ΠΑΝΕΠΙΣΤΉΜΙΟ ΜΑΚΕΔΟΝΊΑΣ

ΠΡΌΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΏΝ ΣΠΟΥΔΏΝ

ΤΜΉΜΑΤΟΣ ΕΦΑΡΜΟΣΜΈΝΗΣ ΠΛΗΡΟΦΟΡΙΚΉΣ

ΠΑΡΆΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΉΣΗ OpenMP

Διπλωματική Εργασία

του

Κοντογιάννη Γεώργιου

Θεσσαλονίκη, Ιούνιος 2021

ΠΑΡΆΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΉΣΗ OpenMP

Κοντογιάννης Γεώργιος

Δίπλωμα Πολιτικού Μηχανικού, ΑΠΘ, 2016

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΎ ΤΊΤΛΟΥ ΣΠΟΥΔΏΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΈΝΗ ΠΛΗΡΟΦΟΡΙΚΉ

Επιβλέπων Καθηγητής
Μαργαρίτης Κωνσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την η/μ/ΕΕΕΑ

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

Κοντογιάννης Γεώργιος

.....

Η σύνταξη της παρούσας εργασίας έγινε στο LaTeX

Περίληψη

Αντικείμενο της διπλωματικής εργασίας είναι η μελέτη του OpenMP, ενός προτύπου παράλληλου προγραμματισμού, που δίνει στο χρήστη τη δυνατότητα ανάπτυξης παράλληλων προγραμμάτων για συστήματα διαμοιραζόμενης μνήμης, τα οποία είναι ανεξάρτητα από τη αρχιτεκτονική του συστήματος και έχουν μεγάλη ικανότητα κλιμάκωσης[24].

Σκοπός της εργασίας είναι η συνοπτική ανακεφαλαίωση των βασικών χαρακτηριστικών των παλαιών εκδόσεων (OpenMP 2.5), η μελέτη και περιγραφή των κύριων χαρακτηριστικών των νεότερων (3.0 και 4.5) καθώς και η υλοποίηση αλγορίθμων σειριακά και παράλληλα εκτελέσιμων, με σκοπό τη συγκριτική μελέτη των παραλλαγών του κάθε προβλήματος για την εξαγωγή συμπερασμάτων. Για την παράλληλη υλοποίηση των προβλημάτων χρησιμοποιούνται χαρακτηριστικά όλων των εκδόσεων του OpenMP, συμπεριλαμβανομένων των Tasks, SIMD και Offloading[30].

Τον Μάιο του 2008 κυκλοφόρησε η έκδοση του OpenMP 3.0. Στην κυκλοφορία συμπεριλήφθηκε για πρώτη φορά η έννοια των εργασιών(Tasking) αλλά και βελτιώσεις στην υποστήριξη της διεπαφής μέσω της C++. Αποτελεί την πρώτη ενημέρωση μετά την έκδοση 2.5 με σημαντικές αλλαγές. Το 2011 κυκλοφόρησε η OpenMP 3.1 χωρίς αξιοσημείωτες νέες προσθήκες. Νέα χαρακτηριστικά ωστόσο, εισήχθησαν στο OpenMP 4.0 που κυκλοφόρησε τον Ιούλιο του 2013, όπου έγινε υποστήριξη της αρχιτεκτονικής cc-NUMA, του ετερογενούς προγραμματισμού, της διαχείρισης σφαλμάτων στην περιοχή παράλληλου κώδικα και της διανυσματικοποίησης μέσω SIMD. Τον Ιούλιο του 2015 σημαντική βελτίωση έγινε στα παραπάνω χαρακτηριστικά με την έκδοση OpenMP 4.5[31].

Τα προαναφερθέντα χαρακτηριστικά χρησιμοποιήθηκαν για την υλοποίηση αλγορίθμων σε διάφορες παραλλαγές, με σκοπό τη συγκριτική μελέτη τους για την εξαγωγή συμπερασμάτων αναφορικά με τη βελτίωση της απόδοσης σε σχέση με τη σειριακή υλοποίηση, καθώς επίσης την αξιολόγηση της ευχρηστίας της υλοποίησής τους. Στόχος της έρευνας είναι η συλλογή και καταγραφή παρατηρήσεων που προκύπτουν σε κάθε υλοποίηση, για την καλύτερη κατανόηση των εννοιών του παράλληλου

προγραμματισμού.

Για την ικανότητα παραλληλοποίησης του κώδικα, απαιτούνται αλγόριθμοι που αποτελούνται από εργασίες ανεξάρτητες μεταξύ τους και ικανές να εκτελεστούν ταυτόχρονα, σε διαφορετικούς επεξεργαστές. Τέτοιοι αλγόριθμοι είναι οι εξής:

- μετασχηματισμός *Fourier - (Discrete Fourier Transform)*,
- ταξινόμηση *Mergesort*,
- ταξινόμηση *Quicksort*,
- *Producer-Consumer*,
- υπολογισμός π,
- υπολογισμός πρώτων αριθμών,
- υπολογισμός εσωτερικού γινομένου,
- πολλαπλασιασμός μητρώων,
- Single precision A X plus Y - *SAXPY*,
- *Linked list* traversal

Υλοποιήσεις των παραπάνω προβλημάτων χρησιμοποιούνται στα πλαίσια της παρούσας εργασίας. Δημιουργήθηκαν παραλλαγές του κάθε προβλήματος που χρησιμοποιούν την κεντρική μονάδα επεξεργασίας (**CPU**) για σειριακή και παράλληλη εκτέλεση. Όπου είναι εφικτό υλοποιείται παραλλαγή για εκτέλεση μέσω της μονάδας επεξεργασίας της κάρτας γραφικών(**GPU**). Οι χρονικές καταγραφές συγκρίνονται μεταξύ τους. Ακόμη, γίνεται αξιολόγηση της ευχρηστίας για την υλοποίηση της κάθε παραλλαγής καθώς και καταγραφή παρατηρήσεων που προκύπτουν.

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

Abstract

This study examines the OpenMP, a cross-platform parallel programming Application Programmers Interface (API) built into all modern C and C++ compilers. The API makes it relatively easy to add parallelism to sequentially implemented programs, as well as created new from scratch.

The purpose of this study is to briefly summarize the main features of the older OpenMP versions (until version 2.5), the description of the main features of newer versions such as 3.0 and 4.5, as well as the implementation of algorithms both sequentially and in parallel alternatives, in order to compare them with each other.

Tasking is the main feature introduced in OpenMP version 3.0(May, 2008). In 2011, version 3.1 contained mainly bug fixes. More features were introduced in version 4.0, which was released in July 2013. Some of them are: cc-NUMA, heterogenous computing, error-handling and vectorization using SIMD. Enhancements in the aforementioned are included in version 4.5.

The features described above are used for the implementation of some algorithms in multiple variations, in order to compare them and draw conclusions regarding their efficiency, simplicity and ease of implementation. The chosen algorithms contain independent tasks, are amenable to parallelism and capable of running by different threads. Those algorithms are:

- Fourier - (Discrete Fourier Transform),
- Mergesort,
- Quicksort,
- Producer-Consumer,
- π calculation,
- Prime numbers calculation,
- Dot product calculation,
- Matrix Multiplication,
- Single precision A X plus Y - **SAXPY**,
- Linked list traversal

Keywords: OpenMP, parallel programming, threads, examples, SIMD, vectorization, GPU, offloading, heterogeneous computing, tasks, cc-NUMA

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

Κατάλογος Πινάκων

Λίστα Συμβολισμών

1 Εισαγωγή

1.1 Συνοπτικά για το OpenMP

Το OpenMP είναι μια Διεπαφή Προγραμματισμού Εφαρμογών (*API*) που χρησιμοποιείται για παραλληλοποίηση συστημάτων διαμοιραζόμενης μνήμης από λογισμικά γραμμένα σε γλώσσες **C/C++** και **Fortran**. Η διεπαφή αποτελείται από τα παρακάτω σύνολα[31]:

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

Οι εντολές παραλληλοποίησης εφαρμόζονται στο τμήμα του κώδικα που ακολουθεί της οδηγίας. Κάθε κατασκευή ξεκινάει με ***#pragma omp*** ακολουθούμενη από οδηγίες για το μεταγλωττιστή. Το δομημένο τμήμα κώδικα μπορεί να αποτελείται από μια εντολή ή ένα σύνολο εντολών[13]. Οι εντολές που βρίσκονται εντός της περιοχής παράλληλου κώδικα, εκτελούνται από όλα τα νήματα που δημιουργούνται κατά τη διάρκεια της παραλληλοποίησης. Η παραλληλοποίηση ολοκληρώνεται με το πέρας της εκτέλεσης των εντολών εντός αυτής της περιοχής.

Συμβ. 1: Γραμματική σύνταξης οδηγίας OpenMP

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

Με τη χρήση του *OpenMP* οι εφαρμογές εκμεταλλεύονται την ύπαρξη πολλαπλών επεξεργαστικών μονάδων, με σκοπό την επίτευξη αύξησης 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, δηλαδή με χρονομερισμό[37].

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

Σκοπός της παρούσας εργασίας είναι η μελέτη της βιβλιογραφίας της διεπαφής του *OpenMP*, δίνοντας μεγαλύτερη βαρύτητα στις εκδόσεις από 3.0 έως 4.5, στα χαρακτηριστικά και τις δυνατότητες που εισήχθησαν σε αυτές, καθώς επίσης και στην κατασκευή παραδειγμάτων υλοποίησης αλγορίθμων με αυτά τα χαρακτηριστικά. Αναλύονται σε θεωρητικό επίπεδο οι οδηγίες και φράσεις (*clauses*) των νέων εκδόσεων, ενώ γίνεται μια προσπάθεια υλοποίησης και συγκριτικής μελέτης απλών και σύνθετων προβλημάτων, επιλυμένων με διαφορετικές μεθόδους και παραλλαγές, που βασίζονται στα καινούργια χαρακτηριστικά της διεπαφής. Στόχος είναι η σαφής κατανόηση των εισαγόμενων χαρακτηριστικών της διεπαφής στις συγκεκριμένες εκδόσεις, η εξαγωγή συμπερασμάτων μέσα από τις υλοποιήσεις των προβλημάτων αλλά και η σύγκριση των διαφορετικών μεθόδων επίλυσης κάθε επιμέρους προβλήματος με βάση τις επιδόσεις τους.

1.3 Διάρθρωση της μελέτης

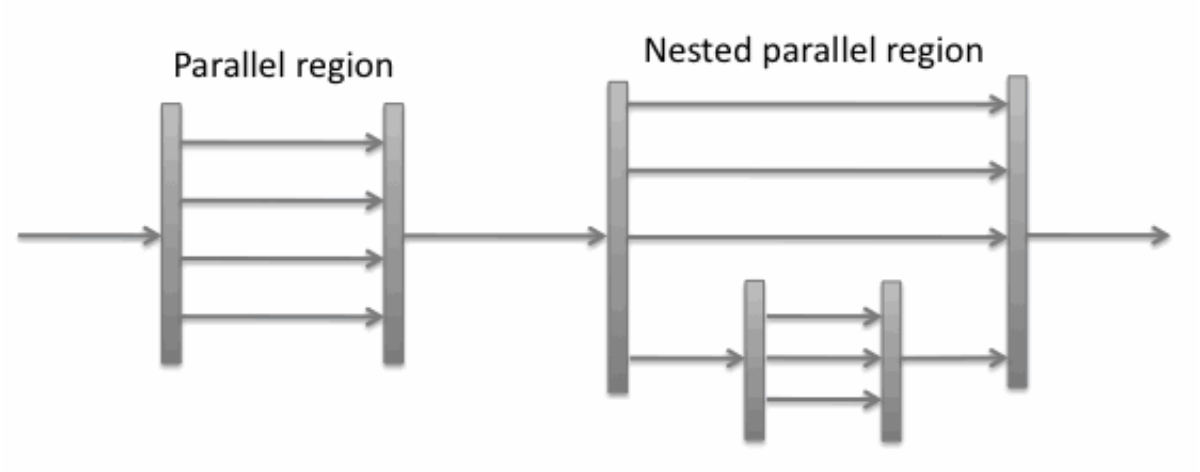
Στα επόμενα κεφάλαια γίνεται μια σύντομη περιγραφή του μοντέλου προγραμματισμού της διεπαφής *OpenMP* και της αλληλεπίδρασης των παραγόμενων νημάτων με το περιβάλλον δεδομένων. Στη συνέχεια, γίνεται σύντομη ανασκόπηση σε έννοιες απαραίτητες για την μελέτη των νέων χαρακτηριστικών που εισήχθησαν στις εκδόσεις μετά την 2.5. Η ανασκόπηση περιορίζεται κυρίως στις οδηγίες και τις φράσεις που υπάρχουν στις παλαιότερες εκδόσεις της διεπαφής. Ακολουθεί ανάλυση σε θεωρητικό υπόβαθρο των σημαντικότερων εννοιών που εισήχθησαν στις εκδόσεις 3.0 - 4.5 όπως αυτή των εργασιών-*Tasking*, του *offloading*, της διανυσματικοποίησης -*vectorization* κ.α. Αμέσως μετά, ακολουθεί η υλοποίηση, ο σχολιασμός και η συγκριτική μελέτη απλών και σύνθετων προβλημάτων. Στο τέλος γίνεται η καταγραφή των σημαντικότερων συμπερασμάτων, εξαγόμενων από τις υλοποιήσεις.

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, που ονομάζεται κύριο (*master thread*), που εκτελεί εντολές σειριακού κώδικα. Η ταυτότητα (*id*) αυτού του νήματος είναι πάντα μηδέν και η διάρκεια ζωής του είναι μέχρι το πέρας της εκτέλεσης του προγράμματος[24].

Όταν το κύριο νήμα εισέρχεται στην περιοχή παράλληλου κώδικα (parallel region) που ορίζεται από το OpenMP, τότε δημιουργούνται περισσότερα νήματα και το τμήμα εκτελείται ταυτόχρονα από τα παραγόμενα νήματα. Με την ολοκλήρωση της εκτέλεσης του παράλληλου τμήματος, όλα τα νήματα που δημιουργήθηκαν τερματίζουν και συνεχίζει μόνο το κύριο, μέχρι να βρεθεί κάποιο άλλο τμήμα παράλληλου κώδικα (fork-join μοντέλο)[24]. Το κύριο νήμα είναι υπεύθυνο για την δημιουργία των επιπλέον νημάτων για τη συνολική εκτέλεση. Τα νήματα που είναι ενεργά σε μια παράλληλη περιοχή αναφέρονται ως "ομάδα" (*thread team*). Πάνω από μία ομάδες νημάτων μπορεί να είναι ενεργές ταυτόχρονα[15].

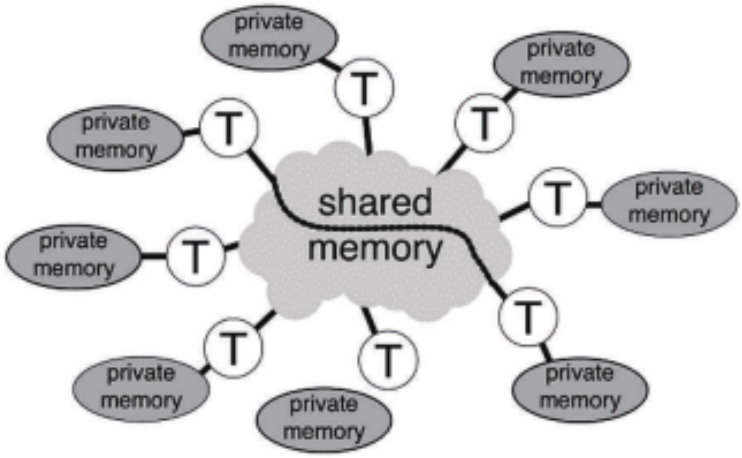


Σχήμα 1: Κύριο νήμα και ομάδες νημάτων

2.2 Αλληλεπίδραση νημάτων και περιβάλλοντος δεδομένων

Όπως προαναφέρθηκε, η εκτέλεση του προγράμματος ξεκινάει από το κύριο νήμα. Το νήμα αυτό συσχετίζεται με ένα περιβάλλον δεδομένων. Το περιβάλλον δεδομένων για ένα νήμα είναι ο χώρος διευθύνσεων μνήμης στον οποίο εισάγονται όλες οι μεταβλητές του προγράμματος, περιλαμβανομένων των *καθολικών* μεταβλητών, των μεταβλητών που είναι αποθηκευμένες στη μνήμη *stack* και αυτών που είναι αποθηκευμένες στη *heap*[28].

Στο μοντέλο μνήμης του OpenMP, τα δεδομένα χωρίζονται σε δύο βασικές κατηγορίες μνήμης: στα ιδιωτικά(*private*) και τα κοινόχρηστα(*shared*). Όλα τα νήματα έχουν πρόσβαση χωρίς περιορισμούς σε μεταβλητές που είναι αποθηκευμένες στην κοινόχρηστη μνήμη[33].



Σχήμα 2: Μοντέλο μνήμης OpenMP

2.2.1 Ιδιωτική μνήμη

Πρόκειται για τη μνήμη που είναι προσβάσιμη και μπορεί να τροποποιηθεί από ένα μοναδικό νήμα. Κάθε νήμα δε μπορεί να έχει πρόσβαση στην ιδιωτική μνήμη των υπόλοιπων νημάτων. Η διάρκεια ζωής μιας μεταβλητής στην ιδιωτική μνήμη είναι περιορισμένη και διαρκεί όσο εκτελείται ο παράλληλος κώδικας. Από προεπιλογή, κάθε ιδιωτική μεταβλητή δεν είναι αρχικοποιημένη στην αρχή της παράλληλης περιοχής[34].

2.2.2 Κοινόχρηστη μνήμη

Εκτός από την ιδιωτική, η δεύτερη κατηγορία μνήμης είναι η κοινόχρηστη, όπως φαίνεται στο προηγούμενο σχήμα. Σε αντίθεση με την ιδιωτική, υπάρχει μόνο μία κοινόχρηστη μνήμη κατά τη διάρκεια εκτέλεσης του προγράμματος, η οποία είναι προσπελάσιμη από όλα τα νήματα. Έτσι, 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 Συνθήκη συναγωνισμού (Race Condition)

Το φαινόμενο αυτό εμφανίζεται σε περιπτώσεις που μια ρουτίνα χρησιμοποιεί δεδομένα από τη κοινόχρηστη μνήμη. Αν αποτελεί τμήμα παράλληλου κώδικα, πολλά νήματα ενδέχεται να προσπαθήσουν να τροποποιήσουν ταυτόχρονα την ίδια διεύθυνση μνήμης, μέσω αυτής της ρουτίνας. Το πρόβλημα αυτό ονομάζεται *race condition* και οδηγεί σε εσφαλμένους υπολογισμούς.

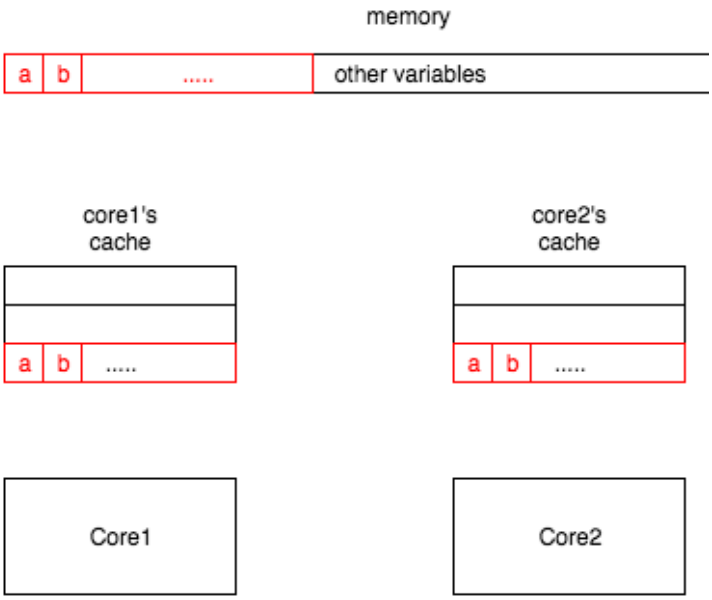
Συμβ. 3: Παράδειγμα συνθήκης συναγωνισμού

```
#include <omp.h>

int main(void) {int sum = 0;

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

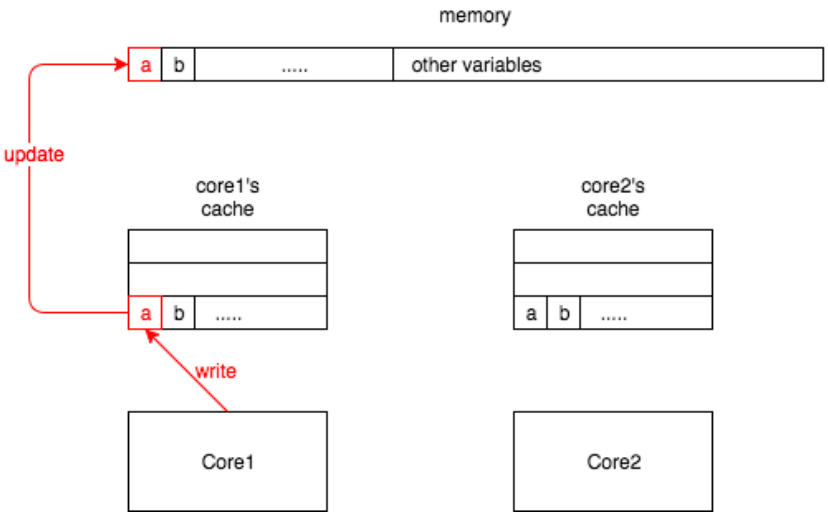
Η απλούστερη λύση, είναι η χρήση της κατάλληλης οδηγίας που επιβάλλει στο πρόγραμμα την προσπέλαση της μνήμης μόνο από ένα νήμα κάθε χρονική στιγμή. Τέτοιες οδηγίες είναι για παράδειγμα οι *pragma omp critical/atomic*. Εναλλακτικά, θα πρέπει αν είναι εφικτό, να δημιουργείται ιδιωτικό αντίγραφο μεταβλητών για κάθε νήμα. Έτσι, πολλά νήματα μπορούν ταυτόχρονα να τροποποιούν δεδομένα που βρίσκονται σε διαφορετικές θέσεις μνήμης γιατί οι μεταβλητές ορίζονται στο ιδιωτικό περιβάλλον δεδομένων του κάθε νήματος.

2.2.2.2 Ψευδής διαμοιρασμός (False Sharing)

Το *false sharing* είναι ένα συχνό πρόβλημα στην παράλληλη επεξεργασία κοινόχρηστης μνήμης. Εμφανίζεται όταν δύο ή περισσότεροι πυρήνες κρατούν αντίγραφο της ίδιας γραμμής προσωρινής μνήμης (*cache*).



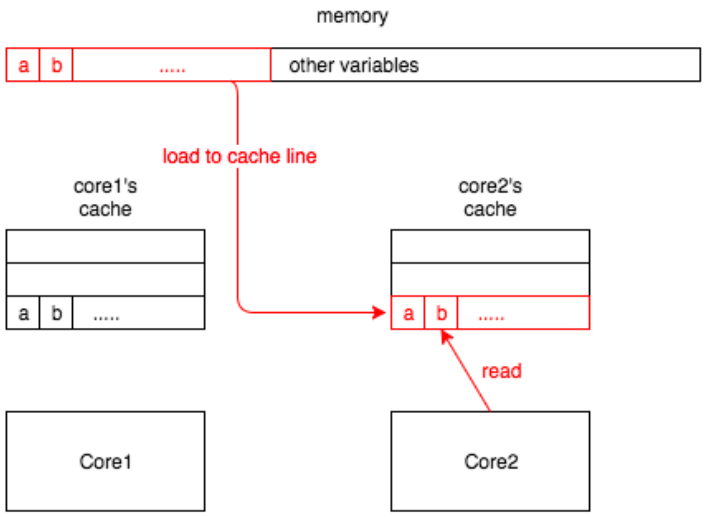
Σχήμα 3: False sharing (1/3)



Σχήμα 4: False sharing (2/3)

Όταν ένα νήμα τροποποιεί μια μεταβλητή, η γραμμή της μνήμης που βρίσκεται η μεταβλητή ακυρώνεται στους υπόλοιπους πυρήνες. Η μνήμης θα πρέπει να ακυρωθεί ακόμη και αν ένας πυρήνας μπορεί να μη τροποποιεί τη συγκεκριμένη θέση μνήμης, αλλά να θέλει να τροποποιήσει ένα άλλο δεδομένο που βρίσκεται σε αυτή.

Ο δεύτερος πυρήνας θα πρέπει να φορτώσει εκ νέου τη γραμμή μνήμης, προτού αποκτήσει ξανά πρόσβαση στα δεδομένα της. Η προσπέλαση δεδομένων της κοινόχρηστης μνήμης συχνά επηρεάζει την απόδοση του προγράμματος[16]. Πιθανή λύση στο πρόβλημα του false sharing, αποτελεί η εισαγωγή τεχνητού κενού (padding) ανάμεσα στα δεδομένα της γραμμής, με σκοπό την τοποθέτησή τους σε ξεχωριστές γραμμές μνήμης.



Σχήμα 5: False sharing (3/3)

2.3 Ανασκόπηση σε βασικά χαρακτηριστικά του *OpenMP*

Στα κεφάλαια που ακολουθούν αναφέρονται επιγραμματικά βασικές έννοιες και χαρακτηριστικά του *OpenMP* που συμπεριλαμβάνονται στην έκδοση 2.5 και χρησιμοποιούνται στα υλοποιημένα παραδείγματα των κεφαλαίων που ακολουθούν. Τα χαρακτηριστικά αυτά εμπλουτίστηκαν ή προστέθηκαν νέα στις επόμενες εκδόσεις.

2.3.1 Μοντέλο συνέπειας μνήμης

Για την αποφυγή φαινομένων *race condition* που οδηγούν σε λανθασμένα αποτελέσματα, απαιτείται συχνά ο συντονισμός πρόσβασης των νημάτων στις μεταβλητές της κοινόχρηστης μνήμης. Η έννοια ¨συνέπεια μνήμης¨, ορίζει πως κατά την εκτέλεση ενός προγράμματος, η διαδοχή των μεταβολών στις τιμές των μεταβλητών φαίνεται σε όλα τα νήματα με την ίδια σειρά. Ο όρος *¨συγχρονισμός¨* αναφέρεται σε μηχανισμούς συνέπειας. Οι οδηγίες συγχρονισμού εγγυώνται την έγκυρη ανάγνωση της σωστής τιμής μιας μεταβλητής στην κοινόχρηστη μνήμη μετά από οποιαδήποτε ενημέρωσή της. Μηχανισμοί συγχρονισμού είναι οι εξής[29]:

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

2.3.2 Οδηγίες διαμοιρασμού εργασίας

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

Μια οδηγία διαμοιρασμού εργασίας δεν έχει κάποιο υποκείμενο εμπόδιο συγχρονισμού(¨*barrier*¨) κατά την είσοδο στον τμήμα της παράλληλης περιοχής. Ωστόσο υπάρχει ένας υποκείμενο φράγμα στο τέλος της οδηγίας. Το φράγμα μπορεί να αναιρεθεί με τη χρήση της ¨*φράσης*¨ (clause) **nowait**. Εάν υπάρχει, το πρόγραμμα θα παραλείψει το φράγμα στο τέλος της οδηγίας και τα νήματα προχωρούν άμεσα στις οδηγίες που ακολουθούν εντός της παράλληλης περιοχής[9].

2.3.2.1 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: Γραμματική οδηγίας διαμοιρασμού εργασίας βρόγχου

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

Συμβ. 5: 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)
```

2.3.2.2 Οδηγία *sections*

Η οδηγία **section** χρησιμοποιείται για τον μη επαναληπτικό διαμοιρασμό εργασίας σε μια παράλληλη περιοχή. Καθορίζει ότι τα εσωκλειώμενα τμήματα κώδικα θα διαμοιραστούν μεταξύ των νημάτων της ομάδας. Μια οδηγία *sections* μπορεί να περιέχει περισσότερες από μία ανεξάρτητες οδηγίες *section*. Κάθε *section* εκτελείται μια φορά από ένα νήμα της ομάδας και διαφορετικά *sections* εκτελούνται από διαφορετικά νήματα. Η σύνταξη μιας οδηγίας *sections* φαίνεται παρακάτω[24]:

Συμβ. 6: Γραμματική 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: Φράσεις οδηγίας sections

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

2.3.2.3 Οδηγία *master*

Η οδηγία *master* καθορίζει μια περιοχή κώδικα της παράλληλης περιοχής που εκτελείται μόνο από το κύριο νήμα. Η χρήση της οδηγίας δεν υπονοεί φράγμα ούτε στην είσοδο αλλά ούτε και στην έξοδο της περιοχής[22].

2.3.2.4 Οδηγία *single*

Η οδηγία **single** εισάγεται εντός της περιοχής παράλληλου κώδικα, καθορίζει ότι το εσωκλειόμενο τμήμα στην οδηγία εκτελείται από ένα μόνο νήμα, αλλά όχι απαραίτητα από το κύριο. Τα υπόλοιπα νήματα της ομάδας, παραμένουν αδρανή στο υποκείμενο φράγμα που βρίσκεται στο τέλος της οδηγίας *single*, εκτός εάν έχει οριστεί η φράση *nowait*[9].

Συμβ. 8: Γραμματική οδηγίας single

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

Συμβ. 9: Φράσεις οδηγίας single

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

2.3.2.5 Οδηγία *flush*

Η οδηγία **flush** ή αλλιώς φράκτης μνήμης(memory fence) διασφαλίζει ότι όλα τα νήματα μιας ομάδας στην παράλληλη περιοχή είναι συγχρονισμένα σχετικά με τις τιμές που υπάρχουν σε συγκεκριμένες μεταβλητές. Η οδηγία *flush* υπονοείται στις παρακάτω περιπτώσεις[23]:

Συμβ. 10: Γραμματική οδηγίας flush

```
#pragma omp flush [memory-order-clause] [(list)] new-line
```

- *pragma omp barrier*
- Είσοδος και έξοδος στην οδηγία *omp critical*
- Έξοδος από την οδηγία *omp parallel*
- Έξοδος από την οδηγία *omp for*
- Έξοδος από την οδηγία *omp sections*
- Έξοδος από την οδηγία *omp single*

2.3.3 Φράσεις - Clauses

Δεδομένου ότι το *OpenMP* είναι ένα μοντέλο προγραμματισμού κοινής μνήμης, οι μεταβλητές είναι από προεπιλογή ορατές από όλα τα νήματα της παράλληλης περιοχής. Οι ιδιωτικές μεταβλητές χρησιμοποιούνται για να αποφευχθούν φαινόμενα *race conditions* και υπάρχει ανάγκη μεταβίβασης τους μεταξύ σειριακού κώδικα και παράλληλης περιοχής με συγκεκριμένες ιδιότητες. Η διαχείριση των δεδομένων επιτυγχάνεται με τη χρήση φράσεων **(clauses)**. Εκτός από τη διαχείριση διαμοιρασμού δεδομένων, υπάρχουν και άλλες κατηγορίες φράσεων που χρησιμοποιούνται για την διαχείριση της παραλληλοποίησης μέσω *OpenMP*. Αυτές αναφέρονται στις επόμενες παραγράφους.

2.3.3.1 Φράσεις διαμοιρασμού δεδομένων - Data sharing attribute clauses

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

2.3.3.1.1 Φράση shared

Η χρήση των μεταβλητών που δημιουργούνται εκτός της παράλληλης περιοχής, επιτρέπεται από όλα τα νήματα. Αν η μεταβλητή τροποποιηθεί από ένα νήμα, η αλλαγή θα είναι ορατή στα υπόλοιπα νήματα της ομάδας. Οι μεταβλητές με αυτό το χαρακτηριστικό διατηρούν την τελευταία τιμή τους και μετά την έξοδο από το παράλληλο τμήμα.

2.3.3.1.2 Φράση private

Τα δεδομένα που δηλώνονται εντός της φράσης είναι ιδιωτικά για κάθε νήμα. Κάθε νήμα θα έχει ένα τοπικό αντίγραφο της μεταβλητής στην ιδιωτική του μνήμη. Η ιδιωτική μεταβλητή δεν αρχικοποιείται κατά την είσοδο στη παράλληλη περιοχή που φέρει τη φράση **private** και η τελική του τιμή δεν διατηρείται για χρήση εκτός της παράλληλης περιοχής.

2.3.3.1.3 Φράση default

Δίνει τη δυνατότητα 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 *none* δηλώνει τον υποχρεωτικό ορισμό της κάθε μεταβλητής που χρησιμοποιείται μέσα στην περιοχή παράλληλου κώδικα ως shared ή private, ώστε να καθίσταται σαφές αν θα είναι ιδιωτική ή κοινόχρηστη μέσω των φράσεων διαμοιρασμού μνήμης που προαναφέρθηκαν.

2.3.3.1.4 Φράση firstprivate

Η μοναδική διαφορά της φράσης *firstprivate* από τη *private* είναι ότι στη πρώτη, η μεταβλητή αρχικοποιείται χρησιμοποιώντας την τιμή της μεταβλητής που υπάρχει με το ίδιο όνομα εκτός της παράλληλης περιοχής.

2.3.3.1.5 Φράση lastprivate

Η μοναδική διαφορά της φράσης *lastprivate* από την *private* είναι ότι σε αντίθεση με την τελευταία, η αρχική τιμή ανανεώνεται μετά το πέρας της οδηγίας στην οποία χρησιμοποιήθηκε η συγκεκριμένη φράση, με βάση τη τελευταία ενημέρωση της μεταβλητής στην αντίστοιχη σειριακή εκτέλεση. Μια μεταβλητή μπορεί να είναι δηλωμένη ταυτόχρονα και ως *firstprivate* αλλά και ως *lastprivate*.

2.3.3.1.6 Φράση threadprivate

Η φράση *threadprivate* ορίζει ότι καθολικά αντικείμενα (ή μεταβλητές) μπορούν να γίνουν προσωρινά ιδιωτικά για κάποιο νήμα. Με αυτό τον τρόπο, μπορούν να δημιουργηθούν καθολικά αντικείμενα, αλλά να μετατρέψουμε την εμβέλειά τους σε τοπική 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, εφόσον τα νήματα δεν είναι δυναμικά και ο αριθμός νημάτων στις παράλληλες περιοχές παραμένει ίδιος[5].

2.3.3.1.7 Φράση ordered

Σε περιπτώσεις παράλληλου βρόγχου επανάληψης, οι επαναλήψεις εκτελούνται με τη σειρά με την οποία θα εκτελούνταν αν ο κώδικας ήταν σειριακός.

2.3.3.1.8 Φράση reduction

Εκτελεί μία πράξη αναγωγής για κοινόχρηστες μεταβλητές. Οι μεταβλητές που βρίσκονται σε μία παράλληλη περιοχή και υπάρχουν στη λίστα της φράσης *reduction*, μεταφέρονται 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[24].

Συμβ. 11: Γραμματική φράσης reduction

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

2.3.3.2 Φράσεις συγχρονισμού - *Data sharing attribute clauses*

Σε αυτή την κατηγορία, ανήκουν οι φράσεις που χρησιμοποιούνται για τον συντονισμό των νημάτων μιας ομάδας και την αποφυγή λανθασμένων υπολογισμών που προκύπτουν από προβλήματα **race conditions** σε κοινόχρηστα δεδομένα.

2.3.3.2.1 Φράση critical

Το τμήμα κώδικα παράλληλης περιοχής που περικλείεται σε αυτή τη φράση, εκτελείται υποχρεωτικά από ένα νήμα κάθε φορά. Χρησιμοποιείται συχνά για την προστασία κοινόχρηστων δεδομένων από το *race condition* πρόβλημα.

2.3.3.2.2 Φράση atomic

Η ενημέρωση μνήμης (ανάγνωση-τροποποίηση-εγγραφή) στην οδηγία που ακολουθεί εκτελείται ατομικά. Δεν καθιστά ολόκληρη την έκφραση *atomic* άλλα μόνο τις εντολές που αφορούν ενημέρωση μνήμης. Ο μεταγλωττιστής μπορεί να χρησιμοποιεί ειδικές οδηγίες *hardware* για καλύτερη επίδοση από ό,τι όταν χρησιμοποιείται το *critical clause*.

2.3.3.2.3 Φράση barrier

Κάθε νήμα περιμένει έως ότου όλα τα άλλα νήματα μιας ομάδας φτάσουν σε αυτό το σημείο. Υπάρχουν οδηγίες, όπως αυτές που χρησιμοποιούνται για διαμοιρασμό εργασιών βρόγχου, που υπονοούν φράγμα συγχρονισμού *barrier* στο τέλος της εκτέλεσης τους.

2.3.3.2.4 Φράση nowait

Χρησιμοποιείται για να ορίσει ότι τα νήματα που ολοκληρώνουν την εργασία τους μπορούν να προχωρήσουν στην εκτέλεση εντολών της παράλληλης περιοχής, χωρίς να περιμένουν να τελειώσουν όλα τα νήματα της ομάδας. Ελλείψει αυτής της φράσης, τα νήματα συγχρονίζονται με *barrier* στο τέλος της οδηγίας.

2.3.3.2.5 Γραμματική φράσης schedule

Συμβ. 12: Φράση schedule

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

Χρησιμοποιείται στην οδηγία διαμοιρασμού εργασίας βρόγχου. Οι επαναλήψεις της οδηγίας ανατίθενται στα νήματα σύμφωνα με τον *τύπο* που ορίζεται μέσα στη φράση. • Οι τρεις τύποι *scheduling* είναι[35]:

- *static*
- *dynamic*
- *guided*

1. *static*:

Οι επαναλήψεις κατανέμονται σε κάθε νήμα πριν την εκτέλεση του βρόγχου και χωρίζονται ισόποσα σε όλα τα νήματα. Η μεταβλητή *chunk* αποτελεί έναν ακέραιο που ορίζει τον αριθμό των συνεχόμενων επαναλήψεων που θα εκτελέσει κάθε νήμα της ομάδας.

Όταν δεν ορίζεται το όρισμα *chunk*, ο αριθμός των επαναλήψεων που ορίζεται σε κάθε νήμα είναι ίσως με: $NumberOfIterations/NumberOfThreads$

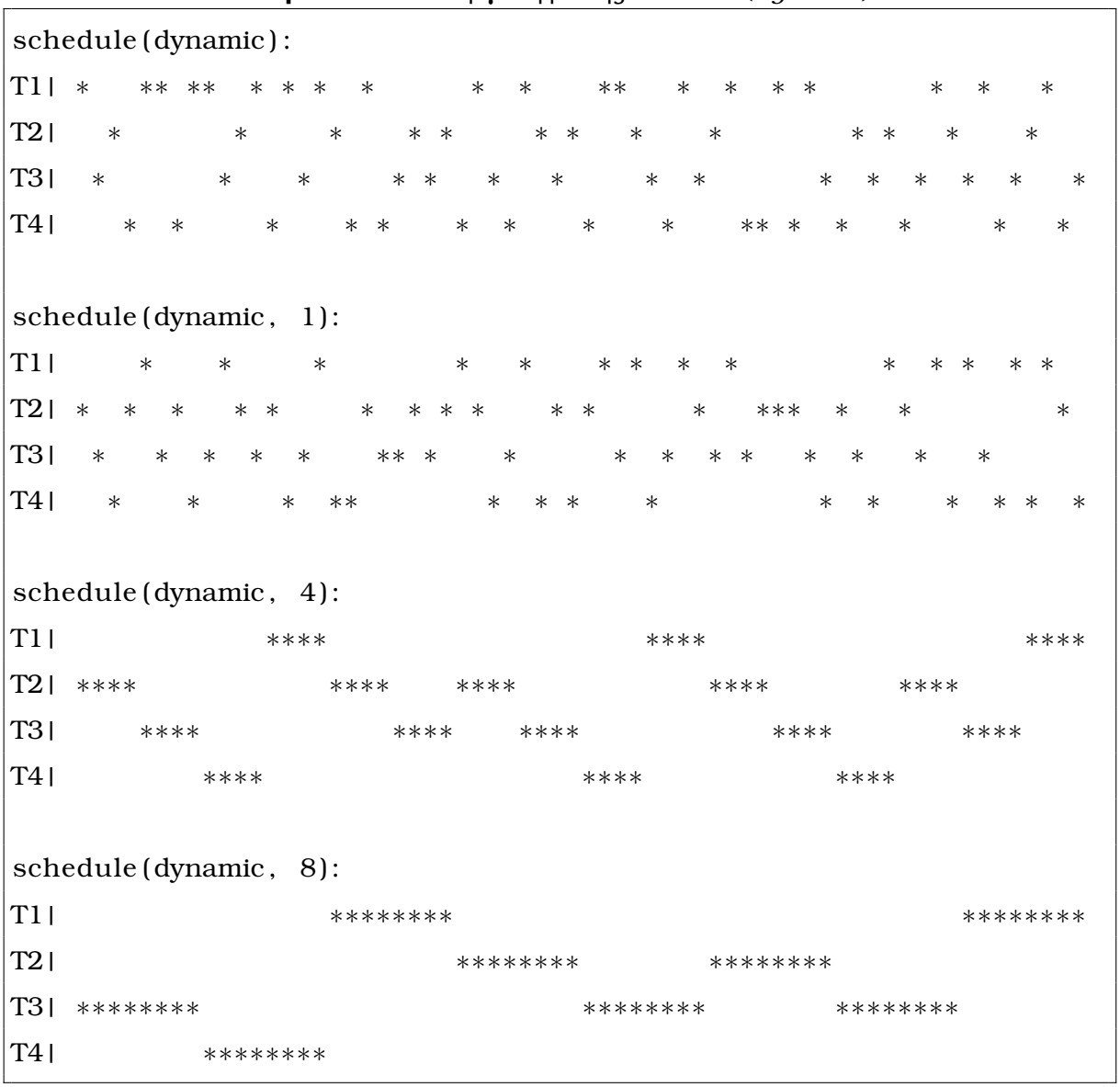
Συμβ. 13: Λειτουργία φράσης schedule(static)

```
schedule(static):
T1| ****************
T2|                 ****************
T3|                                 ****************
T4|                                                 ****************



schedule(static, 4):
T1| ****            ****            ****            ****
T2|     ****            ****            ****            ****
T3|         ****            ****            ****            ****
T4|             ****            ****            ****            ****



schedule(static, 8):
T1| ********                        ********
T2|         ********                        ********
T3|                 ********                        ********
T4|                         ********                        ********
```

2. *dynamic*:

Από το σύνολο των επαναλήψεων, ένα τμήμα κατανέμεται στα νήματα. Μόλις ένα συγκεκριμένο νήμα ολοκληρώσει την εκχωρημένη σε αυτό επανάληψη, συνεχίζει παίρνοντας μία από τις επαναλήψεις που απομένουν. Το ακέραιο όρισμα *chunk* καθορίζει τον αριθμό των συνεχόμενων επαναλήψεων που εκχωρούνται σε ένα νήμα κάθε φορά. Σε περίπτωση που δεν οριστεί αριθμός επαναλήψεων, τότε η προεπιλεγμένη τιμή είναι 1.

