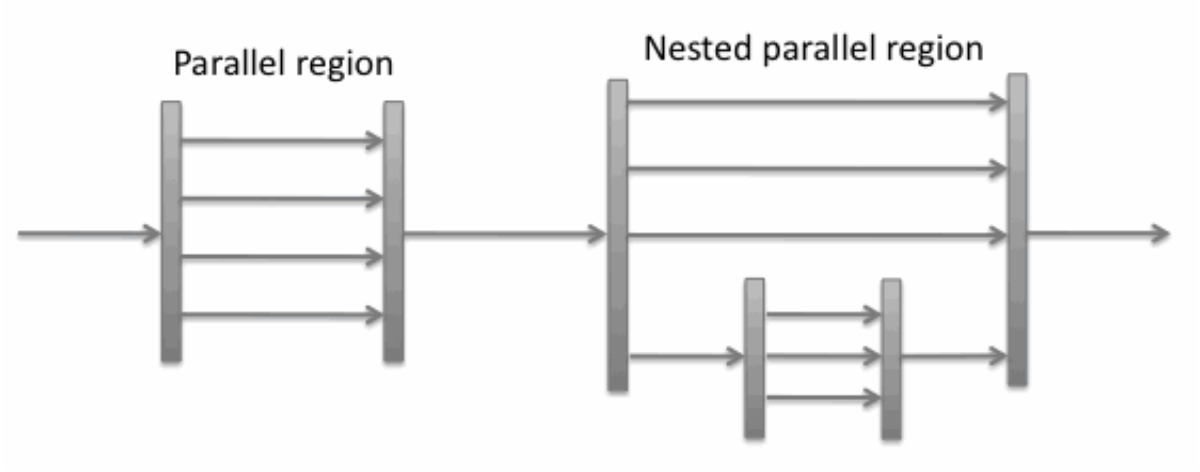
Εδώ περιγράφουμε τα κεφάλαια της διπλωματικής. Συνήθως η ενότητα αυτή έχει την παρακάτω μορφή (δεν θα σας πάρει πάνω από 1 μεγάλη παράγραφο): Εργασίες σχετικές με το αντικείμενο της διπλωματικής παρουσιάζονται στο Κεφάλαιο 2. Το Κεφάλαιο 3 συζητά. . . . Στο Κεφάλαιο 4 αναπτύσσουμε . . . κλπ.

2 Θεωρητικό Υπόβαθρο

2.1 Το μοντέλο Προγραμματισμού OpenMP

Το μοντέλο προγραμματισμού του OpenMP βασίζεται στο πολυνηματικό μοντέλο παραλληλισμού. Η εφαρμογή ξεκινάει με ένα μόνο νήμα, το κύριο (master thread), που εκτελεί εντολές σειριακού κώδικα. Το 'id' αυτού του νήματος είναι πάντα μηδέν και υπάρχει μέχρι το τέλος εκτέλεσης του προγράμματος[12].

Οταν το κύριο νήμα εισέρχεται στο μπλοκ παράλληλου κώδικα (parallel region), τότε δημιουργούνται περισσότερα νήματα και το μπλοκ αυτό εκτελείται παράλληλα. Οταν ολοκληρωθεί ο υπολογισμός της παράλληλης περιοχής, όλα τα νήματα τερματίζουν και συνεχίζει μόνο το κύριο νήμα μέχρι να βρεθεί κάποιο άλλο μπλοκ παράλληλου κώδικα (fork-join μοντέλο)[12]. Το κύριο νήμα είναι υπεύθυνο για την δημιουργία επιπλέον νημάτων για τη συνολική εκτέλεση. Τα νήματα που είναι ενεργά σε μια παράλληλη περιοχή αναφέρονται ως "ομάδα"(thread team). Πολλές ομάδες μπορεί να είναι ενεργές ταυτόχρονα[7].

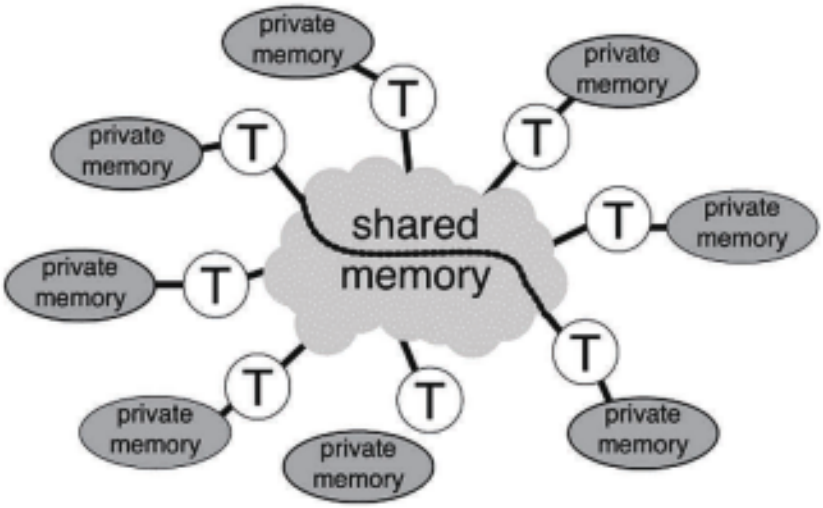


Σχήμα 1: Κύριο νήμα και ομάδες νημάτων

2.2 Αλληλεπίδραση νημάτων και περιβάλλοντος δεδομένων

Η εκτέλεση του προγράμματος ξεκινάει από το κύριο νήμα το οποίο έχει συσχετιστεί με ένα περιβάλλον εκτέλεσης. Το περιβάλλον εκτέλεσης για ένα νήμα είναι ο χώρος διεύθυνσης μνήμης που περιέχει όλες τις μεταβλητές του προγράμματος, περιλαμβανομένων των global μεταβλητών, των μεταβλητών που είναι αποθηκευμένες στη stack και αυτών που είναι αποθηκευμένες στη heap[14].

Στο μοντέλο μνήμης OpenMP, υπάρχουν δύο διαφορετικοί βασικοί τύποι μνήμης: η ιδιωτική(private) και η κοινόχρηστη(shared). Όλα τα νήματα έχουν πρόσβαση σε μεταβλητές που είναι αποθηκευμένες στην κοινόχρηστη μνήμη[19].



Σχήμα 2: Μοντέλο μνήμης OpenMP

2.2.1 Ιδιωτική μνήμη

Κάθε νήμα έχει ιδιωτική μνήμη στην οποία έχει πρόσβαση και μπορεί να την τροποποίηση, αλλά δε μπορεί να έχει πρόσβαση στην ιδιωτική μνήμη άλλου νήματος. Η διάρκεια ζωής μιας μεταβλητής στην ιδιωτική μνήμη είναι περιορισμένη και διαρκεί όσο εκτελείται ο παράλληλος κώδικας. Προεπιλεγμένα, η ιδιωτική μεταβλητή δεν είναι αρχικοποιημένη στην αρχή της παράλληλης περιοχής[20].

2.2.2 Κοινόχρηστη μνήμη

Εκτός από την ιδιωτική μνήμη, κάθε νήμα έχει πρόσβαση και σε ένα άλλο είδος μνήμης, την κοινόχρηστη. Σε αντίθεση με την ιδιωτική, υπάρχει μόνο μία κοινόχρηστη μνήμη κατα τη διάρκεια εκτέλεσης του προγράμματος, και είναι προσπελάσιμη απο όλα τα νήματα. Ετσι, κάθε νήμα έχει την δυνατότητα τροποποίησης οποιασδήποτε μεταβλητής βρίσκεται στη κοινόχρηστη μνήμη. Η ταυτόχρονη προσπέλαση κοινόχρηστης μνήμης από διαφορετικά νήματα, προκαλεί τα παρακάτω προβλήματα:

2.2.2.1 Race Condition

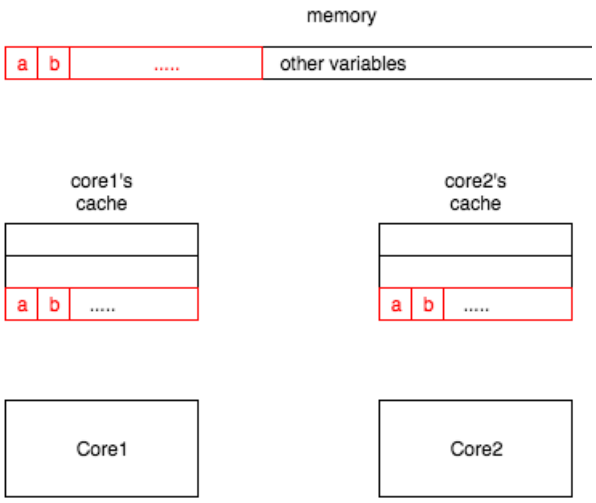
Εμφανίζεται στις περιπτώσεις όπου μια συνάρτηση χρησιμοποιεί δεδομένα από της κοινόχρηστης μνήμης. Αν η συνάρτηση καλείται παράλληλα, πολλά νήματα ενδέχεται να τροποποιήσουν ταυτόχρονα την ίδια διεύθυνση μνήμης. Το φαινόμενο αυτό ονομάζεται ρασε ςονδιτιον και η απλούστερη λύση του ειναι η δημιουργία ιδιωτικού αντίγραφου για κάθε νήμα. Με αυτό τον τρόπο, πολλά νήματα μπορούν ταυτόχρονα να τροποποιούν δεδομένα που βρίσκονται σε διαφορετικές θέσεις μνήμης.

Συμβ. 3: Παράδειγμα κώδικα με race condition

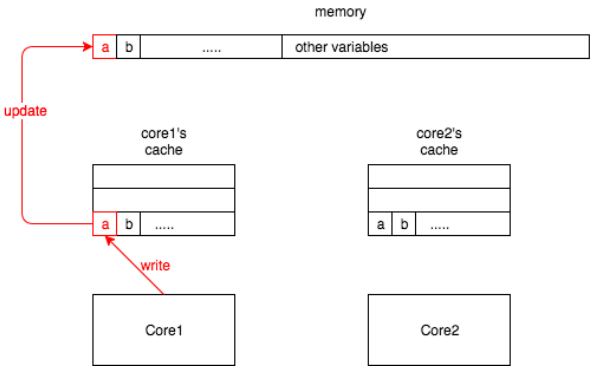
```
#include <omp.h>

int main(void) {
  int sum = 0;

  #pragma omp parallel for
  for (int i = 0;  i < 100; ++i) {
    sum += i;
  }
}
```

2.2.2.2 False Sharing

Το false sharing ειναι ενα συχνό πρόβλημα στην παράλληλη επεξεργασία κοινόχρηστης μνήμης. Εμφανίζεται όταν δυο ή περισσότεροι πυρήνες κρατούν ένα αντίγραφο της ίδιας γραμμής προσωρινής μνήμης.

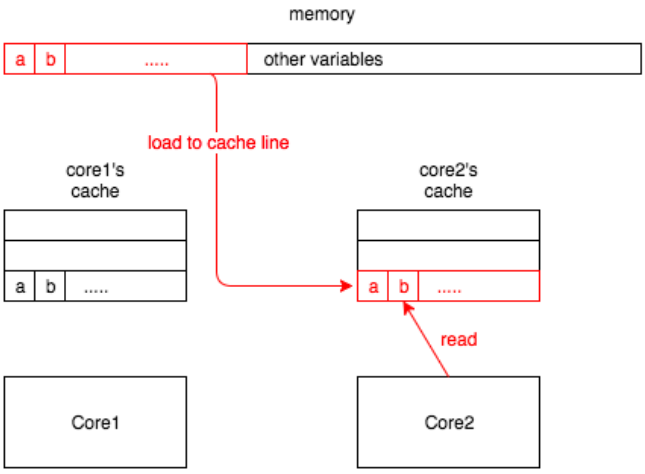


Σχήμα 3: False sharing (1/3)



Σχήμα 4: False sharing (2/3)

Οταν ένας πυρήνας τροποποιεί μια μεταβλητή, η γραμμή της μνήμης που βρίσκεται η μεταβλητή ακυρώνεται σε άλλους πυρήνες. Παρόλο που ένας πυρήνας μπορεί να μή τροποποιεί της συγκεκριμένη θέση μνήμης, μπορεί να χρησιμοποιεί ένα αλλο στοιχείο δεδομένων στην ίδια γραμμή μνήμης.



Σχήμα 5: False sharing (3/3)

Ο δεύτερος πυρήνας θα πρέπει να φορτώσει εκ νέου τη γραμμή προτού αποκτήσει ξανά πρόσβαση στα δεδομένα της . Η χρήση της κοινόχρηστης μνήμης μπορεί να επηρεάσει την απόδοση του προγράμματος[8]. Λύση στο πρόβλημα του false sharing, αποτελεί η εισαγώγή τεχνιτού κενού (padding) ανάμεσα στα στοιχεία ενός πίνακα.

2.3 Μοντέλο συνοχής μνήμης

Για την αποφυγή φαινομένων race condition που οδηγούν σε λανθασμένα αποτελέσματα, είναι συχνά απαραίτητος ο συντονισμός της πρόσβασης των νημάτων στις μεταβλητές της κοινόχρηστης μνήμης. Ο όρος ¨*συγχρονισμός*¨ αναφέρεται στους μηχανισμούς συντονισμού των νημάτων. Οι μέθοδοι συγχρονισμού εγγυώνται το διάβασμα της σωστής τιμής μια μεταβλητής στην κοινόχρηστη μνήμη μετά απο οποιαδήποτε ενημέρωσή της. Μηχανισμοί συγχρονισμού ειναι οι[15]:

- #pragma omp critical
- #pragma omp atomic
- #pragma omp barrier
- #pragma omp ordered
- #pragma omp flush

2.4 Οδηγίες διαμοιρασμού εργασίας

Η εντολή #pragma omp parallel κατασκευάζει ένα SPMD πρόγραμμα ("Single Program Multiple Data") οπου κάθε νήμα εκτελεί τον ίδιο κώδικα. Ο όρος *¨οδηγία διαμοιρασμού εργασίας¨* (worksharing construct) χρησιμοποιείται για να περιγραφεί η κατανομή της εκτέλεσης της αντίστοιχης περιοχής μεταξύ των νημάτων μιας ομάδας που τη συναντά.

Μια οδηγιά διαμοιρασμού εργασίας δεν έχει κανένα εμπόδιο συγχρονισμού(\barrier") κατά την είσοδο. Ωστόσο υπάρχει ένα σιωπηρό εμπόδιο στο τέλος της οδηγίας. Το εμπόδιο μπορεί να αναιρεθεί με τη χρήση της *¨φράσης¨* (clause) *νοωαιτ*. Εάν υπάρχει, το πρόγραμμα μπορεί να παραλείψει το εμπόδιο στο τέλος της οδηγίας. Σε αυτή την περίπτωση, τα νήματα που τελειώνουν νωρίτερα μπορούν να προχωρήσουν στις υπόλοιπες οδηγίες που ακολουθούν στην παράλληλη περιοχή[1].

2.4.1 Οδηγία διαμοιρασμού εργασίας βρόγχου - for

Η οδηγία διαμοιρασμού εργασίας βρόγχου καθορίζει ότι οι επαναλήψεις ενός ή περισσότερων βρόχων θα εκτελούνται παράλληλα από μια ομάδα νημάτων. Οι επαναλήψεις διανέμονται στα ήδη υπάρχοντα νήματα της ομάδας νημάτων της παράλληλης περιοχής.

