ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΗΣΗ OpenMP

Διπλωματική Εργασία

του

Κοντογιάννη Γεώργιου

Θεσσαλονίκη, Φεβρουάριος 2021

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΧΡΗΣΗ OpenMP

Κοντογιάννης Γεώργιος

Δίπλωμα Πολιτικού Μηχανικού, ΑΠΘ, 2016

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Μαργαρίτης Κωνσταντίνος

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

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

Κοντογιάννης Γεώργιος

.....

Η σύνταξη της παρούσας εργασίας έγινε στο LaTeX

Περίληψη

Αντικείμενο της διπλωματικής εργασίας είναι η μελέτη του OpenMP, ενός προτύπου παράλληλου προγραμματισμού, που δίνει στο χρήστη τη δυνατότητα αναπτύξης παράλληλων προγραμμάτων για συστήματα μοιραζόμενης μνήμης, τα οποία είναι ανεξάρτητα από τη αρχιτεκτονική του συστήματος και έχουν μεγάλη ικανότητα κλιμάκωσης[18].

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 θα γίνει χρήση της Διεπαφής Προγραμματισμού Εφαρμογών (Application Programming Interface - API) OpenMP, με χαρακτηριστικά που εισήχθησαν στις εκδόσεις OpenMP 3.0 που δημοσιεύθηκε το 2008 και OpenMP 4.5 που δημοσιεύθηκε 2015 καθώς και χαρακτηριστικά παλαιότερων εκδόσεων[23].

Τον Μάιο του 2008 κυκλοφόρησε η έκδοση του OpenMP 3.0. Στην κυκλοφορία συμπεριλήφθηκε για πρώτη φορά η έννοια των διεργασιών (Tasking) αλλά και βελτιώσεις στην υποστήριξη της διεπαφής μέσω της C++. Αποτελεί την πρώτη ενημέρωση μετά την έκδοση 2.5 με σημαντικές βελτιώσεις. 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[24].

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, τη μεταξύ τους σύγκριση 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 προγραμματισμου.

Για την ικανότητα παραλληλοποίησης του κώδικα, απαιτούνται αλγόριθμοι που αποτελούνται απο διεργασίες 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**,
- διεργασία **Producer-Consumer**,
- υπολογισμός π,
- υπολογισμός πρώτων αριθμών,
- υπολογισμός εσωτερικού γινομένου,
- πολλαπλασιασμός πινάκων,
- 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 εκτελεσή 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 τυχόν προέκυψαν.

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

Abstract

Keywords:

Ευχαριστίες

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

Περιεχόμενα

Κατάλογος Εικόνων (αν υπάρχουν)

Κατάλογος Πινάκων (αν υπάρχουν)

Λίστα Συμβολισμών

1 Εισαγωγή

1.1 Συνοπτικά για το OpenMP

Το OpenMP είναι μια Διεπαφή Προγραμματισμού Εφαρμογών (*API*) που χρησιμοποιείται για παραλληλοποίηση συστημάτων μοιραζόμενης μνήμης από λογισμικά γραμμένα σε γλώσσες **C/C++** και **Fortran**. Η διεπαφή αποτελείται από τα παρακάτω σύνολα[24]:

- σύνολο οδηγιών **(directives)** για τον μεταγλωττιστή που έχουν ως στόχο τον καθορισμό και τον έλεγχο της παραλληλοποίησης.
- σύνολο ενσωματωμένων ρουτίνων 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 απλή εντολή ή ένα σύνολο απλών εντολών[7]. Οι εντολές που βρίσκονται εντός της περιοχής παράλληλου κώδικα, εκτελούνται από όλα τα νηματα 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[29].

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 νέες οδηγίες και φράσεις (*clauses*) των εκδόσεων αυτών, ενώ γίνεται μια προσπάθεια υλοποίησης και συγκριτικής μελέτης απλών και σύνθετων προβλημάτων, επιλυόμενων 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

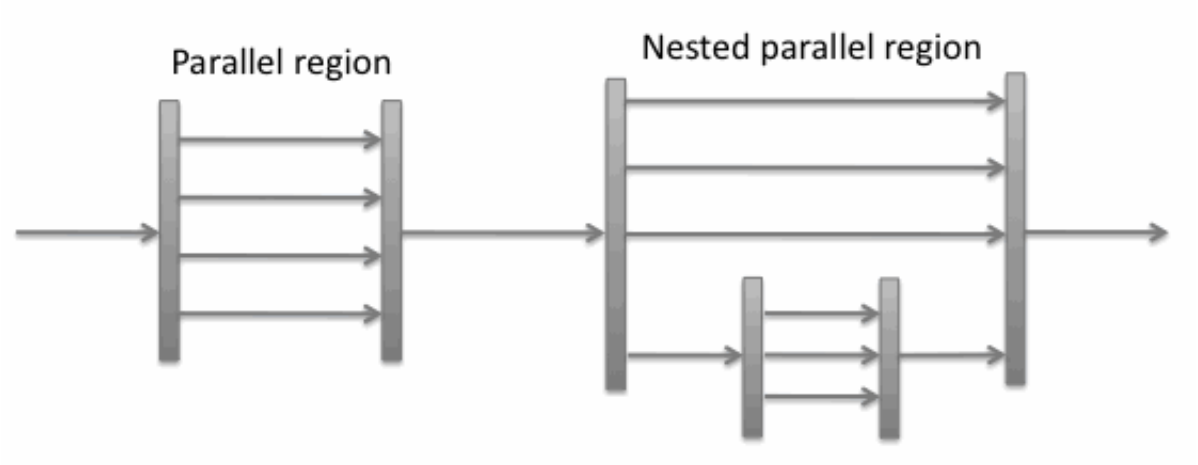
Στα επόμενα κεφάλαια γίνεται μια σύντομη περιγραφή του μοντέλου προγραμματισμού της διεπαφής *OpenMP* και της αλληλεπίδρασης των παραγόμενων νημάτων με το περιβάλλον δεδομένων. Στη συνέχεια, γίνεται σύντομη αναδρομή σε έννοιες απαραίτητες για την μελέτη των νέων χαρακτηριστικών που εισήχθησαν στις εκδόσεις μετά την 2.5. Η αναδρομή περιορίζεται κυριώς 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 διεργασιών-*Tasking*, του *offloading*, της διανυσματικοποίησης -*vectorization* κ.α. Αμέσως μετά, ακολουθεί η υλοποίηση, ο σχολιασμός και η συγκριτική μελέτη απλών και σύνθετων προβλήμάτων. 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[18].

Όταν το κύριο νήμα εισέρχεται στην περιοχή παράλληλου κώδικα (parallel region), 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)[18]. 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[9].

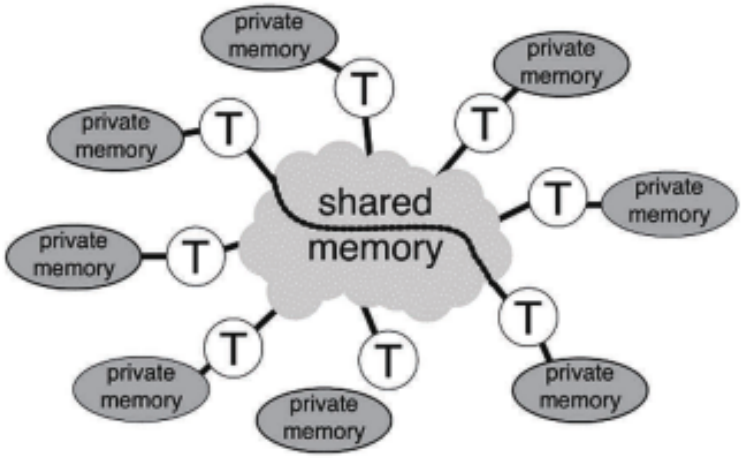


Σχήμα 1: Κύριο νήμα και ομάδες νημάτων

2.2 Αλληλεπίδραση νημάτων και περιβάλλοντος δεδομένων

Οπως 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*[21].