Συμβ. 14: Λειτουργία φράσης *schedule(dynamic)*



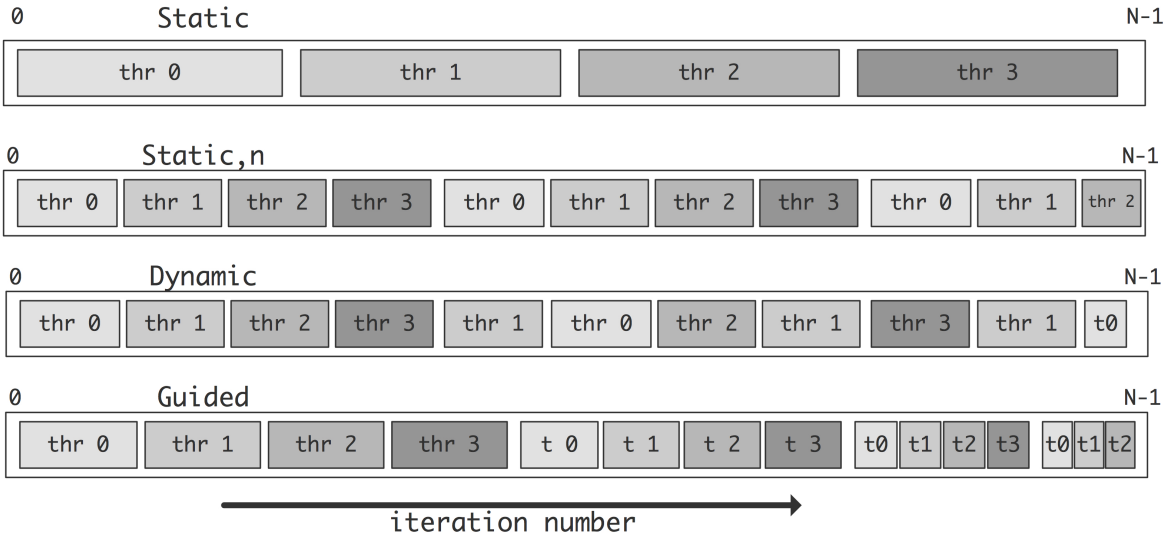
3. guided:

Ο τύπος αυτός μοιάζει με τον *dynamic*. Οι επαναλήψεις χωρίζονται σε ομάδες και κάθε νήμα εκτελεί μια ομάδα και στη συνέχεια ζητάει την επόμενη για εκτέλεση. Η διαδικασία συνεχίζεται έως ότου δεν υπάρχουν άλλες επαναλήψεις.

Η διαφορά με το δυναμικό τύπο είναι στο μέγεθος των ομάδων που διαχωρίζονται οι επαναλήψεις. Το μέγεθος είναι ανάλογο των μη εκχωρημένων επαναλήψεων διαιρούμενο με τον αριθμό των νημάτων. Επομένως το μέγεθος των ομάδων σταδιακά μειώνεται. Το ελάχιστο δυνατό μέγεθος των ομάδων ορίζεται από το όρισμα *chunk* του οποίου η προεπιλεγμένη τιμή είναι 1. Παρόλα αυτά, μόνο η τελευταία ομάδα επαναλήψεων μπορεί να είναι μικρότερη από το *chunk*.

Συμβ. 15: Λειτουργία φράσης *schedule(guided)*

```
schedule(guided):
T1|                              *********
T2|                  ************                        *******   ***
T3|                                       *******                    *
T4| ****************                                 *****       **    *

schedule(guided, 2):
T1|                  ************                        ****      **
T2|                                       *******          ***    **
T3|                              *********
T4| ****************                                 *****       **    **

schedule(guided, 4):
T1|                                       *******
T2|                  ************                        ****    ****
T3|                              *********
T4| ****************                                 *****    ****    ***

schedule(guided, 8):
T1|                  ************                     ********        ***
T2| ****************
T3|                                       ********
T4|                              *********                 ********
```



Σχήμα 6: Τύποι φράσης Schedule

2.3.4 Μεταβλητές Περιβάλλοντος

Για λόγους πληρότητας, η παράγραφος αναφέρεται επιγραμματικά στις μεταβλητές περιβάλλοντος που καθορίζουν τις ρυθμίσεις των *ICV (Internal Control Variables)* που επηρεάζουν την εκτέλεση των προγραμμάτων. Τα ονόματα των μεταβλητών περιβάλλοντος πρέπει να είναι κεφαλαία. Η τροποποίηση των μεταβλητών κατά τη διάρκεια εκτέλεσης του προγράμματος δεν είναι εφικτή και αγνοείται από το μεταγλωττιστή. Παρόλα αυτά, μερικές μπορούν να τροποποιηθούν κατά την εκτέλεση μέσω της χρήσης των αντίστοιχων οδηγιών της διεπαφής[12].

Πίνακας 1: Πίνακας μεταβλητών περιβάλλοντος.

OMP_SCHEDULE	Επιτρέπει τον καθορισμό των χαρακτηριστικών της φράσης schedule σε ένα βρόγχο επανάληψης.
OMP_DYNAMIC	Ελέγχει τη δυναμική ρύθμιση του αριθμού των νημάτων που θα χρησιμοποιηθούν για την εκτέλεση παράλληλων περιοχών
OMP_WAIT_POLICY	Παρέχει μια υπόδειξη σχετικά με την επιθυμητή συμπεριφορά των νημάτων που βρίσκονται σε αναμονή
OMP_THREAD_LIMIT	Ορίζει τον μέγιστο αριθμό νημάτων για χρήση
OMP_CANCELLATION	Ενεργοποιεί ή απενεργοποιεί το μοντέλο ακύρωσης
OMP_DISPLAY_AFFINITY	Εμφανίζει πληροφορίες συγγένειας(affinity) για όλα τα νήματα κατά την είσοδο στην πρώτη παράλληλη περιοχή.
OMP_DEFAULT_DEVICE	Ορίζει την συσκευή που θα χρησιμοποιηθεί ως συσκευή προορισμού
OMP_TARGET_OFFLOAD	Καθορίζει τη συμπεριφορά του προγράμματος, όταν η εκτέλεση του μεταφέρεται στη συσκευή στόχου.
OMP_TOOL_LIBRARIES	Ορίζει τις βιβλιοθήκες που χρησιμοποιούνται σε μια συσκευή που εκτελεί πρόγραμμα με OpenMP
OMP_ALLOCATORS	Καθορίζει τον επιλεγμένο εκχωρητή μνήμης
OMP_AFFINITY_FORMAT	Καθορίζει τη μορφή εμφάνισης πληροφοριών σχετικά με τη συγγένεια νημάτων.
OMP_MAX_TASK_PRIORITY	Ορίζει το μέγιστο αριθμό προτεραιότητας που μπορεί να έχει μια εργασία(Task)
OMP_TOOL	Ελέγχει αν ένα πρόγραμμα θα μπορεί να καταχωρήσει ένα εργαλείο third party κατά τη διάρκεια εκτέλεσης
OMP_DEBUG	Ενεργοποιεί την εμφάνιση πληροφοριών που εξάγονται από μια OpenMP βιβλιοθήκη.
OMP_PLACES	Χρησιμοποιείται για να καθορίσει πως τα νήματα σε ένα πρόγραμμα που εκτελείται με OpenMP συνδέονται με τους επεξεργαστές.
OMP_DISPLAY_ENV	Δίνει εντολή κατά τη διάρκεια εκτέλεσης να εμφανίσει τις πληροφορίες όπως αυτές περιγράφονται από τη ρουτίνα omp_display_env
OMP_MAX_ACTIVE_LEVELS	Ορίζει τον μέγιστο αριθμό ένθετων παράλληλων περιοχών
OMP_NESTED	Ενεργοποιεί Απενεργοποιεί τον ένθετο παραλληλισμό
OMP_PROC_BIND	Καθορίζει αν τα νήματα μπορούν να μετακινηθούν μεταξύ επεξεργαστών
OMP_NUM_THREADS	Καθορίζει τον αριθμό των νημάτων που θα χρησιμοποιηθούν στην παράλληλη περιοχή

2.3.5 *Runtime Functions*

Σκοπός της παρούσας παραγράφου είναι η συνοπτική αναφορά των ενσωματωμένων ρουτινών της βιβλιοθήκης *OpenMP*. Η κεφαλίδα **<omp.h>**, περιλαμβάνει ένα σύνολο ρουτινών που χωρίζονται σε τρεις μεγάλες κατηγορίες[7]:

- Τις ρουτίνες ελέγχου περιβάλλοντος εκτέλεσης που χρησιμοποιούνται για τον συντονισμό των νημάτων και της εκτέλεσης του παράλληλου τμήματος
- Τις ρουτίνες κλειδώματος των νημάτων που χρησιμοποιούνται για τον συγχρονισμό της πρόσβασης στα δεδομένα
- Τις ρουτίνες καταμέτρησης χρόνου

Πίνακας 2: Πίνακας ενσωματωμένων ρουτινών(1/2)

Όνομα	**Σύντομη περιγραφή**
omp_get_wtime	Επιστρέφει το χρόνο που έχει παρέλθει σε δευτερόλεπτα.
omp_get_wtick	Επιστρέφει την ακρίβεια του χρονοδιακόπτη που χρησιμοποιείται από την omp_get_wtime, δηλαδή τον αριθμό των δευτερολέπτων μεταξύ δύο διαδοχικών χτύπων του ρολογιού.

Πίνακας 3: Πίνακας ενσωματωμένων ρουτινών(2/2)

Όνομα	**Σύντομη περιγραφή**
omp_set_num_threads *omp_get_num_threads*	Ορίζει/Επιστρέφει τον αριθμό των νημάτων που θα χρησιμοποιηθούν για τις επόμενες παράλληλες περιοχές στις οποίες δεν ορίζεται η φράση num_threads
omp_get_max_threads	Επιστρέφει το μέγιστο αριθμό νημάτων που μπορούν να χρησιμοποιηθούν για τη δημιουργία μιας ομάδας.
omp_get_thread_num	Επιστρέφει τον αριθμό του νήματος στη τρέχουσα ομάδα, που καλεί την ρουτίνα.
omp_get_num_procs	Επιστρέφει τον αριθμό των επεξεργαστών που διατίθενται στη συσκευή.
omp_in_parallel	Επιστρέφει αληθές, αν η εργασία βρίσκεται εντός παράλληλης περιοχής.
omp_set_dynamic *omp_get_dynamic*	Διαχειρίζονται τη δυναμική ρύθμιση του αριθμού των διαθέσιμων νημάτων για την εκτέλεση των επόμενων παράλληλων περιοχών.
omp_set_nested	Επιτρέπει ή όχι ένθετο παραλληλισμό, δηλαδή δημιουργία νημάτων μέσα σε νήματα.
omp_init_lock *omp_destroy_lock*	Αρχικοποιεί/Καταστρέφει το κλείδωμα.
omp_set_lock *omp_unset_lock*	Παρέχουν ένα μέσο ρύθμισης ενός OpenMP κλειδώματος. Η εργασία συμπεριφέρεται σαν να είχε τεθεί σε αναστολή έως ότου το κλείδωμα μπορεί να ρυθμιστεί από αυτή την εργασία.
omp_test_lock	Προσπαθεί να ορίσει ένα κλείδωμα, χωρίς όμως να αποκλείσει την εκτέλεση του νήματος.
omp_init_nest_lock *omp_destroy_nest_lock* *omp_set_nest_lock* *omp_unset_nest_lock*	Αντίστοιχες ρουτίνες που αναφέρονται σε ένθετα κλειδώματα.

3 Χαρακτηριστικά εκδόσεων OpenMP 3.0 - 4.5

Η έκδοση του *OpenMP* που ακολούθησε της 2.5, ήταν η 3.0. Η νέα έκδοση περιείχε σημαντικές προσθήκες και αλλαγές για τα δεδομένα του παράλληλου προγραμματισμού. Χαρακτηριστικά προηγούμενων εκδόσεων αναβαθμίστηκαν, ωστόσο τη σημαντικότερη αλλαγή αποτέλεσε η εισαγωγή της έννοιας των εργασιών (**Tasking**). Η επόμενη έκδοση του *OpenMP* (4.0) κυκλοφόρησε τον Ιούλιο του 2013 και περιελάμβανε τα παρακάτω νέα χαρακτηριστικά:

- *Threads affinity*,
- Ετερογενής προγραμματισμός,
- Διαχείριση σφαλμάτων,
- Διανυσματικοποίηση μέσω *SIMD*,
- *User-Defined Recuctions (UDRs)*

Τα τελευταία χρόνια η ανάγκη για εφαρμογές κατασκευασμένες με υψηλά επίπεδα παραλληλισμού είναι αυξημένη. Κύρια αιτία αποτελεί η ευκολία πρόσβασης σε μεγάλο όγκο δεδομένων από το ευρύ κοινό, οι μικρές εξαρτήσεις ανάμεσά στα δεδομένα, αλλά και η ανάγκη για εντατικούς υπολογισμούς στα δεδομένα αυτά. Λύση στο πρόβλημα αυξημένου φόρτου υπολογισμών αποτελεί η χρήση ετερογενών συστημάτων προγραμματισμού. Ωστόσο, παρά τα οφέλη της υλοποίησης και χρήσης τέτοιων συστημάτων σε ότι αφορά την απόδοση, ο προγραμματισμός εφαρμογών σε αυτά, δρα ανασταλτικά για την ευρεία χρήση τους. Το *OpenMP* δημιούργησε τα κατάλληλα εργαλεία για την εξάλειψη δυσκολιών υλοποίησης. Με τη δημοσίευση της έκδοσης 4.0, προσέφερε την υποδομή για την υποστήριξη ετερογενών συστημάτων, καθώς η έκδοση περιλαμβάνει ένα σύνολο οδηγιών και φράσεων τα οποία χρησιμοποιούνται για τον προσδιορισμό ρουτινών και δεδομένων, ικανών να μετακινηθούν σε μια συσκευή στόχου (επιταχυντή) για να υπολογιστούν. Στόχος είναι η αύξηση των επιδόσεων υπολογισμού αλλά και η μείωση της κατανάλωσης ισχύος.

Εκτός από την υποστήριξη ετερογενών συστημάτων, το *OpenMP* υποστηρίζει την επεξεργασία δεδομένων μέσω διανυσματικοποίησης *SIMD*. Η επεξεργασία *SIMD (Simple Instruction Multiple Data)* εκμεταλλεύεται τον παραλληλισμό σε επίπεδο δεδομένων,

πράγμα που σημαίνει ότι οι πράξεις που γίνονται σε ένα σύνολο μιας συστοιχίας γίνονται ταυτόχρονα μέσω απλών εντολών.

Στις ενότητες του κεφαλαίου που ακολουθούν, εκτός από την αναλυτικότερη περιγραφή των παραπάνω βασικών χαρακτηριστικών, γίνεται περιγραφή της έννοιας του *Thread Affinity*, των *User-Defined Reductions* και των εργασιών*(Tasking)*.

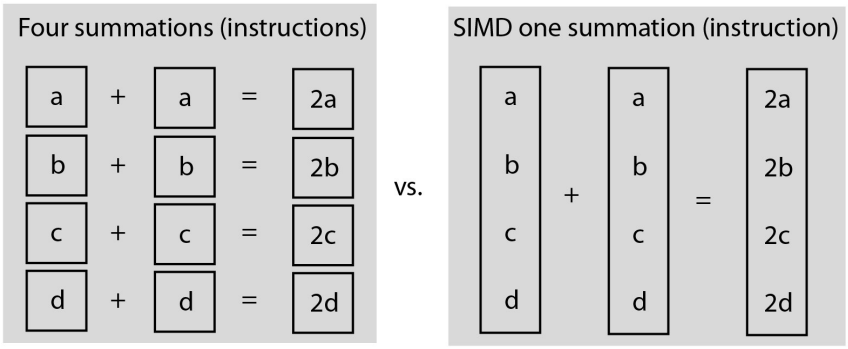
3.1 Διανυσματικοποίηση μέσω *SIMD*

Κατά την διάρκεια εκτέλεσης ενός τυπικού προγράμματος από έναν μη παράλληλο υπολογιστή, οι εντολές που εκτελούνται είναι απλές, εφαρμοζόμενες σε απλά, μοναδιαία δεδομένα. Το μοντέλο αυτό ονομάζεται (*SISD - Single Instruction Single Data*) και αποτελούσε για πολλά χρόνια το επικρατέστερο μοντέλο υλοποίησης και εκτέλεσης προγραμμάτων. Αποτελεί ένα από τα τέσσερα μοντέλα της ταξινόμησης *Flynn* που προτάθηκε το 1966[20]. Τα άλλα τρία μοντέλα είναι:

- *SIMD - Single Instruction Multiple Data*
- *MISD - Multiple Instruction Single Data*
- *MIMD - Multiple Instructions Multiple Data*

Στο μοντέλο *Single Instruction Multiple Data (SIMD)*, εκτελούνται απλές λειτουργίες για την διαχείριση ενός συνόλου δεδομένων τοποθετημένων στη σειρά. Το σύνολο αυτών των δεδομένων ονομάζεται πίνακας.

Στο παρακάτω σχήμα παρουσιάζεται η πρόσθεση των στοιχείων δυο πινάκων και η εκχώρηση των αποτελεσμάτων σε ένα τρίτο, με τον συμβατικό τρόπο και με την χρήση *SIMD* μεθόδου. Το πλεονέκτημα της δεύτερης μεθόδου είναι ότι οι *SIMD* οδηγίες εκτελούνται το ίδιο γρήγορα με τις αντίστοιχες βαθμωτές. Με άλλα λόγια η πράξη του σχήματος, θα γίνει 4 φορές πιο γρήγορα στη δεύτερη περίπτωση.



Σχήμα 7: Πρόσθεση πινάκων βαθμωτά και με διανυσματικοποίηση

Στο κεφάλαιο αυτό, περιγράφονται τα χαρακτηριστικά και οι δυνατότητες που είναι διαθέσιμες στο *OpenMP* και αφορούν λειτουργίες διανυσματικοποίησης μέσω *SIMD*.

3.1.1 Η οδηγία *simd*

Αν ο μεταγλωττιστής δεν εφαρμόζει διανυσματικοποίηση ή δεν χρησιμοποιείται κάποια ειδική βιβλιοθήκη, τότε η επόμενη καλύτερη μέθοδος διανυσματικοποίησης είναι με τη χρήση του *OpenMP*. Η διεπαφή εφοδιάζει το χρήστη με ένα σύνολο οδηγιών και φράσεων που σκοπό έχουν να ενημερώσουν το μεταγλωττιστή, να εκτελέσει παράλληλα ή με διανυσματικοποίηση βρόγχους επανάληψης.

Βασικότερη οδηγία της διεπαφής αποτελεί η ***simd***. Εφαρμόζεται σε βρόγχους των οποίων η δομή είναι ίδια με την δομή των κοινών βρόγχων της *C++*. Η εισαγωγή της οδηγίας *simd* δίνει εντολή στο μεταγλωττιστή να δημιουργήσει έναν *simd* βρόγχο.

Ιδιαίτερη προσοχή απαιτείται σε περιπτώσεις χρήσης μεταβλητών όπως δείκτες μέσα στο βρόγχο. Η λανθασμένη χρήση τους μπορεί να προκαλέσει απροσδιόριστη συμπεριφορά. Για παράδειγμα, στο παρακάτω τμήμα κώδικα, αν ο δείκτης *k* ή *m* είναι ταυτόσημος με τον δείκτη t, τότε αναμένονται λάθος αποτελέσματα.

Συμβ. 16: Εφαρμογή οδηγίας 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 παράδειγμα, η μεταβλητή *i* είναι ιδιωτική. Σε ένα τσιπ υπάρχουν πολλοί πυρήνες που μπορούν να τρέξουν ταυτόχρονα. Μέσα σε έναν πυρήνα, υπάρχουν πολλά νήματα που τρέχουν ταυτόχρονα, και μέσα σε ένα νήμα υπάρχουν λωρίδες SIMD. Η διαφορά με την ιδιωτική μεταβλητή ενός παράλληλου βρόγχου είναι ότι η ιδιωτικότητα αναφέρεται στη λωρίδα *SIMD - (SIMD lane)*. Οι τιμές των πινάκων *t, k, m* είναι κοινόχρηστες. Το καταλληλότερο μέγεθος πίνακα για την διανυσματικοποίηση, εξαρτάται από την αρχιτεκτονική του μηχανήματος και επιλέγεται από αυτό.

3.1.2 Φράσεις οδηγίας *simd*

Ένα σύνολο φράσεων ακολουθούμενων της οδηγίας *simd* υποστηρίζεται από το *OpenMP*. Οι φράσεις *private, lastprivate, reduction, collapse, ordered*, έχουν την ίδια χρησιμότητα που προαναφέρθηκε στην οδηγίες των προηγούμενων κεφαλαίων(πχ οδηγία διαμοιρασμού βρόγχου). Το σύνολο των φράσεων που υποστηρίζονται από την οδηγία εμφανίζονται παρακάτω:

Συμβ. 17: Φράσεις οδηγίας *simd*

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

3.1.2.1 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 πίνακα που χρησιμοποιείται από τις παραγόμενες *simd* οδηγίες.

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

3.1.2.2 Φράση safelen

Η φράση *safelen* δέχεται ως όρισμα ένα θετικό ακέραιο αριθμό. Η τιμή αυτή καθορίζει το ανώτατο όριο του μεγέθους πίνακα. Είναι το μήκος το οποίο είναι ασφαλές για τον βρόγχο. Το τελικό μέγεθος πίνακα επιλέγεται από τον μεταγλωττιστή, αλλά δεν υπερβαίνει ποτέ την τιμή της φράσης *safelen*.

Στο παρακάτω παράδειγμα απαιτείται η φράση *safelen*. Πρόκειται για έναν βρόγχο που περιέχει εξάρτηση(dependence) στην προσπέλαση των στοιχείων του πίνακα. Πιο συγκεκριμένα, το διάβασμα του *[i-10]* στην επανάληψη *i* δεν μπορεί να υλοποιηθεί, αν δεν ολοκληρωθεί η εγγραφή στο *k[i]* της προηγούμενης επανάληψης.

Συμβ. 18: Παράδειγμα κώδικα με simd

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

3.1.2.3 Φράση linear

Όταν χρησιμοποιείται σε περιβάλλον βρόχου SIMD, η φράση linear εκτελεί γραμμική αύξηση σε μια μεταβλητή χρησιμοποιώντας SIMD οδηγίες. Η φράση linear παίρνει μια ακέραια μεταβλητή και προσθέτει το γραμμικό βήμα στη μεταβλητή σε κάθε επανάληψη του βρόγχου. Η διαδικασία εξελίσσεται με τη δημιουργία δύο ιδιωτικών πινάκων εντός του βρόχου SIMD. Ο πρώτος πίνακας περιέχει τη γραμμική ακολουθία που δημιουργήθηκε προσθέτοντας την τιμή βήματος στην αρχική τιμή της μεταβλητής. Ο δεύτερος πίνακας περιέχει το γραμμικό βήμα που αυξάνει τον προηγούμενο πίνακα. Για παράδειγμα, εάν η φράση έχει μια γραμμική μεταβλητή N = 1 με ένα γραμμικό βήμα 2 και χρησιμοποιείται έναν πίνακα τεσσάρων θέσεων, τότε ο πρώτος ιδιωτικός πίνακας θα περιείχε 1, 3, 5, 7. Μετά από μια άλλη επανάληψη σε βρόχο SIMD, ο πίνακας θα ήταν 9, 11, 13, 15. Αυτό γίνεται με την προσθήκη ενός διανύσματος που περιέχει 8, 8, 8, 8 μετά από κάθε SIMD lane[2].

3.1.2.4 Φράση aligned

Η ευθυγράμμιση των δεδομένων είναι σημαντική για την καλή απόδοση του προγράμματος. Αν ένα στοιχείο πίνακα δεν είναι ευθυγραμμισμένο σε διεύθυνση μνήμης που είναι πολλαπλάσιο του μεγέθους του στοιχείου σε *byte*, προκύπτει ένα επιπλέον κόστος καθυστέρησης για την τροποποίηση του στοιχείου αυτού.

Για παράδειγμα, σε ορισμένες αρχιτεκτονικές ενδέχεται να μην είναι δυνατή η φόρτωση και εγγραφή από μια διεύθυνση μνήμης που δεν είναι ευθυγραμμισμένη με το μέγεθος του δεδομένου που χρησιμοποιείται. Έτσι, οι λειτουργίες γίνονται κανονικά, αλλά με μεγαλύτερο κόστος. Σε περίπτωση διανυσματικοποίησης μέσω της οδηγίας *simd*, η επιλογή 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 τιμή της ευθυγράμμισης πρέπει να είναι ένας σταθερός ακέραιος αριθμός. Σε περίπτωση έλλειψης της φράσης, μια προεπιλεγμένη τιμή καθορίζεται από την υλοποίηση.

3.1.3 Η σύνθετη οδηγία βρόγχου *SIMD*

Η σύνθετη οδηγία βρόγχου *SIMD*, συνδυάζει παραλληλισμό νημάτων και διανυσματικοποίηση μέσω *SIMD*.

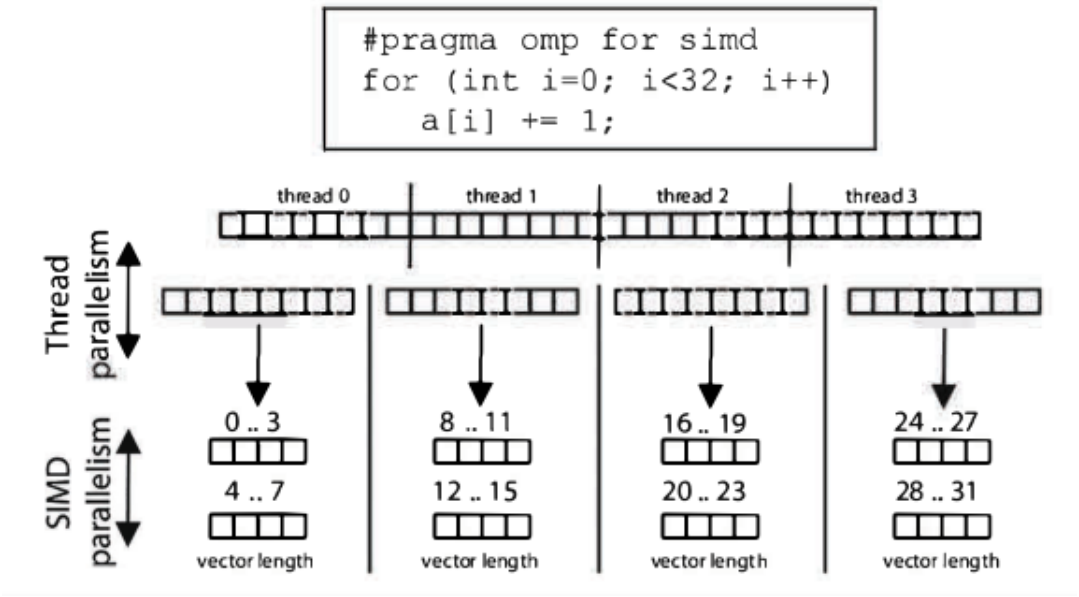
Συμβ. 19: Γραμματική οδηγίας omp for simd

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

Οι επαναλήψεις ενός βρόγχου διαιρούνται σε τμήματα και διανέμονται σε μια ομάδα νημάτων. Στη συνέχεια, τα τμήματα αυτά εκτελούνται βάση της οδηγίας επαναληπτικού *simd* βρόγχου. Οι φράσεις που μπορούν να χρησιμοποιηθούν στην οδηγία διαμοιρασμού βρόγχου ή στην οδηγία *simd* μπορούν να χρησιμοποιηθούν και σε αυτές τις σύνθετες οδηγίες.

Στη σύνθετη οδηγία, τμήματα επαναλήψεων του βρόγχου διανέμονται στην ομάδα νημάτων με τη μέθοδο που ορίζουν οι φράσεις διαμοιρασμού που χρησιμοποιούνται στην

οδηγία διαμοιρασμού επαναλήψεων. Στη συνέχεια, τα τμήματα επαναλήψεων βρόγχου μετατρέπονται σε οδηγίες *simd* με μέθοδο που ορίζεται από τις φράσεις για την οδηγία *simd*. Τα παραπάνω φαίνονται σχηματικά στην επόμενη εικόνα:



Σχήμα 8: Βήματα εργασιών οδηγίας *for simd*

Κάθε νήμα ενός επαναληπτικού βρόγχου έχει ένα συγκεκριμένο ποσοστό εργασίας που πρέπει να εκτελέσει. Αν ο αριθμός των νημάτων αυξηθεί, το ποσοστό της εργασίας ανά νήμα θα μειωθεί. Προσθέτοντας παραλληλισμό *simd*, δεν είναι απαραίτητη η βελτίωση της απόδοσης, ειδικά στις περιπτώσεις που ο επαναληπτικός *simd* βρόγχος που ανήκει σε ένα νήμα μειώσει το μήκος του.

3.1.4 Συναρτήσεις *SIMD*

Οι κλήσεις συναρτήσεων εντός της περιοχής βρόγχου *SIMD* εμποδίζουν τη δημιουργία αποτελεσματικών *SIMD* δομών. Στη χειρότερη περίπτωση η κλήση της συνάρτησης θα γίνει χρησιμοποιώντας βαθμωτά δεδομένα, κάτι που πιθανότατα θα επηρεάσει αρνητικά την αποτελεσματικότητα.

Για την πλήρη εκμετάλλευση του παραλληλισμού με διανυσματικοποίηση, για κάθε συνάρτηση που καλείται μέσα από ένα βρόγχο *SIMD*, πρέπει να ορίζεται μια ισοδύναμη έκδοσή της για εκτέλεση μέσα σε βρόγχους *SIMD*. Έτσι, ο μεταγλωττιστής θα δημιουργήσει αυτή την ειδική έκδοση της συνάρτησης με *SIMD* παραμέτρους και οδηγίες.

Η οδηγία *declare simd* χρησιμοποιείται για να δημιουργήσει ο μεταγλωττιστής μια ή περισσότερες εκδόσεις μιας συνάρτησης. Οι εκδόσεις αυτές χρησιμοποιούνται αποκλειστικά από βρόγχους *SIMD*.

3.1.4.1 Η οδηγία *declare simd*

Η οδηγία *declare simd* χρησιμοποιείται για να ενημερωθεί ο μεταγλωττιστής ότι μια συνάρτηση μπορεί να χρησιμοποιηθεί από μια περιοχή παραλληλισμού *simd*.

Συμβ. 20: Γραμματική οδηγίας *declare simd*

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

Συμβ. 21: Φράσεις οδηγίας *declare simd*

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

Ο μεταγλωττιστής μπορεί να δημιουργήσει πολλές *simd* εκδόσεις μιας συνάρτησης και να επιλέξει την κατάλληλη για να κληθεί στο *simd* τμήμα. Από τη μεριά του, ο χρήστης μπορεί να προσαρμόσει την λειτουργία μιας συνάρτησης με τη χρήση εξειδικευμένων φράσεων.

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

3.1.5 Χαρακτηριστικά παραμέτρων *SIMD* συναρτήσεων

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

Όταν μια παράμετρος εμπεριέχεται στη φράση *uniform*, η τιμή της παραμέτρου είναι σταθερή για όλες τις ταυτόχρονες κλήσεις κατά την εκτέλεση της οδηγίας *simd* βρόχου. Η φράση **uniform(arg1, arg2)** δίνει την οδηγία στο μεταγλωττιστή να δημιουργήσει μια *simd* συνάρτηση που προϋποθέτει ότι αυτές οι δύο μεταβλητές είναι ανεξάρτητες από τον βρόγχο.

Συμβ. 22: Χρήση φράσεων uniform - simdlen

```
#pragma omp declare simd simdlen(16) uniform(a, b, offset)
float do_work(float *a, float *b, int i, int offset)
{
        return a[i] ? a[i] + b[i] : b[i + offdset];
}
void fun(float *a, float *b, int offset, int len)
{
#pragma omp simd  aligned(a:64, b:64)
        for(int i = 0; i < len; i++)
        {
                a[i] = do_work(a, b, i, off);
        }
}
```

Η φράση *linear* ορίζει ότι ένα όρισμα που μεταβιβάζεται σε μια συνάρτηση έχει γραμμική σχέση μεταξύ των παράλληλων επικλήσεων μιας συνάρτησης. Κάθε *simd lane* παρατηρεί την τιμή του ορίσματος στην πρώτη λωρίδα και προσθέτει την μετατόπιση της *simd* λωρίδας από την πρώτη, επί το γραμμικό βήμα.

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

Συμβ. 23: Χρήση φράσεων uniform - simdlen

```
#pragma omp declare simd simdlen(16) uniform(a, b, offset)
float do_work(float *a, float *b, int i, int offset)
{
        return a[i] ? a[i] + b[i] : b[i + offdset];
}
void fun(float *a, float *b, int offset, int len)
{
#pragma omp simd  aligned(a:64, b:64)
        for(int i = 0; i < len; i++)
        {
                a[i] = do_work(a, b, i, off);
        }
}
```

Η φράση aligned δηλώνει ότι το αντικείμενο στο οποίο κάθε σημείο της λίστας δείχνει, είναι ευθυγραμμισμένο με τον αριθμό των byte που δηλώνονται στην φράση. Ο αριθμός της πρέπει να είναι μια θετική ακέραια έκφραση.

Η φράση *inbranch* χρησιμοποιείται όταν μια συνάρτηση καλείται πάντα μέσα σε ένα *simd* βρόγχο και 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.

Αντίθετα, 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. 24: Χρήση 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 η έννοια που περιλαμβάνει την βελτιστοποίηση του χρόνου εκτέλεσης ενός προγράμματος, μέσω βελτιστοποιήσεων στο εύρος ζώνης μνήμης, την αποφυγή καθυστέρησης μνήμης ή της καθυστέρησης χρήσης προσωρινής μνήμης.

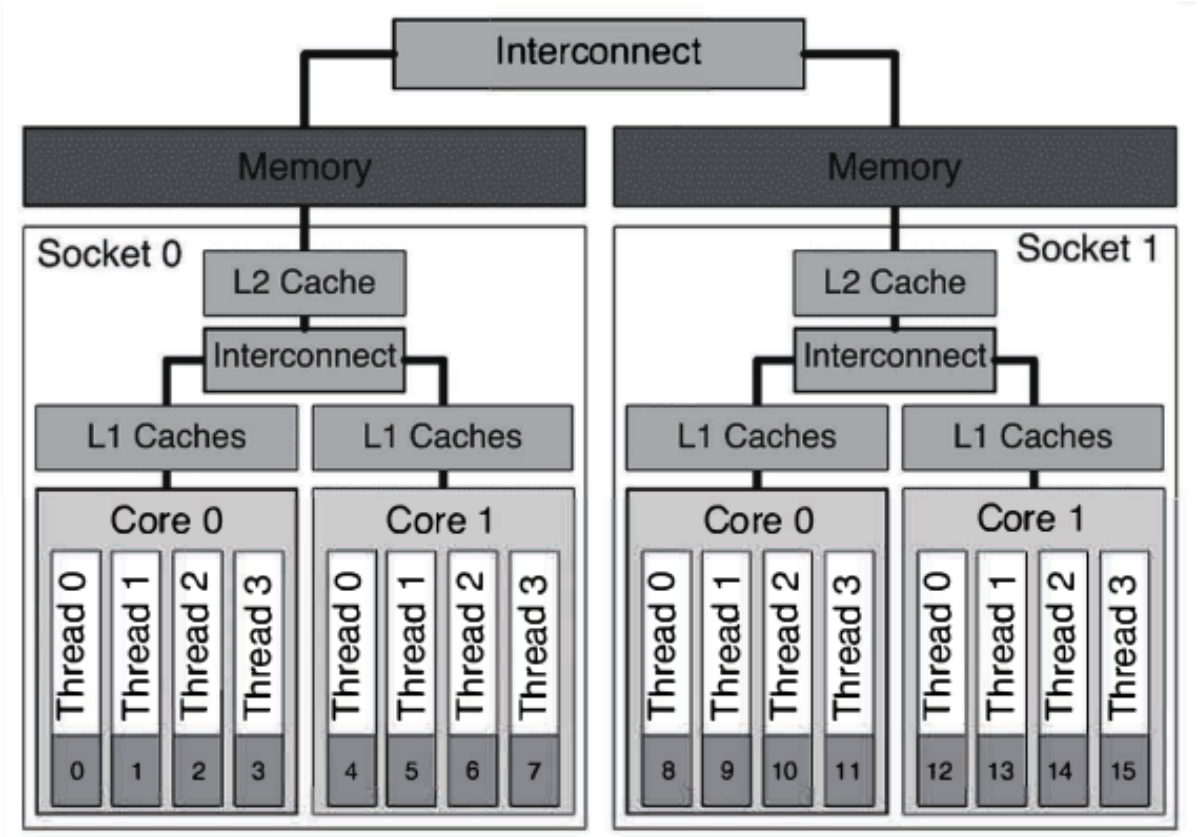
Το *OpenMP 4.0* εισάγει ένα σύνολο οδηγιών για την υποστήριξη του *thread affinity*[11]. Η πλειοψηφία πλέον των μηχανημάτων βασίζονται στην *cc-NUMA* αρχιτεκτονική. Η *cc-NUMA* αρχιτεκτονική συντίθεται από δυο ειδικότερες έννοιες:

- **Συνοχή Μνήμης(Cache Coherence)**: Στη σύγχρονα πολυπύρηνα συστήματα, σε κάθε πυρήνα εννοείται η ύπαρξη μνήμης Cache που έχει ως στόχο τη μείωση της κυκλοφορίας στο δίαυλο μνήμης. Ο όρος συνοχή σημαίνει οτι όταν ένας επεξεργαστής ενημερώνει μια θέση της κρυφής μτου μνήμης, τότε όλοι οι άλλοι επεξεργαστές και η κύρια μνήμη, ενημερώνονται έγκαιρα για την τροποποίηση αυτή[;].
- **NUMA - Non Uniform Memory Access**: Σε ένα πολυπύρηνο σύστημα επεξεργαστών, ένας επεξεργαστής έχει τη δυνατότητα προσπέλασης της μνήμης του άλλου, αλλά ο απαιτούμενος χρόνος προσπέλασής της δεν είναι ίσος με αυτό της τοπικής του μνήμης[24].

Ο λόγος που κυριάρχησε το σύστημα μνήμης, 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 ενιαίας μνήμης στον χρήστη. Ένα 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, πλέον ο χρόνος πρόσβασης σε αυτά δεν είναι σταθερός καθώς εξαρτάται από τη θέση τους στη μνήμη.



Σχήμα 9: Αρχιτεκτονική cc-NUMA[32]

3.2.1 *Thread affinity στο OpenMP 4.0*

Με το *thread affinity* μπορεί να αποτραπεί η μετάβαση μιας διαδικασίας *MPI* ή ενός νήματος του *OpenMP* σε απομακρυσμένο υλικό. Η μετάβαση αυτή θα μπορούσε να προκαλέσει μείωση της απόδοσης του προγράμματος. Η έκδοση 4.0 του *OpenMP* εισήγαγε ρυθμίσεις για το χειρισμό του *affinity* μέσω των μεταβλητών περιβάλλοντος *OMP_PLACES* και *OMP_PROC_BIND*[26].

3.2.1.1 *Thread Binding*

Οι προαναφερθείσες μεταβλητές, μπορούν να καθορίσουν σε ποια θέση στο υλικό θα ανατεθούν τα νήματα μια ομάδας που δημιουργήθηκε για να εκτελέσει μια εργασία.

Παράδειγμα: Αν υπάρχουν δύο *sockets* και ορισθεί

$$OMP\_PLACES = sockets$$

τότε:

- το νήμα 0 θα πάει στο *socket 0*
- το νήμα 1 θα πάει στο *socket 1*
- το νήμα 2 θα πάει στο *socket 0* κοκ.

Επίσης, αν δυο *socket* έχουν συνολικά 16 πυρήνες και ο χρήστης ορίσει

$$OMP\_PLACES = cores$$

και

$$OMP\_PROC\_BIND = close$$

τότε:

- το νήμα 0 θα πάει στον πυρήνα 0 που βρίσκεται στο *socket* 0
- το νήμα 1 θα πάει στον πυρήνα 1 που βρίσκεται στο *socket* 0
- το νήμα 2 θα πάει στον πυρήνα 2 που βρίσκεται στο *socket* 0
- ...
- το νήμα 7 στον πυρήνα 7 του *socket* 0
- το νήμα 8 στον πυρήνα 8 του *socket* 1, κλπ

Η μεταβλητή *OMP_PROC_BIND* καθορίζει τον τρόπο με τον οποίο τα νήματα ανατίθενται στους πόρους. Η επιλογή *OMP_PROC_BIND = close* σημαίνει ότι η ανάθεση περνά διαδοχικά στις διαθέσιμες θέσεις. Μια άλλη αποδεκτή τιμή για το *OMP_PROC_BIND* είναι η *spread*. Η λειτουργία της φαίνεται στο παρακάτω παράδειγμα:

Παράδειγμα, για:

$$OMP\_PLACES = cores$$

$$OMP\_PROC\_BIND = spread$$

- το νήμα 0 πάει στον πυρήνα 0, που βρίσκεται στο *socket* 0
- το νήμα 1 πάει στον πυρήνα 8, που βρίσκεται στο *socket* 1
- το νήμα 2 πάει στον πυρήνα 1, που βρίσκεται στο *socket* 0
- ...
- το νήμα 15 πάει στον πυρήνα 15, που βρίσκεται στο *socket* 1

Η επιλογή *OMP_PROC_BIND* = *master* αναθέτει τα νήματα στο ίδιο σημείο που είναι και το κύριο νήμα της ομάδας. Αυτή η επιλογή χρησιμοποιείται όταν δημιουργούνται πολλές ομάδες αναδρομικά.

Εκτός από τις επιλογές *cores* και *sockets* για τη μεταβλητή *OMP_PLACES*, υπάρχει και η *threads* που χρησιμοποιείται σε ειδικές περιπτώσεις αρχιτεκτονικής, δηλαδή σε περιπτώσεις που οι επεξεργαστές περιέχουν νήματα(hardware threads)[18].

Το παράδειγμα που ακολουθεί, επιβεβαιώνει την προαναφερθείσα θεωρία. Στην παρακάτω ρουτίνα, κάθε νήμα αναλαμβάνει να εκτυπώσει ενα μήνυμα. Το μήνυμα αναφέρει σε ποιον πυρήνα ανήκει το νήμα αυτό[3].