Συμβ. 4: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

```
#pragma omp for [clause[[ ,]clause]...] new-line
        for-loops
```

Συμβ. 5: Αποδεκτές φράσεις οδηγίας for

```
private(list)
firstprivate(list)
lastprivate([lastprivate-modifier:]list)
linear(list[:linear-step])
reduction([reduction-modifier,]reduction-identifier : list)
schedule([modifier[, modifier]:]kind[, chunk_size])
collapse(n)
ordered[(n)]
allocate([allocator :]list)
order(concurrent)
```

2.4.2 Οδηγία sections

Η οδηγία sections είναι μια μη επαναληπτική περιοχή διαμοιρασμού εργασίας. Καθορίζει ότι τα εσωκλειώμενα τμήματα κώδικα θα διαμοιραστούν μεταξύ των νημάτων της ομάδας. Μια οδηγία sections μπορεί να περιέχει περισσότερες από μία, ανεξάρτητες, οδηγίες σεστιον. Κάθε τμήμα εκτελείται μια φορά από ένα νήμα της ομάδας, ενώ διαφορετικά τμήματα εκτελούνται από διαφορετικά νήματα. Η σύνταξη μιας οδηγίας σεστιονς φαίνεται παρακάτω[12]

Συμβ. 6: Σύνταξη οδηγίας sections

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

Συμβ. 7: Αποδεκτές φράσεις οδηγίας sections

```
private(list)
firstprivate(list)
lastprivate([ lastprivate-modifier:] list)
reduction([reduction-modifier ,] reduction-identifier : list)
allocate([allocator :] list)
nowait
```

2.4.3 Οδηγία single

Η οδηγία single καθορίζει ότι το μπλοκ εκτελείται από ένα μόνο νήμα της ομάδας (όχι απαραίτητα το κύριο νήμα). Τα άλλα νήματα της ομάδας, τα οποία δεν εκτελούν το μπλοκ που βρίσκεται μέσα στην οδηγία σινγλε, περιμένουν σε ένα υπονοούμενο φράγμα στο τέλος της οδηγίας single, εκτός εάν έχει οριστεί η φράση nowait [1].

Συμβ. 8: Σύνταξη οδηγίας single

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

Συμβ. 9: Αποδεκτές φράσεις οδηγίας sections

```
private(list)
firstprivate(list)
copyprivate(list)
allocate([allocator :] list)
nowait
```

2.5 Φράσεις - Clauses

Δεδομένου οτι το OpenMP είναι ένα μοντέλο προγραμματισμού κοινής μνήμης, οι περισσότερες μεταβλητές στον κώδικα OpenMP είναι ορατές σε όλα τα νήματα από προεπιλογή. Οι ιδιωτικές μεταβλητές χρησιμοποιούνται για να αποφευχθούν τα race conditions και υπάρχει ανάγκη μεταβίβασης τιμών μεταξύ σειριακού κώδικα και παράλληλης περιοχής. Η διαχείριση αυτή των δεδομένων επιτυγχάνεται με τις φράσεις *(clauses)*. Οι βασικότερες ομάδες διαχωρισμού φράσεων αναφέρονται στα επόμενα κεφάλαια.

2.5.1 Φράσεις διαμοιρασμού δεδομένων - Data sharing attribute clauses

Οι φράσεις διαμοιρασμού δεδομένων χρησιμοποιούνται σε οδηγίες για να δίνουν στο χρήστη τη δυνατότητα έλεγχου των δεδομένων που χρησιμοποιούνται στην οδηγία.

2.5.1.1 Φράση shared

Τα δεδομένα που δηλώνονται εκτός μιας παράλληλης περιοχής, ειναι διαμοιραζόμενες σε όλα τα νήματα. Αν η μεταβλητή τροποποιηθεί από ένα νήμα, η αλλαγή θα είναι ορατή από όλα τα νήματα. Οι μεταβλητές με αυτό το χαρακτηριστικό διατηρούν την τιμή τους και μετά την έξοδο από το παράλληλο μπλοκ.

2.5.1.2 Φράση private

Τα δεδομένα που δηλώνονται σε μια παράλληλη περιοχή είναι ιδιωτικά για κάθε νήμα. Ετσι, κάθε νήμα θα έχει ένα τοπικό αντίγραφο και θα το χρησιμοποιεί ως προσωρινή μεταβλητή. Μια ιδιωτική μεταβλητή δεν έχει αρχικοποιηθεί και η τιμή δεν διατηρείται για χρήση εκτός της παράλληλης περιοχής.

2.5.1.3 Φράση default

Επιτρέπει στο χρήστη να δηλώσει ότι η προεπιλογή για τα δεδομένα σε μια παράλληλη περιοχή θα είναι ή κοινόχρηστα ή *none* για C / C++ ή firstprivate. Η επιλογή νονε αναγκάζει τον χρήστη να δηλώνει κάθε μεταβλητή στην παράλληλη περιοχή χρησιμοποιώντας φράσης διαμοιρασμού μνήμης.

2.5.1.4 Φράση firstprivate

Η μόναδική διαφορά της από τη φράση private είναι οτι σε αντίθεση με τη private, η μεταβλητή αρχικοποιείται χρησιμοποιώντας την τιμή της μεταβλητής με το ίδιο όνομα που έχει και η μεταβλητή του κύριου νήματος.

2.5.1.5 Φράση lastprivate

Η μοναδική διαφορά της από τη φράση private είναι ότι σε αντιθεση με τη πριατε, η αρχική τιμή ανανεώνεται μετά το ςονστρυςτ. Μια μεταβλητή μπορεί να είναι και φιρστπριατε και λαστπριατε.

2.5.1.6 Φράση threadprivate

Η φράση threadprivate καθορίζει ότι καθολικά αντικείμενα (ή μεταβλητές) μπορούν να γίνουν προσωρινά ιδιωτικά για κάποιο νήμα. Με αυτό τον τρόπο, μπορούμε να ορίσουμε καθολικά αντικείμενα, αλλά να μετατρέψουμε την εμβέλειά τους και να τα κάνουμε τοπικά για κάποιο νήμα. Οι μεταβλητές για τις οποίες ισχύει η φράση threadprivate συνεχίζουν να είναι ιδιωτικές, για κάθε νήμα, ακόμα και σε διαφορετικές παράλληλες περιοχές[12].

2.5.1.7 Φράση reduction

Συμβ. 10: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

```
reduction(operator | intrinsic : list):
```

Η φράση αυτή εκτελεί μία πράξη αναγωγής σε κάποιες μοιραζόμενες μεταβλητές . Όλες οι μεταβλητές που βρίσκονται σε μία παράλληλη περιοχή και υπάρχουν στη λίστα της φράσης ρεδυςτιον, αντιγράφονται σε τοπικά αντίγραφα, ένα για κάθε νήμα. Με την ολοκλήρωση των επαναλήψεων, εφαρμόζεται η πράξη που ορίζεται στο πεδίο *operator* και το τελικό αποτέλεσμα αποθηκεύεται στην αρχική θέση τους[12].

2.5.2 Φράσεις συγχρονισμού - Data sharing attribute clauses

Χρησιμοποιούνται για να αποφευχθούν προβλήματα *race condition* σε κοινόχρηστα δεδομένα.

2.5.2.1 Φράση critical

Το μπλοκ κώδικα που περικλείεται, εκτελείται μόνο από ένα νήμα κάθε φορά και όχι ταυτόχρονα από πολλά νήματα. Χρησιμοποιείται συχνά για την προστασία κοινόχρηστων δεδομένων από *race conditions*.

2.5.2.2 Φράση atomic

Η ενημέρωση μνήμης (εγγραφή ή ανάγνωση-τροποποίηση-εγγραφή) στην επόμενη οδηγία θα εκτελεστεί ατομικά. Δεν καθιστά ολόκληρη στην έκφραση *atomic* αλλα μόνο την ενημέρωση μνήμης. Ένας μεταγλωττιστής μπορεί να χρησιμοποιεί ειδικές οδηγίες hardware για καλύτερη επίδοση από ό,τι όταν χρησιμοποιεί το critical clause.

2.5.2.3 Φράση ordered

Το μπλοκ εκτελείται με τη σειρά με την οποία οι επαναλήψεις θα εκτελούνταν σε σειριακό βρόχο.

2.5.2.4 Φράση barrier

Κάθε νήμα περιμένει έως ότου όλα τα άλλα νήματα μιας ομάδας φτάσουν σε αυτό το σημείο. Η οδηγία διαμοιρασμού εργασιών έχει έναν σιωπηρό συγχρονισμό με *barrier* στο τέλος της.

2.5.2.5 Φράση nowait

Καθορίζει ότι τα νήματα που ολοκληρώνουν την εργασία τους μπορούν να προχωρήσουν χωρίς να περιμένουν να τελειώσουν όλα τα νήματα της ομάδας. Ελλείψει αυτής της φράσης, τα νήματα συγχρονίζονται με *barrier* στο τέλος της οδηγίας διαμοιρασμού εργασιών.

2.5.3 Φράσεις scheduling

Χρησιμοποιείται στην οδηγία διαμοιρασμού εργασίας βρόγχου. Οι επαναλήψεις της οδηγίας ανατίθενται σε κάθε νήμα σε ένα σύμφωνα με τον *τύπο* που ορίζεται στην φράση αυτή.

Συμβ. 11: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

```
schedule(type, chunk):
```

• Οι τρεις τύποι προγραμματισμού είναι:

1. **static**:

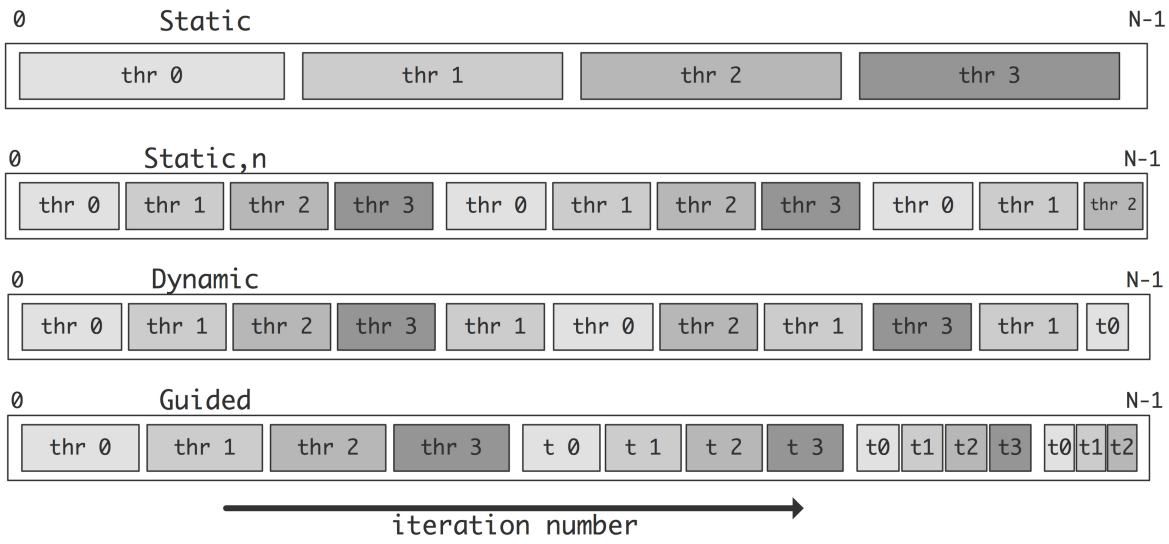
Οι επαναλήψεις κατανέμονται σε κάθε νήμα πριν την εκτέλεση της λουπας, και χωρίζονται ισόποσα σε όλα τα τηρεαδς. Παρόλα αυτά ο καθορισμός ενός ακεραίου αναθέτει ένα συγκεκριμένο αριθμό επαναλήψεων για ένα συγκεκριμένο νήμα.

2. **dynamic**:

Ορισμένες από τις επαναλήψεις κατανέμονται σε μικρό αριθμό νημάτων. Μόλις ένα συγκεκριμένο νήμα ολοκληρώσει την εκχωρημένη επανάληψή του, επιστρέφει για να πάρει μία από τις επαναλήψεις που απομένουν. Το ακέραιο όρισμα καθορίζει τον αριθμό των συνεχόμενων επαναλήψεων που εκχωρούνται σε ένα νήμα κάθε φορά.

3. **guided**:

Ενας μεγάλος αριθμός από συνεχείς επαναλήψεις ανατίθεται σε κάθε νήμα δυναμικά. Το μέγεθος των επαναλήψεων μειώνεται εκθετικά με κάθε διαδοχική κατανομή σε ένα ελάχιστο μέγεθος που καθορίζεται στο κομμάτι της παραμέτρου.



Σχήμα 6: Τύποι φράσης Schedule

2.5.4 Αλλες φράσεις

2.5.5 Φράσεις flush

Η τιμή αυτής της μεταβλητής αποθηκεύεται απο την ρεγιστερ στη μνήμη, για χρήση της εκτός της παράλληλης περιοχής.

2.5.6 Φράσεις master

Εκτελείται μόνο από το κύριο νήμα. Δεν υπάρχει υπονοούμενο βαρριερ. Τα υπόλοιπα νήματα θα συνεχίσουν τους υπολογισμούς.

ΤΟΔΟ μπορο να αλο ενα κεφαλαιο για ΕΝΎΙΡΟΝΜΕΝΤΑΛ ΎΑΛΥΕΣ

ΤΟΔΟ Μπορώ να βάλω ενα κεφάλειο για τα ρυντιμε φυνστιονς.

3 Χαρακτηριστικά εκδόσεων OpenMP 3.0 - 4.5

Το OpenMP 3.0 ήταν η πρώτη ενημέρωση μετά την έκδοση 2.5 με σημαντικές προσθήκες και αλλαγές. Χαρακτηριστικά προηγούμενων εκδόσεων αναβαθμίστηκαν αλλά τη σημαντικότερη αλλαγή αποτέλεσε η εισαγωγή στην έννοια των διεργασιών (*Tasking*). Η επόμενη έκδοση του OpenMP (4.0) κυκλοφόρησε τον Ιούλιο του 2013 και περιελάμβανε τα παρακάτω χαρακτηριστικά:

- threads affinity,
- ετερογενείς προγραμματισμός,
- διαχείριση σφαλμάτων,
- διανυσματικοποίηση μέσω *SIMD*,
- *user-defined recuctions (UDRs)*

Το 2013 με την έκδοση 4.0, οι προδιαγραφές του ΟπενΜΠ έκαναν σημαντικές αλλαγές στην υποστήριξη για ετερογενών συστημάτων. Εισήχθει ένα σύνολο οδηγιών για τον προσδιορισμό συναρτήσεων και δεδομένων που θα μετακινηθούν σε συσκευή προορισμού για να υπολογιστούν. Εκτός όμως απο των υποστήριξη επιταχυντών, εισήχθει και η *SIMD* υποστήριξη για διανύσματα, ο *thread affinity* έλεγχος, τα *user-defined reductions (UDRs)*, η δημιουργία διεργασιών και φράσεις εξάρτησης. Η κυκλοφορία της έκδοσης 4.5 έγινε το 2015, και περιείχε την *taskloop* οδηγία που θα επέτρεπε τον διαχωρισμό των βρόγχων σε διεργασίες, αποτρέποντας όλα τα νήματα να εκτελέσουν τον βρόγχο.

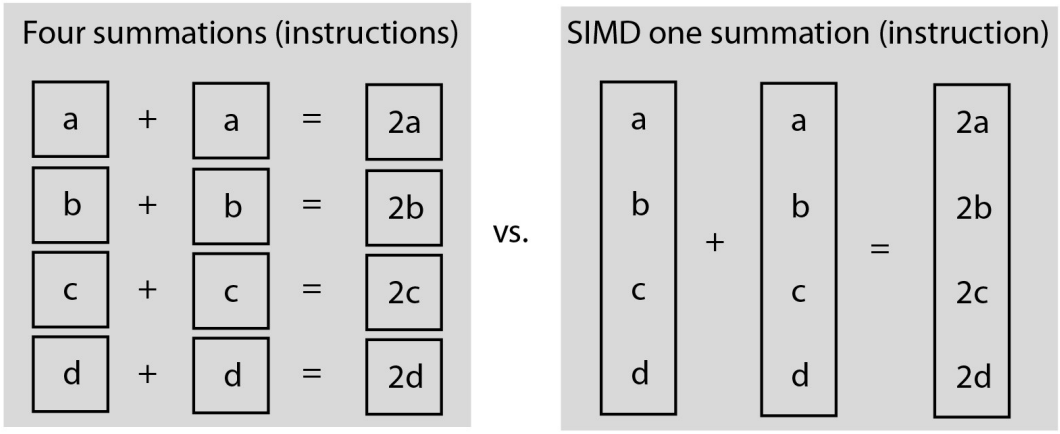
Τέλος, παρείχθει υποστήριξη για βρόχους *doacross* να παραλληλίσουν τους βρόχους με καλά δομημένες εξαρτήσεις και εισήχθη περαιτέρω υποστήριξη για διεργασίες με τη μορφή προτεραιότητας διεργασιών. Επεκτάσεις στο *SIMD* περιελάμβαναν τη δυνατότητα καθορισμού ακριβούς πλάτους *SIMD* και επιπλέον χαρακτηριστικών κοινής χρήσης δεδομένων[4].

