ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής Και Υπολογιστών
Εργαστήριο Παράλληλων Συστημάτων

Επέκταση των τεχνικών δρομολόγησης του προγραμματιστικού μοντέλου OpenMP

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημήτρη Α.
Γαλανόπουλου

Επιβλέπων: Γεώργιος Γκούμας
Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής Και Υπολογιστών
Εργαστήριο Παράλληλων Συστημάτων

Επέκταση των τεχνικών δρομολόγησης του προγραμματιστικού μοντέλου OpenMP

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Δημήτρη Α. Γαλανόπουλου

Επιβλέπων: Γεώργιος Γκούμας
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Φεβρουαρίου 2020.

.....	.....	.....
Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2020

.....
Γαλανόπουλος Δημήτρης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στην εποχή της ανάλυσης 'Μεγάλων Δεδομένων' (Big Data) και των νευρωνικών δικτύων αναδεικνύεται όλο και πιο έντονα η ανάγκη ενίσχυσης της τοπικότητας των δεδομένων και της εύρεσης μεγαλύτερης παραλληλίας στα σύγχρονα προγράμματα. Παράλληλα, η συνεχόμενη αύξηση των επεξεργαστικών μονάδων των υπολογιστών και η ένταση των φαινομένων μη ομοιόμορφης προσπέλασης της μνήμης (NUMA) καθιστούν τον άμεσο χειρισμό των νημάτων (threads) από τους ίδιους τους προγραμματιστές μία αρκετά δύσκολη και χρονοβόρα εργασία.

Πάνω σε αυτήν τη βάση, προτείνεται το "hierarchical schedule" (ή ιεραρχική δρομολόγηση), μια μέθοδος αυτοματοποίησης της οργάνωση των νημάτων (threads) σε οριζόμενες από το χρήστη ομάδες (groups), αλλά και διαμοιρασμού της εργασίας σε αυτές. Είναι βασισμένο στην ευρέως διαδεδομένη πλατφόρμα ανάπτυξης παράλληλων προγραμμάτων OpenMP [1] και αποτελεί επέκταση της λειτουργικότητας της. Συγκεκριμένα, προσθέτει μία επιπλέον πολιτική δρομολόγησης (scheduling) δυναμικού διαμοιρασμού της εργασίας των παράλληλων for βρόχων του OpenMP (omp parallel for), η οποία λαμβάνει υπόψιν τα NUMA χαρακτηριστικά του μηχανήματος και διατηρεί την τοπικότητα του κώδικα, ενώ παράλληλα προσφέρει στο χρήστη περισσότερες δυνατότητες ελέγχου των threads. Η υλοποίηση έγινε επεκτείνοντας τον κώδικα του μεταγλωττιστή gcc (στην έκδοση 8.3.0). Στο τέλος μελετάται η συμπεριφορά του hierarchical schedule σε ένα σύνολο διαφόρων ειδών φόρτου εργασίας, και συγκρίνεται η επίδοσή του σε σχέση με τα άλλα schedules του OpenMP. Οι μετρήσεις έγιναν σε ένα intel μηχάνημα τεσσάρων Xeon E5-4620 επεξεργαστών, όπου κάθε ένας ανήκε σε έναν διαφορετικό NUMA κόμβο μνήμης των 64GB (ώστε συνολικά να έχουμε 256GB μνήμης).

Λέξεις Κλειδιά— κοινή μνήμη, OpenMP, δρομολόγηση, τοπικότητα, NUMA

Abstract

In the era of Big Data analysis and neural networks, the need for greater data locality and finer-grained parallelism becomes increasingly important. Due to the constant rise in the number of cores per processor and the intensity of NUMA effects, manual handling of threads has become a lot more demanding and time consuming.

Based on these facts, we propose "hierarchical schedule", a method of automating the organization of threads into groups and the act of work sharing between them. This work is based on the OpenMP library [1], a widely used platform for shared memory parallel programming. More specifically, it adds a new dynamic scheduling policy for the "parallel for" constructs, which takes into account the NUMA characteristics of the machine and retains the locality of the code as much as possible, while giving the user finer-grained control over the threads. The actual implementation was done by extending the gcc compiler (version 8.3.0). We conclude by studying the behavior of hierarchical schedule over a variety of work loads, and compare it's performance to that of the other OpenMP schedules. All our measurements were taken on an intel machine with four Xeon E5-4620 cpus, each one belonging to a different 64GB NUMA memory node (for a total of 256GB of memory).

Keywords— shared memory, OpenMP, scheduling, locality, NUMA

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η προσπάθεια για βελτίωση της επίδοσης των υπολογιστικών συστημάτων είναι αδιάκοπη. Μετά το τέλος του νόμου του Moore, η έρευνα στράφηκε προς δύο κυρίως κατευθύνσεις, την εκμετάλλευση της παραλληλίας, όπου αυτή υπάρχει στα προγράμματα, και τη βελτίωση της συμπεριφοράς της cache, μέσω τεχνικών αύξησης της τοπικότητας των δεδομένων.

1.1 Σύγχρονα συστήματα κοινής μνήμης

Ο υπολογιστικός κόσμος έχει μεταβεί πλήρως στην εποχή της παράλληλης επεξεργασίας, είτε σε επίπεδο hardware είτε σε επίπεδο software. Οι επεξεργαστές περιέχουν ολοένα και περισσότερες επεξεργαστικές μονάδες, που φτάνουν διψήφιο αριθμό σε πλήθος. Η υπολογιστική ισχύς των μηχανημάτων κοινής μνήμης έχει αυξηθεί δραματικά, και τα μεγέθη δεδομένων που μπορούν πλέον να επεξεργάζονται είναι διόλου ευκαταφρόνητα. Το κόστος τους αντίθετα συνεχώς μειώνεται, ώστε ένας απλός καταναλωτής με ένα εύλογο κεφάλαιο να έχει πλέον πρόσβαση σε μηχανήματα κοινής μνήμης με πάνω από εκατό πυρήνες και εκατοντάδες GByte μνήμης. Παράλληλα, η υλοποίηση προγραμμάτων είναι πολύ πιο εύκολη από ότι σε κατανεμημένα συστήματα, στα οποία ο χρήστης πρέπει να μεριμνήσει ο ίδιος για τη διαχείριση των δεδομένων και τη μεταφορά τους μεταξύ των απομακρυσμένων κόμβων μνήμης. Όλα τα προηγούμενα έχουν καταστήσει τα μηχανήματα κοινής μνήμης αρκετά δημοφιλή στο χώρο των υπολογιστικών συστημάτων υψηλής επίδοσης (High Performance Computing - HPC), και έχουν συμβάλει στη διεύρυνση της χρήσης τέτοιων συστημάτων, οπότε αντίστοιχα και της μελέτης της συμπεριφοράς τους.

Η μεγάλη όμως αυτή συγκέντρωση πυρήνων οδηγεί σε αντίστοιχη αύξηση της κατανάλωσης ενέργειας και της έκλυσης θερμότητας του επεξεργαστή, το οποίο αντισταθμίζεται κυρίως μόνο με μείωση της συχνότητας. Επίσης, το μέγεθος της επιφάνεια των chip συνδέεται με αντιστρόφως ανάλογο τρόπο με το ποσοστό απόδο-

σης της παραγωγής. Αυτό έχει ωθήσει σε αρχιτεκτονικές πολλαπλών επεξεργαστών - sockets, και πρακτικές διαμοιρασμού της μνήμης σε αυτούς κατά αντιστοιχία με τα κατανεμημένα συστήματα, διατηρώντας όμως μία χαρτογράφηση (mapping) της μνήμης για όλους, και με πολύ μικρότερο κόστος προσπέλασης διαφορετικού κόμβου μνήμης. Το φαινόμενο αυτό της μη ομοιόμορφης προσπέλασης της μνήμης (NUMA) συναντάται όλο και πιο συχνά με την αύξηση των πυρήνων, ώστε να έχει αρχίσει να εμφανίζεται ακόμα και σε μηχανήματα που προορίζονται για απλούς καταναλωτές, ενώ αντίστοιχα στα μηχανήματα μεγαλύτερης κλίμακας η επίδρασή του στην επίδοση γίνεται ολοένα και πιο αισθητή. Αυτό φαίνεται και από την άνθηση της έρευνας γύρω από το συγκεκριμένο πεδίο.

Από την άλλη, τα προγράμματα συνεχώς εξελίσσονται ώστε να εκμεταλλεύονται το μεγάλο πλήθος πυρήνων που είναι διαθέσιμο. Προφανώς, σε κάθε τέτοιο πρόγραμμα τίθεται το ζήτημα της δημιουργίας και διαχείρισης των παράλληλων κομματιών κώδικα προς επεξεργασία, αλλά και των threads, τα οποία τελικά θα τα εκτελέσουν. Για τη διευκόλυνση των προγραμματιστών έχει υλοποιηθεί πλήθος βιβλιοθηκών, εκ των οποίων μάλλον η πιο γνωστή και δημοφιλής είναι το OpenMP. Το OpenMP είναι μία βιβλιοθήκη αυτόματης δημιουργίας και διαχείρισης threads. Επιπλέον, προσφέρει στον προγραμματιστή πιο υψηλού επιπέδου δομές περιγραφής τη παραλληλίας του κώδικα, και διευκολύνει την ανάθεση της εργασίας σε αυτά. Η κύρια και πιο συχνή τέτοια δομή θα πρέπει να είναι ο βρόχος παράλληλου for, ο οποίος στην πιο απλή μορφή του εκφράζεται σε C κώδικα ως εξής:

```
#pragma omp parallel for
for (...)
{
    // user code
}
```

Στη δομή αυτή οι επαναλήψεις του βρόχου (loop) μοιράζονται αυτόματα στα threads από τη βιβλιοθήκη, κατά την εκτέλεση του προγράμματος. Όπως βλέπουμε, το μόνο που ουσιαστικά χρειάστηκε να προσθέσει ο προγραμματιστής ήταν η πρώτη γραμμή κώδικα. Ο διαμοιρασμός γίνεται σύμφωνα με ένα από τα schedules που προσφέρει το OpenMP, η επιλογή του οποίου γίνεται από το χρήστη. Επιπλέον, στο OpenMP προσφέρεται ο μηχανισμός των tasks, κατά τον οποίο ο χρήστης ορίζει, και παράγει κατά την εκτέλεση του προγράμματος, κομμάτια κώδικα που μπορούν να εκτελεστούν παράλληλα, και έπειτα η βιβλιοθήκη τα μοιράζει στα threads προς εκτέλεση. Με τον μηχανισμό αυτό είναι δυνατή η έκφραση πιο περίπλοκων σχημάτων παραλληλίας, όπως για παράδειγμα loop στα οποία ο αριθμός των επαναλήψεων δεν είναι γνωστός κατά το χρόνο του compile.

1.2 Ώθηση για την παρούσα εργασία

Ώθηση για την παρούσα δουλειά αποτέλεσαν μια σειρά από παρατηρήσεις προηγούμενων ερευνών πάνω στο parallel computing, και πιο συγκεκριμένα στην παράλληλη επεξεργασία γράφων. Η εξάπλωση της χρήσης κοινωνικών δικτύων, αλλά και η χρήση των αλγορίθμων γράφων σε μεγάλο φάσμα άλλων εφαρμογών, έχουν οδηγήσει σε αξιοσημείωτη άνθηση του πεδίου αυτού. Κατ΄ αναλογία, η ανάπτυξη των αλγορίθμων αυτών έχει συγκεντρώσει το ενδιαφέρον των ερευνητών και έχουν γίνει σημαντικές προσπάθειες για τη βελτιστοποίησή τους. Η μελέτη των αλγορίθμων γράφων έχει επίσης ιδιαίτερο ενδιαφέρον, διότι εμπεριέχουν την επεξεργασία πολύ μεγάλων μεγεθών από δεδομένα, οπότε και εντείνεται η σημασία αύξησης της τοπικότητας σε αυτά, και είναι συνήθως σε μεγάλο ποσοστό παραλληλοποιήσιμοι.

Έργα όπως το [2] αναδεικνύουν τη ύπαρξη τοπικότητας (locality) και ανεκμετάλλευτου memory bandwidth στους αλγορίθμους επεξεργασίας γράφων. Το γεγονός αυτό το διαπιστώσαμε και οι ίδιοι με μετρήσεις της βιβλιοθήκης ligra [3], μιας πλατφόρμας υλοποίησης τέτοιων αλγορίθμων. Στις μετρήσεις των bfs και pagerank παρατηρήθηκε μεγάλο πλήθος από καθυστερήσεις / περιόδους αδράνειας (stalls) των επεξεργαστών εξαιτίας της μνήμης, της τάξης άνω του 60% του χρόνου εκτέλεσης. Κάποια συγγράμματα προσπάθησαν να εκμεταλλευτούν το bandwidth μέσω prefetching των δεδομένων, είτε σε hardware [4], είτε σε software επίπεδο [5]. Σε άλλες περιπτώσεις δοκιμάστηκαν μέθοδοι αλλαγής του αρχικού διαμοιρασμού των δεδομένων, ή της μορφής αναπαράστασής και αποθήκευσής τους, για εξισορρόπηση της εργασίας και βελτίωση της συμπεριφοράς της cache [6]–[8].

Από τη άλλη, σε κάποια συγγράμματα έγινε προσπάθεια αύξησης της τοπικότητας των δεδομένων κάνοντας ριζικές αλλαγές στους ίδιους τους αλγόριθμους γράφων. Για την εκμετάλλευση της τοπικότητας σε παράλληλο κώδικα χρειαζόμαστε σειριοποίηση των προσβάσεων στη μνήμη, απομόνωση των threads και μέριμνα για τα NUMA χαρακτηριστικά του μηχανήματος στο οποίο θα τρέξει ο κώδικας. Χαρακτηριστικό παράδειγμα είναι η βιβλιοθήκη polymer [9], η οποία αποτελεί μία προσπάθεια τροποποίησης της βιβλιοθήκης ligra, ώστε να προσαρμοστεί στα NUMA χαρακτηριστικά των σύγχρονων shared memory μηχανημάτων. Όλα τα προγράμματα τέτοιου είδους καλούνται να διαχειριστούν με κάποιο τρόπο τα threads και την κατανομή της εργασίας σε αυτά. Το OpenMP όπως αναφέραμε διαθέτει μηχανισμούς scheduling για τον αυτόματο διαμοιρασμό της εργασίας στα threads του προγράμματος, όπως επίσης και ανακατανομής της ώστε ο διαμοιρασμός να είναι όσο το δυνατόν πιο ομοιόμορφος. Παρόλα αυτά, παρατηρήθηκε ότι σε όλες τις περιπτώσεις οι συγγραφείς κατέληγαν στη χρήση πιο χαμηλού επιπέδου βιβλιοθηκών όπως η pthreads, οι οποίες βασίζονται στον ίδιο το χρήστη για το χειρισμό των threads και το διαμοιρασμό της εργασίας. Ο λόγος είναι κατά πρώτων ότι τα schedules του OpenMP δεν είναι αρκετά ευέλικτα για να αξιοποιήσουν την τοπικότητα στα δεδομένα, ισομοιράζοντας παράλληλα την εργασία. Επιπλέον, δεν λαμβάνουν υπόψιν τους τα NUMA χαρακτηριστικά του μηχα-

νήματος, αλλά και ούτε δίνουν δυνατότητες στο χρήστη για πιο fine-grained χειρισμό των threads.

Σε αυτή τη βάση, έχουν γίνει διάφορες προσπάθειες για επέκταση του OpenMP, ώστε να αντιμετωπιστούν κάποια από αυτά τα προβλήματα. Για παράδειγμα, στο [10] προτείνεται μια μέθοδος διαχωρισμού της εργασίας σε κομμάτια, τα οποία θα μοιράζονται στα threads κατά την εκτέλεση με δυνατότητα άμεσου ελέγχου της διαδικασίας από το χρήστη. Στο [11] από την άλλη προτείνεται μια μέθοδος κλοπής εργασίας μεταξύ των threads (adaptive scheduling), η οποία αντιμετωπίζει τη δυσκολία επιλογής grain size από τον προγραμματιστή, και επιπλέον διατηρεί σε μεγάλο βαθμό την τοπικότητα που υπάρχει στον κώδικα.

Στην παρούσα εργασία προτείνουμε μια πιο γενικευμένη μορφή scheduling, την οποία και ονομάζουμε hierarchical schedule. Στόχος μας ήταν να δημιουργήσουμε ένα schedule το οποίο:

1. να μπορεί να προσαρμοστεί στα NUMA χαρακτηριστικά του κάθε μηχανήματος,
2. να είναι συμβατό με σειριακές μορφές προσπέλασης δεδομένων, με στόχο την εκμετάλλευση της τοπικότητας του κάθε προγράμματος,
3. να ανακατανέμει αποτελεσματικά, κατά την εκτέλεση του προγράμματος, το φόρτο εργασίας στα threads,
4. να δίνει τη δυνατότητα στο χρήστη να υλοποιεί πιο περίπλοκα σχήματα διαμοιρασμού της εργασίας.

Η υλοποίηση έγινε επεκτείνοντας το OpenMP, και συγκεκριμένα την εκδοχή του που πραγματοποιείται στον gcc compiler, προσθέτοντας στο σχήμα του παράλληλου for το hierarchical schedule, ως μία επιπλέον πολιτική διαμοιρασμού των επαναλήψεων των loops.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Τα Schedules του Omp

Οι βασικές επιλογές στο OpenMP ως προς τα schedules είναι δύο, το static και το dynamic schedule. Διαθέτει επιπλέον άλλα δυο (παράγωγα) schedule, το guided, το οποίο είναι dynamic με αυτόματα μεταβαλλόμενο grain size, και το runtime, στο οποίο επιλέγεται ένα από τα άλλα schedules κατά το τρέξιμο του προγράμματος (σε αντίθεση με το να επιλέγεται στο compile time και να μην μπορεί να αλλάξει).

Παρακάτω φαίνεται οι πιο συνηθισμένες μορφές σύνταξης των omp parallel for loop, και στις οποίες SCHED είναι το επιλεγμένο schedule και GS είναι το grain size. Το grain size μπορεί και να παραλειφθεί, στην οποία περίπτωση κάθε schedule έχει τη δική του default συμπεριφορά.

```
#pragma omp parallel for schedule(SCHED, GS)
for (...)
{
}

#pragma omp parallel
{
#pragma omp for schedule(SCHED, GS)
    for (...)
    {
    }
}
```

Listing 2.1: omp parallel for loop

2.1.1 Static Schedule

Για T threads και N επαναλήψεις του for loop, η default συμπεριφορά (όταν δηλαδή δεν έχει δοθεί grain size) είναι να χωρίζονται οι επαναλήψεις σε T συνεχόμενα κομμάτια μεγέθους N/T και να μοιράζονται σειριακά στα threads. Συνεπώς το thread 0 παίρνει τις $[0 \dots N/T-1]$ επαναλήψεις, το thread 1 παίρνει τις $[N/T \dots 2*N/T-1]$ επαναλήψεις κτλ.

Πλεονεκτήματα:

- Κάθε thread δρα σε ένα συνεχές κομμάτι των συνολικών επαναλήψεων, επομένως διατηρείται το data locality του προγράμματος. Επιπλέον, τα threads δρουν σε όσο το δυνατόν πιο απομονωμένα πεδία των δεδομένων, ώστε να αποφεύγονται συγκρούσεις πάνω στα ίδια ή γειτονικά δεδομένα. Συνεπώς, έχουμε συνήθως καλύτερη συμπεριφορά της cache, με λιγότερες συγκρούσεις των threads πάνω σε κοντινές θέσεις μνήμης, όπως σε atomic operations στα locks και σε concurrent data structures.

- Λόγω του ότι η διαμέριση των επαναλήψεων γίνεται στατικά, το static schedule προσθέτει ελάχιστο overhead.

Μειονεκτήματα:

- Δεν αντιμετωπίζεται τυχόν ανισοκατανομή της εργασίας στις επαναλήψεις, με αποτέλεσμα να μην γίνεται αποτελεσματική παραλληλοποίηση. Κάποια threads αναλαμβάνουν το μεγαλύτερο μέρος της δουλειάς, ενώ τα υπόλοιπα τα περιμένουν να ολοκληρώσουν την εργασία τους, χωρίς να παράγουν έργο.

Σε περίπτωση που έχει δοθεί το grain size, τότε το static schedule μοιράζει τις επαναλήψεις στα threads με round robin τρόπο, σε κομμάτια μεγέθους grain size. Στόχος είναι να ανακατευτούν οι επαναλήψεις, ώστε να αντιμετωπιστεί σε ένα βαθμό η ανισοκατανομή της εργασίας. Παρ΄ όλα αυτά, εφόσον το μοίρασμα παραμένει στατικό, δεν μπορεί να εγγυηθεί καλή κατανομή της συνολικής εργασίας, ειδικά εφόσον όλα τα threads εξακολουθούν να παίρνουν ίδιο αριθμό από επαναλήψεις (N / T).

Πλεονεκτήματα:

- Όπως αναφέρθηκε και πριν, λόγω του ότι οι επαναλήψεις μοιράζονται με interleaved τρόπο, συνήθως μετριάζεται η ανισοκατανομή εργασίας.

- Η διαμέριση πάλι γίνεται στατικά, οπότε έχουμε μικρό overhead.

Μειονεκτήματα:

- Χάνουμε τα πλεονεκτήματα της απομόνωσης των threads και της διατήρησης του data locality που είχαμε στην πρώτη περίπτωση.

- Η στατική διαμέριση δεν μπορεί να εγγυηθεί πάντα ικανοποιητική ισορρόπηση του φόρτου εργασίας.

Όταν το loop είναι balanced, η πρώτη μορφή του static schedule έχει περισσότερα πλεονεκτήματα. Όταν το loop είναι unbalanced, είναι προτιμότερος ο δυναμικός διαμοιρασμός. Επομένως, επειδή η δεύτερη μορφή σπάνια έχει τη βέλτιστη επίδοση, όταν θα αναφερόμαστε σε static schedule στο κείμενο θα εννοούμε την πρώτη μορφή, διαφορετικά θα αναφέρεται καθαρά ως η 'δεύτερη μορφή του static schedule'.

2.1.2 Dynamic Schedule

Στο dynamic schedule τα threads παίρνουν και εκτελούν κομμάτια μεγέθους grain size, από μια κεντρική δομή με το σύνολο όλων των επαναλήψεων του loop. Συγκεκριμένα, υπάρχει ένας δείκτης, ο οποίος αρχικοποιείται με τον αριθμό της πρώτης επανάληψης. Έπειτα, κάθε thread αυξάνει ατομικά (i.e. με atomic operation) τον δείκτη κατά grain size, και εκτελεί τις επαναλήψεις από την προηγούμενη έως τη νέα τιμή (i.e. $[i \dots i+grain\_size-1]$). Σε περίπτωση που δεν δοθεί, το grain size παίρνει αυτόματα την τιμή 1.

Πλεονεκτήματα:

- Επιτυγχάνεται ισοκατανομή του φόρτου εργασίας, με resolution μεγέθους grain size. Δηλαδή αντίστροφα, η σειριακή δουλειά που θα απομείνει στο τέλος λόγο ανισοκατανομής του έργου θα είναι το πολύ ίση με το μέγιστο χρονικό κόστος grain size συνεχόμενων επαναλήψεων (ακραία περίπτωση, κατά μέσο όρο αρκετά μικρότερη από αυτό).

Μειονεκτήματα:

- Λόγω της πιο περίπλοκης λογικής στο διαμοιρασμό των επαναλήψεων, όπως επίσης και των μηχανισμών συγχρονισμού των threads που εισάγονται (π.χ., τα atomic operations), το επιπλέον overhead του scheduling παύει να είναι αδιάφορο. Παρ΄ όλα αυτά, για επαρκώς μεγάλα φορτία εργασίας και ικανοποιητικού μεγέθους grain size, το κόστος του overhead, αναλογικά με αυτό της πραγματικής δουλειάς, παραμένει πολύ μικρό.

- Όπως και στη δεύτερη μορφή του static scheduling, το γεγονός ότι εναλλάσσονται τα threads στην εκτέλεση γειτονικών επαναλήψεων του loop μειώνει δραστικά την τοπικότητα των δεδομένων, και μπορεί να αυξήσει την σύγκρουση των threads στα κοινά δεδομένα.

2.2 Αρχιτεκτονικές NUMA

Στις NUMA (non-uniform memory access) αρχιτεκτονικές το σύνολο των επεξεργαστών χωρίζεται σε ομάδες, με κάθε μία να βρίσκεται σε ένα NUMA node. Τα nodes μπορεί να βρίσκονται σε ξεχωριστά sockets/packages, ή ακόμα και στο ίδιο. Κάθε node διαθέτει τη δική του κύρια μνήμη, την οποία μπορεί να προσπελαύνει άμεσα.

Στις shared memory αρχιτεκτονικές, για τις οποίες και ενδιαφερόμαστε, κάθε κόμβος μπορεί να προσπελαύνει τη μνήμη των άλλων, αλλά τα δεδομένα θα πρέπει να περάσουν πρώτα από τον κόμβο στον οποίο ανήκει η μνήμη, οπότε και εισάγεται overhead. Αντίστοιχα, αυξάνεται η πολυπλοκότητα των πρωτοκόλλων για να διατηρείται cache coherence μεταξύ των cores σε όλα τα NUMA nodes, ενώ η προσπέλαση της L3 cache ενός άλλου κόμβου καταλήγει να είναι αρκετά ακριβή, και μπορεί να φτάνει σε κόστος την προσπέλαση της τοπικής στο κόμβο κύριας μνήμης.

2.2.1 Προσπέλαση των δεδομένων στη μνήμη

Θα διακρίνουμε δύο τύπους αλγορίθμων, αυτούς που προσπελαύνουν σειριακά τα δεδομένα τους, και αυτούς που τα προσπελαύνουν τυχαία.

Σειριακή προσπέλαση δεδομένων

Σε αυτήν την περίπτωση υπάρχει τοπικότητα στα δεδομένα, την οποία οι αλγόριθμοι μπορούν και να εκμεταλλευτούν, συνεπώς είναι και η περίπτωση που μας ενδιαφέρει κυρίως.

Το static schedule είναι συμβατό με αυτή τη λογική προσπέλασης δεδομένων. Κάθε thread εκτελεί σειριακά τις επαναλήψεις του loop οι οποίες του αναλογούν, χωρίς να μπλέκεται στις επαναλήψεις των άλλων threads. Θα πάρουμε ως παράδειγμα την απλή προσπέλαση ενός πίνακα N στοιχείων μέσω παράλληλου for loop από T threads. Με static schedule, κάθε thread t θα προσπελάσει τα στοιχεία $[t * N/T \dots (t+1) * N/T)$ του πίνακα, θα περιοριστεί δηλαδή σε ένα συνεχόμενο κομμάτι της μνήμης, το οποίο θα διασχίσει σειριακά. Αυτό έχει ως συνέπεια καλύτερη συμπεριφορά της cache και της κύριας μνήμης, αλλά και μεγαλύτερα ποσοστά επιτυχίας των prefetchers των επεξεργαστών.

Αντίθετα, στο dynamic schedule ο διαμοιρασμός των block επαναλήψεων στα threads είναι ουσιαστικά τυχαίος, και μάλιστα δεν είναι καν σταθερός μεταξύ διαφορετικών εκτελέσεων του προγράμματος. Ως συνέπεια, η σειρά προσπέλασης της μνήμης (και γενικά η σειρά εκτέλεσης για κάθε thread) είναι πρακτικά αδύνατο να προβλεφθεί. Στο παράδειγμά μας, έστω ότι GS είναι το grain size του dynamic schedule, οπότε και ο πίνακας θα προσπελαύνεται σε blocks μεγέθους GS. Κάθε block από τα συνολικά N/GS θα δεχθεί επεξεργασία από ένα μόνο thread. Η εκτέλεση αρχίζει

από το πρώτο block προχωρώντας σειριακά, και όποιο thread προλαβαίνει παίρνει το επόμενο. Αν λοιπόν ένα thread διασχίζει το i block του πίνακα, το πιο πιθανό είναι να μην προλάβει να πάρει το i+1, αλλά κάποιο i+j block που βρίσκεται πιο μακριά. Τα πιο συχνά προβλήματα που δημιουργούνται είναι:

1. Όλα τα prefetched στοιχεία του i+1 block όχι μόνο πάνε χαμένα, αλλά επιπλέον μολύνουν τις L1, L2 caches με άχρηστα δεδομένα. Υπάρχει βέβαια η πιθανότητα το thread που θα πάρει το j+1 block να βοηθηθεί, βρίσκοντας δεδομένα στην L3 cache, μόνο όμως αν μοιράζονται την ίδια L3 cache, και φυσικά μόνο αν το prefetching έχει γίνει από πριν, αλλιώς απλά θα δημιουργηθεί άχρηστο traffic στο κοινό τους bus.

2. Τα αρχικά στοιχεία του i+j block δεν θα είναι prefetched, οπότε θα υποστούμε όλο το latency για να τα φέρουμε από τη μνήμη, ή τουλάχιστον από την L3 cache (δική μας ή, χειρότερα, ενός άλλου NUMA node).

3. Δυσκολεύει η δουλειά των branch predictors, αφού αν υπήρχε τοπικότητα στα branches / jumps, ανακατεύοντας τις επαναλήψεις αυτή περιορίζεται.

Πάνω σε αυτά προστίθεται ο αναγκαίος συγχρονισμός μεταξύ των threads με atomic operations, όποτε ένα από αυτά παίρνει το επόμενο block. Συνδυάζοντας τα προηγούμενα συνθέτουμε το overhead του dynamic schedule. Εδώ πρέπει να τονιστεί η σημασία του μεγέθους του grain size, αλλά και της σχέσης μεταξύ της ωφέλιμης εργασίας και του overhead. Αν εστιάσουμε στην λειτουργία του dynamic schedule εντός ενός μόνο block επαναλήψεων για κάθε thread, αυτό που προκύπτει είναι πρακτικά πανομοιότυπο με το static schedule. Το overhead δηλαδή του dynamic schedule, για επαρκώς μεγάλα grain size, υπάρχει κυρίως μεταξύ των εκτελέσεων των blocks, και συγκεκριμένα στην αρχή κάθε block. Για να γίνει πιο κατανοητό, έστω *wpi* η μέση ωφέλιμη δουλειά που χρειάζεται να γίνει σε κάθε επανάληψη, και έστω *opb* το overhead που εισάγεται σε κάθε αρχή block. Τότε η συνολική εργασία N επαναλήψεων θα είναι:

$$W_{total} = N * wpi + \frac{N}{GS} * opb \tag{2.1}$$

Ο πρώτος όρος υπολογίζει τη δουλειά που θα είχε το static schedule, όπου θεωρούμε ότι το static δεν έχει αξιοσημείωτο overhead. Ο δεύτερος όρος υπολογίζει το overhead του dynamic στην αρχή των blocks. Αυτό μπορεί να γραφτεί και ως:

$$W_{total} = N * wpi * \left(1 + \frac{opb}{wpi * GS}\right)$$

Θεωρώντας το *opb* περίπου σταθερό, βλέπουμε ότι όσο αυξάνεται ο όρος $wpi * GS$, το συνολικό overhead τείνει να γίνει αμελητέο. Ο όρος αυτός είναι ουσιαστικά το μέσο ωφέλιμο έργο σε κάθε grain size επαναλήψεις, δηλαδή στις επαναλήψεις ενός

block. Το wpi είναι χαρακτηριστικό του αλγορίθμου που εκτελείται και δεν μπορούμε να το αλλάξουμε, οπότε έχουμε έλεγχο μόνο στο GS.

Στην περίπτωση που το GS παίρνει πολύ μικρές τιμές, μπορεί να εμφανιστούν επιπλέον παράγοντες overhead. Αν το μέγεθός του είναι συγκρίσιμο με το μέγεθος των cache line των επεξεργαστών, τότε εμφανίζονται φαινόμενα false sharing, όπου διαφορετικά threads δουλεύουν στην ίδια cache line και το ένα κάνει invalidate την cache του άλλου. Σε αυτήν την περίπτωση, εμφανίζεται overhead όχι μόνο στην αρχή, αλλά καθ' όλη την εκτέλεση του block. Αν στο παράδειγμα επιπλέον θεωρήσουμε ότι για να προσπελάσουμε ένα στοιχείο του πίνακα χρειάζεται να το κάνουμε lock (ή να χρησιμοποιήσουμε atomic operation), τότε το κόστος θα είναι πολύ μεγαλύτερο. Ειδικά στην περίπτωση των NUMA μηχανημάτων, τα threads μπορεί να μην βρίσκονται στο ίδιο NUMA node, οπότε και να μην μοιράζονται την ίδια L3 cache. Σε αυτήν την περίπτωση, ένα invalidation μιας cache line σημαίνει ότι για να πάρει το thread την καινούρια τιμή, θα πρέπει να τη φέρει από πιθανώς απομακρυσμένη cache. Το αποτέλεσμα είναι να βλέπουμε πολύ μεγάλες μειώσεις της επίδοσης, που δεν μπορούν να ισοσταθμιστούν από τον καλύτερο διαμοιρασμό της εργασίας. Φυσικά, τα ίδια ακριβώς προβλήματα μπορούμε να τα συναντήσουμε και στη δεύτερη μορφή του static schedule.

Από την άλλη, αν το $wpi * GS$ είναι επαρκώς μεγάλο, τότε overhead του dynamic schedule μπορεί πράγματι να γίνει αμελητέο. Για να έχουμε ένα ποσοτικό παράδειγμα, έστω $N = 10^6, GS = 10^3$. Τότε $\frac{N}{GS} = 10^3$, συνεπώς υπάρχουν αρκετά blocks για να έχουμε ικανοποιητικό work balance. Από την εξίσωση 2.1, ο λόγος του overhead προς το ωφέλιμο έργο θα είναι $\frac{opb}{wpi * GS} = \frac{opb}{10^3 * wpi}$. Αν για παράδειγμα το opb είναι της τάξης του $10 * wpi$, τότε το συνολικό overhead θα είναι της τάξης του 1% του ωφέλιμου έργου.