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[26].



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[27].

2.2.2 Κοινόχρηστη μνήμη

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

2.2.2.1 Race Condition

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.

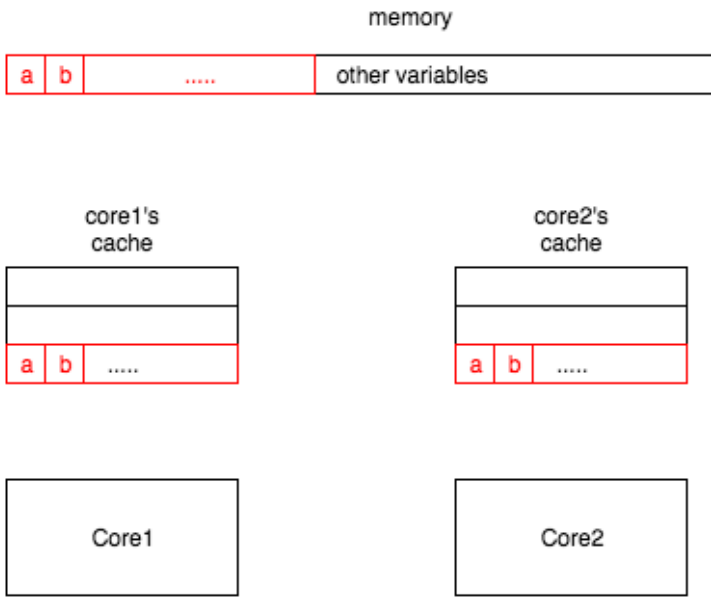
Συμβ. 3: Παράδειγμα race condition

```
#include <omp.h>

int main(void) {int sum = 0;

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

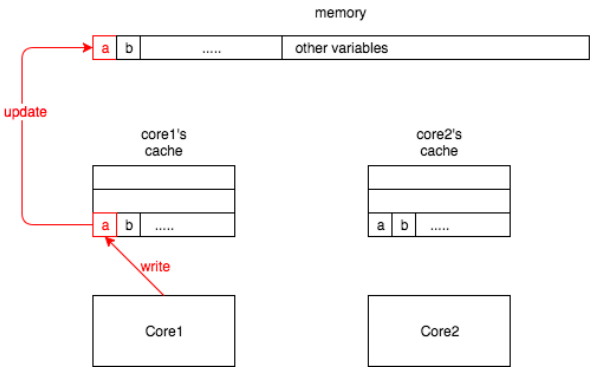
Η απλούστερη λύση, είναι η χρήση της κατάλληλης οδηγίας που επιβάλλει στο πρόγραμμα την προσπέλαση της μνήμης μόνο από ένα νήμα κάθε χρονική στιγμή. Τέτοιες οδηγίες είναι για παράδειγμα οι *pragma omp critical/atomic*. Εναλλακτικά, θα πρέπει αν ειναι εφικτό, να δημιουργείται ιδιωτικού αντίγραφου μεταβλητών 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 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*).



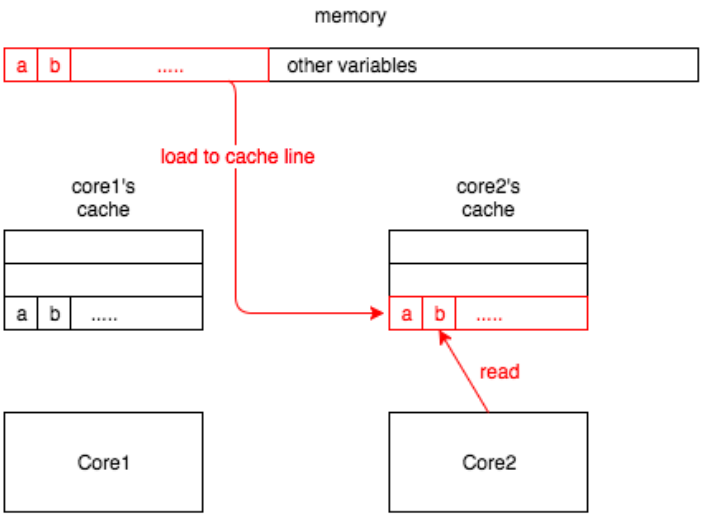
Σχήμα 3: False sharing (1/3)



Σχήμα 4: False sharing (2/3)

Οταν ένα νήμα τροποποιεί μια μεταβλητή, η γραμμή της μνήμης που βρίσκεται η μεταβλητή ακυρώνεται στους υπόλοιπους πυρήνες. Η γραμμή 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[10]. 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 Αναδρομή 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 συνοχής μνήμης

Για την αποφυγή φαινομένων *race condition* που οδηγούν σε λανθασμένα αποτελέσματα, απαιτείται συχνά ο συντονισμός πρόσβασης των νημάτων στις μεταβλητές της κοινόχρηστης μνήμης. Ο όρος *¨συγχρονισμός¨* αναφέρεται σε τέτοιους μηχανισμούς συντονισμού. Οι οδηγίες συγχρονισμού εγγυώνται την έγγυρη ανάγνωση της σωστής τιμής μιας μεταβλητής στην κοινόχρηστη μνήμη μετά από οποιαδήποτε ενημέρωσή της. Μηχανισμοί συγχρονισμού ειναι οι εξής[22]:

- #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 υποκείμενος φραγμός στο τέλος της οδηγίας. Ο φραγμός 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 που ολοκληρώνουν την περιοχή διαμοιρασμένης εργασίας, προχωρούν κατευθείαν στις επόμενες οδηγίες που ακολουθούν εντός της παράλληλης περιοχής[3].

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: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

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

Συμβ. 5: Αποδεκτές φράσεις οδηγίας for

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

2.3.2.2 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[18]:

Συμβ. 6: Σύνταξη οδηγίας sections

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

Συμβ. 7: Αποδεκτές φράσεις οδηγίας sections

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

2.3.2.3 Οδηγία *single*

Η οδηγία **single** εισάγεται μέσα στην περιοχή παράλληλου κώδικα, με μια ομάδα νημάτων ενεργή και καθορίζει ότι το εσωκλειόμενο τμήμα σε αυτή την 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*[3].

Συμβ. 8: Σύνταξη οδηγίας single

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

Συμβ. 9: Αποδεκτές φράσεις οδηγίας sections

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

2.3.2.4 Οδηγία *flush*

Η οδηγία **flush** διασφάλίζει οτι 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[16]:

- *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*

Συμβ. 10: Σύνταξη οδηγίας flush

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

2.3.2.5 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[17].

2.3.3 Φράσεις - Clauses

Δεδομένου οτι 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.

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.

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 κατα την είσοδο στη παράλληλη περιοχή που φέρει τη φράση **private** και η τελική του τιμή δεν διατηρείται για χρήση εκτός της παράλληλης περιοχής.

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, ωστε να καθιστάται σαφές αν θα είναι ιδιωτική ή κοινόχρηστη μέσω των φράσεων διαμοιρασμού μνήμης που προαναφέρθηκαν.

Φράση firstprivate

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

Φράση lastprivate

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

Φράση threadprivate

Η φράση *threadprivate* ορίζει ότι καθολικά αντικείμενα (ή μεταβλητές) μπορούν να γίνουν προσωρινά ιδιωτικά για κάποιο νήμα. Με αυτό τον τρόπο, μπορούν 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[18].

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[18].

Συμβ. 11: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

```
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.

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.

Φράση atomic

Η ενημέρωση μνήμης (ανάγνωση-τροποποίηση-εγγραφή) στην οδηγία που ακολουθεί εκτελείται ατομικά. Δεν καθιστά ολόκληρη την έκφραση *atomic* αλλα 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*.