3.1 SIMD

ΝΑ ΔΩ ΤΟ ΚΡΙΣΤΥ ΓΟΥΝΤΣ ΒΕΚΤΟΡΙΖΑΤΙΟΝ ! Σύμφωνα με τον *Flynn* [10], ένας επεξεργαστής *Single Instruction Multiple Data (SIMD)* δημιουργεί παραλληλισμό δεδομένων παρέχοντας οδηγίες που λειτουργούν συνολικά σε μπλοκ δεδομένων που ονομάζονται διανύσματα. Αυτό έρχεται σε αντίθεση με τις βαθμωτές οδηγίες που εκτελούνται σε μοναδιαία στοιχεία κάθε φορά.

Στο *OpenMP*, διανυσματικοποίηση αναφέρεται ως ο παραλληλισμός *SIMD*. Το *SIMD* παρέχει παραλληλισμό δεδομένων σε επίπεδο εντολών, δηλαδή μια οδηγία λειτουργεί ταυτόχρονα με πολλά στοιχεία δεδομένων. Οι *SIMD* οδηγίες χρησιμοποιούν ειδικούς καταχωρητές που περιέχουν πολλαπλά δεδομένα. Το μέγεθος αυτών των καταχωρητών ορίζει το μήκος του διανύσματος που είναι ο αριθμός των βαθμωτών δεδομένων που μπορούν να υποστούν επεξεργασία παράλληλα σε μια οδηγία *SIMD*.

Στο παρακάτω σχήμα παρουσιάζεται η προσθεση των στοιχείων δυο διανυσμάτων και η εκχώρηση των αποτελεσμάτων σε ένα τρίτο, με τον συμβατικό τρόπο, αλλά και με την χρήση *SIMD* μεθόδου. Το πλεονέκτημα είναι οτι οι *SIMD* οδηγίες εκτελούνται το ίδιο γρήγορα με τις αντίστοιχες βαθμωτές. Με άλλα λόγια η πράξη της παρακάτω εικόνας, θα γίνει 4 φορές που γρήγορα αν χρησιμοποιηθεί *SIMD* οδηγία.



Σχήμα 7: Πρόσθεση διανυσμάτων βαθμωτά και με SIMD

3.1.1 Η οδηγία *simd*

Η οδηγία *simd* ισχύει για έναν ή περισσότερους βρόγχους. Η δομή του βρόγχου στον οποίο εφαρμόζεται η οδηγία είναι ίδια με την δομή του βρόγχου της *C++*. Οι φράσεις που υποστηρίζονται είναι οι παρακάτω:

Συμβ. 12: Φράσεις που υποστηρίζονται από simd

```
private (list)
lastprivate (list)
reduction (reduction-identifier : list)
collapse (n)
simdlen (length)
safelen (length)
linear (list[: linear-step J)
aligned (list{:alignment])
```

Η προσθήκη της οδηγίας *simd* δίνει εντολή στο μεταγλωττιστή να δημιουργήσει έναν *simd* βρόγχο. Η χρήση δεικτών μέσα στο βρόγχο μπορεί να προκαλέσει απροσδιοριστη συμπεριφορα υπο συνθήκες. Για παράδειγμα, αν ο δείκτης *k* ή *m* είναι ταυτόσημος με τον δείκτη t, τότε αναμένεται τα αποτελέσματα να είναι λάθος.

Συμβ. 13: Παράδειγμα κώδικα με simd

```
void accumulate(int *t, int *k, int *m, int n) {
        #pragma omp simd
        for (int i = 0; i < n; ++i) {
                t[i] = k[i] + m[i];
        }
}
```

Στο παραπάνω παράδειγμα, η μεταβλητή *i* είναι ιδιωτική. Η διαφορά με την ιδιωτική μεταβλητή ενός παράλληλου βρόγχου είναι οτι η ιδιωτικότητα αναφέρεται σε ένα *SIMD lane*. Οι τιμές των διανυσμάτων *t, k, m* είναι κοινόχρηστες. Ο μεταγλωττιστής επιλέγει το κατάλληλο μήκος του διανύσματος για τη συγκεκριμένη αρχιτεκτονική. Οι φράσεις *private, lastprivate, reduction, collapse, ordered*, έχουν την ίδια χρησιμότητα που προαναφέρθηκε στις παραπάνω οδηγίες (πχ οδηγία διαμοιρασμού βρόγχου).

3.1.1.1 Η φράση *simdlen*

Η φράση *simdlen* δέχεται ως όρισμα ένα θετικό ακέραιο αριθμό που καθορίζει τον προτιμώμενο αριθμό επαναλήψεων ενός βρόγχου που θα εκτελούνται ταυτόχρονα. Επηρεάζει το μήκος του διανύσματος που χρησιμοποιείται από τις παραγόμενες *simd* οδηγίες.

Η τιμή του ορίσματος είναι προτιμητέα αλλα όχι υποχρεωτική. Ο μεταγλωττιστής έχει την ελευθερία να αποκλίνει από αυτή την επιλογή και να επιλέξει διαφορετικό μήκος. Ελλείψη αυτής της φράσης ορίζεται μια προεπιλεγμένη τιμή που καθορίζεται από την υλοποίηση. Σκοπός της φράσης *simdlen* είναι να καθοδηγήσει τον μεταγλωττιστή. Χρησιμοποιείται από τον χρήστη όταν έχει καλή εικόνα των χαρακτηριστικών του βρόγχου και γνωρίζει ότι κάποιο συγκεκριμένο μήκος μπορεί να ωφελήσει στην απόδοση.

3.1.1.2 Η φράση *safelen*

Η φράση *safelen* δέχεται ως όρισμα ένα θετικό ακέραιο αριθμό. Η τιμή αυτή καθορίζει το ανώτερο όριο του μήκους διανύσματος. Είναι ο αριθμός που στο οποίο είναι ασφαλές για τον βρόγχο. Το τελικό μήκος διανύσματος επιλέγεται από τον μεταγλωττιστή, αλλά δεν υπερβαίνει την τιμή της φράσης *safelen*.

Στο παρακάτω παράδειγμα απαιτείται η φράση *safelen*. Πρόκειται για έναν βρόγχο που περιέχει δέσμευση στην προσπέλαση των στοιχείων του διανύσματος. Για παράδειγμα, το διάβασμα του *[i-10]* στην επανάληψη *i* δεν μπορεί να γίνει, μέχρι να ολοκληρωθεί ή εγγραφή στο *k[i]* της προηγούμενης επανάληψης.

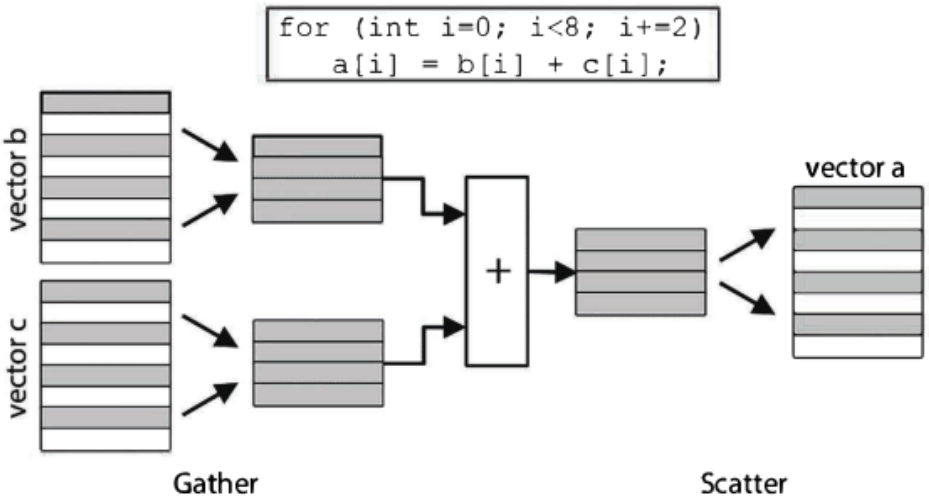
Συμβ. 14: Παράδειγμα κώδικα με simd

```
void dep_loop(float *k, float c, int n) {
        for (int i=10; i<n; i++) {
                k[i] = k[i-10] * c;
        }
}
```

3.1.1.3 Η φράση *linear*

Τα βαθμωτά στοιχεία εισέρχονται σε διανύσματα, στα οποία εκτελούνται οδηγίες *simd* και εξέρχονται. Οταν τα στοιχεία είναι προσβάσιμα με γραμμικό τρόπο, το διάνυσμα κατασκευάζεται εύκολα. Για παράδειγμα, δεδομένου ότι διατηρείται η ευθυγράμμιση των δεδομένων, μια απλή *simd* οδηγία φόρτωσης και αποθήκευσης, χρησιμοποιείται για να γράψει και να διαβάσει διαδοχικά δεδομένα στο διάνυσμα.

Σε περιπτώσεις που τα βαθμωτά παρατίθενται διαδοχικά στη μνήμη, η πρόσβαση στα στοιχεία γίνεται με *μοναδιαίο βήμα*. Αν τα δεδομένα δεν είναι διατεταγμένα σε διαδοχικές θέσεις μνήμης, αλλά με κάποια μετατόπιση μεταξύ τους, τότε η προσέγγιση του μπορεί να γίνει με μεγαλύτερο βήμα. Στην περίπτωση αυτή το βήμα είναι ίσο με τη μετατόπιση μεταξύ των στοιχείων.



Σχήμα 8: Σχηματική απεικόνιση φράσης *linear*

Για την κατασκευή του διανύσματος, μια λειτουργία συγκέντρωσης διαβάζει γραμμικά αλλά με βήμα μεγαλύτερο του ενός. Ομοίως, μια λειτουργία γράφει τα δεδομένα του διανύσματος πίσω στη μνήμη γραμμικά, αλλα με βήμα μεγαλύτερο του ενος.

3.1.1.4 Η φράση *aligned*

Η ευθυγράμμιση των δεδομένων είναι σημαντική για την καλή απόδοση του προγράμματος. Εαν ένα στοιχεία δεν είναι ευθυγραμμισμένο σε διευθυνση μνήμης που είναι πολλαπλάσιο του μεγέθους του στοιχείου σε βψτε, προκύπτει ένα επιπλέον κόστος κατά την πρόσβαση στο στοιχείο αυτό.

Για παράδειγμα, σε ορισμένες αρχιτεκτονικές ενδέχεται να μην είναι δυνατή η φόρτωση και εγγραφή απο μια διεύθυνση μνήμης που δεν ειναι ευθυγραμμισμένη με το μέγεθος του δεδομένου που χρησιμοποιείται. Αν ισχύει κάτι τέτοιο, οι λειτουργίες γίνονται κανονικά, με μεγαλύτερο κόστος. Σε περίπτωση διανυσματοκοποίησης μέσω της χρήσης οδηγίας *simd*, αλλά η προκύπτουσα απόδοση είναι κακή, η προσαρμογή ευθυγράμμισης δεδομένων μπορεί να βελτιώσει την εκτέλεση.

Η φράση ευθυγράμμισης υποστηρίζεται τόσο από την οδηγία *simd* όσο και από την οδηγία *declare simd*. Η φράση δέχεται ως όρισμα μια λίστα μεταβλητών. Η ευθυγράμμιση πρέπει να είναι ενας σταθερός ακεραιος αριθμός. Σε περίπτωση έλλειψης της φράσης, μια προεπιλεγμένη τιμή καθορίζεται από την υλοποίηση.

3.1.1.5 Η σύνθετη οδηγία βρόγχου *SIMD*

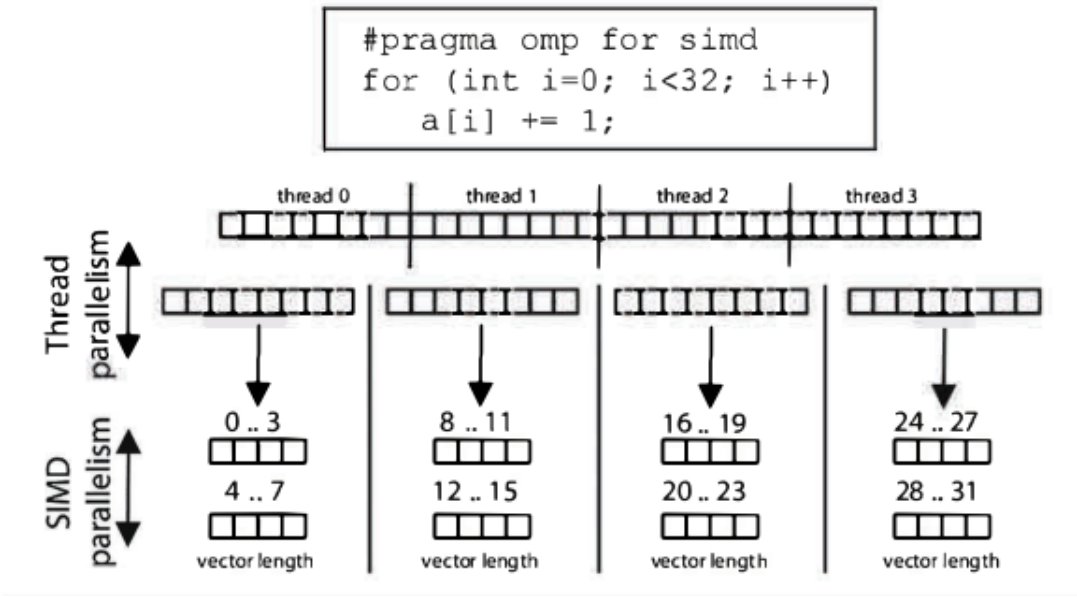
Συμβ. 15: Παράδειγμα κώδικα με simd

```
#pragma omp for simd [clause[[,] clause] ...] new-line
        for-loops
```

Η σύνθετη οδηγία βρόγχου *SIMD*, συνδυάζει παραλληλισμό νημάτων και *SIMD*. Κομμάτια επαναλήψεων βρόγχου διανέμονται στα νήματα σε μια ομάδα. Στη συνέχεια εκτελούνται τα κομμάτια των επαναλήψεων με βρόγχο *simd*. Μια φράση που μπορεί να εμφανιστεί

στην οδηγία διαμοιρασμού βρόγχου είτε στην οδηγία *simd* μπορεί να εμφανιστεί και στο σύνθετο όρο.

Στη σύνθετη οδηγία, τμήματα επαναλήψεων του βρόγχου κατανέμονται σε μια ομάδα νημάτων με τη μέθοδο που ορίζεται από τις φράσεις που ορίζονται και για την οδηγία διαμοιρασμού βρόγχου. Στη συνέχεια, τα κομμάτια επαναλήψεων βρόγχου μπορούν να μετατραπούν σε οδηγίες *simd* με μέθοδο που καθορίζεται από της φράσης που ορίζεται και για την οδηγία *simd*. Τα παραπάνω φαίνονται σχηματικά στην επόμενη εικόνα:



Σχήμα 9: Βήματα διεργασιών οδηγίας *for simd*

Κάθε βρόγχος έχει ένα συγκεκριμένο ποσοστό εργασίας που πρέπει να εκτελέσει. Αν ο αριθμός των νημάτων αυξηθεί, το ποσοστό της εργασίας ανα νήμα θα μειωθεί. Προσθέτοντας παραλληλισμό *simd*, δεν ειναι απραίτητη η βελτίωση της απόδοσης, ειδικά αν ο σιμδ βρόγχος που ανήκει σε ένα νήμα, μειώσει το μήκος του.

3.1.2 Συναρτήσεις *SIMD*

Οι κλίσεις συναρτήσεων εντός βρόγχου *simd* εμποδίζουν τη δημιουργία αποτελεσματικών *simd* δομών. Στη χειρότερη περίπτωση η κλήση της συνάρτησης θα γίνει χρησιμοποιώντας βαθμωτά δεδομένα, και πιθανότατα θα επηρεάσει αρνητικά την αποτελεσματικότητα.

Για την πλήρη εκμετάλλευση του παραλληλισμού ΣΙΜΔ, μια συνάρτηση που καλείται μέσα από ένα βρόγχο *simd*, πρέπει να έιναι σε μια ισοδύναμη έκδοση της συνάρτησης *simd*. Ετσι, ο μεταγλωτιστής πρέπει να δημιουργήσει αυτή την ειδική έκδοση της συνάρτησης αυτής με *simd* παραμέτρους και οδηγίες.