Συμβ. 25: Παράδειγμα Affinity

```
int main( int argc, char**argv )
{
#pragma omp parallel
    {
#pragma omp critical
        std::cout << "Running from thread " <<
            omp_get_thread_num() + 1 <<
            " of " << omp_get_num_threads() <<
            " on cpu " << sched_getcpu() <<
            std::endl;
    }
    return 0;
 }
```

Συμβ. 26: Affinity: Αποτελέσματα εκτέλεσης

```
Running from thread 15 of 16 on cpu 12
Running from thread 3  of 16 on cpu 11
Running from thread 1  of 16 on cpu 15
Running from thread 6  of 16 on cpu 8
Running from thread 12 of 16 on cpu 10
Running from thread 9  of 16 on cpu 9
Running from thread 8  of 16 on cpu 6
Running from thread 14 of 16 on cpu 14
Running from thread 2  of 16 on cpu 3
Running from thread 7  of 16 on cpu 0
Running from thread 10 of 16 on cpu 1
Running from thread 13 of 16 on cpu 2
Running from thread 16 of 16 on cpu 13
Running from thread 11 of 16 on cpu 7
Running from thread 5  of 16 on cpu 5
Running from thread 4  of 16 on cpu 4
```

Εκτελώντας το ίδιο πρόβλημα αφού πρώτα χρησιμοποιηθεί η εντολή $OMP\_PLACES = "\{0\}"$ τα αποτελέσματα δείχνουν ότι όλα τα νήματα δημιουργούνται απο τον πυρήνα με id 0:

Συμβ. 27: Affinity: Αποτελέσματα εκτέλεσης $OMP\_PLACES = "\{0\}"$

```
Running from thread 1  of 16 on cpu 0
Running from thread 10 of 16 on cpu 0
Running from thread 11 of 16 on cpu 0
Running from thread 12 of 16 on cpu 0
Running from thread 13 of 16 on cpu 0
Running from thread 14 of 16 on cpu 0
Running from thread 7  of 16 on cpu 0
Running from thread 15 of 16 on cpu 0
Running from thread 16 of 16 on cpu 0
Running from thread 6  of 16 on cpu 0
Running from thread 8  of 16 on cpu 0
Running from thread 5  of 16 on cpu 0
Running from thread 9  of 16 on cpu 0
Running from thread 4  of 16 on cpu 0
Running from thread 3  of 16 on cpu 0
Running from thread 2  of 16 on cpu 0
```

Αντίστοιχα, με χρήση της $OMP\_PLACES = "\{0:4\}"$, τα αποτελέσματα είναι τα παρακάτω:

Συμβ. 28: Affinity: Αποτελέσματα εκτέλεσης $OMP\_PLACES = "\{0:4\}"$

```
Running from thread 15 of 16 on cpu 0
Running from thread 12 of 16 on cpu 1
Running from thread 2  of 16 on cpu 2
Running from thread 4  of 16 on cpu 3
Running from thread 13 of 16 on cpu 1
Running from thread 1  of 16 on cpu 0
Running from thread 3  of 16 on cpu 2
Running from thread 11 of 16 on cpu 3
Running from thread 14 of 16 on cpu 1
Running from thread 6  of 16 on cpu 2
Running from thread 9  of 16 on cpu 3
Running from thread 16 of 16 on cpu 1
Running from thread 7  of 16 on cpu 2
Running from thread 10 of 16 on cpu 3
Running from thread 8  of 16 on cpu 0
Running from thread 5  of 16 on cpu 1
```

Επίσης, στο παράδειγμα που ακολουθεί, η οδηγία OMP_PROC_BIND είναι υπεύθυνη για τη δημιουργία όλων των νημάτων στον ίδιο πυρήνα με το κύριο νήμα:

Συμβ. 29: Παράδειγμα Affinity

```
int main( int argc, char**argv )
 {
     for (int i=1; i<100; i++){
     #pragma omp parallel
         {
             printf( "Running from thread %d of %d on cpu %2d!\n",
                 omp_get_thread_num()+1,
                 omp_get_num_threads(),
                 sched_getcpu());
         }
     }
     return 0;
 }
```

οπου για $OMP\_PROC\_BIND = "master"$, τα αποτελέσματα που εκτυπώνονται είναι τα εξής:

Συμβ. 30: Affinity: Αποτελέσματα εκτέλεσης $OMP\_PROC\_BIND = "master"$

```
Running from thread 1  of 16 on cpu  0!
Running from thread 7  of 16 on cpu  0!
Running from thread 11 of 16 on cpu  0!
Running from thread 10 of 16 on cpu  0!
Running from thread 12 of 16 on cpu  0!
Running from thread 13 of 16 on cpu  0!
Running from thread 14 of 16 on cpu  0!
Running from thread 15 of 16 on cpu  0!
Running from thread 16 of 16 on cpu  0!
Running from thread 8  of 16 on cpu  0!
...
Running from thread 2  of 16 on cpu  0!
Running from thread 1  of 16 on cpu  0!
Running from thread 7  of 16 on cpu  0!
Running from thread 11 of 16 on cpu  0!
Running from thread 10 of 16 on cpu  0!
Running from thread 12 of 16 on cpu  0!
Running from thread 13 of 16 on cpu  0!
Running from thread 14 of 16 on cpu  0!
Running from thread 15 of 16 on cpu  0!
Running from thread 16 of 16 on cpu  0!
Running from thread 8  of 16 on cpu  0!
Running from thread 9  of 16 on cpu  0!
```

Αξίζει τέλος να σημειωθεί η χρήση του προγράμματος numactl, που έχει ως στόχο τον έλεγχο της πολιτικής διεργασιών και διαχείριση της κοινόχρηστης μνήμης. Στο μηχάνημα εκτέλεσης των προγραμμάτων η εντολή *numactl -H* δίνει τα παρακάτω αποτελέσματα:

Συμβ. 31: Αποτελέσματα numactl -H

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3
node 0 size: 3944 MB
node 0 free: 2660 MB
node 1 cpus: 4 5 6 7
node 1 size: 4030 MB
node 1 free: 1863 MB
node 2 cpus: 12 13 14 15
node 2 size: 4030 MB
node 2 free: 2655 MB
node 3 cpus: 8 9 10 11
node 3 size: 4030 MB
node 3 free: 2105 MB
node distances:
node   0   1   2   3
  0:  10  16  16  16
  1:  16  10  16  16
  2:  16  16  10  16
  3:  16  16  16  10
```

3.2.2 Το *thread affinity* στην πράξη

Δυστυχώς, δεν υπάρχει ιδανική επιλογή για τη ρύθμιση της τοποθέτησης νήματος και της συγγένειας. Η επιλογή εξαρτάται από την εκάστοτε εφαρμογή που πρόκειται να εκτελεστεί. Ακόμη, ο χρήστης θα πρέπει να λαμβάνει υπόψη τις εξής παραμέτρους[19]:

- Ο χρόνος εκτέλεσης ενός προγράμματος μπορεί να επηρεαστεί από άλλες εργασίες που εκτελούνται στο ίδιο μηχάνημα εκείνη τη στιγμή και μοιράζονται πρόσβαση στο δίκτυο, τον δίαυλο μνήμης και στην προσωρινή μνήμη.
- Το λειτουργικό σύστημα του μηχανήματος εκτελεί ταυτόχρονα τις δικές του εργασίες

3.3 Tasking

Η έννοια των εργασιών(**Tasks**) εισήχθει στο *OpenMP* το 2008, με την έκδοση 3.0[17]. Οι εργασίες παρέχουν τη δυνατότητα παραλληλοποίησης αλγορίθμων με ακανόνιστη και μη εξαρτώμενη από το χρόνο ροή εκτέλεσης εργασιών. Ένας μηχανισμός ουράς διαχειρίζεται δυναμικά την ανάθεση εργασιών που πρέπει να εκτελεστούν στα διαθέσιμα νήματα. Τα νήματα παραλαμβάνουν εργασίες από την ουρά έως ότου αυτή αδειάσει. Η εργασία είναι ένα τμήμα κώδικα που ορίζεται εντός της παράλληλης περιοχής και εκτελείται ασύγχρονα και ταυτόχρονα με τις υπόλοιπες εργασίες της ίδιας περιοχής.

Οι εργασίες αποτελούν τμήμα της παράλληλης περιοχής κώδικα. Χωρίς ειδική μέριμνα, κάθε εργασία μπορεί να εκτελεστεί από κάθε νήμα. Αυτό αποτελεί πρόβλημα, καθώς οι οδηγίες διαμοιρασμού καθορίζουν με σαφήνεια τον τρόπο που οι εργασίες διανέμονται στα νήματα. Για να εγγυηθεί το *OpenMP* ότι κάθε εργασία εκτελείται μόνο μια φορά, η κατασκευή τους θα πρέπει να ενσωματωθεί σε μια οδηγία *single* ή *master*.

Συμβ. 32: Χρήση εργασιών

```
#include <omp.h>

int main(void) {
    #pragma omp parallel          // Beginning 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
}
```

Στο προηγούμενο παράδειγμα έστω ότι υπάρχουν δύο νήματα. Το ένα νήμα συναντά την οδηγία *single* και εκτελεί το τμήμα κώδικα που βρίσκεται εντός αυτής. Δύο εργασίες δημιουργούνται χωρίς να έχει ξεκινήσει όμως η εκτέλεσή τους. Ταυτόχρονα, το δεύτερο νήμα συναντά την υποκείμενη οδηγία *barrier* στο τέλος της οδηγίας *single* και περιμένει εκεί έως ότου εισαχθεί μια εργασία στην ουρά εργασιών προς εκτέλεση. Τα νήματα που καταλήγουν στο υποκείμενο φράγμα, είναι διαθέσιμα για την εκτέλεση των εργασιών. Ως συνέπεια, το νήμα που δημιουργεί τις εργασίες καταλήγει επίσης στο υποκείμενο φράγμα όταν ολοκληρωθεί η διαδικασία παραγωγής εργασιών και είναι και αυτό διαθέσιμο για εκτέλεση των εργασιών που απομένουν. Η σειρά εκτέλεσης των εργασιών δεν είναι προκαθορισμένη.

3.3.1 Η οδηγία *task*

Βασική οδηγία της έννοιας των εργασιών είναι η **pragma omp task** που ορίζει μια ρητή εργασία. Το περιβάλλον δεδομένων της εργασίας δημιουργείται σύμφωνα με τις *φράσεις διαμοιρασμού δεδομένων* κατά την κατασκευή των εργασιών και τυχόν προεπιλογές που ισχύουν για τις φράσεις αυτές[31].

Συμβ. 33: Γραμματική οδηγίας *task*

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

Συμβ. 34: Φράσεις οδηγίας *task*

```
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 που δέχεται η συνθήκη **if** ως όρισμα σε μια εργασία, αξιολογείται ως ψευδής ή αληθής. Όταν η έκφραση αξιολογείται ως ψευδής, το νήμα **A** που δημιουργεί τη νέα εργασία αναστέλλεται έως ότου αυτή ολοκληρωθεί. Η εργασία του νήματος **A** δε μπορεί να συνεχιστεί από κάποιο άλλο νήμα, ακόμα και αν αυτό είναι untied)[8]. Ανεξαρτήτου αποτελέσματος της έκφρασης ωστόσο, δημιουργείται πάντα ένα νέο περιβάλλον δεδομένων για τη εργασία.

Ως εκ τούτου η φράση *if* αποτελεί λύση σε αλγόριθμους όπως οι αναδρομικοί, όπου το υπολογιστικό κόστος μειώνεται καθώς αυξάνεται το βάθος, αλλά το όφελος από τη δημιουργία μιας νέας εργασίας μειώνεται λόγω του γενικού κόστους κατασκευής της. Παρόλα αυτά, το κόστος της ρύθμισης περιβάλλοντος δεδομένων μπορεί να είναι το κυρίαρχο για τη δημιουργία εργασίας, διότι για λόγους ασφαλείας οι μεταβλητές που αναφέρονται σε μια οδηγία κατασκευής εργασιών είναι στις περισσότερες περιπτώσεις από προεπιλογή *firstprivate*[17].

3.3.1.2 Φράση final

Όταν η φράση *final* χρησιμοποιείται μέσα στην οδηγία *task*, η έκφραση αξιολογείται κατά τη διάρκεια δημιουργίας της εργασίας. Αν είναι αληθής, η εργασία θεωρείται τελική. Όλες οι εργασίες παιδιά που δημιουργούνται μέσα σε αυτή, αγνοούνται και εκτελούνται στο πλαίσιο της. Επί της ουσίας πρόκειται για άμεση σειριακή εκτέλεση του κώδικα εντός της περιοχής της νέας εργασίας από το ίδιο νήμα[8].

Υπάρχουν δύο διαφορές ανάμεσα στη φράση *if* και στη *final*. Οι διαφορές αυτές αφορούν:

- τις εργασίες που επηρεάζει κάθε φράση
- τον τρόπο με τον οποίο διαχειρίζεται ο μεταγλωττιστής της κατασκευασμένες εργασίες[10].

Πίνακας 4: Διαφορές ανάμεσα στις φράσεις *if* και *final* όταν εισάγονται σε κατασκευή εργασίας.

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

3.3.1.3 Φράση mergeable

Μια *¨συγχωνευμένη¨* εργασία είναι αυτή της οποίας το περιβάλλον δεδομένων είναι ίδιο με αυτό της εργασίας που τη δημιούργησε. Όταν εισάγεται η φράση *mergeable* σε μια οδηγία *task* τότε η υλοποίηση της δημιουργεί μια συγχωνευμένη εργασία.

3.3.1.4 Φράση depend

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

Συμβ. 35: Φράση *depend*

```
depend([depend-modifier, ]dependency-type : locator-list)

dependence-type:
        in
        out
        inout
        mutexinoutset
        depobj

depend-modifier:
        iterator(iterators-definition)
        depend([source|depend(sink: vec)])
```

Οι εξαρτήσεις των εργασιών καθορίζονται από τον τύπο που ορίζεται σε μια φράση εξάρτησης και την λίστα των περιεχόμενων αντικειμένων που ακολουθεί. Ο τύπος εξάρτησης μπορεί να είναι ένας από τους ακόλουθους[27].

Για τον τύπο **in(x)**, η προκύπτουσα εργασία (y) θα είναι εξαρτημένη όλων των προηγούμενων συγγενικών εργασιών που έχουν τύπο εξάρτησης *out* ή *inout* και το **x** αναφέρεται στην λίστα. Η y θα ξεκινήσει μετά την ολοκλήρωση των εξαρτήσεών της.

Για τους τύπους εξάρτησης ***out(x)*** και ***inout***, η προκύπτουσα εργασία θα εκτελεστεί και ολοκληρωθεί πριν από την εκτέλεση των εργασιών τύπου *in ή inout* και περιέχουν την μεταβλητή **x** στη λίστα.

Παράδειγμα

- Η φράση *depend(in:x)* θα δημιουργήσει μια εξάρτηση με όλες τις εργασίες που δημιουργήθηκαν με τη φράση *depend(out:x) ή depend(inout: x)*.
- Η φράση *depend(out:x) ή depend(inoout:x)* θα δημιουργήσει μια εργασία εξαρτημένη με όλες τις προηγούμενες εργασίες που αναφέρουν τη μεταβλητή *x* στη φράση εξάρτησης.

Από τα παραπάνω, προκύπτει η εξής εξάρτηση εργασιών:

Συμβ. 36: Παράδειγμα εξάρτησης εργασιών

```
#pragma omp task depend(out:x) // task1
...
#pragma omp task depend(in:x) depend(out:y) // task2
...
#pragma omp task depend(inout:x) // task3
...
#pragma omp task depend(in:x, y) // task4
...


task1(out:x) -> task2(in:x, out:y) -> task4(in:x, y)
                |
                -> task3(inout:x)
```

3.3.1.5 Φράση untied

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

3.3.2 Συγχρονισμός εργασιών

Οι εργασίες δημιουργούνται όταν το πρόγραμμα συναντήσει την οδηγία **pragma omp task**. Η στιγμή εκτέλεσης τους δεν είναι προκαθορισμένη. Το *OpenMP* εγγυάται ότι θα έχουν ολοκληρωθεί όταν ολοκληρωθεί η εκτέλεση του προγράμματος ή όταν προκύψει κάποια οδηγία συγχρονισμού νημάτων.

Στην περίπτωση της δημιουργίας εργασιών μέσω οδηγίας ***single*** στο τέλος της οδηγίας υπάρχει υποκείμενο φράγμα εκτέλεσης, σε αντίθεση με την οδηγία ***master***.

Πίνακας 5: Οδηγίες συγχρονισμού εργασιών.

Οδηγία	Περιγραφή
#pragma omp barrier	Μπορεί να υπονοείται μέσω μιας οδηγίας ή να δηλωθεί ρητά από το χρήστη.
#pragma omp taskwait	Αναμένει μέχρι να ολοκληρωθούν όλες οι εργασίες-παιδιά της συγκεκριμένης εργασίας.
#pragma omp taskgroup	Αναμένει μέχρι να ολοκληρωθούν όλες οι εργασίες παιδιά της συγκεκριμένης εργασίας αλλά και οι απόγονοι τους.

Συμβ. 37: Χρήση taskwait

```
#pragma omp parallel {
        #pragma omp single
        {
                #pragma omp task
                { ... } // Task #1

                #pragma omp task
                { ... } // 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**(CPU)** και μονάδων επεξεργασίας γραφικών (και άλλων τύπων επιταχυντών) στον ίδιο δίαυλο, με κοινόχρηστη μνήμη και εργασίες[21].

Παρόλο που οι GPU λειτουργούν σε χαμηλότερες συχνότητες, έχουν συνήθως περισσότερους πυρήνες από τις CPU. Έτσι, μπορούν να επεξεργαστούν περισσότερες εικόνες και γραφικά δεδομένα ανα δευτερόλεπτο από μια παραδοσιακή CPU. Η μετεγκατάσταση των δεδομένων σε μορφη γραφικών και στη συνέχεια η χρήση της GPU για σάρωση και ανάλυση μπορεί να αυξήσει εκθετικά την επιτάχυνση. Η αύξηση της δημοτικότητας των ετερογενών αρχιτεκτονικών σε όλους τους τύπους υπολογιστών είχε αξιοσημείωτο αντίκτυπο στην ανάπτυξη λογισμικών υψηλών προδιαγραφών.

Για την εκμετάλλευση των ετερογενών συστημάτων, οι χρήστες πρέπει να κατασκευάζουν λογισμικό που εκτελεί υπολογισμούς σε διαφορετικές συσκευές. Σε αυτούς τους υπολογισμούς περιλαμβάνονται οι βρόγχοι επανάληψης, που χρησιμοποιούνται ευρέως στην ανάπτυξη λογισμικών.

Ωστόσο, τα μοντέλα προγραμματισμού για ετερογενή συστήματα είναι δύσκολο να χρησιμοποιηθούν λόγω της αυξημένης δυσκολίας διαχείρισής τους. Συνήθως, τμήματα κώδικα γράφονται σε δύο εκδόσεις, μία φορά για τον επεξεργαστή γενικού σκοπού και μια για τον επιταχυντή. Η συντήρηση και ανάπτυξη διπλού κώδικα, αποτελεί ένα από τα μεγαλύτερα προβλήματα στη διαδικασία του προγραμματισμού.

Για τις ανάγκες διευκόλυνσης, το *OpenMP* επέκτεινε τις λειτουργίες του με σκοπό την υποστήριξη και ευρεία χρήση τέτοιων τύπων συστημάτων[14]. Τα αποτελέσματα της εργασίας αρχικά δημοσιεύτηκαν στην έκδοση 4.0 και εξελίχθηκαν περαιτέρω στην έκδοση 4.5. Οι χρήστες μπορούν πλέον να χρησιμοποιήσουν τη διεπαφή για τη δημιουργία λογισμικών γραμμένων σε γλώσσες προγραμματισμού υψηλότερου επιπέδου και εκτελέσιμων από συσκευές επεξεργασίας γραφικών. Έτσι επιτυγχάνεται η διατήρηση μίας μόνο έκδοσης του κώδικα, η οποία μπορεί να τρέξει είτε σε μονάδα επεξεργασίας γραφικών ή σε επεξεργαστή γενικής χρήσης(*CPU*).

Με τον όρο συσκευή στόχου, εννοείται ένας υπολογιστικός πόρος στον οποίο μπορεί να εκτελεστεί μια περιοχή κώδικα. Παραδείγματα τέτοιων συσκευών είναι οι *GPU, CPU, DSP, FPGA* κ.α. Οι συσκευές στόχου 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 στη συσκευή στόχου (επιταχυντή).

Συμβ. 38: Εκτέλεση κώδικα στη συσκευή στόχου

```
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];
    }
}
```

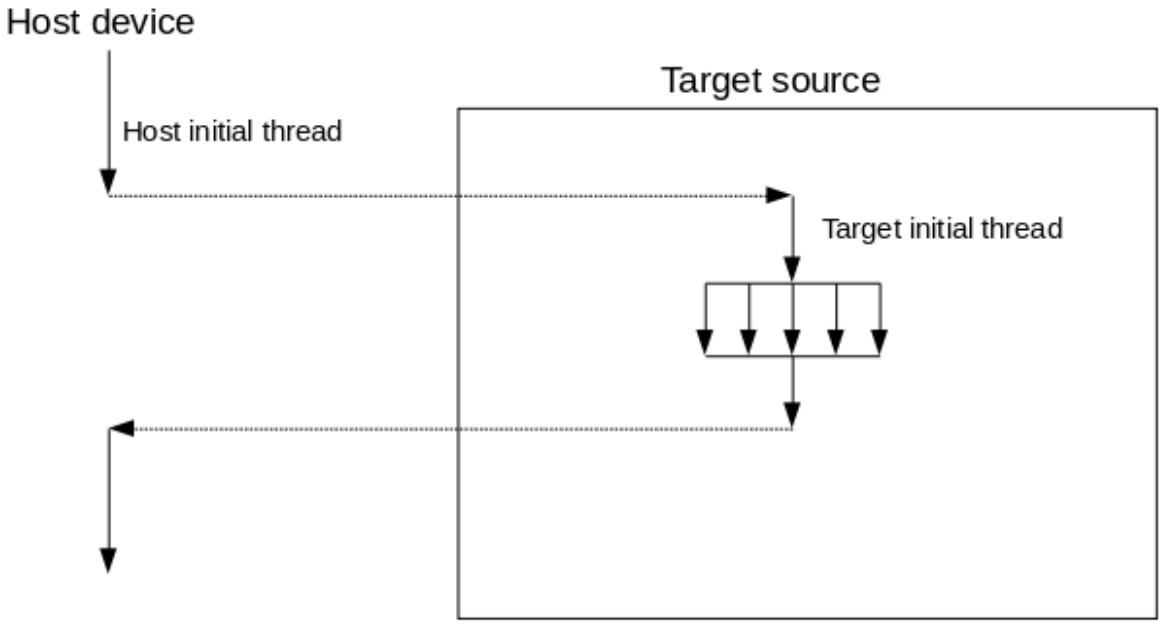
Όπως δείχνει το παραπάνω παράδειγμα, ένα νήμα της κύριας συσκευής(*host*) συναντάει την οδηγία ***target***. Το τμήμα κώδικα που ακολουθεί μεταφέρεται και εκτελείται στη συσκευή στόχου, αν αυτός υπάρχει. Από προεπιλογή, το νήμα που συναντά την οδηγία περιμένει την ολοκλήρωση της εκτέλεσης της παράλληλης περιοχής προτού συνεχίσει.

Πριν ένα καινούργιο νήμα που βρίσκεται στη συσκευή στόχου αρχίσει να εκτελεί την περιοχή που περικλείεται στην οδηγία *target*, οι μεταβλητές *A, B, C* απεικονίζονται(*map* στον επιταχυντή. Η φράση *map* είναι το εργαλείο που χρησιμοποιεί το *OpenMP* για να εξασφαλίσει την πρόσβαση της συσκευής στόχου στις μεταβλητές αυτές.

3.4.1 Το αρχικό νήμα της συσκευής στόχου

Το νήμα που ξεκινάει την εκτέλεση ενός προγράμματος ονομάζεται κύριο νήμα και ανήκει πάντα στην κεντρική συσκευή. Με άλλα λόγια, ένα πρόγραμμα σε μια αρχιτεκτονική ετερογενούς προγραμματισμού δε ξεκινάει ποτέ από τη συσκευή στόχου.

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 στόχου, την αναλαμβάνει ένα νέο αρχικό νήμα και όχι το νήμα που συνάντησε την οδηγία *target*. Το νήμα αυτό μπορεί να συναντήσει οδηγίες παραλληλισμού και να δημιουργήσει υποομάδες νημάτων.



Σχήμα 10: Διάγραμμα ομάδων νημάτων σε ετερογενή αρχιτεκτονική

3.4.2 Μοντέλο μνήμης ετερογενούς αρχιτεκτονικής

Στις επόμενες παραγράφους, ακολουθεί μια περιγραφή του μοντέλου μνήμης της ετερογενούς αρχιτεκτονικής, περιγράφονται έννοιες που σχετίζονται με τις μεταβλητές και η γνώση του θεωρητικού υπόβαθρου καθίσταται απαραίτητη για την ορθή υλοποίηση προγραμμάτων σε τέτοιες αρχιτεκτονικές.

3.4.2.1 Η οδηγία *map*

Συμβ. 39: Σύνταξη οδηγίας map

```
map ([[map-type-modifier[,]} map-type:} list)
```

Συμβ. 40: Φράσεις map

```
alloc
to
from
tofrom -> default
release
delete
```

Όπως τα νήματα των κύριων συσκευών, έτσι και τα νήματα που δημιουργούνται στις συσκευές στόχου μπορούν να έχουν ιδιωτικές μεταβλητές. Αντίγραφα μεταβλητών στη συσκευή στόχου με χαρακτηριστικό ιδιωτικής μνήμης, δημιουργούνται όταν η οδηγία *target* ακολουθείται από τη φράση *private ή firstprivate*.

Στις αρχιτεκτονικές ετερογενούς προγραμματισμού και σε επίπεδο υλικού, η κύρια συσκευή με τον επιταχυντή μπορεί να μοιράζονται κοινόχρηστη φυσική μνήμη. Η φράση *map* χρησιμοποιείται για την αντιστοίχιση δεδομένων ανάμεσα στις δύο συσκευές, αποκρύπτοντας παράλληλα χαρακτηριστικά της φυσικής υλοποίησης. Για παράδειγμα, όταν οι δυο συσκευές δεν έχουν κοινόχρηστη φυσική μνήμη, η μεταβλητή αντιγράφεται στον επιταχυντή. Αντίθετα, στην περίπτωση της υλοποίησης με κοινόχρηστη μνήμη, δεν απαιτείται δημιουργία αντίγραφου. Η φράση *map* απαλλάσσει τον χρήστη από τον έλεγχο των χαρακτηριστικών της υλοποίησης σε επίπεδο υλικού, και η διεπαφή ενεργεί ανάλογα με την αρχιτεκτονική που χρησιμοποιείται.

Πίνακας 6: Ενέργειες κατά την απεικόνιση μνήμης

	memory allocation	**copy**	**flush**
Απεικονισμένη Μνήμη	Ναι	Ναι	Ναι
Διαμοιρασμένη Μνήμη	Όχι	Όχι	Ναι

Η αντιστοίχιση μεταβλητών στον επιταχυντή, χωρίζεται σε τρεις φάσεις:

1. Η φάση εισόδου(*map-enter*) στην αρχή της εκτέλεσης της οδηγίας *target*, όπου η μεταβλητή απεικονίζεται στον επιταχυντή. Σε αυτή, δεσμεύεται μνήμη του επιταχυντή για την αποθήκευση της μεταβλητής και αντιγράφεται από την κύρια συσκευή.
2. Η φάση υπολογισμού, που προκύπτει όταν κατά τη διάρκεια εκτέλεσης της παράλληλης περιοχής, τα νήματα που εκτελούν το πρόγραμμα αποκτούν πρόσβαση στην απεικονισμένη μεταβλητή.
3. Η φάση εξόδου όπου ολοκληρώνεται η αντιστοίχιση των μεταβλητών στον επιταχυντή. Η τιμή της μεταβλητής στον επιταχυντή αντιγράφεται στην αντίστοιχη θέση της κύριας συσκευής. Η δεσμευμένη μνήμη του επιταχυντή ελευθερώνεται.

Οι φάσεις **1** και **3** διαχειρίζονται την αποθήκευση και την αντιγραφή των μεταβλητών ανάμεσα σε δυο συσκευές. Ο τύπος της αντιστοίχισης επηρεάζει την αντιγραφή μεταβλητών στον επιταχυντή ή την κύρια συσκευή. Ο καθορισμός του τύπου αντιστοίχισης επηρεάζει την απόδοση του κώδικα.

Πίνακας 7: Απαιτούμενη αντιγραφή για κάθε τύπο μεταβλητής κατά τις φάσεις εισόδου-εξόδου

map-type	**Είσοδος**	**Έξοδος**
alloc	Όχι	Όχι
to	Ναι	Όχι
from	Όχι	Ναι
tofrom	Ναι	Ναι
release	-	Όχι
delete	-	Όχι

Συμβ. 41: Παράδειγμα χρήσης τύπου αντιστοίχισης μεταβλητών

```
void foo(double A[1024], double B[1024], double C[1024) {
        #pragma omp target map(from : A) map(to: B)
                           map(alloc: C) // map enter
        { ... } // map exit
}
```

Στο προηγούμενο παράδειγμα:

Η μεταβλητή **A**:

- Δεν αρχικοποιείται στον επιταχυντή.
- Η τιμή της αντιγράφεται στην κύρια συσκευή.
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στην κύρια συσκευή.

Η μεταβλητή **B**:

- Η τιμή της αντιγράφεται στον επιταχυντή.
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στην κύρια συσκευή.

Η μεταβλητή **C**:

- Η τιμή της αντιγράφεται στον επιταχυντή.
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στην κύρια συσκευή.

3.4.2.2 Περιβάλλον δεδομένων συσκευής

Κάθε επιταχυντής έχει ένα περιβάλλον μνήμης στο οποίο περιέχεται το σύνολο των μεταβλητών που είναι προσβάσιμες από νήματα που ενεργούν σε αυτή τη συσκευή. Η οδηγία *map* διασφαλίζει ότι η μεταβλητή μεταφέρεται από την κύρια συσκευή, στο περιβάλλον δεδομένων του επιταχυντή και είναι προσβάσιμη από αυτόν.

Ανάλογα με την αρχιτεκτονική και τη διαθεσιμότητα κοινόχρηστης μνήμης μεταξύ της κύριας συσκευής*(host)* και της συσκευής στόχου, η πρωτότυπη μεταβλητή που δημιουργήθηκε στην κύρια συσκευή και η αντίστοιχη της συσκευής στόχου είναι είτε η ίδια μεταβλητή που βρίσκεται στη κοινόχρηστη μνήμη των δυο συσκευών ή αντίγραφα σε διαφορετικές θέσεις μνήμης. Στη δεύτερη περίπτωση, απαιτούνται εργασίες αντιγραφής και ενημέρωσης για να διατηρηθεί η συνέπεια μεταξύ των δυο θέσεων.

Η βελτιστοποίηση της ποσότητας της μεταφοράς δεδομένων ανάμεσα στις δυο συσκευές, αποτελεί κρίσιμο σημείο για την επίτευξη καλύτερης απόδοσης στις ετερογενείς αρχιτεκτονικές. Η συνεχής αντιστοίχιση μεταβλητών που επαναχρησιμοποιούνται είναι αναποτελεσματική και οδηγεί σε πτώση της απόδοσης.

3.4.2.3 Δείκτες μεταβλητών συσκευής

Αν η κύρια συσκευή και ο επιταχυντής δεν μοιράζονται την κοινόχρηστη μνήμη, οι τοπικές μεταβλητές τους βρίσκονται σε διαφορετικές θέσεις μνήμης. Όταν μια μεταβλητή απεικονίζεται στο περιβάλλον δεδομένων ενός επιταχυντή, γίνεται μια αντιγραφή και η καινούργια μεταβλητή είναι διαφορετική από την μεταβλητή της κύριας συσκευής.

Οι διευθύνσεις μνήμης αποθηκεύονται σε μεταβλητές που ονομάζονται δείκτες (*pointers*). Ένα νήμα της κύριας συσκευής δε μπορεί να έχει πρόσβαση σε μνήμη μέσω ενός δείκτη που περιέχει διεύθυνση μνήμης του επιταχυντή. Ακόμη, ο επιταχυντής και η κύρια συσκευή 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 στη κύρια συσκευή και περιέχει την διεύθυνση μνήμης στο περιβάλλον δεδομένων του επιταχυντή. Το *OpenMP* παρέχει ρουτίνες και οδηγίες που καθιστούν εφικτή τη δέσμευση μνήμης στο περιβάλλον του επιταχυντή μέσω της κύριας συσκευής, όπως φαίνεται στο παρακάτω παράδειγμα:

Συμβ. 42: omp_target_alloc

```
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;
```

3.4.3 Η οδηγία target

Σκοπός της οδηγίας *target* είναι η μεταφορά και εκτέλεση ενός τμήματος κώδικα στον επιταχυντή. Η εκτέλεση γίνεται από ένα αρχικό νήμα στη συσκευή. Σε περίπτωση έλλειψης επιταχυντή στο σύστημα, ο κώδικας που προορίζεται να εκτελεστεί εκεί μέσω της οδηγίας *target* θα εκτελεστεί στην κύρια συσκευή αγνοώντας τις οδηγίες *#pragma*.

Συμβ. 43: Σύνταξη οδηγίας target

```
#pragma omp target [clause[[,] clause]...]
```

Συμβ. 44: Παράδειγμα εκτέλεσης στον επιταχυντή

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

Η οδηγία *target* δημιουργεί μια εργασία που εκτελείται στον επιταχυντή. Η εργασία για την κύρια συσκευή ολοκληρώνεται όταν ολοκληρωθεί η εκτέλεση στον επιταχυντή. Οι φράσεις *nowait* και *depend* επηρεάζουν τον τύπο και την ασύγχρονη συμπεριφορά της εργασίας. Από προεπιλογή, η εργασία στόχου περιλαμβάνει φράγμα εκτέλεσης στο τέλος της. Το νήμα που τη συναντά περιμένει μέχρι την ολοκλήρωση της εκτέλεσής της.

Οι δείκτες μεταβλητών που εισάγονται στη φράση *map*, είναι ιδιωτικές (*private*) μέσα στη συσκευή στόχου. Οι ιδιωτικές μεταβλητές δείκτη διεύθυνσης αρχικοποιούνται με την τιμή της διεύθυνσης του επιταχυντή. Οι φράσεις που υποστηρίζονται από την οδηγία είναι:

Συμβ. 45: Φράσεις οδηγίας *target*

```
if (/target:] scalar-expression)
map ([[map-type-modifier[, map-type:] list]
device (integer-expression)
private (list)
firstprivate (list)
is_device_ptr (list)
defaultmap( tofrom:scalar)
nowait
depend(dependence-type: list)
```

3.4.4 Η οδηγία target teams

Η οδηγία *target teams* κατασκευάζει ομάδες νημάτων (*league*) που λειτουργούν σε έναν επιταχυντή. Η κάθε ομάδα αποτελείται αρχικά από ένα νήμα. Η λειτουργία αυτή είναι παρόμοια με μια οδηγία *parallel* με τη διαφορά ότι σε αυτή την περίπτωση δημιουργούνται ομάδες που αρχικά αποτελούνται από ένα νήμα και στη συνέχεια τα νήματα της ομάδας πολλαπλασιάζονται. Νήματα διαφορετικών ομάδων δε συγχρονίζονται μεταξύ τους.

Όταν ένα νήμα συναντά μια οδηγία teams, δημιουργείται 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 τους.

Συμβ. 46: Φράσεις οδηγίας *target teams*

```
num_teams (integer-expression)
threadJimit (integer-expression)
default(shared I none)
private (list)
firstprivate (list)
shared (list)
reduction (reduction-identifier : list)
```

3.4.5 Η οδηγία distribute

Συμβ. 47: Σύνταξη οδηγίας *distribute*

```
#pragma omp distribute {clause[[,} clause}. . . j
        for-loops
```

Συμβ. 48: Φράσεις υποστηριζόμενες από την οδηγία *distribute*

```
private {list)
firstprivate {list)
lastprivate {list)
collapse (n)
dist_schedule {kind[, chunk_sizej)
```

Η οδηγία *distribute* χρησιμοποιείται για τον δήλωση διαμοιρασμού των επαναλήψεων ενός βρόγχου στα αρχικά νήματα των ομάδων που δημιουργήθηκαν από την οδηγία *target teams*. Η οδηγία δεν περιλαμβάνει υπονοούμενο φράγμα εργασιών στο τέλος της, πράγμα που σημαίνει ότι τα κύρια νήματα των ομάδων δε συγχρονίζονται στο τέλος της οδηγίας.

Η φράση *dist_schedule* καθορίζει τον τρόπο που διαμοιράζονται οι επαναλήψεις σε τμήματα. Συγκριτικά με την οδηγία *for*, η *distribute* έχει πιθανότητες για καλύτερη απόδοση καθώς ο μεταγλωττιστής μπορεί να πετύχει καλύτερο επίπεδο βελτιστοποίησης.

3.4.6 Σύνθετες οδηγίες επιταχυντών

Οι συνδυασμένες οδηγίες είναι ισοδύναμες με τις επιμέρους οδηγίες από τις οποίες αποτελούνται. Για παράδειγμα, η οδηγία *parallel for* έχει την ίδια σημασία με την *parallel* ακολουθούμενη από την οδηγία *for*. Παρόλα αυτά, ορισμένες φορές οι συνδυασμένες οδηγίες μπορούν να επιτύχουν καλύτερες επιδόσεις. Σε αυτή την παράγραφο, οι οδηγίες χωρίζονται σε δύο κατηγορίες, τις συνδυασμένες με *target* και αυτές που συνδυάζονται με *target teams*.

Συμβ. 49: Συνδυασμένες οδηγίες συσκευής στόχου

```
#pragma omp target parallel [clause[[,] clause]...]
        structured blocks
#pragma omp target parallel for [clause[[,] clause]...]
        for-loops
#pragma omp target parallel for simd [clause[[,] clause]...]
        for-loops
#pragma omp target simd [clause[[,] clause]...]
        for-loops
```

Συμβ. 50: Συνδυασμένες οδηγίες διαμοιρασμού εργασιών στη συσκευή στόχου

```
#pragma omp distribute parallel for [clause[[,] clause]...]
        for-loops
#pragma omp distribute simd [clause[[,] clause]...]
        for-loops
#pragma omp distribute parallel for simd [clause[[,] clause]...]
        for-loops
```

3.4.7 Οδηγία *declare target*

Η οδηγία *declare target* χρησιμοποιείται για συναρτήσεις και μεταβλητές. Μια συνάρτηση που καλείται μέσα στο τμήμα του *target* κώδικα, θα πρέπει να δηλώνεται στην οδηγία *declare target*. Ακόμη, η οδηγία χρησιμοποιείται για την αντιστοίχιση καθολικών μεταβλητών στο περιβάλλον δεδομένων του επιταχυντή.

Συμβ. 51: Οδηγία declare target

```
#pragma omp declare target
        declarations-definitions-seq
#pragma omp end declare target
#pragma omp declare target(extended-list)
#pragma omp declare target clause[[,] clause]...]

