Παράλληλη Επεξεργασία

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πανεπιστήμιο Πατρών

Εργασία εξαμήνου 2015-16

Διδάσκων: Ιωάννης Ε. Βενέτης

Παράδοση: 07/06/2016

Περιεχόμενα

1 Σειριακός αλγόριθμος

Σε αυτό το σημείο, θα αξιολογηθεί η επίδοση της δοθείσας υλοποίησης του αλγορίθμου *Smith Waterman*, χρησιμοποιώντας το εργαλείο βελτιστοποίησης και ανάλυσης Scalasca[1].

1.1 Ανάλυση αλγορίθμου

Η υλοποίηση αυτή βασίζεται στην χρήση ενός δισδιάστατου πίνακα H (scoring matrix) διαστάσεων N_a, N_b (μήκη των δύο ακολουθιών εισόδου αντίστοιχα). Ο υπολογισμός των score γίνεται ανά γραμμή, ενώ χρησιμοποιούνται δύο βοηθητικοί lookup tables I_i, I_j ίδιων διαστάσεων, για την υλοποίηση της οπισθοδρόμησης. Μαζί με την εύρεση μεγίστου διακρίνονται τρεις βασικές λειτουργίες.

1.2 Μετρήσεις χρόνου και επίδοσης

Απαραίτητο για την καλύτερη ανάλυση του αλγορίθμου, είναι η αναδιάταξη του κώδικα. Οι βασικές λειτουργίες μετατρέπονται σε συναρτήσεις, ώστε να πάρουμε αντιπροσωπευτικούς χρόνους εκτέλεσης στο Scalasca. Ακόμη θα γίνει χρήση των performance counter, της βιβλιοθήκης PAPI [2].

Οι μετρητές που χρησιμοποιήθηκαν αναφέρονται σε όλα τα level των cache misses, TLB misses καθώς και στον υπολογισμό των συνολικών branches και αποτυχημένων προβλέψεων αυτών. Οι μετρήσεις πραγματοποιήθηκαν με είσοδο $N_a = 19999$ και $N_b = 19999$, χωρίς βελτιστοποιήσεις.

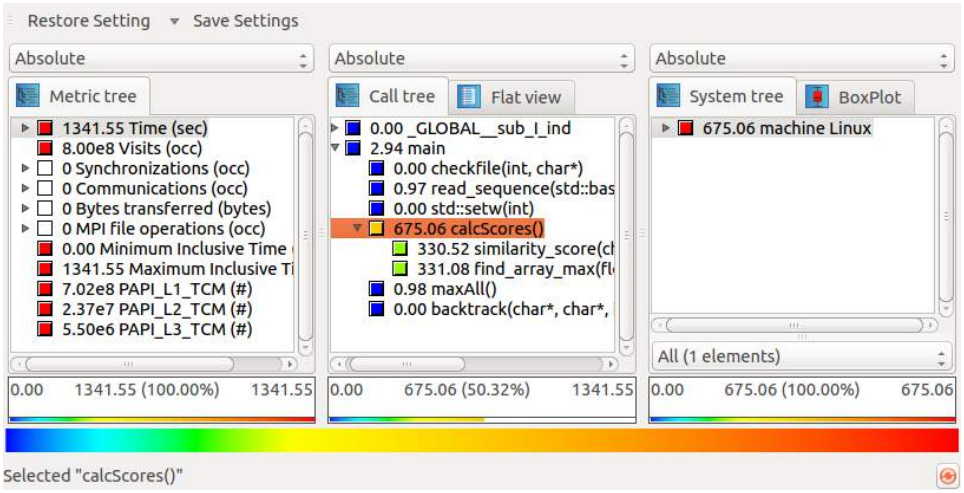
1.3 Σχολιασμός αποτελεσμάτων

Παρατηρείται πως η διαδικασία που απαιτεί τον περισσότερο χρόνο, είναι ο υπολογισμός του πίνακα *H*, σχήμα 1.1 , με μικρότερο κόστος την εύρεση μέγιστου στοιχείου, καθώς και η οπισθοδρόμηση.

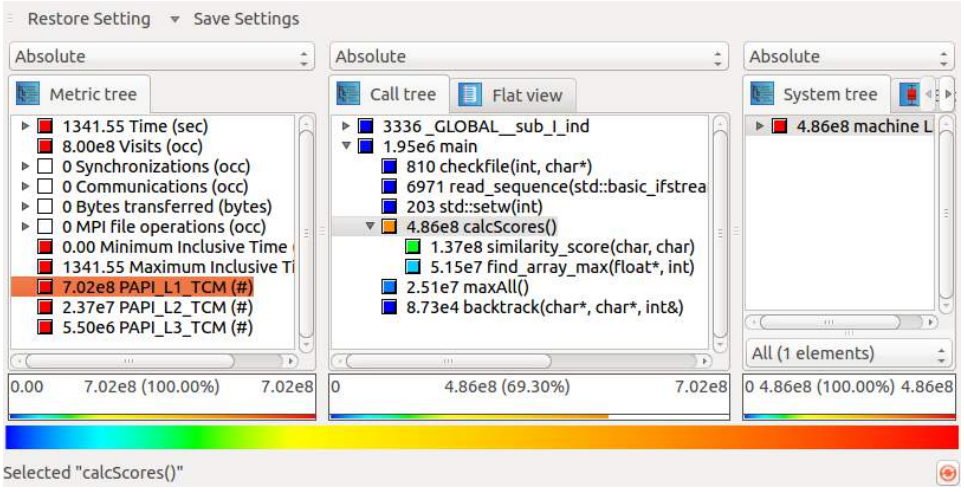
Από τους μετρητές απόδοσης, σχήματα 1.2 έως 1.4 , υπάρχει ένα σημαντικό ποσοστό cache misses με σημαντικότερο αυτών της L3-Cache, καθώς το πρόγραμμα επιβαρύνεται με την μεταφορά πληροφορίας από την αργότερη κύρια μνήμη.

Επίσης ένα σημαντικό ποσό, είναι αυτό των TLB misses, σχήμα 1.5 , το οποίο προκύπτει πάλι κατά τον υπολογισμό του *H*, όσο και από τον υπολογισμό του μέγιστου στοιχείου.