Η οδηγία *simd directive* χρησιμοποιείται για να δημιουργήσει ο μεταγλωττιστής μια ή περισσότερες *simd* εκδόσεις μιας συνάρτησης. Αυτές οι εκδόσεις χρησιμοποιούνται απο βρόγχους *simd*.

3.1.2.1 Η οδηγία *declare simd*

Η οδηγία *declare simd* χρησιμοποιείται για να δηλώσει οτι μια *simd* παραλλαγή μιας συνάρτησης μπορεί να χρησιμοποιηθεί από μια περιοχή παραλληλισμού *simd*.

Συμβ. 16: Χρήση οδηγίας declare simd

```
#pragma omp declare simd [clause [[,] clause] ...] new-line
 function declaration definitions
```

Συμβ. 17: Φράσεις που υποστηρίζονται από simd

```
simdlen (length)
linear (list[: linear-step J)
aligned (list[:alignmentj)
uniform ( argument-list)
inbranch
notinbranch
```

Ο μεταγλωττιστής μπορεί να δημιουργήσει πολλές εκδόσεις μιας συνάρτησης *simd* και να επιλέξει την κατάλληλη για να κληθεί σε μια συγκεκριμένη τοποθεσία μιας κα-

τασκευής *simd*. Ο χρήσης μπορεί να προσαρμόσει την λειτουργία μιας συνάρτησης με εξειδικευμένες φράσεις.

Υπάρχουν δύο περιορισμοί. Αν μια μεταβλητή αλλάζει ως αποτέλεσμα μιας τροποποίησης μιας φαινομενικά άσχετης μεταβλητής η συμπεριφορά είναι απροσδιόριστη. Επιπλέον, μια συνάρτηση που εμφανίζεται κάτω απο οδηγία *declare simd*, δεν επιτρέπονται τα *exceptions*.

Μια παρενέργεια στο πλαίσιο του προγραμματισμού λέγεται ότι συμβαίνει εάν μια μεταβλητή αλλάζει ως αποτέλεσμα μιας τροποποίησης σε μια φαινομενικά άσχετη μεταβλητή. Αυτό μπορεί να συμβεί λόγω του ψευδώνυμου του δείκτη για παράδειγμα. Σε μια τέτοια κατάσταση, δύο διαφορετικοί δείκτες δείχνουν στην ίδια θέση μνήμης. Αλλαγή ενός δείκτη επηρεάζει τους άλλους. Ένα άλλο παράδειγμα είναι μια συνάρτηση που τροποποιεί τα παγκόσμια δεδομένα. Δεδομένου ότι τα παγκόσμια δεδομένα χρησιμοποιούνται (δυνητικά) από άλλες λειτουργίες, αυτό μπορεί να προκαλέσει καταστροφή σε ένα παράλληλο πρόγραμμα και πρέπει να ληφθεί μέριμνα για την αντιμετώπιση αυτής της κατάστασης.

3.1.2.2 Χαρακτηριστικά παραμέτρων συνάρτησης *simd*

Οι φράσεις *uniform, linear, simdlen, aligned* χρησιμοποιούνται για τον καθορισμό χαρακτηριστικών για τις παραμέτρους της συνάρτησης *simd*. Εκτός από τη ρήτρα *simdlen*, οι μεταβλητές που εμφανίζονται στις υπόλοιπες φράσεις πρέπει να είναι παράμετροι της συνάρτησης για την οποία ισχύει η οδηγία.

Οταν μια παράμετρος βρίσκεται στη φράση *uniform*, υποδεικνύει οτι η τιμή της παραμέτρου έχει την ίδια τιμή για όλες τις ταυτόχρονες κλήσεις κατά την εκτέλεση μιας οδηγίας *simd* βρόχου. Η φράση *linear* έχει διαφορετική σημασία όταν εμφανίζεται σε οδηγία *simd*. Δείχνει οτι ένα όρισμα που μεταβιβάζεται σε μια συνάρτηση έχει γραμμική σχέση μεταξύ των παράλληλων επικλήσεων μιας συνάρτησης.

Κάθε *simd* λωρίδα παρατηρεί την τιμή του ορίσματος στην πρώτη λωρίδα και προσθέτει την μετατόπιση της *simd* λωρίδας από την πρώτη, επί το γραμμική βήμα.

$$val_{curr} = val_1 + offset * step$$

Η φράση *uniform(arg1, arg2)* ενημερώνει τον *compiler* να δημιουργήσει μια *simd* συνάρτηση που προυποθέτει οτι αυτές οι δύο μεταβλητές είναι ανεξάρτητες από τον βρόγχο.

Η φράση *inbranch* υποστηρίζει οτι μια συνάρτηση καλείται πάντα μέσα σε ένα βρόγχο *simd* που περιέχει *if condition*. Ο μεταγλωττιστής πρέπει να αναδιαρθρώσει τον κώδικα για να χειριστεί την πιθανότητα ότι μια λωρίδα *simd* μπορεί να μην εκτελέσει τον κώδικα μιας συνάρτησης.

Η φράση *notinbranch* χρησιμοποιείται όταν η συνάρτηση δεν εκτελείται ποτέ μέσα από ένα *if condition* σε ένα *simd* βρόγχο. Επιτρέπει τον μεταγλωττιστή κάνει μεγαλύτερες βελτιστοποιήσεις στην απόδοση του κώδικα μιας συνάρτησης που χρησιμοποιεί *simd* οδηγίες.

Συμβ. 18: Παράδειγμα χρήσης φράσεων inbranch - notinbranch

```
#pragma omp declare simd inbranch
float pow(float x) {
        return (x * x);
}

#pragma omp declare simd notinbranch
extern float div(float);

void simd_loop(float *a, float *b, int n)
{
        #pragma omp simd
        for (int i=0; i<n; i++) {
                if (a[i] < 0.0 )
                        b[i] = pow(a[i]);
                b[i] = div(b[i]);
        }
}
```

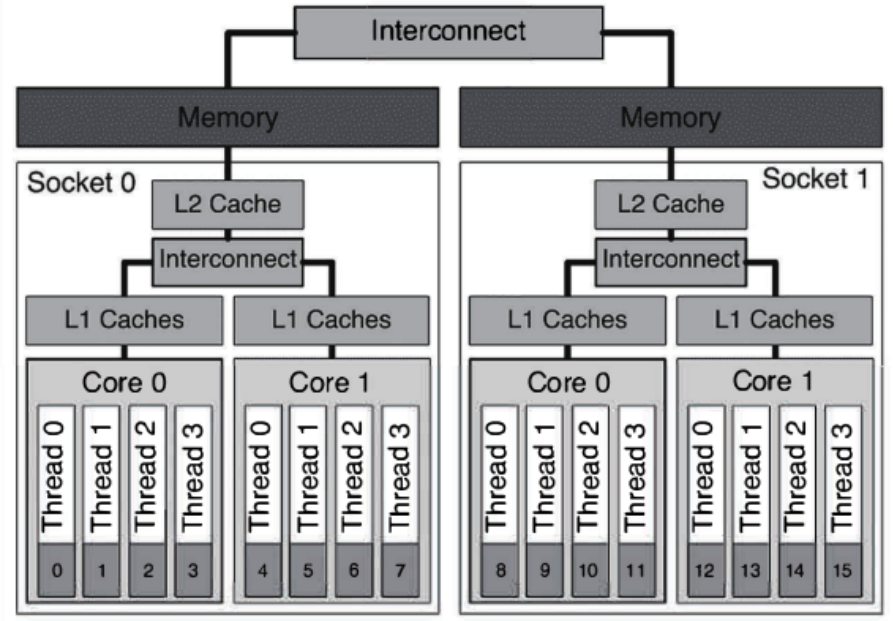
3.2 Thread Affinity

Thread Affinity είναι μια ευρύτερη έννοια που περιλαμβάνει την βελτιστοποίηση του χρόνου εκτέλεσης ενός προγράμματος, μέσω βελτιστοποιήσεων στο εύρος ζώνης μνήμης, τη ν αποφυγή καθυστέρησης μνήμης ή της καθυστέρησης χρήσης προσωρινής μνήμης.

Το *OpenMP 4.0* εισάγει ένα σύνολο οδηγιών για την υποστήριξη του *thread affinity*[3]. Η πλειοψηφία πλέον των μηχανημάτων βασίζονται στην *cc-NUMA* αρχιτεκτονική. Ο λόγος που αυτό το σύστημα μνήμης έγινε κυρίαρχο, είναι η συνεχής αύξηση του αριθμού των επεξεργαστών. Η μονολιθική διασύνδεση μνήμης με σταθερό εύρος ζώνης μνήμης θα αποτελούσε πρόβλημα στην ραγδαία αύξηση των επεξεργαστών.

Στη *cc-NUMA* αρχιτεκτονική κάθε υποδοχή συνδέεται με ένα υποσύνολο της συνολικής μνήμης του συστήματος. Μία διασύνδεση ενώνει τα υποσύνολα μεταξύ τους και δημιουργεί την εικόνα ενιαιας μνήμης στον χρήστη. Ενα τέτοιο σύστημα είναι ευκολότερο να επεκταθεί.

Το πλεονέκτημα της διασύνδεσης είναι ότι η εφαρμογή έχει πρόσβαση σε όλη την μνήμη του συστήματος, ανεξάρτητα από το που βρίσκονται τα δεδομένα. Ωστόσο, πλεον ο χρόνος πρόσβασης σε αυτά δεν ειναι ο σταθερός καθώς εξαρτάται από τη θέση τους στη μνήμη.



Σχήμα 10: Αρχιτεκτονική cc-NUMA[18]

3.3 Tasking

Η έννοια των διεργασιών (*Tasking*) εισήχθει στο OpenMP το 2008 με την έκδοση 3.0[9]. Οι διεργασίες παρέχουν τη δυνατότητα, οι αλγόριθμοι με ακανόνιστη και εξαρτώμενη από το χρόνο ροή εκτέλεσης να μπορούν να παραλληλιστούν. Ένας μηχανισμός ουράς διαχειρίζεται δυναμικά την εκχώρηση νημάτων στη διεργασία που πρέπει να εκτελεστεί. Τα νήματα παραλαμβάνουν εργασίες από την ουρά εως ότου αυτή αδειάσει. Διεργασία (*task*) είναι ενα μπλοκ κώδικα σε μια παράλληλη περιοχή που εκτελειται ταυτόχρονα με μια αλλη διεργασία στην ίδια περιοχή.

Οι διεργασίες είναι τμήμα της παράλληλης περιοχής. Χωρίς ειδική μέριμνα, η ίδια διεργασία μπορεί να εκτελεστεί από διαφορετικά νήματα. Αυτό αποτελεί ασάφεια, καθώς οι οδηγίες διαμοιρασμού μνήμης, καθορίζουν με σαφήνεια τον τρόπο που οι εργασίες διανέμονται στα νήματα. Για να εγγυηθεί το OpenMP ότι κάθε διεργασία εκτελείται μόνο μια φορά, η κατασκευή τους θα πρέπει να ενσωματωθούν σε μια οδηγία *single* ή *master*.

Συμβ. 19: Παράδειγμα κώδικα με διεργασίες

```
#include <omp.h>

int main(void) {
    #pragma omp parallel          // Begin of parallel region
    {
        #pragma omp single
        {
            #pragma omp task
                func_task_1();    // First task creation
            #pragma omp task
                func_task_2();    // Second task creation
        }// end of single region  // Implicit barrier
    } //end of parallel region    // Implicit barrier
}
```

Το παραπάνω παράδειγμα αποτελεί επεξήγηση της λειτουργίας των διεργασιών. Υπάρχουν δύο νήματα, ένα νήμα συναντά την οδηγία *single* και αρχίζει να εκτελεί το μπλοκ κώδικα. Δύο διεργασίες δημιουργούνται, αλλά δεν έχουν ακόμη εκτελεστεί. Το άλλο νήμα συναντά και περιμένει στο *barrier* στο τέλος της οδηγίας *single*. Στην περίπτωση την διεργασιών, τα αδρανή νήματα δεν περιμένουν στο *barrier*. Αντι αυτου, αυτά τα νήματα είναι διαθέσιμα για την εκτέλεση των διεργασιών. Το νήμα που δημιουργεί τις διεργασίες καταλήγει επίσης στο φράγμα, μόλις ολοκληρωθεί η φάση παραγωγής και μπορεί να εκτελεί και αυτό διεργασίες. Η σειρά με την οποία θα εκτελεστούν οι διεργασίες δεν καθορίζεται.

3.3.1 Η οδηγία διεργασιών

Ορίζει μια ρητή διεργασία. Το περιβάλλον δεδομένων της διεργασίας δημιουργείται σύμφωνα με τις φράσεις χαρακτηριστικών κοινής χρήσης δεδομένων κατασκευή διεργασίων και τυχόν προεπιλογές που ισχύουν[17].

Συμβ. 20: Σύνταξη διεργασίας

```
#pragma omp task [clause[ [, ]clause] ...]
structured-block
```

Συμβ. 21: Αποδεκτές φράσεις οδηγίας sections

```
if([ task :] scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
in_reduction(reduction-identifier : list)
depend([depend-modifier,] dependence-type : locator-list)
priority(priority-value)
```

```
allocate([allocator :] list)
affinity([aff-modifier :] locator-list)
detach(event-handle)
```

3.3.1.1 Φράση if

Η έκφραση της συνθήκης *if* μιας διεργασίας, θα πρέπει να αξιολογείται ώς ψευδής ή αληθής. Η οδηγία των διεργασιών παρέχει μια φράση *if*, που δέχεται μια έκραση ως όρισμα. Αν η έκφραση αξιολογείται ώς ψευδής, απαγορεύει η αναβολή της διεργασίας. Ανεξαρήτου αποτελέσματος της έκφρασης, δημιουργείται πάντα ένα νέο περιβάλλον δεδομένων εργίας. Αν η έκφραση αξιολογείται ως ψευδής, δεν γίνονται όλοι οι υπολογισμοί που απετούνται για μια διεργασία αν δεν υπήρχε η φράση ιφ, επομένως θα μπορούσαν να αποφευχθούν ορισμένα κόστη επιδόσεων.

Ως εκ τούτου η φράση *if* αποτελεί λύση σε καταστάσεις, όπως οι αναδρομικοί αλγόριθμοι, όπου το υπολογιστικό κόστος μειώνεται καθώς αυξάνεται το βάθος και το όφελος από τη δημιουργία μιας νέας διεργασίας μειώνεται λόγω του γενικού κόστους. Ωστόσο, για λόγους ασφαλείας [9], οι μεταβλητές που αναφέρονται σε μια οδηγία κατασκευής διεργασιών είναι, στις περισσότερες περιπτώσεις, από προεπιλογή *firstprivate*, επομένως το κόστος της ρύθμισης περιβάλλοντος δεδομένων μπορεί να είναι το κυρίαρχο συστατικό της δημιουργίας διεργασιών.

3.3.1.2 Φράση final

Η φράση χρησιμοποιείται για να ελέγχεται με ευκρίνεια η ευαισθησία των διεργασιών. Οταν χρησιμοποιείται μέσα στην οδηγεία *task*, η έκφραση αξιολογείται κατά τη διάρκεια δημιουργίας αυτού. Αν είναι αληθής, η διεργασία θεωρείται τελική. Όλες οι διεργασίες που δημιουργούνται μέσα σε αυτή τη διεργασία, αγνοούνται και εκτελούνται στο πλαίσιο της.

Υπάρχουν δυο διαφορές ανάμεσα στη φράση *if* και στη *final*: την κατασκευή διεργασιών που επηρεάζει και τον τρόπο με τον οποίο οι κατασκευές αγνοούνται[2].

Πίνακας 1: Διαφορές ανάμεσα στις φράσεις *if* και *final* οταν εισάγονται σε κατασκευή διεργασίας.

***if* clause**	***final* clause**
Επηρεάζει την διεργασία που κατασκευάζεται από τη συγκεκριμένη οδηγιά κατασκευής διεργασιών	Επηρεάζει όλες τις διεργασίες ¨απογόνους¨ δηλαδή όλες τις διεργασίες που πρόκειται να δημιουργηθούν μέσα από αυτή που περιείχε τη φράση *final*
Η οδηγία κατασκευής διεργασίας αγνοείται εν μέρει. Η διεργασία και το περιβάλλον μεταβλητών δημιουργείται ακόμα και αν η έκφραση αξιολογηθεί ως *false*	Αγνοείται εντελώς η κατασκευή διεργασιών δηλαδή δε θα δημιουργηθεί νέα εργασία ούτε νέο περιβάλλον δεδομένων.