CLAUSE:
to (extended-list)
link (list)
```

4 Υλοποιημένα παραδείγματα

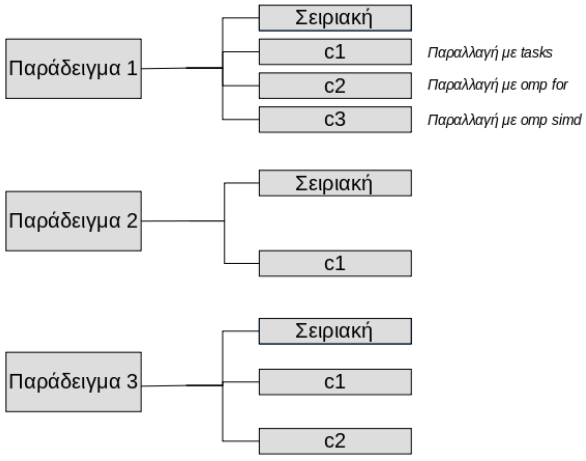
Οι ενότητες που ακολουθούν αφορούν τη μελέτη των νέων κυρίως χαρακτηριστικών του *OpenMP* που συζητήθηκαν στα προηγούμενα κεφάλαια, μέσω της υλοποίησης παραδειγμάτων. Τα προβλήματα που θα αναπτυχθούν, αντιπροσωπεύουν κατηγορίες βασικών προβλημάτων που συναντώνται συχνά σε ζητήματα παράλληλου προγραμματισμού. Για κάθε πρόβλημα υλοποιείται η σειριακή μέθοδος επίλυσης και παραλλαγές με παραλληλισμό που κυρίως χρησιμοποιεί χαρακτηριστικά που εισήχθησαν στις εκδόσεις μετά την 2.5. Στόχος είναι η σύγκριση των διαφορετικών παραλλαγών του ίδιο προβλήματος σε επίπεδο χρονικών επιδόσεων με χρήση διαγραμμάτων και πινάκων, αλλά και την εξαγωγή παρατηρήσεων και συμπερασμάτων που προκύπτουν από αυτές. Σε κάθε ανάλυση παρατίθενται και τμήματα του πηγαίου κώδικα της επιμέρους παραλλαγής.

Τα παραδείγματα που χρησιμοποιήθηκαν αναφέρονται επιγραμματικά παρακάτω και με λεπτομερέστερη περιγραφή στα αντίστοιχα κεφάλαια τους:

- Υπολογισμός π
- Linked list traversal
- SAXPY
- Υπολογισμός πρώτων αριθμών
- Πολλαπλασιασμός πινάκων
- Quicksort
- Mergesort
- Producer-Consumer
- Discrete Fourier Transform

4.1 Μεθοδολογία σύνταξης προβλημάτων

Για όλα τα προβλήματα χρησιμοποιήθηκε η ίδια μεθοδολογία επίλυσης. Τα προβλήματα έχουν χωριστεί σε ξεχωριστούς φακέλους ανά πρόβλημα, όπου κάθε φάκελος περιέχει το βασικό πηγαίο κώδικα που ξεκινάει το πρόγραμμα και υποφακέλους που περιέχουν τις επιμέρους παραλλαγές. Με τη βοήθεια της παραδοχής κοινών ονομάτων και κοινού signature συναρτήσεων, για τη μεταγλώττιση των διαφορετικών παραλλαγών του ίδιου προβλήματος, με τη μόνη διαφορά να είναι ο φάκελος στον οποίο θα κάνει link ο μεταγλωττιστής. Με αυτό, τον τρόπο, ο βασικός κορμός του προβλήματος παραμένει ίδιος και αποφεύγεται ο διπλότυπος κώδικας. Η υλοποίηση των προβλημάτων είναι διαθέσιμη στον σύνδεσμο: `https://github.com/gkonto/openmp`



Σχήμα 11: Διάρθρωση παραδειγμάτων στο *github.com*

Για παράδειγμα, έστω ότι επιλύεται ένα πρόβλημα που ονομάζεται *Foo* με διαφορετικές παραλλαγές μιας συνάρτησης *fun()*. Τότε δημιουργείται ένα κεντρικό αρχείο *run.cpp* που περιέχει τη *main()* και σε κάθε υποφάκελο (*c1, c2*) ένα αρχείο *test.cpp* με διαφορετική παραλλαγή της *fun()*. Τότε στο αρχείο run.cpp γίνεται *#include "test.hpp"* και η εντολές για *compile* θα είναι οι εξής:

g++ run.cpp ./c1/test.hpp -I c1

g++ run.cpp ./c2/test.hpp -I c2

4.2 Αρχιτεκτονική μηχανήματος και επιλογές μεταγλώττισης

Τα προβλήματα που ακολουθούν εκτελέστηκαν σε μηχάνημα με λειτουργικό *linux* και μεταγλωττιστή *gcc-7.5.0*. Οι προδιαγραφές υλικού του μηχανήματος που εκτελέστηκαν τα προβλήματα, αναφέρονται στον πίνακα που ακολουθεί. Για τη μεταγλώττιση των παραλλαγών χρησιμοποιήθηκε η επιλογή -O2, με σκοπό την εξαγωγή συμπερασμάτων σχετικά με τη διανυσματικοποίηση μέσω της οδηγίας *simd* σύμφωνα με την οποία δεν υπάρχει υποκείμενη διανυσματικοποίηση. Ακόμη, γίνεται η χρήση της εντολής no-inline με σκοπό την απαγόρευση της μετατροπής συναρτήσεων σε inline αυτόματα από το μεταγλωττιστή, για την αποφυγή εσφαλμένων συμπερασμάτων σχετικά με τη βελτίωση της απόδοσης από άλλους παράγοντες. Αξίζει επίσης να σημειωθεί η χρήση της επιλογής -fno-stack-protector σε παραλλαγές με χρήση GPU. Η επιλογή απαιτείται σε προβλήματα δέσμευσης μεγάλου τμήματος stack μνήμης. Περισσότερες λεπτομέρειες σχετικά με τις επιλογές μεταγλώττισης δίνονται στις επιμέρους παραλλαγές.

Πίνακας 8: Χαρακτηριστικά Μηχανήματος Εκτέλεσης

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	16
Thread(s) per core	1
Core(s) per socket	8
Socket(s)	2
NUMA node(s)	4
Model name	AMD Opteron(tm) Processor 6128 HE
L1d cache	64K
L2 cache	512K
L3 cache	5118K
Memory	16036

4.3 Πρόσθεση πινάκων αριθμών μικρής ακρίβειας - *SAXPY*

Μια από τις λειτουργίες που κατέχει θεμελιώδη θέση σε εφαρμογές της γραμμικής άλγεβρας, αποτελεί η πρόσθεση πινάκων πραγματικών αριθμών μικρής ακρίβειας (*floats*), γνωστή ως **SAXPY**.

Στο παράδειγμα *SAXPY*, ο λόγος του μεγέθους υπολογισμών προς το μέγεθος των δεδομένων που τελούν υπό επεξεργασία είναι μικρός. Ως εκ τούτου, αποτελεί πρόβλημα περιορισμένης επεκτασιμότητας. Παρόλα αυτά, πρόκειται για ένα χρήσιμο παράδειγμα και για αυτό μελετάται.

4.3.1 Περιγραφή προβλήματος

Η λειτουργία SAXPY δέχεται ως δεδομένα δυο μονοδιάστατους πίνακες πραγματικών αριθμών. Οι πίνακες *x,y* πρέπει να έχουν το ίδιο μέγεθος. Ο πρώτος πίνακας πολλαπλασιάζεται με μια σταθερά *a* και το αποτέλεσμα προστίθεται στο δεύτερο πίνακα*y*.

Ο υπολογισμός αυτός εμφανίζεται συχνά στην αριθμητική. Το όνομα *SAXPY* δόθηκε από την βιβλιοθήκη **BLAS** (*"Basic Linear Algrbra Subprograms"*) για πραγματικούς αριθμούς μικρής ακρίβειας (*floats*). Ο αντίστοιχος αλγόριθμος διπλής ακρίβειας ονομάζεται *DAXPY*, ενώ για μιγαδικούς αριθμούς ονομάζεται *CAXPY*. Η μαθηματική διατύπωση του *SAXPY* είναι:

$$\mathbf{y} = a * \mathbf{x} + \mathbf{y}$$

όπου ο πίνακας x χρησιμοποιείται ως είσοδος, ο y ως είσοδος και έξοδος. Δηλαδή ο αρχικός πίνακας y τροποποιείται. Εναλλακτικά, η λειτουργία *SAXPY* μπορεί να περιγραφεί ως συνάρτηση που δρα σε μεμονωμένα στοιχεία, όπως φαίνεται παρακάτω:

$$f(t, p, q) = tp + q$$
$$\forall_i : y_i \leftarrow f(a, x_i, y_i)$$

Οι συναρτήσεις τύπου f δέχονται ως ορίσματα δύο είδη παραμέτρων. Τις παραμέτρους όπως την a που παραμένουν σταθερές και ονομάζονται *uniform*, οι παράμετροι που είναι μεταβλητές σε κάθε κλήση της f ονομάζονται *varying*. Το μοτίβο *map* καλεί τη συνάρτηση f τόσες φορές όσες και ο αριθμός των στοιχείων του πίνακα.[25].

4.3.2 Περιγραφή κοινού τμήματος αλγορίθμου SAXPY

Το πρόβλημα ξεκινάει δημιουργώντας ένα στοιχείο τύπου *Containers*, που περιέχει του πίνακες που εισάγονται στον αλγόριθμο SAXPY. Ο ρόλος του Containers είναι για την δυναμική διαχείριση μνήμης. Οι πίνακες και η σταθερά cons αρχικοποιούνται με τυχαίους αριθμούς μικρής ακρίβειας. Το μέγεθος των πινάκων (μέγεθος προβλήματος) ορίζεται από τον χρήστη μέσω της γραμμής εντολών. Στη συνέχεια, καλείται η ρουτίνα που γίνεται υπολογισμός των επιμέρους στοιχείων SAXPY και μόλις ολοκληρωθεί γίνεται επαλήθευση των αποτελεσμάτων, όπου αν επαληθευτούν σωστά, γίνεται εκτύπωση του χρόνου εκτέλεσης της παραλλαγής.

Συμβ. 52: SAXPY: main

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    Containers c(o.size);
    c.setRandomValues();
    float cons = float(rand()) / float(RAND_MAX);
    auto start = omp_get_wtime();
    saxpy(c.m_size, cons, c.m_a, c.m_b);
    auto end = omp_get_wtime();
    verify(c.m_size, cons, c.m_a, c.m_b, c.m_verification);
    std::cout << "Execution Time : " << std::fixed
        << end - start << std::setprecision(5);
    std::cout << " sec " << std::endl;
    return 0;
}
```

Συμβ. 53: SAXPY: class Containers

```
struct Containers {
    explicit Containers(size_t containers_size);
    ~Containers();

    size_t m_size;
    float *m_a;
    float *m_verification;
    float *m_b;
};

Containers::Containers(size_t containers_size)
    : m_size(containers_size) {
    srand(time(nullptr));
    m_a = new float[containers_size];
    m_verification = new float[containers_size];
    m_b = new float[containers_size];
}

Containers::~Containers() {
    delete []m_a;
    delete []m_b;
    delete []m_verification;
}

Containers::setRandomValues() {
    fill_random_arr(m_a, m_size);
    fill_random_arr(m_b, m_size);
}
```

Συμβ. 54: SAXPY: verify

```
static void verify(size_t size, float c, float *a, float *b,
                   float *verification) {
    for (size_t i = 0; i < size; ++i) {
        if (abs(c * a[i] + verification[i] - b[i]) >= 10e-6) {
            std::cout << "Failed index: " << i <<
             ". " << c * a[i] + verification[i] <<
             " =! " << b[i] << std::endl;
            exit(1);
        }
    }
}
```

Συμβ. 55: SAXPY: fill_random_arr

```
static void fill_random_arr(float *arr, size_t size) {
    for (size_t k = 0; k < size; ++k) {
        arr[k] = (float)(rand()) / RAND_MAX;
    }
}
```

4.3.3 Σειριακή εκτέλεση

Η υλοποίηση της σειριακής παραλλαγής της συνάρτησης saxpy περιλαμβάνει έναν επαναληπτικό βρόγχο στον οποίο γίνεται ο υπολογισμός για κάθε στοιχείο των πινάκων.

Συμβ. 56: SAXPY: Σειριακή υλοποίηση

```
void saxpy(size_t n, float a, const float *x, float *y) {
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

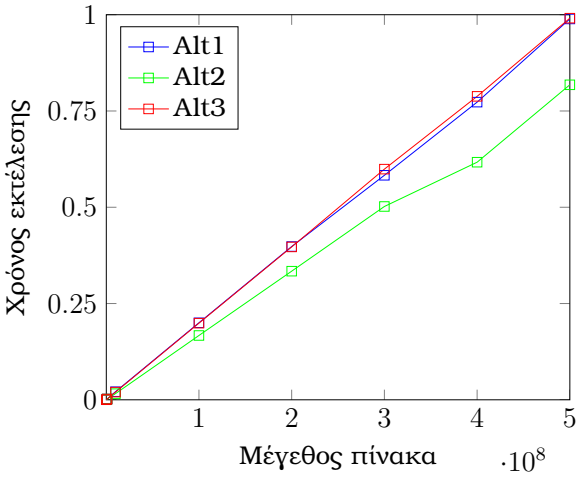
Η μεταγλώττιση έγινε με τους παρακάτω τρόπους και οι χρόνοι εκτέλεσης καταγράφονται στους πίνακες που ακολουθούν. Ο δεύτερος τρόπος μεταγλώττισης περιλαμβάνει διανυσματικοποίηση(*-ftree-vectorize*) ενώ η πρώτη όχι(*-fno-tree-vectorize*). Στη τρίτη περίπτωση, δεν ορίζεται κάποια σχετική οδηγία διανυσματικοποίησης, γιαυτο ο μεταγλωττιστής θα χρησιμοποιήσει την προκαθορισμένη επιλογή για -O2, δηλαδή μεταγλώττιση χωρίς διανυσματικοποίηση.

Πίνακας 9: SAXPY: Επιλογές μεταγλώττισης Alt1, Alt2, Alt3

Label	**Options**
Alt1	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt2	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt3	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 10: SAXPY: Αποτελέσματα Alt1, Alt2 και Alt3

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt1	**Alt2**	**Alt3**
100000	0.0004	0.0001	0.0001
1000000	0.002	0.002	0.002
10000000	0.021	0.016	0.020
100000000	0.200	0.167	0.199
200000000	0.398	0.334	0.397
300000000	0.583	0.502	0.599
400000000	0.773	0.617	0.788
500000000	0.989	0.818	0.991



Σχήμα 12: SAXPY: Σύγκριση Alt1, Alt2, Alt3

Μέγεθος	Επιτάχυνση (%)
100000	-
1000000	-
10000000	23
100000000	17
200000000	16
300000000	13.9
400000000	20

Πίνακας 11: SAXPY: Ποσοστιαία σύγκριση Alt1, Alt2

4.3.3.1 Παρατηρήσεις

Η επιλογή για διανυσματικοποίηση κατά τη μεταγλώττιση επιφέρει περίπου 20% βελτίωση στις χρονικές επιδόσεις των εκτελέσεων με διαφορετικά μεγέθη διανύσματος. Ακόμη, η μεταγλώττιση με *-fno-tree-vectorize* έχει τις ίδιες επιδόσεις με την παραλλαγή Alt3, πράγμα που επιβεβαιώνει ότι η -O2 δεν υπονοεί αυτόματη διανυσματικοποίηση. Τα αποτελέσματα αυτά θα χρησιμοποιηθούν για συγκρίσεις με τις παραλλαγές που θα ακολουθήσουν.

4.3.4 Παραλλαγή με οδηγία parallel for

Η πρώτη υλοποίηση παραλλαγής με παραλληλισμό της συνάρτησης saxpy περιλαμβάνει τον επαναληπτικό βρόγχο ενσωματωμένο στην οδηγία *parallel for* στον οποίο γίνεται ο υπολογισμός για κάθε στοιχείο των πινάκων. Τα αποτελέσματα φαίνονται στον πίνακα που ακολουθεί.

Συμβ. 57: SAXPY: Υλοποίηση με parallel for

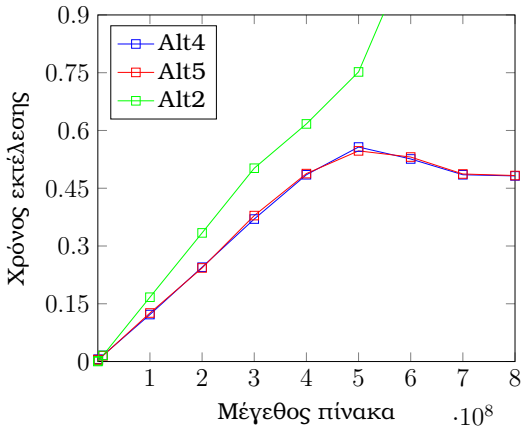
```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp parallel for
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 12: SAXPY: Επιλογές μεταγλώττισης Alt4, Alt5

Label	**Options**
Alt4	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt5	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 13: SAXPY: Αποτελέσματα Alt4 και Alt5

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt4	**Alt5**
100000	0.005	0.005
1000000	0.006	0.003
10000000	0.016	0.015
100000000	0.122	0.126
200000000	0.245	0.243
300000000	0.370	0.379
400000000	0.485	0.488
500000000	0.557	0.547
600000000	0.526	0.531
700000000	0.485	0.487
800000000	0.482	0.483



Σχήμα 13: SAXPY: Σύγκριση Alt2, Alt4, Alt5

4.3.4.1 Παρατηρήσεις

Με τη χρήση της οδηγίας *pragma omp parallel for* επετεύχθη μείωση του χρόνου εκτέλεσης του αλγορίθμου σε σύγκριση με τη σειριακή υλοποίηση. Δεν προκύπτει καμία διαφοροποίηση της μεταγλώττισης με εντολή διανυσματικοποίησης ή χωρίς. Σύμφωνα ωστόσο με τον ορισμό του false sharing, υπάρχει πιθανότητα περαιτέρω βελτίωσης της παραλλαγής με οδηγία parallel for, κάτι που εξετάζεται στην επόμενη ενότητα.

4.3.5 Παραλλαγές με χρήση οδηγίας *SIMD*

Η συγκεκριμένη ενότητα καθώς και ορισμένες που ακολουθούν αφορούν την επίλυση του προβλήματος *SAXPY* με χρήση της οδηγίας διανυσματικοποίησης **SIMD**. Όπως προαναφέρθηκε, η οδηγία δεν έχει ως στόχο την παραλληλοποίηση τμήματος κώδικα, αλλά την ταυτόχρονη εκτέλεση εντολών ως μία εντολή **SIMD**. Στη περίπτωση του g++, ο μεταγλωττιστής εφαρμόζει διανυσματικοποίηση αυτόματα όταν χρησιμοποιείται η επιλογή -O3. Για το λόγο αυτό στα προηγούμενα παραδείγματα χρησιμοποιήθηκε επιλογή μεταγλώττισης -O2 με ταυτόχρονη χρήση των εντολών **-fno-tree-vectorize** και **ftree-vectorize** που επιτρέπουν στο χρήστη να επιλέγει μεταγλώττιση με διανυσματικοποίηση ή χωρίς. Παράλληλα, χρησιμοποιούνται οι εντολές **-fno-inline** και **-fopt-info-vec**. Η πρώτη απαγορεύει στον μεταγλωττιστή τη δημιουργία inline συναρτήσεων, ενώ η δεύτερη δημιουργεί ένα αρχείο με χρήσιμα μηνύματα κατά τη διάρκεια της μεταγλώττισης.

4.3.5.1 Παραλλαγή με *omp simd*

Η εντολή *pragma omp simd* αποσκοπεί στη διανυσματικοποίηση του τμήματος κώδικα που ακολουθεί, χωρίς ωστόσο να γίνεται διαμοιρασμός των επαναλήψεων του βρόγχου σε διαφορετικά νήματα όπως θα γίνόταν με την οδηγία *pragma omp parallel for simd*.

Συμβ. 58: SAXPY: Υλοποίηση με omp simd

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 14: SAXPY: Επιλογές μεταγλώττισης Alt8, Alt9, Alt10

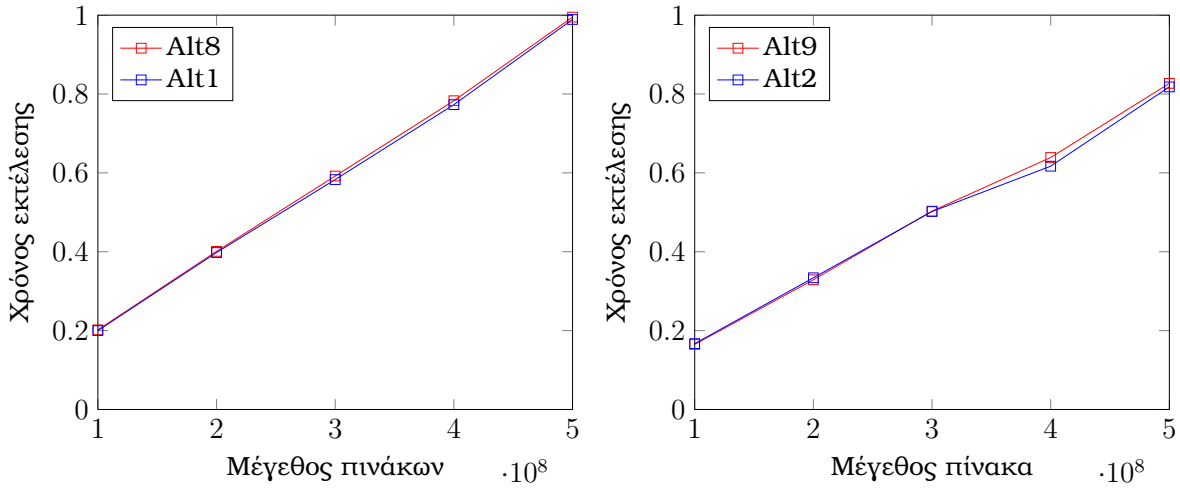
Label	Options
Alt8	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt9	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt10	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 15: SAXPY: Αποτελέσματα Alt8, Alt9 και Alt10

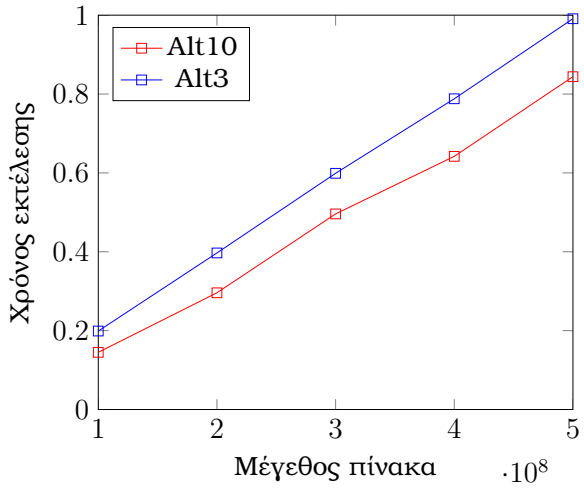
Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		
	Alt8	**Alt9**	**Alt10**
100000	0.001	0.001	0.001
1000000	0.002	0.002	0.002
10000000	0.021	0.017	0.017
100000000	0.202	0.165	0.145
200000000	0.401	0.329	0.296
300000000	0.592	0.503	0.496
400000000	0.783	0.639	0.642
500000000	0.995	0.827	0.844

Πίνακας 16: SAXPY: Alt8: Αναφορά επιτυχίας διανυσματικοποίησης

Επιλογή μεταγλώττισης	Σειριακή	OpenMP - omp simd
-fno-tree-vectorize	Όχι	Όχι
-ftree-vectorize	Ναι	Ναι
None	Όχι	Ναι



Σχήμα 14: SAXPY: Σύγκριση αποτελεσμάτων Alt1-Alt8, Alt2-Alt9



Σχήμα 15: SAXPY: Σύγκριση Alt3, Alt10

4.3.5.1.1 Παρατηρήσεις

Από τα παραπάνω διαγράμματα διαφαίνεται ότι η μεταγλώττιση με επιλογή *-fno-tree-vectorize* και *-ftree-vectorize* παρακάμπτει την οδηγία omp simd και εκτελείται ως σειριακή, με διανυσματικοποίηση ή χωρίς, ανάλογα με την επιλογή. Στη περίπτωση που δε δοθεί η επιλογή ωστόσο, τότε διανυσματικοποίηση εφαρμόζεται μέσω του *OpenMP* και της οδηγίας *omp simd* αν υπάρχει.

4.3.5.2 Παραλλαγή με *omp parallel for simd*

Στην παραλλαγή αυτής της ενότητας χρησιμοποιείται ο συνδυασμός παραλληλοποίησης μέσω της οδηγίας *parallel for* με διανυσματικοποίηση μέσω *simd*.

Συμβ. 59: SAXPY: parallel for simd

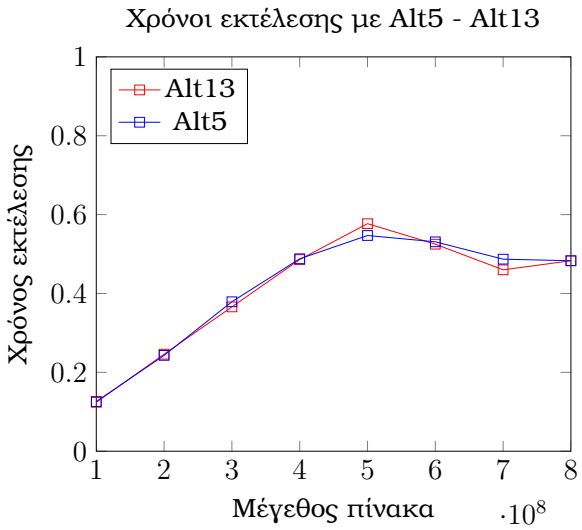
```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp parallel for simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 17: SAXPY: Επιλογές μεταγλώττισης Alt11, Alt12, Alt13

Label	**Options**
Alt11	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt12	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt13	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 18: SAXPY: Αποτελέσματα Alt11, Alt12 και Alt13

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt11	**Alt12**	**Alt13**
100000	0.006	0.002	0.005
1000000	0.006	0.002	0.004
10000000	0.015	0.016	0.016
100000000	0.129	0.123	0.124
200000000	0.247	0.251	0.246
300000000	0.368	0.368	0.366
400000000	0.489	0.486	0.486
500000000	0.578	0.576	0.577
600000000	0.515	0.458	0.525
700000000	0.496	0.496	0.460
800000000	0.487	0.500	0.483



Σχήμα 16: SAXPY: Σύγκριση Alt5, Alt13

4.3.5.2.1 Παρατηρήσεις

Από τις παραπάνω εκτελέσεις του αλγόριθμου προκύπτει το συμπέρασμα ότι η οδηγία διανυσματικοποίησης μέσω simd δεν λαμβάνεται υπόψη όταν εφαρμόζεται σε συνδυασμό με την οδηγία parallel for.

4.3.5.3 Παραλλαγή με *omp declare simd uniform*

Υλοποίηση παραλλαγής με χρήση της φράσης *uniform*. Θεωρητικά δεν υπάρχει κέρδος στις επιδόσεις του αλγορίθμου σε σχέση με της προηγούμενες παραλλαγές. Η εναλλακτική αυτή χρησιμοποιείται για την επαλήθευση της διανυσματικοποίησης μέσω *OpenMP*.

Συμβ. 60: SAXPY: declare simd uniform

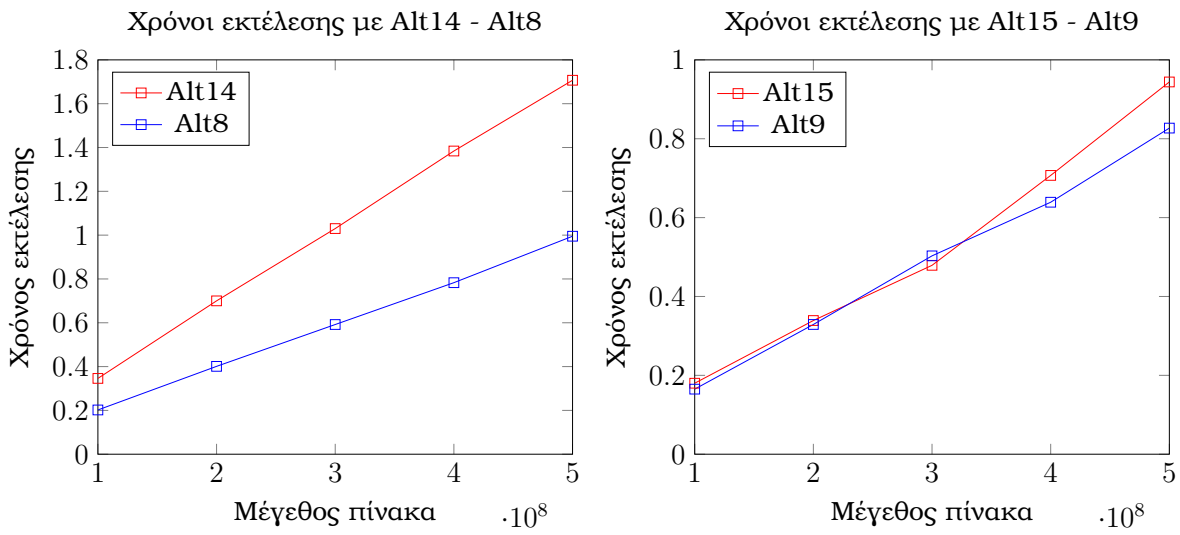
```
#pragma omp declare simd uniform(a)
float do_work(float a, float b, float c)
{
    return a * b + c;
}

void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = do_work(a, x[i], y[i]);
    }
}
```

Πίνακας 19: SAXPY: Επιλογές μεταγλώττισης Alt14, Alt15, Alt16

Label	Options
Alt14	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt15	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt16	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 20: SAXPY: Αποτελέσματα Alt14, Alt15 και Alt16

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt14	**Alt15**	**Alt16**
100000	0.0003	0.0001	0.0003
1000000	0.003	0.002	0.002
10000000	0.035	0.018	0.018
100000000	0.346	0.180	0.179
200000000	0.700	0.339	0.318
300000000	1.030	0.479	0.469
400000000	1.384	0.707	0.637
500000000	1.707	0.944	0.911



Σχήμα 17: SAXPY: Σύγκριση αποτελεσμάτων Alt8-Alt14, Alt9-Alt15

4.3.5.3.1 Παρατηρήσεις

Η παραλλαγή της παραγράφου συγκρίνεται με τις υλοποιήσεις Alt8 Alt9 που προαναφέρθηκαν. Η μοναδική διαφορά ανάμεσά τους, είναι ότι σε αυτή την παράγραφο η πράξη της πρόσθεσης δύο στοιχείων γίνεται μέσω της ρουτίνας do_work που καλείται μέσα στο βρόγχο επανάληψης. Ως συνέπεια, η μικρή αύξηση που παρατηρείται, οφείλεται πιθανότατα στο χρόνο που απαιτείται για την κλήση της ρουτίνας αυτής.

4.3.5.4 Παραλλαγή με *omp declare simd uniform notinbranch*

Στη περίπτωση που ακολουθεί, μελετάται η υλοποίηση με χρήση της φράσης*notinbranch* και το κέρδος που μπορεί να επιφέρει σε σύγκριση με τη προηγούμενη παραλλαγή.

Συμβ. 61: SAXPY: declare simd uniform notinbranch

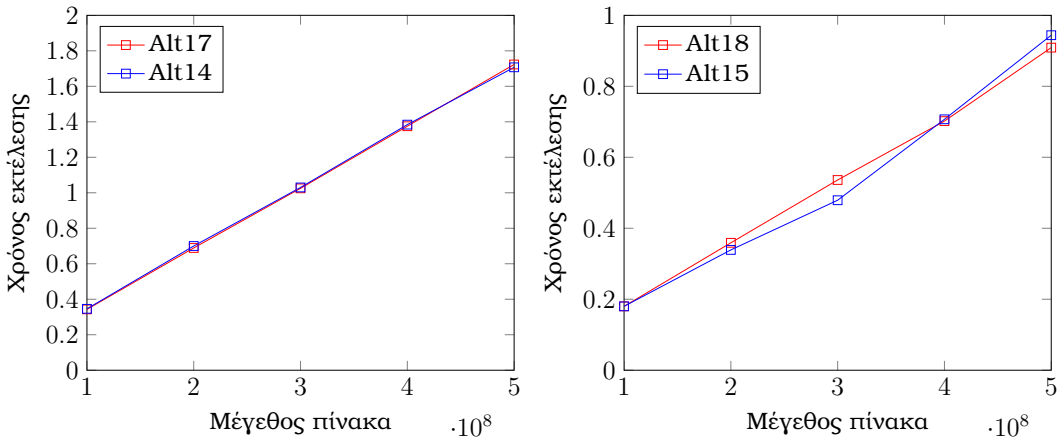
```
#pragma omp declare simd uniform(a) notinbranch
float do_work(float a, float b, float c)
{
    return a * b + c;
}

void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp simd
    for (size_t i = 0; i < n; ++i) {
        y[i] = do_work(a, x[i], y[i]);
    }
}
```

Πίνακας 21: SAXPY: Επιλογές μεταγλώττισης Alt17, Alt18, Alt19

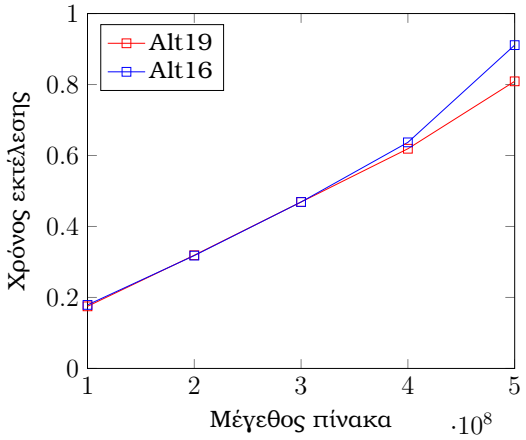
Label	**Options**
Alt17	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt18	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt19	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 22: SAXPY: Αποτελέσματα Alt17, Alt18 και Alt19

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt17	**Alt18**	**Alt19**
100000	0.001	0.001	0.001
1000000	0.003	0.002	0.002
10000000	0.035	0.018	0.018
100000000	0.343	0.180	0.175
200000000	0.689	0.359	0.319
300000000	1.025	0.536	0.469
400000000	1.375	0.702	0.619
500000000	1.723	0.909	0.809



Σχήμα 18: SAXPY: Σύγκριση αποτελεσμάτων Alt17-Alt14, Alt18-Alt15



Σχήμα 19: SAXPY: Σύγκριση αποτελεσμάτων Alt16-Alt19

4.3.6 Παραλλαγές με *offloading*

Η ομάδα παραλλαγών αυτής της ενότητας, αφορά τη μεταφορά του αλγορίθμου σε άλλο μέσο για την επίλυση του. Το βασικότερο τμήμα του *SAXPY*, εκτελείται στη μονάδα επεξεργασίας κάρτας γραφικών - *GPU*. Στα πλαίσια της μεταφοράς, συμπεριλαμβάνεται και η μεταφορά των μεταβλητών από τη μνήμη της κεντρικής μονάδας στη μνήμη της κάρτας γραφικών. Αυτή η μεταφορά γίνεται με διάφορους τρόπους και τεχνικές που αναφέρονται στις επόμενες παραγράφους. Για την επιτυχία της μεταγλώττισης προβλημάτων που γίνεται η χρήση της **GPU**, είναι απαραίτητη η χρήση των επιλογών -fno-stack-protector -foffload=nvptx-none="-O2.

4.3.6.1 Παραλλαγή με *target map*

Στη συγκεκριμένη παραλλαγή γίνεται απλή μεταφορά του κώδικα και των μεταβλητών στην κάρτα γραφικών για την εκτέλεση του *SAXPY* σε αυτή. Πρόκειται για σειριακή υλοποίηση του προβλήματος, η οποία έχει μεταφερθεί στη μονάδα επεξεργασίας της κάρτας γραφικών.

Συμβ. 62: SAXPY: target map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 23: SAXPY: Επιλογές μεταγλώττισης Alt20, Alt21

Label	Options
Alt20	-fopt-info-vec=info.log -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt21	-fopt-info-vec=info.log -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 24: SAXPY: Αποτελέσματα Alt20, Alt21

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt20	**Alt21**
100000	0.881	0.873
1000000	1.200	1.169
10000000	3.848	3.752
100000000	29.984	29.937
200000000	59.055	59.894

4.3.6.1.1 Παρατηρήσεις

Παρόλο που η συγκεκριμένη παραλλαγή προσομοιώνει σειριακή εκτέλεση στη μονάδα επεξεργασίας της κάρτας γραφικών, η δημιουργία ενός νέου περιβάλλοντος δεδομένων και η εκτέλεσή του στη μονάδα επεξεργασίας γραφικών μέσω *OpenMP*, αποτελεί μια διαδικασία, πολύ πιο χρονοβόρα από τη σειριακή εκτέλεση στη *CPU*.

4.3.6.2 Παραλλαγή με *target simd map*

Η προηγούμενη παραλλαγή, εμπλουτίζεται με τη φράση *simd*, όπως φαίνεται στον πίνακα που ακολουθεί:

Συμβ. 63: SAXPY: target simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target simd map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 25: SAXPY: Επιλογές μεταγλώττισης Alt22, Alt23, Alt24

Label	**Options**
Alt22	-fopt-info-vec=builds/alt22.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt22
Alt23	-fopt-info-vec=builds/alt23.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt23
Alt24	-fopt-info-vec=builds/alt24.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt24

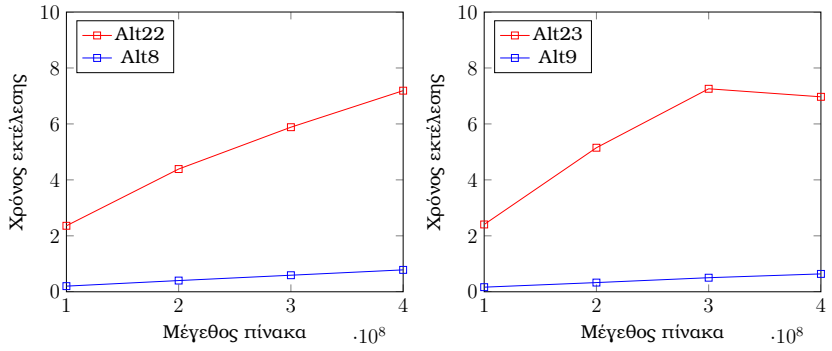
Πίνακας 26: SAXPY: Αποτελέσματα Alt22, Alt23 και Alt24

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt22	**Alt23**	**Alt24**
100000	0.888	0.809	0.837
1000000	0.841	0.837	0.833
10000000	1.018	1.018	0.997
100000000	2.359	2.406	2.392
200000000	4.387	5.149	5.157
300000000	5.883	7.256	7.494
400000000	7.189	6.969	7.155

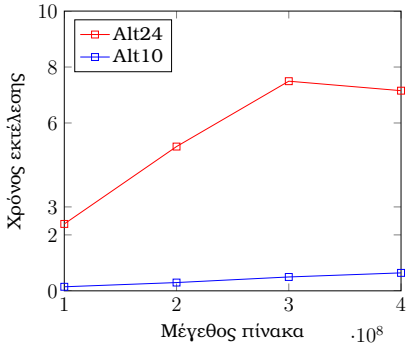
4.3.6.2.1 Παρατηρήσεις

Συγκριτικά με τις αντίστοιχες παραλλαγές υπολογισμού στη *CPU*, οι χρονικές επιδόσεις σε *GPU* είναι χειρότερες όπως φαίνεται στα διαγράμματα που προηγήθηκαν. Από το συνολικό χρόνο εκτέλεσης της *saxpy* ρουτίνας, αξίζει να αναφερθεί ο χρόνος που

απαιτείται για την αντιγραφή των πινάκων στη μνήμη της *GPU*. Για ορισμένα μεγέθη προβλήματος το ποσοστό αυτό φτάνει στο 50%. Τέλος, παρατηρείται οτι σε σύγκριση με τις παραλλαγές Alt20 - Alt21, η μοναδική διαφορά είναι η φράση simd. Παρόλα αυτά εμφανίζεται σημαντική μείωση στους χρόνους εκτέλεσης της παρούσας παραλλαγής.



Σχήμα 20: SAXPY: Σύγκριση αποτελεσμάτων Alt8-Alt22, Alt9-Alt23



Σχήμα 21: SAXPY: Σύγκριση αποτελεσμάτων Alt19-Alt24

Πίνακας 27: SAXPY: Alt22: Ποσοστά χρόνου εργασιών με offloading

Μέγεθος προβλήματος	**Ποσοστό (%)**		
	saxpy	**memcpy DtoH**	**memcpy HtoD**
100000	79.07	14.05	6.87
1000000	77.42	15.27	7.31
10000000	75.25	16.32	8.42
100000000	62.62	20.76	16.63
200000000	60.29	23.67	16.04
300000000	50.38	35.48	14.14
400000000	70.80	19.26	9.94

4.3.6.3 Παραλλαγή με *target parallel for*

Σε αυτή την παραλλαγή εφαρμόζεται η οδηγία *pragma omp parallel for* στη συσκευή στόχου, όπως φαίνεται στο παρακάτω τμήμα κώδικα.

Συμβ. 64: SAXPY: target parallel for map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target parallel for map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 28: SAXPY: Επιλογές μεταγλώττισης Alt25, Alt26

Label	Options
Alt25	-fopt-info-vec=builds/alt24.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt24
Alt26	-fopt-info-vec=builds/alt25.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt25

Πίνακας 29: SAXPY: Αποτελέσματα Alt25, Alt26

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt25	**Alt26**
100000	1.103	0.825
1000000	0.869	0.912
10000000	1.264	1.224
100000000	4.776	4.657
200000000	9.058	8.983
300000000	13.10	13.145

4.3.6.3.1 Παρατηρήσεις

Από τον πίνακα αποτελεσμάτων, φαίνεται πως η χρήση της οδηγίας *parallel for* στη συσκευή στόχου, καταλήγει σε εκτέλεση με χαμηλές επιδόσεις.

4.3.6.4 Παραλλαγή με *target parallel for simd*

Η προηγούμενη παραλλαγή, εμπλουτίζεται με τη φράση *simd*, όπως φαίνεται στον πίνακα που ακολουθεί:

Συμβ. 65: SAXPY: target parallel for simd

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target parallel for simd map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 30: SAXPY: Επιλογές μεταγλώττισης Alt27, Alt28, Alt29

Label	**Options**
Alt27	-fopt-info-vec=builds/alt26.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt26
Alt28	-fopt-info-vec=builds/alt27.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt27
Alt29	-fopt-info-vec=builds/alt28.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt29

Πίνακας 31: SAXPY: Αποτελέσματα Alt27, Alt28 και Alt29

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt27	**Alt28**	**Alt29**
100000	0.940	0.872	0.823
1000000	0.867	0.850	0.827
10000000	0.885	0.917	0.880
100000000	1.471	1.479	1.456
200000000	2.001	2.118	2.103
300000000	4.472	3.135	2.614
400000000	3.218	3.174	3.183
500000000	3.798	4.061	3.877
600000000	4.392	4.709	4.275

4.3.6.4.1 Παρατηρήσεις

Όπως και σε *προηγούμενη περίπτωση*, η εισαγωγή της φράσης simd αποτέλεσε παράγοντα μείωσης του χρόνου εκτέλεσης σε σύγκριση με την *υλοποίηση* χωρίς τη φράση αυτή. Οι χρόνοι εκτέλεσης ωστόσο παραμένουν υψηλοί.

4.3.6.5 Παραλλαγή με *target teams map*

Συμβ. 66: SAXPY: target teams map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 32: SAXPY: Επιλογές μεταγλώττισης Alt30, Alt31

Label	Options
Alt30	-fopt-info-vec=builds/alt30.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt30
Alt31	-fopt-info-vec=builds/alt31.log -O2 -fno-inline -fno-stack-protector -ftree-vectorize -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt31

4.3.6.5.1 Παρατηρήσεις

Κατά τη διάρκεια πειραματικών ελέγχων με διάφορα μεγέθη πινάκων, όπως έγινε και τις προηγούμενες παραγράφους, παρατηρήθηκε λάθος υπολογισμός του τελικού διανύσματος. Ως συνέπεια, δεν έγινε καταγραφή των χρόνων επίλυσης του προβλήματος, για διαφορετικά μεγέθη. Το πρόβλημα οφείλεται στο φαινόμενο *race condition* καθώς στη συγκεκριμένη υλοποίηση, σε νήματα διαφορετικών ομάδων λαμβάνονται ίδια i.

4.3.6.6 Παραλλαγή με *target teams distribute map*

Συμβ. 67: SAXPY: target teams distribute map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 33: SAXPY: Επιλογές μεταγλώττισης Alt32, Alt33

Label	Options
Alt32	-fopt-info-vec=builds/alt32.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt32
Alt33	-fopt-info-vec=builds/alt33.log -O2 -fno-inline -fno-stack-protector -ftree-vectorize -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt33

Πίνακας 34: SAXPY: Αποτελέσματα Alt32, Alt33

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		**Ποσοστό συνολικού χρόνου (%)**		
	Alt32	**Alt33**	**saxpy**	**memcpy DtoH**	**memcpy HtoD**
100000	0.869	0.837	75.67	16.88	7.45
1000000	0.830	0.835	64.30	24.58	11.12
10000000	0.974	0.936	62.09	25.46	12.44
100000000	1.838	1.857	57.08	28.79	14.12
200000000	2.867	3.353	49.92	25.81	24.26
300000000	3.911	3.723	54.28	30.60	15.11
400000000	4.911	4.881	56.33	29.41	14.26
500000000	5.873	5.948	55.45	30.14	14.41

4.3.6.6.1 Παρατηρήσεις

Η οδηγία *distribute* δεν αντιστοιχίζεται επακριβώς σε κάποια οδηγία του *OpenMP* για εκτέλεση στη *CPU*. Έτσι, δεν γίνεται κάποια σύγκριση της συγκεκριμένης παραλλαγής. Ο προηγούμενος πίνακας ωστόσο δείχνει τις χαμηλές επιδόσεις της συγκεκριμένης παραλλαγής συγκριτικά με εκτέλεση στη κεντρική μονάδα επεξεργασίας. Ακόμα, παρατηρείται ότι στις περισσότερες περιπτώσεις μεγεθών απαιτείται σχεδόν ο μισός χρόνος της συνολικής εκτέλεσης για τη μεταφορά των δεδομένων ανάμεσα στις δυο συσκευές.

4.3.6.7 Παραλλαγή με *target teams distribute parallel for map*

Συμβ. 68: SAXPY: target teams distribute parallel for

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute parallel for map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 35: SAXPY: Επιλογές μεταγλώττισης Alt34, Alt35

Label	Options
Alt34	-fopt-info-vec=builds/alt34.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt34
Alt35	-fopt-info-vec=builds/alt35.log -O2 -fno-inline -fno-stack-protector -ftree-vectorize -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt35

Πίνακας 36: SAXPY: Αποτελέσματα Alt34, Alt35

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt34	**Alt35**
100000	0.818	0.858
1000000	0.819	0.857
10000000	0.917	0.874
100000000	1.455	1.489
200000000	2.279	2.612
300000000	3.187	2.803
400000000	3.458	3.459
500000000	4.598	4.629

4.3.6.7.1 Παρατηρήσεις

Ο ταυτόχρονος διαμοιρασμός των εργασιών βρόχου σε ομάδες νημάτων και παραλληλισμός των εργασιών μεταξύ των νημάτων αυτών, οδηγεί σε μια βελτιωμένη έκδοση της επίλυσης του προβλήματος με χρήση της μονάδας επεξεργασίας γραφικών. Ωστόσο, είναι χρήσιμη η δημιουργία παρόμοιας παραλλαγής, άλλα με εφαρμογή διανυσματικοποίησης και ο συνδυασμός των δυο.

4.3.6.8 Παραλλαγή με *target teams distribute simd map*

Συμβ. 69: SAXPY target teams distribute simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute simd map(from: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 37: SAXPY: Επιλογές μεταγλώττισης Alt36, Alt37, Alt38

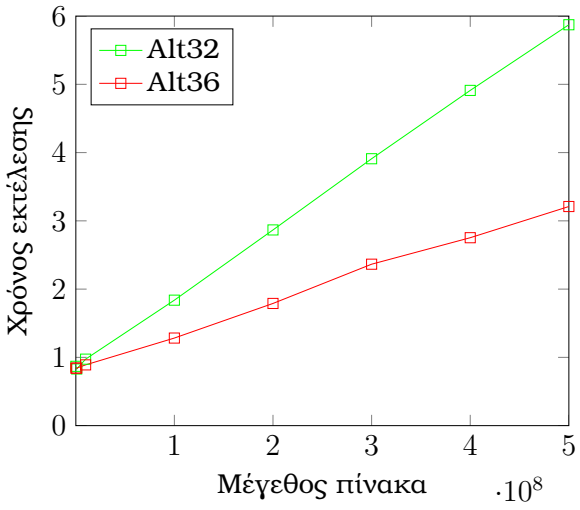
Label	**Options**
Alt36	-fopt-info-vec=builds/alt36.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt36
Alt37	-fopt-info-vec=builds/alt37.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt37
Alt38	-fopt-info-vec=builds/alt38.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt38

Πίνακας 38: SAXPY: Αποτελέσματα Alt36, Alt37 και Alt38

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt36	**Alt37**	**Alt38**
100000	0.842	0.865	0.849
1000000	0.843	0.883	0.830
10000000	0.890	0.876	0.899
100000000	1.283	1.360	1.295
200000000	1.790	1.842	1.794
300000000	2.365	2.366	2.239
400000000	2.753	2.806	2.773
500000000	3.210	3.244	3.225
600000000	3.689	3.553	3.770

4.3.6.8.1 Παρατηρήσεις

Η εισαγωγή διανυσματικοποίησης επιφέρει βελτίωση σε σύγκριση *με την αντίστοιχη υλοποίηση* χωρίς αυτήν.



Σχήμα 22: SAXPY: Σύγκριση Alt36, Alt32

Μέγεθος	Επιτάχυνση (%)
100000	–
1000000	–
10000000	8.6
100000000	30.2
200000000	37.6
300000000	39.5
400000000	43.9
500000000	45.3

Πίνακας 39: SAXPY: Ποσοστιαία σύγκριση Alt36, Alt32

4.3.6.9 Παραλλαγή με *target teams distribute parallel for simd map*

Στην τελευταία παραλλαγή του προβλήματος, εφαρμόζονται όλες οι διαθέσιμες οδηγίες παραλληλοποίησης στην μονάδα επεξεργασίας γραφικών μέσω του *OpenMP*. Τα αποτελέσματα της εκτέλεσης για διάφορα μεγέθη ακολουθούν.

Συμβ. 70: SAXPY teams distribute parallel for simd map