Φράση ordered

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

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.

Φράση nowait

Χρησιμοποιείται για να ορίσει ότι τα νήματα που ολοκληρώνουν τη 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.

Φράση schedule

Χρησιμοποιείται στην οδηγία διαμοιρασμού εργασίας βρόγχου. Οι επαναλήψεις της οδηγίας ανατίθενται στα νήματα σύμφωνα με τον *τύπο* που ορίζεται μέσα στη φράση.

Συμβ. 12: Σύνταξη οδηγίας διαμοιρασμού εργασίας βρόγχου

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

• Οι τρεις τύποι *scheduling* είναι[28]:

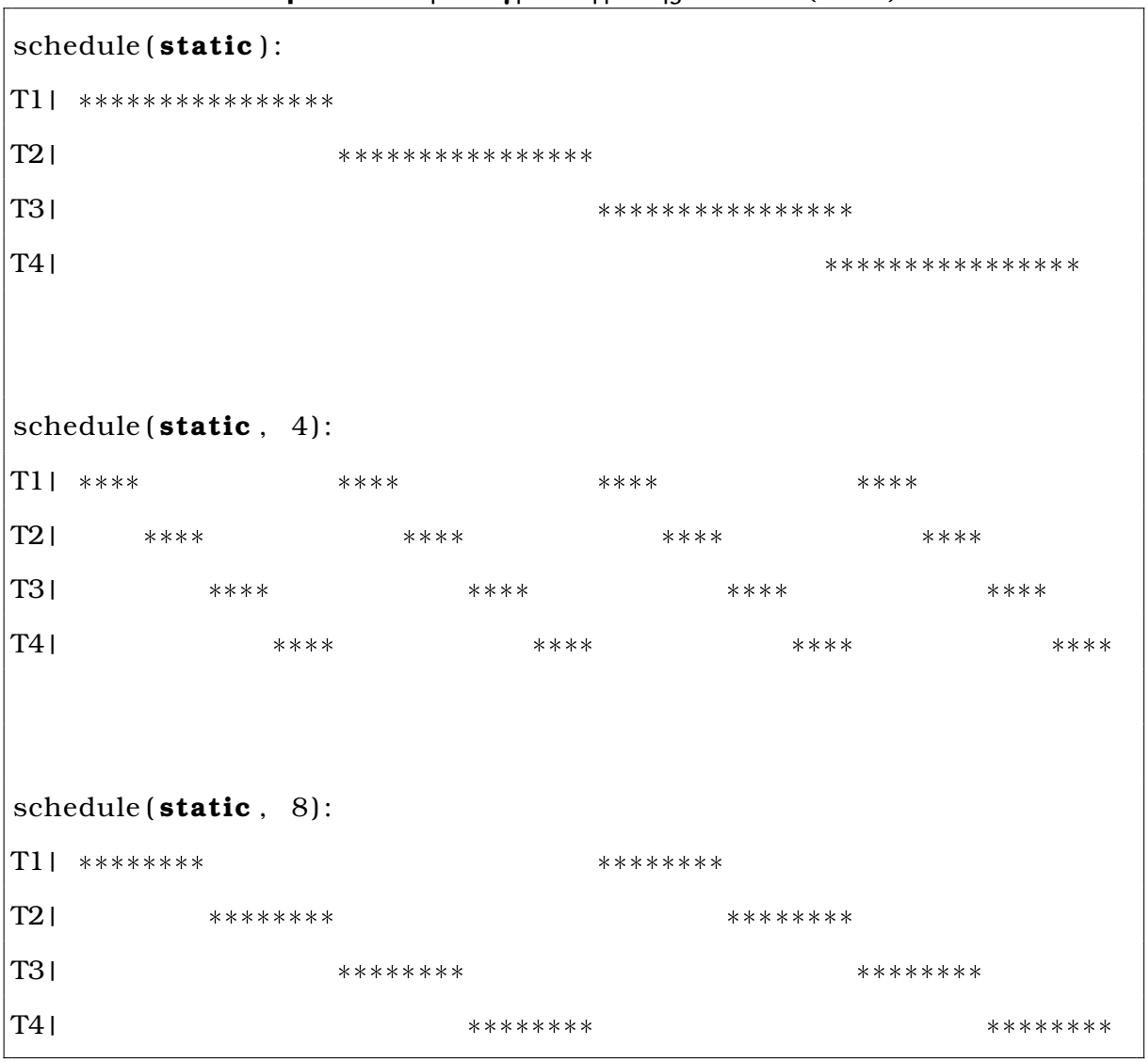
1. *static*:

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.

Οταν δεν ορίζεται το όρισμα *chunk*, ο αριθμός των επαναλήψεων που ορίζεται σε κάθε νήμα είναι ίσως με:

$$NumberOfIterations/NumberOfThreads$$

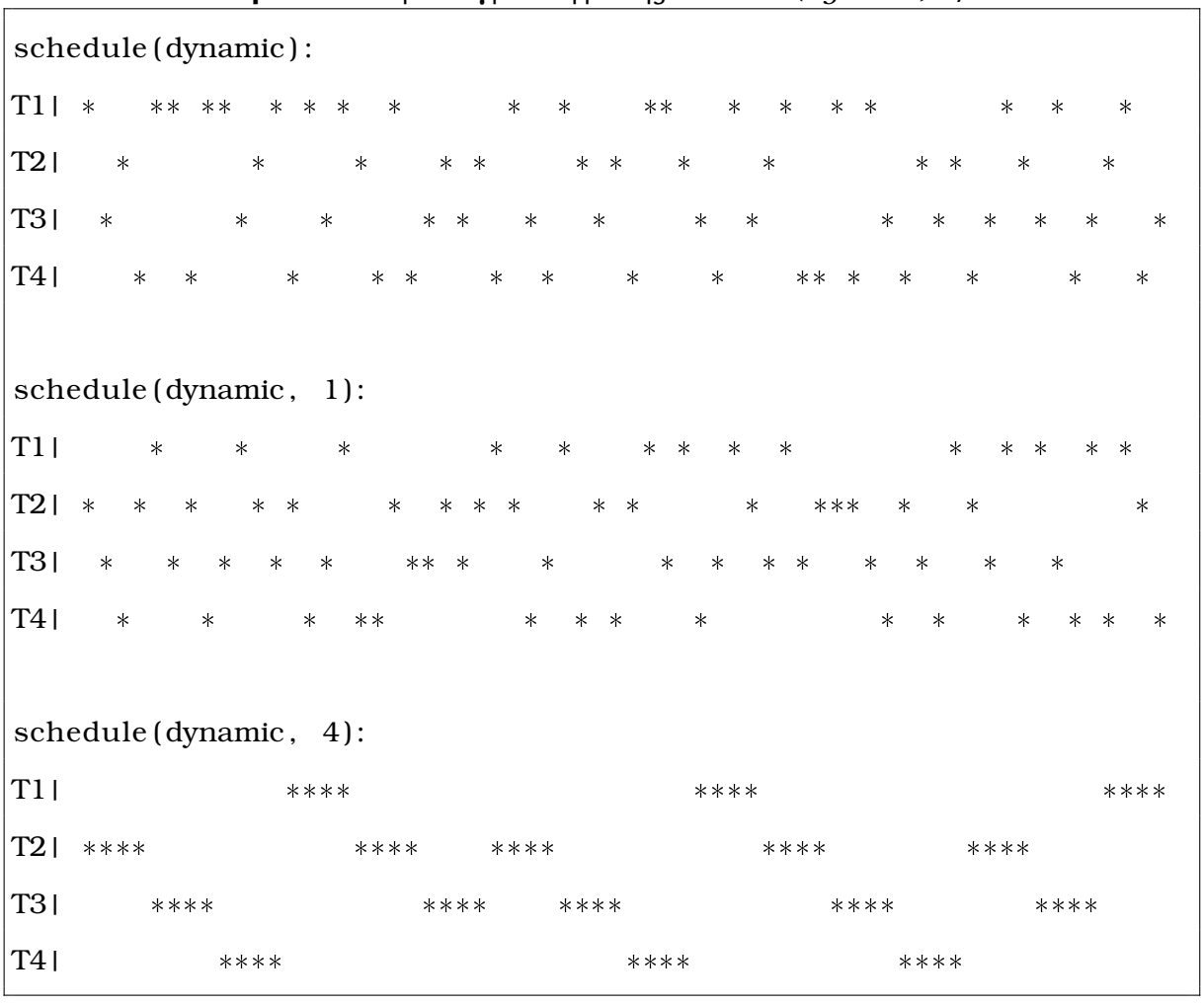
Συμβ. 13: Παραδείγματα φράσης schedule(static)