3.3.1.3 Φράση mergeable

Μια συγχωνευμένη διεργασία είναι η διεργασία της οποίας το περιβάλλον δεδομένων είναι ίδιο με το περιβάλλον που δημιούργησε την διεργασία. Οταν εισάγεται η φράση *mergeable* σε μια οδηγία δημιουργίας διεργασίας τότε η υλοποίηση μπορεί να δημιουργήσει μια συγχωνευμένη διεργασία.

3.3.1.4 Φράση depend

Η φράση *depend* επιβάλλει πρόσθετους περιορισμούς στη δημιουργία διεργασιών. Αυτοί οι περιορισμοί δημιουργούν εξαρτήσεις μόνο μεταξύ συγκενικών διεργασιών.

3.3.1.5 Φράση untied

Κατά την επανάληψη μιας περιοχής διεργασίας που έχει τεθεί σε αναστολή, μια δεσμευμένη διεργασία θα πρέπει να εκτελεστεί ξανά από το ίδιο νήμα. Με τη φράση *untied*, δεν υπάρχει τέτοιος περιορισμός και η διεργασία συνεχίζεται από οποιοδήποτε νήμα.

3.3.2 Συγχρονισμός διεργασιών

Οι διεργασίες δημιουργούνται όταν υπάρχει μια οδηγία δημιουργίας διεργασιών. Η στιγμή εκτέλεσης τους δεν είναι καθορισμένη. Το *OpenMP* εγγυάται οτι θα έχουν ολοκληρωθεί όταν ολοκληρωθεί η εκτέλεση του προγράμματος, ή όταν προκύψει κάποια οδηγία συγχρονισμού διεργασίας.

Πίνακας 2: Οδηγιες συγχρονισμού διεργασιών.

Οδηγία	**Περιγραφή**
#pragma omp barrier	Μπορεί είτε να υπάρχει υπονοούμενη είτε να δηλωθεί ρητά.
#pragma omp taskwait	Περιμένει μέχρι να ολοκληρωθούν όλες οι διεργασίες παιδιά της συγκεκριμένης διεργασίας.
#pragma omp taskgroup	Περιμένει μέχρι να ολοκληρωθούν όλες οι διεργασίες παιδιά της συγκεκριμένης διεργασίας αλλά και οι απόγονοι τους.

Στην περίπτωση της δημιουργίας διεργασιών μέσω οδηγίας σινγλε, υπάρχει υπονοούμενο φράγμα εκτέλεσης, σε αντίθεση με την οδηγία μαστερ που δεν υπονοείται.

Συμβ. 22: Παράδειγμα taskwait

```
#pragma omp parallel {
        #pragma omp single
        {
                #pragma omp task
                {
                        function1();
                }//Task #1
                #pragma omp task
                {
                        function2();
                } // Task #2

                #pragma omp taskwait
                        last_to_be_executed();
        } // End of single region
} // End of parallel region
```

3.4 Ετερογενής Αρχιτεκτονική

Η ετερογενής αρχιτεκτονική είναι ένα σύνολο προδιαγραφών που επιτρέπουν την ενσωμάτωση κεντρικών μονάδων επεξεργασίας και επεξεργαστών γραφικών στον ίδιο δίαυλο, με κοινόχρηστη μνήμη και διεργασίες[11].

Εξειδικευμένη επεξεργαστές επιταχυντών που βελτιώνουν δραματικά την απόδοση υπολογισμών, πολλαπλασιάζονται και οι επεξεργαστές γενικής χρήσης συνδέονται σε κάποιον επιταχυντή. Η αύξηση της δημοτικότητας αυτών των ετερογενών αρχιτεκτονικών σε όλους τους τύπους υπολογιστών είχε αξιοσημείωτο αντίκτυπο στην ανάπτυξη λογισμικού.

Για την εκμετάλλευση αυτών των συστημάτων, οι χρήστες πρέπει να κατασκευάζουν λογισμικό που εκτελεί διάφορες περιοχές κώδικα σε διαφορετικούς τύπους συσκευών. Το μεγαλύτερο κίνητρο αποτελεί η επιτάχυνση των υπολογισμών στους βρόγχους επανάληψης.

Ωστόσο, τα μοντέλα προγραμματισμού για αυτά τα συστήματα είναι δύσκολο να χρησιμοποιηθούν. Συχνά, οι ενότητες κώδικα γράφονται δύο φορές, μία φορά για τον επεξεργαστή γενικού σκοπού και μια για τον επιταχυντή. Η έκδοση του επιταχυντή γράφεται γλώσσα χαμηλότερου επιπέδου. Τα παραπάνω προβλήματα οδηγούν σε δυσκολία συντήρησης λογισμικού και ανάγκη διατήρησης δύο εκδόσεων ίδιου κώδικα.

Για τις ανάγκες διευκόλυνσης, το *OpenMP* επέκτεινε τις λειτουργίες του με σκοπό να υποστηρίξει αυτούς τους τύπους συστημάτων[6]. Τα αποτελέσματα της εργασίας αρχικά δημοσιεύτηκαν στην έκδοση 4.0 και στη ανανεώθηκαν στην έκδοση 4.5.

Ο χρήστης μπορεί να χρησιμοποιήσει την διεπαφή *OpenMP*, για να προγραμματίσει επιταχυντές σε υψηλότερο επίπεδο γλώσσας και για τη διατήρηση μίας μόνο έκδοσης του κώδικα, η οποία μπορεί να τρέξει είτε σε επιταχυντή είτε σε επεξεργαστή γενικής χρήσης.

Κίνητρο για την εκτέλεση κώδικα σε μια ετερογενή αρχιτεκτονική είναι να εκτελεστεί ένα κομμάτι κώδικα σε έναν επιταχυντή. Ο ρόλος των επιταχυντών είναι η δραματική βελτίωση της απόδοσης ενός προγράμματος αξιοποιώντας τις δυνατότητες υλικού των συσκευών αυτών. Το *OpenMP* παρέχει τα μέσα για τη διανομή εκτέλεσης ενός προγράμματος σε διαφορετικές συσκευές με ετερογενή αρχιτεκτονική. Συσκευή είναι ένας υπολογιστικός πόρος όπου μια περιοχή κώδικα μπορεί να εκτελεστεί. Παραδείγματα

τέτοιων συσκευών είναι ΓΠΥ, ΄ΠΥ, ΔΣΠ, ΦΠΓΑ κ.α.

Οι συσκευές έχουν τα δικά τους νήματα, των οποίων η μετεγκατάσταση σε άλλες συσκευές δεν είναι δυνατή. Η εκτέλεση του προγράμματος ξεκινάει από την κεντρική συσκευή (*host device*. Η κεντρική συσκευή είναι υπεύθυνη για την μεταφορά του κώδικα και των δεδομένων στον επιταχυντή.

Συμβ. 23: Παράδειγμα taskwait

```
void add_arrays(double *A, double *B, double *C, size_t size) {
    size_t i = 0;
#pragma omp target map(A, B, C)
    for (i = 0; i < size; ++i) {
            C[i] = A[i] + B[i];
    }
}
```

Οταν ένα νήμα του εξυπηρετητή συναντάει την οδηγία *target*, η περιοχή που ακολουθεί εκτελείται σε έναν επιταχυντή. Απο προεπιλογή, το νήμα που συναντά την οδηγία περιμένει την ολοκλήρωση της εκτέλεσης της παράλληλης περιοχής, προτού συνεχίσει.

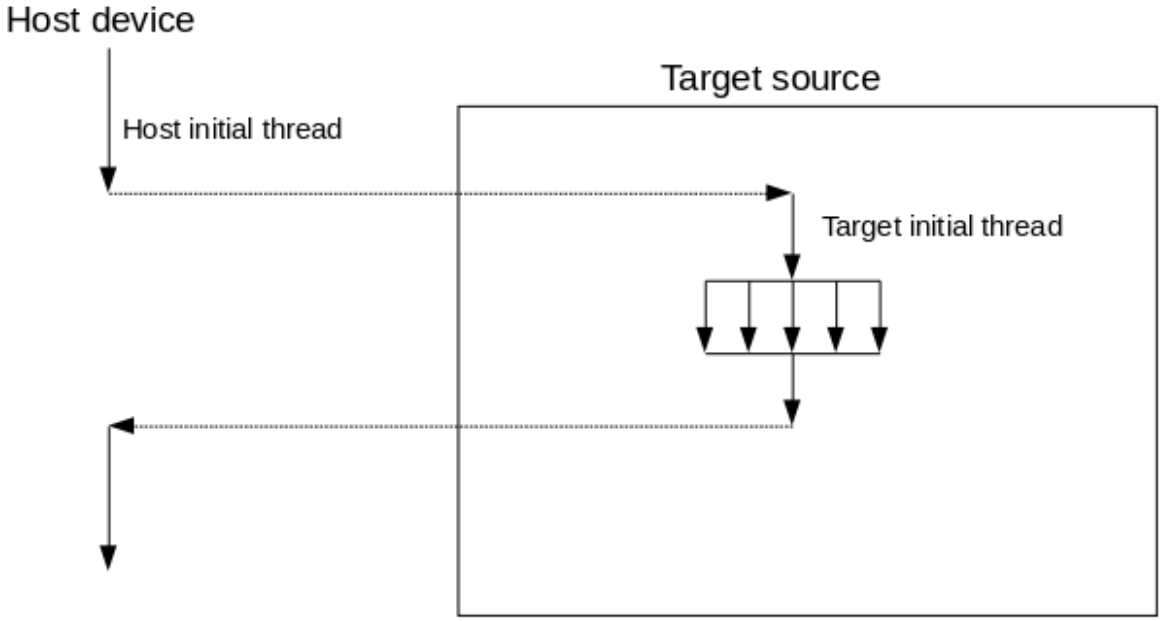
Πριν το νέο νήμα αρχίσει να εκτελεί την παράλληλη περιοχή, οι μεταβλητές *A, B, C* αντιστοιχίζονται στον επιταχυντή. Η φράση *mapped* είναι η έννοια που χρησιμοποιεί το *OpenMP* για να περιγράψει τον τρόπο κοινής χρήσης των μεταβλητών σε όλες τις συσκευές.

3.4.1 Το αρχικό νήμα της συσκευής προορισμού

Το νήμα που ξεκινάει την εκτέλεση ενός προγράμματος και όλου του σειριακού τμήματος του κώδικα, ονομάζεται κύριο νήμα. Στην περίπτωση του ετερογενούς προγραμματισμού, το κύριο νήμα ανήκει στην κεντρική συσκευή, δηλαδή το πρόγραμμα ξεκινάει να εκτελείται στην κεντρική συσκευή.

Με την εισαγωγή της οδηγίας *target* στο *OpenMP* 4.0, πολλαπλά αρχικά νήματα μπορούν να δημιουργηθούν κατά τη διάρκεια εκτέλεσης ενός προγράμματος. Την εκτέλεση του τμήματος κώδικα στη συσκευή προορισμού, την αναλαμβάνει ένα νέο αρχικό νήμα

και όχι το νήμα που συνάντησε το *target*. Το νήμα αυτό μπορεί να συναντήσει οδηγίες παραλληλισμού και να δημιουργήσει υποομάδες νημάτων.



Σχήμα 11: Διάγραμμα ομάδων νημάτων σε ετερογενή αρχιτεκτονική

3.4.2 Η οδηγία *target teams*

Η οδηγία *target teams* κατασκευάζει μια ομάδα (*league* που λειτουργούν σε έναν επιταχυντή. Κάθε μία από αυτές τις ομάδες είναι ένα αρχικό νήμα που εκτελεί παράλληλα την επόμενη δήλωση κώδικα. Η λειτουργία αυτή είναι παρόμοια με μια οδηγία *parallel* με τη διαφορά ότι τώρα κάθε νήμα είναι μια ομάδα. Τα νήματα σε διαφορετικές ομάδες δεν μπορούν να συγχρονιστούν μεταξύ τους.

Όταν μια παράλληλη περιοχή συναντάται από μια ομάδα, κάθε αρχικό τηρεαδ ομάδας γίνεται κύριο σε μια νέα υποομάδα. Το αποτέλεσμα είναι ένα σύνολο υποομάδων, οπου κάθε υποομάδα αποτελείται από ένα ή περισσότερο νήματα.

Αυτή η δομή χρησιμοποιείται για να εκφράζεται ένας τύπος χαλαρού παραλληλισμού, όπου ομάδες νημάτων εκτελούν παράλληλα, αλλά με μικρή αλληλεπίδραση μεταξύ των υποομάδων.

3.4.3 Μοντέλο μνήμης ετερογενούς αρχιτεκτονικής

3.4.3.1 Η φράση *map*

Τα νήματα που εκτελούνται σε έναν επιταχυντή μπορούν να έχουν ιδιωτικές μεταβλητές. Το κύριο νήμα που ξεκινά την εκτέλεση μιας παράλληλης περιοχής λαμβάνει μια ιδιωτική μεταβλήτη που εμφανίζεται στη φράση *private* ή *firstprivate* στην οδηγία *target*.

Η φράση *map* χρησιμοποιείται για τον διαμοιρασμό κοινής μνήμης από τον *host* στον επιταχυντή. Οταν οι δυο συσκευές δεν έχουν κοινόχρηστη μνήμη, η μεταβλητή αντιγράφεται στον επιταχυντή. Η φράση *en* αποκρύπτει αν η μεταβλητή μοιράζεται ή αντιγράφεται στη συσκευή στόχου. Το *OpenMP* ενεργεί ανάλογα με την αρχιτεκτονική που χρησιμοποιείται.

Πίνακας 3: Ενέργειες που απαιτούνται στο *map* ανάμεσα σε διαμοιραζόμενη και κοινόχρηστη μνήμη

Τύπος Μνήμης	**memory allocation**	**copy**	**flush**
Διαμοιρασμένη	Ναι	Ναι	Ναι
Κοινόχρηστη	Οχι	Οχι	Ναι

3.4.3.2 Περιβάλλον δεδομένων συσκευής

Ο επιταχυντής έχει ένα περιβάλλον μνήμης που περιέχει το σύνολο των μεταβλητών που είναι προσβάσιμες από νήματα που εκτελούνται σε αυτή τη συσκευή. Η αντιστοίχιση των δεδομένων διασφαλίζει ότι η μεταβλητή βρίσκεται στο περιβάλλον δεδομένων του επιταχυντή.

Μία μεταβλητής του εξυπηρετητή, αντιστοιχίζεται στην αντίστοιχη μεταβλητή του περιβάλλοντος δεδομένων του επιταχυντή. Ανάλογα με τη διαθεσιμότητα της κοινόχρηστης μνήμης μεταξύ του εξυπηρετητή *host* και της συσκευής προορισμού, η πρωτότυπη μεταβλητή του εξυπηρετητή και η αντίστοιχη μεγαβλητή της συσκευής προορισμού είναι

είτε η ίδια μεταβλητή που βρίσκεται στη κοινόχρηστη μνήμη ή βρίσκεται σε διαφορετικές θέσεις, με αποτέλεσμα να απαιτούνται εργασίες αντιγραφής και ενημέρωσης για να διατηρηθεί η συνέπεια μεταξύ των δυο θέσεων.

Η ελαχιστοποίηση της μεταφοράς δεδομένων ανάμεσα στον εξυπηρετητή και τον επιταχυντή, αποτελεί κρίσιμο σημείο για την επίτευξη καλύτερης επίδοσης στις ετερογενείς αρχιτεκτονικές. Η επαναληπτική αντιστοίχιση μεταβλητών που επαναχρησιμοποιούνται είναι αναποτελεσματική.

3.4.3.3 Δείκτες μεταβλητών συσκευής

Αν ο εξυπηρετητής και ο επιταχυντής δεν μοιράζονται τη κοινόχρηστη μνήμη, οι τοπικές μεταβλητές τους βρίσκονται σε διαφορετικές θέσεις μνήμης. Οταν μια μεταβλητή αντιστοιχίζεται στο περιβάλλον δεδομένων ενός επιταχυντή, γίνεται μια αντιγραφή και η καινούργια μεταβλήτή ειναι διαφορετική από την μεταβλητή του εξυπηρετητή.