Τυχαία προσπέλαση δεδομένων

Αντίθετα με πριν, σε αυτήν την περίπτωση υπάρχει ελάχιστη έως καθόλου τοπικότητα στα δεδομένα για να εκμεταλλευτούμε. Το static schedule δεν έχει κάποιο πλεονέκτημα σε σχέση με το dynamic, πέρα του ότι το δεύτερο χρησιμοποιεί επιπλέον atomic operations για τη δέσμευση των blocks επαναλήψεων. Συνεπώς, το dynamic schedule θα έχει καλύτερη επίδοση στα unbalanced φορτία, εκτός των περιπτώσεων που το grain size είναι υπερβολικά μικρό, οπότε μπορεί να εμφανίζονται και τα φαινόμενα που περιγράφηκαν πάνω.

2.2.2 Τοποθέτηση των δεδομένων στη μνήμη

Όπως αναφέρθηκε, στις αρχιτεκτονικές NUMA κάθε κόμβος επεξεργαστών διαθέτει τη δική του κύρια μνήμη, την οποία μπορεί να προσπελαύνει πολύ πιο γρήγορα

από τις μνήμες των υπόλοιπων κόμβων. Συνεπώς, είναι προτιμότερο τα threads του κόμβου να βρίσκουν τα δεδομένα που χρειάζονται στην τοπική μνήμη. Επιπλέον, στην περίπτωση των NUMA μηχανημάτων πολλοί αλγόριθμοι μπορεί να εμφανίζουν ανισοκατανομή των memory accesses, όσων αφορά τα NUMA nodes, η οποία μάλιστα μπορεί να μεταβάλλεται κατά τη διάρκεια εκτέλεσης του αλγορίθμου. Ως εκ τούτου θα πρέπει να δοθεί σημασία στον τρόπο που τοποθετούνται τα δεδομένα στη μνήμη.

Στα linux, η προεπιλεγμένη πολιτική δέσμευσης μνήμης είναι η first touch policy. Σύμφωνα με αυτή, η πραγματική δέσμευση της μνήμης δεν γίνεται αμέσως, με το που τη ζητάμε δηλαδή από τον kernel, αλλά κατά την πρώτη προσπέλασή της από το πρόγραμμα. Συγκεκριμένα, δεσμεύεται μόνο η σελίδα (memory page) στην οποία ανήκει η θέση μνήμης η οποία προσπελάστηκε. Θα πρέπει δηλαδή να αγγίξουμε μία τουλάχιστον θέση μνήμης από κάθε σελίδα της μνήμης που ζητήσαμε. Με άλλα λόγια, ενδιαφερόμαστε ουσιαστικά για τον τρόπο που θα γίνει η φάση της αρχικοποίησης των δεδομένων του προγράμματος.

Με βάση αυτά έχουμε τις εξής παρατηρήσεις:

- Για να έχει νόημα η προσεκτική τοποθέτηση των δεδομένων που συζητάμε, θα πρέπει τα threads να είναι καρφωμένα (pinned) σε ένα πυρήνα το καθένα, και να μην μετακινούνται (τουλάχιστον) μεταξύ των NUMA nodes.

- Η αρχικοποίηση θα πρέπει να γίνει παράλληλα. Διαφορετικά, ο NUMA node στον οποίο βρισκόταν το thread που έκανε την αρχικοποίηση θα συγκεντρώσει όλα τα δεδομένα, και όλα τα threads θα καταλήξουν να κάνουν πρόσβαση στη μνήμη αυτού μόνο του κόμβου.

Θα πρέπει λοιπόν κάθε NUMA κόμβος (i.e. τα threads που βρίσκονται σε αυτόν τον κόμβο) να αρχικοποιήσει τα δεδομένα τα οποία θα προσπελάσει στην εκτέλεση, ή τουλάχιστον το μεγαλύτερο μέρος αυτών. Συνεπώς, ο τρόπος τοποθέτησης των δεδομένων θα εξαρτηθεί από το είδος του αλγορίθμου. Έχουμε δύο στρατηγικές:

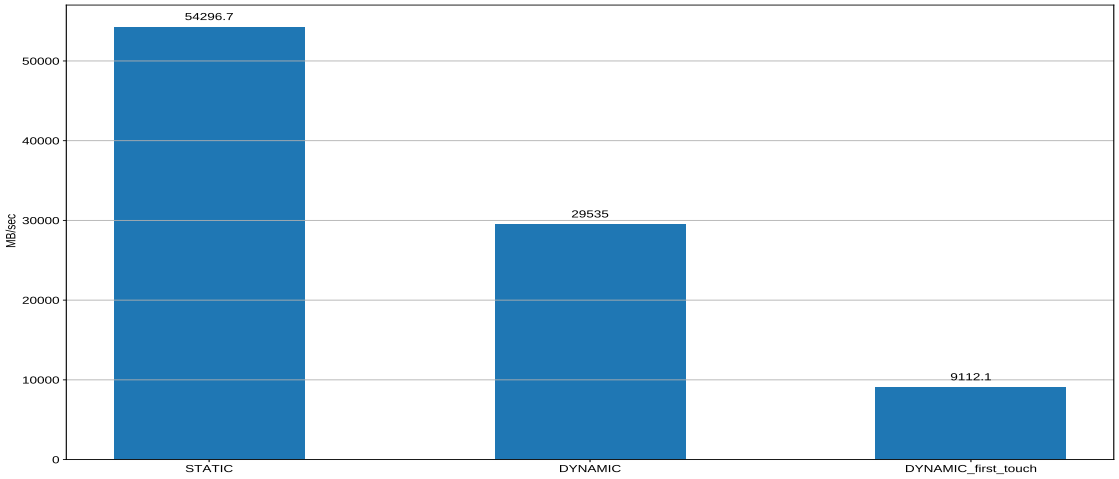
1. Απλό διαμοιρασμό των δεδομένων στους NUMA nodes με round robin τρόπο. Αυτό επιτυγχάνεται εύκολα, θέτοντας interleaved memory policy με προγράμματα όπως το numactl, χωρίς να χρειάζεται καμία άλλη παρέμβαση στον κώδικα. Επίσης, η παράλληλη αρχικοποίηση των δεδομένων χρησιμοποιώντας το dynamic schedule έχει ουσιαστικά το ίδιο αποτέλεσμα.

2. Πιο προσεκτικό διαμοιρασμό, λαμβάνοντας υπόψιν τον αλγόριθμο, ώστε οι προσβάσεις στη μνήμη να γίνονται σειριακά. Φυσικά, δεν είναι πάντα εύκολο να γνωρίζουμε ποια δεδομένα θα προσπελάσει κάθε thread, οπότε σε πολλές περιπτώσεις η στρατηγική αυτή μπορεί να μην είναι πρακτική.

Για το static schedule, θα προσεγγίσουμε το πρόβλημα με βάση το διαχωρισμό που κάναμε στην προηγούμενη ενότητα (2.2.1). Αν είναι σειριακή η προσπέλαση των δεδομένων, τότε συνήθως θα είναι εύκολο να εντοπίσουμε τα δεδομένα που θα χρειαστεί κάθε thread, οπότε μπορούμε και να του αναθέσουμε την αρχικοποίηση τους. Τις περισσότερες φορές θα αρκεί να μοιράσουμε τους πίνακες δεδομένων σε συνεχόμενα κομμάτια, πλήθους όσα είναι και τα threads. Αυτό μπορεί να επιτευχθεί εύκολα, παραλληλοποιώντας με static schedule τα loops αρχικοποίησης. Αν αντίθετα η προσπέλαση είναι τυχαία, τότε η πρώτη στρατηγική διαμοιρασμού μπορεί να είναι προτιμότερη. Πιθανή ανισοκατανομή στις προσπελάσεις των NUMA κόμβων μνήμης μπορεί να μετριαστεί με το απλό ανακάτεμα των δεδομένων ανάμεσα στους κόμβους.

Το dynamic schedule, λόγω του τρόπου που λειτουργεί, απαιτεί εντελώς διαφορετική προσέγγιση. Αντίθετα με το static, όπου κάθε thread ξεκινά από διαφορετικό σημείο στο σύνολο των επαναλήψεων, στο dynamic όλα τα threads ξεκινούν από την αρχή του loop, και διεκδικούν το επόμενο κάθε φορά κομμάτι grain size επαναλήψεων. Για να γίνει πιο κατανοητό, ας δούμε ως παράδειγμα την παράλληλη προσπέλαση ενός πίνακα, όπου τα δεδομένα έχουν μοιραστεί σε ίσα συνεχόμενα κομμάτια, ένα σε κάθε NUMA κόμβο. Στο dynamic schedule όλα τα threads θα ξεκινήσουν να παίρνουν δουλειά από το κομμάτι που περιέχει την πρώτη επανάληψη, το οποίο θα βρίσκεται αποκλειστικά σε έναν κόμβο. Όταν τελειώσουν με αυτό θα πάνε στο επόμενο κομμάτι, το οποίο πάλι θα βρίσκεται σε έναν μόνο κόμβο. Ως αποτέλεσμα, τα threads θα προσπελαύνουν τον περισσότερο χρόνο τη μνήμη ενός μόνο από τους κόμβους, παρόλο που τα δεδομένα είναι ισομοιρασμένα, μειώνοντας έτσι δραστικά το διαθέσιμο bandwidth. Συνεπώς, η interleaved (ή μία τυχαία) κατανομή των δεδομένων θα είναι σχεδόν πάντα προτιμότερη, ενώ ένας πιο προσεκτικός συνεχής διαμοιρασμός συνήθως δε βελτιώνει την επίδοση, αλλά αντίθετα μπορεί να τη βλάψει. Αυτή η ιδιαιτερότητα του dynamic schedule περιορίζει τις επιλογές των χρηστών για βελτίωση των προγραμμάτων τους.

Στο σχήμα 2.1 παρουσιάζονται μετρήσεις του throughput σε MB/sec από το stream benchmark, και συγκεκριμένα από τον add kernel. Το stream είναι ένα καθαρά στατικό και balanced φορτίο εργασίας, οπότε και βλέπουμε το static να έχει την καλύτερη επίδοση. Στο dynamic έχουμε δύο μετρήσεις, ανάλογα με τη μέθοδο αρχικοποίησης των πινάκων του προγράμματος. Στην DYNAMIC μέτρηση, η αρχικοποίηση και γενικά όλα τα παράλληλα loop εκτελέστηκαν με το dynamic schedule, οπότε τα δεδομένα μοιράστηκαν με γενικά τυχαίο τρόπο στους NUMA κόμβους. Στην DYNAMIC_first_touch μέτρηση αντίθετα, η αρχικοποίηση των πινάκων έγινε με static (η επεξεργασία τους έγινε και πάλι με dynamic schedule), ώστε τα δεδομένα να μοιραστούν με συνεχόμενο τρόπο στους κόμβους. Παρατηρούμε το φαινόμενο που περιγράψαμε πάνω, καλύτερη δηλαδή επίδοση του dynamic όταν τα δεδομένα είναι ανακατεμένα στους NUMA κόμβους σε σχέση με την ακολουθία επαναλήψεων.



Σχήμα 2.1: Add kernel του stream με $N = 2 * 10^9$ μέγεθος πίνακα και grain size = 1024

Κεφάλαιο 3

Hierarchical Schedule

Σε αυτό το κεφάλαιο θα περιγράψουμε τη λογική πίσω από το hierarchical schedule και θα παρουσιάσουμε τις λεπτομέρειες υλοποίησής του, δίνοντας έμφαση στην αιτιολόγηση των βασικών επιλογών. Η σύνταξη ενός hierarchical parallel for loop θα είναι τελείως αντίστοιχη με τα υπόλοιπα schedules του OpenMP:

```
#pragma omp parallel for schedule(hierarchical, grain_size)
for (...)
{
}

#pragma omp parallel
{
#pragma omp for schedule(hierarchical, grain_size)
    for (...)
    {
    }
}
```

3.1 Μια πρώτη προσέγγιση

Όπως περιγράψαμε στην προηγούμενη ενότητα, το dynamic schedule μειώνει αποτελεσματικά την ανισοκατανομή της εργασίας, αλλά, αντίθετα με το static schedule, αδυνατεί να εκμεταλλευτεί την τοπικότητα του προγράμματος. Θα θέλαμε λοιπόν να βρούμε έναν τρόπο να συνδυάσουμε τα δύο αυτά χαρακτηριστικά, με όσο το δυνατόν λιγότερους συμβιβασμούς. Για να το πετύχουμε αυτό, χρειάζεται τα threads να δουλεύουν σε όσο το δυνατόν πιο απομονωμένα πεδία δεδομένων, και να επικοινωνούν μεταξύ τους μόνο όταν χρειάζεται να ρυθμιστεί η κατανομή της εργασίας.

Η γενική ιδέα, για να δημιουργήσουμε το νέο αυτό schedule, είναι να ξεκινήσουμε όπως το static. Κάθε thread t από τα $0 \dots T-1$ θα αναλάβει το block επαναλήψεων

$[t * N/T \ldots (t+1) * N/T - 1]$ (συνεπώς στην αρχή θα υπάρχουν T blocks, ένα για κάθε thread). Στη συνέχεια, θα αρχίζει την εκτέλεση των επαναλήψεων από την αρχή του δικού του block, και θα συνεχίζει σειριακά σε βήματα μεγέθους "grain size" επαναλήψεων. Τα βήματα αυτά θα είναι ατομικά, υπό την έννοια ότι κανένα άλλο thread δεν θα μπορεί να του κλέψει επαναλήψεις από το βήμα που έχει δεσμεύσει. Η διαδικασία θα θα γίνεται όπως στο dynamic, δηλαδή το thread με μία ατομική εντολή (atomic add) θα αποσπά το βήμα από τη δομή με τις επαναλήψεις. Αυτή τη φορά όμως, η δομή αυτή θα μπορεί να είναι ξεχωριστή για κάθε block επαναλήψεων. Σε αυτό το σημείο έχουμε δύο περιπτώσεις, ανάλογα με το αν η δουλειά είναι ισοκατανεμημένη (balanced) ή όχι.

Σε περίπτωση που το φορτίο εργασίας είναι απόλυτα balanced, όλα τα threads θα ολοκληρώσουν τα blocks τους ταυτόχρονα. Τότε το νέο schedule θα έχει ουσιαστικά ίδια συμπεριφορά με το static, συν το overhead των ατομικών εντολών για τη δέσμευση των βημάτων. Επειδή όμως αυτές θα γίνονται από ένα μόνο thread και σε private δομή του thread αυτού, το κόστος τους θα είναι μικρό.

Η άλλη περίπτωση είναι να υπάρξει κάποια στιγμή unbalance, δηλαδή κάποιο thread να ολοκληρώσει το block επαναλήψεων που του ανήκει πριν από τα υπόλοιπα. Σε αυτό το σημείο, το thread αυτό θα πρέπει να κλέψει επαναλήψεις από το block ενός άλλου thread. Στόχος μας είναι να ελαχιστοποιήσουμε τις φορές που θα χρειαστούν τα threads να κλέψουν. Για να το πετύχουμε αυτό, αντί τα threads να κλέβουν πολλές φορές από μικρά κομμάτια δουλειάς, θα αφαιρούνται κατευθείαν οι μισές επαναλήψεις από το block του θύματος. Με αυτόν τον τρόπο, διευρύνουμε τα χρονικά πεδία στα οποία τα threads θα εκτελούν σειριακά τις επαναλήψεις τους, απομονωμένα από τα υπόλοιπα και χωρίς να διακόπτονται, διατηρώντας έτσι την τοπικότητα του κώδικα. Η διαδικασία της κλοπής θέλουμε να είναι απλή και γρήγορη, οπότε το thread που κλέβει απλά θα δεσμεύει με atomic operation (atomic add) ένα διευρυμένο βήμα, ίσο με τις μισές εναπομένουσες επαναλήψεις του block. Επιπλέον, το thread θα πρέπει να σιγουρευτεί ότι η δομή block δεν θα καταστραφεί μέχρι να ολοκληρωθεί η κλοπή. Οπότε τα βήματα θα είναι:

1. Δέσμευση της δομής με κάποιο lock για να αποτραπεί η καταστροφή του block, αλλά και για να μην πάει κάποιο άλλο thread να κλέψει.

2. Πρόσθεση στο δείκτη της αρχής του block, με ατομική εντολή (atomic add operation), του μισού του πλήθους επαναλήψεών του.

3. Απελευθέρωση του lock της δομής.

Έπειτα το thread θα δημιουργήσει μια καινούρια private δομή block, την οποία θα αρχικοποιήσει με τις επαναλήψεις που πήρε. Παρόμοια δουλειά έχει γίνει στο [11], στο οποίο αναφερθήκαμε και στην εισαγωγή.

3.2 Παρατηρήσεις και περιορισμοί

Το νέο schedule που περιγράψαμε συνεχίζει να είναι αρκετά άκαμπτο. Όπως και τα ήδη υπάρχοντα, δεν δίνει ιδιαίτερα πολλές επιλογές στον χρήστη για να το προσαρμόσει σε ειδικές περιπτώσεις, και απλά τα αντικαθιστά σε προβλήματα που ήδη μπορούσαν να λύσουν. Επίσης, όσον αφορά τη διαδικασία της κλοπής, θα πρέπει επιπλέον να ληφθεί υπόψιν η επιλογή του κατάλληλου θύματος. Για παράδειγμα, μπορεί σε κάποια μηχανήματα να είναι μη πρακτική η επιλογή ενός thread που ανήκει σε άλλο NUMA node. Χρειάζεται να δοθεί ένας πιο ευέλικτος τρόπος στους χρήστες να χειρίζονται τα threads του προγράμματός τους.

Για το διαμοιρασμό της εργασίας ουσιαστικά είναι δύο οι δρόμοι που μπορούμε να ακολουθήσουμε. Η πιο εύκολη επιλογή είναι ο όσο το δυνατόν καλύτερος διαμοιρασμός της εργασίας στατικά στην αρχή, και η χρήση του static schedule κατά την εκτέλεση, ώστε να διατηρείται η τοπικότητα των δεδομένων, μέσω του διαμοιρασμού που υλοποιήσαμε. Ο δυναμικός διαμοιρασμός της εργασίας δεν είναι απλός, καθώς κάθε αλγόριθμος χειρίζεται διαφορετικά τα δεδομένα. Το dynamic schedule του OpenMP δεν είναι αρκετά ευέλικτο για να χειριστεί πιο περίπλοκες περιπτώσεις, και όπως αναφέραμε και πριν, συχνά καταστρέφει την τοπικότητα του κώδικα. Ως αποτέλεσμα, πολλές φορές είναι υποχρεωτικός ο άμεσος χειρισμός των threads και ο διαμοιρασμός της εργασίας από τον ίδιο τον αλγόριθμο, με πιο low level βιβλιοθήκες (π.χ. pthreads). Παρ΄ όλα αυτά, η λειτουργία αυτού του είδους των προγραμμάτων συνήθως συνοψίζεται στα παρακάτω κοινά σημεία:

1. Χωρίζουν τα threads σε ομάδες (συνήθως με βάση το NUMA node που ανήκουν).

2. Μοιράζουν τις επαναλήψεις / τα δεδομένα στις ομάδες των threads, με τρόπο που να ικανοποιούνται κάποιες ιδιότητες που αφορούν τον συγκεκριμένο αλγόριθμο.

3. Εκτελούν τον αλγόριθμο, με κάθε thread να επεξεργάζεται συγκεκριμένα δεδομένα που ανήκουν σε αυτό, ή πιο γενικά στην ομάδα του.

4. Αν χρειαστεί εκκινούν μια διαδικασία ανακατανομής της εργασίας (κλέψιμο επαναλήψεων), η οποία θα πρέπει να διατηρεί τις παραπάνω ιδιότητες.

Παρατηρούμε ότι το μεγαλύτερο μέρος αυτών των βημάτων μπορεί να αυτοματοποιηθεί, και να ανατεθεί η υλοποίησή του στη βιβλιοθήκη ελέγχου των threads (OpenMP). Ο χρήστης θα αρκεί να υλοποιεί μόνο τα μέρη που αφορούν το δικό του πρόγραμμα, και η βιβλιοθήκη θα τα χρησιμοποιεί όποτε πρέπει.

3.3 Οργάνωση των threads στο hierarchical schedule

Το OpenMP υποστηρίζει nested parallelism. Κάθε thread μιας παράλληλης περιοχής μπορεί να ξεκινήσει μια νέα, με τη δημιουργία επιπλέον threads τα οποία θα ζουν μέχρι να τελειώσει η νέα περιοχή. Αυτά που απαρτίζουν κάθε nesting επίπεδο αποτελούν μία ομάδα, που στο OpenMP ονομάζεται thread team, ενώ το αρχικό thread που ξεκίνησε την ομάδα είναι ο team master. Τα ids τους στο OpenMP δεν είναι σταθερά, αλλά σε κάθε nesting επίπεδο τα threads παίρνουν σειριακά τις τιμές από το 0 μέχρι το πλήθος τους μείον 1 στο επίπεδο αυτό. Κάθε master μιας ομάδας έχει id το 0. Αν η ομάδα αυτή είναι nested, τότε ο master θα ανήκει και στην ομάδα του προηγούμενου επιπέδου, όπου το id του θα είναι διαφορετικό, και θεωρείται το parent id της nested ομάδας.

Η ομάδα του nesting επιπέδου 0, δηλαδή η αρχική ομάδα που δημιουργείται, παραμένει καθ' όλη τη διάρκεια εκτέλεσης του προγράμματος. Όταν η εκτέλεση βρίσκεται εκτός παράλληλης περιοχής (σε σειριακό κώδικα) τα threads της ομάδας αυτής δεν καταστρέφονται, αλλά περιμένουν σε ένα thread pool, μέχρι η εκτέλεση του κώδικα να φτάσει στην επόμενη παράλληλη περιοχή. Το hierarchical schedule χτίζεται πάνω στο αρχικό αυτό thread pool, και προς το παρόν δεν υποστηρίζει nested parallelism. Τα threads αυτά χωρίζονται από την αρχή του προγράμματος σε ομάδες. Για να μην υπάρχει σύγχυση με τις ομάδες των nesteds περιοχών, ονομάζουμε τις ομάδες αυτές groups. Από την αρχή λοιπόν επιλέγεται το μέγεθος των groups, είτε από τον χρήστη μέσω της μεταβλητής περιβάλλοντος "OMP_MAX_THREAD_GROUP_SIZE", είτε αυτόματα παίρνει την τιμή 1 (με την οποία κάθε thread αποτελεί ένα group). Τα threads παίρνουν επιπλέον id από το 0 μέχρι το πλήθος τους μείον 1, με τρόπο ώστε όσα βρίσκονται στο ίδιο group να έχουν συνεχόμενα ids. Τα ids του hierarchical schedule δίνονται μία φορά στην αρχή, και δεν αλλάζουν ποτέ καθ' όλη τη διάρκεια του προγράμματος. Το μέγεθος των groups θα επιλέγεται συνήθως μικρότερο ή ίσο του αριθμού των cpu cores των sockets του μηχανήματος όπου θα τρέξει ο κώδικας, ώστε τα threads κάθε group να βρίσκονται στο ίδιο socket. Σε κάθε group το πρώτο thread ορίζεται ως master, και αυτό διαχειρίζεται τις δομές του group, αλλά και αναλαμβάνει επιπλέον υποχρεώσεις, όπως την αρχικοποίηση της εργασίας.

Για παράδειγμα, αν έχουμε $T = 8$ threads και $group\_size = 3$ μέγεθος group, τότε τα groups και τα thread ids θα είναι τα εξής: $group\_0 = (0, 1, 2)$, $group\_1 = (3, 4, 5)$, $group\_2 = (6, 7)$. Τα threads 0, 3 και 6 θα είναι group masters. Όπως φαίνεται και στο παράδειγμα, αν το T δεν είναι ακέραιο πολλαπλάσιο του $group\_size$, τότε το τελευταίο group θα έχει τα εναπομείναντα threads.

Ακολουθούν οι βασικές δομές του hierarchical schedule που περιγράφουν τα threads

και τα groups.

gomp_thread_data

```
struct gomp_thread_data {
    pthread_t tid;
    int tnum;
    int tgnum;
    int tgpos;
    int max_group_size;
    int num_threads;
    struct gomp_thread_group_data * group_data;
    struct gomp_group_work_share * gws;
    struct gomp_group_work_share * gws_next;
    char padding[0] __attribute__ ((aligned (CACHE_LINE_SIZE)));
} __attribute__ ((aligned (CACHE_LINE_SIZE)));
```

Σε κάθε thread αντιστοιχίζεται ένα gomp_thread_data struct, το οποίο περιέχει τα ιδιωτικά δεδομένα του thread.

tid: Το unique id του συγκεκριμένου thread, το οποίο δίνεται από τη βιβλιοθήκη pthreads.

tnum: Το unique id του συγκεκριμένου thread, όπως αυτό περιγράφηκε πριν.

tgnum: Το id του group στο οποίο ανήκει το thread.

tgpos: Η θέση που κατέχει το thread μέσα στο group.

max_group_size: Το μέγιστο μέγεθος των groups.

num_threads: Το συνολικό πλήθος των threads.

group_data: Δείκτης προς δομή με τα δεδομένα του group στο οποίο ανήκει το thread.

gws: Δείκτης προς το κομμάτι εργασίας στο οποίο δουλεύει το group.

gws_next: Δείκτης προς το επόμενο κομμάτι εργασίας.

Το gcc στοιχείο "___attribute___ ((aligned (size)))" εξασφαλίζει ότι η μεταβλητή στην οποία αναφέρεται θα αποθηκευτεί στη μνήμη με ευθυγραμμισμένο τρόπο, δηλαδή σε θέση πολλαπλάσια του "size". Μαζί με τον πίνακα μηδενικών στοιχείων char padding[0] και θέτοντας size = CACHE_LINE_SIZE αναγκάζουμε το μέγεθος του struct να είναι πολλαπλάσιο του μεγέθους της cache line του επεξεργαστή. Σκοπός αυτού είναι να απομονωθεί από γειτονικές θέσεις μνήμης, δηλαδή στις cache lines

στα οποία βρίσκεται να μην υπάρχει θέση μνήμης η οποία να μην ανήκει στο struct. Εφόσον το struct είναι ιδιωτικό του thread και επειδή περιέχει δεδομένα τα οποία θα χρησιμοποιούνται συχνά, δεν θέλουμε να επηρεάζεται από προσπελάσεις γειτονικών θέσεων μνήμης από άλλα threads (false sharing).

Τα στοιχεία gws , gws_next τύπου "struct gomp_group_work_share *" είναι δείκτες προς δομές που περιγράφουν κομμάτια εργασίας, τα οποία θα επεξεργαστεί το group στο οποίο ανήκει το thread, και θα επεξηγηθούν πιο μετά.

gomp_thread_group_data

```
1  struct gomp_thread_group_data {
2      int tgnum;
3      int max_group_size;
4      int group_size;
5      int num_groups;
6      int master_tnum;
7      struct gomp_group_work_share * gws_buffer __attribute__
           ((aligned (CACHE_LINE_SIZE)));
8      int gws_buffer_size;
9      int gws_current;
10     char padding[0] __attribute__ ((aligned (CACHE_LINE_SIZE)));
11 } __attribute__ ((aligned (CACHE_LINE_SIZE)));
```

Σε κάθε group αντιστοιχίζεται ένα gomp_thread_group_data struct, το οποίο περιέχει πληροφορίες για το group.

tgnum:	Το unique id του συγκεκριμένου group.
max_group_size:	Το μέγιστο μέγεθος των groups.
group_size:	Το πραγματικό μέγεθος του συγκεκριμένου group.
num_groups:	Το πλήθος των groups.
master_tnum:	Το id (tnum) του master του group.
gws_buffer:	Πίνακας με τα δεδομένα των κομματιών εργασίας.
gws_buffer_size:	Πλήθος στοιχείων του gws_buffer.
gws_current:	Η θέση στο gws_buffer του τρέχοντος gws.

Κάθε τέτοιο struct προσπελαύνεται από όλα τα threads του group, επομένως μπορεί να δημιουργηθεί συνωστισμός των threads. Για να αντιμετωπιστεί όσο γίνεται το πρόβλημα, το struct χωρίζεται στα στοιχεία τα οποία παραμένουν σταθερά και στα στοιχεία τα οποία θα αλλάζουν τιμή στη διάρκεια εκτέλεσης. Ο διαχωρισμός επιτυγχάνεται με χρήση του gcc attribute "aligned (CACHE_LINE_SIZE)" το οποίο

περιγράφηκε και πριν, ώστε τα σταθερά στοιχεία να απομονωθούν στις δικές τους cache lines. (Ουσιαστικά το "gws_current" θα αλλάζει τιμή, αλλά αυτό συνδέεται στενά με τα άλλα δύο στοιχεία που βρίσκονται μαζί του.)

3.4 Διαμοιρασμός της εργασίας

Όπως και στα άλλα schedules, έτσι και στο hierarchical ο διαμοιρασμός της εργασίας εκφράζεται μέσω του διαμοιρασμού των επαναλήψεων του loop. Το master thread του group αναλαμβάνει τη διαχείριση των δομών που περιγράφουν τα κομμάτια εργασίας, όπως επίσης και το κλέψιμο επαναλήψεων από δομές άλλων groups. Θα αναφερόμαστε σε αυτές ως "work share" δομές (είτε "ws" για συντομία), και η υλοποίησή τους περιγράφεται παρακάτω.

gomp_group_work_share

```
struct gomp_group_work_share {
    gomp_group_work_share_status_t status;
    int owner_group;
    union {
        long end;
        long long end_ull;
    };
    union {
        long start;
        long long start_ull;
    } __attribute__ ((aligned (CACHE_LINE_SIZE)));
    int workers_sem __attribute__ ((aligned (CACHE_LINE_SIZE)));
    int steal_lock __attribute__ ((aligned (CACHE_LINE_SIZE)));
    char padding[0] __attribute__ ((aligned (CACHE_LINE_SIZE)));
} __attribute__ ((aligned (CACHE_LINE_SIZE)));
```

status: Η κατάσταση στην οποία βρίσκεται το κομμάτι εργασίας.

end: Η τελευταία επανάληψη, η οποία δεν συμπεριλαμβάνεται στο κομμάτι.

start: Η αρχική επανάληψη.

workers_sem: Ατομικός μετρητής του πλήθους των threads που δουλεύουν στη δομή. Προσμετρούνται μόνο threads του group στο οποίο ανήκει η δομή (άλλωστε μόνο αυτά μπορούν να πάρουν άμεσα εργασία από τη δομή).

steal_lock: Μηχανισμός lock που αποτρέπει την ανακύκλωση/καταστροφή της δομής, αλλά και την κλοπή επαναλήψεων από αυτή.

Το OpenMP έχει δύο υλοποιήσεις για κάθε schedule, μία για iterator που χωράει σε ακέραιο τύπου long, και μία για τύπο unsigned long long, ο οποίος χωράει το μεγαλύτερο 64 bit (τουλάχιστον, ανάλογα με την αρχιτεκτονική) θετικό, σε αντίθεση με τον signed long long. Η διπλή υλοποίηση είναι υποχρεωτική, γιατί ο unsigned τύπος απαιτεί ελαφρά διαφορετική μεταχείριση, εφόσον μάλιστα το OpenMP το χρησιμοποιεί και όταν ο iterator είναι signed και δεν χωρά σε long ακέραιο. Μια όμως διπλή υλοποίηση του πολύ πιο περίπλοκου hierarchical schedule θεωρήθηκε για τώρα μη πρακτική, οπότε αντί για unsigned long long χρησιμοποιήθηκε ο τύπος long long και ο κώδικας διπλασιάστηκε με τη χρήση του preprocessor της C. Οι καταλήξεις "_ull" έμειναν για ομοιομορφία με τις υπόλοιπες δομές και συναρτήσεις του OpenMP.

Το βήμα με το οποίο μεταβάλλεται ο iterator του παράλληλου OpenMP loop θα είναι γνωστό και σταθερό στην εκτέλεση, οπότε δεν χρειάζεται να αποθηκεύεται στη δομή. Αν πάρουμε για παράδειγμα βήμα "s" θετικό, τότε οι επαναλήψεις του κομματιού εργασίας θα ξεκινούν από την *start* επανάληψη και θα τελειώνουν στην $end - s$. Οι παραλλαγές με start_ull και end_ull είναι τελείως αντίστοιχες, οπότε δεν θα αναφερόμαστε σε αυτές.

Το μέλος "status" είναι ένα απλό enumeration όπως φαίνεται παρακάτω. Αυτό δείχνει σε ποια από τις δύο συνολικά καταστάσεις βρίσκεται η συγκεκριμένη gomp_group_work_share δομή. Αυτή είτε θα είναι GWS_READY, οπότε και θα είναι έτοιμη για χρήση ή ήδη θα χρησιμοποιείται από τα threads, είτε θα είναι GWS_CLAIMED, οπότε θα έχει εξαντληθεί η δουλειά που περιελάμβανε, και η δομή θα έχει ανακυκλωθεί από το master thread για χρήση ξανά στο μέλλον.

```
typedef enum {
    GWS_CLAIMED,
    GWS_READY,
} gomp_group_work_share_status_t;
```

3.4.1 Αρχικοποίηση εργασίας και αρχικός διαμοιρασμός

Περιγραφή αρχικοποίησης των work share δομών

Έστω ότι "GD" είναι η gomp_thread_group_data δομή του group. Το master thread του group αναλαμβάνει να αρχικοποιήσει μία από τις gomp_group_work_share δομές του GD.gws_buffer πίνακα. Τα βήματα της αρχικοποίησης παρουσιάζονται ακολούθως, και είναι τα ίδια που θα ακολουθούνται είτε στην περίπτωση που ξεκινάμε καινούριο loop, είτε όταν αρχικοποιούμε δουλειά που έχει κλαπεί από άλλο group. Συγκεκριμένα το master thread:

1. Βρίσκει μία ελεύθερη work share δομή από τα στοιχεία του GD.gws_buffer πίνακα, η οποία δεν χρησιμοποιείται από κανένα άλλο thread. Ας ονομάσουμε

"WS" το στοιχείο που επιλέχθηκε. Σημειώνεται ότι το WS.status θα είναι ήδη GWS_CLAIMED.

2. Θέτει το GD.gws_current να δείχνει στη WS δομή. Όπως αναφέρθηκε και πριν, το στοιχείο αυτό δείχνει την τρέχουσα δομή του GD.gws_buffer πίνακα, η οποία εμπεριέχει το κομμάτι εργασίας που το group εκτελεί αυτή τη στιγμή. Ο λόγος που το κάνουμε νωρίς σε αυτό το σημείο είναι για να διώξουμε τα threads που ψάχνουν για δουλειά (μέσω του GD.gws_current) από το προηγούμενο work share, η εργασία του οποίου θα έχει ήδη μοιραστεί πλήρως σε threads του group.