```
void saxpy(size_t n, float a, const float *x, float *y) {
#pragma omp target teams distribute parallel for simd\
                        map(tofrom: y[0:n]) map(to: x[0:n])
    for (size_t i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Πίνακας 40: SAXPY: Επιλογές μεταγλώττισης Alt39, Alt40, Alt41

Label	Options
Alt39	-fopt-info-vec=builds/alt39.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt39
Alt40	-fopt-info-vec=builds/alt40.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt40
Alt41	-fopt-info-vec=builds/alt41.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt41

Πίνακας 41: SAXPY: Αποτελέσματα Alt39, Alt40 και Alt41

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**			**Ποσοστό συνολικού χρόνου (%)**		
	Alt39	**Alt40**	**Alt41**	**saxpy**	**memcpy HtoD**	**memcpy DtoH**
100000	0.908	0.828	0.844	14.29	59.19	26.51
1000000	0.829	0.848	0.861	11.65	61.26	27.09
10000000	0.885	0.862	0.864	3.74	64.13	32.14
100000000	1.268	1.269	1.310	2.83	65.14	32.03
200000000	1.747	1.737	1.776	2.69	65.38	31.93
300000000	2.313	2.259	2.183	2.67	66.60	30.73
400000000	2.585	2.644	2.856	2.66	66.50	30.83
500000000	3.169	3.128	3.060	2.68	65.55	31.78
600000000	3.592	3.848	3.580	2.69	65.45	31.85

4.3.6.9.1 Παρατηρήσεις

Σύμφωνα με το συνοπτικό πίνακα καταγραφής χρόνου εκτέλεσης του προβλήματος για διαφορετικά μεγέθη, προκύπτει ότι ο χρόνος εκτέλεσης του προβλήματος είναι ο καλύτερος σε σχέση με όλες της προηγούμενες προσπάθειες εκτέλεσης στη μονάδα επεξεργασίας γραφικών. Παρόλα αυτά, η απόδοση είναι πολύ χαμηλότερη σε σχέση με οποιαδήποτε προσπάθεια επίλυσης του προβλήματος στην *CPU*. Ωστόσο, με τη χρήση του **nvprof**, προκύπτει ότι το σημαντικότερο ποσοστό του συνολικού χρόνου εκτέλεσης, αποτελεί η μετακίνηση των μεταβλητών από το περιβάλλον δεδομένων της κεντρικής μονάδας, σε αυτό της κάρτας γραφικών. Συγκεκριμένα, το 97% του χρόνου καταναλώνεται από την οδηγία *map*. Με άλλα λόγια, αν θεωρητικά δεν απαιτείται η μεταφορά του διανύσματος μεγέθους 6e8 στο περιβάλλον της συσκευής στόχου, τότε ο χρόνος εκτέλεσης του προβλήματος θα ήταν $3.06 * 0.0269 = 0.082$ δευτερόλεπτα

4.3.6.10 Παραλλαγή με *target teams distribute parallel for simd is_device_ptr*

Για την επίλυση του προβλήματος της χρονοβόρας διαδικασίας μεταφοράς δεδομένων ανάμεσα στις δυο συσκευές (*host and device*) το OpenMP προσφέρει τη δυνατότητα στο χρήση της απευθείας δημιουργίας και διαχείρισης της μνήμης σε περιβάλλον εργασίας της κάρτας γραφικών. Έτσι, ο αλγόριθμος θεωρητικά θα πρέπει να εκτελείται σε πολύ μικρότερο χρονικό διάστημα συγκριτικά με την παρατήρηση της προηγούμενης παραγράφου.

Συμβ. 71: SAXPY: teams distribute parallel for simd is_device_ptr

```
void saxpy(size_t n, float a, const float *x, float *y) {
    #pragma omp target teams distribute parallel for simd is_device_ptr(y, x)
        for (size_t i = 0; i < n; ++i) {
            y[i] = a * x[i] + y[i];
        }
}
```

Πίνακας 42: SAXPY: Επιλογές μεταγλώττισης Alt42, Alt43, Alt44

Label	Options
Alt42	-fopt-info-vec=builds/alt39.log -O2 -fno-tree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-tree-vectorize -fno-inline" -fopenmp -o ./builds/Alt39
Alt43	-fopt-info-vec=builds/alt40.log -O2 -ftree-vectorize -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -ftree-vectorize -fno-inline" -fopenmp -o ./builds/Alt40
Alt44	-fopt-info-vec=builds/alt41.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt41

4.3.6.10.1 Παρατηρήσεις

Όπως ήταν αναμενόμενο, η εκτέλεση του προβλήματος με δέσμευση μνήμης απευθείας στο περιβάλλον δεδομένων της κάρτας γραφικών, οδήγησε στην κατακόρυφη πτώση του χρόνου εκτέλεσης του προβλήματος. Ο χρόνος είναι συγκρίσιμος με τη θεωρητική προσέγγιση των παρατηρήσεων της προηγούμενης παραγράφου. Ακόμα, η συγκεκριμένη παραλλαγή αποτελεί την καλύτερη λύση σε ότι αφορά τις χρονικές επιδόσεις επίλυσης του προβλήματος.

Πίνακας 43: SAXPY: Αποτελέσματα Alt42, Alt43 και Alt44

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt42	**Alt43**	**Alt44**
100000	0.006	0.006	0.006
1000000	0.006	0.006	0.006
10000000	0.007	0.007	0.007
100000000	0.018	0.019	0.019
200000000	0.031	0.031	0.031
300000000	0.042	0.043	0.043
400000000	0.054	0.054	0.054
500000000	0.067	0.067	0.067
600000000	0.078	0.079	0.078

Πίνακας 44: SAXPY: nvprof - 100000

Time(%)	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
65.85%	814.6us	2	407.3us	392.4us	422.1us	fill_random_arr
33.49%	414.3us	1	414.3us	414.3us	414.3us	saxpy
0.42%	5.22us	3	1.74us	1.6us	1.85us	CUDA memcpy HtoD
0.24%	2.9us	1	2.91us	2.91us	2.91ενυς	CUDA memcpy DtoH

Πίνακας 45: SAXPY: nvprof - 1000000

Time(%)	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
77.8%	1.9ms	2	947.6us	943.6us	951.9us	fill_random_arr
21.9%	533.0us	1	533us	532.99us	533us	saxpy
0.21%	5.1us	3	1.69us	1.44us	1.92us	CUDA memcpy HtoD
0.13%	3.1us	1	3.07us	3.07us	3.07us	CUDA memcpy DtoH

Πίνακας 46: SAXPY: nvprof - 10000000

Time(%)	Time	Calls	Avg	Min	Max	Name
87.1%	11.6ms	2	5.8us	5.8us	5.8us	fill_random_arr
12.9%	1.7ms	1	1.7ms	1.7ms	1.7ms	saxpy
0.04%	4.9us	3	1.6us	1.5us	1.7us	CUDA memcpy HtoD
0.02%	3.0us	1	3.0us	3.0us	3.0us	CUDA memcpy DtoH

Πίνακας 47: SAXPY: nvprof - 100000000

Time(%)	Time	Calls	Avg	Min	Max	Name
89.23%	108.3ms	2	54.1ms	54.11ms	54.1ms	fill_random_arr
10.76%	13.1ms	1	13.1ms	13.1ms	13.1ms	saxpy
0.00%	5.1us	3	1.9us	1.6us	1.7us	CUDA memcpy HtoD
0.00%	2.8us	1	2.8us	2.8us	2.8us	CUDA memcpy DtoH

Πίνακας 48: SAXPY: nvprof - 200000000

Time(%)	Time	Calls	Avg	Min	Max	Name
88.7%	204.6ms	2	102.3ms	97.9ms	106.7us	fill_random_arr
11.3%	26.0ms	1	26.0ms	26.0ms	26.0ms	saxpy
0.00%	4.8us	3	1.6us	1.57us	1.6us	CUDA memcpy HtoD
0/00%	2.8us	1	2.8us	2.8us	2.8us	CUDA memcpy DtoH

Πίνακας 49: SAXPY: nvprof - 300000000

Time(%)	Time	Calls	Avg	Min	Max	Name
88.84%	299.2ms	2	149.12ms	143.12ms	156.1ms	fill_random_arr
11.15%	37.6ms	1	37.6ms	37.6ms	37.6ms	saxpy
0.00%	4.8us	3	1.6us	1.59us	1.63us	CUDA memcpy HtoD
0.00%	2.7us	1	2.65us	2.66us	2.66us	CUDA memcpy DtoH

Πίνακας 50: SAXPY: nvprof - 400000000

Time(%)	Time	Calls	Avg	Min	Max	Name
88.7%	386.3ms	2	193.2ms	182.0ms	204.3ms	fill_random_arr
11.3%	49.2ms	1	49.2ms	49.2ms	49.2ms	saxpy
0.00%	4.8us	3	1.6us	1.5us	1.7us	CUDA memcpy HtoD
0.00%	2.5us	1	2.5us	2.5us	2.5us	CUDA memcpy DtoH

Πίνακας 51: SAXPY: nvprof - 500000000

Time(%)	Time	Calls	Avg	Min	Max	Name
88.4%	473.0ms	2	236.5ms	219.3ms	253.7ms	fill_random_arr
11.6%	62.1ms	1	62.1ms	62.1ms	62.1ms	saxpy
0.00%	4.7us	3	1.6us	1.5us	1.6us	CUDA memcpy HtoD
0.00%	2.6us	1	2.6us	2.6us	2.6us	CUDA memcpy DtoH

Πίνακας 52: SAXPY: nvprof - 600000000

Time(%)	Time	Calls	Avg	Min	Max	Name
88.3%	552.9ms	2	276.5ms	255.4ms	297.5ms	fill_random_arr
11.7%	73.2ms	1	73.2ms	73.2ms	73.2ms	saxpy
0.00%	4.5us	3	1.5us	1.4us	1.5us	CUDA memcpy HtoD
0.00%	2.4us	1	2.4us	2.4us	2.4us	CUDA memcpy DtoH

4.4 Αριθμητικός υπολογισμός σταθεράς π

Στο παράδειγμα που ακολουθεί εφαρμόζεται η μέθοδος της αριθμητικής ολοκλήρωσης για τον υπολογισμό της σταθεράς π. Ο αλγόριθμος αποτελείται από έναν βρόγχο επανάληψης που εκτελεί ένα προκαθορισμένο αριθμό επαναλήψεων. Σε κάθε επανάληψη υπολογίζεται μία τιμή x που αποτελεί το μερικό αποτέλεσμα του τελικού υπολογισμού και λειτουργεί ανεξάρτητα από τις υπόλοιπες επαναλήψεις. Στο τέλος του βρόγχου το άθροισμα των επιμέρους τιμών αποτελεί το τελικό αποτέλεσμα. Όσο πιο μεγάλος είναι ο αριθμός των επαναλήψεων, τόσο πιο ακριβής είναι και ο υπολογισμός του π.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \sum_{n=1}^{\infty} \frac{4}{1+x^2}\Delta$$

$$\Delta = \frac{1}{N}$$

$$x_i = \frac{i-0.5}{\Delta}$$

4.4.1 Περιγραφή κοινού ρουτίνας main του προβλήματος υπολογισμού π

Συμβ. 72: PI: main

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    auto seconds = omp_get_wtime();
    double p = pi(o.num_steps);
    std::cout << "Elapsed Time: " <<
                        omp_get_wtime() - seconds << std::endl;
    std::cout << "pi Value: " << p << std::endl;
    return 0;
}
```

4.4.2 Σειριακή εκτέλεση

Η σειριακή εκτέλεση του προβλήματος είναι απλή και τα αποτελέσματα της θα χρησιμοποιηθούν ως μέτρο σύγκρισης για τις παραλλαγές που θα ακολουθήσουν και θα εμπεριέχουν παραλληλισμό.

Συμβ. 73: PI: Σειριακή υλοποίηση

```
double pi(long num_steps) {
    int upper_limit = 1;
    double step = upper_limit/(double)num_steps;
    double sum = .0, pi = 0.0;

    for (int i = 0; i < num_steps; ++i){
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x*x);
    }
    pi = step * sum;
    return pi;
}
```

Πίνακας 53: PI: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 54: PI: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt1	**Alt2**
100000000	1.076	1.098
200000000	2.132	2.123
300000000	3.188	3.185
400000000	4.321	4.295
500000000	5.393	5.371
600000000	6.394	6.370

4.4.2.1 Παρατηρήσεις

Όπως φαίνεται από τα παραπάνω αποτελέσματα, δεν επιτυγχάνεται διανυσματικοποίηση μέσω της δοθείσης εντολής στο μεταγλωττιστή. Η αδυναμία οφείλεται στην μεταβλητή *sum* και στο άθροισμα των επιμέρους υπολογισμών.

4.4.3 Παραλλαγή με omp parallel

Η παραλλαγή αυτής της παραγράφου αποτελεί πολύπλοκη λύση για ένα απλό πρόβλημα παραλληλοποίησης. Πριν την έναρξη του βρόγχου επανάληψης, υπολογίζεται ο συνολικός αριθμός των νημάτων που τρέχουν σε μια παράλληλη περιοχή. Για κάθε νήμα, δημιουργείται μια μεταβλητή sum στην οποία θα γίνεται το άθροισμα των επιμέρους τμημάτων του βρόγχου επανάληψης. Το άθροισμα των μεταβλητών αυτών αποτελεί και το τελικό αποτέλεσμα. Στη συνέχεια, ο κώδικας εισέρχεται στην παράλληλη περιοχή όπου σε κάθε νήμα ανατίθεται ένας αριθμός επαναλήψεων που θα εκτελέσει. Αφού τα νήματα ολοκληρώσουν τους υπολογισμούς, το τελικό αποτέλεσμα αθροίζεται στην μεταβλητή pi.

Συμβ. 74: PI: omp parallel

```
double pi(long num_steps) {
    double pi = .0;
    int num_threads = 0;
#pragma omp parallel shared(num_threads)
    {
        int id = omp_get_thread_num();
        if (id == 0) {
            num_threads = omp_get_num_threads();
        }
    }

    particle *sum = new particle[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        sum[i].val = 0.0;
    }
    double step= 1.0/(double)num_steps;

#pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        int numthreads = omp_get_num_threads();
        int low = num_steps * thread_num / numthreads;
        int high = num_steps * (thread_num + 1)/ numthreads;

        for (int i = low; i < high; ++i) {
            double x = (i + 0.5)*step;
            sum[thread_num].val += 4.0/(1.0 + x*x);
        }
    }

    for (int i = 0; i < num_threads; ++i) {
        pi += sum[i].val * step;
    }

    delete []sum;
    return pi;
}
```

Συμβ. 75: Particle

```
struct particle {
       double val;
};
```

Πίνακας 55: PI: Επιλογές μεταγλώττισης Alt3, Alt4

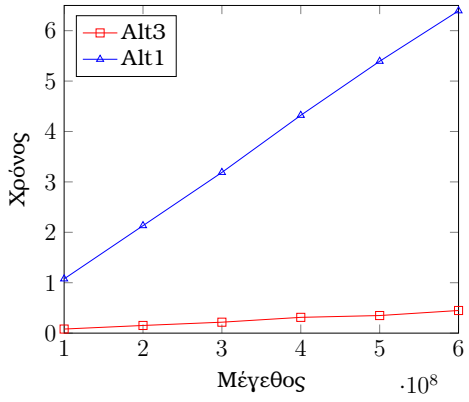
Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 56: PI: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt3	**Alt4**
100000000	0.081	0.093
200000000	0.152	0.154
300000000	0.217	0.220
400000000	0.313	0.285
500000000	0.349	0.373
600000000	0.449	0.420

4.4.3.1 Παρατηρήσεις

Παρά τη μεγάλη πολυπλοκότητα της υλοποίησης του προβλήματος με παραλληλοποίηση, ο χρόνος εκτέλεσης του προβλήματος είναι πολύ μικρότερος σε σύγκριση με τη σειριακή εκτέλεση.



Σχήμα 23: PI: Σύγκριση Alt1, Alt3

Μέγεθος	Επιτάχυνση (%)
100000000	92.4
200000000	92.9
300000000	93.2
400000000	92.8
500000000	93.5
600000000	92.9

Πίνακας 57: PI: Ποσοστιαία σύγκριση Alt1 και Alt3

4.4.4 Παραλλαγή με χρήση κυκλίκής κατανομής - omp atomic

Σε αυτή την ενότητα γίνεται αντικατάσταση του πίνακα sum της προηγούμενης παραγράφου με ιδιωτικές μεταβλητές σε κάθε νήμα και ενοποίηση των δυο επιμέρους παράλληλων τμημάτων κώδικα σε μια. Επίσης ο διαμοιρασμός της εργασίας ανάμεσα στα νήματα γίνεται με κυκλική κατανομή.

Συμβ. 76: PI: omp parallel - atomic

```
double pi(long num_steps) {
    int nthreads = 0;
    double pi = .0;
    double step= 1.0/(double)num_steps;
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        double sum = 0.0, x = 0.0;

        if (id == 0) {
            nthreads = nthrds;
        }

        for (int i = id; i < num_steps; i += nthreads) {
            x = (i + 0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }
        sum *= step;
#pragma omp atomic
        pi += sum;
    }

    return pi;
}
```

Πίνακας 58: PI: Επιλογές μεταγλώττισης Alt5, Alt6

Label	Options
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt6

4.4.4.1 Παρατηρήσεις

Συγκριτικά με την παραλλαγή Alt3, ο χρόνος εκτέλεσης της Alt5 δε μεταβάλλεται. Είναι εμφανές ωστόσο, ότι σε αυτή την παράγραφο η υλοποίηση είναι πιο κατανοητή και εύκολη. Θα ήταν ακόμα πιο εύχρηστο, αντί για χρήση κυκλικής κατανομής, να γίνει χρήση της οδηγίας pragma omp parallel for.

Πίνακας 59: PI: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt5	Alt6
100000000	0.103	0.098
200000000	0.160	0.149
300000000	0.219	0.215
400000000	0.287	0.283
500000000	0.353	0.343
600000000	0.422	0.418

4.4.5 Παραλλαγή με χρήση omp for - reduction

Στο παράδειγμα της παραγράφου, έγινε αντικατάσταση της κυκλικής κατανομής με την οδηγίας pragma omp parallel for. Επιπλέον όφελος σε αυτή τη μεταβολή είναι η δυνατότητα χρήσης της φράσης reduction.

Συμβ. 77: PI: omp parallel for reduction

```
double pi(long num_steps) {
    double pi = .0;
    double step= 1.0/(double)num_steps;
    double sum = 0.0;

#pragma omp parallel
    {
        double x = 0.0;

#pragma omp for reduction(+:sum)
        for (int i = 0; i < num_steps; i++) {
            x = (i + 0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }
    }
    pi = step * sum;
    return pi;
}
```

Πίνακας 60: PI: Επιλογές μεταγλώττισης Alt7, Alt8

Label	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8

Πίνακας 61: PI: Αποτελέσματα Alt7, Alt8

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt7	**Alt8**
100000000	0.080	0.087
200000000	0.154	0.166
300000000	0.216	0.230
400000000	0.294	0.293
500000000	0.371	0.369
600000000	0.421	0.415

4.4.5.1 Παρατηρήσεις

Ως συνέπεια της αντικατάσης της κυκλικής κατανομής, η οδηγία omp parallel for οδήγησε σε πιο ευανάγνωστη υλοποίηση. Ωστόσο, οι χρόνοι εκτέλεσης παρέμειναν σταθεροί σε σύγκριση με την προηγούμενη εκτέλεση.

4.4.6 Παραλλαγή με χρήση parallel for simd reduction

Σε αυτή την ενότητα γίνεται χρήση της οδηγίας omp parallel for σε συνδυασμό με τις φράσεις reduction και simd με σκοπό να ελεγχθεί η δυνατότητα του OpenMP να εφαρμόσει διανυσματικοποίηση, παρόλο που ο μεταγλωττιστής δε μπορεί.

Συμβ. 78: PI: omp parallel for simd

```
double pi(long num_steps) {
    double dH = 1.0/(double)num_steps;
    double dX, dSum = 0.0;

#pragma omp parallel for simd private(dX) \
    reduction(+:dSum) schedule(simd:static)
    for (int i = 0; i < num_steps; i++) {
        dX = dH * ((double) i  + 0.5);
        dSum += (4.0 / (1.0 + dX * dX));
    } // End parallel for simd region

    return dH * dSum;
}
```

Πίνακας 62: PI: Επιλογές μεταγλώττισης Alt9, Alt10, Alt11

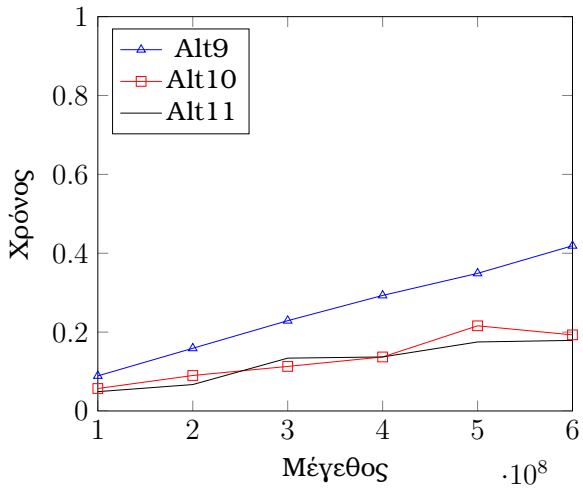
Label	Options
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt9
Alt10	-fopt-info-vec=builds/alt10.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt10
Alt11	-fopt-info-vec=builds/alt11.log -O2 -fno-inline -fopenmp -o ./builds/Alt11

Πίνακας 63: PI: Αποτελέσματα Alt9, Alt10, Alt11

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		
	Alt9	**Alt10**	**Alt11**
100000000	0.089	0.057	0.049
200000000	0.159	0.090	0.067
300000000	0.229	0.113	0.134
400000000	0.293	0.137	0.137
500000000	0.349	0.216	0.175
600000000	0.419	0.193	0.179

4.4.6.1 Παρατηρήσεις

Όπως είναι εμφανές, η διανυσματικοποίηση στις παραλλαγές Alt10 και Alt11 ήταν επιτυχής, σε αντίθεση με την παραλλαγή Alt9. Αυτό επιβεβαιώνεται και από τα παραγόμενα αρχείο *.log κατά τη μεταγλώττιση του προγράμματος. Το όφελος από την διανυσματικοποίηση φαίνεται στο διάγραμμα που ακολουθεί.



Σχήμα 24: PI: Σύγκριση Alt9, Alt10, Alt11

Μέγεθος	Επιτάχυνση (%)
100000000	44.9
200000000	57.8
300000000	41.5
400000000	53.2
500000000	49.9
600000000	57.3

Πίνακας 64: PI: Ποσοστιαία σύγκριση Alt9 και Alt11

4.4.7 Παραλλαγή με χρήση offloading

Συμβ. 79: PI: omp target teams distribute parallel

```
double pi(long num_steps) {
    double dH = 1.0/(double)num_steps;
    double dX = 0.0, dSum = 0.0;

#pragma omp target teams distribute parallel for simd\
                map(tofrom: dSum), map(to:dX,  dH, num_steps)\
                reduction(+:dSum)
    for (int i = 0; i < num_steps; i++) {
        dX = dH * ((double) i  + 0.5);
        dSum += (4.0 / (1.0 + dX * dX));
    } // End parallel for simd region

    return dH * dSum;
}
```

Πίνακας 65: PI: Επιλογές μεταγλώττισης Alt12

Label	Options
Alt12	-fopt-info-vec=builds/alt12.log -O2 -fno-inline -fno-tree-vectorize -fno-stack-protector -fopenmp -foffload=nvptx-none="-O2 -fno-inline" -o ./builds/Alt12

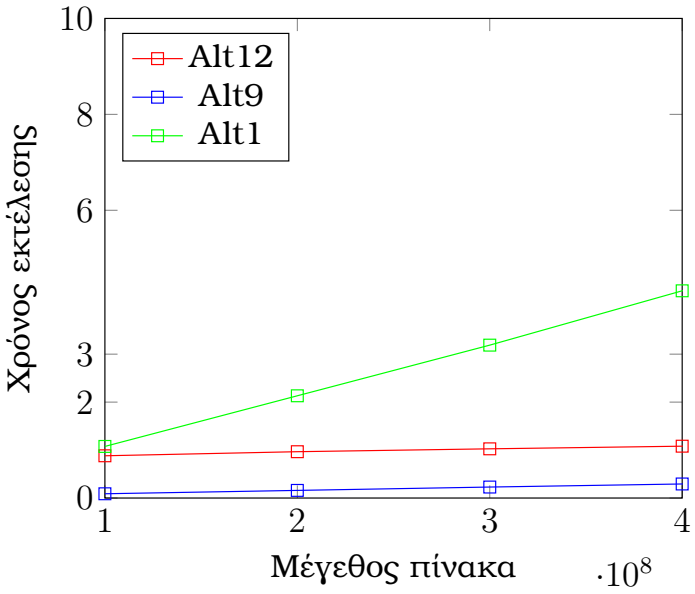
Πίνακας 66: PI: Αποτελέσματα Alt12

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt12
100000000	0.881
200000000	0.966
300000000	1.026
400000000	1.081
500000000	1.111
600000000	1.154

4.4.7.1 Παρατηρήσεις

Οι χρόνοι εκτέλεσης του προγράμματος μεταφερόμενου στην μονάδα επεξεργασίας της κάρτας γραφικών δε δίνει τα αναμενόμενα αποτελέσματα. Οι επιδόσεις είναι αρκετά χαμηλότερες σε σύγκριση με την παραλλαγή Alt11 που αποτελεί την καλύτερη λύση. Η

καθυστέρηση δεν οφείλεται στη μεταφορά των δεδομένων ανάμεσα στις δυο συσκευές. Το μέγεθος των δεδομένων που απαιτείται για να μεταφερθεί στο περιβάλλον της κάρτας γραφικών είναι πολύ μικρότερο σε σχέση με το παράδειγμα SAXPY.



Σχήμα 25: SAXPY: Σύγκριση αποτελεσμάτων Alt12-Alt9-Alt1

4.4.8 Σύγκριση atomic - critical

Στο παράδειγμα της παραγράφου, γίνεται μια προσπάθεια σύγκρισης της οδηγίας critical και της atomic. Στη γενική εικόνα, η οδηγία atomic επιτρέπει στο μεταγλωττιστή να έχει περισσότερες ευκαιρίες για βελτιστοποίηση. Για μονοδιάστατους πίνακες επίσης, η atomic επιτρέπει την ταυτόχρονη τροποποίηση μεταβλητών σε διαφορετικές θέσεις του πίνακα από διαφορετικά νήματα ενώ η critical όχι.

Συμβ. 80: PI: Σύγκριση atomic - critical

```
double pi(long num_steps) {
    double pi = .0;
    double step= 1.0/(double)num_steps;
    double sum = 0.0;

#pragma omp parallel firstprivate(sum) shared(pi)
    {
        double x = 0.0;

#pragma omp for
        for (int i = 0; i < num_steps; i++) {
            x = (i + 0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }

#pragma omp critical //atomic
        pi += step * sum;
    }

    return pi;
}
```

Πίνακας 67: PI: Επιλογές μεταγλώττισης Alt12

Label	Options
Alt13	-fopt-info-vec=builds/alt13.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt13

4.4.8.1 Παρατηρήσεις

Τα αποτελέσματα του πίνακα δε δείχνουν κάποια ιδιαίτερη βελτιστοποίηση στο χρόνο εκτέλεσης του προβλήματος. Παρόλα αυτά, τα αποτελέσματα δικαιολογούνται, καθώς στη συγκεκριμένη υλοποίηση, η μεταβλητή pi μεταβάλλεται λίγες φορές και συγκεκριμένα, όσα είναι και τα νήματα του παράλληλου τμήματος κώδικα.

Πίνακας 68: PI: Αποτελέσματα Atomic - Critical

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Atomic	Critical
100000000	0.088	0.095
200000000	0.167	0.172
300000000	0.231	0.215
400000000	0.296	0.290
500000000	0.348	0.367
600000000	0.424	0.438

4.4.9 Σύγκριση atomic - critical (2)

Στο συγκεκριμένο παράδειγμα, ο στόχος δε είναι η σωστή υλοποίηση του προβλήματος, αλλά η η σύγκριση των οδηγιών critical και atomic, όταν αυτές είναι τοποθετημένες σε σημείο που θα κληθούν πολλές φορές. Για το λόγο αυτό, το συνολικό μέγεθος του προβλήματος μειώνεται σε σχέση με τα προηγούμενα, καθώς η εισάγωγή της atomic/-critical εντός του βρόγχου επανάληψης, μετατρέπει το πρόβλημα σε σειριακό και μάλιστα με επιπλέον χρονική επιβάρυνση, λόγω του κλειδώματος που απαιτείται όταν καλούνται αυτές οι οδηγίες.

Συμβ. 81: PI: Σύγκριση atomic - critical(2)

```
double pi(long num_steps) {
    double pi = .0;
    double step= 1.0/(double)num_steps;

#pragma omp parallel shared(pi)
    {
        double x = 0.0;

#pragma omp for
        for (int i = 0; i < num_steps; i++) {
            x = (i + 0.5)*step;
#pragma omp atomic //critical
            pi += step * 4.0/(1.0 + x*x);
        }
    }

    return pi;
}
```

Πίνακας 69: PI: Επιλογές μεταγλώττισης Alt13

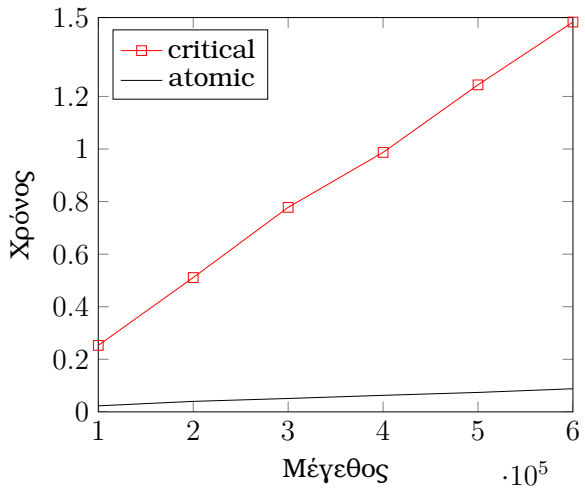
Label	Options
Alt14	-fopt-info-vec=builds/alt14.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt14

Πίνακας 70: PI: Αποτελέσματα Atomic - Critical

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Atomic	**Critical**
100000	0.023	0.253
200000	0.040	0.511
300000	0.051	0.778
400000	0.063	0.987
500000	0.074	1.244
600000	0.088	1.483

4.4.9.1 Παρατηρήσεις

Από τον παραπάνω πίνακα, είναι προφανές ότι η οδηγία atomic πρέπει να αντικαταστήσει την critical όπου είναι εφικτό, καθώς ο όφελος σε σύγκριση με τη χρήση της critical είναι μεγαλύτερο.



Σχήμα 26: PI: Σύγκριση atomic - critical

4.5 Υπολογισμός Εσωτερικού Γινομένου - *Dot Product*

Δοθέντων δυο μονοδιάστατων πινάκων **a** και **b**, όπου το καθένα αποτελείται από **n** στοιχεία, το εσωτερικό γινόμενο $a * b$ ορίζεται ως ένας αριθμός x που υπολογίζεται ως εξής:

$$x = \sum a_i * b_i$$

όπου a_i και b_i δηλώνουν το i στοιχείο του διανύσματος **a** και **b**.

Κατά την παραλληλοποίηση του προγράμματος, η αναλογική επιτάχυνση είναι σχεδόν απίθανη λόγω του μικρού μεγέθους υπολογισμών. Όπως και με το πρόβλημα *SAXPY* είναι πιθανό πως ο χρόνος υπολογισμού θα είναι μικρότερος από τον χρόνο προσπέλασης της μνήμης μέσω βρόγχου επανάληψης.

Ωστόσο, το πρόβλημα εσωτερικού γινομένου κατατάσσεται στην κατηγορία του συνδυασμού μοτίβων που ονομάζεται *map - reduce*, που είναι ευρέως χρήσιμο και γιαυτό μελετάται.

4.5.1 Περιγραφή κοινού τμήματος αλγορίθμου DotProd

Το πρόβλημα ξεκινάει δημιουργώντας δυο διανύσματα ίσου μεγέθους και βάζοντας τυχαίους αριθμούς σε κάθε θέση τους. Στη συνέχεια καλείται η ρουτίνα υπολογισμού εσωτερικού γινομένου και όταν τελειώσει επαληθεύεται το αποτέλεσμα.

Συμβ. 82: DotProd: verify()

```
void verify(size_t size, float *a, float *b, float got) {
    float temp = 0.0;
    for (size_t i = 0; i < size; ++i) {
        temp += a[i] * b[i];
    }

    std::cout << "Verification result: " << temp << std::endl;
    std::cout << "In verification Gor: " << got << std::endl;

    if (!(fabs(got - temp) < 1e-6)) {
        std::cout << "FAILED! Not correct result. -> Expected: " <<
                        temp << ". Got: " << got << std::endl;
        exit(1);
    }
}
```

Συμβ. 83: DotProd: main()

```
int main(int argc, char **argv)
{
    Opts o;
    parseArgs(argc, argv, o);
    srand(time(nullptr));
    float *a = new float[o.size];
    float *b = new float[o.size];
    fill_random_arr(a, o.size);
    fill_random_arr(b, o.size);
    auto start = omp_get_wtime();
    float got = dprod(o.size, a, b);
    auto end = omp_get_wtime();
    if (o.verify) {
       verify(o.size, a, b, got);
    }
    std::cout << std::endl << "Execution time: " << std::fixed <<
                end - start << std::setprecision(5) << std::endl;
    std::cout << "Result: " << got << std::endl;

    delete []a;
    delete []b;

    return 0;
}
```

4.5.2 Σειριακή εκτέλεση

Ξεκινώντας την δημιουργία παραλλαγών για τον υπολογισμό του εσωτερικού γινομένου, η πρώτη αναφέρεται στη σειριακή εκτέλεση του προβλήματος με επιλογές -O2 με διανυσματικοποίηση και χωρίς.

Συμβ. 84: DotProd: Σειριακή εκτέλεση

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 71: DotProd: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt2	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 72: DotProd: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt1	**Alt2**
100000000	0.298	0.298
200000000	0.918	0.936
300000000	1.546	1.642
400000000	2.254	1.847
500000000	3.507	3.214

4.5.2.1 Παρατηρήσεις

Όπως ήταν αναμενόμενο, η εφαρμογή διανυσματικοποίησης στο συγκεκριμένο παράδειγμα δεν είναι εφικτή λόγω της φύσης του προβλήματος. Η διανυσματικοποίηση δε μπορεί να εφαρμοστεί καθώς το γινόμενο των στοιχείων των δύο ανυσμάτων αθροίζεται στα υπόλοιπα. Η αδυναμία διανυσματικοποίησης επιβεβαιώνεται από το αρχείο *info.log* που εξάγεται κατά τη διάρκεια της μεταγλώττισης -λόγω της επιλογής -fopt-info-vec-, καθώς είναι άδειο.

4.5.3 Παραλλαγές με pragma omp parallel for

Οι παραλλαγές που ακολουθούν, έχουν ως κύριο χαρακτηριστικό την χρήση της οδηγίας *parallel for* συνδυαζόμενη με διαφορετικές φράσεις.

4.5.3.1 Χρήση φράσης critical

Η οδηγία *parallel for* συνδυάζεται με τη φράση *pragma omp critical* με σκοπό την αποφυγή του προβλήματος *race condition*. Είναι προφανές, ότι η χρήση της φράσης *critical* και της παραλληλοποίησης γενικότερα οδηγεί σε μεγαλύτερα προβλήματα σε ότι αφορά τις επιδόσεις, καθώς η χρήση της πρακτικά μετατρέπει την παραλληλοποίηση σε σειριακή εκτέλεση, εισάγοντας ακόμη μια επιπλέον καθυστέρηση λόγω των εργασιών που απαιτούνται για αποκλειστικό διάβασμα/εγγραφή από/στη τη μνήμη.

Συμβ. 85: DotProd: omp parallel for - critical

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp parallel for
    for (size_t i = 0; i < num; ++i) {
#pragma omp critical
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 73: DotProd: Επιλογές μεταγλώττισης Alt3, Alt4

Label	Options
Alt3	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt4	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

4.5.3.1.1 Παρατηρήσεις

Πράγματι, κατά την προσπάθεια εκτέλεσης του προβλήματος με μέγεθος διανύσματος 1e8, παρατηρήθηκε πολύ μεγάλος χρόνος εκτέλεσης. Για τον ίδιο λόγο με την υπόθεση, δεν καταγράφεται η παραλλαγή με αντικατάσταση της critical με atomic. Έπειτα απόπειραματική υλοποίηση, τα συμπεράσματα είναι τα ίδια.

4.5.3.2 Χρήση private μεταβλητών για κάθε νήμα

Στην παραλλαγή χρησιμοποιείται σε μειωμένη έκταση σε σχέση με την προηγούμενη, η φράση *critical*. Εδώ, το άθροισμα των επιμέρους παραλλαγών εισάγεται σε τοπική μεταβλητή του νήματος. Με την ολοκλήρωση του βρόγχου επανάληψης οι τοπικές μεταβλητές αθροίζονται σε μία κοινόχρηστη, μέσω της φράσης *critical*.

Συμβ. 86: DotProd: parallel for critical

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp parallel shared(res)
    {
        double temp = 0.0;
#pragma omp for
        for (size_t i = 0; i < num; ++i) {
            temp += a[i] * b[i];
        }
#pragma omp critical
        res += temp;

    }
    return res;
}
```

Πίνακας 74: DotProd: Επιλογές μεταγλώττισης Alt7, Alt8

Label	Options
Alt7	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt8	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

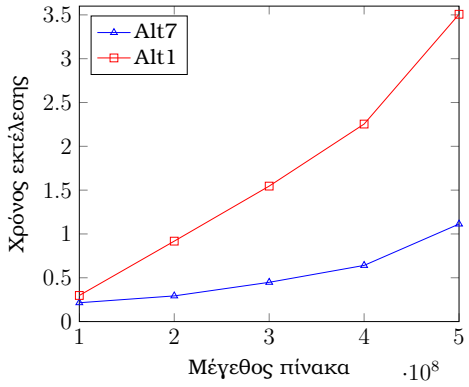
Πίνακας 75: DotProd: Αποτελέσματα Alt7, Alt8

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt7	**Alt8**
100000000	0.215	0.217
200000000	0.293	0.292
300000000	0.448	0.524
400000000	0.641	0.622
500000000	1.113	1.191

4.5.3.2.1 Παρατηρήσεις

Συγκρινόμενη με την προηγούμενη παράγραφο, ο χρόνος εκτέλεσης είναι πεπερασμένος και μικρός. Μάλιστα είναι μικρότερος σε σύγκριση με τη σειριακή εκτέλεση,

όπως φαίνεται στο διάγραμμα που ακολουθεί. Ωστόσο είναι εμφανής η μεγαλύτερη πολυπλοκότητα της υλοποίησης. Λόγο του μικρού αριθμού κλήσεων της εντολής *res += temp;*, η υλοποίηση με atomic δεν οδηγεί σε καλύτερες χρονικές επιδόσεις.



Σχήμα 27: DotProd: Σύγκριση Alt7, Alt1

Μέγεθος	Πόσοστό μείωσης χρόνου (%)
1e8	27.8
2e8	68.1
3e8	71.0
4e8	71.6
5e8	68.2

Πίνακας 76: DotProd: Πόσοστιαία σύγκριση Alt7 και Alt1

4.5.4 Παραλλαγές με εφαρμογή διανυσματικοποίησης

Η ομάδα παραλλαγών που ακολουθεί έχει ως επίκεντρο την εφαρμογή διανυσματικοποίησης μέσω simd. Οπως προαναφέρθηκε, δεν αναμένονται ιδιαίτερα αποτελέσματα, καθώς η φύση του προβλήματος δεν επιτρέπει την εφαρμογή διανυσματικοποίησης.