Οι διευθύνσεις μνήμης αποθηκεύονται σε μεταβλητές που ονομάζονται δείκτες (*pointers*. Ένα νήμα του εξυπηρετητή δε μπορεί να έχει πρόσβαση σε μνήμη μέσω ενός δείκτη που περιέχει διεύθυνση μνήμης του επιταχυντή. Ακόμη, ο επιταχυντής και ο εξυπηρετητής μπορεί να έχουν διαφορετική αρχιτεκτονική, δηλαδή ένας τύπος μεταβλητής μπορεί να είναι διαφορετικού μεγέθους ανάμεσα στις δύο συσκευές.

Ο δείκτης συσκευής *(device pointer)* είναι ένας δείκτης που αποθηκεύεται στον εξυπηρετητή και περιέχει την διεύθυνση μνήμης στο περιβάλλον δεδομένων του επιταχυντή.

Συμβ. 24: Παράδειγμα taskwait

```
int device = omp_get_default_device();
char *device_ptr = omp_target_alloc(n, device);
#pragma omp target is_device_ptr (device_ptr)
for (int j=0; j<n ; j++)
        *device_ptr++ = 0;
```

Εχω θέμα το add vector doubles c10.

3.4.4 Η οδηγία target

Σκοπός της οδηγίας target είναι η μεταφορά και εκτέλεση ενός τμήματος κώδικα στον επιταχυντή.

Συμβ. 25: Παράδειγμα εκτέλεσης στον επιταχυντή

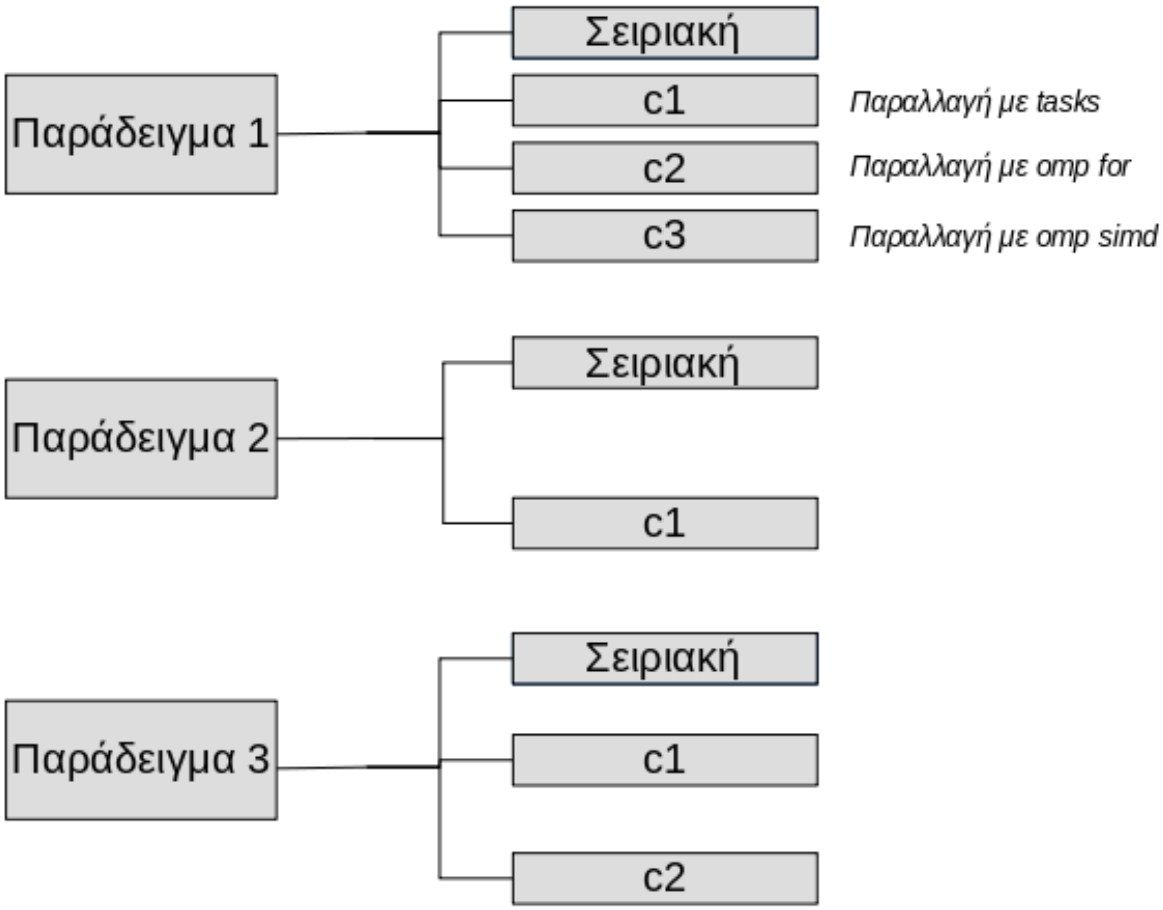
```
void test() {
        int flag = 0;
        #pragma omp target map(flag)
        {
                if (!omp_is_initial_device()) {
                        flag = 1;
                } else {
                        flag = 2;
                }
        }
        if (flag == 1) {
                printf("Running on accelerator\n");
        } else if (flag == 2) {
                printf("Running on host\n");
        }
}
```

4 Υλοποιημένα παραδείγματα

Οι ενότητες που ακολουθούν αφορούν την επίλυση προβλημάτων με διαφορετικές μεθόδους επίλυσης. Τα προβλήματα συγκεντρώθηκαν και επιλύθηκαν με στόχο την σύγκριση των αποτελεσμάτων και την εξαγωγή συμπερασμάτων.Ο πηγαίος των προβλημάτων βρίσκεται στο παρακάτω link

https : //github.com/gkonto/openmp/

Κάθε πρόβλημα περιέχει υποφακέλους, οπου κάθε υποφάκελος αποτελεί και μια παραλλαγή του προβλήματος.



Σχήμα 12: Διάρθρωση παραδειγμάτων στο github

4.1 Πρόβλημα προσπέλασης συνδεδεμένης λίστας

Το παρακάτω παράδειγμα διατρέχει μια συνδεδεμένη λίστα, όπου κάθε κόμβος της θα χρησιμοποιεί τα δεδομένα του για την εκτέλεση μια μεμονομένης διεργασίας, ξεχωριστής από τους υπόλοιπους κόμβους. Στην προκειμένη περίπτωση τα δεδομένα των κόμβων είναι ένας ακέραιος αριθμός.

Κάθε κόμβος της λίστας αρχικοποιείται με τον αριθμό **33**. Η κάθε εργασία θα έχει ως στόχο τον υπολογισμό του αριθμού της ακολουθίας *fibonacci*. Το μέγεθος του προβλήματος επηρεάζεται από τον αριθμό των συνολικών κόμβων της λίστας και όχι απο την τιμή που πρέπει να υπολογιστεί για την ακολουθία *φιβοναςςι*η οποία είναι πάντα σταθερή και ίση με **33**. Η ανεξαρτησία των κόμβων διευκολύνει τον παράλληλο υπολογισμό. Ωστόσο, οι συνδεδεμένες λίστες διετρέχονται σειριακά, καταστώντας τον παραλληλισμό μή εφικτό.

Συμβ. 26: Συνάρτηση αρχικοποίησης κόμβων

```
Node *init_nodes(int num, int value) {
      Node *head = new Node(value);
      Node *temp = nullptr;

      Node *p = head;
      for (int i = 0; i < num; ++i) {
          temp =  new Node(value);
          p->next_ = temp;
          p = temp;
          p->data_ = value;
      }
      p->next_ = nullptr;
      return head;
 }
```

Συμβ. 27: Δομή κόμβου συνδεδεμένης λίστας

```
struct Node {
    Node *next_ = nullptr; // pointer to the next node
    int data_ = 0;         // the stored data is an integer in this example
};
```

Λύση στο πρόβλημα, αποτελεί η αρχική προσπέλαση της λίστας με σκοπό την εισαγωγή των κόμβων της σε διάνυσμα και στη συνέχεια η παραλληλη εκτέλεση μέσω οδηγίας διαμοιρασμού εργασίας βρόγχου - *for*. Αυτή η λύση ωστόσο προκαλεί τα παρακάτω προβλήματα:

1. Γίνεται αρχικά μία προσπέλαση σε όλους τους κόμβους της λιστας, με σκοπό την αποθήκευση τους σε ένα σειριακό μέσο αποθήκευσης.

2. Σε περίπτωση που ο αριθμός των κόμβων της λίστας δεν ειναι γνωστός, θα πρέπει να γίνουν περισσότερες απο μία δεσμεύσεις μνήμης *malloc*, διαδικασίας χρονοβόρας.

3. Γίνεται άσκοπη δέσμευση μνήμης για το διάνυσμα που θα αποθηκευτούν οι κόμβοι. Σε περίπτωση μεγάλου αριθμού κόμβων λίστας, αυτό μπορει να αποτελέσει πρόβλημα.

Συμβ. 28: Λύση προβλήματος συνδεδεμένης λίστας με χρηση οδηγίας διαμοιρασμού εργασίας

```
std::vector<Node *> nodes_(o.num_nodes_);
  for (int i = 0; i < o.num_nodes_; ++i) {
      nodes_[i] = p;
      p = p->next_;
  }
#pragma omp parallel
  {
 #pragma omp for schedule(static, 1)
      for (int i = 0; i < o.num_nodes_; ++i) {
          fib(p->data_);
      }
  }
```

Τα παραπάνω προβλήματα επιλύονται με την χρήση διεργασιών. Ο κώδικας διέρχεται μια φορά από την λίστα με σκοπό την δημουργία μιας διεργασίας για κάθε κόμβο. Οι διεργασίας ξεκινούν να εκτελούνται αυτόματα, μόλις βρεθεί μή ενεργό νήμα, δηλαδή μόλις ένα νήμα φτάσει στο φράγμα κώδικα.

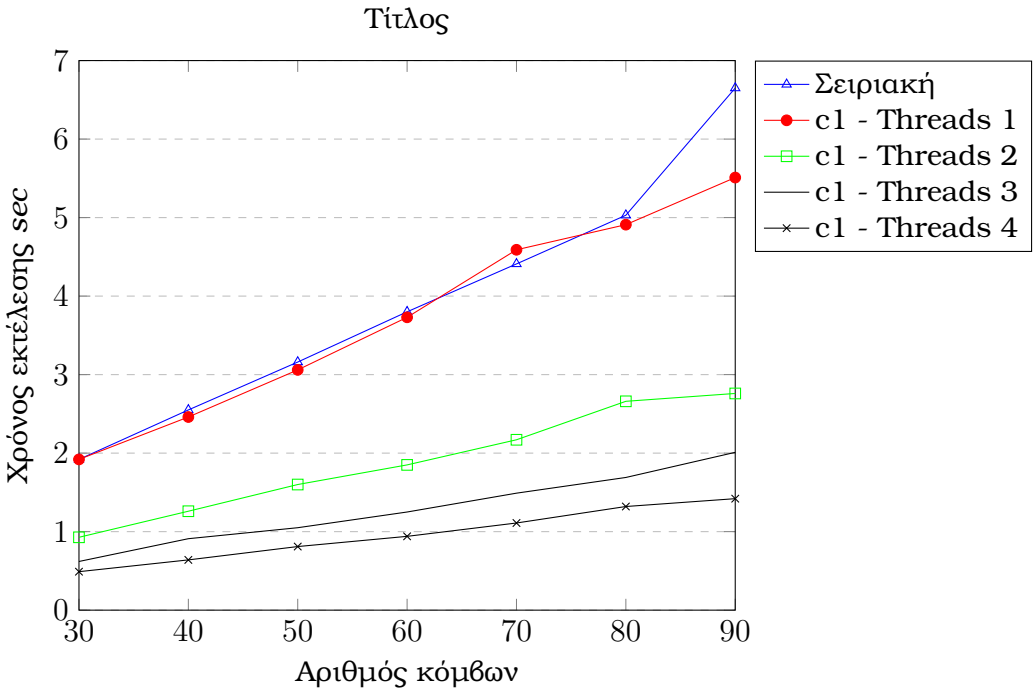
Συμβ. 29: Λύση προβλήματος συνδεδεμένης λίστας με χρηση οδηγίας διαμοιρασμού εργασίας

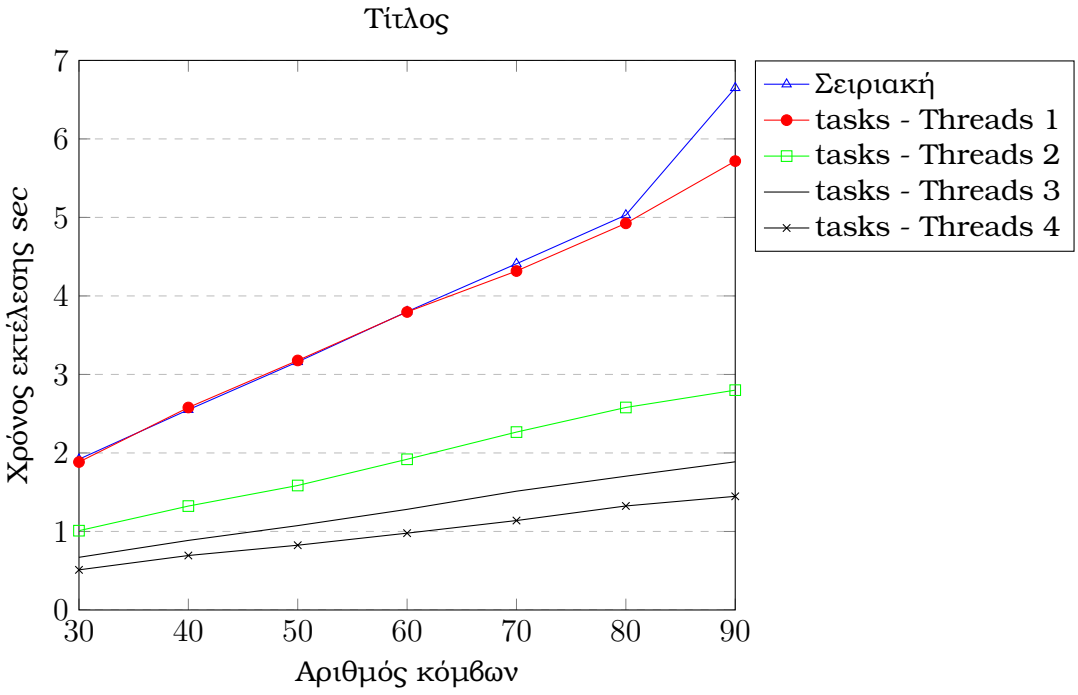
```
#pragma omp parallel
{
 #pragma omp single
 {
     Node *p = head;
     while (p) {
         #pragma omp task firstprivate(p)
             fib(p->data_);
         p = p->next_;
```

```
            }
        }
    }
```

4.1.1 Αποτελέσματα εκτέλεσης

Στα παρακάτω διαγράμματα γίνεται σύγκριση με αποτελέσματα απο του αλγόριθμου με διαφορετικές μεθόδους.





4.1.2 Συμπεράσματα και παρατηρήσεις

4.2 Πρόβλημα πολλαπλασιασμού πινάκων

Σε αυτό το πρόβλημα γίνεται προσπάθεια επίλυσης πολλαπλασιασμού διδιάστατων πινάκων, αποτελούμενων απο τυχαίους ακεραίους:

$$C[K][M] = A[K][N] * B[N][M]$$

Για απλούστευση κώδικα, οι πίνακες που χρησιμοποιήθηκαν είναι τετραγωνικοί.