3. Δεσμεύει το WS.steal_lock, ώστε να είμαστε σίγουροι ότι κανένα άλλο master thread δεν προσπαθεί να κλέψει από αυτή τη δομή.

4. Δεσμεύει το WS.workers_sem, θέτοντάς το ίσο με μείον το μέγεθος του group (- group_data.group_size). Το στοιχείο αυτό είναι ένας σημαφόρος, που επιτελεί τη δουλειά ενός ατομικού μετρητή. Κάθε thread του group, για να μπορέσει να πάρει δουλειά από το work share, θα πρέπει πρώτα να εκτελέσει κάποια βήματα 'εισόδου' στη δομή. Το thread θα πρέπει να αυξήσει κατά 1 το σημαφόρο, και μόνο αν το αποτέλεσμα που λάβει είναι θετικός αριθμός, αν δηλαδή δεν έχει προηγουμένως δεσμευτεί από το master, θα μπορεί να πάρει δουλειά από τη δομή. Αν δεν είναι, δεν επαναλαμβάνεται η προσπάθεια, αλλά το thread ελέγχει από την αρχή ποια είναι η τρέχουσα δομή, το status της οποίας θα πρέπει να είναι GWS_READY για να δοκιμάσει να εισέλθει σε αυτήν. Αντίστοιχα, όταν η εργασία της δομής εξαντληθεί και το thread βγει από αυτήν, θα πρέπει να μειώσει το σημαφόρο κατά 1. Με αυτόν τον τρόπο, όταν ο master δεσμεύσει το σημαφόρο, η τιμή του θα είναι αδύνατο να γίνει θετική από τα υπόλοιπα threads του group, οπότε και αυτός θα έχει την αποκλειστική κυριότητα της δομής.

5. Θέτει τα WS.start, WS.end στα όρια του κομματιού εργασίας που θα αναλάβει το group και αρχικοποιεί οποιαδήποτε άλλα δεδομένα της δομής χρειάζεται. Επίσης, αποθηκεύει στο WS.owner_group το id του group στο οποίο ανήκουν οι επαναλήψεις της work share δομής. Στην περίπτωση του αρχικού διαμοιρασμού, αυτό είναι το id του ίδιου του group. Σε περίπτωση που έχει κλαπεί, θα είναι το id του group στο οποίο ανήκε εξ αρχής η δουλειά. Θα διευκρινιστεί περισσότερο στις επόμενες παραγράφους.

6. Ελευθερώνει το steal_lock και το σημαφόρο (τον θέτει ίσο με 0), και αλλάζει την κατάσταση της δομής σε GWS_READY.

Από τη στιγμή που τα threads του group δουν ότι το WS.status έχει πάρει την τιμή GWS_READY μπορούν να προσπαθήσουν να μπουν στο work share για να

πάρουν δουλειά. Αντίστοιχα, τα άλλα master threads θα μπορούν να προσπαθήσουν να κλέψουν δουλειά από το work share.

Αρχικός διαμοιρασμός

Στο hierarchical schedule, ο προεπιλεγμένος τρόπος αρχικού διαμοιρασμού των επαναλήψεων μοιάζει με αυτόν του static schedule. Η διαφορά είναι ότι, αντί να μοιράζονται απευθείας στα threads, οι επαναλήψεις μοιράζονται μεταξύ των groups. Κάθε group g από τα $0 \dots G-1$ ξεκινά με το σύνολο επαναλήψεων $[g * N/G \dots (g+1) * N/G)$. Στόχος είναι να έχουμε τα αντίστοιχα οφέλη που περιγράψαμε στην ενότητα του static schedule (2.1.1), απομονώνοντας όσο είναι δυνατόν τα πεδία στα οποία δουλεύουν τα groups. Πιο συγκεκριμένα, μπορούμε να σκεφτούμε ως παράδειγμα την περίπτωση όπου κάθε group θα βρίσκεται εξ ολοκλήρου σε ένα NUMA κόμβο. Δύο threads του ίδιου group θα δουλεύουν στο ίδιο πεδίο, τα δεδομένα του οποίο μπορούμε πιθανώς να φροντίσουμε να είναι στον NUMA κόμβο τους. Αντίθετα, δυο threads διαφορετικών group θα δουλεύουν σε διαφορετικά πεδία, αρκετά απομακρυσμένα ώστε να μην επηρεάζει το ένα το άλλο.

Πέραν όμως του προεπιλεγμένου τρόπου, δίνεται επίσης η δυνατότητα στο χρήστη να επιλέξει διαφορετικό αρχικό διαμοιρασμό των επαναλήψεων του loop στα groups. Ο χρήστης μπορεί να περάσει στη βιβλιοθήκη δική του συνάρτηση, η οποία θα εκτελείται πριν το parallel loop από κάθε thread master, και η οποία θα πρέπει να δέχεται ως είσοδο την αρχή και το τέλος του loop και να επιστρέφει ως έξοδο την αρχή και το τέλος του κομματιού επαναλήψεων που θα αναλάβει το group του master. Για το σκοπό αυτό παρέχεται η επόμενη συνάρτηση:

```
1 void omp_set_loop_partitioner(void fun(long start, long end, long
        * part_start, long * part_end));
```

Η συνάρτηση αυτή θα περνάει τη συνάρτηση αρχικοποίησης του χρήστη στη βιβλιοθήκη, και θα πρέπει να εκτελείται από το κύριο thread, αμέσως πριν από κάθε loop που μας ενδιαφέρει. Δεν επιβάλλονται περιορισμοί στην επιλογή των κομματιών επαναλήψεων. Για παράδειγμα, ο χρήστης μπορεί να αναθέσει σε κάθε group όλες τις επαναλήψεις του loop.

3.4.2 Ανακατανομή της εργασίας

Με το διαχωρισμό των threads σε ομάδες δημιουργούνται δύο λογικά επίπεδα διαμοιρασμού της εργασίας:

1. Το “local” επίπεδο των threads που ανήκουν στο ίδιο group.

2. Το "global" επίπεδο των groups και των αντιπροσώπων τους, των group master threads.

Σε κάθε επίπεδο υπάρχει και ένας μηχανισμός εξισορρόπησης του φόρτου εργασίας.

Διαμοιρασμός εντός των Groups

Η επικοινωνία εντός των groups γίνεται άμεσα μεταξύ των threads που ανήκουν σε αυτά.

Κάθε group θα έχει ένα τρέχων work share, έστω "WS", από το οποίο τα threads που ανήκουν σε αυτό θα παίρνουν επαναλήψεις προς εκτέλεση. Τα threads μέσα σε ένα group θεωρούνται ότι βρίσκονται σε κοντινές 'αποστάσεις' μεταξύ τους, και ότι η επικοινωνία ανάμεσά τους έχει σχετικά μικρό κόστος (π.χ. όταν βρίσκονται στον ίδιο NUMA κόμβο). Επίσης, θέλουμε η εργασία σε αυτό το πιο χαμηλό επίπεδο να μοιραστεί όσο το δυνατόν πιο ομοιόμορφα, οπότε θα πρέπει να την κόβουμε σε μικρά κομμάτια και αυτό θα επιφέρει συχνή επικοινωνία μεταξύ των threads. Συνεπώς, θέλουμε το overhead που προστίθεται από τον κώδικα του schedule να είναι μικρό. Γι΄ αυτό, επιλέξαμε η διαδικασία αποκόμισης επαναλήψεων να είναι όπως και στο dynamic schedule. Τα threads θα αυξάνουν με ατομική εντολή το WS.start κατά grain size, παίρνοντας έτσι προς εκτέλεση το σύνολο επαναλήψεων $[WS.start \dots WS.start + grain\_size)$. Η ίδια διαδικασία θα συνεχίζεται μέχρις ότου τελειώσουν οι επαναλήψεις του work share.

Διαμοιρασμός μεταξύ των Groups - Hierarchical Stealing

Η επικοινωνία μεταξύ των groups γίνεται δια αντιπροσώπων τους, δηλαδή μόνο μεταξύ των group master.

Τα groups θεωρούνται ότι έχουν σχετικά μεγάλο κόστος επικοινωνίας μεταξύ τους (π.χ. η επικοινωνία μεταξύ διαφορετικών NUMA κόμβων). Οπότε, σε αυτό το πιο υψηλό επίπεδο, θέλουμε να εισάγουμε το διαχωρισμό των πεδίων που αυτά δουλεύουν, αντιμετωπίζοντας παράλληλα την ανισοκατανομή της εργασίας. Θα χρησιμοποιήσουμε λοιπόν μια διαδικασία ανακατανομής της εργασίας μέσω κλοπής των επαναλήψεων άλλων group, παρόμοια με αυτή που περιγράψαμε στην ενότητα 3.1, την οποία και θα ονομάζουμε "hierarchical stealing".

Όλη η διαδικασία θα αναλαμβάνεται και πάλι από τα master threads. Να θυμίσουμε ότι κάθε group έχει μία τρέχουσα work share δομή. Το work share λοιπόν του group που επιθυμεί να κλέψει θα έχει στερέψει από επαναλήψεις και το status του θα τεθεί σε GWS_CLAIMED. Τα βήματα που θα ακολουθήσει ο master είναι τα εξής:

1. Θα διατρέξει τα work share των υπολοίπων group, και από όσα είναι στην κατάσταση GWS_READY θα επιλέξει ένα, από το οποίο θα προσπαθήσει να κλέψει επαναλήψεις. Ο τρόπος επιλογής είναι ένα επιπλέον σημείο που μπορεί να επηρεάσει ουσιαστικά την επίδοση, και θα μελετηθεί αμέσως μετά.

2. Αφού επιλέξει μια work share δομή, έστω "WS_remote", θα πρέπει να δεσμεύσει τη δομή, μέσω του steal_lock στοιχείου της, ώστε να είμαστε σίγουροι ότι η δομή δεν θα καταστραφεί/αποσυρθεί για όσο τη χρειαζόμαστε, αλλά και ότι δεν θα προσπαθήσουν άλλα master threads να κλέψουν από αυτήν.

3. Σε αυτό το σημείο μπορεί να πραγματοποιηθεί η κλοπή. Όπως και στην ενότητα 3.1, θα κλαπούν οι μισές επαναλήψεις του WS_remote. Θεωρητικά, εφόσον τα threads του ξένου group εκτελούν επαναλήψεις από την αρχή του work share, είναι προτιμότερο να κλαπούν επαναλήψεις από το τέλος του, όπως γίνεται και στο [11]. Παρ΄ όλα αυτά, κάτι τέτοιο προϋποθέτει πιο περίπλοκο σχεδιασμό, εφόσον θα καθιστά πλέον την τελευταία επανάληψη του work share μεταβαλλόμενη, κάτι που θα επιβαρύνει με επιπλέον εντολές συγχρονισμού και ελέγχους ορθότητας την εκτέλεση, και μάλιστα όχι μόνο κατά την κλοπή, αλλά σε κάθε δέσμευση ενός κομματιού grain size επαναλήψεων. Επομένως, η κλοπή γίνεται κατ΄ αντιστοιχία με τρόπο που τα threads στο εσωτερικό των groups δεσμεύουν βήματα εργασίας, αυξάνοντας δηλαδή το WS_remote.start με ατομική εντολή κατά το μισό των επαναλήψεων που έχουν μείνει.

4. Απελευθερώνει το steal_lock.

Μετά την κλοπή, ο master θα αρχικοποιήσει με τα όρια του τμήματος επαναλήψεων που πήρε μία από τις work share δομές του group του, έστω "WS", με τον τρόπο που περιγράφηκε στη ενότητα 3.4.1. Επίσης, αποθηκεύει στο WS.owner_group την τιμή του WS_remote.owner_group. Σε οποιαδήποτε δηλαδή στιγμή, το στοιχείο "owner_group" κάθε work share δομής περιέχει το id του group στο οποίο ανατέθηκαν οι επαναλήψεις της δομής κατά τον αρχικό διαμοιρασμό.

3.4.3 Επιλογή θυμάτων στο hierarchical stealing

Σημαντικό σημείο στην όλη διαδικασία της κλοπής μεταξύ κόμβων είναι η επιλογή του κατάλληλου θύματος, δηλαδή group από το οποίο θα κλέψουμε την εργασία. Παρότι η διαδικασία επιλογής δεν θέλουμε να είναι πολύ κοστοβόρα για να μην επιβαρύνει το πρόγραμμα, θα πρέπει να επισημάνουμε ότι το πλήθος των αναγκαίων κλοπών με hierarchical stealing μεταξύ των groups θα είναι περιορισμένο. Αν σκεφτούμε τον τρόπο που γίνεται η κλοπή, μπορούμε να καταλάβουμε το γιατί. Συγκεκριμένα, ας πάρουμε ως παράδειγμα την περίπτωση που έχουμε δύο group, και έστω έχουμε N επαναλήψεις, από τις οποίες το group 1 θα πάρει τις πρώτες μισές και το group 2 τις υπόλοιπες. Ως μια κακή περίπτωση κατανομής της εργασίας, που να είναι όμως δυνατόν να διορθωθεί με ανακατανομή της εργασίας, μπορούμε να σκεφτούμε αυτή που η εργασία αυξάνεται ανάλογα με τον αριθμό της επανάληψης, οπότε το group 2 θα έχει πάρει την περισσότερη. Το group 1 θα αναγκάζεται να κλέβει από το group 2, αλλά πάντα το δεύτερο θα έχει περισσότερη εργασία από το πρώτο. Ακόμα και

τότε, το group 1, επειδή μέσω hierarchical stealing θα κλέβει κάθε φορά τις μισές εναπομένουσες επαναλήψεις του group 2, θα χρειαστεί να κλέψει το πολύ $\log_2 \frac{N}{2}$ φορές. Μάλιστα, στην περίπτωση με 2 μόνο group, όποια και να είναι η κατανομή του φόρτου στις επαναλήψεις, το μέγιστο πλήθος κλοπών θα είναι και πάλι $\log_2 \frac{N}{2}$. Αυτό διότι, για να δημιουργηθεί η ανάγκη για κλοπή, θα πρέπει να έχουν εξαντληθεί οι μισές από τις συνολικές επαναλήψεις που είχαν μείνει από την προηγούμενη χρονικά κλοπή, από οποιοδήποτε από τα δύο group. Σε περιπτώσεις με περισσότερα threads βέβαια θα έχει σημασία η επιλογή του θύματος, αλλά και πάλι ο ίδιος μηχανισμός θα περιορίζει σημαντικά το πλήθος των κλοπών που θα χρειαστούν. Είναι προτιμότερο λοιπόν να σπαταλήσουμε λίγο παραπάνω χρόνο για να βρούμε το σωστό group θύμα.

Έγινε προσπάθεια η υλοποίηση της πολιτικής του hierarchical stealing να επιτρέπει την εύκολη επεκτασιμότητα της με διαφορετικά κριτήρια στο μέλλον. Για το σκοπό αυτό χρησιμοποιήθηκε ένα σύστημα πόντων. Τα work structs των groups αξιολογούνται από το master του group που θέλει να κλέψει και βαθμολογούνται με πόντους, ανάλογα με μια σειρά από κριτήρια που εφαρμόζονται σε φάσεις. Μετά από διάφορες δοκιμές και μετρήσεις, παρατηρήσαμε ότι, τουλάχιστον στα δικά μας μηχανήματα, το πιο σημαντικό κριτήριο ουσιαστικά ήταν η κλοπή από το group με το μεγαλύτερο μέγεθος εναπομένουσας εργασίας. Αυτό φυσικά δεν μπορούμε να το γνωρίζουμε, οπότε το κριτήριο που εφαρμόστηκε ήταν η εύρεση του group με το μεγαλύτερο πλήθος επαναλήψεων. Αυτό είναι λογικό, διότι έτσι μειώνεται το πλήθος των κλοπών, αλλά και ο κατακερματισμός της εργασίας. Έτσι, τα βήματα που ακολουθούνται στη συγκεκριμένη υλοποίηση είναι τα παρακάτω, και όπως πάντα εκτελούνται όλα από το master του group:

1. Φιλτράρονται έξω όλα τα work share των groups που δεν έχουν πλέον εργασία.

2. Από αυτά που έμειναν, βρίσκεται το work share που περιέχει το μεγαλύτερο πλήθος επαναλήψεων προς εκτέλεση, οι οποίες έστω είναι "max_iter".

3. Τα work share βαθμολογούνται με βάση το πλήθος των επαναλήψεών τους. Πιο συγκεκριμένα γίνεται μία χβαντοποίηση των επαναλήψεων με βάση το max_iter. Στην υλοποίηση έχει επιλεχθεί ένας αριθμός, έστω "ceil", ο οποίος θα είναι η μέγιστη βαθμολόγηση του βήματος αυτού. Υπολογίζεται ο αριθμός $div = \frac{max\_iter}{ceil}$, και έπειτα για κάθε work share με έστω "iter" επαναλήψεις, η βαθμολογία του υπολογίζεται ως: $score = \frac{iter}{div}$. Η χβαντοποίηση αυτή μπορεί να εξισώσει work share με παραπλήσια πλήθη επαναλήψεων, αλλά το σφάλμα της τάξης του $\frac{max\_iter}{div}$ δεν θα επηρεάσει σοβαρά την επίδοση. Ο αριθμός ceil επιλέγεται να είναι δύναμη του 2, ώστε οι διαιρέσεις να μετατρέπονται σε απλές ολισθήσεις των bits (όλοι οι αριθμοί και τα αποτελέσματα είναι ακέραιοι). Στη συγκεκριμένη υλοποίηση επιλέχθηκε $ceil = 2^6$.

4. Εξετάζουμε το group στο οποίο ανήκουν οι επαναλήψεις του κάθε work share. Σημειώνεται ότι μπορεί να είναι διαφορετικό group από τον ιδιοκτήτη της δομής

(δηλαδή να έχουν κλαπεί ξανά από αλλού), και εμάς μας ενδιαφέρει ο αρχικός ιδιοκτήτης, ο οποίος είναι πιο πιθανό να έχει και τα δεδομένα τα οποία θα προσπελαστούν στις επαναλήψεις αυτές. Αυτό επιτυγχάνεται μέσω του owner_group στοιχείου της work share δομής, το οποίο, όπως αναφέρθηκε και σε προηγούμενη ενότητα, περιέχει το group στο οποίο ανατέθηκαν οι επαναλήψεις της δομής κατά τον αρχικό διαμοιρασμό. Αν βρίσκεται λοιπόν στον ίδιο NUMA κόμβο με το group του master τότε παίρνει επιπλέον πόντους. Στη δικιά μας υλοποίηση επιλέχθηκε ένα μετριοπαθές bonus, αυξάνουμε δηλαδή απλά κατά 1 τη βαθμολογία για να σπάσουμε ισοβαθμίες. Σε άλλα μηχανήματα με πιο έντονα NUMA χαρακτηριστικά θα μπορούσε να επιλεχθεί πιο ακραία μεροληψία.

5. Στο τέλος γίνεται μία ταξινόμηση των work share που έχουν μείνει, με βάση τη βαθμολόγησή τους. Μας ενδιαφέρει η ταξινόμηση να γίνεται γρήγορα. Το βήμα 3 παράγει βαθμολογήσεις με μικρούς αριθμούς, ώστε να είναι πρακτικοί οι αλγόριθμοι ταξινόμησης γραμμικού χρόνου, όπως ο counting sort.

Σε άλλα μηχανήματα μπορεί να υπάρχουν επιπλέον κριτήρια που να παίζουν σοβαρό ρόλο, για παράδειγμα η τοπολογία του μηχανήματος να καθιστά κάποια ζεύγη από group πολύ απομακρυσμένα για να είναι πρακτική η κλοπή μεταξύ τους. Σε αυτές τις περιπτώσεις το σύστημα πόντων καθιστά εύκολη την επέκταση του υπάρχοντος με οποιαδήποτε άλλα κριτήρια.

3.5 Επέκταση λειτουργικότητας

Ένας από τους στόχους της εργασίας είναι η επέκταση της λειτουργικότητας του OpenMP, ώστε να δοθεί στο χρήστη μεγαλύτερος έλεγχος πάνω στην οργάνωση των threads και το διαμοιρασμό της εργασίας σε αυτά. Η οργάνωση σε groups με τον τρόπο που έχει περιγραφεί μας δίνει τη δυνατότητα μεγαλύτερης ευελιξίας, ώστε το schedule να προσαρμόζεται καλύτερα σε κάθε αλγόριθμο αλλά και σε κάθε μηχάνημα. Ο χρήστης δεν χρειάζεται να ελέγχει άμεσα τα threads όπως σε πιο low-level βιβλιοθήκες (π.χ. pthreads), αλλά αντί για αυτού του προσφέρονται συναρτήσεις με τις οποίες μπορεί να ρυθμίσει τη συμπεριφορά του schedule.

Καταρχήν, υλοποιούνται συναρτήσεις με τις οποίες ο χρήστης λαμβάνει βασικές πληροφορίες σχετικά με τις καινούριες έννοιες που εισάγαμε:

```
1 int omp_get_cpu_node_size();             // το μέγεθος των NUMA κόμβων
2 int omp_get_max_thread_group_size();     // μέγιστο μέγεθος group
3 int omp_get_thread_group_size();         // πραγματικό μέγεθος του group
4 int omp_get_num_thread_groups();         // πλήθος των group
5 int omp_get_thread_group_num();          // id του group
6 int omp_get_thread_group_pos();          // θέση στο group (tgpos)
7 int omp_get_thread_group_master_num(); // id του master
```

Έπειτα, πολλοί αλγόριθμοι απαιτούν συγκεκριμένο διαμοιρασμό σε κάθε group, ώστε να μην είναι επιτρεπτή η επεξεργασία ξένων σε αυτό δεδομένων, παρά μόνο των private δικών του. Άλλες φορές μπορεί το κόστος μεταφοράς δεδομένων μεταξύ των groups να είναι ασύμφορα μεγάλο. Ο χρήστης έχει την επιλογή να απενεργοποιήσει τελείως το hierarchical stealing. Δίνονται δύο συναρτήσεις:

```
int omp_get_hierarchical_stealing();
void omp_set_hierarchical_stealing(int v);
```

Η πρώτη επιστρέφει την κατάσταση του hierarchical stealing, ενώ η δεύτερη το ενεργοποιεί ή το απενεργοποιεί, ανάλογα με την τιμή που της δοθεί. Χρησιμοποιώντας αυτές τις συναρτήσεις και μαζί με τη συνάρτηση omp_set_loop_partitioner(), ο χρήστης μπορεί εύκολα να ορίσει επακριβώς τις επαναλήψεις που θα αναλάβει κάθε group. Τα groups δεν θα μπορούν να κλέβουν εργασία, αλλά στο εσωτερικό τους θα συνεχίσει να υπάρχει η εξισορρόπηση του φόρτου, όπως αυτή έχει περιγραφεί στο πρώτο μέρος της ενότητας 3.4.2.

Σε πολλές περιπτώσεις, αυτοί οι αλγόριθμοι μπορούν να προσαρμοστούν ώστε να υποστηρίζουν επιπλέον διαμοιρασμό της εργασίας και μεταξύ των groups. Τέτοιου είδους πρόγραμμα είναι για παράδειγμα το polymer. Για να το πετύχουν αυτό, τα groups που κλέβουν εργασία χρειάζεται να προετοιμάσουν με κάποιο τρόπο τα ξένα δεδομένα. Θα εκτελέσουν δηλαδή μετά από την κλοπή κάποιο κομμάτι κώδικα, ώστε να μπορούν να επεξεργαστούν τα ξένα δεδομένα (ή πιο γενικά τις ξένες επαναλήψεις) σαν να ήταν δικά τους. Για αυτό το σκοπό προσφέρεται στο χρήστη οι παρακάτω συναρτήσεις:

```
void omp_set_after_stealing_fun(void fun(int owner_group, long
        start, long end));
int omp_get_gws_owner_thread_group_num();
```

Ο χρήστης θα υλοποιεί μία δική του συνάρτηση που θα δέχεται ως ορίσματα τα όρια του πεδίου επαναλήψεων που έκλεψε μαζί με τον αρχικό ιδιοκτήτη αυτών. Τη συνάρτηση αυτή θα την περνάει στο OpenMP μέσω της omp_set_after_stealing_fun, και οι master των groups θα την εκτελούν αμέσως μετά από κάθε κλοπή εργασίας, ακριβώς πριν αυτή δοθεί στα threads προς εκτέλεση (πριν δηλαδή το βήμα 6 της αρχικοποίησης των work share). Επιπλέον, μέσω της "omp_get_gws_owner_thread_group_num()" κάθε thread μπορεί να μάθει το αρχικό group που ανήκαν οι επαναλήψεις του work share στο οποίο δουλεύει.

3.6 Αλλαγές στον κώδικα του gcc

Σε αυτήν την ενότητα θα παρουσιάσουμε επιγραμματικά τα αρχεία πηγαίου κώδικα του gcc που χρειάστηκε να επεκτείνουμε, αλλά και τα αρχεία που προσθέσαμε.

Οι διαδρομές (paths) των αρχείων είναι σχετικές ως προς το βασικό φάκελο (base folder) με τον κώδικα του gcc. Στόχος μας στην υλοποίηση ήταν η μικρότερη δυνατή μεταβολή του ήδη υπάρχοντα κώδικα, ώστε να μην επηρεάζουμε την επίδοση των άλλων schedule κατά την εκτέλεση και να έχουμε έγκυρες μετρήσεις. Συνεπώς, ο κώδικας του hierarchical είναι όσο το δυνατόν πιο διακριτός από τον υπόλοιπο κώδικα. Σε τελικό στάδιο θα μπορούσε να ενσωματωθεί σε μεγαλύτερο βαθμό, το οποίο μπορεί να βελτίωνε και ελάχιστα την επίδοση.

- libgomp/libgomp.h
  Το αρχείο αυτό περιγράφει τις κύριες δομές του OpenMP, από τις οποίες επεκτείναμε δύο. Συγκεκριμένα, στην "struct gomp_thread" δομή προσθέσαμε ένα "struct gomp_thread_data * t_data" στοιχείο με τα ιδιωτικά δεδομένα του thread για το hierarchical schedule, και αντίστοιχα στην "struct gomp_thread_pool" δομή ένα "struct gomp_thread_group_data ** groups" πίνακα με τις πληροφορίες για όλα τα group.
- libgomp/team.c , libgomp/pool.h
  Στα αρχεία αυτά περιγράφονται η δημιουργία και η λειτουργία των ομάδων (teams) του OpenMP και των thread pools αντίστοιχα, όπως περιγράψαμε στην ενότητα 3.3. Επεκτείναμε τη λειτουργικότητα, ώστε κατά την αρχικοποίηση να δημιουργούνται και δικές μας δομές για το hierarchical schedule, με τις οποίες σχηματίζονται τα thread groups.
- libgomp/parallel.c
  Σε αυτό το αρχείο προσθέσαμε τις διάφορες συναρτήσεις που προσφέρονται στον χρήστη του OpenMP, οι οποίες είτε προσφέρουν πληροφορίες (π.χ. το μέγεθος του group), είτε μεταβάλουν τη συμπεριφορά του hierarchical (π.χ. η συνάρτηση omp_set_loop_partitioner() την οποία αναφέραμε στην ενότητα 3.4.1).
- libgomp/loop.c , libgomp/loop_ull.c
  Εδώ γίνονται οι αρχικοποιήσεις των παράλληλων loop, στις οποίες προσθέσαμε περιπτώσεις για το δικό μας schedule.
- libgomp/iter.c , libgomp/iter_ull.c
  Οι συναρτήσεις αυτές καλούνται στην αρχή κάθε κομματιού επαναλήψεων (επομένως στο static καλούνται μία μόνο φορά). Προσθέσαμε τη συνάρτηση gomp_iter_hierarchical_next() η οποία ουσιαστικά είναι απλό wrapper για την gomp_iter_l_ull_hierarchical_next() που περιέχεται στο αρχείο libgomp/hierarchical_schedule/iter_hierarchical.h και υλοποιεί το κύριο μέρος του hierarchical schedule.
- libgomp/env.c
  Εδώ γίνεται το parsing των μεταβλητών περιβάλλοντος και επιπλέον ορίζονται

global μεταβλητές που καθορίζουν τη συμπεριφορά του OpenMP. Σε αυτές προσθέσαμε τις δικές μας, που αφορούν το hierarchical schedule.

- libgomp/icv.c , libgomp/libgomp.map , libgomp/libgomp_g.h , libgomp/omp.h.in , libgomp/omp_lib.f90.in , libgomp/omp_lib.h.in
  Υπόλοιπα αρχεία, που αναφέρονται κυρίως σε declarations, στην ορατότητα των συναρτήσεων του hierarchical κατά το compiling και linking προγραμμάτων με τη OpenMP βιβλιοθήκη και άλλες λεπτομέρειες.

Στην ιεραρχία έχει προστεθεί ο φάκελος "hierarchical_schedule", ο οποίος περιέχει τα αρχεία με τον κύριο όγκο του κώδικα του hierarchical schedule.

- libgomp/hierarchical_schedule/hier_sched_structs.h
  Στο αρχείο αυτό ορίζονται όλες οι δομές που χρησιμοποιούνται στο hierarchical schedule.
- libgomp/hierarchical_schedule/iter_hierarchical.h
  Ο κύριος κώδικας του hierarchical schedule, από την αρχικοποίηση της εργασίας και το διαμοιρασμό μεταξύ των group, μέχρι τη κλοπή επαναλήψεων μεταξύ των work shares.
- libgomp/hierarchical_schedule/stealing_policy_scores.h
  Οι πολιτικές του hierarchical stealing, με τις οποίες επιλέγεται το θύμα κατά τη διαδικασία της κλοπής. Εδώ υλοποιούνται οι φάσεις βαθμολόγισης του κάθε work struct πριν την κλοπή.
- libgomp/hierarchical_schedule/macros.h
  Διάφορες μακροεντολές της C που χρησιμοποιούνται στον κώδικα.

Τέλος, παρατίθενται τα αρχεία που έχουν να κάνουν κυρίως με το parsing κομμάτι της υλοποίησης, ώστε να αναγνωρίζεται ο όρος "hierarchical" ως ένα νέο schedule στις "#pragma omp for schedule()" εντολές. Ουσιαστικά, οπουδήποτε στον κώδικα υπήρχε αναφορά στα διάφορα schedules του OpenMP, προσθέσαμε και το δικό μας.

- gcc/c/c-parser.c
- gcc/c/c-typeck.c
- gcc/cp/parser.c
- gcc/cp/semantics.c
- gcc/omp-builtins.def
- gcc/omp-expand.c
- gcc/omp-general.c
- gcc/params-enum.h

- gcc/params-list.h
- gcc/params-options.h
- gcc/params.c
- gcc/params.def
- gcc/tree-core.h
- gcc/tree-parloops.c
- gcc/tree-pretty-print.c

Κεφάλαιο 4

Μετρήσεις

Σε αυτό το κεφάλαιο παρουσιάζονται μετρήσεις από διάφορα προγράμματα / benchmarks, στις οποίες θα αξιολογηθούν οι αποδόσεις του κάθε schedule. Οι μετρήσεις έγιναν σε ένα intel μηχάνημα τεσσάρων NUMA κόμβων. Κάθε κόμβος είναι ένα socket με έναν Xeon E5-4620 επεξεργαστή των 8 πυρήνων, και διαθέτει 64GB μνήμη. Συνολικά λοιπόν το μηχάνημα έχει 32 πυρήνες και 256GB μνήμη. Κάθε πυρήνας διαθέτει 32KB L1 data cache, 32KB L1 instruction cache και 256KB L2 cache. Κάθε socket διαθέτει επίσης 16384KB L3 cache, την οποία και μοιράζονται οι 8 πυρήνες του. Οι επεξεργαστές επιπλέον υποστηρίζουν hyper-threading (δηλαδή 64 συνολικά λογικούς πυρήνες), παρ΄ όλα αυτά δεν χρησιμοποιείται στις μετρήσεις. Τα threads είναι πάντα καρφωμένα (pinned) στους πυρήνες, το οποίο το πετυχαίνουμε θέτοντας τα affinities τους μέσω της "GOMP_CPU_AFFINITY" μεταβλητής περιβάλλοντος του gnu OpenMP. Επίσης, σε κάθε μέτρηση, τα ατομικά βήματα των επαναλήψεων που θα εκτελούν τα threads στο dynamic θα είναι πάντα ίδιου μεγέθους με αυτά του hierarchical (στο διαμοιρασμό στο εσωτερικό των groups), και θα αναφερόμαστε στο μέγεθος αυτό ως grain size.

4.1 Stream

Αρχικά παρουσιάζονται μετρήσεις του stream benchmark, στο οποίο έχουμε ήδη αναφερθεί στην ενότητα 2.2.2. Το stream είναι ένα σύνηθες benchmark, το οποίο χρησιμοποιείται για τη μέτρηση του bandwidth αλγορίθμων και μηχανημάτων. Χρησιμοποιεί τέσσερα kernel:

1. Copy: Αντιγραφή πίνακα.
2. Scale: Πολλαπλασιασμό στοιχείων πίνακα με αριθμό.
3. Add: Πρόσθεση δύο πινάκων.

4. Triad: Γραμμικός συνδυασμός δύο πινάκων (i.e. $a = b + scalar * c$).

Ως φορτίο εργασίας είναι συνεπώς στατικό και απόλυτα balanced. Λόγω του ότι είναι ένα πολύ απλό και προβλέψιμο πρόγραμμα, μπορούμε μέσω αυτού να δούμε πιο καθαρά τη συμπεριφορά του hierarchical schedule.