2. *dynamic*:

Ο τύπος αυτός χρησιμοποιείται όταν από το σύνολο των επαναλήψεων, ένα τμήμα κατανέμεται στα νήματα. Μόλις ένα συγκεκριμένο νήμα ολοκληρώσει την εκχωρημένη επανάληψή του, 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: Παραδείγματα φράσης *schedule(dynamic)* 1/2



Συμβ. 15: Παραδείγματα φράσης *schedule(dynamic)* 2/2

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

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.

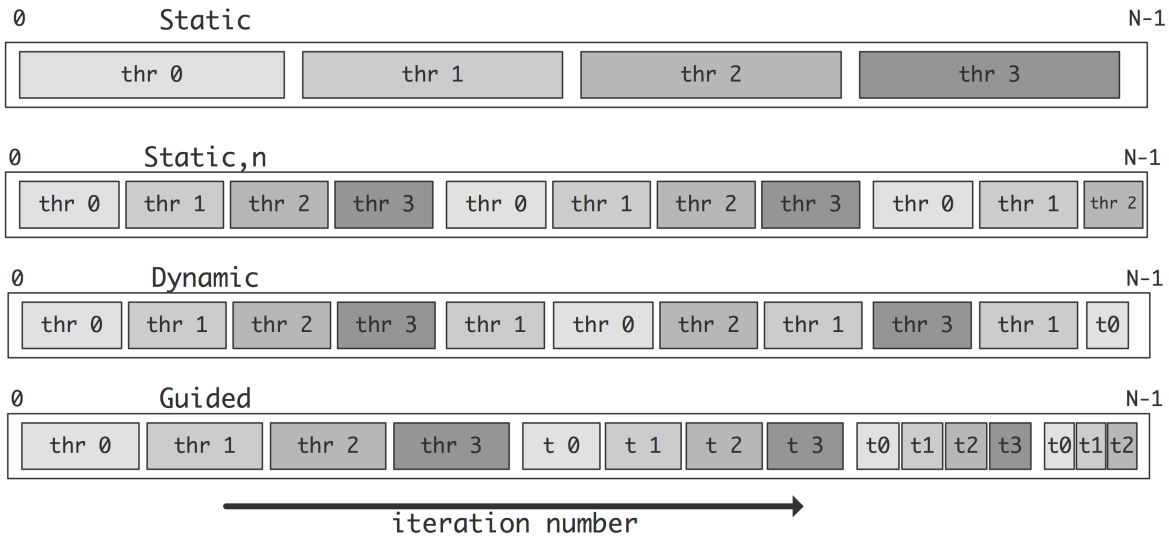
Η διαφορά με το δυναμικό τύπο είναι στο μέγεθος των ομάδων που διαχωρίζονται οι επαναλήψεις. Το μέγεθος είναι ανάλογο των μή εκχωρημένων επαναλήψεων διαιρούμενο με τον αριθμό των νημάτων. Επομένως το μέγεθος των ομάδων σταδιακά μειώνεται. Το ελάχιστο δυνατό μέγεθος των ομάδων ορίζεται από το όρισμα *chunk*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