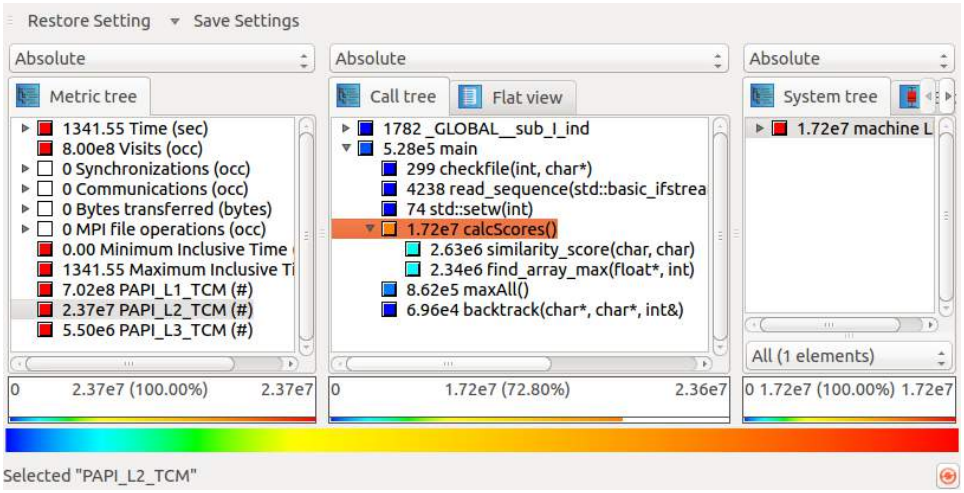
[1] Βλέπε Βιβλιογραφία
[2] Βλέπε Βιβλιογραφία