4.5.4.1 Χρήσης οδηγίας simd

Συμβ. 87: DotProd: omp simd

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp simd
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 77: DotProd: Επιλογές μεταγλώττισης Alt9

Label	Options
Alt9	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 78: DotProd: Αποτελέσματα Alt9

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt9
100000000	0.275
200000000	0.859
300000000	1.356
400000000	2.452
500000000	2.909

4.5.4.1.1 Παρατηρήσεις

Οι χρονικές καταγραφές της εκτέλεσης είναι συγκρίσιμες με τη σειριακή. Δεν εφαρμόζεται διανυσματικοποίηση και αυτό επιβεβαιώνεται και από το αρχείο *info.log*

4.5.4.2 Χρήση φράσης reduction

Συμβ. 88: DotProd: parallel for reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp parallel for reduction(+ : res)
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 79: DotProd: Επιλογές μεταγλώττισης Alt5, Alt6

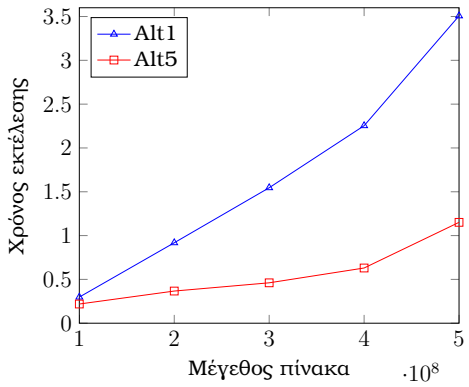
Label	Options
Alt5	-fopt-info-vec=info.log -fno-inline -fno-tree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2
Alt6	-fopt-info-vec=info.log -fno-inline -ftree-vectorize -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 80: DotProd: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt5	**Alt6**
100000000	0.219	0.216
200000000	0.367	0.309
300000000	0.461	0.576
400000000	0.630	0.654
500000000	1.151	1.266

4.5.4.2.1 Παρατηρήσεις

Είναι εμφανής η βελτίωση των επιδόσεων και η επίτευξη της διανυσματικοποίησης λόγω της αντικατάστασης της φράσης *critical* με την *reduction*. Η διανυσματικοποίηση δεν είναι εφικτή ούτε μέσω *OpenMP*. Τα ποσοστά μείωσης του χρόνου εκτέλεσης σε σχέση με τη σειριακή εκτέλεση είναι περίπου 60 - 70% όπως φαίνεται και στο διάγραμμα που ακολουθεί.



Σχήμα 28: DotProd: Σύγκριση Alt1, Alt5

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
1e8	26.5
2e8	60
3e8	70
4e8	72
5e8	67

Πίνακας 81: DotProd: Ποσοστιαία σύγκριση Alt1 και Alt5

4.5.4.3 Χρήσης φράσης reduction

Συμβ. 89: DotProd: simd reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp simd reduction(+ : res)
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 82: DotProd: Επιλογές μεταγλώττισης Alt10

Label	**Options**
Alt10	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 83: DotProd: Αποτελέσματα Alt10

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**
	Alt10
100000000	0.251
200000000	0.949
300000000	1.395
400000000	1.856
500000000	2.798

4.5.4.3.1 Παρατηρήσεις

Οπως ήταν αναμενόμενο, η εισαγωγή της φράσης *reduction* δε προσέδωσε κανένα κέρδος στην επίλυση του προβλήματος καθώς δεν έχει εφαρμοστεί διανυσματικοποίηση. Ακόμη, δε εκτελέστηκε καμία εργασία παράλληλα, καθώς για να γίνει αυτό απαιτείται η οδηγία *parallel for*.

4.5.4.4 Χρήσης φράσης declare simd notinbranch

Η μεταφορά του υπολογισμού του εσωτερικού γινομένου εντός συνάρτησης, και η κλήση της με χρήση διανυσματικοποίησης γίνεται με την οδηγία *omp declare simd notinbranch*.

Συμβ. 90: DotProd: parallel for reduction

```
#pragma omp declare simd notinbranch
double mult(double *a, double *b) {
    return *a * *b;
}

double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp simd reduction(+ : res)
    for (size_t i = 0; i < num; ++i) {
        res += mult(&a[i], &b[i]);
    }
    return res;
}
```

Πίνακας 84: DotProd: Επιλογές μεταγλώττισης Alt11

Label	Options
Alt11	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 85: DotProd: Αποτελέσματα Alt11

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt11
100000000	1.150
200000000	2.344
300000000	3.911
400000000	5.512
500000000	6.633

4.5.4.4.1 Παρατηρήσεις

Η χρήση της οδηγίας *declare simd notinbranch* συντελεί στην πτώση της απόδοσης του αλγορίθμου. Ο χρόνος εκτέλεσης αυξάνεται δύο φορές περισσότερο σε σύγκριση με τη σειριακή εκτέλεση.

4.5.4.5 Χρήση οδηγίας simd parallel for reduction

Συμβ. 91: DotProd: parallel for simd reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp parallel for simd reduction(+ : res)
    for (size_t i = 0; i < num; ++i) {
        res += a[i] * b[i];
    }
    return res;
}
```

Πίνακας 86: DotProd: Επιλογές μεταγλώττισης Alt12

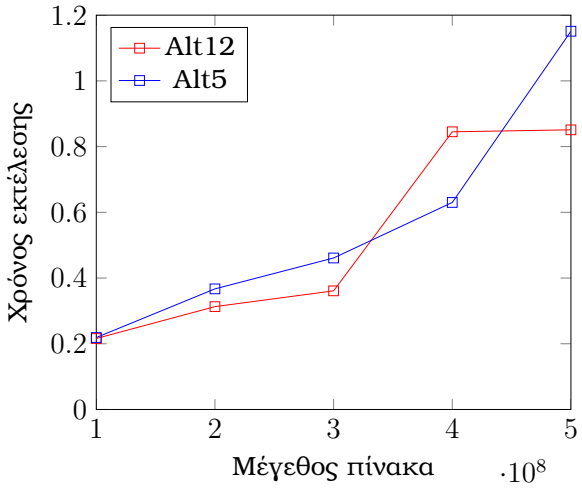
Label	Options
Alt12	-fopt-info-vec=info.log -fno-inline -fopenmp -Wall -Wextra -std=c++14 -O2

Πίνακας 87: DotProd: Αποτελέσματα Alt12

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt12
100000000	0.216
200000000	0.313
300000000	0.361
400000000	0.845
500000000	0.851

4.5.4.5.1 Παρατηρήσεις

Από το διάγραμμα σύγκρισης των παραλλαγών Alt5 - Alt12, προκύπτει διακύμανση της καλύτερης παραλλαγής αναμεσά τους. Δυστυχώς δε μπόρεσαν να εξαχθούν ασφαλή συμπεράσματα για τα αίτια της διακύμανσης αυτής.



Σχήμα 29: DotProd: Σύγκριση Alt12, Alt5

4.5.5 Παραλλαγές με offloading

Οι παραλλαγές που ακολουθούν, αφορούν παραλλαγές υλοποιημένες στη κάρτα γραφικών.

4.5.5.1 Χρήση οδηγίας parallel for reduction

Συμβ. 92: DotProd: target parallel for reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target map(tofrom: res) map(to: a[0:num], b[0:num])
#pragma omp parallel for reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 88: DotProd: Επιλογές μεταγλώττισης Alt13

Label	Options
Alt13	-fopt-info-vec=builds/alt13.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt13

Πίνακας 89: DotProd: Αποτελέσματα Alt13

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	**Ποσοστό συνολικού χρόνου (%)**		
	Alt13	**DotProd**	**memcpy DtoH**	**memcpy HtoD**
100000000	3.201	77.99	22.01	0
200000000	5.871	78.91	21.09	0
300000000	8.683	78.11	21.89	0
400000000	11.861	75.32	24.68	0
500000000	14.788	75.07	24.93	0

4.5.5.1.1 Παρατηρήσεις

Παρατηρείται ότι ο χρόνος εκτέλεσης του προβλήματος είναι μεγάλος συγκριτικά με όλες τις προηγούμενες παραλλαγές. Η καθυστέρηση δεν οφείλεται στη μεταφορά των μεταβλητών ανάμεσα στις δυο συσκευές καθώς αποτελεί μόνο το 22% του συνολικού χρόνου που καταναλώνεται στην κάρτα γραφικών. Ωστόσο, δε περιμένουμε καλύτερες επιδόσεις καθώς οι εργασίες δε μοιράζονται σε ομάδες νημάτων της συσκευής στόχου μέσω τον σχετικών οδηγιών.

4.5.5.2 Χρήση οδηγίας target simd reduction(+ : res)

Στο παράδειγμα της παραγράφου γίνεται αντικατάσταση της οδηγίας parallel for με την simd.

Συμβ. 93: DotProd: target simd reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target map(tofrom: res) map(to: a[0:num], b[0:num])
#pragma omp simd reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 90: DotProd: Επιλογές μεταγλώττισης Alt14

Label	Options
Alt14	-fopt-info-vec=builds/alt14.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt14

Πίνακας 91: DotProd: Αποτελέσματα Alt14

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	Ποσοστό συνολικού χρόνου (%)		
	Alt14	DotProd	memcpy DtoH	memcpy HtoD
100000000	1.581	52.98	47.02	0.00
200000000	3.10	48.02	51.98	0.00
300000000	4.203	51.52	48.48	0.00
400000000	5.833	48.17	51.83	0.00
500000000	7.182	48.02	51.98	0.00

4.5.5.2.1 Παρατηρήσεις

Η αντικατάσταση της οδηγίας simd οδήγησε στη μείωση των χρονικών επιδόσεων. Η συμπεριφορά είναι αντίστροφη των αντίστοιχων παραλλαγών, εκτελεσμένων μέσω CPU. Επίσης, ο απαιτούμενος χρόνος μεταφοράς των δεδομένων στη στο περιβάλλον της μονάδας επεξεργασίας της κάρτας γραφικών αποτελεί το 50% του συνολικού χρόνου εκτέλεσης του προβλήματος.

4.5.5.3 Χρήση οδηγίας target parallel for simd reduction(+ : res)

Συμβ. 94: DotProd: parallel for simd reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target map(tofrom: res) map(to: a[0:num], b[0:num])
#pragma omp parallel for simd  reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 92: DotProd: Επιλογές μεταγλώττισης Alt15

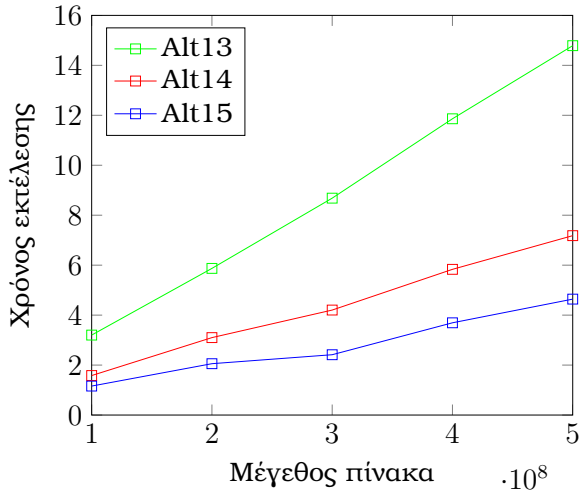
Label	Options
Alt15	-fopt-info-vec=builds/alt15.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt15

Πίνακας 93: DotProd: Αποτελέσματα Alt15

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	Ποσοστό συνολικού χρόνου (%)		
	Alt15	DotProd	memcpy DtoH	memcpy HtoD
100000000	1.160	16.90	83.09	0.00
200000000	2.056	14.20	85.80	0.00
300000000	2.415	17.69	82.31	0.00
400000000	3.693	14.51	85.49	0.00
500000000	4.641	14.00	86.00	0.00

4.5.5.3.1 Παρατηρήσεις

Η εισαγωγή ταυτόχρονα των οδηγιών parallel for και simd μείωσε τους χρόνους εκτέλεσης σε σύγκριση με τις προηγούμενες παραλλαγές με offloading. Απο το συνολικό χρόνο εκτέλεσης του προβλήματος, το 85% αφιερώνεται στην αντιστοίχιση μνήμης ανάμεσα στις δυο συσκευές.



Σχήμα 30: DotProd: Σύγκριση Alt13, Alt14, Alt15

4.5.5.4 Χρήση οδηγίας target teams parallel for simd reduction(+ : res)

Συμβ. 95: DotProd: target teams parallel for reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target map(tofrom: res) map(to: a[0:num], b[0:num])
#pragma omp teams
#pragma omp parallel for reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 94: DotProd: Επιλογές μεταγλώττισης Alt16

Label	Options
Alt16	-fopt-info-vec=builds/alt16.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt16

Πίνακας 95: DotProd: Αποτελέσματα Alt16

Μέγεθος προβλήματος (%)	Χρόνοι εκτέλεσης (sec)	Ποσοστό συνολικού χρόνου (%)		
	Alt16	**DotProd**	**memcpy DtoH**	**memcpy HtoD**
100000000	3.878	80.87	19.13	0.00
200000000	3.394	80.83	19.17	0.00
300000000	6.524	79.00	21.00	0.00
400000000	9.731	75.21	24.79	0.00
500000000	16.500	76.43	23.57	0.00

4.5.5.5 Χρήση οδηγίας teams distribute parallel for

Σε σχέση με την προηγούμενη υλοποίηση, εισάγεται η οδηγία distribute.

Συμβ. 96: DotProd: target teams reduction distribute parallel for reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target defaultmap(tofrom: scalar)\
                                (to: a[0:num], b[0:num])
#pragma omp teams reduction(+ : res)
#pragma omp distribute parallel for reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 96: DotProd: Επιλογές μεταγλώττισης Alt17

Label	Options
Alt17	-fopt-info-vec=builds/alt17.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt17

Πίνακας 97: DotProd: Αποτελέσματα Alt17

Μέγεθος προβλήματος (%)	Χρόνοι εκτέλεσης (sec)	Ποσοστό συνολικού χρόνου (%)		
	Alt17	DotProd	memcpy DtoH	memcpy HtoD
100000000	1.133	12.05	87.95	0.00
200000000	1.728	11.62	88.38	0.00
300000000	2.540	11.00	89.00	0.00
400000000	3.434	17.90	82.10	0.00
500000000	4.739	12.30	87.70	0.00

4.5.5.6 Χρήση οδηγίας teams distribute parallel for simd reduction

Οι οδηγίες και φράσεις που χρησιμοποιούνται στην παραλλαγή αυτής της ενότητας, χρησιμοποιήθηκαν και στη παραλλαγή του προβλήματος *saxpy* στην οποία καταγράφηκαν οι καλύτεροι χρόνοι εκτέλεσης σε σύγκριση με τις υπόλοιπες παραλλαγές με *offloading*.

Συμβ. 97: DotProd: target teams distribute parallel for simd reduction

```
double dprod(size_t num, double *a, double *b) {
    double res = 0.0;
#pragma omp target teams defaultmap(tofrom: scalar)\
          map(to: a[0:num], b[0:num]) reduction(+ : res)
#pragma omp distribute parallel for simd reduction(+ : res)
        for (size_t i = 0; i < num; ++i) {
            res += a[i] * b[i];
        }
    return res;
}
```

Πίνακας 98: DotProd: Επιλογές μεταγλώττισης Alt18

Label	Options
Alt18	-fopt-info-vec=builds/alt18.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt18

Πίνακας 99: DotProd: Αποτελέσματα Alt18

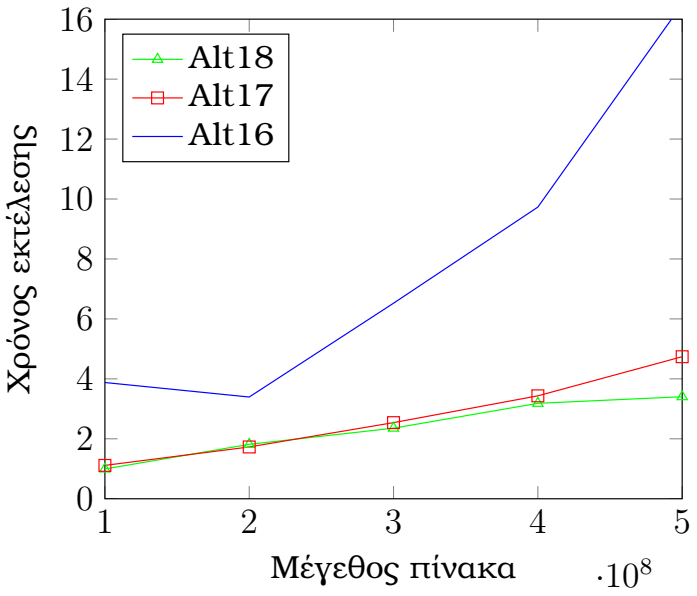
Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	Ποσοστό συνολικού χρόνου (%)		
	Alt18	DotProd	memcpy DtoH	memcpy HtoD
100000000	0.993	1.49	98.51	0.00
200000000	1.815	1.27	98.72	0.00
300000000	2.355	1.40	98.60	0.00
400000000	3.184	1.90	98.10	0.00
500000000	3.404	1.69	98.31	0.00

4.5.5.6.1 Παρατηρήσεις

Οπως ήταν αναμενόμενο, η συγκεκριμένη παραλλαγή εμφανίζει τις καλύτερες επιδόσεις συγκριτικά με τις υπόλοιπες παραλλαγές σε GPU. Το 92% του χρόνου που απαιτείται, περιλαμβάνει τη μεταφορά των δεδομένων ανάμεσα στις δυο συσκευές. Αξίζει να σημειωθεί η αξία της φράσης to έναντι της tofrom στην οδηγία map σε περίπτωση που δε χρειάζεται, όπως είναι στο παράδειγμα του εσωτερικού γινομένου. Με τη χρήση της, δεν απαιτείται αντιγραφή των δεδομένων από τη συσκευή στόχου πίσω στην κύρια συσκευή. Τέλος, οι χρόνοι εκτέλεσης αν δεν απαιτείται αντιγραφή των δεδομένων θα ήταν οι εξής:

Πίνακας 100: DotProd: Χρόνοι εκτέλεσης χωρίς μεταφορά δεδομένων στη GPU - Alt18

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**
100000000	0.015
200000000	0.024
300000000	0.033
400000000	0.06
500000000	0.06



Σχήμα 31: DotProd: Σύγκριση Alt16, Alt17, Alt18

4.6 Υπολογισμός πρώτων αριθμών

Ο αλγόριθμος αυτής της ενότητας περιγράφει τον υπολογισμό του πλήθους των πρώτων αριθμών που υπάρχουν στο εύρος αριθμών (0, n). Ο αλγόριθμος ακολουθεί μια απλοϊκή μέθοδο υπολογισμού στην οποία διατρέχονται όλοι οι αριθμοί από 2 έως n-1 και ελέγχεται αν υπάρχει τουλάχιστον ένας ακέραιος που τον διαιρεί με μηδενικό υπόλοιπο.

Ο αλγόριθμος υπολογισμού πρώτων αριθμών δεν έχει ιδιαίτερη χρηστικότητα, αλλά εξετάζεται καθώς εμπεριέχει διπλό βρόγχο επανάληψης και δεν απαιτείται δέσμευση μνήμης.

4.6.1 Περιγραφή κοινού τμήματος αλγορίθμου πρώτων αριθμών

Συμβ. 98: Prime Numbers: main()

```
static int prime_number_sweep(int n_hi) {
    return prime_number(n_hi);
}

int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    double start = omp_get_wtime();
    int primes = prime_number_sweep(o.max);
    std::cout << "Execution Time: " <<
                        omp_get_wtime() - start <<
                        " seconds"<< std::endl;
    std::cout << "Prime numbers: " << primes << std::endl;
    return 0;
}
```

4.6.2 Σειριακή εκτέλεση

Η σειριακή εκτέλεση του προγράμματος αποτελείται από δύο επαναληπτικούς βρόγχους. Ο πρώτος διατρέχει το σύνολο του εύρους των ακεραίων, από το 2 έως τον δοσμένο μέγιστο ακέραιο. Για κάθε έναν από τους ακέραιους i ελέγχεται αν υπάρχει τουλάχιστον ένας ακέραιος στο εύρος [2, i) που να διαιρείται με τον ι με μηδενικό υπόλοιπο.

Συμβ. 99: Prime Numbers: Σειριακή εκτέλεση

```
int prime_number(int n) {
    int prime = 0;
    int total = 0;

    for (int i = 2; i <= n; i++) {
        prime = 1;
        for (int j = 2; j < i; j++) {
            if ((i % j) == 0) {
                prime = 0;
                break;
            }
        }
        total += prime;
    }
    return total;
}
```

Πίνακας 101: Prime Numbers: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 102: Prime Numbers: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt1	**Alt2**
10000	0.113	0.114
20000	0.411	0.413
30000	0.890	0.891
40000	1.557	1.560
50000	2.397	2.399
60000	3.414	3.419

4.6.3 Παραλλαγή με parallel for reduction

Συμβ. 100: Prime Numbers: parallel for reduction

```
int prime_number(int n) {
    int total = 0, prime = 1;
#pragma omp parallel for firstprivate(prime) reduction(+ : total)
    for (int i = 2; i <= n; i++) {
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
            prime = 1;
        }
        total += prime;
    }

    return total;
}
```

Πίνακας 103: Prime Numbers: Επιλογές μεταγλώττισης Alt3, Alt4

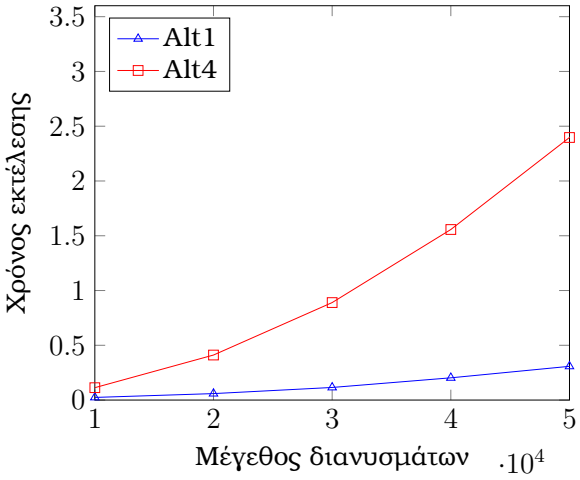
Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 104: Prime Numbers: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt3	**Alt4**
10000	0.024	0.025
20000	0.059	0.061
30000	0.115	0.117
40000	0.203	0.205
50000	0.308	0.306
60000	0.450	0.450
100000	1.148	1.149
130000	2.901	2.879
160000	2.860	2.879
200000	4.376	4.388
250000	6.774	6.785

4.6.3.1 Παρατηρήσεις

Όπως και στα προηγούμενα προβλήματα, η οδηγία pragma omp parallel for σε συνδυασμό με τη φράση reduction συμβάλλει σημαντικά την βελτίωση των επιδόσεων.



Σχήμα 32: Prime Numbers: Σύγκριση Alt1, Alt4

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
1e4	78.6
2e4	85.6
3e4	87.1
4e4	86.9
5e4	87.1
6e4	86.8

Πίνακας 105: Prime Numbers: Ποσοστιαία σύγκριση Alt1 και Alt4

4.6.4 Παραλλαγή με parallel for και critical

Συμβ. 101: Prime Numbers: parallel for - critical

```
int prime_number(int n) {
    int total = 0, prime = 1;
#pragma omp parallel for firstprivate(prime)
    for (int i = 2; i <= n; i++) {
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
            prime = 1;
        }
#pragma omp critical
        total += prime;
    }

    return total;
}
```

Πίνακας 106: Prime Numbers: Επιλογές μεταγλώττισης Alt5, Alt6

Label	Options
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt6

Πίνακας 107: Prime Numbers: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt5	**Alt6**
10000	0.022	0.023
20000	0.054	0.059
30000	0.113	0.109
40000	0.203	0.194
50000	0.297	0.299
60000	0.420	0.423

4.6.4.1 Παρατηρήσεις

Παρατηρείται ότι η αντικατάσταση της οδηγίας reduction με την critical αποδίδει το ίδιο στο συγκεκριμένο πρόβλημα.

4.6.5 Παραλλαγή με simd reduction

Σε αυτή την ενότητα, υλοποιείται μια σειριακή εκτέλεση με τη προσπάθεια εισαγωγής διανυσματικοποίησης μέσω του OpenMP και της φράσης simd reduction. Η αδυναμία εφαρμογής διανυσματικοποίησης από το μεταγλωττιστή αλλά και από τη διεπαφή επιβεβαιώνεται τόσο από τον πινάκα των αποτελεσμάτων αλλά και από τα αρχεία που εξάγονται κατά τη μεταγλώττιση.

Συμβ. 102: Prime Numbers: simd reduction

```
int prime_number(int n) {
    int total = 0;
#pragma omp simd reduction(+ : total)
    for (int i = 2; i <= n; i++) {
        int prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
        }
        total += prime;
    }

    return total;
}
```

Πίνακας 108: Prime Numbers: Επιλογές μεταγλώττισης Alt7, Alt8, Alt9

Label	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -fopenmp -o ./builds/Alt9

Πίνακας 109: Prime Numbers: Αποτελέσματα Alt7, Alt8, Alt9

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		
	Alt7	**Alt8**	**Alt9**
10000	0.115	0.114	0.113
20000	0.412	0.411	0.406
30000	0.893	0.889	0.886
40000	1.557	1.560	1.554
50000	2.396	2.398	2.397
60000	3.417	3.417	3.414

4.6.6 Παραλλαγή με ordered simd - critical

Συμβ. 103: Prime Numbers: ordered simd - critical

```
int prime_number(int n) {
    int total = 0;
#pragma omp ordered simd
    for (int i = 2; i <= n; i++) {
        int prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
        }
#pragma omp critical
        total += prime;
    }

    return total;
}
```

Πίνακας 110: Prime Numbers: Επιλογές μεταγλώττισης Alt10, Alt11, Alt12

Label	Options
Alt10	-fopt-info-vec=builds/alt10.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt10
Alt11	-fopt-info-vec=builds/alt11.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt11
Alt12	-fopt-info-vec=builds/alt12.log -O2 -fno-inline -fopenmp -o ./builds/Alt12

Πίνακας 111: Prime Numbers: Αποτελέσματα Alt10, Alt11, Alt12

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt10	**Alt11**	**Alt12**
10000	0.113	0.113	0.112
20000	0.414	0.411	0.411
30000	0.890	0.891	0.890
40000	1.557	1.558	1.560
50000	2.399	2.400	2.397
60000	3.414	3.417	3.416

4.6.7 Παραλλαγή ordered simd reduction

Συμβ. 104: Prime Numbers: ordered simd - reduction

```
int prime_number(int n) {
    int total = 0;
#pragma omp parallel for simd ordered reduction(+ : total)
    for (int i = 2; i <= n; i++) {
        int prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
        }
        total += prime;
    }
    return total;
}
```

Πίνακας 112: Prime Numbers: Επιλογές μεταγλώττισης Alt13, Alt14, Alt15

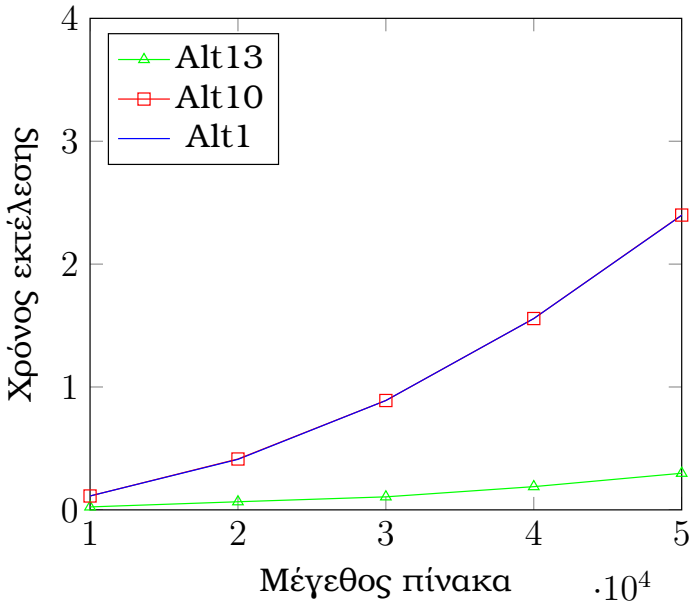
Label	Options
Alt13	-fopt-info-vec=builds/alt13.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt13
Alt14	-fopt-info-vec=builds/alt14.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt14
Alt15	-fopt-info-vec=builds/alt15.log -O2 -fno-inline -fopenmp -o ./builds/Alt15

Πίνακας 113: Prime Numbers: Αποτελέσματα Alt13, Alt14, Alt15

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt13	**Alt14**	**Alt15**
10000	0.024	0.022	0.023
20000	0.066	0.059	0.058
30000	0.108	0.113	0.128
40000	0.189	0.203	0.207
50000	0.298	0.292	0.311
60000	0.415	0.423	0.445

4.6.7.1 Παρατηρήσεις

Τα αποτελέσματα του παραπάνω πίνακα είναι τα ίδια με τις παραλλαγές Alt5. Είναι αναμενόμενο καθώς με την οδηγία simd δεν εφαρμόζεται διανυσματικοποίηση σε καμία παραλλαγή.



Σχήμα 33: Prime Numbers: Σύγκριση Alt13, Alt10, Alt1

4.6.8 Παραλλαγή με offloading (1)

Στη πρώτη υλοποίηση του προβλήματος με μεταφορά του αλγορίθμου στη συσκευή στόχου, χρησιμοποιείται η οδηγία parallel for. Οι χρόνοι εκτέλεσης που καταγράφηκαν είναι υψηλοί σε σύγκριση με προηγούμενες εκτελέσεις, συμπεριλαμβανομένης της σειριακής.

Συμβ. 105: Prime Numbers: target parallel for - reduction

```
int prime_number(int n) {
    int total = 0;
#pragma omp target map(tofrom: total)
#pragma omp parallel for reduction(+ : total)
    for (int i = 2; i <= n; i++) {
        int prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
        }
        total += prime;
    }
    return total;
}
```

Πίνακας 114: Prime Numbers: Επιλογές μεταγλώττισης Alt16

Label	Options
Alt16	-fopt-info-vec=builds/alt16.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt16

Πίνακας 115: Prime Numbers: Αποτελέσματα Alt16

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt16
10000	1.144
20000	1.875
30000	2.976
40000	4.463
50000	6.377
60000	8.840

4.6.9 Παραλλαγή με offloading (2)

Συμβ. 106: Prime Numbers: target parallel for - critical

```
int prime_number(int n) {
    int total = 0;
#pragma omp target map(tofrom: total)
#pragma omp parallel for
    for (int i = 2; i <= n; i++) {
        int prime = 1;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                prime = 0;
                break;
            }
        }
#pragma omp critical
        total += prime;
    }

    return total;
}
```

Πίνακας 116: Prime Numbers: Επιλογές μεταγλώττισης Alt17

Label	Options
Alt17	-fopt-info-vec=builds/alt17.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt17

Πίνακας 117: Prime Numbers: Αποτελέσματα Alt17

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt17
10000	1.165
20000	1.888
30000	3.005
40000	4.516
50000	6.441
60000	8.847

4.6.9.1 Παρατηρήσεις

Όπως και στις αντίστοιχες υλοποιήσεις στη CPU, οι υλοποιήσεις Alt17 - Alt16 εμφανίζουν ίδια συμπεριφορά μεταξύ τους. Τα αποτελέσματα ωστόσο είναι χειρότερα από αυτά της CPU.

4.6.10 Παραλλαγή με offloading (3)

Στη παραλλαγή αυτής της παραγράφου, εφαρμόζονται υπολογισμοί στη GPU και συγκεκριμένα οι οδηγίες teams, distribute parallel for. Ακόμη, χρησιμοποιήθηκε η οδηγία omp cancel for η χρησιμότητα της είναι ίδια με την εντολή break στη σειριακή εκτέλεση.

Συμβ. 107: target teams reduction distribute parallel for

```
int prime_number(int n) {
    int total = 0;
#pragma omp target map(tofrom: total)
#pragma omp teams reduction (+:total)
#pragma omp distribute
    for (int i = 2; i <= n; i++) {
    int prime = 1;
#pragma omp parallel
    {
#pragma omp for
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
            prime = 0;
#pragma omp cancel for
            }
        }
    }
        total += prime;
    }
     return total;
}
```

Πίνακας 118: Prime Numbers: Επιλογές μεταγλώττισης Alt21

Label	Options
Alt21	-fopt-info-vec=builds/alt21.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt21

Πίνακας 119: Prime Numbers: Αποτελέσματα Alt21

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt21
10000	0.946
20000	1.173
30000	1.507
40000	1.944
50000	2.535
60000	3.205

4.6.11 Παραλλαγή με offloading (4)

Συμβ. 108: Prime Numbers: target teams distribute parallel for - reduction

```
int prime_number(int n) {
     int total = 0;

#pragma omp target teams
        distribute parallel for map(tofrom: total) \
        reduction(+ : total)
     for (int i = 2; i <= n; i++) {
     int prime = 1;
         for (int j = 2; j < i; j++) {
             if (i % j == 0) {
                 prime = 0;
                 break;
             }
         }
         total += prime;
     }

     return total;
 }
```

Πίνακας 120: Prime Numbers: Επιλογές μεταγλώττισης Alt22

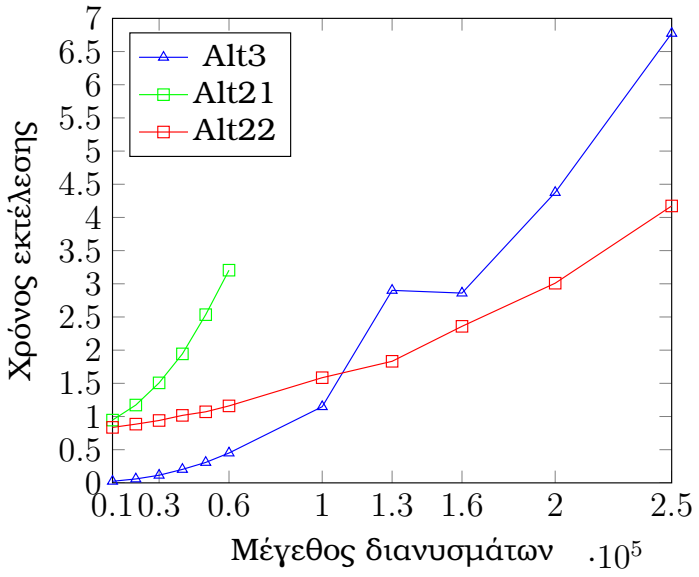
Label	Options
Alt22	-fopt-info-vec=builds/alt22.log -O2 -fno-inline -fno-stack-protector -foffload=nvptx-none="-O2 -fno-inline" -fopenmp -o ./builds/Alt22

Πίνακας 121: Prime Numbers: Αποτελέσματα Alt22

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt22
10000	0.836
20000	0.885
30000	0.940
40000	1.017
50000	1.071
60000	1.160
100000	1.585
130000	1.832
160000	2.359
200000	3.009
250000	4.173

4.6.11.1 Παρατηρήσεις

Στη παραλλαγή της παραγράφου χρησιμοποιήθηκε ένας συνδυασμός οδηγιών που απέδωσε καλύτερα σε προηγούμενα παραδείγματα. Αξιοσημείωτη παρατήρηση είναι ότι το παράδειγμα δεν αποδίδει για μικρά μεγέθη, καθώς συγκρινόμενο με την καλύτερη υλοποίηση (Alt3) εκτελείται σε μεγαλύτερο χρόνο. Ωστόσο κατά την αύξηση του μεγέθους στο πρόβλημα, παρατηρείται καλύτερη συμπεριφορά σε σχέση με τις υπόλοιπες παραλλαγές, όπως φαίνεται στο διάγραμμα που ακολουθεί. Η παρατήρηση δικαιολογείται, καθώς φαίνεται στα προηγούμενα προβλήματα η αντιστοίχιση μεταβλητών προς και από τη συσκευή έχει σημαντικό χρονικό αντίκτυπο. Στον υπολογισμό πρώτων αριθμών αντιστοιχίζεται μια μεταβλητή και όχι πίνακας με δεδομένα). Επομένως είναι αναμενόμενο ότι όσο αυξάνεται το μέγεθος προβλήματος θα υπάρχουν περισσότεροι υπολογισμοί, στους οποίους η GPU έχει πλεονέκτημα έναντι της CPU.



Σχήμα 34: Prime Numbers: Σύγκριση Alt22, Alt21, Alt3

4.7 Διάσχιση συνδεδεμένης λίστας

Η διάσχιση συνδεδεμένης λίστας αποτελεί μια από τις πιο βασικές εργασίες σε κάθε σενάριο προβλήματος που εμπεριέχει συνδεδεμένες λίστες. Με το όρο διάσχιση, ορίζεται η επίσκεψη σε κάθε κόμβο της λίστας τουλάχιστον μια φορά, με σκοπό να εκτελεστεί κάποια λειτουργία σε αυτόν. Στο πρόβλημα που ακολουθεί, οι κόμβοι της λίστας αποτελούνται από έναν ακέραιο ο κάθε ένας.

Ως εργασία της σε κάθε κόμβο, εφαρμόζεται αδρανοποίηση στο συγκεκριμένο νήμα που διατρέχει τον κόμβο, για τυχαίο χρόνο εύρους (1, 5) νανοδευτερολέπτων. Για επαλήθευση της σωστής εκτέλεσης της κάθε παραλλαγής, θα πρέπει το άθροισμα των ακεραίων κάθε κόμβου να ισούται με $\sum_{k=0}^{N-2} k$

4.7.1 Περιγραφή κοινού τμήματος διάσχισης συνδεδεμένης λίστας

Συμβ. 109: Linked List Traversal: main()

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    srand(time(nullptr));

    Llist<int> l(0);
    int verify_val = fillList(l, o);

    auto start = omp_get_wtime();
    int ret = calculate(l);
    auto end = omp_get_wtime();

    if (verify_val != ret) {
        std::cout << "Error: Expected : " << verify_val <<
          " Got: " << ret << std::endl;
        exit(1);
    }

    std::cout << "Execution Time : " << std::fixed
        << std::setprecision(3) << end - start;
    std::cout << " sec " << std::endl;


    return 0;
}
```

Στο βασικό κορμό του προβλήματος, δημιουργείται η συνδεδεμένη λίστα μέσω της ρουτίνας *fillList* που δέχεται ως όρισμα ένα αντικείμενο τύπου *Llist<int>* και το μέγεθος των κόμβων από τους οποίους θα αποτελείται. Στη συνέχεια καλείται η *calculate* που διαφοροποιείται σε κάθε παραλλαγή. Τέλος, γίνεται επαλήθευση του αποτελέσματος και η καταγραφή του χρόνου εκτέλεσης.

Συμβ. 110: Linked List Traversal: fillList()

```
int fillList(Llist<int> &list, Opts &o) {
    Lnode<int> *head = list.head();
    Lnode<int> *temp = head;
    int verify_val = 0;
    for (size_t i = 0; i < o.size - 1; ++i) {
        temp = temp->addNode(i);
        verify_val += i;
    }
    return verify_val;
}
```

Συμβ. 111: Linked List Traversal: Llist

```
template<typename T>
class Llist {
public:
    Llist(T val) {
        head_ = new Lnode<T>(val);
    }
    Lnode<T> *head() { return head_; }

    ~Llist() {
        delete head_;
    }

    size_t size() const {
        Lnode<T> *node = head_;
        size_t size = 0;
        for (; node;) {
            ++size;
            node = node->next();
        }
        return size;
    }

    void forEveryNode(FOR_EVERY_FUN fn, void *args) {
        head_->forNext(fn, args);
    }

private:
        Lnode<T> *head_ = nullptr;
};
```

Συμβ. 112: Linked List Traversal: Lnode

```
template<typename T>
class Lnode {
public:
    explicit Lnode(T val) : val_(val) {}
    Lnode<T> *addNode(T val) {
        next_ = new Lnode<T>(val);
        return next_;
    }

    ~Lnode() {
        if (!next_) {
            delete next_;
        }
    }

    const T &data() const { return val_;}
    Lnode<T> *next() const { return next_; }

    void forNext(FOR_EVERY_FUN fn, void *args) {
        fn(*this, args);
        if (next_) {
            next_->forNext(fn, args);
        }
    }

private:
        T val_;
        Lnode<T> *next_ = nullptr;
 };
```

4.7.2 Σειριακή εκτέλεση

Η πρώτη παραλλαγή του προβλήματος αφορά την σειριακή εκτέλεσή του. Στη ρουτίνα *calculate* εισάγεται ως όρισμα η αρχή της συνδεδεμένης λίστας και στη συνέχεια για κάθε κόμβο της σειριακά, εφαρμόζεται αδράνεια για ένα μικρό χρονικό διάστημα. Η παραλλαγή αυτή θα χρησιμοποιηθεί ως σημείο αναφοράς για την εξαγωγή συμπερασμάτων σχετικά με τις παράλληλες εκτελέσεις.

Συμβ. 113: Linked List Traversal: Σειριακή εκτέλεση

```
using namespace std;
void dowork(Lnode<int> &node, void *args) {
     int *num_nodes = static_cast<int *>(args);
     this_thread::sleep_for(
        chrono::nanoseconds(rand() % 5 + 1));
     *num_nodes += node.data();
 }

 int calculate(Llist<int> &l) {

     int number_of_nodes = 0;
     l.forEveryNode(dowork, &number_of_nodes);

     return number_of_nodes;
 }
```

Πίνακας 122: Linked List Traversal: Επιλογές μεταγλώττισης Alt1

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fopenmp -o ./builds/Alt1

Πίνακας 123: Linked List Traversal: Αποτελέσματα Alt1

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)
	Alt1
10000	0.762
20000	1.508
30000	2.243
40000	2.973
50000	3.770
60000	4.516
70000	5.275

4.7.2.1 Παρατηρήσεις

Η βασικότερη παρατήρηση που μπορεί να γίνει σχετικά με την εκτέλεση του προβλήματος για διαφορετικά μεγέθη, είναι η γραμμική πολυπλοκότητα. Με άλλα λόγια, αν διπλασιαστεί το μέγεθος της συνδεδεμένης λίστας, διπλασιάζεται και ο χρόνος εκτέλεσης.

4.7.3 Παραλλαγή με parallel for

Η παράλληλη διαχείριση των κόμβων της συνδεδεμένης λίστας είναι ένα δύσκολο πρόβλημα προς επίλυση, καθώς η προσπέλαση του κάθε κόμβου εξαρτάται από τον αμέσως προηγούμενο. Στην παραλλαγή που ακολουθεί, γίνεται προσπάθεια παραλληλοποίησης του προβλήματος με τη χρήση της κλασικής οδηγίας *parallel for*. Τα δεδομένα του προβλήματος μεταφέρονται σε ένα *std::vector* πάνω στο οποίο εφαρμόζεται η οδηγία.