Συμβ. 16: Παραδείγματα φράσης *schedule(guided)* 1/2

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


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

Συμβ. 17: Παραδείγματα φράσης *schedule(guided)* 2/2

```
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 απο το μεταγλωττιστή. Παρόλα αυτά, μερικές μπορούν να τροποποιηθούν κατα την εκτέλεση μέσω της χρήσης των αντίστοιχων οδηγίων της διεπαφής[6].

Πίνακας 1: Πίνακας μεταβλητών περιβάλλοντος.

OMP_SCHEDULE	*OMP_NUM_THREADS*
OMP_DYNAMIC	*OMP_PROC_BIND*
OMP_WAIT_POLICY	*OMP_NESTED*
OMP_THREAD_LIMIT	*OMP_MAX_ACTIVE_LEVELS*
OMP_CANCELLATION	*OMP_DISPLAY_ENV*
OMP_DISPLAY_AFFINITY	*OMP_AFFINITY_FORMAT*
OMP_DEFAULT_DEVICE	*OMP_MAX_TASK_PRIORITY*
OMP_TARGET_OFFLOAD	*OMP_TOOL*
OMP_TOOL_LIBRARIES	*OMP_DEBUG*
OMP_ALLOCATORS	

2.3.5 *Runtime Functions*

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

- 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 ρουτίνων

Ρουτίνες περιβάλλοντος εκτέλεσης	**Ρουτίνες κλειδώματος**
omp_set_num_threads	*omp_init_lock*
omp_get_num_threads	*omp_destroy_lock*
omp_get_max_threads	*omp_set_lock*
omp_get_thread_num	*omp_unset_lock*
omp_get_num_procs	*omp_test_lock*
omp_in_parallel	*omp_init_nest_lock*
omp_set_dynamic	*omp_destroy_nest_lock*
omp_get_dynamic	*omp_set_nest_lock*
omp_set_nested	*omp_unset_nest_lock*
Ρουτίνες καταμέτρησης χρόνου	
omp_get_wtime	*omp_get_wtick*

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 διεργασιών (**Tasking**). Η επόμενη έκδοση του *OpenMP* (4.0) κυκλοφόρησε τον Ιούλιο του 2013 και περιελάμβανε τα παρακάτω νέα χαρακτηριστικά:

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

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 ρουτίνων και δεδομένων, ικανών να μετακινηθούν σε μια συσκευή προορισμού (επιταχυντής) 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 στοιχείων διανύσματος 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 διεργασιών *(Tasking)*.

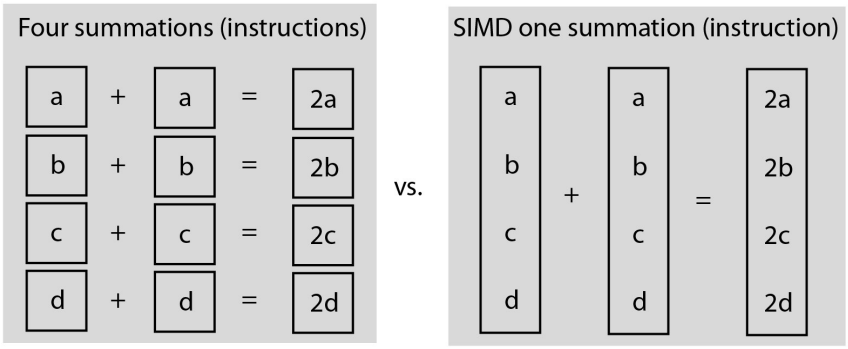
3.1 Διανυσματικοποίηση μέσω *SIMD*

Κατά την διάρκεια εκτέλεσης ενός τυπικού προγράμματος από έναν μή παράλληλο υπολογιστή, οι εντολές που εκτελούνται είναι απλές, εφαρμοζόμενες σε απλά, μοναδικά δεδομένα. Το μοντέλο αυτό ονομάζεται (*SISD - Single Instruction Single Data*) και αποτελούσε για πολλά χρόνια το επικρατέστερο μοντέλο υλοποίησης και εκτέλεσης προγραμμάτων. Αποτελεί ένα απο τα τέσσερα μοντέλα της ταξινόμησης *Flynn* που προτάθηκε το 1966[14]. 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 διάνυσμα.

Στο παρακάτω σχήμα παρουσιάζεται η πρόσθεση των στοιχείων δυο διανυσμάτων και η εκχώρηση των αποτελεσμάτων σε ένα τρίτο, με τον συμβατικό τρόπο και με την χρήση *SIMD* μεθόδου. Το πλεονέκτημα της δεύτερης μεθόδου είναι οτι 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 διανυσμάτων 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 χρήσης 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. 18: Παράδειγμα κώδικα με simd

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

Στο παράδειγμα, η μεταβλητή *i* είναι ιδιωτική. Η διαφορά με την ιδιωτική μεταβλητή ενός παράλληλου βρόγχου είναι οτι η ιδιωτικότητα αναφέρεται σε ένα *SIMD lane*. Οι τιμές των διανυσμάτων *t, k, m* είναι κοινόχρηστες. Το καταλληλότερο μήκος διανύσματος για την διανυσματικοποίησης, εξαρτάται από την αρχιτεκτονική του μηχανήματος και επιλέγεται από αυτό.

3.1.2 Φράσεις οδηγίας *simd*

Ένα σύνολο φράσεων ακολουθούμενων της οδηγίας *simd* υποστηρίζεται απο 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. 19: Φράσεις που υποστηρίζονται από την οδηγία *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 διανύσματος που χρησιμοποιείται από τις παραγόμενες *simd* οδηγίες.

Η τιμή του ορίσματος είναι προτιμητέα αλλα όχι υποχρεωτική. Ο μεταγλωττιστής έχει την ελευθερία να αποκλίνει από αυτή την επιλογή και να επιλέξει διαφορετικό μήκος. Ελλείψη αυτής της φράσης ορίζεται μια προεπιλεγμένη τιμή που καθορίζεται από 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 μήκους διανύσματος. Είναι το μήκος το οποίο είναι ασφαλές για τον βρόγχο. Το τελικό μήκος διανύσματος 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 δέσμευση στην προσπέλαση των στοιχείων του διανύσματος. Πιο συγκεκριμένα, το διάβασμα του *[i-10]* στην επανάληψη *i* δεν μπορεί να υλοποιηθει, αν δεν ολοκληρωθεί η εγγραφή στο *k[i]* της προηγούμενης επανάληψης.

Συμβ. 20: Παράδειγμα κώδικα με simd

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

3.1.2.3 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 διανυσμάτων εντός του βρόχου SIMD. Το πρώτο διάνυσμα κρατά τη γραμμική ακολουθία που δημιουργήθηκε προσθέτοντας την τιμή βήματος στην αρχική τιμή της μεταβλητής. Το δεύτερο διάνυσμα περιέχει το γραμμικό βήμα που αυξάνει το προηγούμενο διάνυσμα. Για παράδειγμα, εάν η φράση έχει μια γραμμική μεταβλητή N = 1 με ένα γραμμικό βήμα 2 και χρησιμοποιείται ένα διάνυσμα τεσσάρων θέσεων, τότε το πρώτο ιδιωτικό διάνυσμα θα περιείχε 1, 3, 5, 7. Μετά από μια άλλη επανάληψη σε βρόχο ΣΙΜΔ, το διάνυσμα 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. Εαν ένα στοιχείο διανύσματος δεν είναι ευθυγραμμισμένο σε διευθυνση 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 απο μια διεύθυνση μνήμης που δεν ειναι ευθυγραμμισμένη με το μέγεθος του δεδομένου που χρησιμοποιείται. Ετσι, οι λειτουργίες γίνονται κανονικά, αλλά με μεγαλύτερο κόστος. Σε περίπτωση διανυσματοκοποίησης 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 ενας 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*.

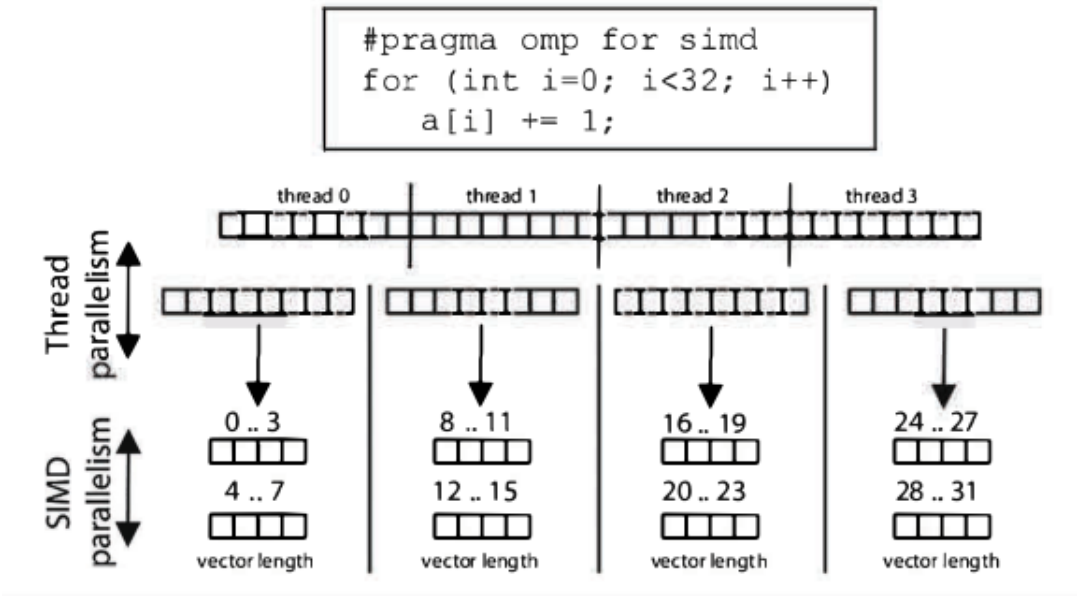
Συμβ. 21: Σύνθετη οδηγία βρόγχου με *simd*

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

Οι επαναλήψεις ενός βρόγχου διαιρούνται σε τμήματα και διανέμονται 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 διεργασιών 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 ανα νήμα θα μειωθεί. Προσθέτοντας παραλληλισμό *simd*, δεν ειναι 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 κλίσεις συναρτήσεων εντός της περιοχής βρόγχου *SIMD* εμποδίζουν τη δημιουργία αποτελεσματικών *SIMD* δομών. Στη χειρότερη περίπτωση η κλίση 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*. Ετσι, 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 οτι μια συνάρτηση μπορεί να χρησιμοποιηθεί από μια περιοχή παραλληλισμού *simd*.

Συμβ. 22: Οδηγία *declare simd*

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

Συμβ. 23: Φράσεις που υποστηρίζονται από *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 ωστόδο δύο περιορισμοί. Αν μια μεταβλητή αλλάζει ως αποτέλεσμα μιας τροποποίησης μιας φαινομενικά άσχετης μεταβλητής η συμπεριφορά είναι απροσδιόριστη. Επιπλέον, μια συνάρτηση που εμφανίζεται κάτω απο οδηγία *declare simd* δεν επιτρέπει τα *exceptions*.

3.1.5 Χαρακτηριστικά παραμέτρων SIMD συνάρτησης

Οι φράσεις ***uniform, linear, simdlen, aligned*** χρησιμοποιούνται για τον καθορισμό χαρακτηριστικών στις παραμέτρους μιας *simd* συνάρτησης. Εκτός από τη φράση *simdlen*, οι μεταβλητές που εισάγωνται 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 ίδια για όλες τις ταυτόχρονες κλίσεις κατά την εκτέλεση της οδηγίας *simd* βρόχου. Η φράση *uniform(arg1, arg2)* ενημερώνει τον μεταγλωττιστή να δημιουργήσει μια *simd* συνάρτηση που προϋποθέτει οτι αυτές οι δύο μεταβλητές είναι ανεξάρτητες από τον βρόγχο. Η φράση *linear* έχει διαφορετική σημασία όταν εμφανίζεται σε οδηγία *simd*. Δείχνει οτι ένα όρισμα που μεταβιβάζεται σε μια συνάρτηση έχει γραμμική σχέση μεταξύ των παράλληλων επικλήσεων μιας συνάρτησης. Κάθε *simd lane* παρατηρεί την τιμή του ορίσματος στην πρώτη λωρίδα και προσθέτει την μετατόπιση της *simd* λωρίδας από την πρώτη, επί το γραμμικό βήμα.

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

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: Παράδειγμα χρήσης φράσεων inbranch - notinbranch

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

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

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

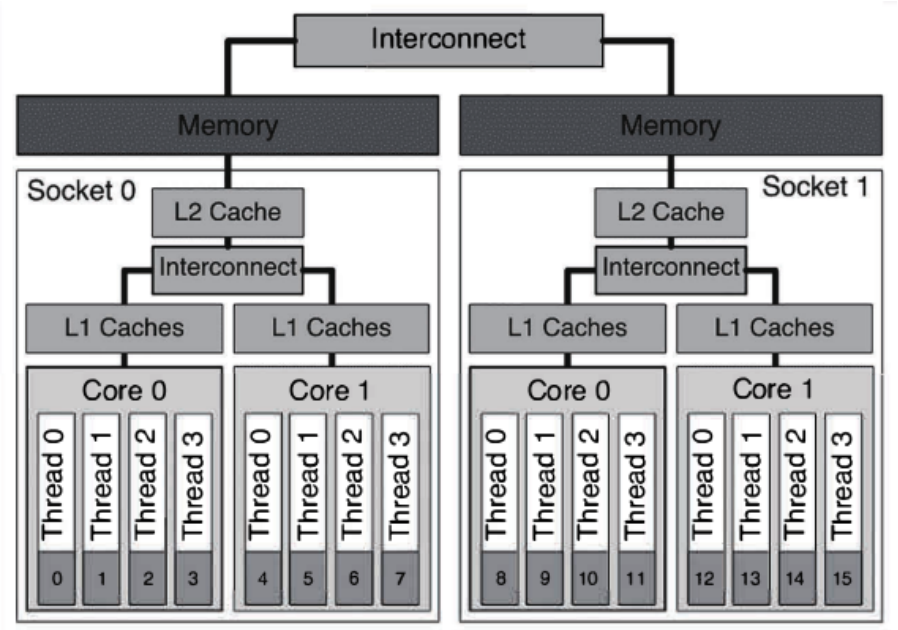
3.2 Thread Affinity

Thread Affinity είναι 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*[5]. Η πλειοψηφία πλέον των μηχανημάτων βασίζονται στην *cc-NUMA* αρχιτεκτονική. 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. Ενα τέτοιο σύστημα είναι ευκολότερο να επεκταθεί.

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



Σχήμα 9: Αρχιτεκτονική cc-NUMA[25]

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*[19].

3.2.1.1 *Thread Binding*

Οι προαναφερθήσες μεταβλητές, μπορούν να καθορίσουν σε ποια θέση στο υλικό θα ανατεθούν τα νήματα μια ομάδας που δημιουργήθηκε για να εκτελέσει μια διεργασία.
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 2* 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 ανατίθεται στους πόρους. Η επιλογή *OMP_PROC_BIND* = *close* σημαίνει ότι η ανάθεση περνά διαδοχικά στις διαθέσιμες θέσεις. Μια αλλη 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[12].

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[13]:

- 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 διεργασίες

3.3 Tasking

Η έννοια των διεργασιών(*Tasking*) εισήχθει στο *OpenMP* το 2008, με την έκδοση 3.0[11]. Οι διεργασίες παρέχουν τη δυνατότητα παραλληλοποίησης αλγορίθμων με ακανόνιστη και εξαρτώμενη από το χρόνο ροή εκτέλεσης μορφή. Ένας μηχανισμός ουράς διαχειρίζεται δυναμικά την εκχώρηση νημάτων στη διεργασία που πρέπει να εκτελεστεί. Τα νήματα παραλαμβάνουν διεργασίες από την ουρά εως ότου αυτή αδειάσει. Η διεργασία είναι ένα τμήμα κώδικα της παράλληλης περιοχής που εκτελειται ταυτόχρονα με μια άλλη διεργασία της ίδιας περιοχής.

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

Συμβ. 25: Παράδειγμα κώδικα με διεργασίες