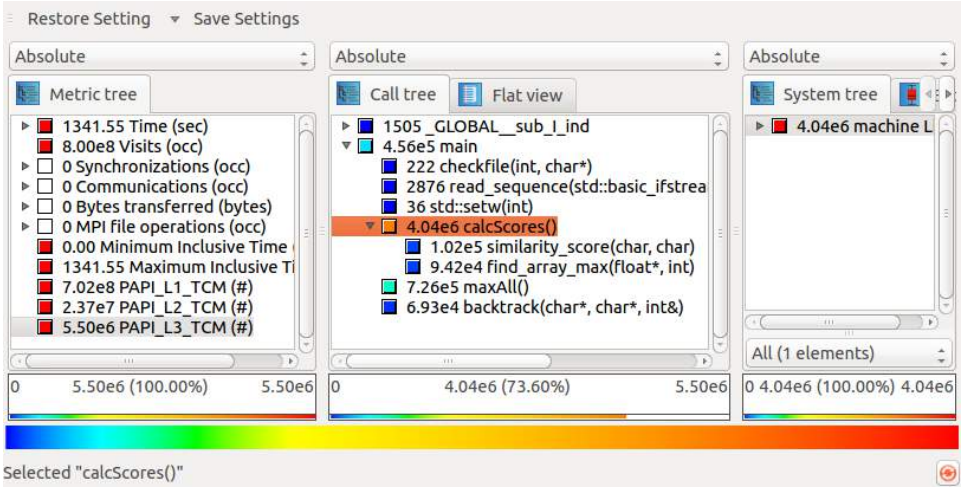
Σχήμα 1.1 : Χρόνοι εκτέλεσης



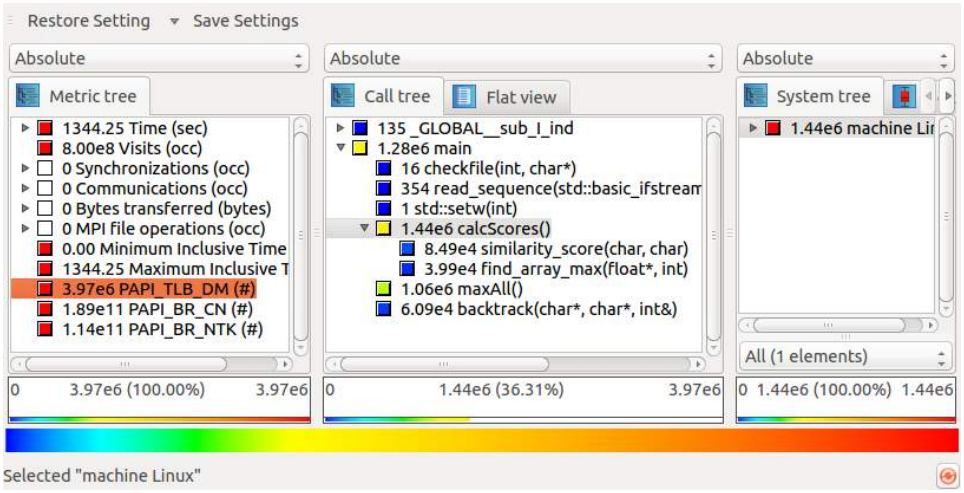
Σχήμα 1.2 : Papi L1 Cache Misses



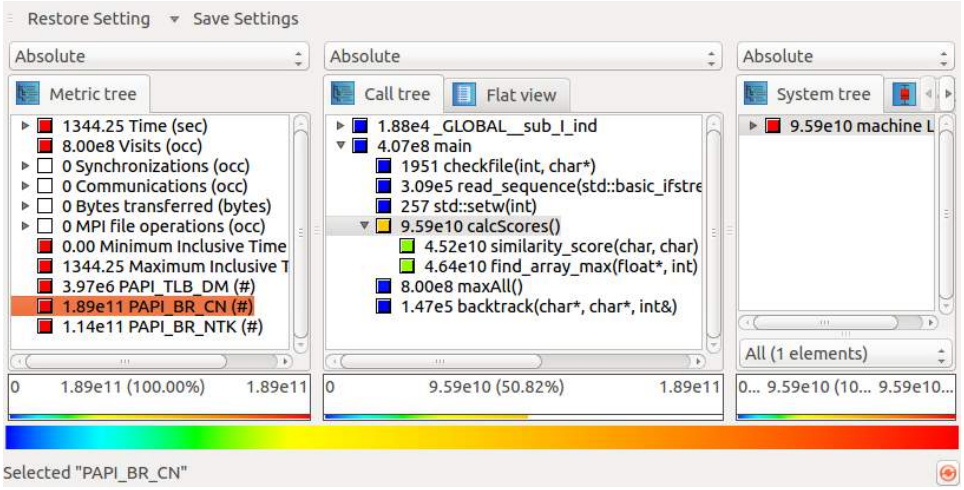
Σχήμα 1.3 : Papi L2 Cache Misses



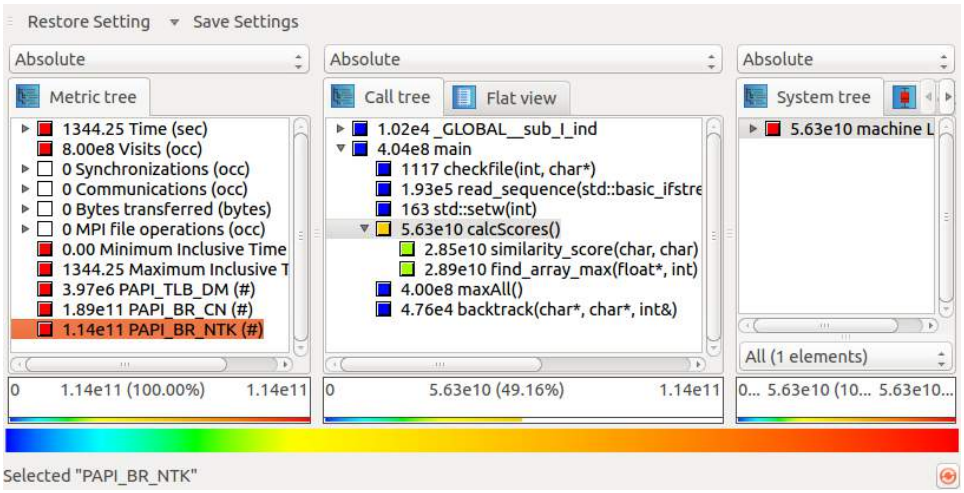
Σχήμα 1.4 : Papi L3 Cache Misses



Σχήμα 1.5 : Papi TLB Misses



Σχήμα 1.6 : Papi Branches Count



Σχήμα 1.7 : Papi Branches Not Taken

Ένα μεγάλο πρόβλημα που οδηγεί στα παραπάνω αποτελέσματα είναι πως η υλοποίηση αυτή, δεν εκμεταλλεύεται αρκετά την τοπικότητα των αναφορών. Όταν γίνεται ο υπολογισμός ανά γραμμή, για μεγάλο πλήθος στηλών, η cache δεν μπορεί να διατηρήσει πληροφορία που έχει μεταφερθεί ήδη. Έτσι υπάρχει πολύ μικρή επαναχρησιμοποίηση, που επιδεινώνεται όταν υπάρχουν πολλαπλές μεταφορές μεγάλων δομών (π.χ. η χρήση των δύο lookup tables).

Ακόμη μία σημαντική καθυστέρηση στον κώδικα είναι οι διακλαδώσεις. Οι σύγχρονοι επεξεργαστές κάνουν pipeline πολλαπλές εντολές, αλλά όταν πρόκειται για διακλάδωση, δηλαδή όταν η σειρά εκτέλεσης αυτών δεν είναι γνωστή, υπάρχουν προβλήματα. Η χρήση μεθόδων πρόβλεψης, ώστε να βρεθεί η σειρά εκτέλεσης, έχει μεγάλο λόγο επιτυχίας (π.χ.πρόβλεψη πλήθους επαναλήψεων), αλλά αποτυγχάνει όταν η διακλάδωση εξαρτάται από τυχαία δεδομένα(είσοδο). Πολλαπλές λάθος προβλέψεις κοστίζουν, καθώς οι εντολές της λανθασμένης πρόβλεψης αντικαθιστώνται με τις σωστές.

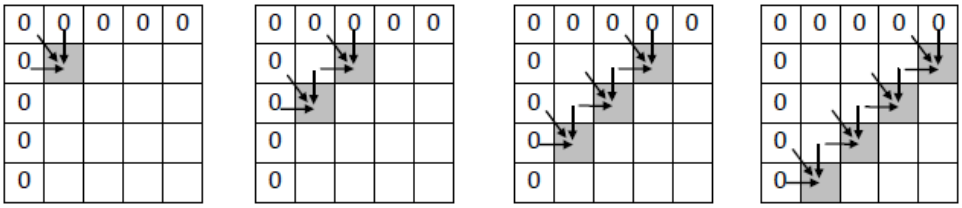
Η υλοποίηση αυτή χρησιμοποιεί μεγάλο πλήθος τέτοιων διακλαδώσεων, αρχικά για τον υπολογισμό του μέγιστου score που προκύπτει σε κάθε θέση του πίνακα H, αλλά και για αποθήκευση πληροφορίας εξαιρούμενης από αυτήν (lookup tables). Αρετές λανθασμένες προβλέψεις, σχήμα 1.7 , έχουν και οι υπόλοιπες δύο λειτουργίες και συγκεκριμένα η *maxAll*, αλλά δεν μπορεί να γίνει κάτι για αυτές.

2 Παραλληλοποίηση με OpenMP

2.1 Αρχικές Κατευθύνσεις

Αρχικά αφαιρέθηκαν οι lookup tables I_i, I_j και αντικαταστάθηκαν με ένα νέο δισδιάστατο πίνακα $charP$. Η αλλαγή αυτή βελτίωσε τον χρόνο εκτέλεσης της *calc Scores*, αλλά βελτίωσε και τον χρόνο της *backtrack*. Στην συνέχεια για λόγους που θα αναφέρουμε στο κεφάλαιο της διανυσματοποίησης, αφαιρέθηκε εντελώς ο P, επιβαρύνοντας χρονικά την οπισθοδρόμηση, αλλά επιταχύνοντας τον υπόλοιπο αλγόριθμο.

Μία μέθοδος που χρησιμοποιείται για την παραλληλοποίηση του *Smith Waterman*, είναι αυτή της κυματομορφής(wave front). Η τελευταία υπολογίζει τα κελιά του πίνακα σε αντιδιαγωνιούς, απαλείφοντας έτσι τις εξαρτήσεις στην σειρά υπολογισμού. Η μέθοδος αυτή υλοποιείται εύκολα με νήματα, αλλά το κόστος υπολογισμού ξεπερνά την σειριακή υλοποίηση, λόγο κακής τοπικότητας αναφορών, συν τον επιπρόσθετο κόστος συγχρονισμού των νημάτων.