Συμβ. 30: Σειριακή μέθοδος πολλαπλασιασμού πινάκων

```
for (int i = 0; i < K; ++i) {
    for (int j = 0; j < M; ++j) {
        int tmp = 0;
        for (int k = 0; k < N; ++k) {
            tmp += A[i][k] * B[k][j];
        }
        C[i][j] = tmp;
    }
}
```

Συμβ. 31: Παράλληλος πολλαπλασιασμός με χρήση οδηγίας διαμοιρασμού βρόγχου

```
#pragma omp parallel for
  for (int i = 0; i < r1; ++i) {
      for (int j = 0; j < c2; ++j) {
          int tmp = 0;
          for (int k = 0; k < c1; ++k) {
              tmp += A[i][k] * B[k][j];
          }
          C[i][j] = tmp;
      }
  }
```

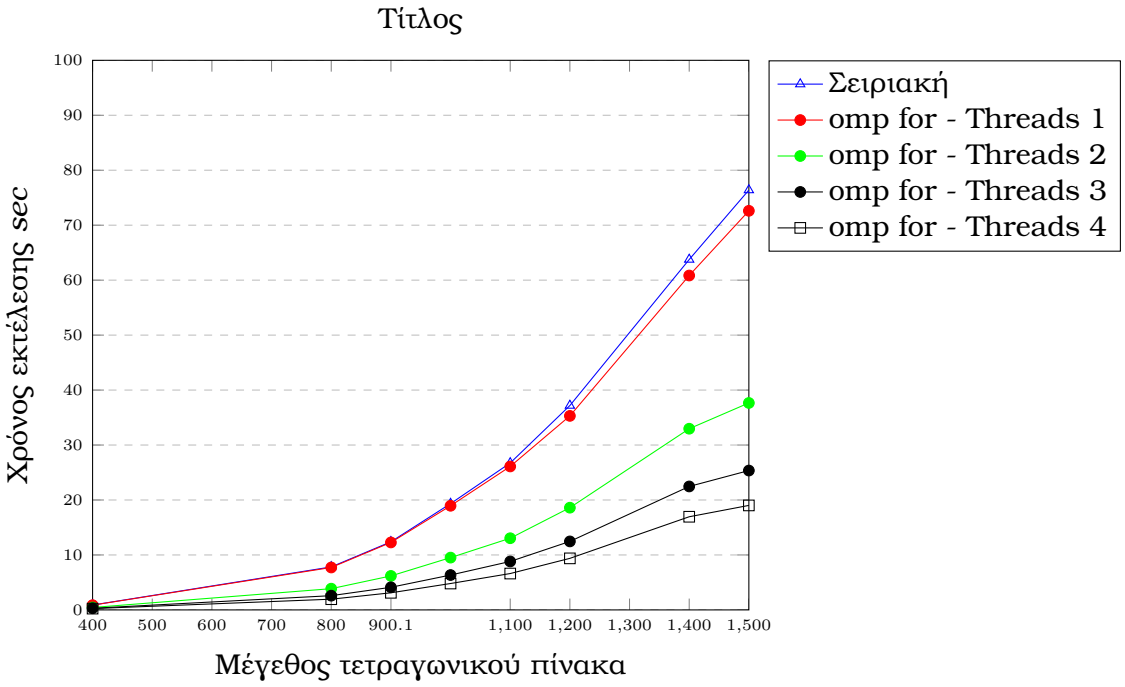
Αλλη λύση του προβλήματος, είναι με διαχωρισμό των πινάκων σε υποπίνακες, και πολλαπλασιασμού των υποπινάκων ως ξεχωριστές διεργασίες[13].

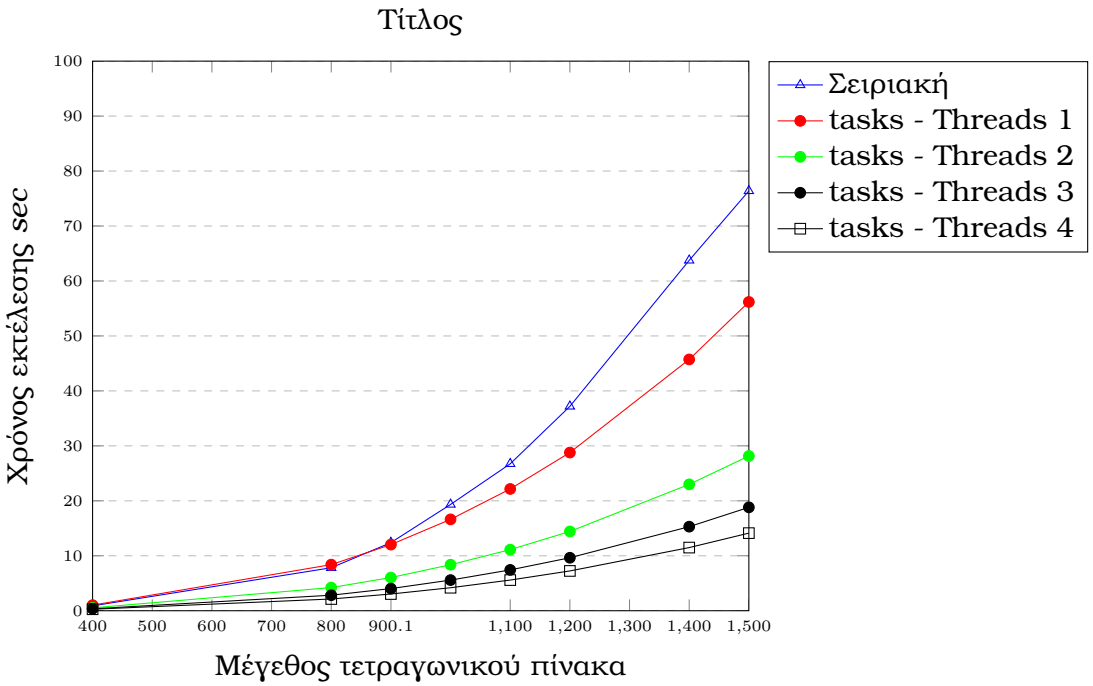
Συμβ. 32: Πολλαπλασιασμός πινάκων με χρήση διεργασιών (Πολλαπλασιασμός με χρήση υποπινάκων)

```
void matmul(int N, int BS, int **A, int **B, int **C)
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
      for (j = 0; j < N; j+=BS) {
          for (k = 0; k < N; k+=BS) {
      // Note 1: i, j, k, A, B, C are firstprivate by default
      // Note 2: A, B and C are just pointers
      #pragma omp task private(ii, jj, kk) \
              depend (in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
              depend (inout: C[i:BS][j:BS] )
      for (ii = i; ii < i+BS; ii++ )
        for (jj = j; jj < j+BS; jj++ )
          for (kk = k; kk < k+BS; kk++ )
            C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
            }

        }
    }
}
```

4.2.1 Αποτελέσματα εκτέλεσης

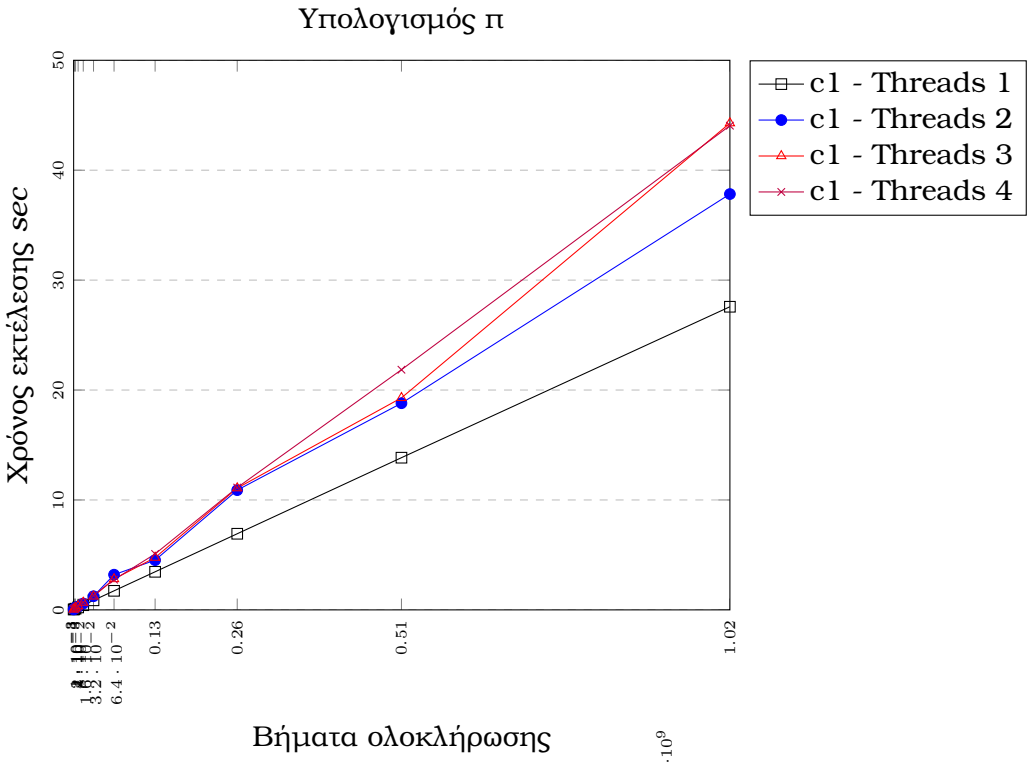
Συγκριτικά διαγράματα εκτελέσης των τριών μεθόδων με διαφορετικά μεγέθη πινάκων αναφέρονται παρακάτω.