Σημειώνεται ότι, ακόμα και σε loops τα οποία είναι φαινομενικά απόλυτα balanced (π.χ. αρχικοποίηση ενός πίνακα), δεν είναι απαραίτητο ότι όλα τα threads θα ολοκληρώσουν την εργασία τους ταυτόχρονα. Μπορεί κάποια threads να καθυστερήσουν για κάποιο λόγο (π.χ. τα δεδομένα τους έτυχε να βρίσκονται σε μακρινό NUMA κόμβο), ή να ολοκληρώσουν γρηγορότερα την εργασία τους (π.χ. να αυξηθεί η συχνότητα των πυρήνων στους οποίους βρίσκονται λόγω turbo boost και αντίστοιχων τεχνικών). Επομένως, σε όλες τις μετρήσεις του hierarchical schedule στο stream, το work stealing μεταξύ των thread groups είναι ενεργό, ώστε να συμπεριλαμβάνεται το επιπλέον overhead που πιθανά αυτό επιφέρει. Δοκιμάζονται δύο μεγέθη πινάκων:

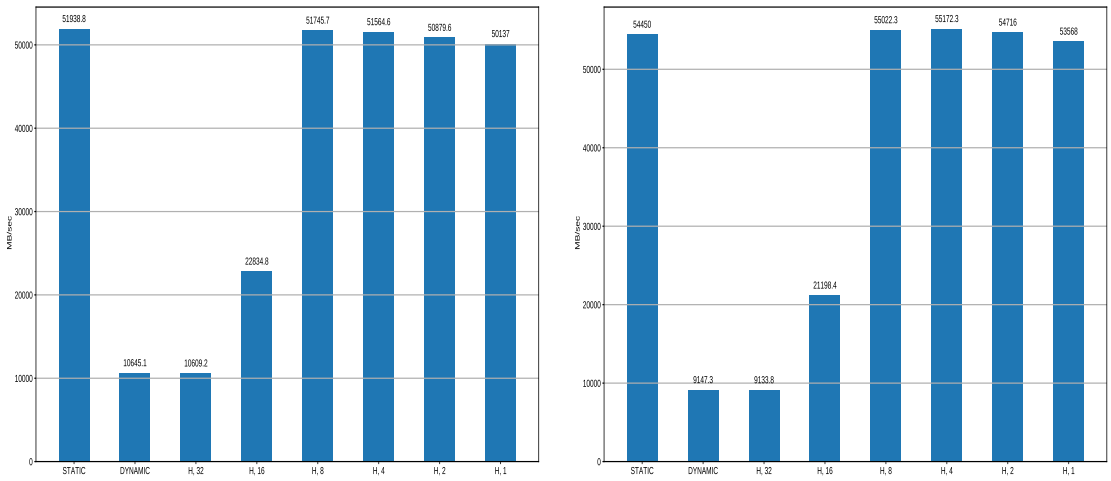
- Μικροί πίνακες της τάξης των 10^7 στοιχείων, με συνολική δέσμευση μνήμης από το πρόγραμμα τη τάξης των 200MB.
- Σχετικά μεγάλοι πίνακες της τάξης των $2*10^9$ στοιχείων, με συνολική δέσμευση μνήμης από το πρόγραμμα τη τάξης των 44.7GB. Θυμίζουμε ότι η συνολική μνήμη του μηχανήματος είναι 256GB.

Επίσης δοκιμάζονται δύο εκδοχές, ανάλογα με τον τρόπο αρχικοποίησης των πινάκων.

Αρχικοποίηση με static schedule

Στην πρώτη περίπτωση οι πίνακες αρχικοποιούνται με static schedule. Συνεπώς, με τη first touch policy των linux κάθε πίνακας του προγράμματος μοιράζεται σε συνεχόμενα κομμάτια στους NUMA κόμβους, ένα κομμάτι στον κάθε κόμβο. Με αυτή την αρχικοποίηση παίρνουμε μετρήσεις του bandwidth για τα διάφορα schedule, και επιπλέον στο hierarchical για τα διαφορετικά μεγέθη group. Για μέγεθος ίσο με 1 κάθε thread δημιουργεί το δικό του group, στο οποίο είναι και master. Για μέγεθος ίσο με 32, και εφόσον έχουμε 32 πυρήνες χωρίς να χρησιμοποιούμε το hyper-threading, όλα τα threads θα ανήκουν στο μοναδικό group, οπότε και το hierarchical schedule μετατρέπεται ουσιαστικά σε dynamic.

Επίσης, πρέπει να σημειωθεί ότι τα threads καρφώνονται σε πυρήνες με τέτοιο τρόπο, ώστε όσα ανήκουν στο ίδιο group να καταλαμβάνουν συνεχόμενες θέσεις (τα threads του πρώτου group θα βρίσκονται στους "group size" πλήθους πρώτους πυρήνες, του δεύτερου group στους επόμενους κ.τ.λ.). Συνεπώς, στα μεγέθη 1, 2, 4 και 8 όλα τα threads κάθε group θα βρίσκονται στον ίδιο NUMA κόμβο. Ομοίως, στα μεγέθη 16 και 32 τα groups θα εκτείνονται σε 2 και 4 αντίστοιχα κόμβους.



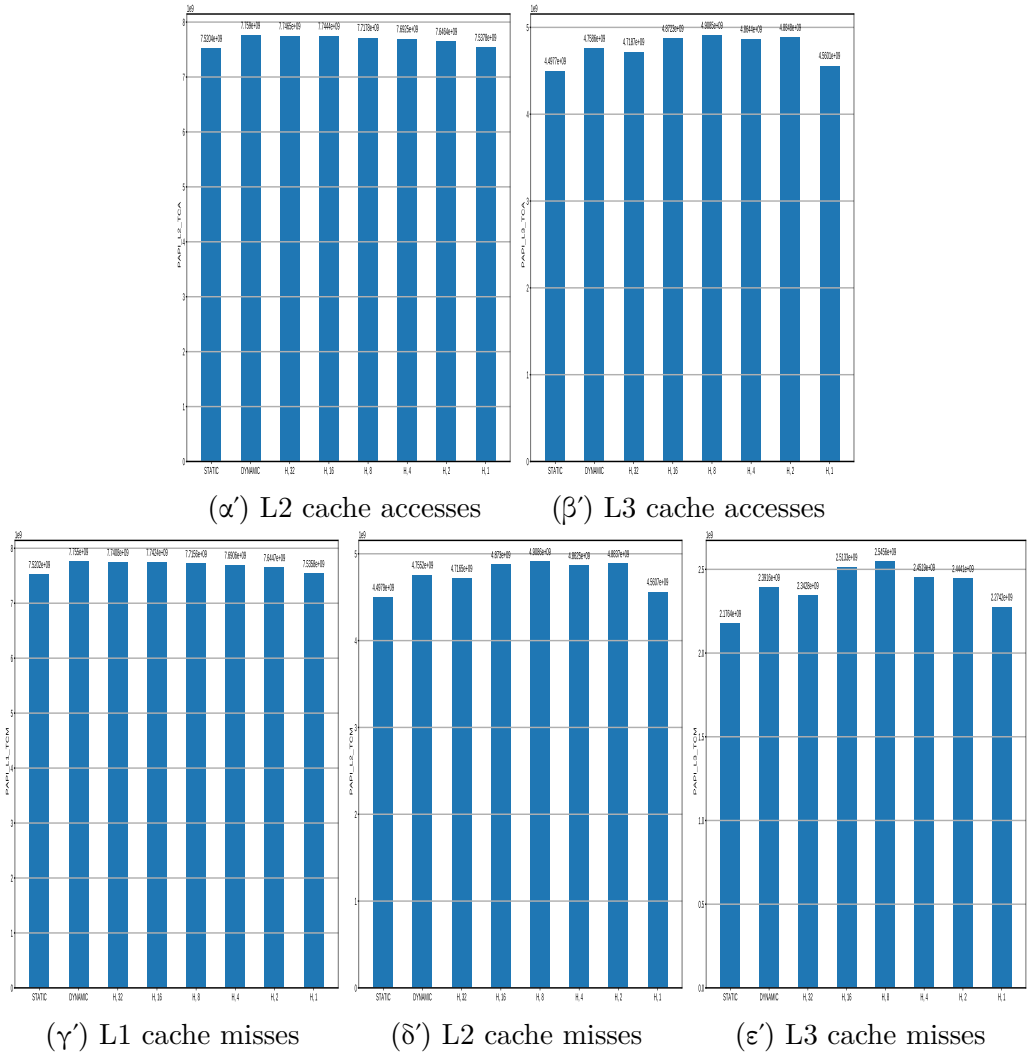
(α′) N = 10^7 (β′) N = $2 * 10^9$

Σχήμα 4.1: Stream. Bandwidth διαγράμματα σε MB / sec του Add kernel του stream, με grain size = 1024 και "N" μέγεθος πινάκων. Τα δεδομένα αρχικοποιούνται με static schedule.

Στο σχήμα 4.1 παρουσιάζονται διαγράμματα για τον Add kernel του stream (τα υπόλοιπα kernels έχουν ακριβώς την ίδια συμπεριφορά). Οι ταμπέλες για το hierarchical schedule είναι της μορφής "H,group size". Παρατηρούμε ίδιες συμπεριφορές των schedules στα δύο μεγέθη πινάκων. Καταρχήν, βλέπουμε ότι το dynamic έχει πολύ κακή επίδοση. Το hierarchical schedule από την άλλη παρουσιάζει δύο διαφορετικές συμπεριφορές, ανάλογα με το μέγεθος των groups. Για μεγέθη στα οποία τα groups χωράνε εξ ολοκλήρου στους NUMA κόμβους (1, 2, 4, 8) βλέπουμε ότι έχει σχεδόν την ίδια επίδοση με το static, παρόλο το επιπλέον overhead που προσθέτει. Αντίθετα, όταν τα groups εκτείνονται σε παραπάνω από έναν κόμβο (16, 32) βλέπουμε κακή επίδοση, αντίστοιχη του dynamic. Μάλιστα, για μέγεθος group ίσο με 32 βλέπουμε ακριβώς ίδια συμπεριφορά με το dynamic, όπως βέβαια και περιμέναμε (γενικά, αυτό θα το παρατηρούμε σε όλες τις μετρήσεις).

Η συμπεριφορά του hierarchical μας κάνει να υποψιαζόμαστε την ύπαρξη φαινομένων NUMA. Για να δούμε σε περισσότερο βάθος τη λειτουργία των schedules, πήραμε επιπλέον μετρήσεις μέσω των performance counters του επεξεργαστή, χρησιμοποιώντας τη βιβλιοθήκη "PAPI". Για τις μετρήσεις στο PAPI επιλέχθηκε το μεγάλο μέγεθος πίνακα, δηλαδή $N = 2 * 10^9$.

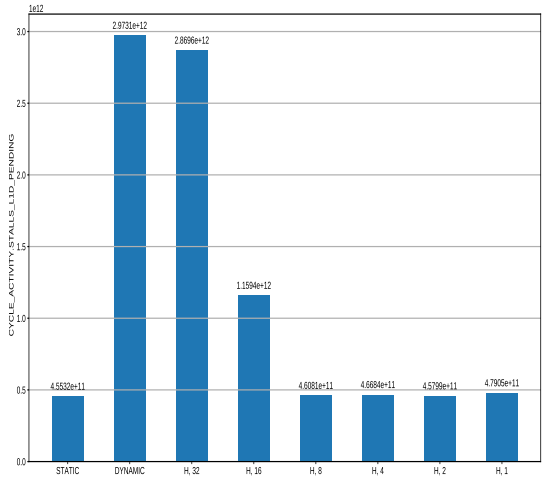
Πράγματι, όπως βλέπουμε στο σχήμα 4.2, τα cache misses όλων των schedules είναι λίγο πολύ ίδια. Τα memory accesses είναι επίσης ίδια, όπως και περιμέναμε, εφόσον το stream είναι balanced και τα threads θα επεξεργάζονται το ίδιο πλήθος δεδομένων σε όλα τα schedules (τα L1 cache accesses δεν μπορούν να μετρηθούν στο συγκεκριμένο επεξεργαστή). Συμπεραίνουμε ότι το grain size είναι αρκετά μεγάλο



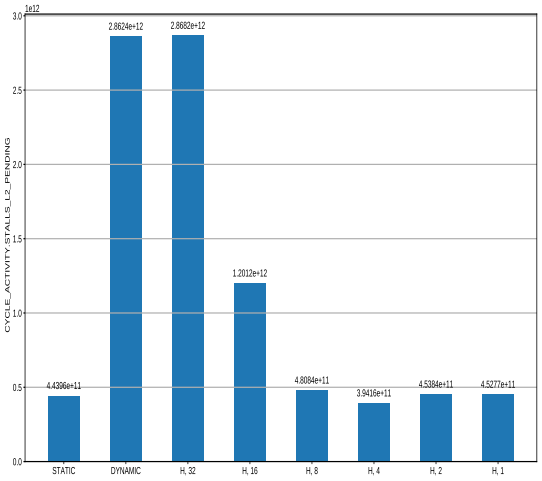
(α′) L2 cache accesses (β′) L3 cache accesses

(γ′) L1 cache misses (δ′) L2 cache misses (ε′) L3 cache misses

Σχήμα 4.2: Stream, $N = 2 * 10^9$. Πλήθη των cache accesses και cache misses από το σύνολο των kernel του stream για τα διάφορα επίπεδα της cache. Τα δεδομένα αρχικοποιούνται με static schedule.



(α′) L1D stalls



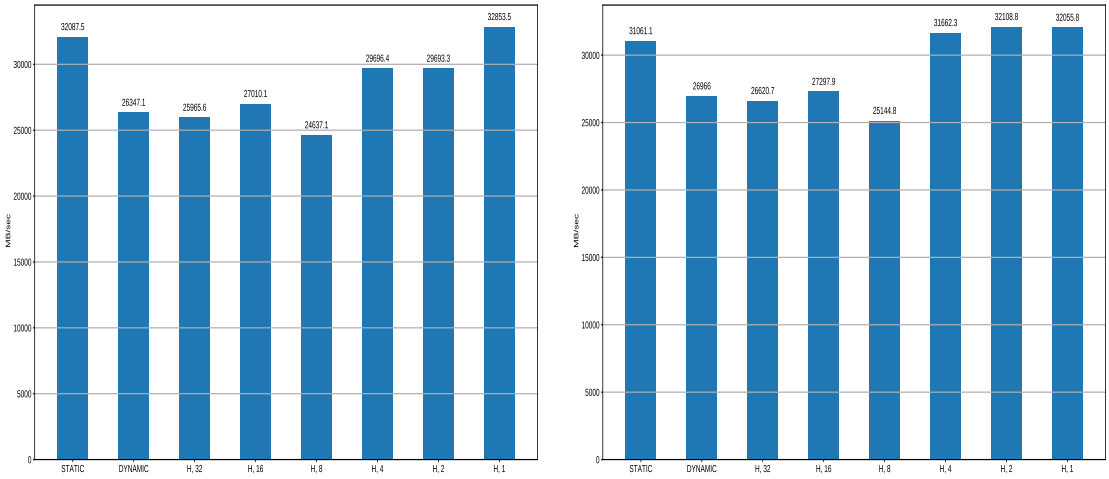
(β′) L2 stalls

Σχήμα 4.3: Stream, $N = 2 * 10^9$. Συνολικό πλήθος των κύκλων που τα thread σπαταλούν περιμένοντας να έρθουν τα δεδομένα που χρειάζονται από το επόμενο επίπεδο της μνήμης. Τα δεδομένα αρχικοποιούνται με static schedule.

ώστε να μην επηρεάζει το ένα thread τη λειτουργία του άλλου, και να αποφεύγονται φαινόμενα όπως false sharing. Διαφορετικά θα βλέπαμε πολύ υψηλά μεγέθη των misses στα δυναμικά schedule, ως συνέπεια αυτών που περιγράψαμε στα κεφάλαια 2.1 και 2.2. Αντίθετα, τα stalls των επεξεργαστών που παρουσιάζονται στο σχήμα 4.3 έρχονται σε απόλυτη συμφωνία με τα αντίστοιχα bandwidth των schedules (τα L3 stalls ομοίως δεν μπορούσαν να μετρηθούν στο μηχάνημα).

Παρότι δηλαδή έχουμε ίδιο αριθμό από memory accesses και cache misses, ο χρόνος που σπαταλούν τα threads στη μνήμη σε κάθε περίπτωση είναι διαφορετικός. Αυτό οφείλεται κατά ένα μέρος στο ότι παρουσιάζονται φαινόμενα NUMA. Στο dynamic και το hierarchical schedule για group που εκτείνονται σε παραπάνω από ένα NUMA κόμβο τα threads προσπελαύνουν συχνά μνήμη η οποία ανήκει σε διαφορετικούς κόμβους, οπότε και οι επεξεργαστές χάνουν περισσότερο χρόνο περιμένοντας τα δεδομένα αυτά. Όπως όμως θα δούμε και στη συνέχεια, δεν είναι μόνο αυτός ο παράγοντας μείωσης της απόδοσης. Επιπρόσθετα, στα schedules αυτά παρουσιάζεται και το φαινόμενο που περιγράψαμε στην ενότητα 2.2.2. Με τη συγκεκριμένη τοποθέτηση των δεδομένων, όλα τα threads στο dynamic πέφτουν κάθε φορά σε ένα μόνο NUMA κόμβο, μειώνοντας δραματικά το διαθέσιμο bandwidth. Ομοίως στο hierarchical με μέγεθος group 32, και σε λιγότερο βαθμό με μέγεθος 16.

Αντίθετα, όταν στο hierarchical το μέγεθος του group περιορίζεται σε ένα μόνο NUMA κόμβο, τα threads βρίσκουν τα δεδομένα τους στον κόμβο που βρίσκονται, οπότε και βλέπουμε ίδιες συμπεριφορές με το static στα bandwidths και στα stalls.



(α′) N = 10^7 (β′) N = $2 * 10^9$

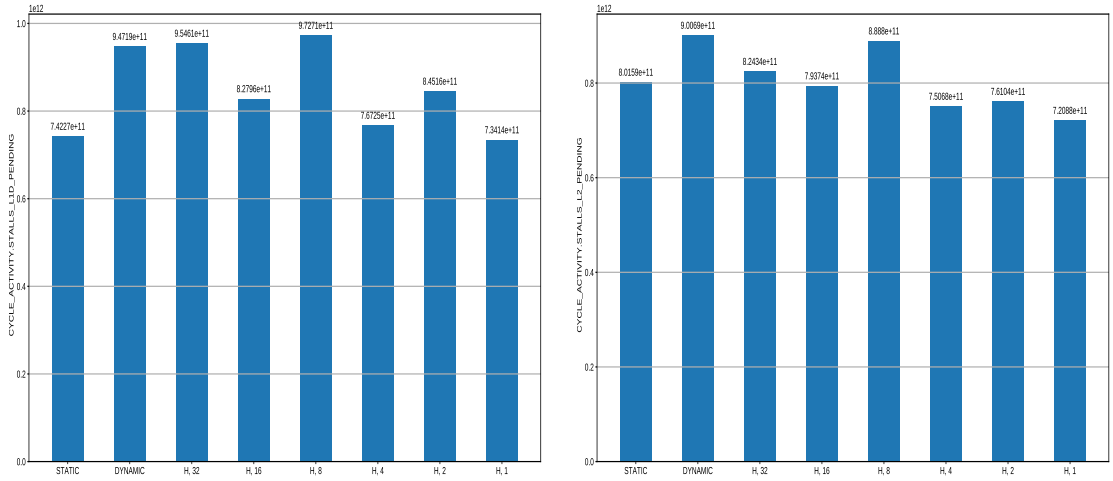
Σχήμα 4.4: Stream. Bandwidth διαγράμματα σε MB / sec του Add kernel του stream, με grain size = 1024 και "N" μέγεθος πινάκων. Τα δεδομένα μοιράζονται με round robin τρόπό, μέσω της δεύτερης μορφής του static schedule.

Αρχικοποίηση με τη δεύτερη μορφή του static

Θα θέλαμε επίσης να δούμε μια πιο τυχαία κατανομή των δεδομένων, ώστε να δείξουμε τη σημασία του αρχικού διαμοιρασμού των δεδομένων σε NUMA μηχανήματα, αλλά και για να βεβαιωθούμε για τους ισχυρισμούς μας ως προς τη συμπεριφορά του dynamic. Σε αυτήν την περίπτωση λοιπόν, οι πίνακες αρχικοποιούνται με τη δεύτερη μορφή του static schedule, με grain size = 1024. Έτσι, τα δεδομένα τοποθετούνται με round robin τρόπο στους NUMA κόμβους, ώστε να προσομοιώνεται μία τυχαία κατανομή τους.

Στο σχήμα 4.4 βλέπουμε ομοίως διαγράμματα του bandwidth με αυτή την αρχικοποίηση. Αρχικά, βλέπουμε ότι η επίδοση του static πέφτει σχεδόν στο μισό. Αυτό είναι φυσιολογικό, εφόσον τώρα τα threads βρίσκουν τα δεδομένα τους σκορπισμένα σε όλους τους NUMA κόμβους, σε αντίθεση με πριν, όπου τα έβρισκαν στον δικό τους μόνο κόμβο.

Βλέπουμε επίσης ότι πράγματι, το dynamic έχει καλύτερη επίδοση από πριν, αν και χειρότερη από το static. Με την προηγούμενη κατανομή, τα threads έπεφταν όλα μαζί σε έναν κόμβο τη φορά. Τώρα, τα δεδομένα τοποθετούνται ανακατεμένα στους NUMA κόμβους, οπότε οι προσβάσεις στη μνήμη μοιράζονται χρονικά, και αξιοποιείται έτσι κάθε στιγμή το bandwidth όλων των κόμβων. Ουσιαστικά, έχουμε καλύτερη παραλληλοποίηση της πρόσβασης στη μνήμη. Αυτό φαίνεται και στο σχήμα 4.5, όπου παρουσιάζονται οι κύκλοι που χάνονται σε memory stalls. Τα ίδια ισχύουν και για το hierarchical, όταν έχει μέγεθος group μεγαλύτερο από τους πυρήνες ενός



(α′) L1D stalls (β′) L2 stalls

Σχήμα 4.5: Stream, $N = 2 * 10^9$. Συνολικό πλήθος κύκλων των memory stalls. Τα δεδομένα αρχικοποιούνται με τη δεύτερη μορφή του static schedule, με grain size = 1024.

κόμβου (εδώ 16 και 32).

Αντίθετα, το ανακάτεμα των δεδομένων μειώνει την επίδοση του hierarchical με μεγέθη group 1, 2, 4 και 8, για τον ίδιο λόγο που μειώνεται και η επίδοση του static. Πράγματι, οι περιπτώσεις αυτές παρουσιάζουν σχεδόν το μισό bandwidth από πριν, και αντίστοιχα σχεδόν τα διπλάσια stalls.

Από τις μετρήσεις στο stream λοιπόν βλέπουμε ότι, σε αντίθεση με το dynamic schedule, το hierarchical έχει την ευελιξία να προσαρμοστεί στην τοπολογία του μηχανήματος. Μοιράζοντας τις επαναλήψεις σε group από threads και σε συνεχή κομμάτια, μας δίνει τη δυνατότητα να απομονώσουμε τα πεδία εργασίας μεταξύ threads διαφορετικών κόμβων, ώστε να μπορούμε να εκμεταλλευτούμε τεχνικές τοποθέτησης των δεδομένων που αντιμετωπίζουν τα φαινόμενα NUMA. Καταφέρνει μάλιστα να φτάσει τις επιδόσεις του static schedule σε ένα φορτίο εργασίας το οποίο είναι απολύτως balanced. Αντίθετα, το dynamic μοιράζει ουσιαστικά τυχαία τις επαναλήψεις στα threads, ώστε το καλύτερο που μπορούμε να κάνουμε να είναι απλά μία interleaved τοποθέτηση των δεδομένων στους NUMA κόμβους, ώστε να μοιράσουμε τις προσβάσεις μνήμης σε κάθε χρονική στιγμή σε όλους τους κόμβους.

4.2 Συνθετικό Benchmark

Σε αυτή την ενότητα παρουσιάζεται ένα συνθετικό memory bound benchmark. Εδώ θα δούμε κυρίως ένα άλλο πρόβλημα που μπορεί να παρουσιαστεί σε τέτοιου είδους προγράμματα, την περίπτωση δηλαδή που το grain size χωρίζει τα δεδομένα σε κομμάτια συγκρίσιμου μεγέθους με αυτού των cache line. Στο συγκεκριμένο επεξεργαστή το cache line έχει το συνηθισμένο μέγεθος των 64 byte, ή αλλιώς 8 double αριθμών. Σε αυτή την περίπτωση, όπως εξηγήσαμε και στην ενότητα 2.2.1, τα threads στο dynamic επηρεάζουν το ένα την εκτέλεση του άλλου, μειώνοντας δραματικά την επίδοση. Επιπλέον, το πρόγραμμα αυτό κατανέμει άνισα την εργασία στα threads, οπότε θα δούμε πως το hierarchical συμπεριφέρεται στο load balancing, ακόμα και σε συνθήκες με πολύ μικρά grain size.

Το benchmark αυτό είναι εμπνευσμένο από blocking αλγορίθμους, οι οποίοι χωρίζουν τα δεδομένα τους σε μικρά blocks τα οποία χωράνε στην L1 cache, ώστε να επιτύχουν λιγότερα misses. Σε αυτό το πρόγραμμα λοιπόν, ένας 2-D πίνακας από τύπου double αριθμούς χωρίζεται σε τετράγωνα blocks μεγέθους 4×4 (δοκιμάστηκαν και άλλα μεγέθη block), τα οποία μοιράζονται στα threads σύμφωνα με το scheduling. Σημειώνεται ότι ο αρχικός διαμοιρασμός έγινε με static, όπως και θα γίνεται γενικά και στα υπόλοιπα benchmark. Ο round robin (ή αλλιώς interleaved) διαμοιρασμός δεν προσφέρει κανέναν ουσιώδη έλεγχο στην τοποθέτηση δεδομένων. Χρησιμοποιείται σε αλγορίθμους που δεν είναι NUMA aware για να εξομαλύνει πιθανές 'κακές' κατανομές δεδομένων στους NUMA κόμβους, εισάγοντας ουσιαστικά τυχαιότητα στην κατανομή. Αυτές οι περιπτώσεις δεν μας ενδιαφέρουν στο πλαίσιο του hierarchical schedule, συνεπώς και δεν θα μελετηθούν πολύ περαιτέρω.

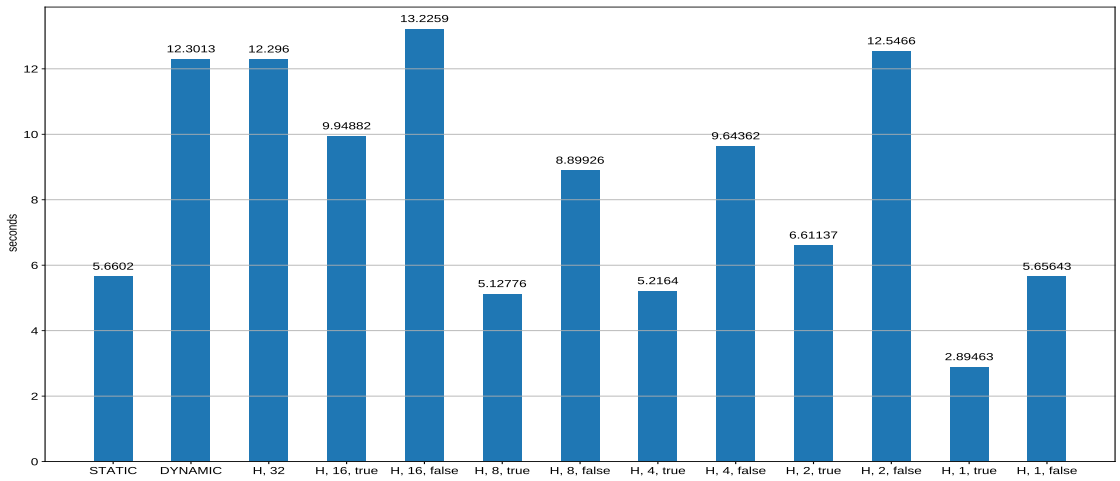
Τα threads επεξεργάζονται τα blocks ατομικά, δηλαδή ένα block θα δεχθεί επεξεργασία από ένα μόνο thread. Αυτή είναι η εξής:

1. Θεωρώντας ότι το στοιχείο (0, 0) βρίσκεται πάνω αριστερά στον δισδιάστατο πίνακα και ότι συνολικά δημιουργούνται "B" blocks, υπολογίζουμε τη θέση "pos" του block, από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, και βρίσκουμε τον αριθμό $W = pos * B/100$.

2. Σε κάθε στοιχείο του block προστίθενται οι τιμές των άμεσα γειτονικών του στοιχείων, δηλαδή των πάνω, κάτω, δεξιά και αριστερά στοιχείων.

3. Επαναλαμβάνουμε το προηγούμενο βήμα άλλες W - 1 φορές.

Εδώ εντοπίζεται ένα racing condition αλλά, εφόσον το πρόγραμμα είναι ένα απλό benchmark, δεν μας ενδιαφέρει το αποτέλεσμα ούτε να είναι 'ορθό' (τι σημαίνει ορθό σε αυτό το πλαίσιο;), ούτε προβλέψιμο (ο χρόνος του προγράμματος δεν εξαρτάται από τις τιμές των στοιχείων). Επιπλέον, το τελευταίο βήμα μας δίνει την ανισοκατανομή της εργασίας που θέλουμε. Αν παρατηρήσουμε, η εργασία αυξάνεται γραμμικά με τη



Σχήμα 4.6: Synthetic Benchmark, N = 25000, block = 4 × 4, grain size = 8. Παρουσιάζονται οι χρόνοι εκτέλεσης του προγράμματος. Για το hierarchical σημειώνεται επίσης το 'group size' και η τιμή του 'gomp_hierarchical_stealing' (stealing μεταξύ των thread groups).

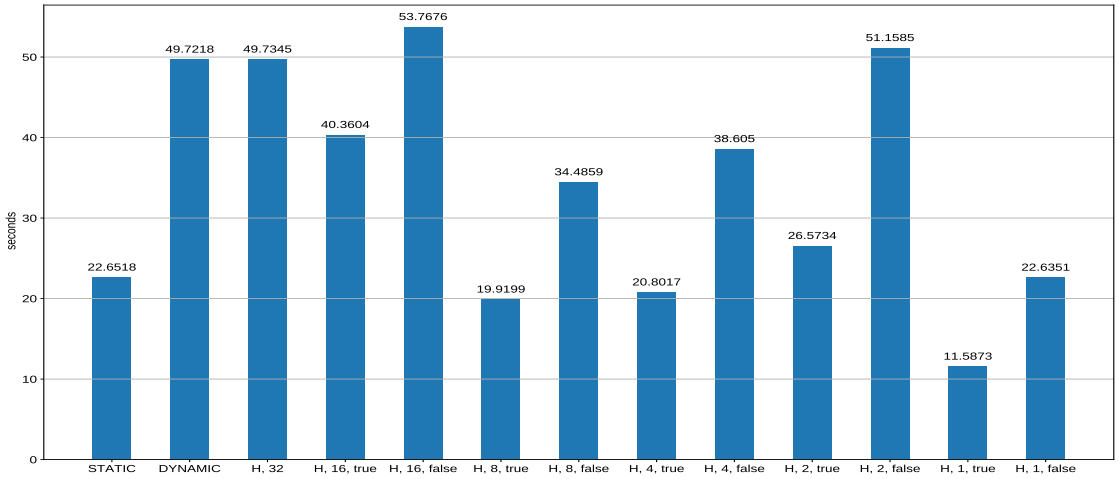
θέση του block, με χβαντισμένο βέβαια τρόπο (σε 100 βήματα), οπότε ένας τέλειος διαμοιρασμός της θα πρέπει να έχει θεωρητικά περίπου τη διπλάσια επίδοση από το στατικό και συνεχόμενο διαμοιρασμό του static schedule.

Παρουσιάζονται δύο μεγέθη πινάκων, N=25000 (5GB) και για N=50000 (20GB), όπου N είναι το πλήθος στοιχείων (double αριθμών) της κάθε πλευράς του τετράγωνου πίνακα. Δοκιμάζονται επίσης διάφοροι συνδυασμοί των scheduling, grain size, group size και stealing μεταξύ των thread groups. Στις μετρήσεις του hierarchical schedule σημειώνεται το 'group size' και η τιμή του hierarchical stealing, δηλαδή αν είναι ενεργοποιημένη η κλοπή εργασίας μεταξύ των thread groups (true / false). Επομένως, για group size = 1 και hierarchical stealing = false, το hierarchical schedule ουσιαστικά γίνεται static (συν το επιπλέον overhead των ατομικών αθροίσεων κατά grain size).

grain size 8

Στα σχήματα 4.6 και 4.7 βλέπουμε τους χρόνους εκτέλεσης για grain size = 8 στα δύο διαφορετικά μεγέθη πινάκων. Εδώ εξετάζεται η πρώτη περίπτωση, όπου το grain size δημιουργεί πολύ μικρά κομμάτια εργασίας, ώστε τα threads να επηρεάζουν το ένα το άλλο. Συγκεκριμένα, η εργασία εκτελείται κατά grain size το πλήθος blocks, δηλαδή ορθογώνια μεγέθους 4 × 32 double αριθμών.

Στο dynamic και στο hierarchical schedule με group size > 1, συχνά δύο γειτονικά τέτοια ορθογώνια θα εκτελούνται από διαφορετικά threads. Επίσης, θυμίζουμε ότι στην επεξεργασία κάθε στοιχείου εμπλέκονται και οι γείτονές του. Επειδή λοιπόν

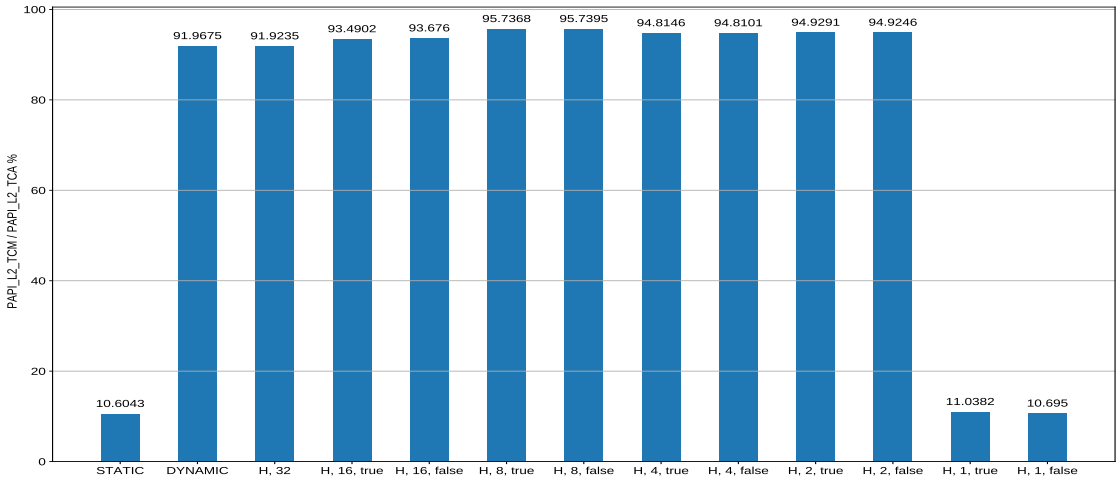


Σχήμα 4.7: Synthetic Benchmark, N = 50000, block = 4 × 4, grain size = 8.