```
#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 αρχίζει να εκτελεί το τμήμα κώδικα που βρίσκεται μέσα σε αυτή. Δύο διεργασίες δημιουργούνται και εισάγονται σχετική στοίβα 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 εως ότου εισαχθεί μια διεργασία στη στοίβα διεργασιών προς εκτέλεση. Τα νήματα αυτά είναι άμεσα διαθέσιμα για την εκτέλεση των διεργασιών. Το νήμα που δημιουργεί τις διεργασίες καταλήγει επίσης στο υποκείμενο φράγμα όταν ολοκληρωθεί η διαδικασία παραγωγής διεργασιών και είναι και αυτό διαθέσιμο για εκτέλεση των διεργασιών που απομένουν. Η σειρά εκτέλεσης των διεργασιών δεν είναι προκαθορισμένη.

3.3.1 Η οδηγία *task*

Βασική οδηγία της έννοιας των διεργασιών είναι η **task** που ορίζει μια ρητή διεργασία. Το περιβάλλον δεδομένων της διεργασίας δημιουργείται σύμφωνα με τις φράσεις χαρακτηριστικών κοινής χρήσης δεδομένων κατά την κατασκευή των διεργασίων και τυχόν προεπιλογές που ισχύουν για τις φράσεις αυτές[24].

Συμβ. 26: Σύνταξη οδηγίας *task*

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

Συμβ. 27: 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 δεχεται η συνθήκη **if** ως όρισμα σε μια διεργασία, αξιολογείται ως ψευδής ή αληθής. Αν η έκφραση αξιολογείται ψευδής, απαγορεύεται η αναβολή εκτέλεσής της από το μεταγλωττιστή. Έτσι, δεν γίνονται όλοι οι υπολογισμοί που απαιτούνται από τον μεταγλωττιστή για την κατασκευή της διεργασίας αυτής και εισαγωγής της στη στοίβα διεργασιών προς εκτέλεση, όπως αν δεν υπήρχε η φράση *if*. Με τον τρόπο αυτό, μπορούν να αποφευχθούν ορισμένοι υπολογισμοί που πιθανόν να οδηγήσουν σε μείωση της απόδοσης. Ανεξαρτήτου αποτελέσματος της έκφρασης ωστόσο, δημιουργείται πάντα ένα νέο περιβάλλον δεδομένων για τη διεργασία.

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 διεργασίας μειώνεται λόγω του γενικού κόστους κατασκευής της. Παρόλα αυτά, το κόστος της ρύθμισης περιβάλλοντος δεδομένων μπορεί να είναι το κυρίαρχο για τη δημιουργία διεργασίας, διότι για λόγους ασφαλείας οι μεταβλητές που αναφέρονται σε μια οδηγία κατασκευής διεργασιών είναι στις περισσότερες περιπτώσεις από προεπιλογή *firstprivate*[11].

3.3.1.2 Φράση final

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

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

- τις διεργασίες που επηρεάζει κάθε φράση
- τον τρόπο με τον οποίο διαχειρίζεται ο μεταγλωττιστής της κατασκευασμένες διεργασίες[4].

Πίνακας 3: Διαφορές ανάμεσα στις φράσεις *if* και *final* οταν εισάγονται σε κατασκευή διεργασίας.