Συμβ. 114: Linked List Traversal: parallel for()

```
using namespace std;

int calculate(Llist<int> &l) {

    Lnode<int> *node = l.head();
    size_t nodes_num = l.size();
    std::vector<int> nodes(nodes_num);

    for (size_t i = 0; i < nodes_num; ++i) {
        nodes[i] = node->data();
        node = node->next();
    }


    int num_nodes = 0;
#pragma omp parallel for shared(num_nodes)
    for (size_t i = 0; i < nodes_num; ++i) {
        this_thread::sleep_for(
                        chrono::milliseconds(randI()));
#pragma omp atomic
        num_nodes += nodes[i];
    }

    return num_nodes;
}
```

Πίνακας 124: Linked List Traversal: Επιλογές μεταγλώττισης Alt2, Alt3

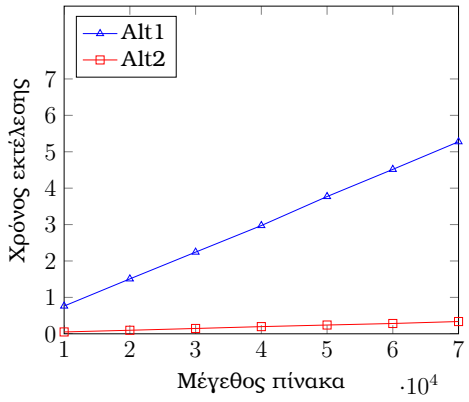
Label	Options
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-tree-vectorize -fno-inline -fopenmp -o ./builds/Alt2
Alt3	-fopt-info-vec=builds/alt3.log -O2 -ftree-vectorize -fno-inline -fopenmp -o ./builds/Alt3

Πίνακας 125: Linked List Traversal: Αποτελέσματα Alt2, Alt3

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt2	**Alt3**
10000	0.051	0.051
20000	0.097	0.099
30000	0.146	0.146
40000	0.197	0.192
50000	0.241	0.243
60000	0.283	0.291
70000	0.337	0.334

4.7.3.1 Παρατηρήσεις

Παρατηρείται εμφανής μείωση του χρόνου εκτέλεσης σε σύγκριση με τη σειριακή εκτέλεση του προβλήματος. Παρόλα αυτά, υπάρχουν αρνητικές επιπτώσεις με τη βασικότερη να είναι η ανάγκη μεταφοράς των κόμβων της συνδεδεμένης λίστας σε σειριακή διάταξη. Κάτι τέτοιο, όχι μόνο ακυρώνει την έννοια της συνδεδεμένης λίστας, αλλά μπορεί και να μην είναι εφικτό καθώς για μεγάλα μεγέθη προβλήματος είναι πιθανό να μην υπάρχει ο απαιτούμενος χώρος για μια τέτοια ενέργεια.



Σχήμα 35: Linked List Traversal: Σύγκριση Alt1, Alt2

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
10000	93.6
20000	93.4
30000	93.5
40000	93.8
50000	93.6
60000	93.7
70000	93.7

Πίνακας 126: Linked List Traversal: Ποσοστιαία σύγκριση μεταξύ Alt1 και Alt2

4.7.4 Παραλλαγή με parallel for (2)

Συμβ. 115: Linked List Traversal: parallel for schedule(static)

```
int calculate(Llist<int> &l) {
    Lnode<int> *node = l.head();
    size_t nodes_num = l.size();
    std::vector<int> nodes(nodes_num);

    for (size_t i = 0; i < nodes_num; ++i) {
        nodes[i] = node->data();
        node = node->next();
    }
    int num_nodes = 0;

#pragma omp parallel for schedule(static, 10) shared(num_nodes)
    for (size_t i = 0; i < nodes_num; ++i) {
        this_thread::sleep_for(chrono::nanoseconds(randI()));
#pragma omp atomic
        num_nodes += nodes[i];
    }

    return num_nodes;
}
```

Πίνακας 127: Linked List Traversal: Επιλογές μεταγλώττισης Alt4, Alt5

Label	Options
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-tree-vectorize -fno-inline -fopenmp -o ./builds/Alt4
Alt5	-fopt-info-vec=builds/alt5.log -O2 -ftree-vectorize -fno-inline -fopenmp -o ./builds/Alt5

Πίνακας 128: Linked List Traversal: Αποτελέσματα Alt4, Alt5

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt4	Alt5
10000	0.052	0.052
20000	0.100	0.099
30000	0.148	0.147
40000	0.197	0.193
50000	0.241	0.242
60000	0.284	0.283
70000	0.339	0.336

4.7.4.1 Παρατηρήσεις

Συγκριτικά με τις εκτελέσεις Alt2 και Alt3 οι χρόνοι εκτέλεσης παραμένουν αμετάβλητοι. Η εισαγωγή της φράσης schedule δεν επηρεάζει στο συγκεκριμένο παράδειγμα. Ο λόγος είναι ότι ο μέσος όρος εκτέλεσης της εργασίας σε κάθε κόμβο είναι σταθερός.

4.7.5 Παραλλαγή με tasks

Στη παραλλαγή αυτής της παραγράφου γίνεται χρήση της οδηγίας *task*. Συγκεκριμένα, ο αλγόριθμος εισέρχεται σε ένα παράλληλο τμήμα όπου δημιουργείται μια ομάδα νημάτων. Ένα από αυτά τα νήματα είναι υπεύθυνο για την δημιουργία των ξεχωριστών εργασιών ενώ τα υπόλοιπα αναμένουν στο υποκείμενο φράγμα που βρίσκεται στο τέλος της παράλληλης περιοχής για την δημιουργία τους. Η φράση nowait επιτρέπει την ταυτόχρονη δημιουργία εργασιών και εκτέλεση τους από τα νήματα που έχουν παραβλέψει το υποκείμενο φράγμα της pragma omp single. Το πλεονέκτημα της διαδικασίας είναι ότι δεν απαιτείται επιπλέον χρόνος και μνήμη για την αποθήκευση των κόμβων ή των δεδομένων σε σειριακή ακολουθία.

Συμβ. 116: Linked List Traversal: task - atomic

```
using namespace std;

int calculate(Llist<int> &l) {
    Lnode<int> *node = l.head();

    int num_nodes = 0;
#pragma omp parallel shared(num_nodes)
    {
#pragma omp single nowait
        {
            while (node) {
#pragma omp task shared(node) mergeable
                {
                    this_thread::sleep_for(
                        chrono::nanoseconds(randI()));
#pragma omp atomic
                    num_nodes += node->data();
                }
                node = node->next();
            }
        }
    }
    return num_nodes;
}
```

Πίνακας 129: Linked List Traversal: Επιλογές μεταγλώττισης Alt6, Alt7

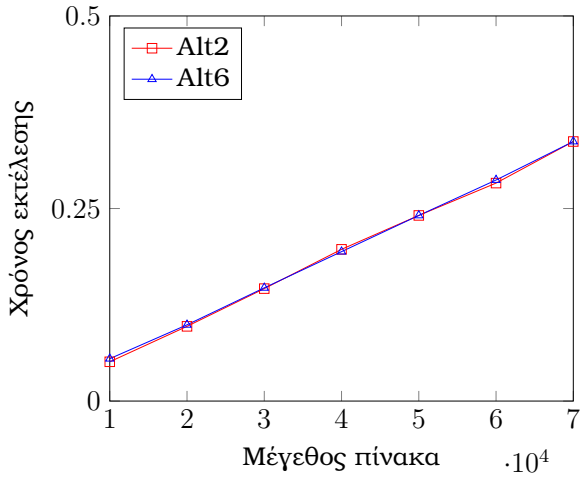
Label	Options
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-tree-vectorize -fno-inline -fopenmp -o ./builds/Alt6
Alt7	-fopt-info-vec=builds/alt7.log -O2 -ftree-vectorize -fno-inline -fopenmp -o ./builds/Alt7

Πίνακας 130: Linked List Traversal: Αποτελέσματα Alt6, Alt7

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt6	Alt7
10000	0.055	0.051
20000	0.099	0.099
30000	0.147	0.147
40000	0.194	0.195
50000	0.241	0.241
60000	0.287	0.288
70000	0.337	0.339

4.7.5.1 Παρατηρήσεις

Η χρήση των εργασιών αποτελεί την καλύτερη λύση για τέτοιου είδους προβλήματα. Οι εργασίες εκτελούνται τυχαία άλλα μπορεί να εκτελεστούν και με τη σειρά με την οποία δημιουργήθηκαν με την εισαγωγή της φράσης *ordered*. Πρέπει ωστόσο να δοθεί ιδιαίτερη προσοχή στον αριθμό των εργασιών που δημιουργούνται σε συνάρτηση με το χρόνο που απαιτείται για την εκτέλεση της κάθε εργασίας. Αν ο χρόνος εκτέλεσης των εργασιών είναι μικρότερος από το χρόνο που απαιτείται για την δημιουργία της εργασίας, τότε ο χρόνος εκτέλεσης θα μεγαλώσει. Όπως φαίνεται και στο παρακάτω παράδειγμα, ο χρόνος εκτέλεσης είναι συγκρίσιμος με την παραλλαγή Alt2. Η απώλεια ανάγκης για δέσμευση επιπλέον μνήμης καθιστά την υλοποίηση αυτής της παραγράφου καλύτερη.



Σχήμα 36: Linked List Traversal: Σύγκριση Alt2, Alt6

4.8 Παράδειγμα Producer-Consumer

Στον προγραμματισμό το πρόβλημα *producer-consumer*, γνωστό και ως *bounded buffer* πρόβλημα, αποτελεί ένα κλασσικό πρόβλημα συγχρονισμού πολλαπλών διαδικασιών. Το πρόβλημα περιλαμβάνει δύο επαναλαμβανόμενες εργασίες, τον παραγωγό*(producer)* και τον καταναλωτή*(consumer)* που μοιράζονται ένα κοινό *buffer* σταθερού μεγέθους που χρησιμοποιείται ως ουρά(*queue*). Ο παραγωγός δημιουργεί συνεχώς δεδομένα και τα εισάγει στο *buffer* ενώ ο καταναλωτής διαβάζει και διαγράφει επαναλαμβανόμενα τα δεδομένα από αυτό.[6]

Στην παρούσα υλοποίηση οι εργασίες που παράγονται και εκτελούνται περιλαμβάνουν την κλήση ρουτινών που δεν έχουν κάποια ιδιαίτερη λειτουργικότητα, παρά μόνο αδρανοποιούν το εκάστοτε νήμα για 60 και 125 *milliseconds* αντίστοιχα. Ταυτόχρονα, στη εργασία καταναλωτή επιστρέφεται και ένας ακέραιος με σκοπό την επαλήθευση σωστής εκτέλεσης του αλγορίθμου.

4.8.1 Περιγραφή κοινού τμήματος αλγορίθμου Producer-Consumer

Συμβ. 117: Prod-Cons: main()

```
int main(int argc, char **argv)
 {
     Opts o;
     parseArgs(argc, argv, o);

     double start = omp_get_wtime();
     int total = producer_consumer(o.size);
     double end = omp_get_wtime();

     int ver = 0;
     for (int i = 0; i < o.size; i++) {
         ver += i;
     }

     if (ver != total) {
         std::cout << "Failed Verification!" << std::endl;
         exit(1);
     }

     std::cout << "Calculation time: " <<
             end - start << " sec" << std::endl;
     return 0;
 }
```

4.8.2 Σειριακή εκτέλεση

Συμβ. 118: Prod-Cons: Σειριακή εκτέλεση

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = gl_buffer->buf_[--gl_buffer->len_].x_;
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
    gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);

    int total = 0;

    for (int i = 0; i < iterations; ++i) {
        produce(i);
        total += consume();
    }

    return total;
}
```

Πίνακας 131: Prod-Cons: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 132: Prod-Cons: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt1	Alt2
10	0.303	0.303
20	0.606	0.615
30	0.908	0.910
40	1.211	1.210
50	1.514	1.514
60	1.817	1.832
70	2.120	2.127
80	2.422	2.398
90	2.725	2.727
100	3.028	3.014

4.8.2.1 Παρατηρήσεις

Ο χρόνος εκτέλεσης του προβλήματος για *k* επαναλήψεις αναλύεται ως εξής:

$$\sum_{n=1}^{k} p_n + c_n = total\_time$$

Για 100 επαναλήψεις, ο ελάχιστος χρόνος που χρειάζεται για την ολοκλήρωση του αλγόριθμου είναι $10 * 100 + 20 * 100 = 3000 milliseconds$, όπως επαληθεύεται και απο τον προηγούμενο πίνακα. Επίσης, ο παρακάτω πίνακας επιβεβαιώνει οτι τα βήματα του αλγόριθμου, είναι εναλλάξ παραγωγή - κατανάλωση.

Πίνακας 133: Prod-Cons: Alt1: Βήματα εργασιών

Βήματα εκτέλεσης	
1-5	**6-10**
Produce: 0	Produce: 6
Consume: 0	Consume: 6
Produce: 1	Produce: 7
Consume: 1	Consume: 7
Produce: 2	Produce: 8
Consume: 2	Consume: 8
Produce: 3	Produce: 9
Consume: 3	Consume: 9
Produce: 4	Produce: 10
Consume: 4	Consume: 10
Produce: 5	
Consume: 5	

4.8.3 Παραλλαγή με parallel for reduction

Όπως φαίνεται και στον κώδικα που ακολουθεί, ο βρόγχος επανάληψης των εργασιών εκτελείται παράλληλα. Ως συνέπεια, κάθε νήμα είναι υπεύθυνο για την εκτέλεση μιας εργασίας αμέσως μετά την παραγωγή μιας άλλης. Η εργασία που εκτελείται δεν είναι απαραίτητα ή ίδια με αυτή που παράγεται από το ίδιο νήμα.

Συμβ. 119: Prod-Cons: parallel for reduction - critical

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = 0;
#pragma omp critical
    {
        num = gl_buffer->buf_[--gl_buffer->len_].x_;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
#pragma omp critical
    {
        gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    long long int total = 0;

#pragma omp parallel for reduction(+: total)
    for (int i = 0; i < iterations; ++i) {
        produce(i);
        total += consume();
    }

    return total;
}
```

Πίνακας 134: Prod-Cons: Επιλογές μεταγλώττισης Alt3, Alt4

Labe	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 135: Prod-Cons: Αποτελέσματα Alt3, Alt4

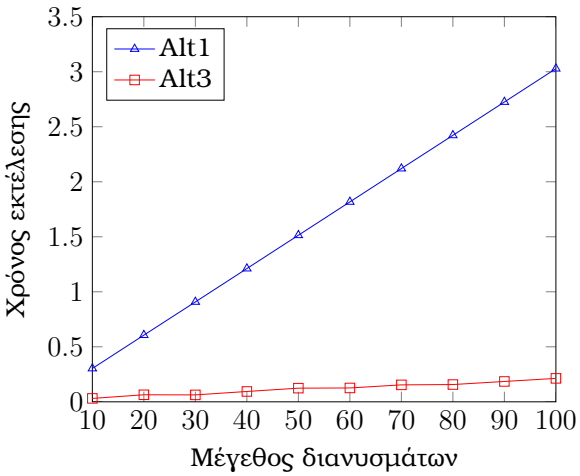
Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt3	**Alt4**
10	0.031	0.031
20	0.064	0.065
30	0.063	0.063
40	0.094	0.093
50	0.124	0.126
60	0.126	0.127
70	0.154	0.153
80	0.157	0.157
90	0.185	0.186
100	0.213	0.215

4.8.3.1 Παρατηρήσεις

Κατα την εκτέλεση του αλγορίθμου, αρχικά γίνονται μαζικές παραγωγές δεδομένων παράλληλα από διαφορετικά νήματα. Στη συνέχεια ακολουθεί η κατανάλωση τους. Η διαδικασία ολοκληρώνεται όταν τελειώσουν τα δεδομένα προς παραγωγή.

Πίνακας 136: Prod-Cons: Alt3: Βήματα εργασιών

Βήματα εκτέλεσης		
1-16	**17-32**	**33-40**
Produce: 19	Consume: 18	Produce: 1
Produce: 0	Consume: 16	Produce: 3
Produce: 14	Consume: 10	Produce: 5
Produce: 12	Consume: 11	Produce: 7
Produce: 8	Consume: 6	Consume: 7
Produce: 13	Consume: 4	Consume: 4
Produce: 9	Consume: 17	Consume: 3
Produce: 15	Consume: 2	Consume: 1
Produce: 2	Consume: 15	
Produce: 17	Consume: 19	
Produce: 4	Consume: 13	
Produce: 6	Consume: 8	
Produce: 11	Consume: 12	
Produce: 10	Consume: 14	
Produce: 16	Consume: 0	
Produce: 18	Consume: 19	



Σχήμα 37: Prod-Cons: Σύγκριση Alt1, Alt3

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
10	89.76
20	89.43
30	93.03
40	92.24
50	91.81
60	93.06
70	92.73
80	93.51
90	93.21
100	92.97

Πίνακας 137: Prod-Cons: Ποσοστιαία σύγκριση μεταξύ Alt1 και Alt3

4.8.4 Παραλλαγή με parallel for (2)

Συμβ. 120: Prod-Cons: parallel - critical

```
Buffer *gl_buffer = 0;

int consume()
{
    int num = 0;
#pragma omp critical
    {
        if (gl_buffer->len_ > 0) {
            num = gl_buffer->buf_[--gl_buffer->len_].x_;
        }
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
    return num;
}

void produce(int key)
{
#pragma omp critical
    {
       gl_buffer->buf_[gl_buffer->len_++].x_ = key;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
}

int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int finished_production = 0;
    int abort = 0;

#pragma omp parallel shared(finished_production) firstprivate(abort)
    {
        if (omp_get_thread_num() == 0) {
            for (int i = 0; i < iterations; ++i) {
                produce(i);
            }
            finished_production = 1;
        } else {
            while (!abort) {
                int temp = consume();
#pragma omp critical
                {
                    total += temp;
                    if (finished_production && gl_buffer->len_ == 0)
                    {
                        abort = 1;
                    }
                }
            }
        }
    }

    return total;
}
```

Στην υλοποίηση αυτής της ενότητας δημιουργείται μια παράλληλη περιοχή και μέσα σε αυτή μόνο ένα νήμα είναι υπεύθυνο για τη δημιουργία των εργασιών. Τα υπόλοιπα νήματα εκτελούν τις εργασίες που βρίσκουν μέσα στο buffer έως ότου αυτές τελειώσουν και ταυτόχρονα έχει τελειώσει και η παραγωγή τους από το αρμόδιο νήμα.

Πίνακας 138: Prod-Cons: Επιλογές μεταγλώττισης Alt5, Alt6

Labe	**Options**
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5

Πίνακας 139: Prod-Cons: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt5	**Alt6**
10	0.225	0.227
20	0.404	0.425
30	0.628	0.651
40	0.829	0.829
50	1.031	1.029
60	1.232	1.223
70	1.434	1.532
80	1.615	1.615
90	1.835	1.823
100	2.038	2.012

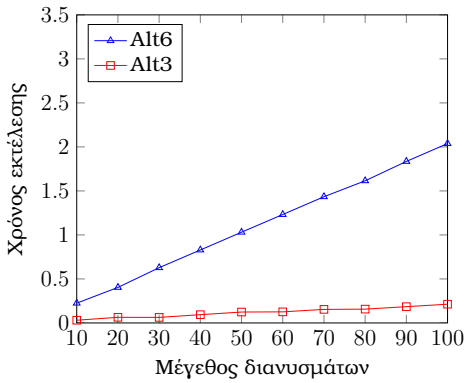
4.8.4.1 Παρατηρήσεις

Η επίδοση του αλγόριθμου υστερεί σε σχέση με την Alt5 καθώς μόνο ένα νήμα είναι υπεύθυνο για την παραγωγή των εργασιών. Η μέθοδος αυτή δεν ενδείκνυται, καθώς σημαντικό ρόλο παίζει ο σχετικός χρόνος ανάμεσα στην παραγωγή και την κατανάλωση μιας εργασίας. Τα νήματα που είναι αρμόδια για την κατανάλωση, θα εκτελέσουν άσκοπα το βρόγχο επανάληψης στην περίπτωση που δεν έχει προλάβει να εισαχθεί μια εργασία από τον παραγωγό ή αν ο αριθμός των εργασιών που υπάρχουν μέσα στο buffer είναι μικρότερος από το συνολικό αριθμό νημάτων καταναλωτών που δεν εκτελούν κάποια εργασία.

Πίνακας 140: Prod-Cons: Alt5: Βήματα εργασιών

Βήματα εκτέλεσης		
1-16	**17-32**	**33-40**
Produce: 0	Produce: 8	Produce: 16
Consume: 0	Produce: 9	Produce: 17
Produce: 1	Consume: 9	Consume: 17
Consume: 1	Consume: 8	Consume: 18
Produce: 2	Produce: 10	Produce: 19
Produce: 3	Produce: 11	Produce: 20
Consume: 3	Consume: 11	Consume: 20
Consume: 2	Consume: 10	Consume: 19
Produce: 4	Produce: 12	
Consume: 4	Produce: 13	
Produce: 5	Consume: 13	
Consume: 5	Consume: 12	
Produce: 6	Produce: 14	
Produce: 7	Produce: 15	
Consume: 7	Consume: 15	
Consume: 6	Consume: 14	

Όπως φαίνεται από τον πίνακα καταγραφής της σειράς παραγωγής-εκτέλεσης των εργασιών, κάθε εργασία που δημιουργείται εκτελείται απευθείας απο ένα αδρανές νήμα.



Σχήμα 38: Prime Numbers: Σύγκριση Alt6, Alt3

4.8.5 Παραλλαγή με tasks (depend)

Με την χρήση των *tasks* η επίλυση προβλημάτων τύπου παραγωγού-καταναλωτή επιλύονται με μεγαλύτερη ευκολία. Στην παραλλαγή αυτής της παραγράφου γίνεται επίλυση με χρήση εργασιών. Ο αλγόριθμος εισέρχεται σε ένα παράλληλο τμήμα κώδικα, όπου ένα νήμα είναι υπεύθυνο για την δημιουργία εργασιών που περιλαμβάνουν εργασίες παραγωγής ή κατανάλωσης. Τα tasks δημιουργούνται από ένα νήμα αλλά εκτελούνται απο όλα μόλις ολοκληρωθεί το τμήμα κώδικα που περιλαμβάνεται στην οδηγία pragma omp single. Μια consume εργασία για να εκτελεστεί, θα πρέπει πρώτα να έχει ολοκληρωθεί η αντίστοιχη produce και αυτό επιτυγχάνεται με την φράση depend.

Συμβ. 121: Prod-Cons: task depend - critical

```
int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int x = 0;
#pragma omp parallel
    {
#pragma omp single
        {
            for (int i = 0; i < iterations; ++i) {
#pragma omp task depend(out : x)
                {
                    produce(i);
                    x = 1;
                }
            }
            for (int i = 0; i < iterations; ++i) {
#pragma omp task depend(in : x)
                {
                    int temp = consume();
#pragma omp critical
                    {
                        total += temp;
                    }
                }
            }
        }
    }
    return total;
}
```

Πίνακας 141: Prod-Cons: Επιλογές μεταγλώττισης Alt7, Alt8

Labe	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8

Πίνακας 142: Prod-Cons: Αποτελέσματα Alt7, Alt8

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt7	**Alt8**
10	0.125	0.127
20	0.250	0.249
30	0.349	0.350
40	0.471	0.471
50	0.593	0.592
60	0.696	0.698
70	0.817	0.819
80	0.919	0.919
90	1.043	1.043
100	1.161	1.162

4.8.5.1 Παρατηρήσεις

Πίνακας 143: Prod-Cons: Alt7: Βήματα εργασιών

Βήματα εκτέλεσης		
1-16	**17-32**	**33-40**
Produce : 0	Produce : 16	Consume: 7
Produce : 1	Produce : 17	Consume: 6
Produce : 2	Produce : 18	Consume: 5
Produce : 3	Produce : 19	Consume: 4
Produce : 4	Consume: 19	Consume: 3
Produce : 5	Consume: 18	Consume: 2
Produce : 6	Consume: 17	Consume: 1
Produce : 7	Consume: 16	Consume: 0
Produce : 8	Consume: 15	
Produce : 9	Consume: 14	
Produce : 10	Consume: 13	
Produce : 11	Consume: 12	
Produce : 12	Consume: 11	
Produce : 13	Consume: 10	
Produce : 14	Consume: 9	
Produce : 15	Consume: 8	

4.8.6 Παραλλαγή με taskloop

Στη τελευταία υλοποίηση producer - consumer γίνεται χρήση της οδηγίας *taskloop* η οποία δημιουργεί μια εργασία προς εκτέλεση για κάθε επανάληψη ενός βρόγχου.

Συμβ. 122: Prod-Cons: taskloop simd - atomic

```
int producer_consumer(int iterations)
{
    gl_buffer = new Buffer(iterations);
    int total = 0;
    int x = 0;
#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp taskloop simd
            for (int i = 0; i < iterations; ++i) {
                produce(i);
            }
#pragma omp taskloop
            for (int i = 0; i < iterations; ++i) {
                int temp = consume();
#pragma omp atomic
                total += temp;
            }
        }
    }
    return total;
}
```

Πίνακας 144: Prod-Cons: Επιλογές μεταγλώττισης Alt9, Alt10

Labe	Options
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt9
Alt10	-fopt-info-vec=builds/alt10.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt10

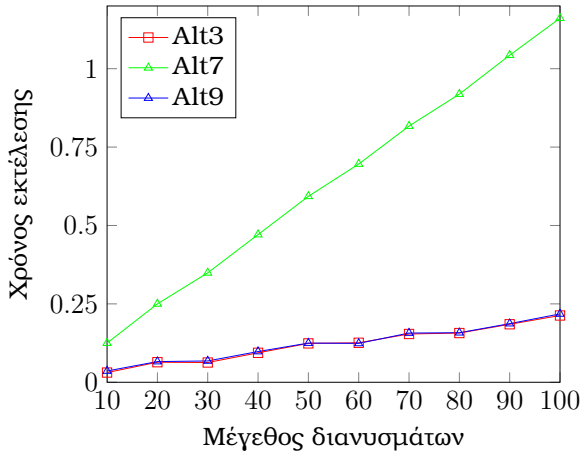
Πίνακας 145: Prod-Cons: Αποτελέσματα Alt9, Alt10

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt9	**Alt10**
10	0.036	0.036
20	0.066	0.067
30	0.068	0.064
40	0.098	0.095
50	0.125	0.126
60	0.124	0.129
70	0.157	0.155
80	0.158	0.156
90	0.187	0.190
100	0.218	0.217

4.8.6.1 Παρατηρήσεις

Πίνακας 146: Prod-Cons: Alt9: Βήματα εργασιών

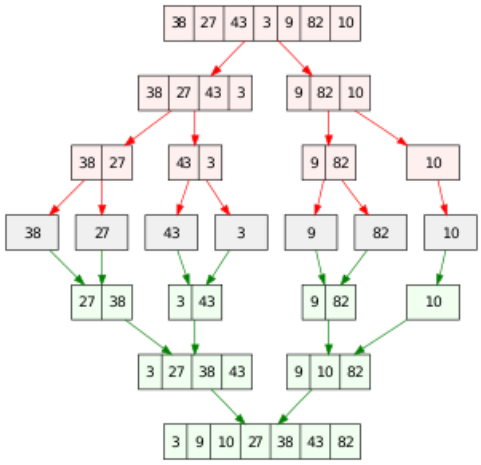
Βήματα εκτέλεσης		
1-16	**17-32**	**33-40**
Produce : 0	Produce : 1	Consume: 8
Produce : 18	Produce : 7	Consume: 4
Produce : 6	Produce : 5	Consume: 2
Produce : 19	Produce : 3	Consume: 9
Produce : 9	Consume: 3	Consume: 19
Produce : 2	Consume: 5	Consume: 6
Produce : 4	Consume: 7	Consume: 18
Produce : 8	Consume: 1	Consume: 0
Produce : 10	Consume: 17	
Produce : 11	Consume: 15	
Produce : 14	Consume: 16	
Produce : 13	Consume: 12	
Produce : 12	Consume: 13	
Produce : 16	Consume: 14	
Produce : 15	Consume: 11	
Produce : 17	Consume: 10	



Σχήμα 39: Prod-Cons: Σύγκριση Alt3, Alt9

4.9 Παράδειγμα ταξινόμησης mergesort

Η ταξινόμηση *mergesort* είναι ένας αλγόριθμος που κατατάσσεται στην κατηγορία Divide and Conquer. Πρόκειται για έναν αναδρομικό αλγόριθμο στο οποίο δοθέντος ενός διανύσματος προς ταξινόμηση, αυτό διαιρείται σε δύο υποσύνολα και για κάθε υποσύνολο εφαρμόζεται η ίδια διαδικασία ταξινόμησης. Όταν ολοκληρωθεί τα δυο υποσύνολα ενώνονται ξανά σε ένα. Η διαδικασία περιγράφεται στο παρακάτω διάγραμμα[39]



Σχήμα 40: Mergesort παράδειγμα

4.9.1 Περιγραφή κοινού τμήματος αλγορίθμου ταξινόμησης mergesort

Συμβ. 123: Mergesort: verify()

```
static void verify(int *a, int size)
{
    for (size_t i = 1; i < size; ++i) {
        if (a[i] < a[i-1]) {
            std::cout << "Verification failed! Aborting ..."
             << std::endl;
            exit(1);
        }
    }

    std::cout << "Successful verification" << std::endl;
}
```

Συμβ. 124: Mergesort: merge()[4]

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int *L = new int[n1];
    int *R = new int[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    delete []L;
    delete []R;
}
```

Συμβ. 125: Mergesort: main()

```
int main(int argc, char **argv)
{
    Opts o;
    parseArgs(argc, argv, o);

    int *arr = new int[o.size];
    fill_random_array(arr, o.size);
    //int arr[] = { 12, 11, 13, 5, 6, 7 };

    auto start = omp_get_wtime();
    mergeSort_wrapper(arr, 0, o.size - 1);
    auto end = omp_get_wtime();
    std::cout << "Execution time: " << std::fixed << end-start <<
        std::setprecision(5) << std::endl;
    if (o.verify) {
        verify(arr, o.size);
    }
    return 0;
}
```

4.9.2 Σειριακή εκτέλεση

Συμβ. 126: Mergesort: Σειριακή Εκτέλεση

```
void mergeSort(int arr[], int l, int r)
{

    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
    mergeSort(arr, lhs, rhs);
}
```

Πίνακας 147: Mergesort: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 148: Mergesort: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt1	**Alt2**
2000000	0.557	0.505
3000000	0.838	0.775
4000000	1.145	1.098
5000000	1.392	1.315
6000000	1.714	1.587
7000000	1.999	1.882

4.9.3 Παραλλαγή με task

Στο παράδειγμα της παραγράφου, οι ρουτίνες mergesort εισάγονται σε εργασίες με σκοπό τη παράλληλη εκτέλεσή τους. Παρόλο που η επίλυση εξάγει σωστά αποτελέσματα, εμφανίζει μεγάλη καθυστέρηση στο χρόνο εκτέλεσης, όπως φαίνεται στον πίνακα που ακολουθεί, γιαυτό δεν αυξάνεται το μέγεθος του προβλήματος.

Συμβ. 127: Mergesort: task - taskwait

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
        // Sort first and second halves
#pragma omp task
        {
            mergeSort(arr, l, m);
        }
#pragma omp task
        {
            mergeSort(arr, m + 1, r);
        }
#pragma omp taskwait
        merge(arr, l, m, r);
    }
}
void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
#pragma omp parallel
#pragma omp single
    {
        mergeSort(arr, lhs, rhs);
    }
}
```

Πίνακας 149: Mergesort: Επιλογές μεταγλώττισης Alt3, Alt4

Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 150: Mergesort: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt3	**Alt4**
2000000	28.648	27.689

4.9.4 Παραλλαγή με task (2)

Συμβ. 128: Mergesort: task - depend

```
void mergeSort(int arr[], int l, int r)
{
    int x, y;
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
#pragma omp task depend(in: x)
        {
            mergeSort(arr, l, m);
        }

#pragma omp task depend(in: y)
        {
            mergeSort(arr, m + 1, r);
        }
#pragma omp task depend(out: x, y)
        {
            merge(arr, l, m, r);
        }
#pragma omp taskwait
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
#pragma omp parallel
#pragma omp single
    {
        mergeSort(arr, lhs, rhs);
    }
}
```

Πίνακας 151: Mergesort: Επιλογές μεταγλώττισης Alt5, Alt6

Label	Options
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt6

Πίνακας 152: Mergesort: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt5	Alt6
2000000	36.707	42.075

4.9.5 Παραλλαγή με task (3)

Σε μια ακόμα προσπάθεια επίλυσης με tasks, εισάγεται η φράση *final* με σκοπό τη μείωση του παραγόμενου αριθμού εργασιών. Λόγω του μεγάλου μεγέθους διανύσματος οι εργασίες που δημιουργούνται είναι τόσες, που το κόστος της δημιουργίας τους επιβαρύνει σημαντικά στο συνολικό χρόνο εκτέλεσης του προβλήματος.

Συμβ. 129: Mergesort: task - taskwait - critical

```
int LIMIT = omp_get_num_threads();
int num_rec = 0;

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
#pragma omp critical
    {
        ++num_rec;
    }

    int x, y;
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
        // Sort first and second halves
#pragma omp task final(num_rec > LIMIT)
        {
            mergeSort(arr, l, m);
        }

#pragma omp task final(num_rec > LIMIT)
        {
            mergeSort(arr, m + 1, r);
        }
#pragma omp taskwait
        {
            merge(arr, l, m, r);
        }
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
#pragma omp parallel
#pragma omp single
    {
        mergeSort(arr, lhs, rhs);
    }
}
```

Πίνακας 153: Mergesort: Επιλογές μεταγλώττισης Alt9, Alt10

Label	Options
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt9
Alt10	-fopt-info-vec=builds/alt10.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt10

Πίνακας 154: Mergesort: Αποτελέσματα Alt9, Alt10

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt9	**Alt10**
2000000	2.612	1.759
3000000	3.929	2.460
4000000	5.395	5.331
5000000	6.683	3.893
6000000	7.916	7.394

4.9.6 Παραλλαγή με task (4)

Η τελευταία παραλλαγή της λύσης του προβλήματος, προκύπτει μετά από αρκετές προσπάθειες αποτυχημένων εκτελέσεων. Η δημιουργία νέων εργασιών παύει όταν το μέγεθος του διαιρεμένου διανύσματος είναι μικρότερο από μια προκαθορισμένη τιμή. Σε περίπτωση που το μέγεθος είναι μικρότερο από WORKLOAD τότε η εργασία ενσωματώνεται στη προηγούμενη.

Συμβ. 130: Mergesort: taskgroup - taskyield

```
#define WORKLOAD 16000

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{

    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
#pragma omp taskgroup
        {
#pragma omp task shared(arr) untied if (r - l >= WORKLOAD)
            mergeSort(arr, l, m);
#pragma omp task shared(arr) untied if (r - l >= WORKLOAD)
            mergeSort(arr, m+1, r);
#pragma omp taskyield
        }
        merge(arr, l, m, r);
    }
}

void mergeSort_wrapper(int *arr, int lhs, int rhs)
{
#pragma omp parallel
#pragma omp single
    mergeSort(arr, lhs, rhs);
}
```

Πίνακας 155: Mergesort: Επιλογές μεταγλώττισης Alt11, Alt12

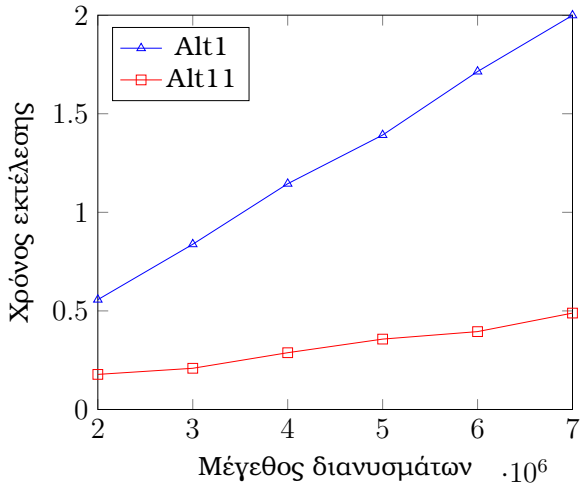
Label	Options
Alt11	-fopt-info-vec=builds/alt11.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt11
Alt12	-fopt-info-vec=builds/alt12.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt12

Πίνακας 156: Mergesort: Αποτελέσματα Alt11, Alt12

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt11	Alt12
2000000	0.178	0.146
3000000	0.209	0.194
4000000	0.288	0.247
5000000	0.357	0.324
6000000	0.395	0.374
7000000	0.489	0.467

4.9.6.1 Παρατηρήσεις

Η παράγραφος αυτή περιέχει τη μοναδική υλοποίηση με χρήση OpenMP που οι χρόνοι εκτέλεσης είναι μικρότεροι από αυτούς της σειριακής. Η χρήση *OpenMP* όπως φάνηκε από τις προηγούμενες προσπάθειες μπορεί να προκαλέσει μείωση των επιδόσεων καθώς το κόστος δημιουργίας νέων εργασιών μπορεί να είναι μεγαλύτερο από το χρόνο εκτέλεσής τους.



Σχήμα 41: MergeSort: Σύγκριση Alt1, Alt11

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
2000000	68
3000000	75
4000000	74
5000000	74
6000000	77
7000000	75

Πίνακας 157: Mergesort: Ποσοστιαία σύγκριση Alt1 και Alt11

4.10 Παράδειγμα ταξινόμησης Quicksort

Ο αλγόριθμος Quicksort αποτελεί έναν από τους αποτελεσματικότερους αλγόριθμους ταξινόμησης. Λειτουργεί επιλέγοντας ένα στοιχείο περιστροφής από τη συστοιχία και χωρίζοντας τα εκατέρωθεν στοιχεία σε δύο υποκατηγορίες ανάλογα αν είναι μικρότερα ή μεγαλύτερα από το στοιχείο περιστροφής. Βασική διαφορά ανάμεσα στους δύο αλγόριθμους είναι ότι ο Mergesort απαιτεί επιπλέον μνήμη για τους υπολογισμούς, ενώ ο Quicksort όχι (*in place algorithm*). Ακόμη, ο Quicksort είναι αποδοτικός για μικρές συστοιχίες ενώ ο Mergesort είναι το ίδιο αποδοτικός για όλα τα μεγέθη.