τα πεδία αυτά των δεδομένων είναι πολύ μικρά, διαφορετικά threads θα τυχαίνει συχνά να προσπελαύνουν δεδομένα είτε κοινά, είτε αρκετά κοντινά ώστε να βρίσκονται στο ίδιο cache line.

Στο hierarchical με group size = 1 αντίθετα, όπου κάθε group αποτελείται από ένα μόνο thread, η ανακατανομή της εργασίας γίνεται μόνο με hierarchical stealing μεταξύ των groups. Αυτό έχει ως αποτέλεσμα κάθε thread να δουλεύει σε απομακρυσμένα πεδία δεδομένων από τα υπόλοιπα, οπότε και βλέπουμε την μεγάλη επίδοση σε αυτήν την περίπτωση. Συγκεκριμένα, για N = 25000 και N = 50000, βλέπουμε ότι ο χρόνος του hierarchical σε σχέση με του static είναι 2.89463/5.6602 = 51.14% και 11.5873/22.6518 = 51.15% αντίστοιχα, το οποίο είναι πολύ κοντά στο θεωρητικό μισό χρόνο που αναφέραμε πιο πάνω. Αυτό υποστηρίζει την υπόθεση που κάναμε ότι το overhead που εισάγει το hierarchical stealing, αλλά και hierarchical schedule γενικότερα, είναι σχετικά μικρό.

Έχοντας υπόψιν τα προηγούμενα, πήραμε μετρήσεις των cache misses, οι οποίες και παρουσιάζονται στο σχήμα 4.9. Όλες οι μετρήσεις των performance counter με τη βιβλιοθήκη PAPI έγιναν για το μέγεθος N = 50000. Πράγματι, στα διαγράμματα των L1 και L2 misses βλέπουμε πολύ μεγάλες αποκλίσεις από το static, πέρα της περίπτωσης με group size = 1. Αυτό είναι αποτέλεσμα του false sharing φαινομένου που περιγράψαμε. Ο ισχυρισμός αυτός υποστηρίζεται και από το γεγονός ότι σχεδόν όλα τα L1 cache misses μετατρέπονται και σε L2 cache misses. Αφού τα L1 misses πάντα αναγκαστικά γίνονται και L2 accesses, θα πρέπει να έχουμε πολύ υψηλά ποσοστά των L2 cache miss ratio, πράγμα το οποίο και αποδεικνύεται από τις μετρήσεις του σχήματος 4.8. Το μεγαλύτερο δηλαδή ποσοστό των προσπελάσεων στη private cache των threads ουσιαστικά ακυρώνεται από προσπελάσεις της μνήμης από άλλα threads. Αντίθετα, για group size = 1, βλέπουμε μετρήσεις σχεδόν ίδιες με αυτές

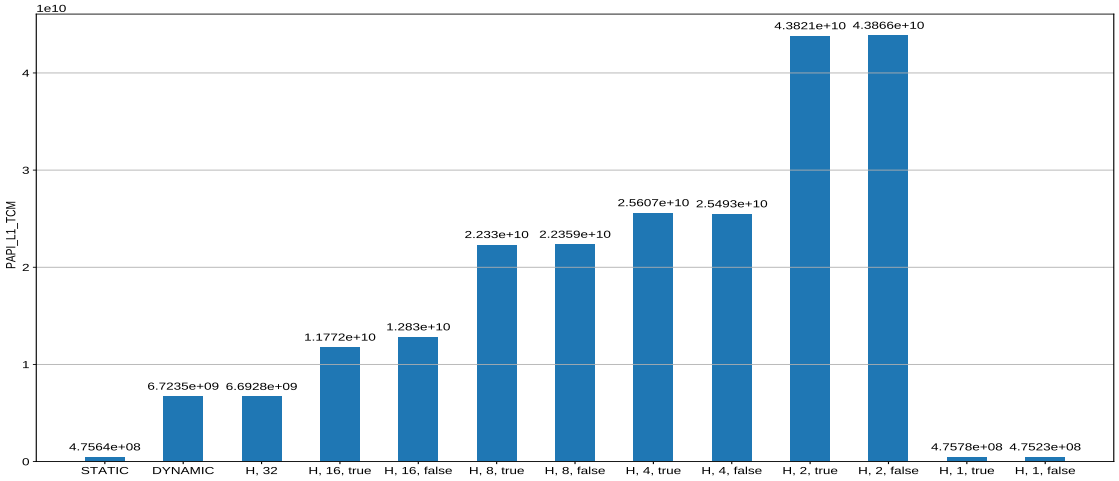


Σχήμα 4.8: Synthetic Benchmark, N = 50000, block = 4×4, grain size = 8. Μετρήσεις του L2 cache miss ratio.

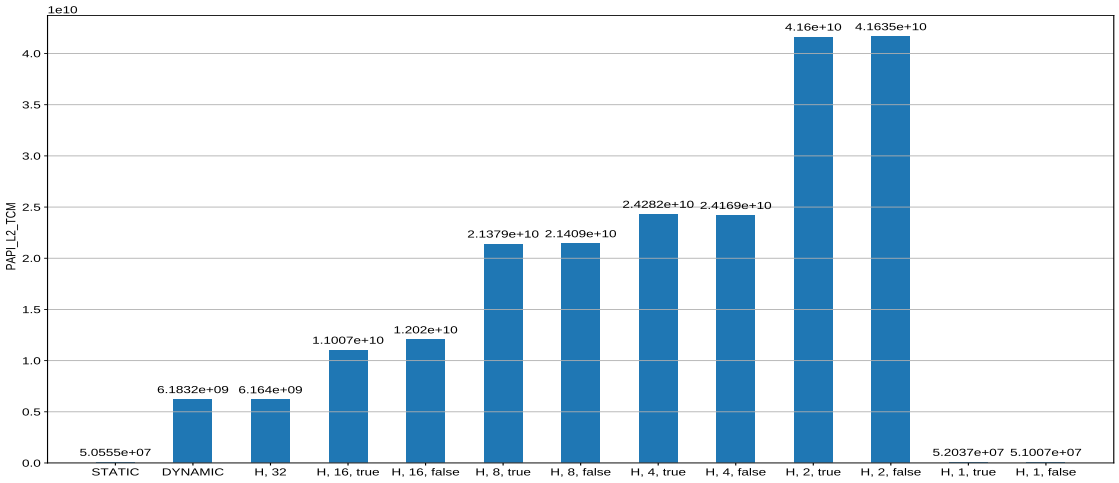
του static. Σε αυτήν την περίπτωση, τα threads δουλεύουν σε απομονωμένα πεδία και δεν δημιουργείται συμφόρηση.

Η άλλη παρατήρηση είναι στα misses της L3 cache. Βλέπουμε μια ακραία μείωση όταν τα groups στο hierarchical περιορίζονται στο μέγεθος του NUMA κόμβου, δηλαδή από 8 και κάτω. Το φαινόμενο αυτό είναι αντίστοιχο με αυτό που παρουσιάσαμε στο stream. Με τον περιορισμό των threads κάθε group σε ένα NUMA κόμβο μειώνεται δραματικά η κίνηση δεδομένων μεταξύ των κόμβων. Ακόμα και αν στα threads ενός group το ένα ακυρώνει την private cache του άλλου, όπως και γίνεται εδώ για group size > 1, τα δεδομένα πάντα θα βρίσκονται στην τοπική L3 cache του κόμβου που βρίσκεται το group, Τα πεδία δεδομένων των groups, λόγω του αρχικού διαμοιρασμού και του τρόπου που δουλεύει το hierarchical stealing, θα είναι αρκετά μεγάλα ώστε τα threads διαφορετικών group να εργάζονται πάντα σε απομακρυσμένα δεδομένα. Αυτός είναι και ο λόγος που βλέπουμε καλύτερη επίδοση ακόμα και στα group size 2, 4 και 8 (με ενεργοποιημένο φυσικά το group stealing), όπου έχουμε κακή επίδοση των L1 και L2 caches.

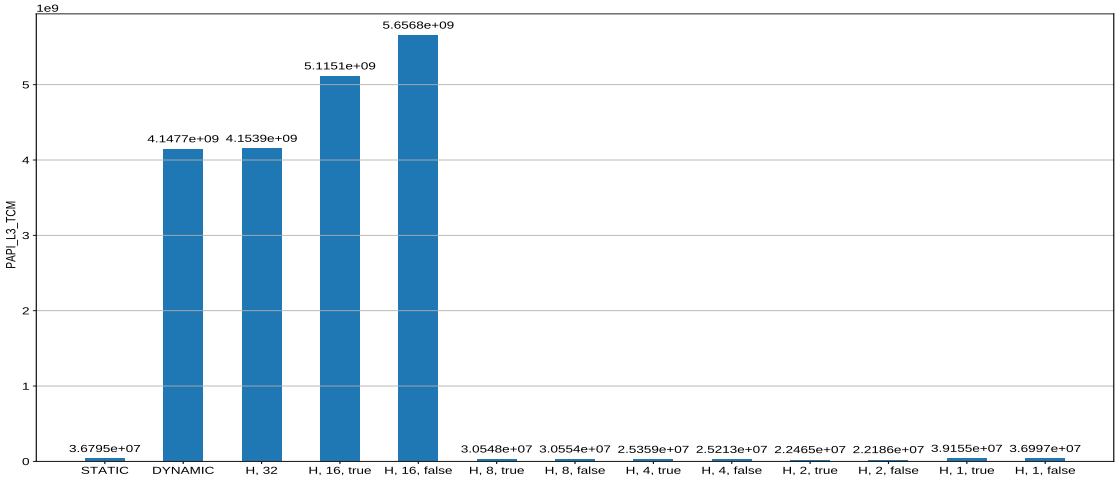
Ένα άλλο ενδιαφέρον φαινόμενο είναι η αύξηση που παρατηρείται στα L1 και L2 cache misses καθώς το group size μειώνεται από 32 μέχρι 2. Ειδικά για group size = 2, τα misses σχεδόν διπλασιάζονται από την περίπτωση με group size = 4. Δεν μπορεί να ευθύνεται το hierarchical stealing, διότι έχουμε τον ίδιο αριθμό από misses ανεξάρτητα του αν είναι ενεργοποιημένο ή όχι. Στο σχήμα 4.10 παρουσιάζονται τα branch mispredictions για κάθε schedule, τα οποία φαίνεται να ακολουθούν το ίδιο μοτίβο. Αυτό θα μπορούσε να εξηγηθεί ως εξής. Στην πλειονότητά τους, γειτονικά blocks έχουν τον ίδιο φόρτο εργασίας, αφού αυτός αλλάζει μόνο 100 φορές σε όλο το πεδίο των επαναλήψεων (βήμα 1 στην επεξεργασία των blocks). Συνεπώς, τα threads



(α′) L1 cache misses



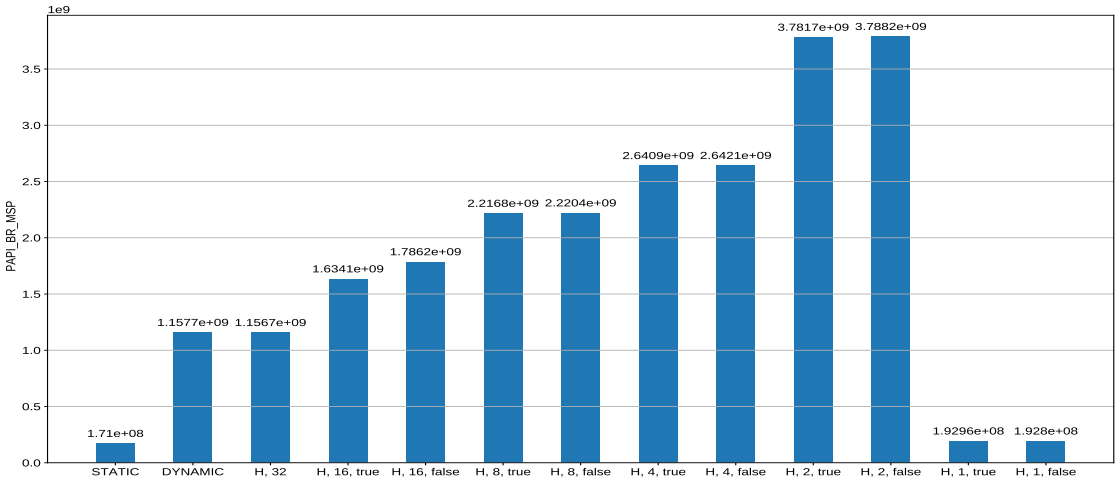
(β′) L2 cache misses



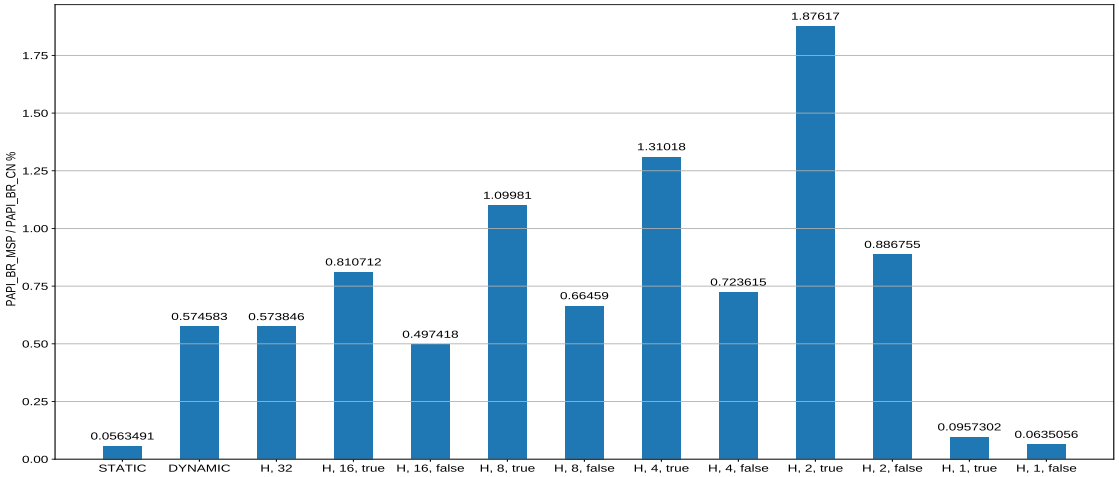
(γ′) L3 cache misses

Σχήμα 4.9: Synthetic Benchmark, N = 50000, block = 4 × 4, grain size = 8. Τιμές των cache misses για τα διάφορα επίπεδα της cache.

ενός group θα χρειάζονται συνήθως τον ίδιο χρόνο για να επεξεργαστούν ένα block. Όταν τα threads ανά group είναι πολλά, η περίπτωση ένα thread να προλάβει να επεξεργαστεί δύο γειτονικά blocks είναι πολύ μικρή, αφού θα έχουν προλάβει τα υπόλοιπα να πάρουν το γειτονικό block. Στην περίπτωση όμως των λίγων threads ανά group, και κυρίως των 2, ή πιθανότητα δεν είναι τόσο μικρή. Συνεπώς, θα είναι δυσκολότερη η σωστή πρόβλεψη από τους branch predictors για το αν θα πρέπει να περάσει η εκτέλεση και στο επόμενο block. Αυτό θα είχε ως αποτέλεσμα λανθασμένες επιλογές, οι οποίες θα μπορούσαν να φέρνουν στις cache λάθος δεδομένα, που αμέσως μετά θα πετάγονταν. Ως αποτέλεσμα αυτού του φαινομένου θα παρουσιάζονταν αυξημένα L1 cache accesses, δυστυχώς όμως η μέτρηση αυτών δεν είναι δυνατή στο συγκεκριμένο μηχάνημα για να το επιβεβαιώσουμε, αν και στο L2 cache miss ratio βλέπουμε σχεδόν ίδια ποσοστά για όλα αυτά τα schedules. Μπορούμε όμως μάλλον να πούμε με σχετική βεβαιότητα ότι το φαινόμενο αυτό είναι ιδιαιτερότητα του συγκεκριμένου benchmark.

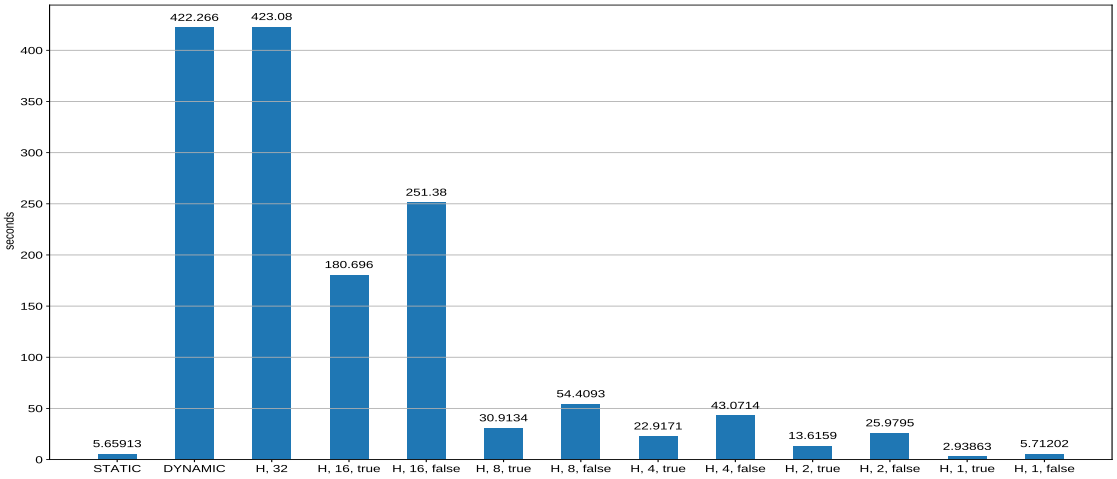


(α′) Branch mispredictions



(β′) Branch misprediction ratios

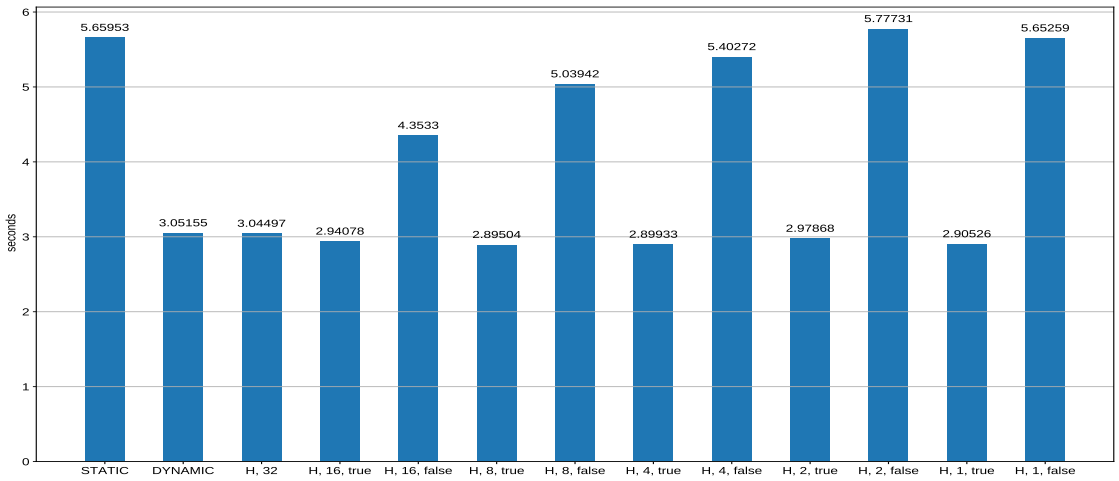
Σχήμα 4.10: Synthetic Benchmark, N = 50000, block = 4×4, grain size = 8. Μετρήσεις των branch prediction για τα διάφορα schedule.



Σχήμα 4.11: Synthetic Benchmark, N = 25000, block = 4×4, grain size = 1.

grain size 1

Για πληρότητα, στο σχήμα 4.11 παρουσιάζεται η ακραία περίπτωση, όπου grain size = 1. Όπως φαίνεται, το hierarchical schedule με μέγεθος group ίσο με 1 ουσιαστικά δεν επηρεάζεται, όσο μικρό και να είναι το grain size, ώστε ακόμα και εδώ ο λόγος προς το χρόνο του static να είναι 2.9386/5.659 = 51.9%. Αυτό το φαινόμενο έχει να κάνει με το ότι, σε αυτή τη μορφή του hierarchical, όλα τα threads δουλεύουν σε απομακρυσμένα μεταξύ τους πεδία δεδομένων καθ' όλη τη διάρκεια της εκτέλεσης, χάρη στον τρόπο που μοιράζεται η εργασία και τα δεδομένα στην αρχή του προγράμματος, αλλά και χάρη στην κλοπή επαναλήψεων σε λίγα και μεγάλα κομμάτια μέσω του hierarchical stealing. Παράλληλα με αυτό, στο συγκεκριμένο πρόγραμμα ο μέσος φόρτος εργασίας των επαναλήψεων είναι αρκετά μεγάλος, ώστε το overhead της επιπλέον εργασίας και συγχρονισμού που επιφέρουν τα dynamic και hierarchical schedules να είναι ουσιαστικά αμελητέο. Με το συνδυασμό αυτών των δύο στοιχείων, προκύπτει η υψηλή επίδοση που βλέπουμε.



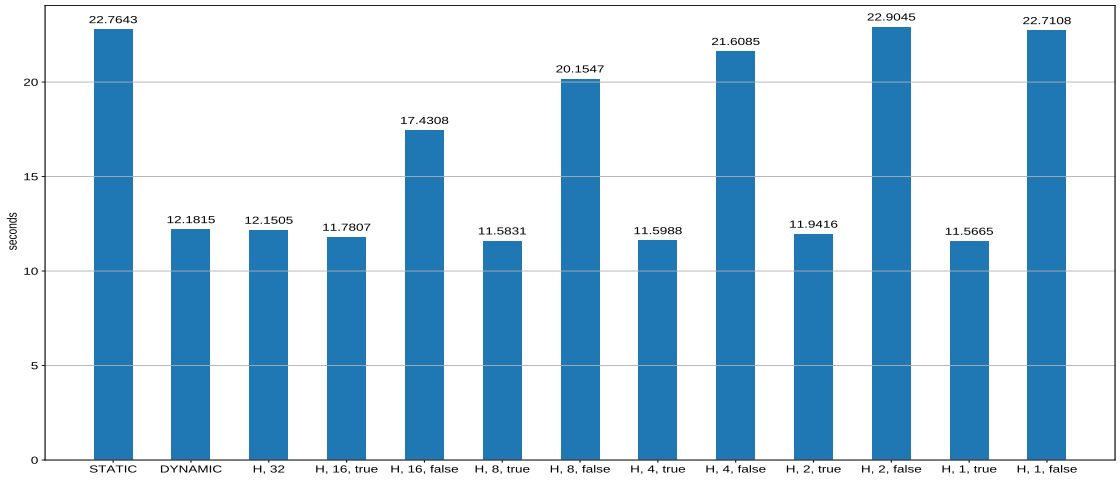
Σχήμα 4.12: Synthetic Benchmark, N = 25000, block = 4×4, grain size = 64. Το grain size είναι αρκετά μεγάλο ώστε στο dynamic scheduling να μην συγκρούονται τα threads.

grain size 64

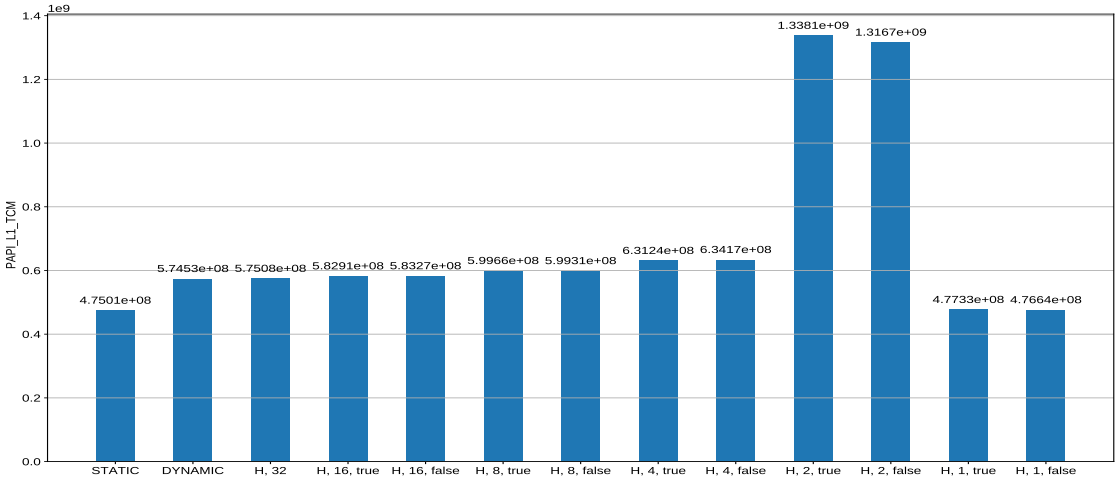
Η περίπτωση αυτή προσφέρεται ως αντίθεση με τις προηγούμενες περιπτώσεις. Στα σχήματα 4.12 και 4.13 παρουσιάζονται οι χρόνοι εκτέλεσης των διαφόρων schedule. Όπως βλέπουμε, το dynamic και όλα τα hierarchical με ενεργοποιημένο το hierarchical stealing πετυχαίνουν σχεδόν τη θεωρητικά βέλτιστη επίδοση.

Στο σχήμα 4.14, όπου παρουσιάζονται τα cache misses, μπορούμε να διακρίνουμε το λόγο αυτής της συμπεριφοράς. Όλα τα δυναμικά schedule παρουσιάζουν πολύ λιγότερα cache misses. Πέρα της περίεργης συμπεριφοράς του hierarchical με μέγεθος group ίσο με 2, όπου και πάλι αν προσέξουμε είναι σχεδόν δύο τάξεις μεγέθους λιγότερα από πριν, όλα τα misses είναι πια συγκρίσιμα με αυτά του static. Στα L3 misses παρατηρούμε μία αύξηση στα dynamic και hierarchical με group πάνω των 8 threads, όπως βέβαια και περιμέναμε να δούμε.

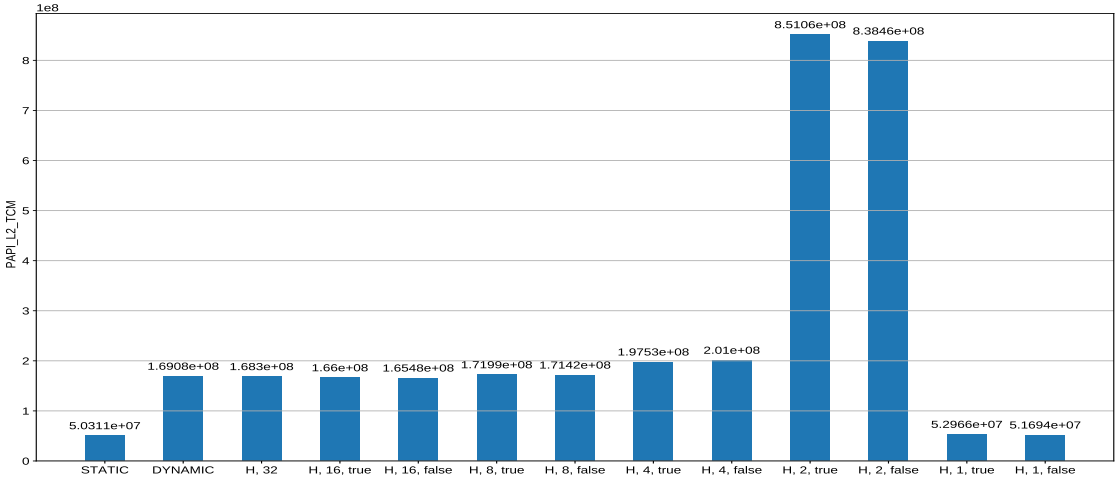
Με grain size ίσο με 64, κάθε κομμάτι εργασίας θα εκτείνεται σε επίσης 64 blocks του πίνακα, δηλαδή ορθογώνια μεγέθους 4×256 double αριθμών, ή αλλιώς $4 \times 2048 Byte$. Τα 2048 byte που απέχουν μεταξύ τους είναι αρκετά περισσότερα από τα 64 byte που είναι το μέγεθος των cache line, ώστε όπως φαίνεται και από τα misses, τα threads σπάνια να τυχαίνει να εργάζονται σε δεδομένα κοινών cache line. Όσον αφορά την περίεργη συμπεριφορά του hierarchical για 2 threads, βλέποντας και την ίδια συμπεριφορά στο σχήμα 4.15 των branch mispredictions, πιθανολογούμε ότι έχουμε τα ίδια φαινόμενα με αυτά που περιγράψαμε στο τέλος της παραγράφου για το grain size = 8.



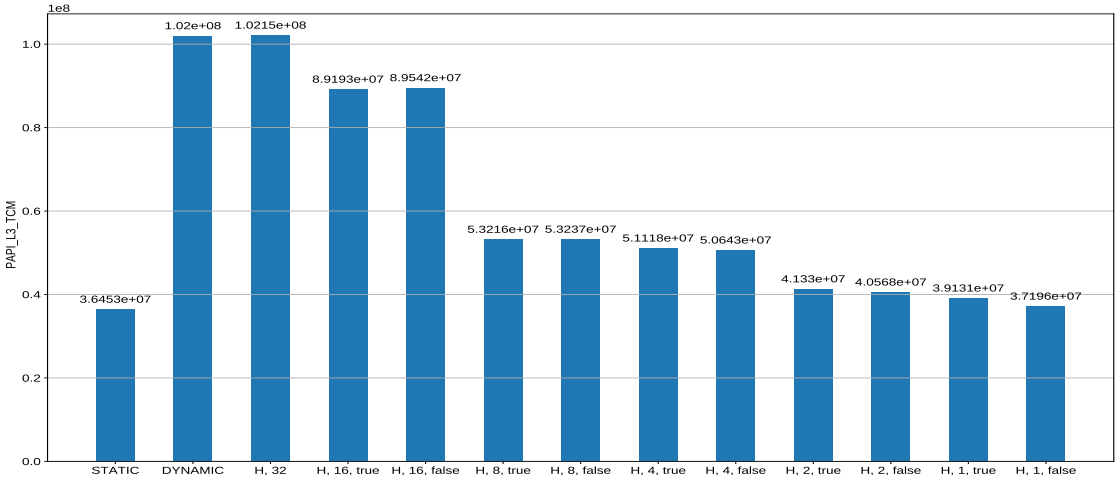
Σχήμα 4.13: Synthetic Benchmark, N = 50000, block = 4×4, grain size = 64. Αντίστοιχη συμπεριφορά για μεγαλύτερο μέγεθος πίνακα.



(α′) L1 cache misses

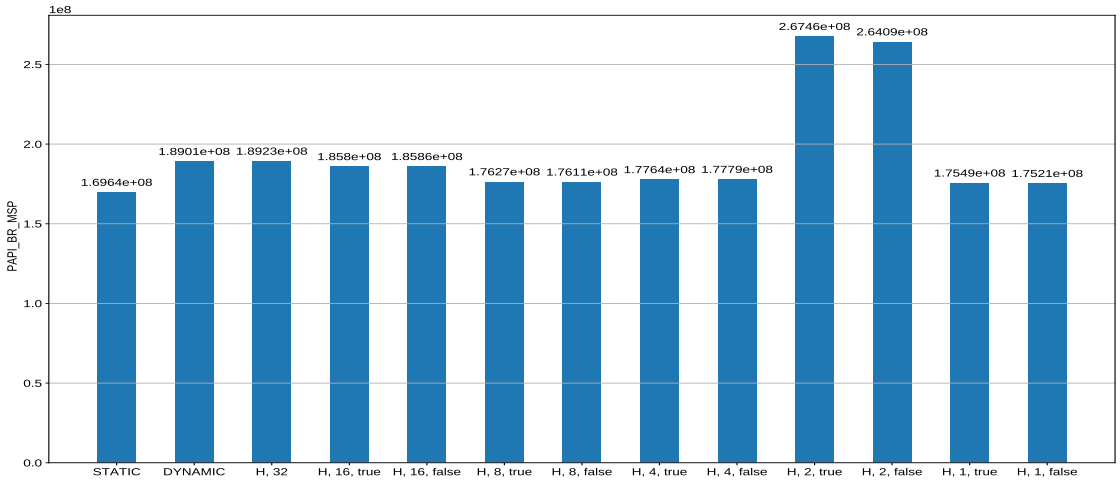


(β′) L2 cache misses

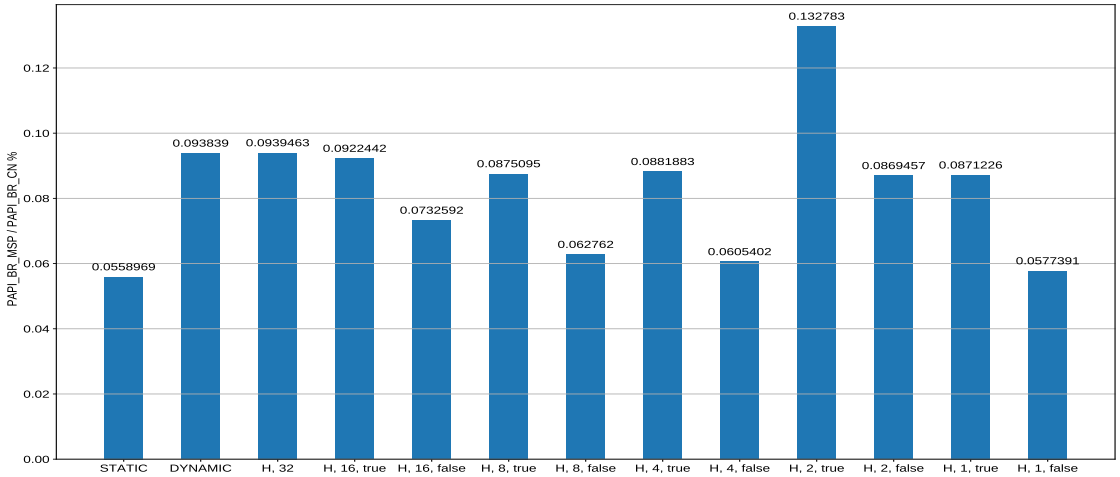


(γ′) L3 cache misses

Σχήμα 4.14: Synthetic Benchmark, N = 50000, block = 4×4, grain size=64. Τιμές των cache misses για τα διάφορα επίπεδα της cache.