Σχήμα 2.8 : Αλγόριθμος wave front

Μία γνωστή λύση βελτίωσης της τοπικότητας των αναφορών, είναι αυτή της πλοκαδοποίησης(blocking). Η σειρά υπολογισμού των block, είναι κρίσιμη καθώς κρύβει εξαρτήσεις δεδομένων σύμφωνα με τον αλγόριθμο των *Smith Waterman*.

Από τα παραπάνω προκύπτει πως ένας συνδυασμός των δύο μεθόδων, απαλείφει τις εξαρτήσεις μεταξύ των δεδομένων, καθώς και θα εκμεταλλεύεται όσο το δυνατόν περισσότερο την τοπικότητα των αναφορών, υπολογίζοντας μικρά blocks του H κάθε φορά.

Ακόμη, τονίζεται πως το backtracking είναι η μόνη λειτουργία στον αλγόριθμο που δεν μπορεί να παραληλοποιηθεί, καθώς η σειρά εκτέλεσης της είναι προκαθορισμένη (ordered), ενώ τέλος η παραλληλοποίηση της διαδικασίας *maxAll*, είναι τετριμμένη.

2.2 Υλοποίηση Wafe Front

Ο αλγόριθμος που σχεδιάστηκε ακολουθεί το μοντέλου του παραγωγού καταναλωτή, το οποίο επιτυγχάνεται μέσω των OpenMP tasks. Ένα νήμα επιφορτίζεται με την δημιουργία των απαραίτητων tasks, ενώ όλα τα νήματα τα υπολογίζουν. Το σχετικό απόσπασμα παρατίθενται παρακάτω:

```
1  #pragma omp single
2  {
3    /**
4     * Calculate optimal block width
5     * height, and numOfDiags
6    **/
7
8    // Task creation loop
9    for (int i = 0; i < numOfDiags; ++i) {
10     tmp = lower;
11     while (tmp.row >= upper.row && tmp.col <= upper.col) {
12       // Create a new Task
13       #pragma omp task firstprivate(tmp)
14       computeBlock(tmp, N_a, N_b);
15       --tmp.row; ++tmp.col;
16     }
17     #pragma omp taskwait
18     // Execute the increments only when tasks have finished
19     if (upper.col < newCols - 1) ++upper.col; else ++upper.
           ↪ row;
20     if (lower.row < newRows - 1) ++lower.row; else ++lower.
           ↪ col;
21   }
22 }
23 /* Threads wait here at the implied barrier while they
24    perform all the available tasks. */
```

Listing 1: Δημιουργία έργων

Το Listing 1 είναι ένα αφαιρετικό απόσπασμα του τι επιτελεί ο παραγωγός και ο καταναλωτής. Το πρώτο νήμα που θα εισέλθει στο single block, είναι και ο "παραγωγός" όλων των task. Τα υπόλοιπα νήματα θα περιμένουν στο τέλος του single block (implied barrier). Όταν υπολογιστεί το ιδανικό μέγεθος block, ο παραγωγός θα αρχίσει να δημιουργεί νέα task, τα οποία καλούν την συνάρτηση *computeBlock* με όρισμα την θέση του block.

Όταν ο παραγωγός ολοκληρώσει την while, έχουν δημιουργηθεί όλα τα block στην αντιδιαγώνιο i, αλλά πιθανώς δεν έχουν υπολογιστεί. Για τον λόγο αυτό χρησιμοποιείται η εντολή *taskwait*, ώστε το νήμα του παραγωγού να μην προχωρήσει στον

υπολογισμό της επόμενης αντιδιαγωνίου, μέχρι να εκτελεστούν όλα τα task.

Σε αυτό το σημείο αξίζει να αναφέρουμε πως ο ρόλος παραγωγού καταναλωτή είναι αλληλένδετος. Δηλαδή ο καταναλωτής στο φράγμα της taskwait, θα κληθεί να υπολογίσει κάποιο διαθέσιμο task από την ουρά. Το ίδιο επιτελούν και οι καταναλωτές, στο φράγμα της single. Ο συγχρονισμός αυτός είναι αρκετά απλός και αποδίδει αρκετά καλύτερα από το built in task dependency manager που διαθέτει η OpenMP με τις νέες εκδόσεις.

2.3 Υπολογισμός block - Οπισθοδρόμηση

Παρακάτω παρατίθενται η συνάρτηση *computeBlock*, η οποία χρησιμοποιεί την *find_max_score* αποθηκεύοντας το αποτέλεσμα σύγκρισης, χωρίς να διατηρείται το index του μέγιστου στοιχείου. Επίσης βρέθηκε, μετά από δοκιμές, πως η χρήση της μεταβλητής *tp* επιταχύνει την εκτέλεση.

```
1  void computeBlock(Point block, int Rows, int Cols) {
2
3    const int row = block.row * Bh,
4          col = block.col * Bw;
5    int i, j;
6
7    Rows = MIN(Rows, row + Bh);
8    Cols = MIN(Cols, col + Bw);
9
10   float tp, tmp[3];
11   for (i = row + 1; i <= Rows; ++i)
12   {
13     tp = H[i][col];
14     for (j = col + 1; j <= Cols; ++j)
15     {
16       tmp[0] = H[i - 1][j - 1] +
17       similarity_score(seq_a[i - 1], seq_b[j - 1]);
18       tmp[1] = H[i - 1][j] - delta;
19       tmp[2] = tp - delta;
20
21       // Compute action that produces maximum score
22       H[i][j] = tp = find_max_score(tmp);
23     }
24   }
25 }
```

Listing 2: Υπολογισμός Block

Ακόμη παρατίθενται ο κώδικας της οπισθοδρόμησης, με όλες τις απαραίτητες αλλαγές, καθώς και η βοηθητική συνάρτηση *movePoint* που αναπτύχθηκε.

```
1  Point curr = { max.p.row, max.p.col },
2      next = movePoint(curr);
3
4  int tick = 0;
5
6  char consensus_a[N_a + N_b + 2],
7     consensus_b[N_a + N_b + 2];
8
9  // Backtracking from H_max
10 while (((curr.row != next.row) || (curr.col != next.col))
11     && (next.row != 0) && (next.col != 0)) {
12
13   if (next.row == curr.row) {
14     // deletion in A
15     consensus_a[tick] = '-';
16   } else {
17     // match/mismatch in A
18     consensus_a[tick] = seq_a[curr.row - 1];
19   }
20
21   if (next.col == curr.col) {
22     // deletion in B
23     consensus_b[tick] = '-';
24   } else {
25     // match/mismatch in B
26     consensus_b[tick] = seq_b[curr.col - 1];
27   }
28
29   curr = next;
30   next = movePoint(next);
31   ++tick;
32 }
```