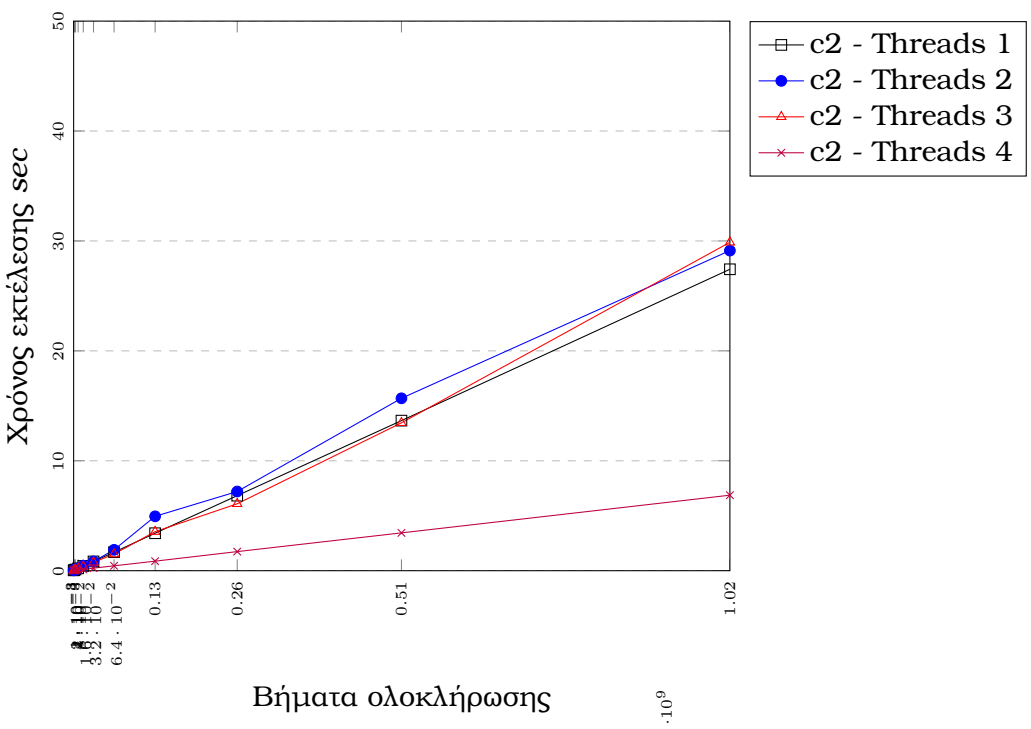


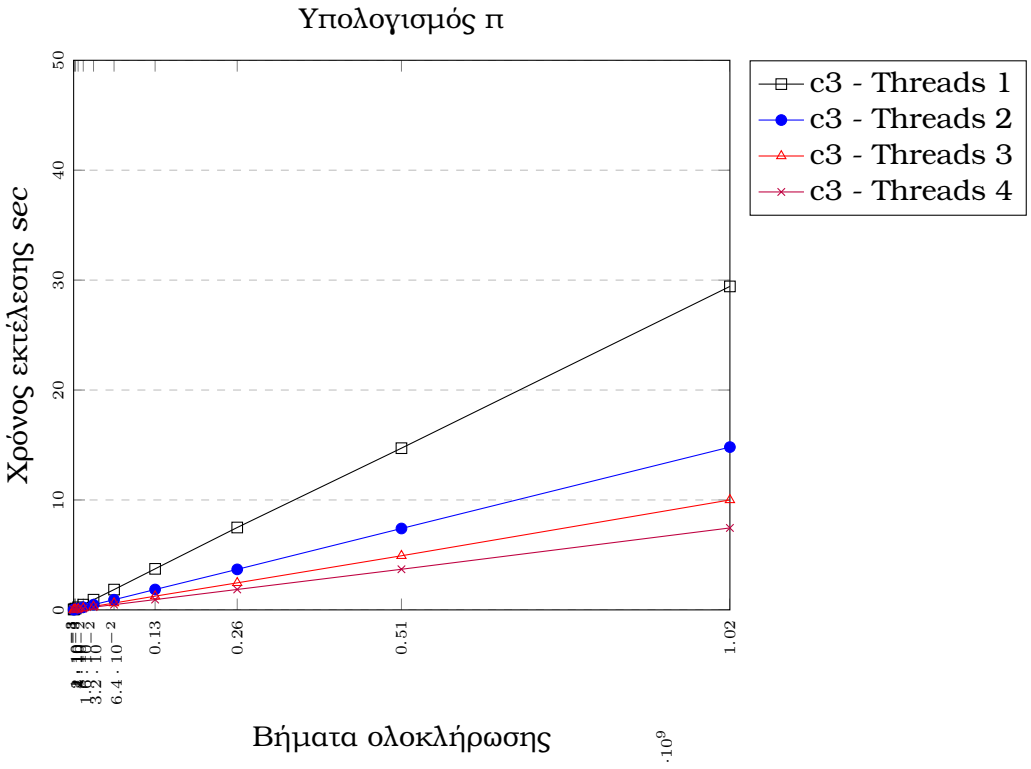


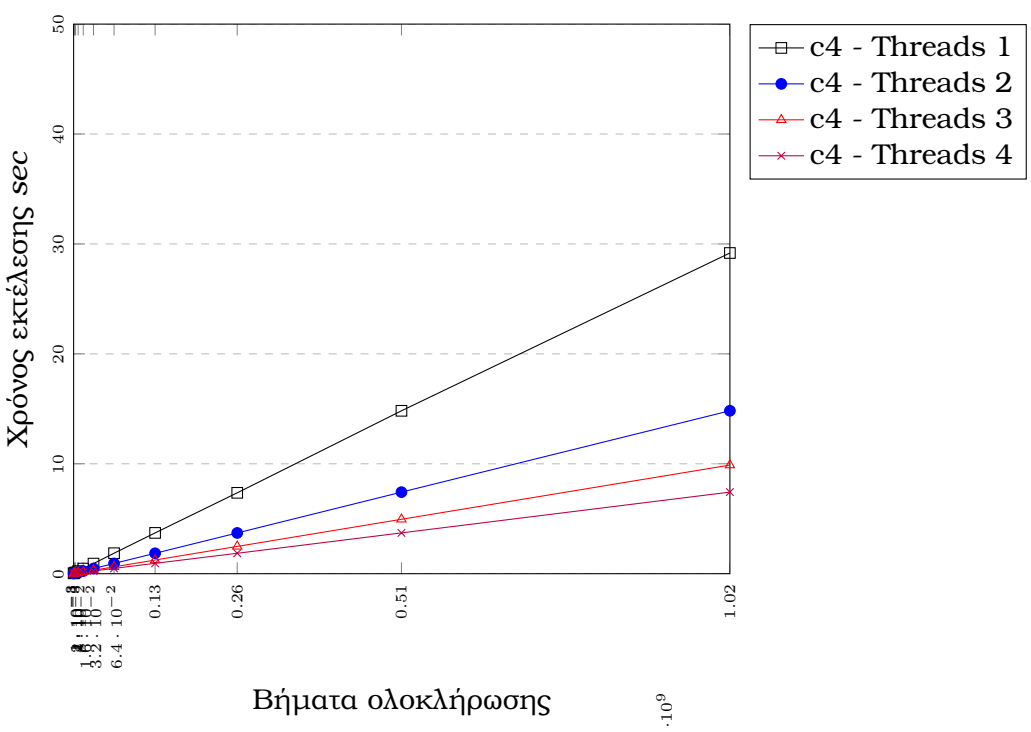
4.3 Υπολογισμού π

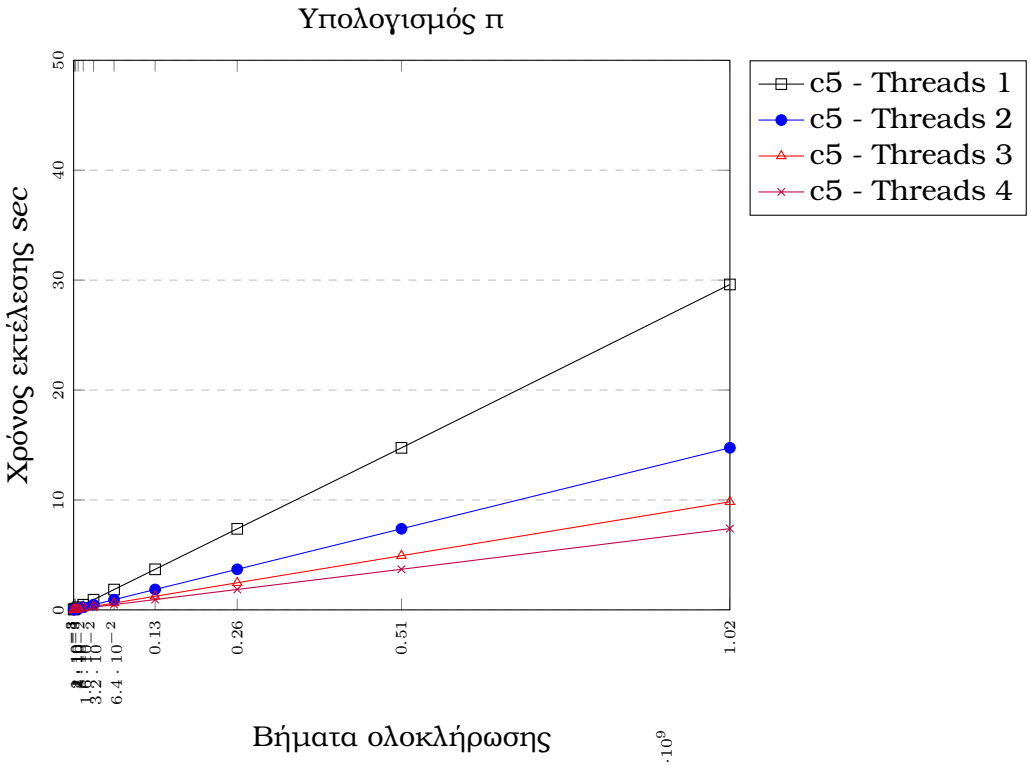
Να γράψω τα βήματα του ιντεγκρατιον και τι εινα ιτο ιντεγκρασιον.

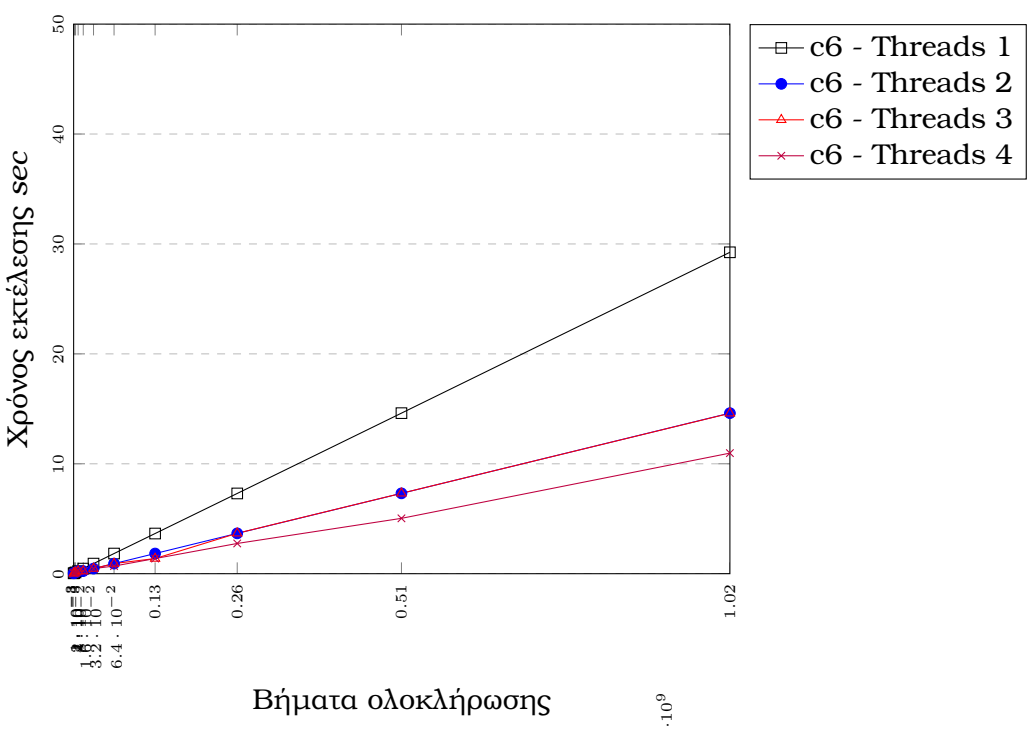

Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c1 - Threads 1
c1 - Threads 2
c1 - Threads 3
c1 - Threads 4


Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c2 - Threads 1
c2 - Threads 2
c2 - Threads 3
c2 - Threads 4


Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c3 - Threads 1
c3 - Threads 2
c3 - Threads 3
c3 - Threads 4

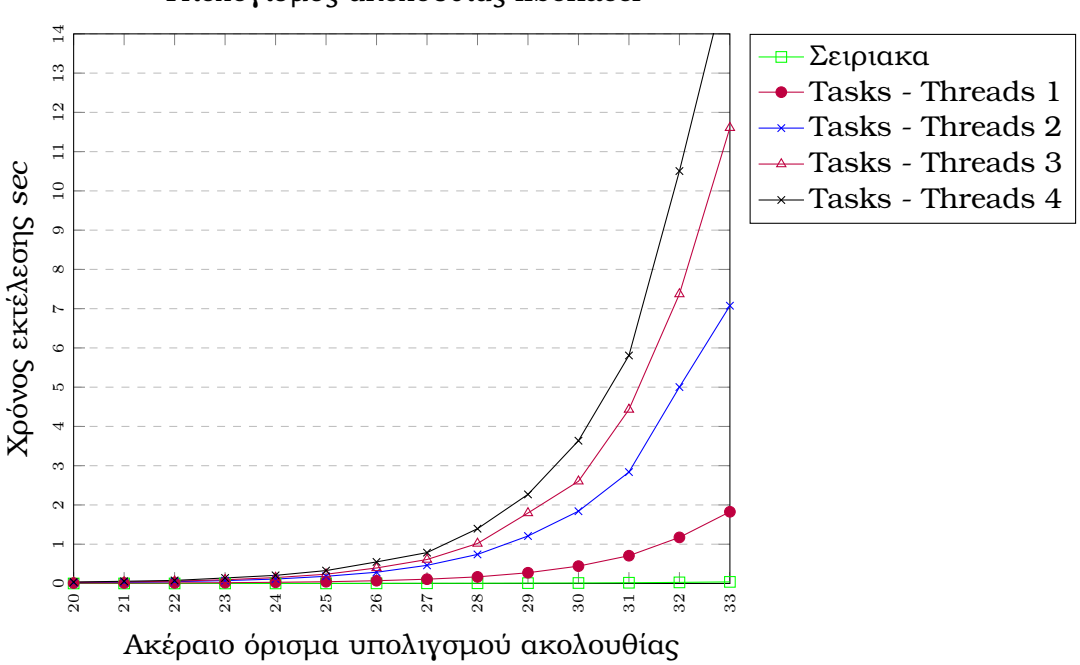

Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c4 - Threads 1
c4 - Threads 2
c4 - Threads 3
c4 - Threads 4


Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c5 - Threads 1
c5 - Threads 2
c5 - Threads 3
c5 - Threads 4


Υπολογισμός π
Χρόνος εκτέλεσης sec
Βήματα ολοκλήρωσης
$\cdot 10^9$
c6 - Threads 1
c6 - Threads 2
c6 - Threads 3
c6 - Threads 4

4.4 Πρόβλημα ακολουθίας Fibonacci

Στο παρακάτω παράδειγμα, γίνεται ο υπολογισμός αριθμών της ακολουθίας Φιβοναςςι.



Ωηιλε τηε ιφ ϛλαυσε αφφεϛτς τηε τασκ βεινγ δεφινεδ βψ α τασκ ϛονστρυϛτ (ανδ ονλψ τηατ ονε), τηε φιναλ ϛλαυσε αφφεϛτς αλλ ιτς δεσϛενδαντ τασκς (εεν ιφ τηεψ δο νοτ ηαε α φιναλ ϛλαυσε τηεμσελες)

Η έκφραση της συνθήκης *if* μιας διεργασίας, θα πρέπει να αξιολογείται ώς ψευδής ή αληθής. Η οδηγία των διεργασιών παρέχει μια φράση *if*, που δέχεται μια έκραση ως όρισμα. Αν η έκφραση αξιολογείται ώς ψευδής, απαγορεύει η αναβολή της διεργασίας. Ανεξαρήτου αποτελέσματος της έκφρασης, δημιουργείται πάντα ένα νέο περιβάλλον δεδομένων εργίας. Αν η έκφραση αξιολογείται ως ψευδής, δεν γίνονται όλοι οι υπολογισμοί που απετούνται για μια διεργασία αν δεν υπήρχε η φράση ιφ, επομένως θα μπορούσαν να αποφευχθούν ορισμένα κόστη επιδόσεων.

Ως εκ τούτου η φράση *if* αποτελεί λύση σε καταστάσεις, όπως οι αναδρομικοί αλγόριθμοι, όπου το υπολογιστικό κόστος μειώνεται καθώς αυξάνεται το βάθος και το όφελος από τη δημιουργία μιας νέας διεργασίας μειώνεται λόγω του γενικού κόστους. Ωστόσο, για λόγους ασφαλείας [9], οι μεταβλητές που αναφέρονται σε μια οδηγία κατασκευής διεργασιών είναι, στις περισσότερες περιπτώσεις, από προεπιλογή *firstprivate*, επομένως το κόστος της ρύθμισης περιβάλλοντος δεδομένων μπορεί να είναι το κυρίαρχο συστατικό της δημιουργίας διεργασιών.

Στην περίπτωση αληθούς έκφρασης Τηε σϛαλαρ εξπρεσσιον ον τηις ϛλαυσε μυστ εαλυατε το τρυε ορ φαλσε. Ιν τηε ϛασε οφ τηε λαττερ, τηε ενϛουντερινγ τασκ ις συσπενδεδ ανδ τηε νεω τασκ ις εξεϛυτεδ ιμμεδιατελψ. Τηε παρεντ τασκ ρεσυμες, ονϛε τηε νεω τασκ ηας ϛομπλετεδ. Τηις φεατυρε ϛαν βε υσεδ βψ αν ιμπλεμεντατιον το ιμπροε τηε περφορμανϛε βψ αοιδινγ χυευεινγ τασκς τηατ αρε τοο σμαλλ. Τηε διφφερενϛε ωιτη τηε φιναλ ϛλαυσε βελοω, ις τηατ, ωιτη τηε ιφ ϛλαυσε, τηε ϛηιλδ τασκς αρε νοτ αφφεϛτεδ. Τηις ις ωηψ ιτ ωας νοτ υσεδ ιν τηε εξαμπλε σηοων ιν Φιγυρε 3.19. Δυε το τηε νατυρε οφ τηε χυιϛκσορτ αλγοριτημ, ονϛε τηε λενγτη οφ τηε αρραψ δροπς βελοω τηε τηρεσηολδ, τηε λενγτη φορ ϛηιλδ τασκς ις αλσο βελοω τηις αλυε.

5 Επίλογος

Εδώ συνοψίζουμε την παρουσίαση της διπλωματικής.

5.1 Σύνοψη και συμπεράσματα

Εδώ συνοψίζουμε τα αποτελέσματα της διπλωματικής και περιγράφουμε τα συμπεράσματα που προέκυψαν, αρνητικά και θετικά. Επιβεβαιώνουμε τη συνεισφορά της διπλωματικής στα προβλήματα που αναφέραμε στην εισαγωγή. Τα συμπεράσματα θα πρέπει να παρουσιάζονται συστηματικά για κάθε αντικειμενικό στόχο ή υπόθεση που έχουμε κάνει.

5.2 Όρια και περιορισμοί της έρευνας

5.3 Μελλοντικές Επεκτάσεις

Εδώ δίνουμε ιδέες για επέκταση της διπλωματικής. Αναφέρουμε ότι θα ακολουθήσει παρουσίαση μελλοντικών κατευθύνσεων έρευνας ανά θεματική περιοχή. Περιγράφουμε προβλήματα που δεν έχουν λυθεί από τις τεχνικές/μεθοδολογίες που παρουσιάσαμε στο προηγούμενο κεφάλαιο. Τα άλυτα αυτά προβλήματα, αποτελούν στην ουσία προκλήσεις για περαιτέρω έρευνα. Ακόμα καλύτερα, θα ήταν ωραία να προτείνουμε τρόπους επίλυσης των προβλημάτων αυτών έστω και ως γενική ιδέα.

References

[1] Worksharing constructs. https://www.openmp.org/spec-html/5.0/openmpse16.html. Accessed: 2020-05-20.

[2] *An Extension to Improve OpenMP Tasking Control (Tsukuba, Japan, June 2010)*, volume 6132 of *Lecture Notes in Computer Science*, Berlin, Germany, 2010. Springer.

[3] *The Design of OpenMP Thread Affinity (Heidelberg, Berlin, June 2012)*, volume 7312 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2012. Springer.

[4] *OpenMP 4.5 Validation and Verification Suite for Device Offload (Barcelona, Spain, September 2018)*, volume 11128 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2018. Springer.

[5] O. ARB. Openmp 4.5 api c/c++ syntax reference guide. https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf. Accessed: 2020-06-20.

[6] R. v. d. P. Barbara Chapman, Gabriele Jost. *Using OpenMP-Portable Shared Memory Programming*. The MIT Press, 2007.

[7] B. Barney. Openmp. https://computing.llnl.gov/tutorials/openMP/. Accessed: 2020-05-20.

[8] I. Corporation. False sharing. https://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_c/main_cls/cref_cls/common/cilk_false_sharing.htm. Accessed: 2020-04-13.

[9] A. D. Eduard Ayguade, Nawal Copty. *The Design of OpenMP Tasks*, pages 404–418. IEEE Trans, 2009.

[10] M. Flynn. *Some Computer Organizations and Their Effectiveness*, pages 948–960. IEEE, 1972.

[11] T. Harware. Amd unveils its heterogeneous uniform memory access (huma) technology. https://www.tomshardware.com/news/AMD-HSA-hUMA-APU,22324.html. Accessed: 2020-06-13.

[12] K.Margaritis. Introduction to openmp. pdplab.it.uom/teaching/llnl-gr/OpenMP.html#Abstract. Accessed: 2020-06-25.

[13] OpenMP. *OpenMP Application Programming Interface Examples*, page 78. OpenMP, 2016.

[14] L. D. Rohit Chandra, Ramesh Menon. *Parallel Programming in OpenMP*, page 21. Morgan Kaufmann, 2000.

[15] L. D. Rohit Chandra, Ramesh Menon. *Parallel Programming in OpenMP*, page 23. Morgan Kaufmann, 2000.

[16] C. T. Ruud van der Pas, E. Stotzer. *Using OpenMP. The Next Step: affinity, accelerators, tasking, SIMD*, page 59. The MIT Press, 2017.

[17] C. T. Ruud van der Pas, E. Stotzer. *Using OpenMP. The Next Step: affinity, accelerators, tasking, SIMD*, page 20. The MIT Press, 2017.

[18] C. T. Ruud van der Pas, E. Stotzer. *Using OpenMP. The Next Step: affinity, accelerators, tasking, SIMD*, page 152. The MIT Press, 2017.

[19] C. T. Ruud van der Pas, E. Stotzer. *Using OpenMP. The Next Step: affinity, accelerators, tasking, SIMD*, page 7. The MIT Press, 2017.

[20] C. T. Ruud van der Pas, E. Stotzer. *Using OpenMP. The Next Step: affinity, accelerators, tasking, SIMD*, page 9. The MIT Press, 2017.

[21] Wikipedia. Παράλληλος Προγραμματισμός. el.wikipedia.org/wiki/Parallel_computing. Accessed: 2020-05-26.