(α′) Branch mispredictions



(β′) Branch misprediction ratios

Σχήμα 4.15: Synthetic Benchmark, N = 50000, block = 4×4, grain size = 64. Μετρήσεις των branch prediction για τα διάφορα schedules.

Δοκιμές με διαφορετικά block sizes

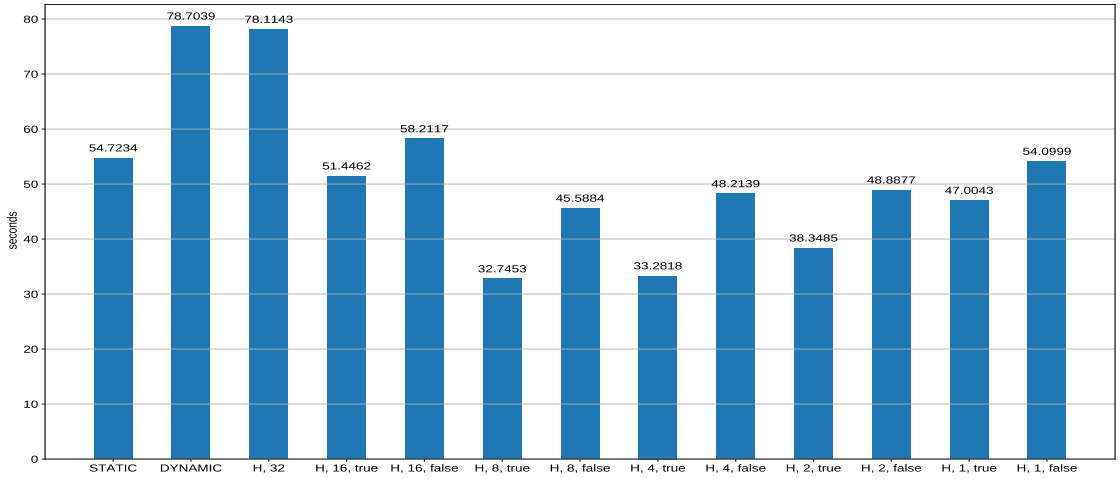
Πήραμε επιπλέον μετρήσεις για διάφορα μεγέθη των blocks. Ιδιαίτερο ενδιαφέρον παρουσιάζει η περίπτωση με μέγεθος 512×512. Στο σχήμα 4.16 βλέπουμε μετρήσεις για blocks τέτοιων διαστάσεων, με μέγεθος πίνακα $N = 50000$ και για grain size 1 και 8. Με αυτά τα δεδομένα, δημιουργούνται συνολικά 9604 blocks, οπότε μπορούμε να πούμε ότι υπάρχει η δυνατότητα ισομερούς διαμοιρασμού στα 32 συνολικά threads. Επιπλέον, τα blocks έχουν αρκετά μεγάλο μέγεθος, ώστε να μπορούμε να χρησιμοποιήσουμε μικρού μεγέθους grain size χωρίς να διατρέχουμε τον κίνδυνο συγκρούσεων μεταξύ των threads στη μνήμη, όπως περιγράψαμε στις προηγούμενες παραγράφους. Αυτό φαίνεται και στο σχήμα από τη χειρότερη επίδοση του dynamic, καθώς αυξάνουμε το grain size από 1 σε 8 (λόγω και χειρότερης ικανότητας διαμοιρασμού, εφόσον από 9604 τα κομμάτια εργασίας γίνονται $9604/8 \approx 1200$).

Στο μηχάνημα όπως είπαμε κάθε socket των 8 πυρήνων διαθέτει 16384KB L3 cache, το οποίο αντιστοιχεί ακριβώς σε 8 πίνακες (έναν για κάθε thread) από doubles μεγέθους 512×512. Συνεπώς, αυτή η περίπτωση φέρνει την L3 cache στα όρια της αποτελεσματικότητάς της, ειδικότερα λόγω της επαναλαμβανόμενης διάσχισης των blocks κατά την επεξεργασία τους. Το γεγονός αυτό, μαζί με το ότι τα blocks έχουν αρκετά μεγάλο μέγεθος, καθιστά την κλοπή εργασίας μεταξύ των groups πολύ πιο κοστοβόρα, λόγω του ότι διαταράσσει τις L3 caches και κατακερματίζει τα πεδία δεδομένων. Η κατάσταση αυτή προσομοιάζει κάπως την περίπτωση των μηχανημάτων με πολύ έντονα NUMA χαρακτηριστικά, όπου η επικοινωνία μεταξύ threads διαφορετικών κόμβων είναι πολύ ακριβή.

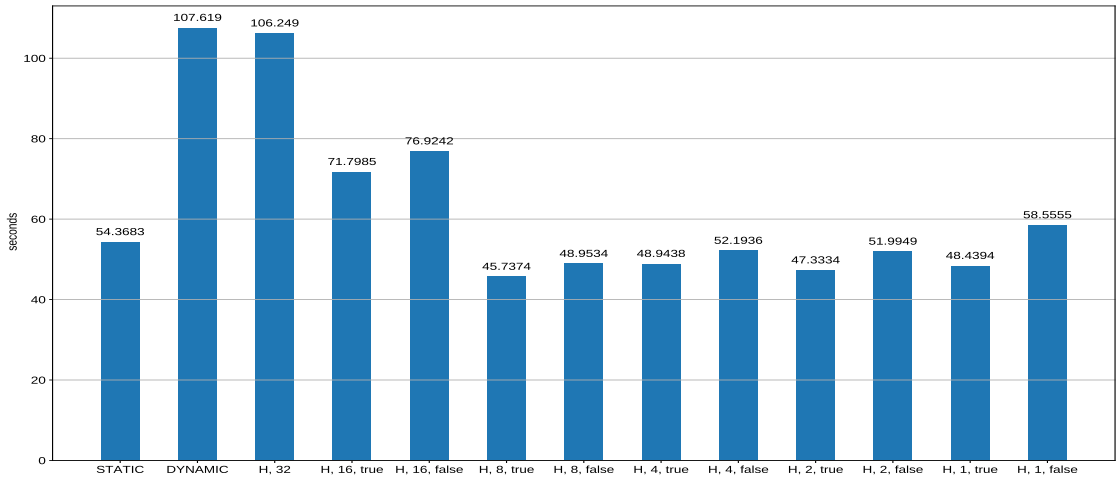
Τα παραπάνω έχουν ως αποτέλεσμα να προτιμούνται συνεχόμενα πεδία εργασίας που να είναι μεγαλύτερα και λιγότερα. Αυτό επιτυγχάνεται με τη χρήση μεγαλύτερων μεγεθών group, ώστε να είναι αντίστοιχα λιγότερα στο πλήθος και η εργασία να κατακερματίζεται σε μικρότερο βαθμό. Πράγματι, όπως βλέπουμε και στο σχήμα 4.16α′, η καλύτερη επίδοση επιτυγχάνεται για group size = 8, το οποίο είναι το μέγιστο μέγεθος group το οποίο χωράει σε έναν NUMA κόμβο. Παρατηρούμε επίσης ότι, ακόμα και αν το hierarchical stealing έχει αυξημένο κόστος, και πάλι σημειώνει βελτίωση της απόδοσης, ώστε τελικά η περίπτωση "H,8,true" να φτάνει το $32.745/54.723 = 59.8\%$ του χρόνου του static, από το θεωρητικό 50% που εξηγήσαμε στις προηγούμενες παραγράφους.

Συγκέντρωση των παρατηρήσεων

Από τις μετρήσεις στο συνθετικό αυτό benchmark φαίνεται πως το overhead που επιφέρει το hierarchical είναι συγκρίσιμο με αυτό του dynamic. Επιπλέον, βλέπουμε πως το hierarchical stealing πράγματι καταφέρνει να ισορροπεί την εργασία μεταξύ των groups, ενώ παράλληλα να διατηρεί την απομόνωση των πεδίων εργασίας τους, ώστε στην περίπτωση με μέγεθος group ίσο με 1 να βλέπουμε επίδοση σχεδόν ίδια



(α′) grain size = 1



(β′) grain size = 8

Σχήμα 4.16: Synthetic Benchmark, N = 50000, block = 512×512.

με την θεωρητικά βέλτιστη, ακόμα και για το μικρότερο δυνατό grain size. Τέλος, είδαμε ότι υπάρχουν περιπτώσεις όπου η επικοινωνία μεταξύ των groups γίνεται πιο ακριβή, ώστε να συμφέρει να αυξήσουμε το μέγεθός τους, κρατώντας το όμως πάντα μικρότερο του μεγέθους του κόμβου.

4.3 Rodinia: Dynamic Loads

Η Rodinia είναι μία βιβλιοθήκη με διάφορα benchmarks, στα οποία περιέχονται και benchmarks για OpenMP. Με στόχο να δούμε τη συμπεριφορά του hierarchical schedule σε πιο "real world" προγράμματα, πήραμε μετρήσεις των benchmarks αυτών, κάποιες από τις οποίες παρουσιάζονται και παρακάτω. Τα προγράμματα τα έχουμε χωρίσει σε "dynamic loads", που παρουσιάζουν δηλαδή ανισοκατανομή της εργασίας στα threads, και σε "static loads", τα οποία είναι balanced. Σε αυτήν την ενότητα περιγράφονται τα πρώτα.

Σημειώνεται ότι σε όλα τα προγράμματα του Rodinia η αρχικοποίηση των δεδομένων γινόταν πάντα από ένα μόνο thread, με αποτέλεσμα (λόγω της first touch πολιτικής των linux) όλα τα δεδομένα να τοποθετούνται σε ένα μόνο NUMA κόμβο. Ως αποτέλεσμα, τα προγράμματα εμφάνιζαν κακές συμπεριφορές στο NUMA μηχάνημά μας, όπως για παράδειγμα ασταθείς και απρόβλεπτους χρόνους εκτέλεσης, και αντίστοιχα κακές επιδόσεις. Συνεπώς, η αρχικοποίηση έχει τροποποιηθεί σε όλα τα προγράμματα ώστε να γίνεται παράλληλα με static schedule. Αυτό έχει και το επιπλέον πλεονέκτημα ότι μειώνεται ο χρόνος της αρχικοποίησης, αν και όλοι οι χρόνοι που μετρούνται στο Rodinia (και που παρουσιάζονται εδώ) είναι οι χρόνοι εκτέλεσης των εκάστοτε αλγορίθμων, και φυσικά δεν συμπεριλαμβάνουν την αρχικοποίηση των δεδομένων (εκτός και αν αναφέρονται ρητά, και ξεχωριστά πάντα από τους χρόνους για την επεξεργασία τους).

4.3.1 LUD

Το LUD είναι μία υλοποίηση του block LU factorization αλγορίθμου, και είναι από τα πιο ενδιαφέροντα προγράμματα του Rodinia, όσον αφορά τη συμπεριφορά του hierarchical και του dynamic schedule.

Το πρόγραμμα χωρίζει τον πίνακα σε chunks μεγέθους 16×16, τα οποία δέχονται επεξεργασία ανεξάρτητα και παράλληλα μεταξύ τους, σε πολλαπλές φάσεις. Παρακάτω συγκεντρώνουμε τα μεγέθη των $N \times N$ πινάκων, τον αριθμό των blocks που προκύπτουν στους πίνακες και τον χώρο μνήμης που καταλαμβάνει κάθε φορά το πρόγραμμα:

- N = 2048 $\rightarrow$ 16384 blocks , $\sim$ 15MB
- N = 8192 $\rightarrow$ 262144 blocks , 250MB
- N = 16384 $\rightarrow$ 1048576 blocks , 1GB

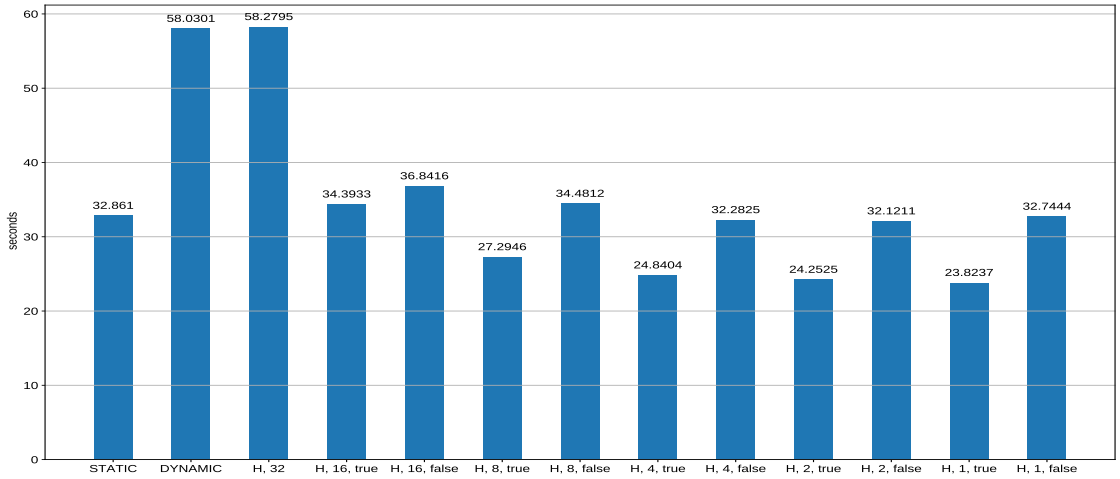
Εδώ, αντίθετα με το synthetic benchmark, δεν μπορούμε να αυξήσουμε πολύ τα μεγέθη των πινάκων, γιατί η εφαρμογή είναι απαιτητική και οι χρόνοι εκτέλεσης πολύ μεγαλύτεροι. Επίσης, ο αλγόριθμος είναι πιο περίπλοκος, με συνέπεια σε κάθε βήμα

να εμπλέκονται πολλές θέσεις μνήμης. Έτσι, όπως βλέπουμε και στο σχήμα 4.17, το dynamic scheduling δεν καταφέρνει ποτέ να βελτιώσει την επίδοση σε σχέση με το static. Αντίθετα, το hierarchical με ενεργοποιημένο το hierarchical stealing και για μεγέθη group μικρότερα του 8 (δηλαδή μικρότερα του NUMA κόμβου) παρουσιάζει σημαντική βελτίωση, και δείχνει ότι πράγματι υπάρχει ανισοκατανομή της εργασίας. Συγκεκριμένα, για μέγεθος group ίσο με 1 εμφανίζει βελτίωση της τάξης του 27% σε σχέση με το static, με μικρή εξάρτηση από το grain size.

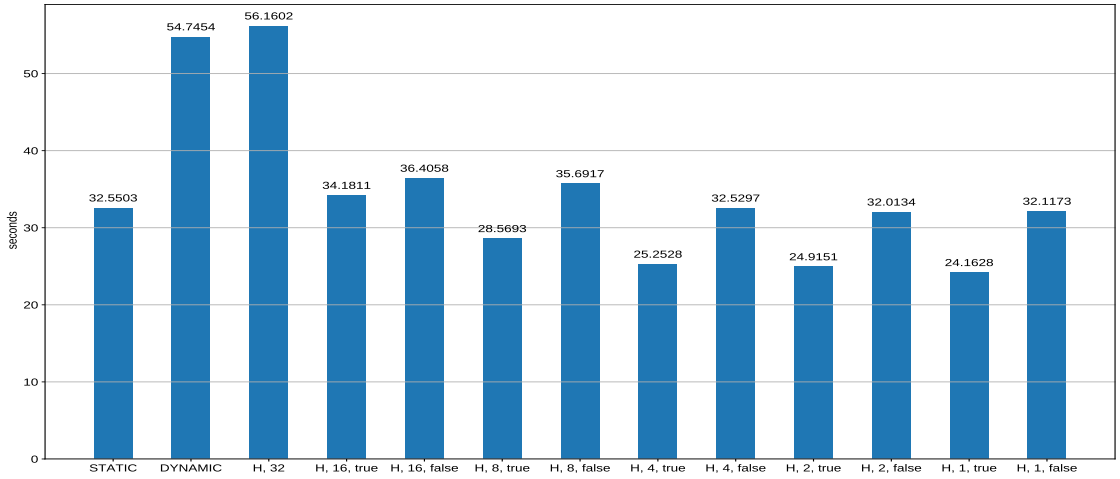
Είναι ενδιαφέρον να δούμε πιο φαινόμενο από αυτά που περιγράψαμε σε προηγούμενες παραγράφους εμφανίζεται εδώ. Στο σχήμα 4.18 παρουσιάζονται τα cache misses. Όπως βλέπουμε, δεν υπάρχουν μεγάλες αποκλίσεις από το static για τα L1 και L2 cache misses. Συνεπώς, συμπεραίνουμε ότι το grain size είναι αρκετά μεγάλο για να μην συγκρούονται τα threads σε κοινά cache line. Στην L3 cache βλέπουμε τη γνωστή συμπεριφορά, όπου εμφανίζεται βελτίωση για group size 8 και κάτω, ειδικά για μέγεθος 1, όπου τα cache misses είναι τα ίδια με το static. Αντίστοιχα, στο σχήμα 4.19 παρατηρούμε ότι, παρότι τα L1 και L2 cache misses είναι ίδια με το static, το dynamic και το hierarchical με μέγεθος group 32 παρουσιάζουν υψηλά επίπεδα των L1 και L2 stalls. Από τα προηγούμενα μπορούμε να συμπεράνουμε ότι κύριο λόγο για τη διαφορά στην επίδοση παίζουν τα NUMA χαρακτηριστικά του μηχανήματος.

Στο σχήμα 4.20 παρουσιάζονται οι μετρήσεις για τα πιο μικρά μεγέθη πινάκων. Παρατηρούμε πιο μέτριες επιδόσεις του hierarchical καθώς οι πίνακες γίνονται μικρότεροι, διότι μειώνεται ο μέσος φόρτος εργασίας των επαναλήψεων και ως αποτέλεσμα δυσχεραίνει η αναλογία με το overhead που επιφέρει το schedule. Για N = 2048 μάλιστα, ο μέσος φόρτος εργασίας είναι τόσο μικρός ώστε να μην μπορεί να αντισταθμίσει το overhead και να έχει χειρότερη επίδοση από το static .

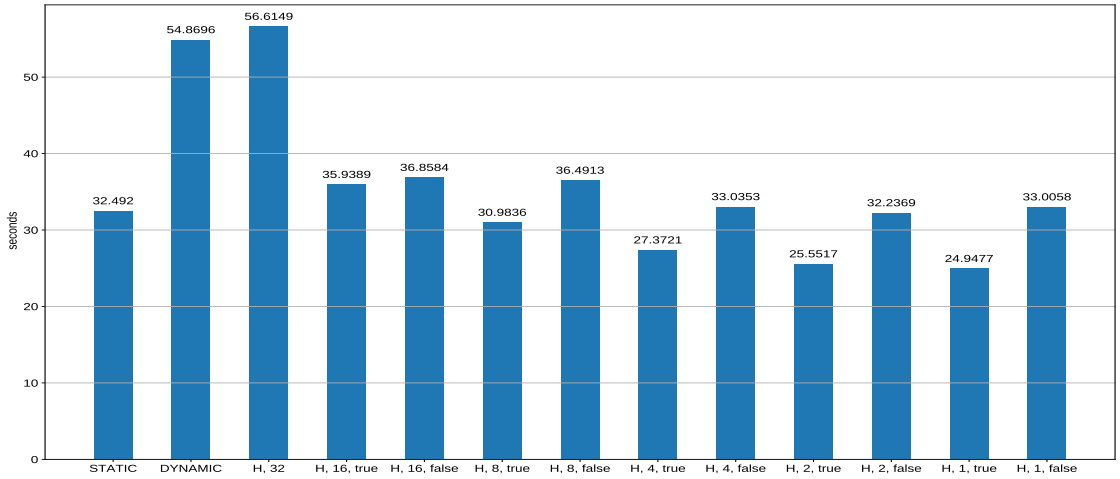
Στο σχήμα 4.21, παρουσιάζονται γραφήματα χωρίς παράλληλη αρχικοποίηση των πινάκων, δηλαδή αρχικοποίηση από ένα μόνο thread, ώστε να καταλήγει όλος ο πίνακας σε ένα μόνο node. Προσφέρονται ως παράδειγμα για να δικαιολογήσουμε την επιλογή μας για παράλληλη αρχικοποίηση. Όπως φαίνεται, όλα τα schedules παρουσιάζουν πολύ χειρότερη επίδοση. Ακόμα και στο static τα threads αναγκάζονται να συνωστίζονται σε ένα μόνο κόμβο μνήμης για να πάρουν δεδομένα, ώστε για N = 16384 να έχει χειρότερη επίδοση από το dynamic.



(α′) grain size = 64

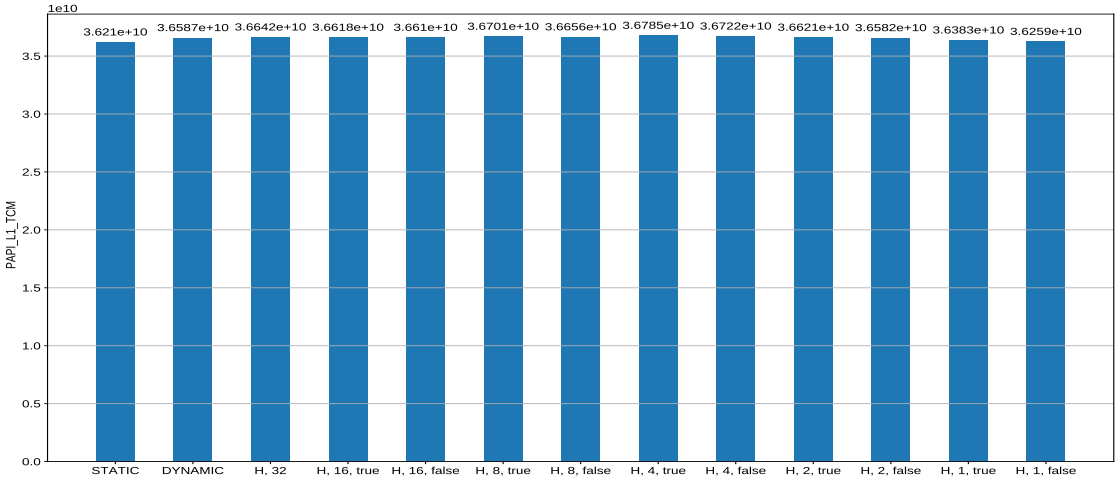


(β′) grain size = 512

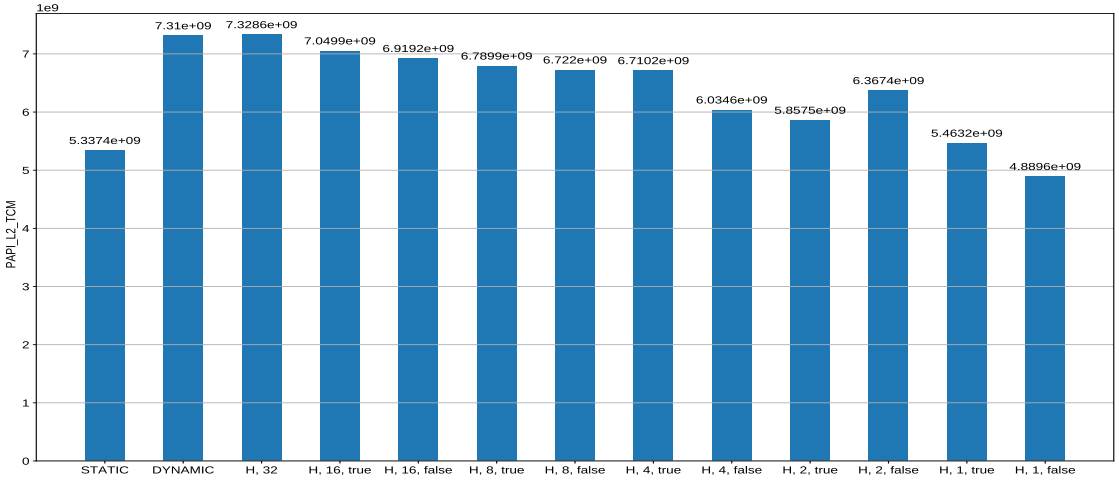


(γ′) grain size = 2048

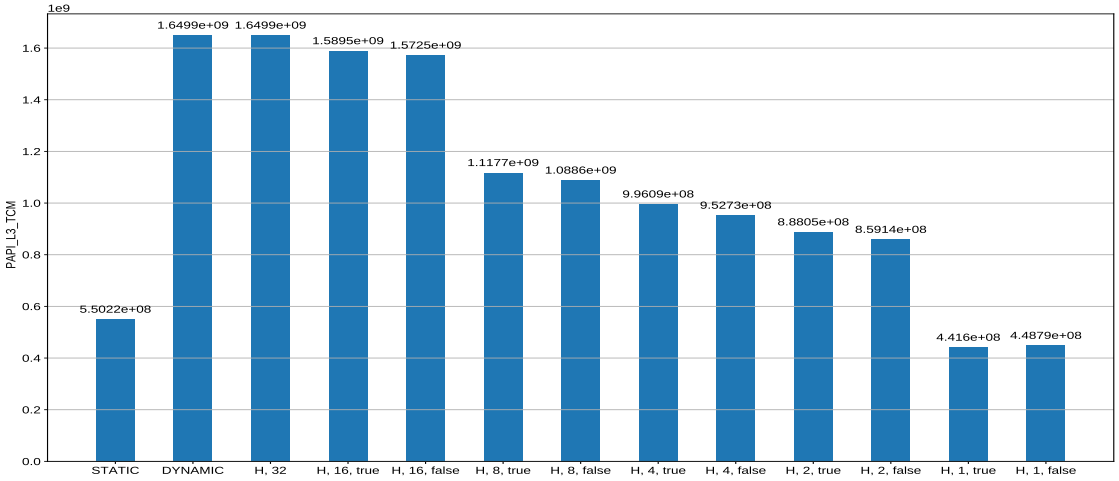
Σχήμα 4.17: LUD, N = 16384. Χρόνοι εκτέλεσης του προγράμματος, όταν η αρχικοποίηση των πινάκων γίνεται με static scheduling.



(α′) L1 cache misses

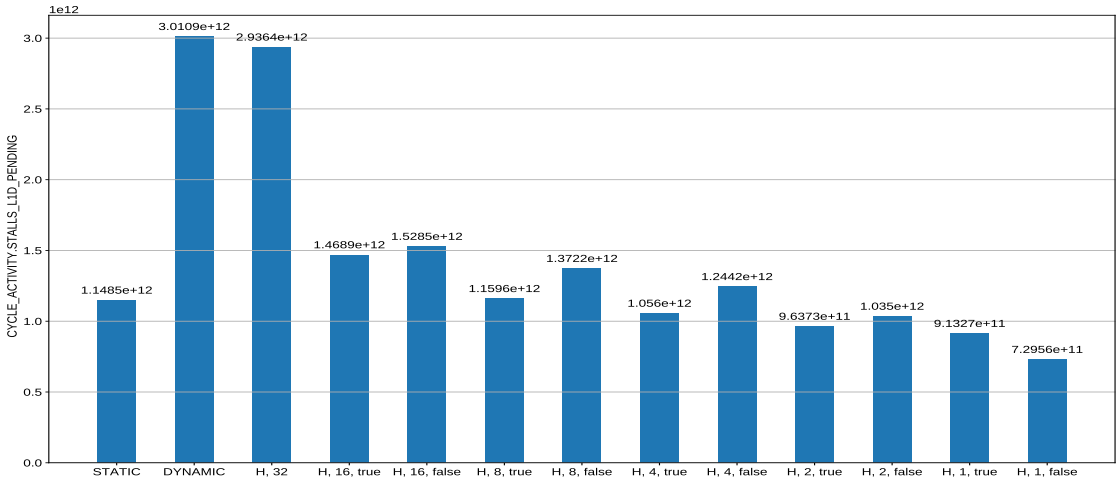


(β′) L2 cache misses

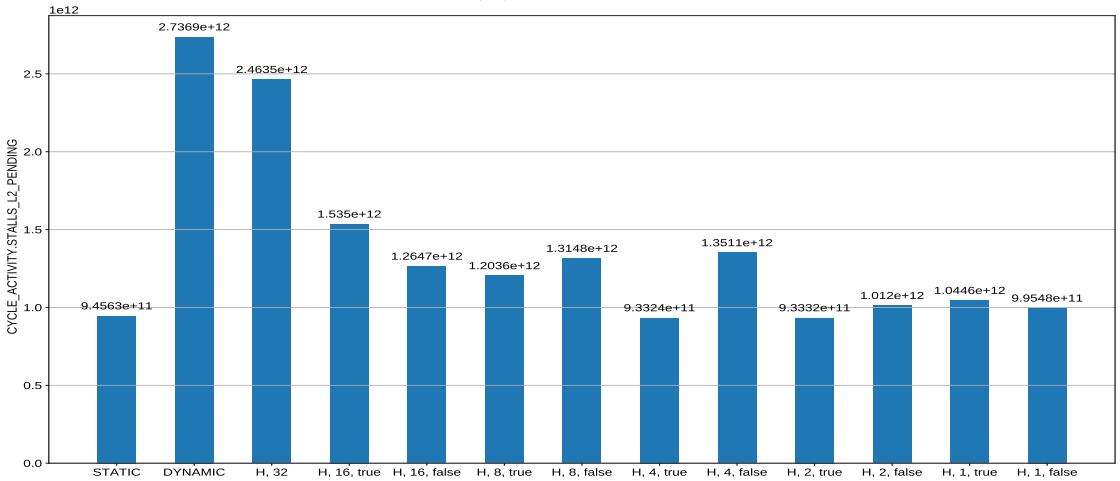


(γ′) L3 cache misses

Σχήμα 4.18: LUD, N = 16384, grain size = 64. Διαγράμματα των cache misses για τα διάφορα επίπεδα της cache.

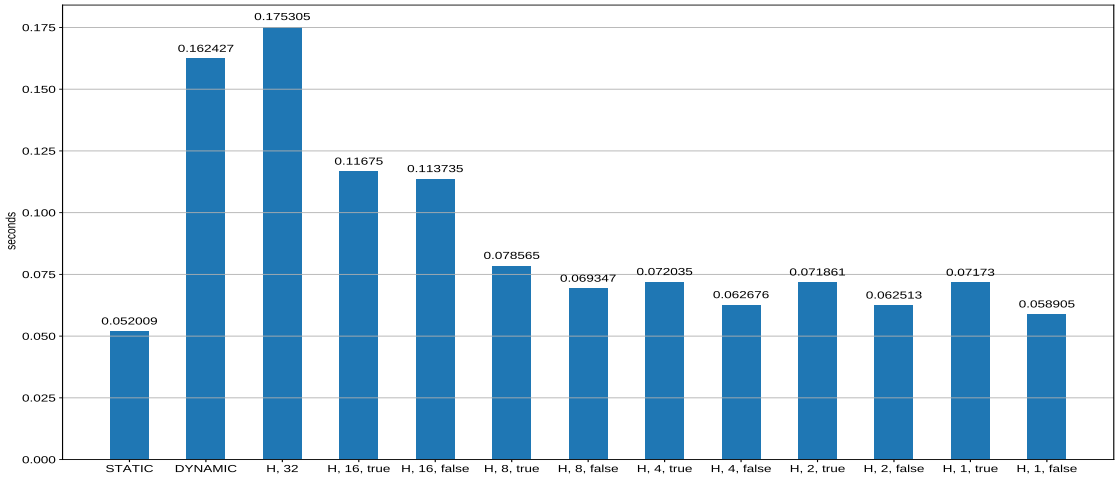


(α′) L1D stalls

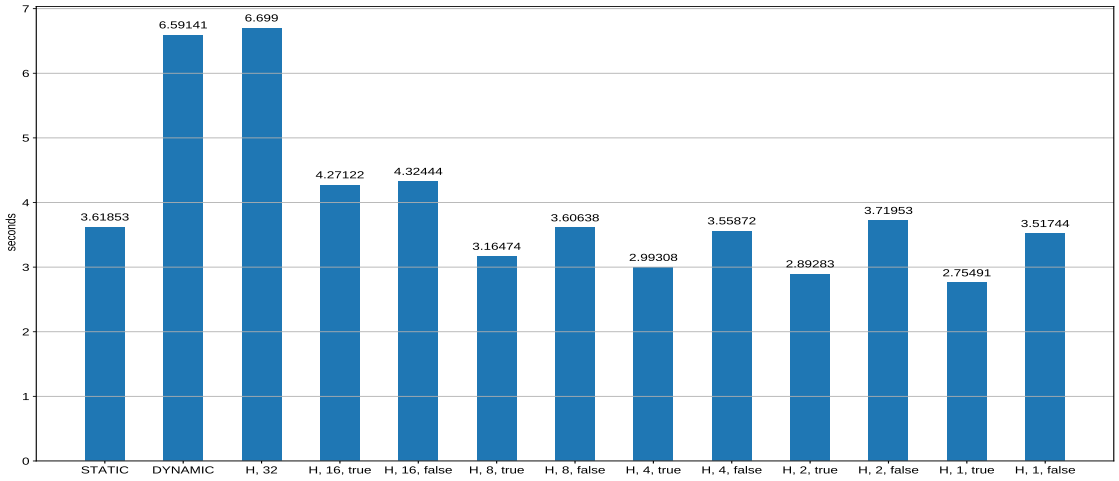


(β′) L2 stalls

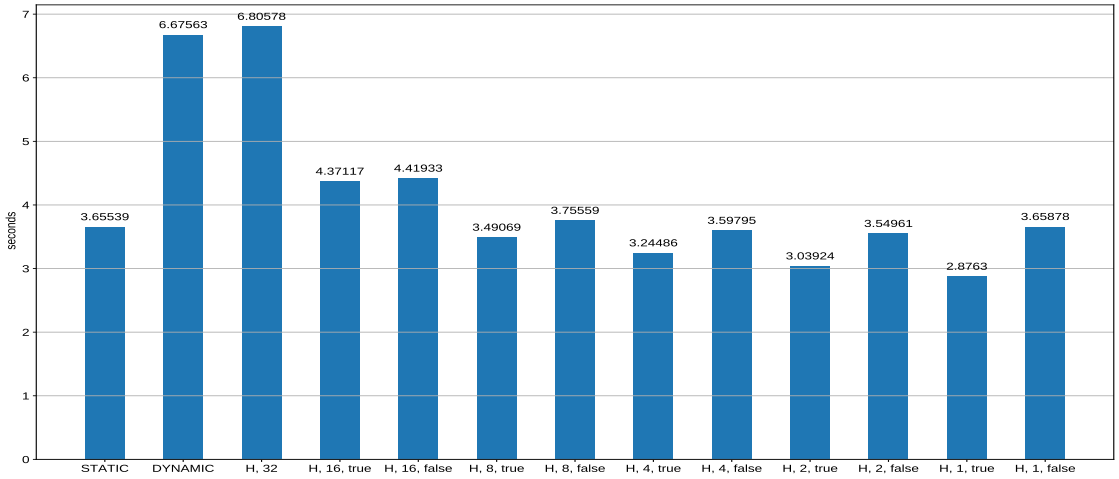
Σχήμα 4.19: LUD, N = 16384, grain size = 64. Διαγράμματα των stalls των επεξεργαστών λόγω L1D και L2 misses αντίστοιχα.