Listing 3: Οπισθοδρόμηση

```
1  Point movePoint(Point old) {
2
3    int row = old.row,
4      col = old.col;
5
6    float Val = H[row][col];
7
8    if (Val == H[row - 1][col - 1] +
9      similarity_score(seq_a[row - 1], seq_b[col - 1])) {
10     --old.row;
11     --old.col;
12   }
13   else if (Val == H[row - 1][col] - delta) {
14     --old.row;
```

```
15    }
16    else if (Val == H[row][col - 1] - delta) {
17      --old.col;
18    }
19    return old;
20  }
```

Listing 4: Βοηθητική συνάρτηση movePoint

Η συνάρτηση *movePoint*, υπολογίζει από πού προήλθε το score του κάθε κελιού. Είναι ξεκάθαρο πως αυτό αποτελεί πλεονασμό, αφού θα μπορούσε να αποθηκευτεί μέσω της *computeBlock*. Η γενική περίπτωση όμως δεν είναι κακιά, καθώς η εύρεση του μεγίστου γίνεται πιο γρήγορη, εξοικονομείται μνήμη και κύκλοι για την προσπέλαση-αποθήκευση, ενώ συνήθως το μέγιστο score προκύπτει από το διαγώνιου στοιχείο, άρα τερματίζεται και στον πρώτο έλεγχο.

2.4 Εύρεση μέγιστου score

Η παραλληλοποίηση της διαδικασίας έγινε με χρήση των OpenMP Reductions. Το σχετικό απόσπασμα παρατίθενται παρακάτω:

```
1  // Create comparator container
2  struct Comparator { float val; Point p; } max;
3
4  // Declare our reduction
5  #pragma omp declare reduction(max : struct Comparator :
       ↪ omp_out = (omp_in.val > omp_out.val || (omp_in.val ==
       ↪ omp_out.val && (omp_in.p.row < omp_out.p.row ||
       ↪ omp_in.p.col < omp_out.p.col))) ? omp_in : omp_out)
6
7  // Set max value as -Inf
8  max.val = -numeric_limits<float>::infinity();
9
10 // Compute reduction
11 #pragma omp for reduction(max: max)
12 for (int i = 1; i <= N_a; ++i) {
13   for (int j = 1; j <= N_b; ++j) {
14     if (H[i][j] > max.val) {
15       max.val = H[i][j];
16       max.p.row = i;
17       max.p.col = j;
18     }
19   }
20 }
```

Listing 5: Εύρεση Μέγιστου

Για την υλοποίηση αυτή, χρειάζεται να οριστεί ένα νέο reduction, το οποίο αποθηκεύει επιπλέον την θέση της μέγιστης τιμής στον *H*. Στο Listing 5 δηλώνεται το container *max* που περιέχει την μέγιστη τιμή καθώς και την θέση της. Ο αλγόριθμος εύρεσης της μέγιστης τιμής μεταβάλλεται ελάχιστα, ενώ θέτουμε αρχική τιμή της μεταβλητής το μείον άπειρο.

Ουσιαστικά το reduction υλοποιεί τον συγκριτή των μερικών αποτελεσμάτων, που διαθέτει το κάθε νήμα, κατά την φάση σύνθεσης του τελικού αποτελέσματος. Ο έλεγχος γίνεται μεταξύ δύο τιμών και υπολογίζεται το τελικό αποτέλεσμα ως το μέγιστο αυτών.

Η παραπάνω λειτουργία εκτελείται με οποιαδήποτε σειρά. Αυτό επιφέρει προβλήματα όταν η μέγιστη τιμή συναντάται πολλές φορές, καθώς μπορεί να επιλεγεί οποιοδήποτε κάθε φορά(διαφορετικές θέσεις). Για τον λόγο αυτό, σε περίπτωση ισότητας, η μέγιστη τιμή θεωρείται αυτή με το μικρότερο δείκτη γραμμής ή και στήλης (αν οι γραμμές είναι ίσες). Ο τελευταίος έλεγχος δεν είναι απαραίτητος αλλά κρατήθηκε για ασφάλεια, για οποιοδήποτε τρόπο προσπέλασης.

2.5 Διαμοιρασμός φόρτου εργασίας

Στο σημείο αυτό είναι κατανοητό πως η επιλογή κατάλληλων διαστάσεων στα block είναι καθοριστική για την καλύτερη κατανομή φόρτου στα νήματα.

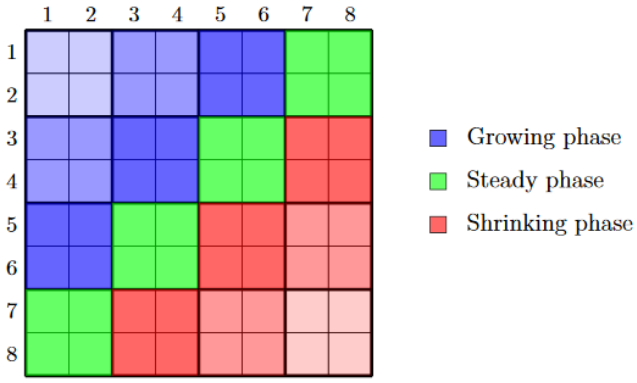
2.5.1 Θεωρητικό υπόβαθρο

Το μέγεθος υπολογισμού εξαρτάται από τις διαστάσεις του *H*, αλλά και το πλήθος των νημάτων. Διαισθητικά πολλά νήματα πρέπει να υπολογίζουν μικρότερα block, ενώ λίγα νήματα μεγαλύτερα, ώστε να εκμεταλλευόμαστε πλήρως όλους τους διαθέσιμους πόρους, χωρίς πάντα το κόστος συγχρονισμού-δημιουργίας νημάτων να επιβαρύνει υπερβολικά το συνολικό χρόνο.