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

3.3.1.3 Φράση mergeable

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

3.3.1.4 Φράση depend

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

Συμβ. 28: 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 διεργασιών 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[20].

Για τον τύπο **in(x)**, η προκύπτουσα διεργασία (y) θα είναι εξαρτημένη όλων των προηγούμενων συγγενικών διεργασιών που εχουν 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 διεργασία θα εκτελεστεί και ολοκληρωθεί πρίν από την εκτέλεση των διεργασιών τύπου *in ή inout* και περιέχουν την μεταβλητή **x** στη λίστα.

Παράδειγμα

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

Από τα παραπάνω, προκύπτει η εξής εξάρτηση διεργασιών:

Συμβ. 29: Παράδειγμα εξάρτησης διεργασιών

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

3.3.2 Συγχρονισμός διεργασιών

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

Στην περίπτωση της δημιουργίας διεργασιών μέσω οδηγίας ***single*** στο τέλος της οδηγίας υπάρχει υποκείμενος φραγμός εκτέλεσης, σε αντίθεση με την οδηγία ***master***.

Πίνακας 4: Οδηγιες συγχρονισμού διεργασιών.

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 διεργασίες-παιδιά της συγκεκριμένης διεργασίας.
#pragma omp taskgroup	Αναμένει μέχρι να ολοκληρωθούν όλες οι διεργασίες παιδιά της συγκεκριμένης διεργασίας αλλά και οι απόγονοι τους.

Συμβ. 30: Παράδειγμα 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 στον ίδιο δίαυλο, με κοινόχρηστη μνήμη και διεργασίες[15].

Οι μονάδες επεξεργασίας γραφικών βελτιώνουν σημαντικά την απόδοση των προγραμμάτων, με αποτέλεσμα τον πολλαπλασιάσμό της ζήτησης τους. 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 έκδοση του επιταχυντή γράφεται γλώσσα χαμηλότερου επιπέδου. Η συντήρηση και ανάπτυξη διπλού κώδικα, αποτελεί ένα από τα μεγαλύτερα προβλήματα στην έννοια του προγραμματισμού.

Για τις ανάγκες διευκόλυνσης, το *OpenMP* επέκτεινε τις λειτουργίες του με σκοπό την υποστήριξη και ευρεία χρήση τέτοιων τύπων συστημάτων[8]. 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 με γλώσσες προγραμματισμού υψηλότερου επιπέδου και εκτελέσιμων από συσκευές επεξεργασίας γραφικών. Ετσι 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 στον στη συσκευή στόχου (επιταχυντή).

Συμβ. 31: Παράδειγμα εκτέλεσης 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];
    }
}
```

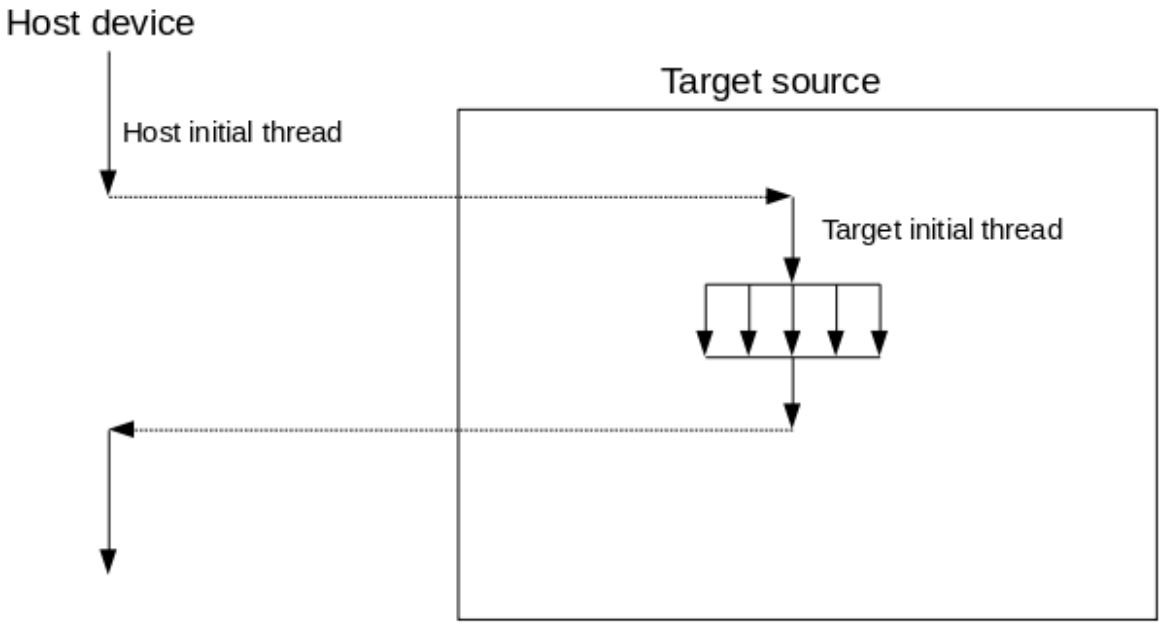
Οπως δείχνει το παραπάνω παράδειγμα, ένα νήμα του εξυπηρετητή (κύρια συσκευή - *host*) συναντάει την οδηγία ***target***. Το τμήμα κώδικα που ακολουθεί μεταφέρεται και εκτελείται στη συσκευή στόχου, αν αυτός υπάρχει. Απο προεπιλογή, το νήμα που συναντά την οδηγία περιμένει την ολοκλήρωση της εκτέλεσης 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 "αντιστοιχίζονται"* στον επιταχυντή. Η φράση *mapped* 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 προορισμού

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 προορισμού, 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*

Οπως τα νήματα των εξυπηρετητών, 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, ο εξυπηρετητής 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* απαλλάσει τον χρήστη από τον έλεγχο των χαρακτηριστικών της υλοποίησης σε επίπεδο υλικού, και η διεπαφή ενεργεί ανάλογα με την αρχιτεκτονική που χρησιμοποιείται.

Πίνακας 5: Ενέργειες που εκτελούνται από την οδηγία*map* ανάλογα με το είδος της αρχιτεκτονικής μνήμης

	memory allocation	**copy**	**flush**
Διαμοιρασμένη Μνήμη	Ναι	Ναι	Ναι
Κοινόχρηστη Μνήμη	Οχι	Οχι	Ναι

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 απο τον εξυπηρετητή, στο περιβάλλον δεδομένων του επιταχυντή και είναι προσβάσιμη από αυτόν.

Ανάλογα με την αρχιτεκτονική και τη διαθεσιμότητα κοινόχρηστης μνήμης μεταξύ του εξυπηρετητή *(host)* και της συσκευής προορισμού, η πρωτότυπη μεταβλητή που δημιουργήθηκε στον εξυπηρετητή και η αντίστοιχη της συσκευής προορισμού 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 συνεχείς αντιστοίχιση μεταβλητών που επαναχρησιμοποιούνται είναι αναποτελεσματική και οδηγεί σε πτώση της απόδοσης.

3.4.2.3 Δείκτες μεταβλητών συσκευής

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

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

Ο δείκτης συσκευής *(device pointer)* είναι ένας δείκτης που αποθηκεύεται στον εξυπηρετητή και περιέχει την διεύθυνση μνήμης στο περιβάλλον δεδομένων του επιταχυντή. Το *OpenMP* παρέχει ρουτίνες και οδηγίες που καθιστούν εφικτή τη δέσμευση μνήμης στο περιβάλλον του επιταχυντή μέσω του εξυπηρετητή, όπως φαίνεται στο παρακάτω παράδειγμα:

Συμβ. 32: Παράδειγμα taskwait

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

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 στον εξυπηρετητή αγνοώντας τις οδηγίες *#pragma*.

Συμβ. 33: Σύνταξη οδηγίας target

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

Συμβ. 34: 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 διεργασία που εκτελείται στον επιταχυντή. Η διεργασία για τον εξυπηρετητή ολοκληρώνεται όταν ολοκληρωθεί η εκτέλεση στον επιταχυντή. Οι φράσεις *nowait* και *depend* επηρεάζουν τον τύπο και την ασύγχρονη συμπεριφορά της διεργασίας. Από προεπιλογή, η διεργασία 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. 35: 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 από ένα νήμα, το αρχικό. Η λειτουργία αυτή είναι παρόμοια με μια οδηγία *parallel*. Η διαφορά σε αυτή την περίπτωση είναι οτι δημιουργούνται ομάδες που αρχικά απότελούνται απο ένα νήμα και στη συνέχεια τα νήματα της ομάδας πολλαπλασιάζονται. Νήματα διαφορετικών ομάδων δε συγχρονίζονται μεταξύ τους.

Όταν μια παράλληλη περιοχή συναντάται από μια ομάδα, τότε το αρχικό νήμα γίνεται κύριο σε μια νέα υποομάδα. Το αποτέλεσμα είναι ένα σύνολο υποομάδων, οπου κάθε υποομάδα αποτελείται από ένα ή περισσότερο νήματα. Αυτή η δομή χρησιμοποιείται για να εκφράζεται ένας τύπος χαλαρού παραλληλισμού, όπου ομάδες νημάτων εκτελούν παράλληλα, αλλά με μικρή αλληλεπίδραση μεταξύ τους.

Συμβ. 36: Φράσεις οδηγίας *target*