(α′) N = 2048 , grain size = 64

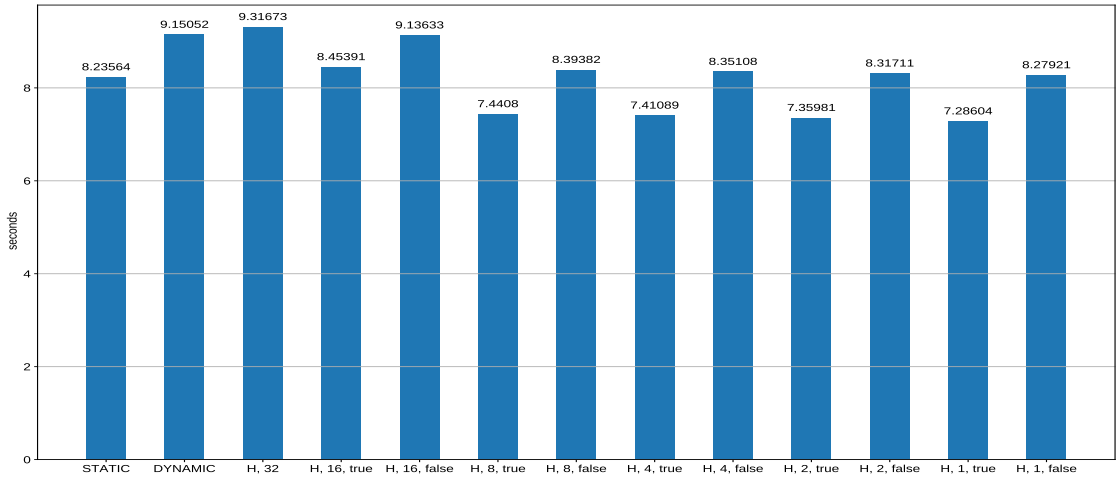


(β′) N = 8192 , grain size = 64

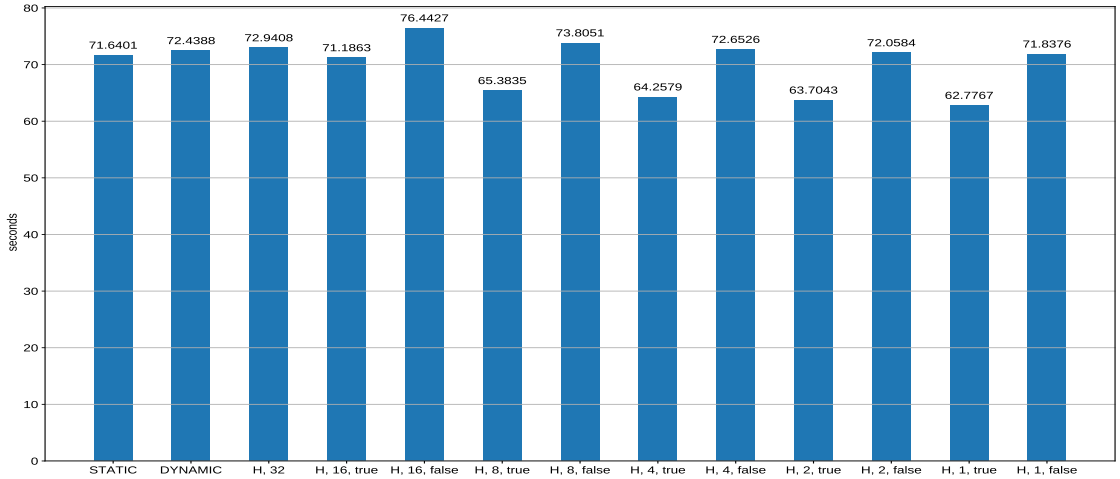


(γ′) N = 8192 , grain size = 512

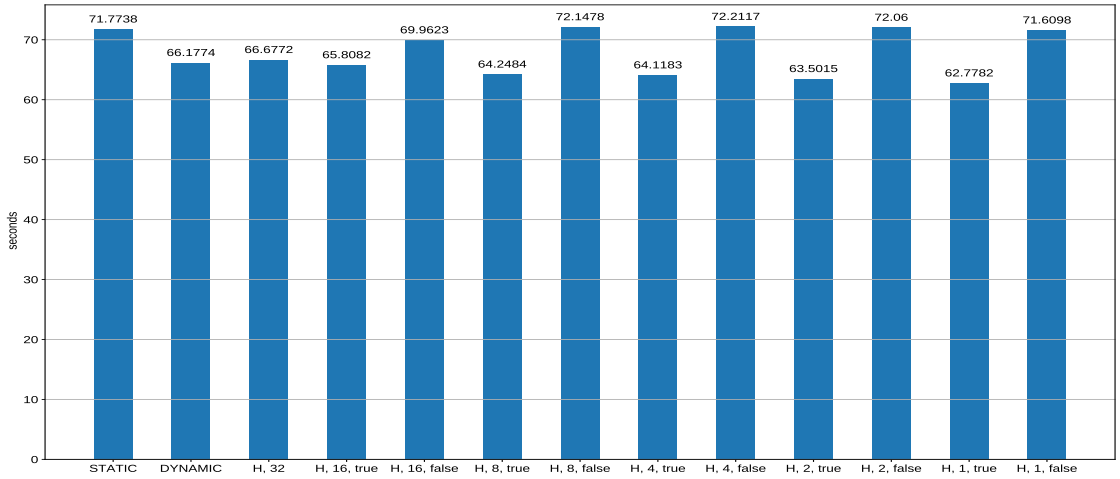
Σχήμα 4.20: LUD. Μετρήσεις για μικρότερα μεγέθη πινάκων. Αρχικοποίηση πινάκων με static scheduling.



(α′) N = 8192 , grain size = 16



(β′) N = 16384 , grain size = 16



(γ′) N = 16384 , grain size = 64

Σχήμα 4.21: LUD. Αρχικοποίηση πινάκων από ένα thread, και επομένως αποθήκευση σε ένα μόνο memory node.

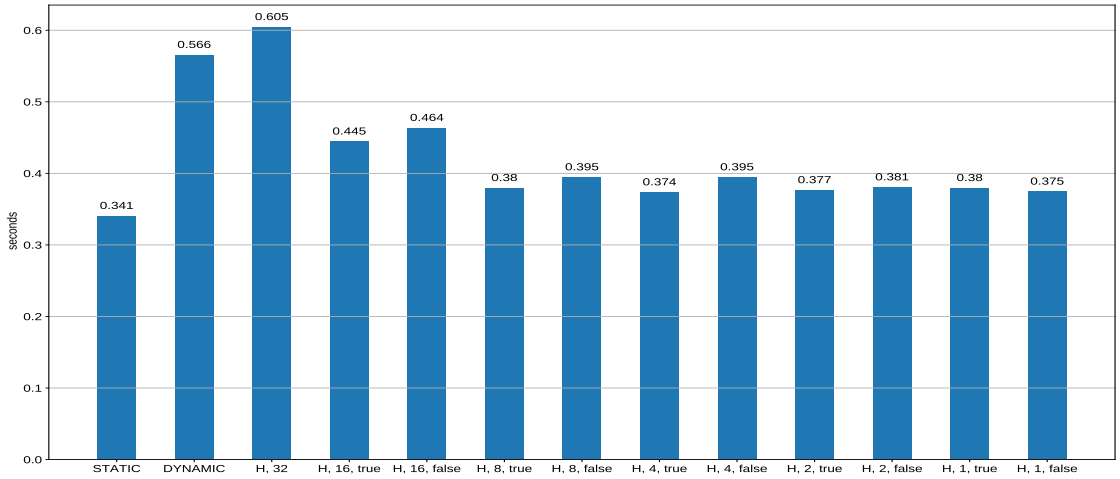
4.3.2 NW (Needleman–Wunsch)

Το πρόγραμμα αυτό έχει παρόμοια δομή όσων αφορά την επεξεργασία των δεδομένων με το lud. Ομοίως, οι πίνακες χωρίζονται σε chunks μεγέθους 16×16, τα οποία τα threads επεξεργάζονται ανεξάρτητα. Λόγω αυτού, όπως βλέπουμε και στις μετρήσεις των σχημάτων 4.22 και 4.23, εμφανίζει και αντίστοιχη συμπεριφορά.

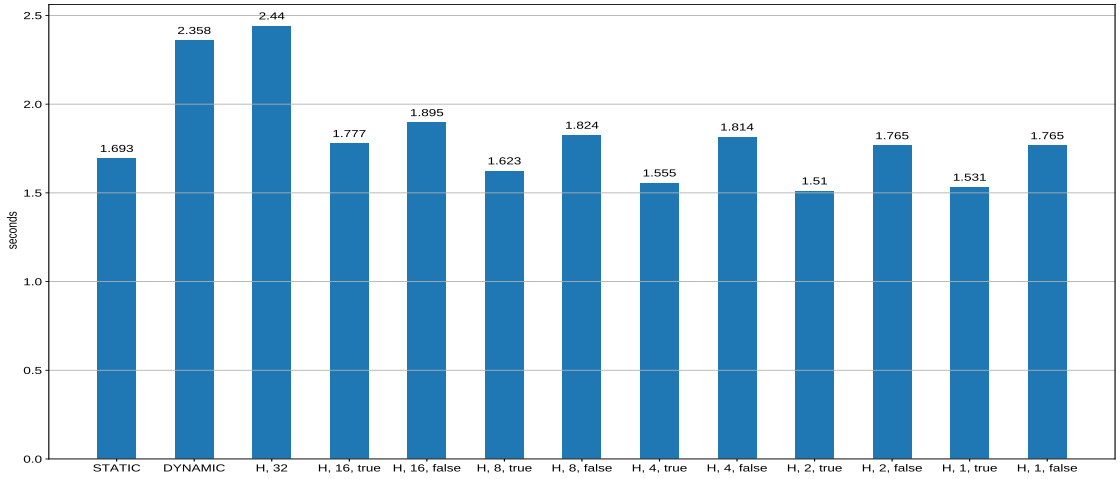
Δοκιμάζονται διάφορα μεγέθη πινάκων, τα οποία παρουσιάζονται παρακάτω μαζί με τη συνολική μνήμη που καταλαμβάνει το πρόγραμμα:

- N = 16384 → 3GB
- N = 32768 → 12GB
- N = 65536 → 48GB

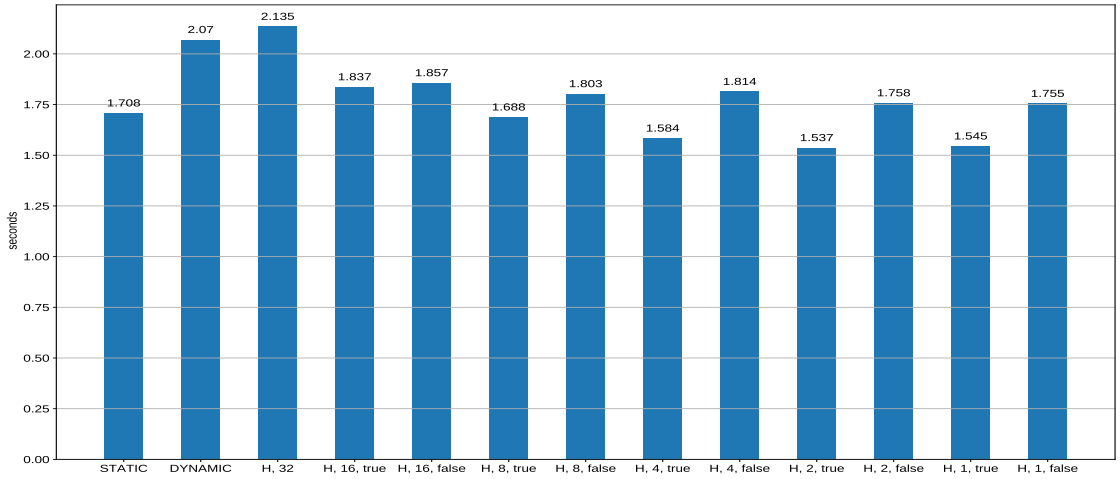
Σε αυτό το benchmark το μέσο κόστος των επαναλήψεων είναι αρκετά μικρότερο από ότι στο lud, ώστε να μπορούμε να δοκιμάσουμε πολύ μεγαλύτερα μεγέθη πινάκων. Αντίστοιχα όμως, βλέπουμε ότι μόνο σε πολύ μικρά grain size καταφέρνουμε να πετύχουμε βελτίωση της απόδοσης. Φυσικά το dynamic δεν μπορεί να αντεπεξέλθει σε τόσο μικρά κομμάτια δεδομένων, ώστε τελικά να έχει χειρότερη απόδοση από το static σε όλες τις μετρήσεις. Το hierarchical από την άλλη καταφέρνει να βελτιώσει την απόδοση στα δύο πιο μεγάλα μεγέθη, και για group μικρότερα του μεγέθους των NUMA κόμβων.



(α′) N = 16384 , grain size = 1

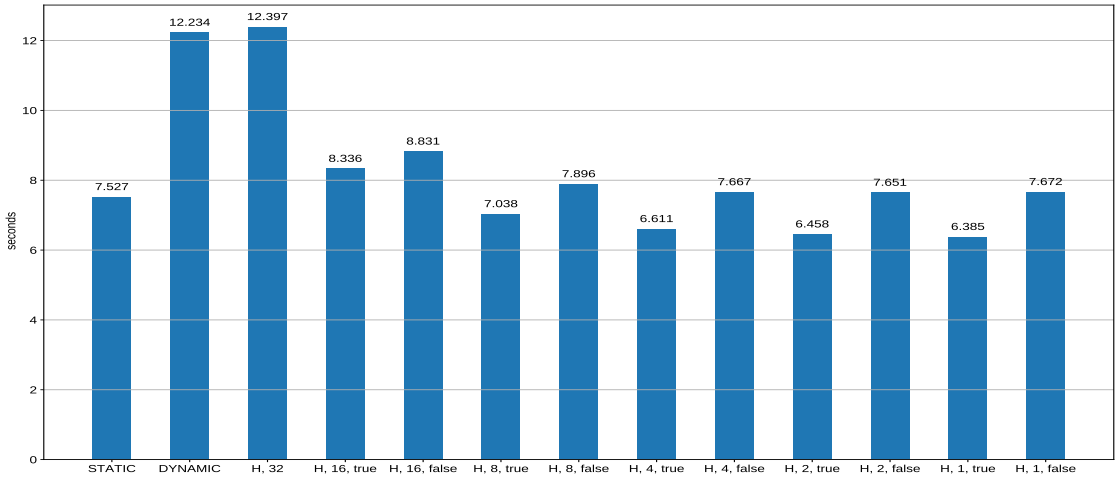


(β′) N = 32768 , grain size = 1

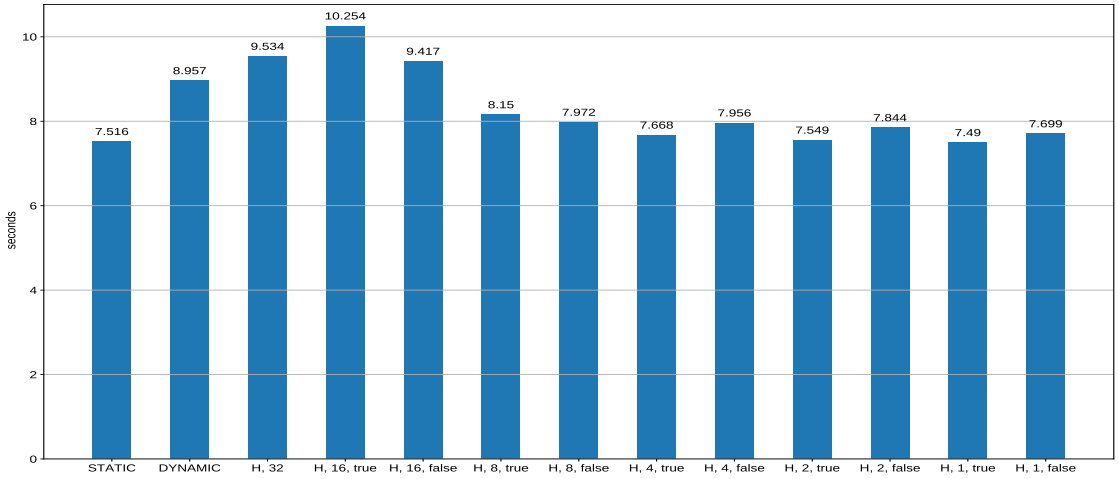


(γ′) N = 32768 , grain size = 8

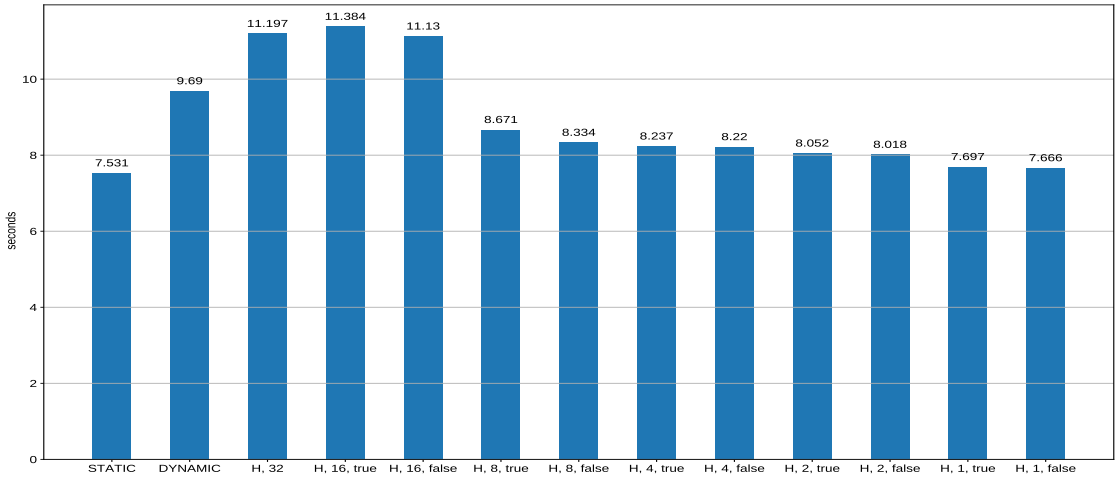
Σχήμα 4.22: NW



(α′) N = 65536 , grain size = 1



(β′) N = 65536 , grain size = 64



(γ′) N = 65536 , grain size = 128

Σχήμα 4.23: NW
Το dynamic δεν σημειώνει καλύτερες επιδόσεις ούτε για τα πιο μεγάλα grain sizes.

4.3.3 Particle Filter

Το benchmark αυτό παράγει αρχικά ένα συνθετικό βίντεο, κάθε frame του οποίου αποθηκεύεται σε έναν δισδιάστατο πίνακα, και έπειτα εκτελεί τον particle filter αλγόριθμο πάνω στο βίντεο αυτό. Το πρόγραμμα αποτελείται από τρία μέρη:

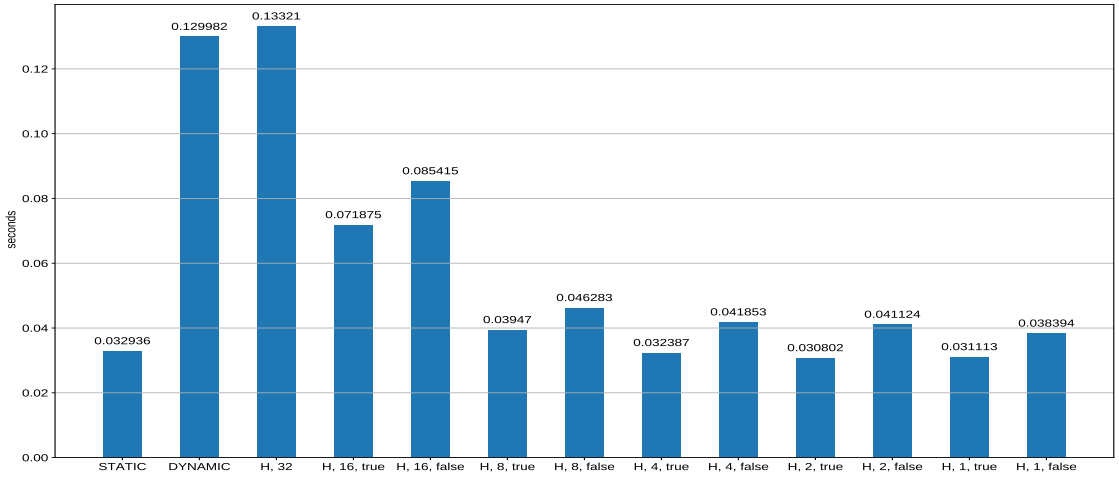
1. Αρχικά δεσμεύει μνήμη για τους πίνακες που θα χρειαστεί. Σε αυτό το σημείο τροποποιούμε το πρόγραμμα, αρχικοποιώντας τους πίνακες με static schedule, για διαμοιρασμό μέσω του first touch σε όλους τους NUMA κόμβους.

2. Σε αυτό το βήμα δημιουργεί το συνθετικό βίντεο, και το αποθηκεύει στους πίνακες. Το βήμα αυτό είναι απλό βήμα αρχικοποίησης (συνεπώς απόλυτα balanced εργασία), και δεν μας ενδιαφέρει η απόδοσή του. Παρ΄ όλα αυτά, επειδή ήταν ιδιαίτερα χρονοβόρο καταλήξαμε να παραλληλοποιήσουμε ως προς την επεξεργασία των frames του βίντεο, τα οποία ήταν ανεξάρτητα μεταξύ τους (το περισσότερο χρόνο τον σπαταλούσε στην αρχικοποίηση με θόρυβο).

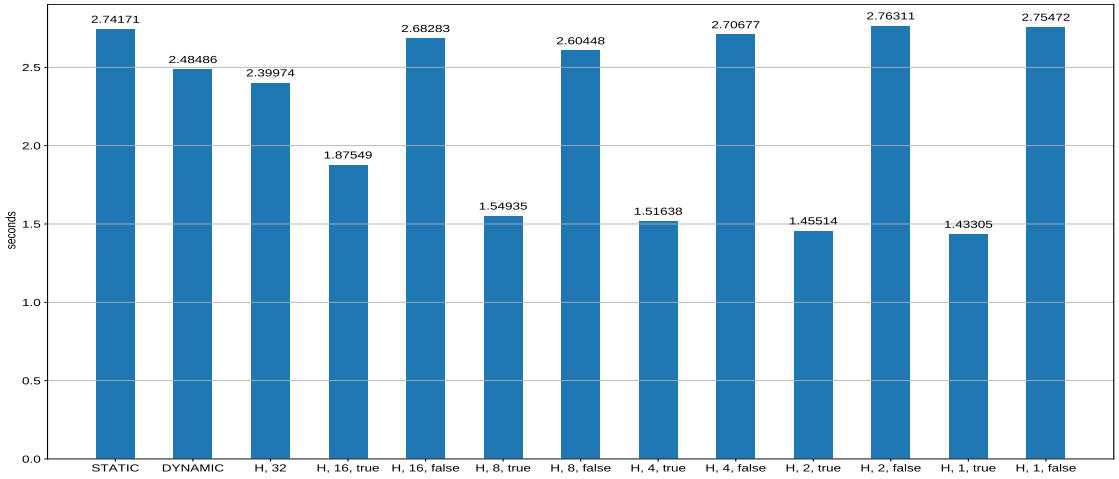
3. Για κάθε frame με τη σειρά εφαρμόζει τον κύριο αλγόριθμο του particle filter στον αντίστοιχο πίνακα. Η παραλληλοποίηση γίνεται μέσα στην επεξεργασία κάθε frame, χρησιμοποιώντας τα αποτελέσματα των προηγούμενων.

Το τελευταίο βήμα είναι αυτό που μελετάμε. Όπως βλέπουμε στα σχήματα 4.24 και 4.25, η εργασία σε αυτό είναι άνισα κατανεμημένη. Το πρόγραμμα είναι αρκετά απαιτητικό σε χρόνο, οπότε επιλέχθηκαν μικρά μεγέθη πινάκων, και πήραμε μετρήσεις για διάφορα πλήθη σωματιδίων (Nparticles στα σχήματα). Όσον αφορά τον αριθμό των frames, η συμπεριφορά του προγράμματος παραμένει ίδια, οπότε επιλέχθηκαν σταθερά 10 για όλες τις μετρήσεις.

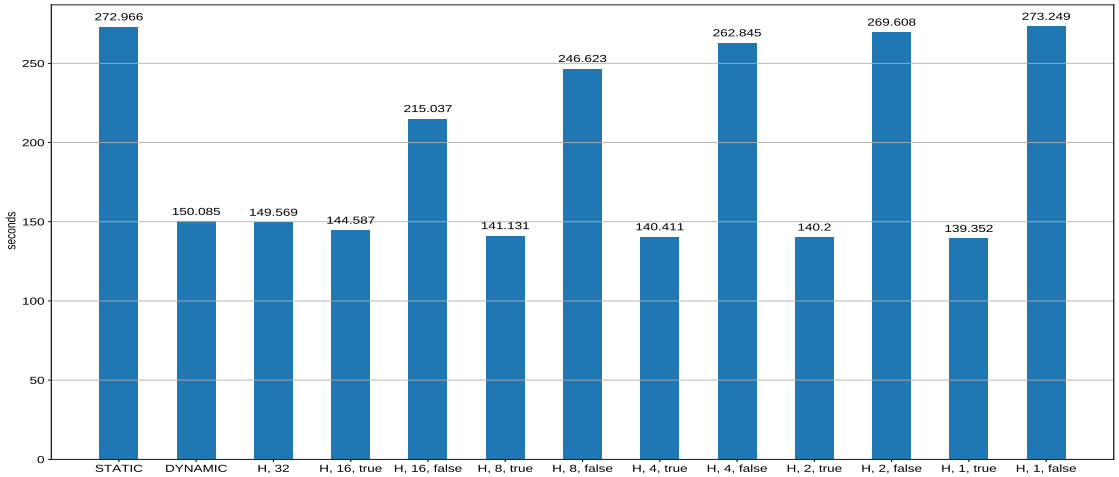
Η συμπεριφορά των schedules είναι πολύ παρόμοια με αυτή του συνθετικού benchmark της παραγράφου 4.2. Στο κύριο κομμάτι της εργασίας, το Nparticles επηρεάζει τις αποστάσεις των δεδομένων που προσπελαύνουν γειτονικές επαναλήψεις. Συνεπώς, το dynamic στα υπερβολικά μικρά grain size και για μικρότερες τιμές του Nparticles εμφανίζει συγκρούσεις των threads στη μνήμη, με αποτέλεσμα αντίστοιχα κακή επίδοση. Αυξάνοντας το ένα από τα δύο εξαλείφεται αυτό το πρόβλημα και το dynamic σημειώνει καλές επιδόσεις. Παράλληλα, όπως και στο synthetic benchmark, βλέπουμε ότι το hierarchical schedule με μέγεθος group από 8 και κάτω εμφανίζει πιο ευέλικτη συμπεριφορά στις μεταβολές των grain size και Nparticles, και έχει αντίστοιχα ίδιες ή καλύτερες επιδόσεις σε σχέση με το dynamic.



(α′) x,y=2048 , Nparticles=10000

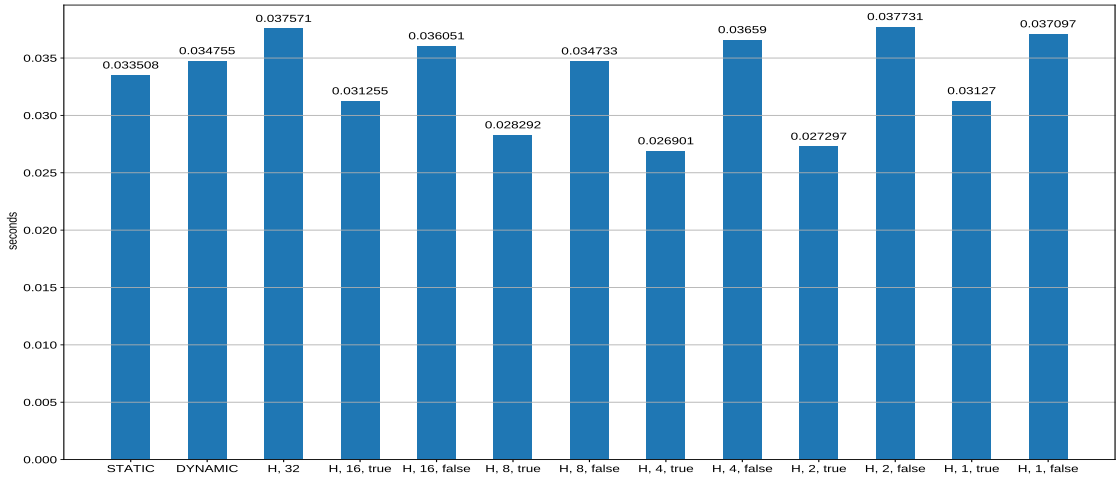


(β′) x,y=2048 , Nparticles=100000

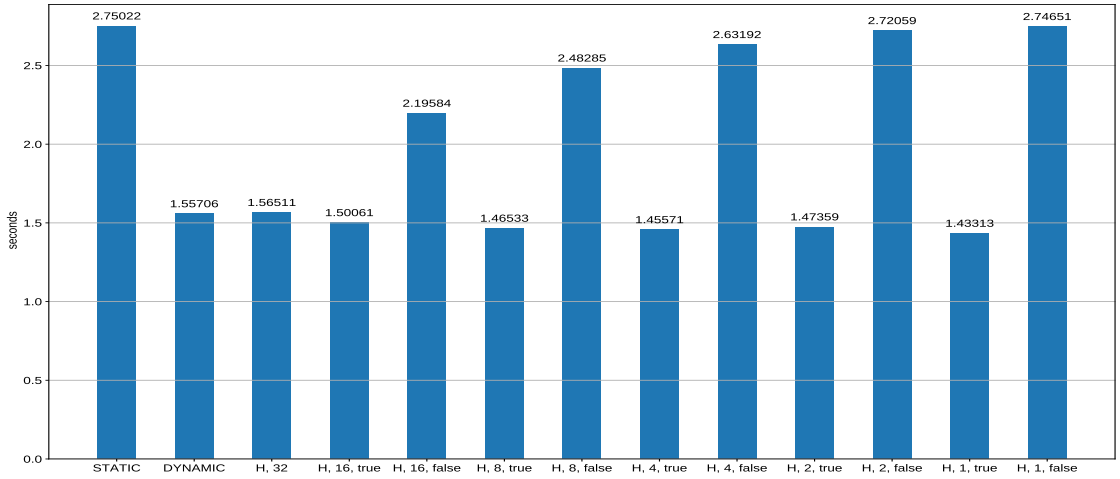


(γ′) x,y=2048 , Nparticles=1000000

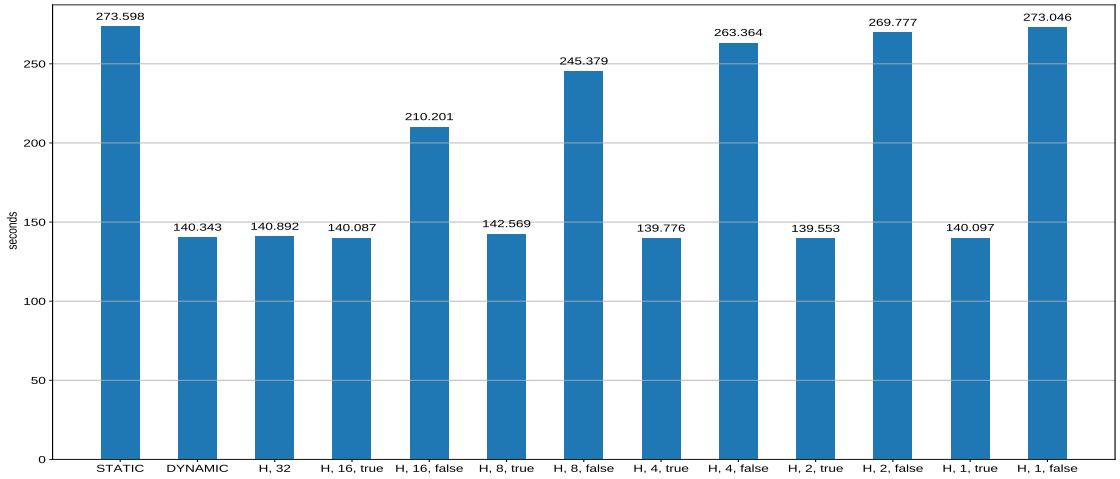
Σχήμα 4.24: Particle Filter, frames = 10, grain size 1.
Για το hierarchical σημειώνεται επίσης το ‘group size’ και η τιμή του ‘gomp_hierarchical_stealing’.