Επιπρόσθετα, λόγο του τρόπου αποθήκευσης των πινάκων στην C++ (row magor) μία υλοποίηση με "πλατύ" block, θα ευνοηθεί περισσότερο σε χρόνο υπολογισμού, αφού τα στοιχεία που επεξεργάζεται βρίσκονται σε διαδοχικές θέσεις στην cache. Επίσης κρίνεται επιθυμητό το πλάτος του block να είναι κάποιο πολλαπλάσιο της γραμμής της cache(64B) γιατί με σωστή ευθυγράμμιση των στοιχείων, θα ελαχιστοποιηθούν τα cache misses.

2.5.2 Μετρήσεις

Στο σχήμα 2.9 απεικονίζονται οι φάσεις υπολογισμού στον H, με διαστάσεις block 2x2 για $N = 4$ νήματα. Παρατηρούνται οι φάσεις ανάπτυξης και η συμμετρική φάση συρρίκνωσης, όπου δεν εκτελούνται όλα τα νήματα λόγο εξαρτήσεων, καθώς και την σταθερή φάση, όπου έχουμε μέγιστη παραλληλοποίηση.



Σχήμα 2.9 : Φάσεις Υπολογισμού πίνακα H με N=4

Ο χρόνος υπολογισμού ενός block ορίζεται από την σχέση 2.5.1, όπου B_w, B_h είναι το πλάτος και ύψος του, ενώ t_{cc}, ο χρόνος υπολογισμού ενός στοιχείου του H. Δηλαδή ο συνολικός χρόνος εκτέλεσης στις φάσεις ανάπτυξης και συρρίκνωσης για N νήματα, δίνεται από την εξίσωση 2.5.2.

$$T_{bc} = B_w * B_h * t_{cc} \tag{2.5.1}$$

$$T_{sf} = 2(N-1)T_{bc} \tag{2.5.2}$$

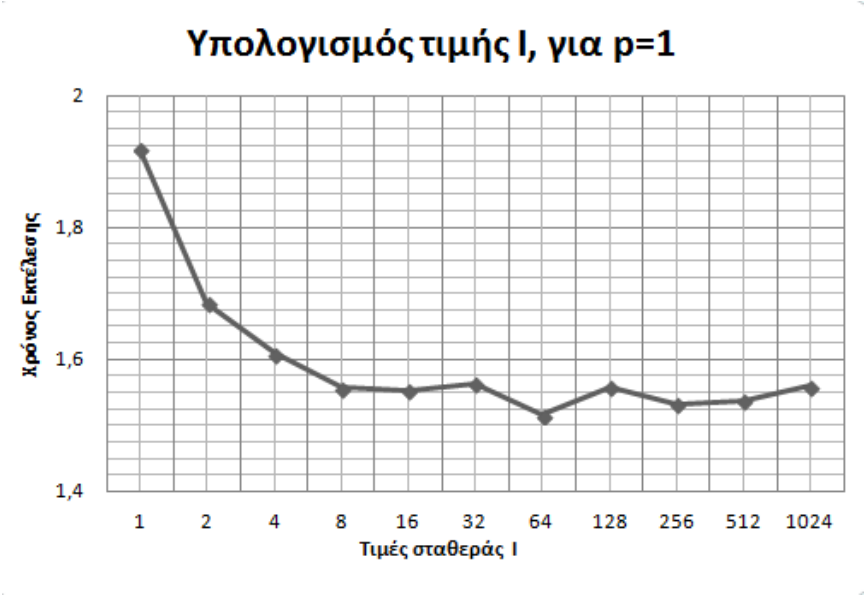
$$T_{all} = (N_a/B_h + N_b/B_w - 1)T_{bc} \tag{2.5.3}$$

Ο συνολικός χρόνος εκτέλεσης ισούται με T_{all} και περιγράφεται στην εξίσωση 2.5.3. Οι εξισώσεις επαληθεύουν πως επιθυμούμε λιγοστό χρόνο στις φάσεις ανάπτυξης και συρρίκνωσης, ώστε να μεταφερθούμε γρήγορα στην σταθερή φάση. Βέβαια μικραίνοντας τις διαστάσεις των block αυξάνεται και ο συνολικός χρόνος εκτέλεσης.

Ένα τέτοιο πρόβλημα βελτιστοποίησης είναι δυσεπίλυτο κρύβοντας πολλαπλές εξαρτήσεις και για το λόγο αυτό επιλέχθηκε να πραγματοποιηθούν πειραματικές μετρήσεις με σκοπό την εύρεση του καλύτερου μεγέθους block στο σύστημά μας.

$$\begin{aligned} B_w &= N_b/(N * p) \\ B_h &= N_a/(N * I) \end{aligned} \tag{2.5.4}$$

Για τις μετρήσεις, έγινε παραμετροποίηση των διαστάσεων block, σχέση 2.5.4. Οι μετρήσεις έγιναν με $N = 4$ νήματα και για εισόδους $N_a = 19999$ και $N_b = 19999$. Αρχικά με σταθερό $p = 1$ και μεταβλητό I, ώστε να μειωθεί το ύψος των block. Τέλος για σταθερό I βρέθηκε η βέλτιστη τιμή του p. Κάθε μέτρηση επαναλήφθηκε πέντε φορές και υπολογίστηκε ο μέσος όρος αυτών.



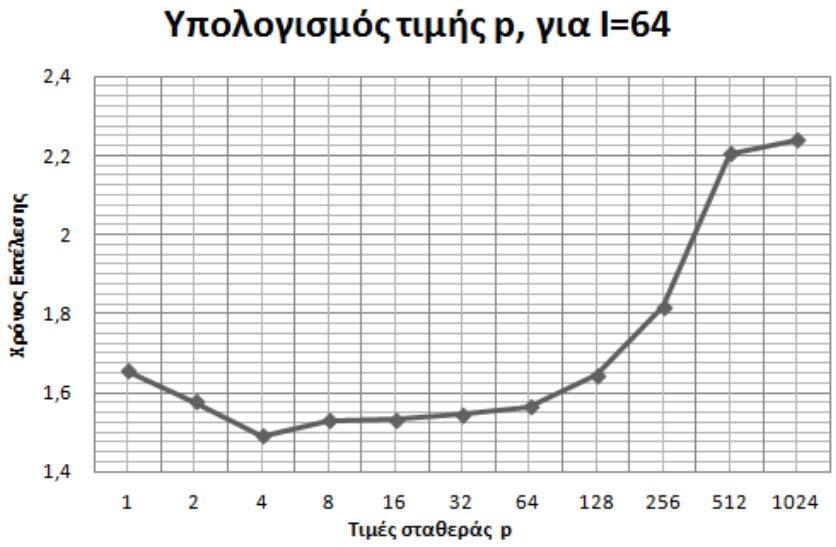
Σχήμα 2.10 : Υπολογισμός βέλτιστου I, για p=1

2.5.3 Περιγραφή αποτελεσμάτων

Τα βέλτιστα αποτελέσματα ισούνται με $I = 64$ και $p = 4$, επαληθεύοντας το θεωρητικό $p < I$. Δηλαδή τα block έχουν μεγαλύτερο "πλάτος" από ότι "ύψος" (επιθυμητό χαρακτηριστικό).