```
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. 37: Σύνταξη οδηγίας *distribute*

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

Συμβ. 38: 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 οτι τα κύρια νήματα των ομάδων δε συγχρονίζονται στο τέλος της οδηγίας.

Η φράση *distschedule* 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 ιδια 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. 39: Συνδυασμένες οδηγίες συσκευής στόχου

```
#pragma omp target parallel [clause[[,] clause]...]
        structured block

#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. 40: 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 Φράσεις οδηγίας *map*

Συμβ. 41: Σύνταξη οδηγίας map

```
map ([[map–type–modifier[,}} map–type:} list)
```

Συμβ. 42: Αποδεκτοί τύποι αντιστοίχισης

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

Η αντιστοίχιση μεταβλητών στον επιταχυντή, χωρίζεται σε τρεις φάσεις:

1. Η φάση *map-enter* στην αρχή της εκτέλεσης της οδηγίας *target*, όπου οι μεταβλητή αντιστοιχίζεται στον επιταχυντή. Σε αυτή, δεσμεύεται μνήμη του επιταχυντή για την αποθήκευση της μεταβλητής και αντιγράφεται από τον εξυπηρετητή.
2. Η φάση υπολογισμού, που προκύπτει όταν κατά τη διάρκεια εκτέλεσης της παράλληλης περιοχής, τα νήματα που εκτελούν το πρόγραμμα αποκτούν πρόσβαση στην αντιστοιχισμένη μεταβλητή.
3. Η φάση εξόδου όπου ολοκληρώνεται η αντιστοίχιση των μεταβλητών στον επιταχυντή. Η τιμή της μεταβλητής στον επιταχυντή αντιγράφεται στην αντίστοιχη θέση του εξυπηρετητή. 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 τον εξυπηρετητή. Ο καθορισμός του τύπου αντιστοίχισης επηρεάζει την απόδοση του κώδικα.

Πίνακας 6: Απαιτούμενη αντιγραφή για κάθε τύπο μεταβλητής κατά τις φάσεις εισόδου-εξόδου

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

Συμβ. 43: 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
- Οι τιμή της αντιγράφεται στον εξυπηρετητή
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στον εξυπηρετητή.

Η μεταβλητή **B**:

- Οι τιμή της αντιγράφεται στον επιταχυντή.
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στον εξυπηρετητή.

Η μεταβλητή **C**:

- Οι τιμή της αντιγράφεται στον επιταχυντή.
- Η μνήμη αποδεσμεύεται κατά την επιστροφή στον εξυπηρετητή.

3.4.8 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 *global* μεταβλητών στο περιβάλλον δεδομένων του επιταχυντή.

Συμβ. 44: Συνδυασμένες οδηγίας επιταχυντή

```
#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[1]

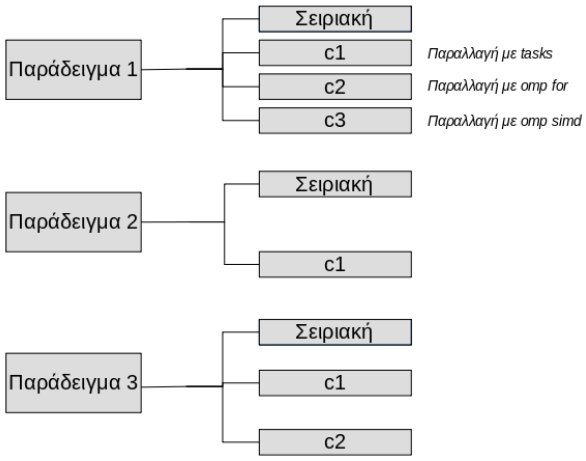
Οι ενότητες που ακολουθούν αφορούν τη μελέτη της θεωρίας που αναφέρθηκε στα προηγούμενα κεφάλαια, μέσω της υλοποίησης παραδειγμάτων. Τα προβλήματα που θα αναπτυχθούν, αντιπροσωπεύουν κατηγορίες βασικών προβλημάτων που συναντώνται συχνά σε ζητήματα παράλληλου προγραμματισμού. Για κάθε πρόβλημα υλοποιείται η σειριακή μέθοδος επίλυσης και παραλλαγές με παραλληλισμό που κυριώς 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 πηγιαίου 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 ανα 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, το compile όλων των διαφορετικών παραλλαγών είναι το ίδιο, με τη μόνη διαφορά να ειναι 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.



Σχήμα 11: Διάρθρωση παραδειγμάτων στο *github.com*

Για παράδειγμα, έστω οτι 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 εντολες για *compile* θα είναι οι εξής:

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

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

4.2 Αναφορά αρχιτεκτονικής μηχανήματος

Τα προβλήματα που ακολουθούν εκτελέστηκαν σε μηχάνημα λειτουργικό *linux* και μεταγλωττιστή *gcc-7.5.0*. Οι προδιαγραφές υλικού του μηχανήματος που εκτελέστηκαν τα προβλήματα, αναφέρονται στο παρακάτω παράδειγμα:

Πίνακας 7: 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

References

[1] . 0.

[2] Effective vectorization with openmp 4.5. https://info.ornl.gov/sites/publications/files/Pub69214.pdf. Accessed: 2017-03-20.

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

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

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

[6] O. API. Openmp api specification: Environmental variables. https://www.openmp.org/spec-html/5.0/openmpch6.html. Accessed: 2020-07-01.

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

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

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

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

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

[12] V. Eijkhout. Openmp topic: Affinity. https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html#OpenMPthreadaffinitycontrol. Accessed: 2020-06-20.

[13] N. Z. eScience Institute. Thread placement and thread affinity. https://support.nesi.org.nz/hc/en-gb/articles/360000995575-Thread-Placement-and-Thread-Affinity. Accessed: 2020-06-20.

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

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

[16] 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. Accessed: 2020-07-01.

[17] 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. Accessed: 2020-07-01.

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

[19] K. A. U. of Science and Technology. Thread affinity with openmp 4.0. https://www.hpc.kaust.edu.sa/tips/thread-affinity-openmp-40. Accessed: 2020-06-20.

[20] Oracle. Task dependence. https://docs.oracle.com/cd/E37069_01/html/E37081/gozsa.html. Accessed: 2020-06-20.

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

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

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

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

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

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

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

[28] J. Speh. Openmp for and scheduling. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html. Accessed: 2020-07-03.

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