(α′) x,y=2048 , Nparticles=10000



(β′) x,y=2048 , Nparticles=100000



(γ′) x,y=2048 , Nparticles=1000000

Σχήμα 4.25: Particle Filter, frames = 10, grain size 8.

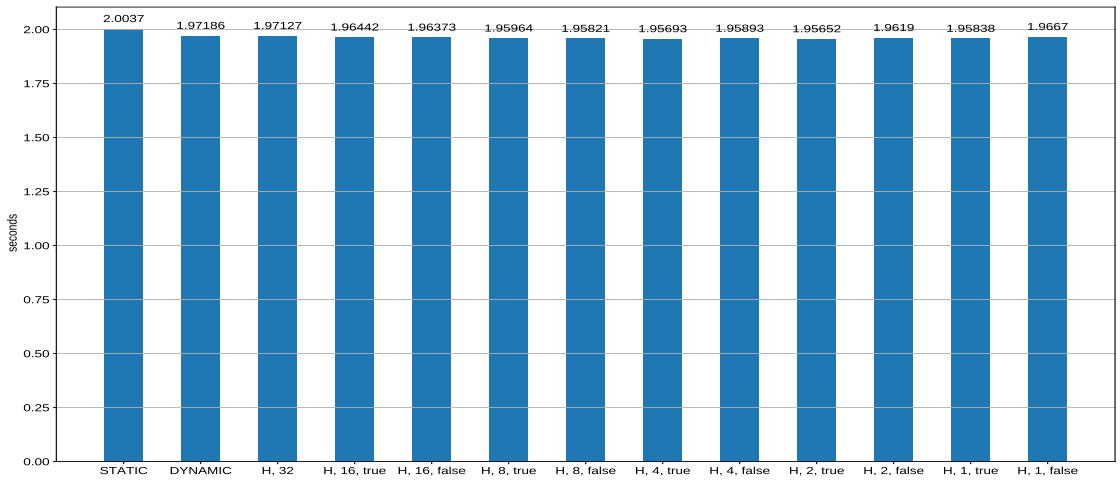
4.4 Rodinia: Static Loads

Τα επόμενα φορτία εργασίας είναι στατικά φορτία του rodinia, και ομαδοποιούνται με κριτήριο την παρόμοια συμπεριφορά. Τα workloads είναι σχετικά balanced, οπότε το work stealing δεν έχει συνήθως κάτι να προσφέρει, παρά μόνο υποβαθμίζει την επίδοση λόγω του επιπλέον overhead που εισάγει. Μπορούν να χωριστούν σε δύο περιπτώσεις:

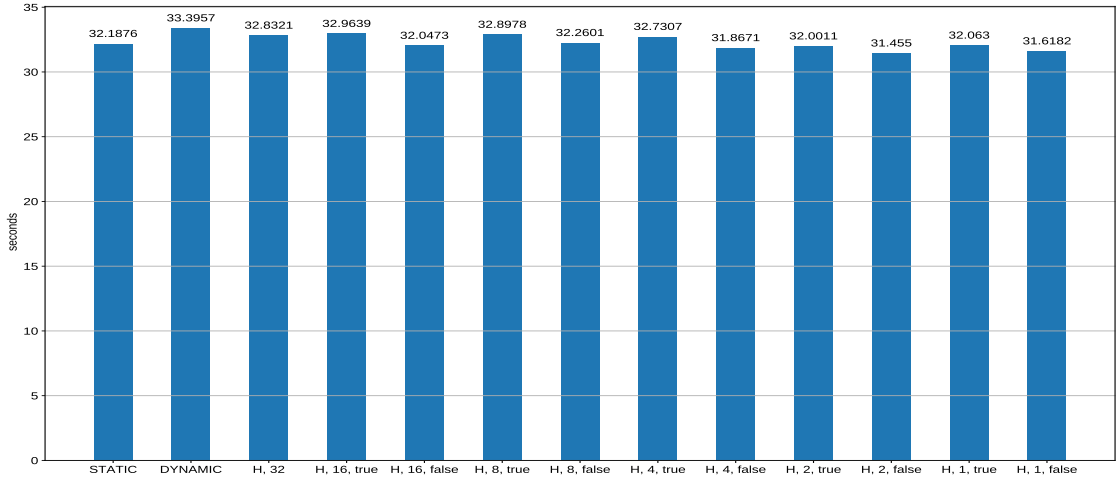
- nn, heartwall, streamcluster

  Το μέσο κόστος κάθε iteration είναι αρκετά μεγάλο ώστε η επίδοση να είναι περίπου ίδια για όλα τα schedules (ελάχιστο έως μηδενικό unbalance). Τροποποιήσεις:

  - nn: Παράλληλο διάβασμα αρχείων για διαμοιρασμό σε όλα τα nodes.
  - streamcluster: Αρχικοποίηση των πινάκων που χρησιμοποιούνται στα parallel regions με parallel for static, για διαμοιρασμό μέσω του first touch.



Σχήμα 4.26: nn: list131072k, mem = 10.6GB, grain size = 1024

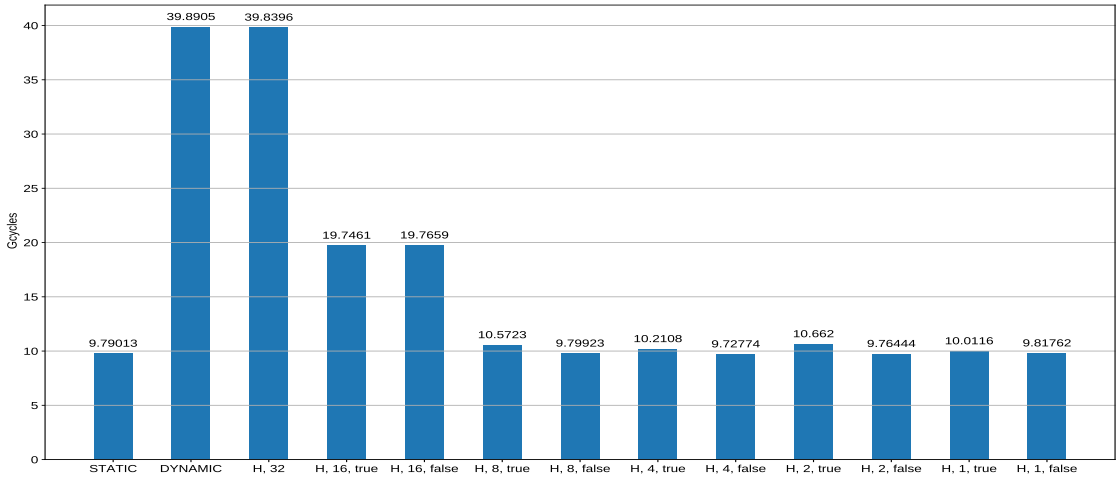


Σχήμα 4.27: streamcluster: N = 2^{17} , mem = 135MB , grain size = 128

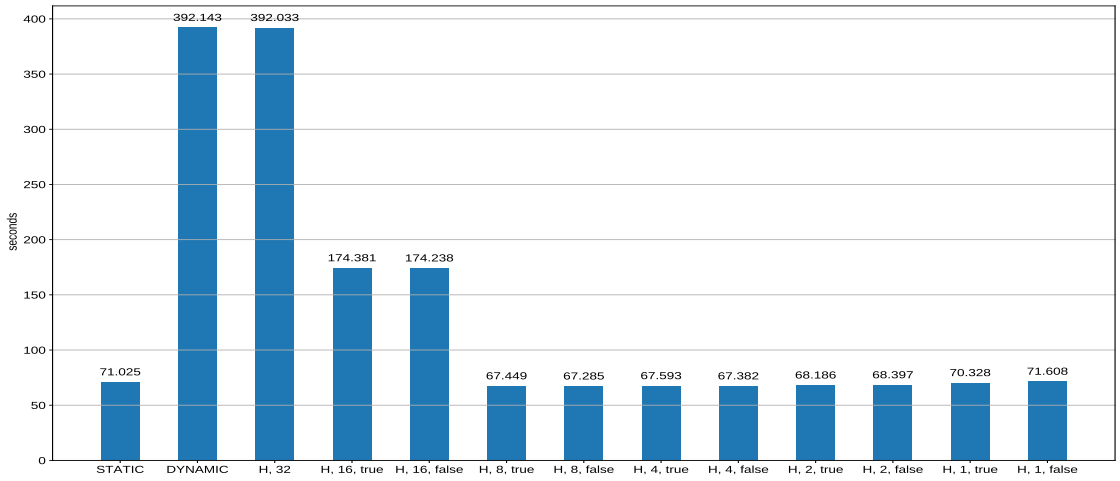
- pathfinder, hotspot, bfs, lavaMD, backprop, srad, cfd, hotspot3D

  Το overhead υποβαθμίζει αισθητά την επίδοση, με το hierarchical schedule να είναι ίδιο ή λίγο χειρότερο από το static στα πιο μικρά node sizes, ενώ στα μεγάλα και στο dynamic η επίδοση να έχει μειωθεί δραματικά. Σε μερικά benchmarks στο hierarchical για nodes sizes μικρότερα του αριθμού των cores στα sockets και χωρίς stealing μεταξύ των thread groups, μπορεί να σημειωθεί ελάχιστη βελτίωση. Τροποποιήσεις:

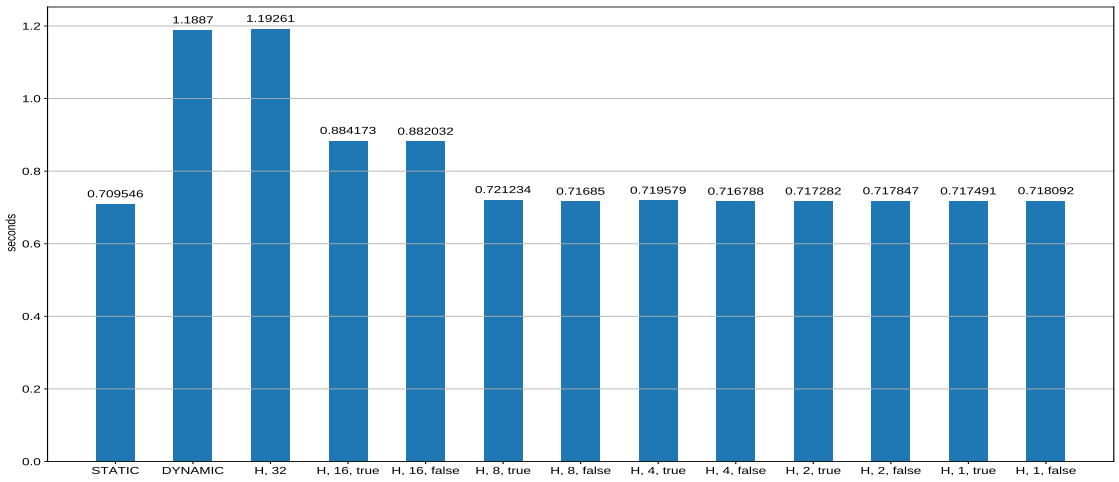
  - pathfinder, lavaMD, srad: Αρχικοποίηση των πινάκων με parallel for static, και αντικατάσταση των εμφανίσεων της rand() με rand_r().
  - hotspot, bfs: Διάβασμα των αρχείων παράλληλα και αρχικοποίηση των πινάκων με parallel for static.
  - backprop: Το πρόγραμμα διατρέχει τους 2D πίνακες κατά στήλες, οπότε αντιμεταθέσαμε τους πίνακές ώστε να είναι αποθηκευμένοι σε column major μορφή, και να διατηρείται η τοπικότητα. Ομοίως με τις προηγούμενες περιπτώσεις, αρχικοποιούνται παράλληλα.



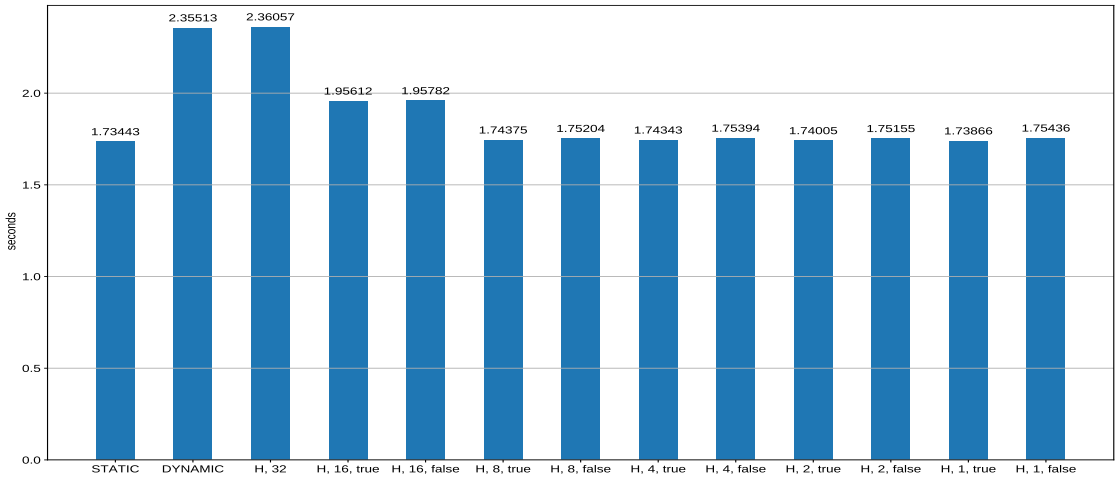
Σχήμα 4.28: pathfinder: N = 10^8, mem = 76GB, grain size = 1024



Σχήμα 4.29: hotspot: N = 16384, mem = 4GB, iterations = 1000, grain size = 8



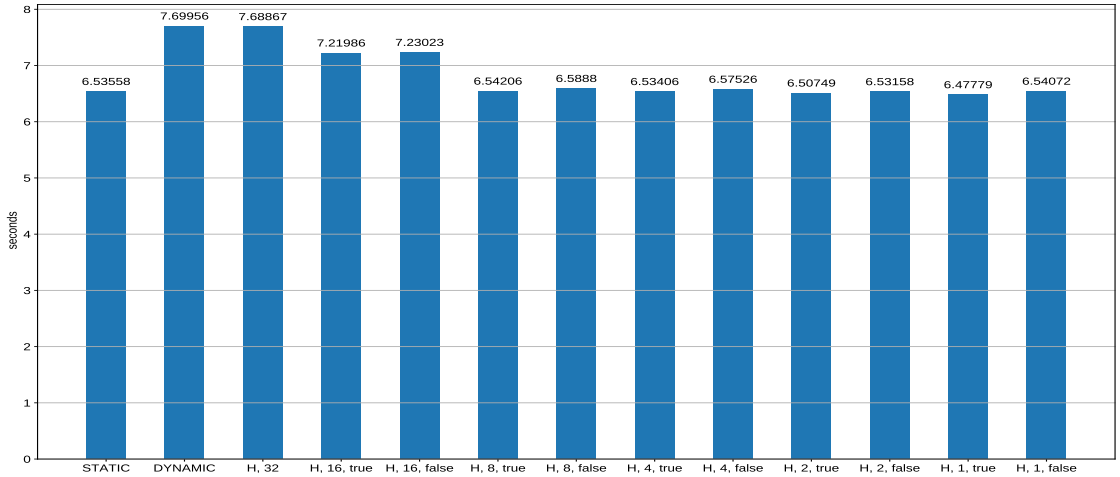
(α′) nodes = 2^{25}, min edges = 4, mem = 2.4GB, grain size = 4096



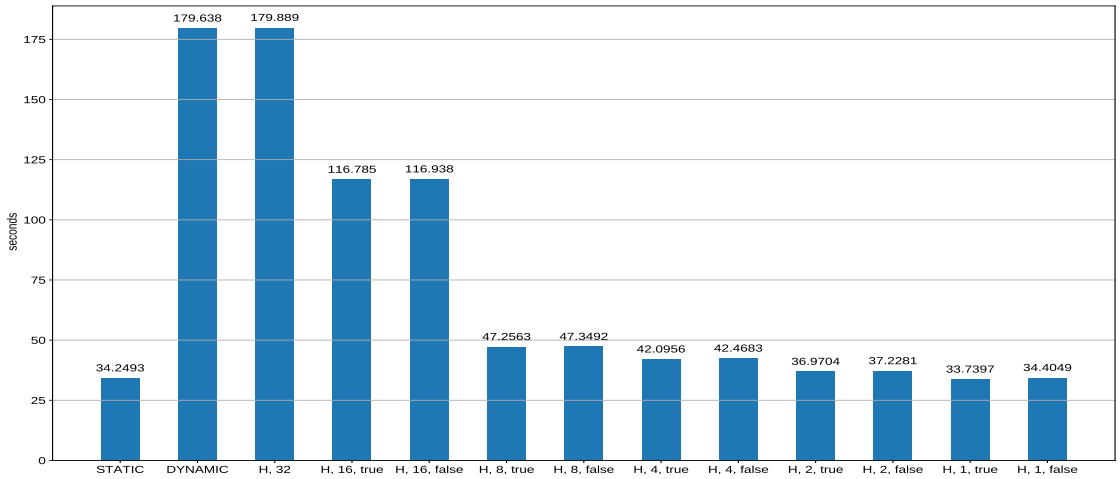
(β′) nodes = 2^{25}, min edges = 16, mem = 20GB, grain size = 1024

Σχήμα 4.30: bfs

Εδώ χρειάστηκε να αυξήσουμε περισσότερο το grain size, γιατί το μέσο κόστος των επαναλήψεων είναι πολύ μικρό.

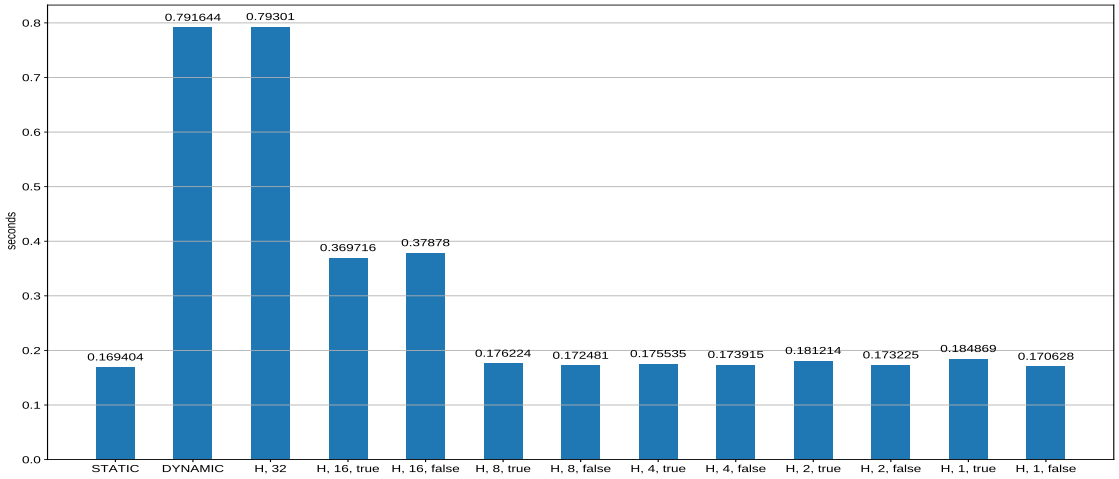


(α′) boxes = 100, numperbox = 10, mem = 1.3GB, grain size = 1

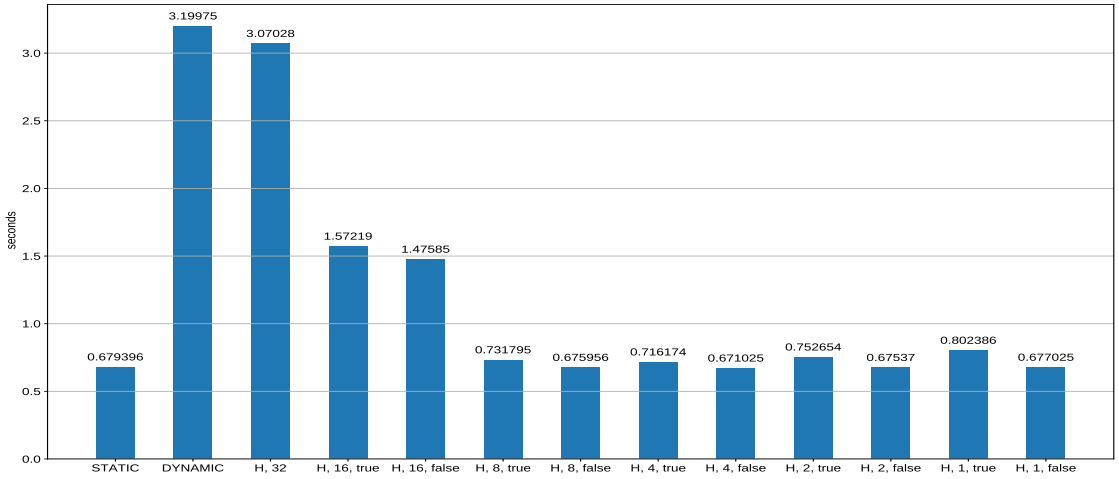


(β′) nodes = 500, numperbox = 2, mem = 93GB, grain size = 1

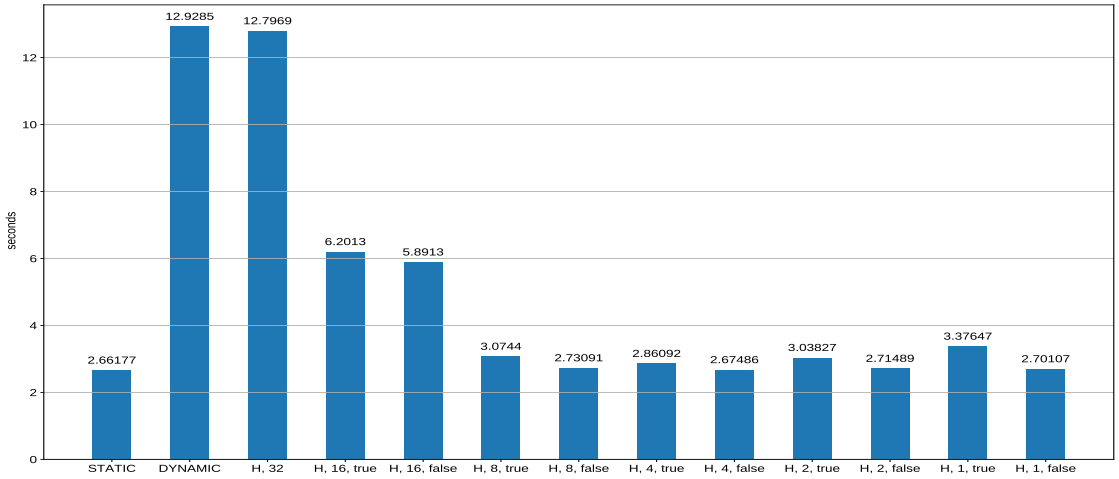
Σχήμα 4.31: lavaMD



(α′) layer size = 2^{20}, nhidden = 512, nout = 512, mem = 4GB, grain size = 1

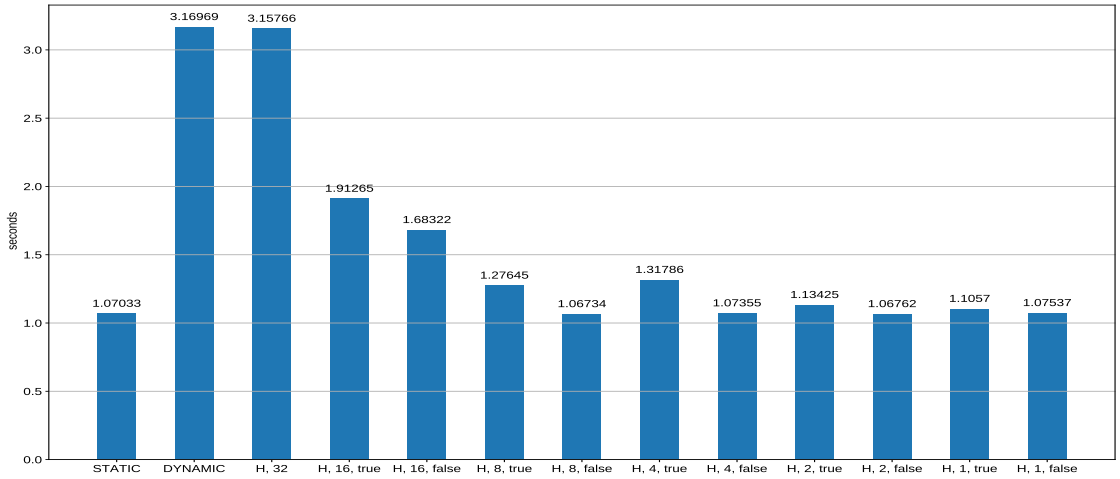


(β′) layer size = 2^{22}, nhidden = 512, nout = 512, mem = 16GB, grain size = 1

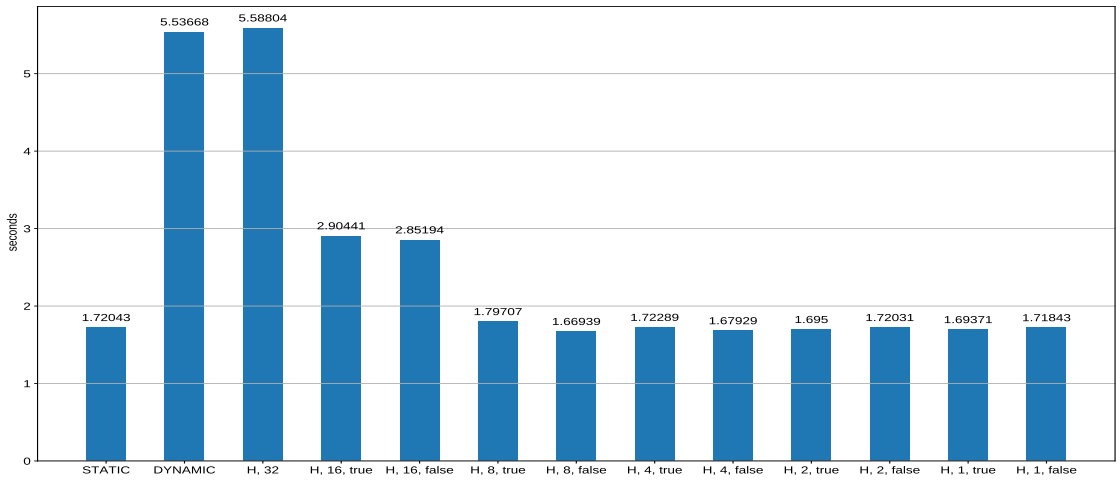


(γ′) layer size = 2^{24}, nhidden = 512, nout = 512, mem = 64GB, grain size = 1

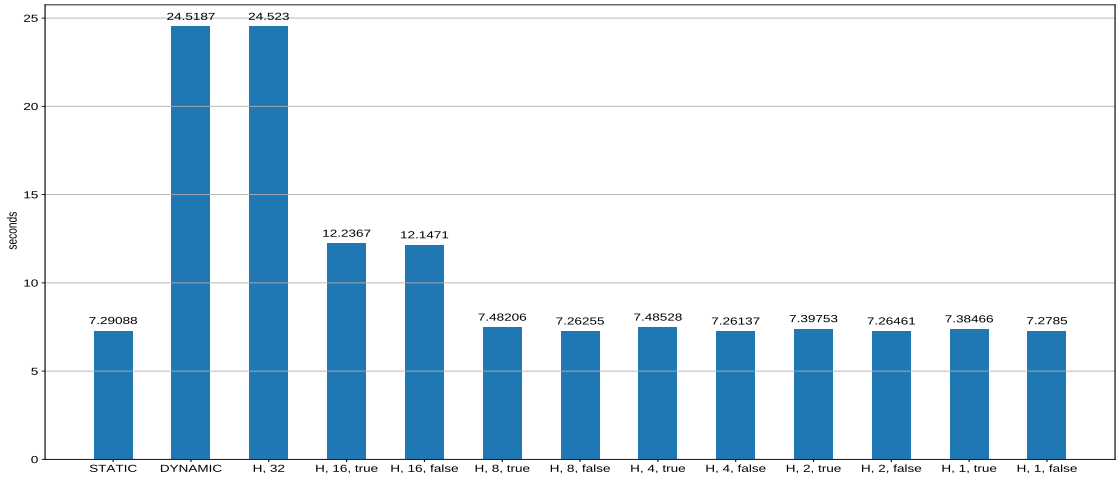
Σχήμα 4.32: backprop



(α′) n = 2^{13}, iterations = 10, mem = 1.8GB, grain size = 64



(β′) n = 2^{15}, iterations = 1, mem = 28GB, grain size = 64



(γ′) n = 2^{16}, iterations = 1, mem = 111GB, grain size = 64

Σχήμα 4.33: srad

Κεφάλαιο 5

Συμπεράσματα και Μελλοντική Έρευνα

Στην εργασία αυτή αρχικά αναλύσαμε τα ήδη υπάρχοντα schedules της βιβλιοθήκης OpenMP. Παρατηρήσαμε ότι στο dynamic schedule, κάθε προσπάθεια κατανομής των δεδομένων στα threads είναι ανώφελη, διότι η αντιστοίχιση της εκτέλεση των επαναλήψεων των παράλληλων βρόχων σε αυτά είναι τυχαία. Από την άλλη, στο static schedule δεν εξασφαλίζεται η ισοκατανομή της εργασίας. Ως συνέπεια των παρατηρήσεων αυτών, προτείναμε την επέκταση του OpenMP με ένα νέο schedule, το hierarchical, στο οποίο τα threads οργανώνονται σε groups. Δείξαμε ότι, με τον αρχικό διαμοιρασμό των επαναλήψεων στα groups, είτε σε συνεχόμενα πεδία είτε σε πεδία οριζόμενα από το χρήστη, αλλά και με την ανακατανομή της εργασίας μεταξύ τους μέσω του hierarchical stealing, μας δίνεται η δυνατότητα να ελαχιστοποιήσουμε τις προσπελάσεις απομακρυσμένων κόμβων μνήμης στα NUMA μηχανήματα, και να απομονώσουμε τα πεδία εργασίας κάθε group από τα υπόλοιπα, ώστε τα threads να εκμεταλλεύονται καλύτερα την τοπικότητα των δεδομένων, ενώ παράλληλα είδαμε ότι μειώνεται η επίδραση του μεγέθους του grain size στην επίδοση του προγράμματος. Στο τέλος, συγκρίναμε τα schedules παρουσιάζοντας μετρήσεις διαφόρων benchmarks, και δείξαμε ότι το hierarchical schedule καταφέρνει συχνά να ξεπεράσει τις επιδόσεις του dynamic σε unbalanced φορτία εργασίας, ενώ σε balanced φορτία καταφέρνει να φτάσει τις επιδόσεις του static.

Ως μελλοντική εργασία, θα μπορούσε να αξιολογηθεί η επίδοση του hierarchical schedule σε μεγαλύτερης κλίμακας μηχανήματα, με πιο έντονα NUMA χαρακτηριστικά. Σε τέτοια μηχανήματα έχει επίσης αξία να μελετηθεί περαιτέρω ο αλγόριθμος επιλογής του κατάλληλου θύματος κατά το hierarchical stealing. Όπως αναφέραμε και στην ενότητα 3.4.3, χρησιμοποιήθηκε ένα σύστημα πόντων και μια διαδικασία βαθμολόγησης σε στάδια, ώστε να είναι εύκολη η εισαγωγή επιπλέον κριτηρίων επιλογής στο μέλλον. Αντίστοιχα, θα μπορούσε να βελτιωθεί η ενσωμάτωση στον κώδικα του gcc και να υλοποιηθούν παραλήψεις στη λειτουργικότητα, όπως για παράδειγμα

η υποστήριξη του nested parallelism. Τέλος, θα είχε ενδιαφέρον να επιλεγεί ένα πρόγραμμα αντίστοιχο του polymer, το οποίο δηλαδή ελέγχει άμεσα τα threads με πιο χαμηλού επιπέδου βιβλιοθήκες (π.χ. pthreads), και να υλοποιηθεί μέσω του hierarchical schedule στο OpenMP, ώστε να συγκριθούν οι διαφορές στην επίδοση, αλλά και στην ευκολία υλοποίησης.

Βιβλιογραφία

[1] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 4.5*. Nov. 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[2] Scott Beamer, Krste Asanović, and David Patterson. "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server". In: (Oct. 30, 2015). DOI: 10.1109/IISWC.2015.12. URL: https://escholarship.org/uc/item/8gd2p3qs.

[3] Shun Julian and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *ACM SIGPLAN Notices* 48 (Aug. 2013), p. 135. DOI: 10.1145/2517327.2442530. URL: https://www.researchgate.net/publication/291179593_Ligra.

[4] Sam Ainsworth and Timothy Jones. "Graph Prefetching Using Data Structure Knowledge". In: June 2016, pp. 1–11. DOI: 10.1145/2925426.2926254.

[5] Sam Ainsworth and Timothy Jones. "Software prefetching for indirect memory accesses". In: Feb. 2017, pp. 305–317. DOI: 10.1109/CGO.2017.7863749.

[6] Jiawen Sun, Hans Vandierendonck, and Dimitrios Nikolopoulos. "VEBO: a vertex- and edge-balanced ordering heuristic to load balance parallel graph processing". In: Feb. 2019, pp. 391–392. DOI: 10.1145/3293883.3295703.

[7] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, et al. "Optimizing Cache Performance for Graph Analytics". In: (Aug. 2016).

[8] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, et al. "Making caches work for graph analytics". In: Dec. 2017, pp. 293–302. DOI: 10.1109/BigData.2017.8257937.

[9] Haibo Chen, Rong Chen, and Kaiyuan Zhang. "NUMA-aware graph-structured analytics". In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* 2015 (Jan. 2015), pp. 183–193. DOI: 10.1145/2688500.2688507. URL: https://www.researchgate.net/publication/282890041.

[10] Seonmyeong Bak, Yanfei Guo, Pavan Balaji, et al. “Optimized Execution of Parallel Loops via User-Defined Scheduling Policies”. In: Aug. 2019, pp. 1–10. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337913.

[11] Marie Durand, Francois Broquedis, Thierry Gautier, et al. “An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines”. In: (Sept. 29, 2013). URL: https://hal.inria.fr/hal-00867438.