2.6 Μετρήσεις - Αποτελέσματα

Η κάθε μέτρηση επαναλήφθηκε πέντε φορές και υπολογίστηκε ο μέσος όρος αυτών. Τα αποτελέσματα παρατίθενται παρακάτω:



Σχήμα 2.11 : Υπολογισμός βέλτιστου p, για I=64

Threads	1	2	4
-O0	11,9433 s	6,4842 s	5,2118 s
-O3	3,1898 s	1,6809 s	1,3759 s

Πίνακας 1: Χρόνοι εκτέλεσης με OpenMP

-O0	-O3
13.7237 s	5.0432 s

Πίνακας 2: Χρόνοι εκτέλεσης σειριακού κώδικα

3 Παραλληλοποίηση με OpenMP + Vectorization

3.1 Αρχικές Κατευθύνσεις

Αρχικά έγινε προσπάθεια να διανυσματοποιηθεί ο κώδικας της *computeBlock* με την εντολή simd της OpenMP. Κάτι τέτοιο ήταν αδύνατο, λόγο των εξαρτήσεων στα δεδομένα. Για τον λόγο αυτό, η διανυσματοποίηση έγινε χειροκίνητα με την χρήση SSE Intrinsics[3].

Η φιλοσοφία του simd βασίζεται στις όμοιες πράξεις σε πολλαπλά δεδομένα (single instruction multiple data). Η τεχνολογία του επεξεργαστών μας, επιτρέπει εντολές AVX με τελούμενα μήκους 256bit. Δηλαδή μπορούν να εκτελεστούν μέχρι οκτώ πράξεις στα δεδομένα του H(float = 4B) ταυτόχρονα.

Πραγματοποιώντας loop-unrolling στην εσωτερική for της *computeBlock*, γίνεται εύκολα η ομαδοποίηση των πράξεων, ενώ στο τέλος μπορούν να αντιμετωπιστούν οι εξαρτήσεις, όπου και υπάρχουν.

3.2 Υλοποίηση Διανυσματοποίησης

Στο σημείο αυτό θα αναλυθεί ο κώδικας της διανυσματοποίησης. Όπως αναφέρθηκε και παραπάνω οι πράξεις θα εκτελεστούν με μέγεθος τελουμένων 256bit, μέσω του τύπου $\_\_m256$(διάνυσμα float οκτώ θέσεων).

```
1  // define max score container
2  float __attribute__((aligned(32))) max_mem[8];
3  float constants[] = { 1.f, -mu, delta, 0.f };
4
5  __m256 seqa, seqb, mask, diag, up, max,
6  // matching scores vector
7  mscore = _mm256_broadcast_ss( constants ),
8  // non maching socres vector
9  nscore = _mm256_broadcast_ss(constants+1),
10 // delta vector
11 dvect = _mm256_broadcast_ss(constants+2),
12 // zero vector
13 zvect = _mm256_broadcast_ss(constants+3);
```

Listing 6: Δηλώσεις μεταβλητών

Στο listing 6, δηλώνονται οι vector μεταβλητές που θα χρησιμοποιηθούν. Κυρίως τα score που επιστρέφονται από την *similarity_score*, καθώς και τα σταθερά διανύσματα delta και μηδέν. Ο πίνακας $max\_mem[8]$ χρησιμοποιείται για διαχείριση

[3] Βλέπε βιβλιογραφία

του περιεχομένου των vector-register ατομικά. Ο πίνακας αυτός, ευθυγραμμίστηκε στο όριο των 32B που υπαγορεύουν τα τελούμενα των 256bit, έτσι ώστε η πρόσβαση να είναι γρηγορότερη.

```
1  const int unrolled  = Cols / 8 * 8,
2        remainder = Cols - Cols % 8;
3
4  for (i = row + 1; i <= Rows; ++i)
5  {
6    tp = H[i][col];
7    for (j = col + 1; j <= unrolled; j += 8)
8    {
9      /**
10      * Vectorized code goes here
11     **/
12   }
13   // Loop for the remainder
14   for (j = remainder + 1; j <= Cols; ++j)
15   {
16     tmp[0] = H[i-1][j-1] + similarity_score(seq_a[i-1], seq_b
           ↪ [j-1]);
17     tmp[1] = H[i-1][ j ] - delta;
18     tmp[2] = tp - delta;
19
20     // Compute action that produces maximum score
21     H[i][j] = tp = find_max_score(tmp);
22   }
23 }
```

Listing 7: Loop unrolling

Στο listing 7 εμφανίζεται η νέα δομή των επαναλήψεων μετά από ξεδίπλωση. Εφόσον επιτελούνται πράξεις με οκτώ τελούμενα, είναι σκόπιμο το βήμα να είναι ίδιο. Αν ο αριθμός των στηλών δεν είναι ακέραιο πολλαπλάσιο του οκτώ, τότε το υπόλοιπο των επαναλήψεων εκτελείται σειριακά.

```
1  // Load sequence a to register
2  seqa = _mm256_set1_ps(seq_a[i-1]);
3  for (j = col + 1; j <= unrolled; j += 8)
4  {
5  // Load sequence b to register
6  seqb = _mm256_load_ps(&seq_b_floats[j-1]);
7
8  // Calculate score from similarity_score condition
9  mask  = _mm256_cmp_ps(seqa, seqb, _CMP_EQ_OQ);
10 diag  = _mm256_or_ps(_mm256_and_ps(mask, mscore),
11             _mm256_andnot_ps(mask, nscore));
```

```
12
13  // Calculate diagonal, upper scores
14  diag   = _mm256_add_ps(_mm256_loadu_ps(&H[i-1][j-1]), diag);
15  up     = _mm256_sub_ps(_mm256_loadu_ps(&H[i-1][ j ]), dvect);
16
17  // Find max score beetween diagonal, upper, 0.f scores
18  max    = _mm256_max_ps(diag, _mm256_max_ps(up, zvect));
19
20  // Store max back to memory
21  _mm256_store_ps(max_mem, max);
22
23  // Dependent data max here
24  H[i][ j ] = tp = (tp - delta > max_mem[0])
25          ? tp - delta : max_mem[0];
26  H[i][j+1] = tp = (tp - delta > max_mem[1])
27          ? tp - delta : max_mem[1];
28  H[i][j+2] = tp = (tp - delta > max_mem[2])
29          ? tp - delta : max_mem[2];
30  H[i][j+3] = tp = (tp - delta > max_mem[3])
31          ? tp - delta : max_mem[3];
32  H[i][j+4] = tp = (tp - delta > max_mem[4])
33          ? tp - delta : max_mem[4];
34  H[i][j+5] = tp = (tp - delta > max_mem[5])
35          ? tp - delta : max_mem[5];
36  H[i][j+6] = tp = (tp - delta > max_mem[6])
37          ? tp - delta : max_mem[6];
38  H[i][j+7] = tp = (tp - delta > max_mem[7])
39          ? tp - delta : max_mem[7];
40  }
```