4.10.1 Περιγραφή κοινού τμήματος αλγορίθμου Quicksort

Συμβ. 131: Quicksort: main()

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);
    int *a = new int[o.size];
    fill_random_arr(a, o.size);

    auto start = omp_get_wtime();
    qsort_wrapper(a, 0, o.size - 1);
    auto end = omp_get_wtime();
    std::cout << "Execution time: " << std::fixed <<
        std::setprecision(3) << end-start <<
        std::setprecision(5);
    std::cout << " sec " << std::endl;

    if (o.verify) {
        verify(a, o.size);
    }
    delete []a;
}
```

4.10.2 Σειριακή εκτέλεση

Συμβ. 132: Quicksort: Σειριακή εκτέλεση

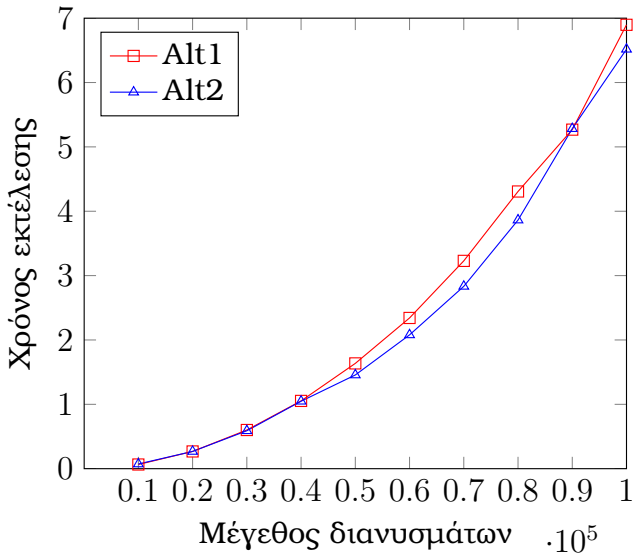
```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        size_t pi = partition(array, low, high);
        qsort(array, low, pi - 1);
        qsort(array, pi + 1, high);
    }
}

void qsort_wrapper(int array[], int low, int high)
{
    qsort(array, low, high);
}
```

Πίνακας 158: Quicksort: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 159: Quicksort: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt1	**Alt2**
10000	0.065	0.073
20000	0.268	0.268
30000	0.601	0.589
40000	1.053	1.048
50000	1.636	1.456
60000	2.342	2.079
70000	3.229	2.833
80000	4.307	3.863
90000	5.268	5.283
100000	6.897	6.517



Σχήμα 42: Quickort: Σύγκριση Alt1, Alt2

4.10.3 Παράλληλη εκτέλεση με tasks (1)

Συμβ. 133: Quicksort: parallel task

```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the left
    // and greater than pivot  on the right of pivot.
    for (int j = low; j <= high; j++) {
        if (array[j] < pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}


void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task
        {
            qsort(array, pi + 1, high);
        }
    }
}

void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, low, high);
    }
}
```

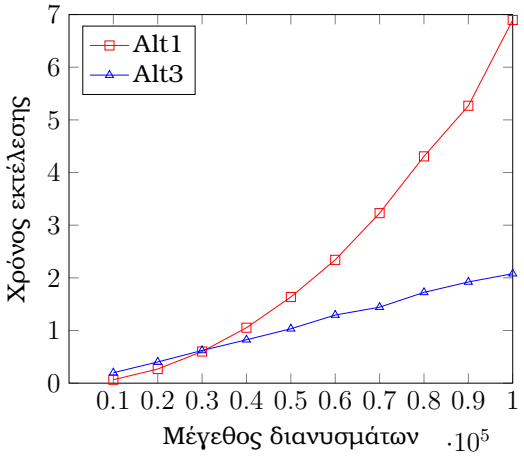
Πίνακας 160: Quicksort: Επιλογές μεταγλώττισης Alt3, Alt4

Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 161: Quicksort: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt3	Alt4
10000	0.198	0.210
20000	0.403	0.388
30000	0.621	0.615
40000	0.824	0.854
50000	1.033	1.059
60000	1.294	1.342
70000	1.443	1.487
80000	1.724	1.699
90000	1.922	1.871
100000	2.079	2.116

4.10.3.1 Παρατηρήσεις



Σχήμα 43: Quickort: Σύγκριση Alt1, Alt3

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
10000	-
20000	-
30000	-
40000	21
50000	36
60000	44
70000	55
80000	59
90000	63
100000	69

Πίνακας 162: Quicksort: Ποσοστιαία σύγκριση μεταξύ Alt1 και Alt3

4.10.4 Παράλληλη εκτέλεση με tasks (2)

Συμβ. 134: Quicksort: parallel for task

```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the left
    // and greater than pivot  on the right of pivot.
#pragma omp parallel for
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}


void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task

        {
            qsort(array, pi + 1, high);
        }
    }
}


void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, 0, high);
    }
}
```

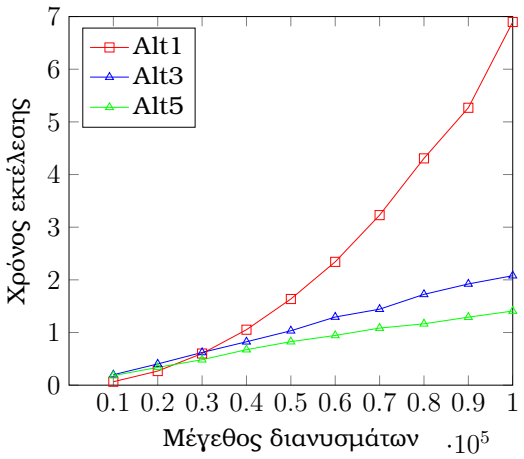
Πίνακας 163: Quicksort: Επιλογές μεταγλώττισης Alt5, Alt6

Label	Options
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt6

Πίνακας 164: Quicksort: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt5	Alt6
10000	0.181	0.179
20000	0.338	0.343
30000	0.483	0.486
40000	0.674	0.671
50000	0.826	0.804
60000	0.944	0.944
70000	1.084	1.066
80000	1.165	1.188
90000	1.291	1.282
100000	1.408	1.431

4.10.4.1 Παρατηρήσεις



Σχήμα 44: Quickort: Σύγκριση Alt1, Alt3, Alt5

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
10000	9
20000	16
30000	20
40000	18
50000	20
60000	27
70000	24
80000	32
90000	32
100000	32

Πίνακας 165: Quicksort: Ποσοστιαία σύγκριση μεταξύ Alt1 και Alt3

4.10.5 Παράλληλη εκτέλεση με tasks (3)

Συμβ. 135: Quicksort: parallel for task - taskwait

```
int partition(int array[], int low, int high)
{
    //select picot element
    int pivot = array[high];
    int i = (low - 1);

    // Put the elements smaller  than pivot on the left
    // and greater than pivot  on the right of pivot.
    for (int j = low; j <= high; j++) {
        if (array[j] < pivot) {
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[high]);
    return (i + 1);
}

void qsort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
#pragma omp task
        {
            qsort(array, low, pi - 1);
        }
#pragma omp task
        {
            qsort(array, pi + 1, high);
        }
#pragma omp taskwait
    }
}

void qsort_wrapper(int array[], int low, int high)
{
#pragma omp parallel
#pragma omp single
    {
        qsort(array, low, high);
    }
}
```

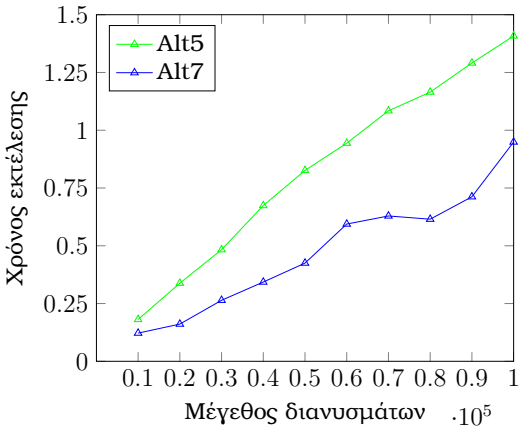
Πίνακας 166: Quicksort: Επιλογές μεταγλώττισης Alt7, Alt8

Label	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8

Πίνακας 167: Quicksort: Αποτελέσματα Alt7, Alt8

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt7	Alt8
10000	0.122	0.122
20000	0.161	0.158
30000	0.264	0.299
40000	0.343	0.294
50000	0.425	0.468
60000	0.594	0.531
70000	0.629	0.701
80000	0.615	0.584
90000	0.712	0.728
100000	0.949	0.999

4.10.5.1 Παρατηρήσεις



Σχήμα 45: Quickort: Σύγκριση Alt5, Alt7

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
10000	38
20000	60
30000	57
40000	58
50000	58
60000	54
70000	45
80000	64
90000	62
100000	54

Πίνακας 168: Quicksort: Ποσοστιαία σύγκριση Alt5 και Alt7

4.11 Παράδειγμα πολλαπλασιασμού μητρώων

Ο πολλαπλασιασμός μητρώων αποτελεί μια πράξη της γραμμικής άλγεβρας, που δημιουργήθηκε από το Γάλλο μαθηματικό Jacrues Philippe Marie Binet το 1812 για να περιγράψει τη σύνθεση δυο μητρώων. Αποτελεί ένα από τα βασικότερα εργαλεία της γραμμικής άλγεβρας και χρησιμοποιείται από ένα ευρύ φάσμα των μαθηματικών, της στατιστικής της φυσικής των οικονομικών και της μηχανικής.[38]

4.11.1 Περιγραφή κοινού τμήματος πολλαπλασιασμού μητρώων

Συμβ. 136: Matrix Multiplication: main()

```
int main(int argc, char **argv) {
    Opts o;
    parseArgs(argc, argv, o);

    srand(time(nullptr));
    int *A = new int [o.dim1 * o.dim2];
    int *B = new int [o.dim2 * o.dim3];

    fill_random_1d(A, o.dim1, o.dim2);
    fill_random_1d(B, o.dim2, o.dim3);

    int r3 = o.dim1;
    int c3 = o.dim3;

    int *C = new int [r3 * c3];

    //Count time
    double start = omp_get_wtime();
    matmul(A, o.dim1, o.dim2, B, o.dim2, o.dim3, C, r3, c3);
    double end = omp_get_wtime();

    // Calculating total time taken by the program.
    double time_passed = end - start;
    std::cout << "Multiplication Time : " << std::fixed
         << time_passed << std::setprecision(5);
    std::cout << " sec " << std::endl;

    // Verification
    if (o.verify) {
        verify_1d(A, B, C, o.dim1, o.dim2, o.dim3);
    }

    delete[] A;
    delete[] B;
    delete[] C;

    return 0;
}
```

4.11.2 Σειριακή εκτέλεση

Το πρόβλημα στη σειριακή του εκτέλεση περιλαμβάνει τρεις βρόγχους επανάληψης. Ο πρώτος βρόγχος διατρέχει της σειρές του μητρώου A ενώ ο δεύτερος διατρέχει τις στήλες του μητρώου B. Ο τρίτος βρόγχος αθροίζει τα στοιχεία της εκάστοτε γραμμής και στήλης των δυο αρχικών μητρώων και εισάγει το αποτέλεσμα στην τελική του θέση στο νέο μητρώο. Η υλοποίηση της περιγραφής δίνεται στο πίνακα που ακολουθεί:

Συμβ. 137: Matrix Multiplication: Σειριακή εκτέλεση

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

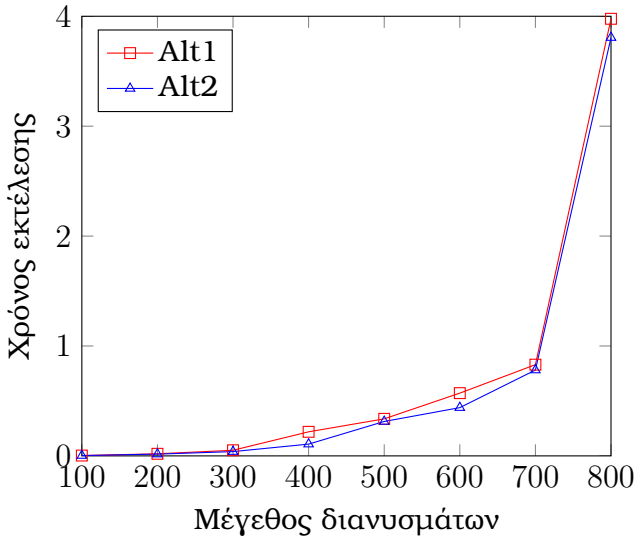
Πίνακας 169: Matrix Multiplication: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 170: Matrix Multiplication: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt1	Alt2
100x100x100	0.003	0.003
200x200x200	0.019	0.016
300x300x300	0.051	0.039
400x400x400	0.219	0.107
500x500x500	0.337	0.313
600x600x600	0.571	0.439
700x700x700	0.830	0.781
800x800x800	3.977	3.806

4.11.2.1 Παρατηρήσεις



Σχήμα 46: Matrix Multiplication: Σύγκριση Alt1, Alt2

4.11.3 Παραλλαγή parallel for (1)

Συμβ. 138: Matrix Multiplication: parallel for

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

#pragma omp parallel for
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

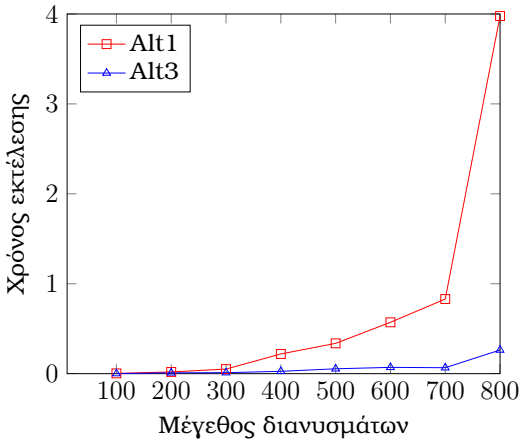
Πίνακας 171: Matrix Multiplication: Επιλογές μεταγλώττισης Alt3, Alt4

Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 172: Matrix Multiplication: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt3	**Alt4**
100x100x100	0.001	0.002
200x200x200	0.009	0.006
300x300x300	0.011	0.013
400x400x400	0.026	0.021
500x500x500	0.054	0.030
600x600x600	0.070	0.042
700x700x700	0.066	0.061
800x800x800	0.263	0.250

4.11.3.1 Παρατηρήσεις



Σχήμα 47: Matrix Multiplication: Σύγκριση Alt1, Alt3

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
100	66.6
200	52.6
300	78.4
400	88.12
500	83.9
600	87.7
700	92.05
800	93.39

Πίνακας 173: Matrix Multiplication: Ποσοστιαία σύγκριση μεταξύ Alt1 και Alt3

4.11.4 Παραλλαγή parallel for simd

Συμβ. 139: Matrix Multiplication: parallel for simd

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }
#pragma omp parallel for simd
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πίνακας 174: Matrix Multiplication: Επιλογές μεταγλώττισης Alt7, Alt8, Alt9

Label	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -fopenmp -o ./builds/Alt9

Πίνακας 175: Matrix Multiplication: Αποτελέσματα Alt7, Alt8, Alt9

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt7	**Alt8**	**Alt9**
100x100x100	0.005	0.006	0.007
200x200x200	0.009	0.008	0.008
300x300x300	0.013	0.008	0.016
400x400x400	0.030	0.022	0.025
500x500x500	0.043	0.048	0.043
600x600x600	0.044	0.045	0.057
700x700x700	0.065	0.070	0.065
800x800x800	0.260	0.286	0.280

4.11.4.1 Παρατηρήσεις

Από τον πίνακα που προηγήθηκε, παρατηρείται ότι η εισαγωγή της οδηγίας simd δεν επιφέρει διανυσματικοποίηση στον αλγόριθμο. Οι χρόνοι εκτέλεσης είναι οι ίδιοι. Αυτό επιβεβαιώνεται και από τα αρχεία που εξάγονται κατά τη μεταγλώττιση λόγο της οδηγίας -fopt-info-vec.

4.11.5 Παραλλαγή target (1)

Συμβ. 140: Matrix Multiplication: target parallel for collapse

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

#pragma omp target map(to: a[0:a_size], b[0: b_size])\
                        map(from: c[0:c_size])
#pragma omp parallel for collapse(2)
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πίνακας 176: Matrix Multiplication: Επιλογές μεταγλώττισης Alt10, Alt11

Label	Options
Alt10	-fopt-info-vec=builds/alt10.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt10
Alt11	-fopt-info-vec=builds/alt11.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt11

Πίνακας 177: Matrix Multiplication: Αποτελέσματα Alt10, Alt11

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt10	**Alt11**
100x100x100	0.864	0.994
200x200x200	1.172	1.138
300x300x300	1.744	1.743
400x400x400	2.838	2.844
500x500x500	4.706	4.693
600x600x600	7.444	7.478
700x700x700	11.484	11.447

4.11.5.1 Παρατηρήσεις

Η μεταφορά του προβλήματος στο περιβάλλον της κάρτας γραφικών και η εκτέλεση του σε αυτή με χρήση της οδηγίας parallel for collapse(2), είχε ως αποτέλεσμα την αύξηση του χρόνου εκτέλεσης του προβλήματος σε σχέση με τις υπόλοιπες παραλλαγές στη κεντρική μονάδα επεξεργασίας. Ακολουθούν προσπάθειες υλοποίησης με offloading και στόχο τη μείωση του χρόνου εκτέλεσης.

4.11.6 Παραλλαγή target (2)

Συμβ. 141: Matrix Multiplication: target parallel for collapse simd reduction

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

#pragma omp target map(to: a[0:a_size], b[0: b_size]) \
                                        map(from: c[0:c_size])
#pragma omp parallel for collapse(2)
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
#pragma omp simd reduction(+ : temp)
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πίνακας 178: Matrix Multiplication: Επιλογές μεταγλώττισης Alt12, Alt13, Alt14

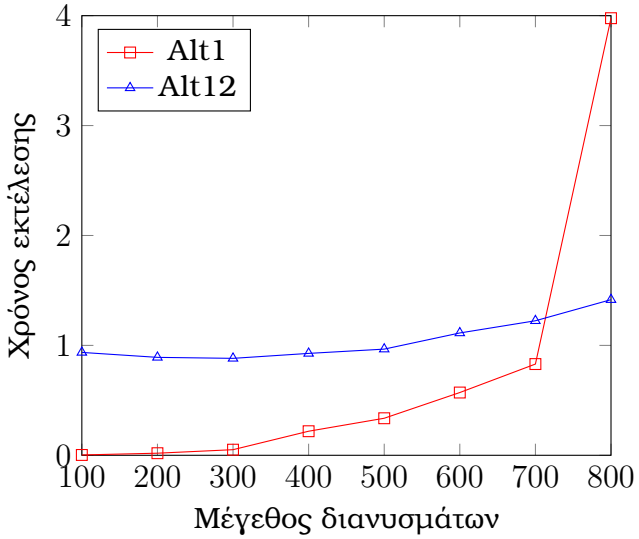
Label	Options
Alt12	-fopt-info-vec=builds/alt12.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt12
Alt13	-fopt-info-vec=builds/alt13.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt13
Alt14	-fopt-info-vec=builds/alt14.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt14

Πίνακας 179: Matrix Multiplication: Αποτελέσματα Alt12, Alt13, Alt14

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		
	Alt12	**Alt13**	**Alt14**
100x100x100	0.936	0.950	0.973
200x200x200	0.891	0.860	0.859
300x300x300	0.882	0.857	0.889
400x400x400	0.927	0.931	0.932
500x500x500	0.966	0.981	0.992
600x600x600	1.113	1.073	1.120
700x700x700	1.224	1.265	1.235
800x800x800	1.416	1.408	1.391

4.11.6.1 Παρατηρήσεις

Η υλοποίηση του προβλήματος είναι ίδια με αυτή της προηγούμενης παραγράφου με τη μόνη διαφορά να είναι η εισαγωγή της οδηγίας omp reduction. Η προσθήκη αυτή όχι απλώς ώθησε σε χρόνο καλύτερο από τη προηγούμενη παράγραφο, αλλά και σε χρόνους συγκρίσιμους με τη σειριακή εκτέλεση. Αξίζει να σημειωθεί ότι καθώς αυξάνεται το μέγεθος του προβλήματος, ο χρόνος εκτέλεσης για υλοποίηση σε GPU μεταβάλλεται με χαμηλό ρυθμό. Ως αποτέλεσμα, για μέγεθος προβλήματος μεγαλύτερο από 700x700x700, η εκτέλεση σε GPU έχει καλύτερες επιδόσεις, ενώ για μικρότερα όχι.



Σχήμα 48: Matrix Multiplication: Σύγκριση Alt1, Alt12

4.11.7 Παραλλαγή target (3)

Συμβ. 142: Matrix Multiplication: target teams parallel for collapse

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

#pragma omp target map(to: a[0:a_size], b[0: b_size])\
                        map(from: c[0:c_size])
#pragma omp teams
#pragma omp parallel for collapse(2)
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πίνακας 180: Matrix Multiplication: Επιλογές μεταγλώττισης Alt15, Alt16

Label	Options
Alt15	-fopt-info-vec=builds/alt15.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt15
Alt16	-fopt-info-vec=builds/alt16.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt16

Πίνακας 181: Matrix Multiplication: Αποτελέσματα Alt15, Alt16

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt15	**Alt16**
100x100x100	0.990	0.954
200x200x200	1.150	1.139
300x300x300	1.795	1.766
400x400x400	2.931	2.918
500x500x500	4.884	4.884
600x600x600	7.924	7.932
700x700x700	12.283	12.535

4.11.8 Παραλλαγή target (4)

Συμβ. 143: Matrix Multiplication: target teams distribute parallel for simd collapse

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
{
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

#pragma omp target map(to: a[0:a_size], b[0: b_size])\
        map(from: c[0:c_size])
#pragma omp teams
#pragma omp distribute parallel for simd collapse(2)
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
}
```

Πίνακας 182: Matrix Multiplication: Επιλογές μεταγλώττισης Alt17, Alt18

Label	Options
Alt17	-fopt-info-vec=builds/alt17.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt17
Alt18	-fopt-info-vec=builds/alt18.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt18

Πίνακας 183: Matrix Multiplication: Αποτελέσματα Alt17, Alt18

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		Ποσοστά εργασιών (%)		
	Alt17	Alt18	matmul	memcpyHtoD	memcpyDtoH
100x100x100	0.962	0.928	98.08	1.24	0.68
200x200x200	0.857	0.838	98.45	1.02	0.54
300x300x300	0.863	0.862	99.10	0.60	0.30
400x400x400	0.896	0.892	99.33	0.45	0.22
500x500x500	0.972	0.967	99.55	0.31	0.14
600x600x600	1.097	1.096	99.61	0.27	0.12
700x700x700	1.231	1.214	99.62	0.28	0.10
800x800x800	1.379	1.391	99.40	0.31	0.30

4.11.9 Παραλλαγή target (5)

Συμβ. 144: Matrix Multiplication: target teams distribute parallel for simd

```
void matmul(int *a, int r1, int c1,
        int *b, int r2, int c2,
        int *c, int r3, int c3)
 {
    if (c1 != r2) {
        std::cout << "Wrong matrix dimensions!" << std::endl;
        return;
    }

    int a_size = r1 * c1;
    int b_size = r2 * c2;
    int c_size = r3 * c3;

 #pragma omp target teams distribute parallel for simd\
            map(to:a[:a_size], b[:b_size])\
                map(from:c[:c_size])
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            int temp = 0;
            for (int k = 0; k < c1; ++k) {
                temp += a[k + i*c1] * b[j + k*c2];
            }

            c[j + i * c3] = temp;
        }
    }
 }
```

Πίνακας 184: Matrix Multiplication: Επιλογές μεταγλώττισης Alt19, Alt20, Alt21

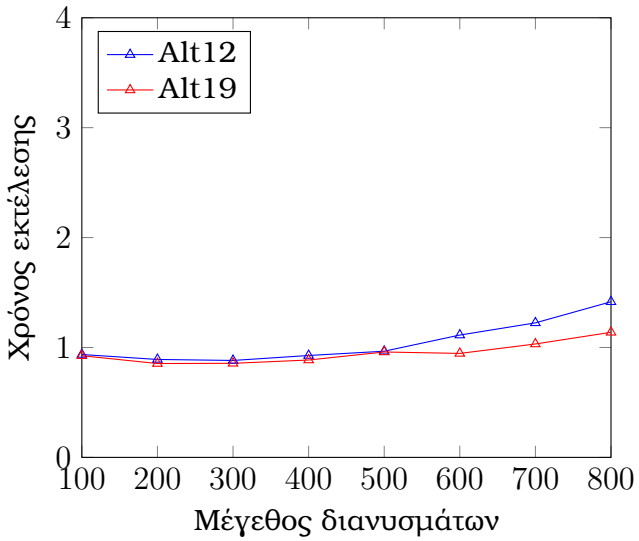
Label	Options
Alt19	-fopt-info-vec=builds/alt19.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt19
Alt20	-fopt-info-vec=builds/alt20.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt20
Alt21	-fopt-info-vec=builds/alt21.log -O2 -foffload=nvptx-none="-O2" -fno-stack-protector -fno-tree-vectorize -fopenmp -fno-inline -o ./builds/Alt21

Πίνακας 185: Matrix Multiplication: Αποτελέσματα Alt19, Alt20, Alt21

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)			Ποσοστά εργασιών (%)		
	Alt19	Alt20	Alt21	matmul	memcpyHtoD	memcpyDtoH
100x100x100	0.925	0.874	0.836	99.29	0.45	0.27
200x200x200	0.854	0.836	0.847	99.22	0.53	0.26
300x300x300	0.856	0.853	0.863	99.38	0.43	0.19
400x400x400	0.886	0.875	0.880	99.27	0.49	0.24
500x500x500	0.959	0.895	0.934	99.27	0.48	0.24
600x600x600	0.946	0.947	0.951	99.36	0.46	0.19
700x700x700	1.032	1.015	1.009	99.21	0.58	0.21
800x800x800	1.138	1.086	1.114	98.77	0.64	0.58

4.11.9.1 Παρατηρήσεις

Από τη μεταφορά των δεδομένων στη μονάδα επεξεργασίας της κάρτας γραφικών και την εκτέλεση του προέκυψαν τα αποτελέσματα του προηγούμενου πίνακα, τα οποία συγκρίνονται με τη σειριακή εκτέλεση αλλά και με την Alt12.



Σχήμα 49: Matrix Multiplication: Σύγκριση Alt9, Alt12

4.12 Παράδειγμα μετασχηματισμού Fourier

Ο μετασχηματισμός Fourier έχει πολλές εφαρμογές στη φυσική και στη μηχανική. Μετατρέπει μια μαθηματική συνάρτηση χρόνου σε μια συνάρτηση που η μονάδα μέτρησης της είναι συνήθως η συχνότητα. Στη μελέτη των σειρών Fourier, πολύπλοκες αλλά περιοδικές συναρτήσεις μπορεί να περιγραφούν ως ένα άθροισμα απλών κυμάτων.[36]

Στο παράδειγμα της παραγράφου υλοποιείται ο αλγόριθμος *DFT* που αποτελεί παραλλαγή του αλγορίθμου μετασχηματισμού Fourier έχοντας όμως ως κύριο χαρακτηριστικό την έλλειψη συνεχούς μέσου και τον μετασχηματισμό ακολουθίας πεπερασμένων τμημάτων σταθερού μεγέθους.

4.12.1 Περιγραφή κοινού τμήματος αλγορίθμου μετασχηματισμού Fourier

Συμβ. 145: DFT: main()

```
int main( int argc, char **argv ) {
    Opts o;
    parseArgs(argc, argv, o);

    double *real = new double[o.size];
    double *imag = new double[o.size];
    fill_arr(real, o.size);
    //fill_arr(imag, o.size);
    double *real_out = new double[o.size];
    double *imag_out = new double[o.size];
    double *ireal_out = nullptr;
    double *iimag_out = nullptr;

    double start = omp_get_wtime();
    dft(real, imag, real_out, imag_out, o.size, 0);
    double end = omp_get_wtime();

    if (o.verify) {
        ireal_out = new double[o.size];
        iimag_out = new double[o.size];
        compute_dft_real_pair(real_out, imag_out,
                ireal_out, iimag_out,
                o.size, 1);
        verify(real, imag, ireal_out, iimag_out, o.size);
    }
    std::cout << "Execution Time: " << std::fixed <<
                 std::setprecision(3) << end - start <<
                 " sec" << std::endl;

    delete []real;
    delete []imag;
    delete []real_out;
    delete []imag_out;
    if (ireal_out) delete []ireal_out;
    if (iimag_out) delete []iimag_out;

    return 0;
}
```

4.12.2 Σειριακή εκτέλεση

Το πρόβλημα στη σειριακή του μορφή αποτελείται από έναν διπλό βρόγχο επανάληψης, μεγέθους ίσου με το μέγεθος της συστοιχίας που δίνεται ως δεδομένο για τον μετασχηματισμό. Στο πρόβλημα εισάγονται δύο συστοιχίες ως δεδομένα, μια που αντιπροσωπεύει το πραγματικό μέρος των μιγαδικών αριθμών και μια που αντιπροσωπεύει το φανταστικό μέρος. Τα αποτελέσματα των υπολογισμών αποθηκεύονται σε δυο νέες συστοιχίες ίδιου μεγέθους.

Συμβ. 146: DFT: Σειριακή εκτέλεση

```
void dft(const double *inreal, const double *inimag,
        double *outreal, double *outimag, size_t n, int inverse) {

    for (size_t k = 0; k < n; k++) {  // For each output element
        double sumreal = 0;
        double sumimag = 0;
        for (size_t t = 0; t < n; t++) {  // For each input element
           double angle = 2 * M_PI * t * k / n;
           if (inverse) angle = -angle;
           sumreal +=  inreal[t] * cos(angle) +
                                   inimag[t] * sin(angle);
           sumimag += -inreal[t] * sin(angle) +
                                   inimag[t] * cos(angle);
        }
        if (inverse) {
            outreal[k] = sumreal/n;
            outimag[k] = sumimag/n;
        } else {
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }
}
```

Πίνακας 186: DFT: Επιλογές μεταγλώττισης Alt1, Alt2

Label	Options
Alt1	-fopt-info-vec=builds/alt1.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt1
Alt2	-fopt-info-vec=builds/alt2.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt2

Πίνακας 187: DFT: Αποτελέσματα Alt1, Alt2

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**	
	Alt1	**Alt2**
1000	0.142	0.142
2000	0.537	0.535
3000	1.185	1.184
4000	2.072	2.070
5000	3.164	3.230
6000	4.635	4.631
7000	6.234	6.220

4.12.3 Παραλλαγή με parallel for (1)

Στην υλοποίηση της παραγράφου, ο πρώτος βρόγχος επανάληψης εκτελείται παράλληλα με τη χρήση της οδηγίας pragma omp parallel for.

Συμβ. 147: DFT: omp parallel for

```
void dft(const double *inreal, const double *inimag,
        double *outreal, double *outimag, size_t n, int inverse) {

#pragma omp parallel for
    for (size_t k = 0; k < n; k++) {  // For each output element
        double sumreal = 0;
        double sumimag = 0;
        for (size_t t = 0; t < n; t++) {  // For each input element
            double angle = 2 * M_PI * t * k / n;
            if (inverse) angle = -angle;
            sumreal +=  inreal[t] * cos(angle) +
                        inimag[t] * sin(angle);
            sumimag += -inreal[t] * sin(angle) +
                        inimag[t] * cos(angle);
        }
        if (inverse) {
            outreal[k] = sumreal/n;
            outimag[k] = sumimag/n;
        } else {
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }
}
```

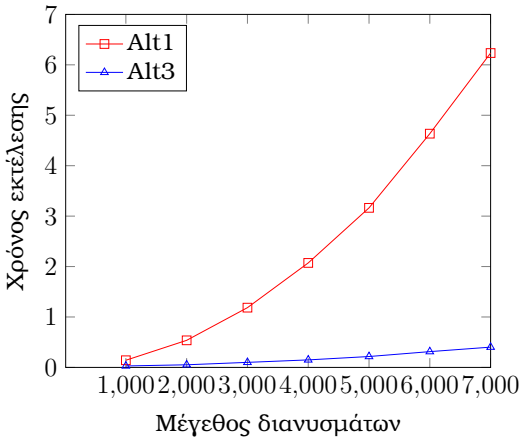
Πίνακας 188: DFT: Επιλογές μεταγλώττισης Alt3, Alt4

Label	Options
Alt3	-fopt-info-vec=builds/alt3.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt3
Alt4	-fopt-info-vec=builds/alt4.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt4

Πίνακας 189: DFT: Αποτελέσματα Alt3, Alt4

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt3	Alt4
1000	0.029	0.026
2000	0.052	0.051
3000	0.100	0.091
4000	0.149	0.152
5000	0.216	0.225
6000	0.313	0.307
7000	0.402	0.401

4.12.3.1 Παρατηρήσεις



Πίνακας 190: DFT: Σύγκριση Alt1 Alt3

Μέγεθος	Ποσοστό μείωσης χρόνου (%)
1000	80
2000	90
3000	92
4000	92
5000	92
6000	93
7000	93

Πίνακας 191: DFT: Ποσοστιαία σύγκριση Alt1 και Alt3

4.12.4 Παραλλαγή με parallel for (2)

Συμβ. 148: DFT: omp parallel for collapse critical

```
void dft(const double *inreal, const double *inimag,
        double *outreal, double *outimag, size_t n, int inverse) {

        double sumreal = 0.0;
        double sumimag = 0.0;
#pragma omp parallel for collapse(2)
    for (size_t k = 0; k < n; k++) {  // For each output element
        for (size_t t = 0; t < n; t++) {  // For each input element
            double angle = 2 * M_PI * t * k / n;
            if (inverse) {
                angle = -angle;
#pragma omp critical
                {
                    outreal[k] += (inreal[t] * cos(angle) +
                        inimag[t] * sin(angle))/n;
                    outimag[k] += (-inreal[t] * sin(angle) +
                        inimag[t] * cos(angle))/n;
                }
            } else {
#pragma omp critical
                {
                    outreal[k] +=  inreal[t] * cos(angle) +
                                   inimag[t] * sin(angle);
                    outimag[k] += -inreal[t] * sin(angle) +
                                   inimag[t] * cos(angle);
                }
            }
        }
    }
}
```

Πίνακας 192: DFT: Επιλογές μεταγλώττισης Alt5, Alt6

Label	Options
Alt5	-fopt-info-vec=builds/alt5.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt5
Alt6	-fopt-info-vec=builds/alt6.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt6

Πίνακας 193: DFT: Αποτελέσματα Alt5, Alt6

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)	
	Alt5	Alt6
1000	2.170	2.086
2000	9.141	9.711

4.12.4.1 Παρατηρήσεις

Η χρήση της οδηγίας critical όχι μόνο έδωσε στο πρόβλημα σειριακή μορφή, αλλά αύξησε κατα πολύ τους χρόνους εκτέλεσης λόγω των υποκείμενων κλειδωμάτων των μεταβλητών από τα εκάστοτε νήματα που τις μορφοποιούν.

4.12.5 Παραλλαγή με simd

Συμβ. 149: DFT: omp simd

```
void dft(const double *inreal, const double *inimag,
        double *outreal, double *outimag, size_t n, int inverse) {

#pragma omp simd
    for (size_t k = 0; k < n; k++) {  // For each output element
        double sumreal = 0;
        double sumimag = 0;
        for (size_t t = 0; t < n; t++) {  // For each input element
            double angle = 2 * M_PI * t * k / n;
            if (inverse) angle = -angle;
            sumreal +=  inreal[t] * cos(angle) +
                        inimag[t] * sin(angle);
            sumimag += -inreal[t] * sin(angle) +
                        inimag[t] * cos(angle);
        }
        if (inverse) {
            outreal[k] = sumreal/n;
            outimag[k] = sumimag/n;
        } else {
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }
}
```

Πίνακας 194: DFT: Επιλογές μεταγλώττισης Alt7, Alt8, Alt9

Label	Options
Alt7	-fopt-info-vec=builds/alt7.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt7
Alt8	-fopt-info-vec=builds/alt8.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt8
Alt9	-fopt-info-vec=builds/alt9.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt9

Πίνακας 195: DFT: Αποτελέσματα Alt7, Alt8, Alt9

Μέγεθος προβλήματος	Χρόνοι εκτέλεσης (sec)		
	Alt7	**Alt8**	**Alt9**
1000	0.142	0.140	0.144
2000	0.532	0.531	0.535
3000	1.185	1.175	1.183
4000	2.065	2.054	2.064
5000	3.162	3.138	3.162
6000	4.617	4.593	4.618
7000	6.207	6.198	6.202

4.12.6 Παραλλαγή με parallel for simd

Συμβ. 150: DFT: omp parallel for simd

```
void dft(const double *inreal, const double *inimag,
        double *outreal, double *outimag, size_t n, int inverse) {

#pragma omp parallel for simd
    for (size_t k = 0; k < n; k++) {  // For each output element
        double sumreal = 0;
        double sumimag = 0;
        for (size_t t = 0; t < n; t++) {  // For each input element
            double angle = 2 * M_PI * t * k / n;
            if (inverse) angle = -angle;
            sumreal +=  inreal[t] * cos(angle) +
                        inimag[t] * sin(angle);
            sumimag += -inreal[t] * sin(angle) +
                        inimag[t] * cos(angle);
        }
        if (inverse) {
            outreal[k] = sumreal/n;
            outimag[k] = sumimag/n;
        } else {
            outreal[k] = sumreal;
            outimag[k] = sumimag;
        }
    }
}
```

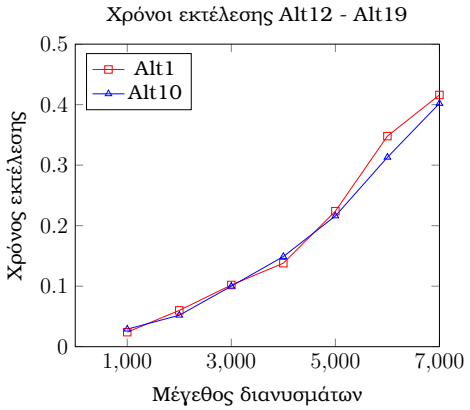
Πίνακας 196: DFT: Επιλογές μεταγλώττισης Alt10, Alt11, Alt12

Label	Options
Alt10	-fopt-info-vec=builds/alt10.log -O2 -fno-inline -fno-tree-vectorize -fopenmp -o ./builds/Alt10
Alt11	-fopt-info-vec=builds/alt11.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt11
Alt12	-fopt-info-vec=builds/alt12.log -O2 -fno-inline -ftree-vectorize -fopenmp -o ./builds/Alt12

Πίνακας 197: DFT: Αποτελέσματα Alt10, Alt11, Alt12

Μέγεθος προβλήματος	**Χρόνοι εκτέλεσης (sec)**		
	Alt10	**Alt11**	**Alt12**
1000	0.024	0.023	0.024
2000	0.060	0.063	0.044
3000	0.102	0.094	0.102
4000	0.138	0.143	0.149
5000	0.224	0.223	0.207
6000	0.348	0.324	0.301
7000	0.416	0.413	0.404

4.12.6.1 Παρατηρήσεις



Πίνακας 198: DFT: Σύγκριση Alt1 Alt10

5 Επίλογος

5.1 Σύνοψη και συμπεράσματα

Σε αυτή την εργασία μελετήθηκε το OpenMP, με την περιγραφη των βασικότερων χαρακτηριστικών, τη συγκέντρωση, την υλοποίηση και την καταγραφή παραδειγμάτων εκτελεσμένων σειριακά αλλά και παράλληλα. Είναι σημαντικό να σημειωθεί ότι η πληθώρα των παρατηρήσεων και συμπερασμάτων καταγράφονται σε κάθε επιμέρους παραλλαγή του προβλήματος. Ωστόσο κάποιες από τις γενικές παρατηρήσεις που προκύπτουν είναι οι παρακάτω:

- Η εισαγωγή των εργασιών (Tasks) αποτέλεσε ένα από τα σημαντικότερα νέα χαρακτηριστικά της διεπαφής του OpenMP, καθώς επιτρέπει την παραλληλοποίηση προβλημάτων για τα οποία δεν ήταν εφικτή ή ήταν δύσκολη ή ήταν εφικτή με μειονεκτήματα η υλοποίηση τους χωρίς αυτό το χαρακτηριστικό.
- Η παραλληλοποίηση πολλές φορές αποδεικνύεται λύση εύκολα υλοποιήσιμη -κυρίως όταν το πρόβλημα εμπεριέχει βρόγχους επανάληψης- και ικανή να μειώσει δραματικά το χρόνο εκτέλεσης σε σύγκριση με τη σειριακή.
- Η μεταφορά του κώδικα στο περιβάλλον της κάρτας γραφικών και η εκτέλεση του αλγορίθμου εκεί επιδέχεται βελτίωσης. Βρίσκεται σε πρώιμο στάδιο καθώς οι χρόνοι εκτέλεσης δεν είναι οι αναμενόμενοι σε σύγκριση με άλλες διεπαφές για εκτέλεση με Offloading.
- Βασικό μειονέκτημα στο offloading αποτελεί η οδηγία map και ο χρόνος που απαιτείται για την αντιστοίχιση των χρόνων εκτέλεσης στο περιβάλλον της κάρτας γραφικών.

5.2 Όρια και περιορισμοί της έρευνας

Ο παράλληλος προγραμματισμός αποτελεί έναν τομέα του ευρύτερου προγραμματισμού που αναπτύσσεται εδώ και χρόνια. Η μελέτη και η κατανόησή του είναι σχεδόν αδύνατο να περιοριστεί στα πλαίσια μιας διπλωματικής εργασίας μεταπτυχιακού επιπέδου, ακόμη και αν η αναφορά γίνεται αποκλειστικά στη διεπαφή του OpenMP. Για το λόγο αυτό, η εργασία περικλείεται από ορισμένα όρια, τα οποία αναφέρονται σε αυτή την παράγραφο.

- Η εργασία αποτελείται από ένα σύνολο παραδειγμάτων, για τη μεταγλώττισή των οποίων χρησιμοποιείται αποκλειστικά ο μεταγλωττιστής GCC/G++. Παρόλα αυτά, υπάρχουν αρκετές έρευνες που αναφέρουν ότι μια από τις βασικές αιτίες χαμηλής επίδοσης των προβλημάτων που χρησιμοποιούν offloading, είναι ο μεταγλωττιστής.[1].
- Σχετικά με την υλοποίηση των προβλημάτων, υπάρχουν πολλές υποψήφιες παραλλαγές. Η εργασία περιορίζεται στις σημαντικότερες από αυτές.
- Στην εργασία δεν χρησιμοποιήθηκε ευρέως η οδηγία is_device_ptr, καθώς για την χρήση της, απαιτείται δέσμευση μνήμης έξω από την παράλληλη περιοχή, στο κοινό τμήμα όλων των παραλλαγών. Κατά συνέπεια, κάθε πρόβλημα θα έπρεπε αποτελείται απο δύο main ρουτίνες και στη μεταγλώττιση να χρησιμοποιούνται εναλλάξ, κάτι που θα οδηγούσε στην μεγάλη αύξηση του μεγέθους της εργασίας. Παρόλα αυτά, οι ιδιότητες της οδηγίας is_device_ptr αναφέρονται στο παράδειγμα SAXPY.
- Στις περιπτώσεις που τα αποτελέσματα είναι εμφανώς ίδια με προηγούμενες παραλλαγές, δε δημιουργήθηκαν συγκριτικά διαγράμματα.
- Οι καταγεγραμμένοι χρόνοι της εργασίας αποτελούν τους μέσους όρους πολλαπλών εκτελέσεων του ίδιου προβλήματος, για να αποφευχθούν καταγραφές με σφάλμα.

5.3 Μελλοντικές Επεκτάσεις

Η εργασία μπορεί να εξελιχθεί μελλοντικά στους παρακάτων τομείς:

- Εμπλουτισμός οδηγιών που δεν έχουν συμπεριληφθεί στην εργασία.
- Επίλυση προβλημάτων με διαφορετικούς μεταγλωττιστές και σύγκριση μεταξύ τους.
- Εισαγωγή χαρακτηριστικών νεότερων εκδόσεων (ΟπενΜΠ 5.0)
- Εμπλουτισμός των ήδη υπαρχόντων παραδειγμάτων με επιπλέον παραλλαγές.
- Εξαγωγή νέων επιπλέον παρατηρήσεων.
- Δημιουργία και καταγραφή νέων παραδειγμάτων και αλγόριθμων.
- Αύξηση του μεγέθους των προβλημάτων με βέλτιστες επιδόσεις σε σχέση με τη σειριακή τους υλοποίηση

References

[1] Analysis of openmp 4.5 offloading in implementations: Correctness and overhead. https://www.researchgate.net/publication/335478417_Analysis_of_OpenMP_45_Offloading_in_Implementations_Correctness_and_Overhead.

[2] Effective vectorization with openmp 4.5. https://info.ornl.gov/sites/publications/files/Pub69214.pdf.

[3] Managing process affinity in linux. https://www.glennklockwood.com/hpc-howtos/process-affinity.html.

[4] Mergesort algorithm. https://www.geeksforgeeks.org/merge-sort/.

[5] Openmp tutorial. https://hpc.llnl.gov/openmp-tutorial#THREADPRIVATE.

[6] Producer-consumer. https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem.

[7] Runtime library routines. https://www.openmp.org/spec-html/5.0/openmpch3.html#x144-6340003.

[8] Task construct. https://www.openmp.org/spec-html/5.0/openmpsu46.html.

[9] Worksharing constructs. https://www.openmp.org/spec-html/5.0/openmpse16.html.

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

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

[12] O. API. Openmp api specification: Environmental variables. https://www.openmp.org/spec-html/5.0/openmpch6.html.

[13] 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.

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

[15] B. Barney. Openmp. https://computing.llnl.gov/tutorials/openMP/.

[16] I. Corporation. False sharing. https://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_c/main_cls/cref_cls/common/cilk_false_sharing.htm.

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

[18] V. Eijkhout. Openmp topic: Affinity. https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html#OpenMPthreadaffinitycontrol.

[19] N. Z. eScience Institute. Thread placement and thread affinity. https://support.nesi.org.nz/hc/en-gb/articles/360000995575-Thread-Placement-and-Thread-Affinity.

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

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

[22] IBM. pragma directives for parallel processing. https://www.ibm.com/support/knowledgecenter/SSGH2K_12.1.0/com.ibm.xlc121.aix.doc/compiler_ref/prag_omp_master.html.

[23] IBM. pragma directives for parallel processing. https://www.ibm.com/support/knowledgecenter/SSGH2K_12.1.0/com.ibm.xlc121.aix.doc/compiler_ref/prag_omp_flush.html.

[24] K.Margaritis. Introduction to openmp. pdplab.it.uom/teaching/llnl-gr/OpenMP.html#Abstract.

[25] J. R. Michael McCool, Arch D.Robison. *Structural Parallel Programming*, pages 124–125. Morgan Kaufmann, 2012.

[26] K. A. U. of Science and Technology. Thread affinity with openmp 4.0. https://www.hpc.kaust.edu.sa/tips/thread-affinity-openmp-40.

[27] Oracle. Task dependence. https://docs.oracle.com/cd/E37069_01/html/E37081/gozsa.html.

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

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

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

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

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

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

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

[35] J. Speh. Openmp for and scheduling. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html.

[36] Wikipedia. Dft. https://el.wikipedia.org/wiki/%CE%9C%CE%B5%CF%84%CE%B1%CF%83%CF%87%CE%B7%CE%BC%CE%B1%CF%84%CE%B9%CF%83%CE%BC%CF%8C%CF%82_%CE%A6%CE%BF%CF%85%CF%81%CE%B9%CE%AD.

[37] Wikipedia. Παράλληλος Προγραμματισμός. el.wikipedia.org/wiki/Parallel_computing.

[38] Wikipedia. Matrix multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication.

[39] Wikipedia. Mergesort. https://en.wikipedia.org/wiki/Merge_sort.