Listing 8: Vectorized code

Αρχικά στο listing 8 θέτονται οι μεταβλητές $seqa,\ seqb$, με περιεχόμενο τους χαρακτήρες $i-1$ και $j-1..j+6$ των $seq\_a,\ seq\_b$ αντίστοιχα. Η ακολουθία seq_b έχει μετατραπεί σε ένα ευθυγραμμισμένο πίνακα float, ώστε οι μετατροπές από char σε float να μην σπαταλούν χρόνο. Εκτελώντας σύγκριση των δύο διανυσμάτων, αποθηκεύεται η μάσκα, η οποία στην συνέχεια θα χρησιμοποιηθεί για τον υπολογισμό όλων των αποτελεσμάτων της *similarity_score*.

Στην συνέχεια αθροίζονται με τα στοιχεία $H[i-1][j-1..j+6]$ ώστε να υπολογιστούν τα match/missmatch score. Αντίστοιχα, προσθέτοντας το διάνυσμα delta με τις τιμές των στοιχείων $H[i-1][j..j+7]$ υπολογίζονται τα deletion score. Έπειτα τα δύο διανύσματα αυτά μαζί με το μηδενικό συγκρίνονται και η μέγιστη τιμή της κάθε θέσης αποθηκεύεται στο διάνυσμα *max*.

Τέλος οι μέγιστες τιμές συγκρίνονται ατομικά με το insertion score, καθώς εκεί

υπάρχουν εξαρτήσεις. Στο σημείο αυτό, είναι ξεκάθαρο γιατί αφαιρέθηκαν όλες οι δομές lookup tables, καθώς δεν θα γινόταν να ωφεληθούμε από τις ταχύτητες των vectorized max, αν έπρεπε να γνωρίζουμε από ποια θέση προήλθε το αποτέλεσμα.

3.3 Ευθυγράμμιση και διανυσματοποίηση

Πολλές φορές αναφέρθηκε η ευθυγράμμιση στοιχείων με βάση την γραμμή της chache(64B bound) είτε ευθυγράμμιση με βάση τους vector register(32B bound). Θεωρητικά οι προσπελάσεις σε μη ευθυγραμμισμένα στοιχεία σπαταλούν περισσότερους υπολογιστικούς κύκλους από τα ευθυγραμμισμένα, αλλά στα σύγχρονα επεξεργαστικά συστήματα η διαφορά μειώνεται συνεχώς. Ακόμη ο πίνακας H δεν μπορεί να ευθυγραμμιστεί για όλες τις προσπελάσεις, έτσι τελικά προτιμήθηκε η ευθυγράμμιση του H στα 64B, ενώ του βοηθητικού πίνακα *seq_b_floats* στα 32B.

3.4 Μετρήσεις - Αποτελέσματα

Η κάθε μέτρηση επαναλήφθηκε πέντε φορές και υπολογίστηκε ο μέσος όρος αυτών. Τα αποτελέσματα παρατίθενται παρακάτω:

Threads	1	2	4
-O0	4,6859 s	2,5375 s	2,0348 s
-O3	1,4511 s	0,8156 s	0,6195 s

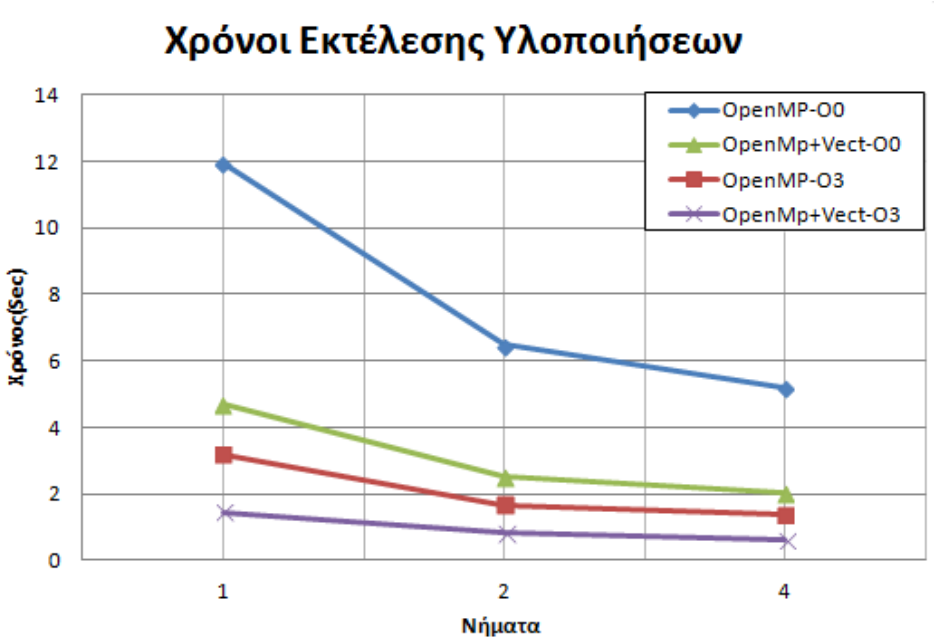
Πίνακας 3: Χρόνοι εκτέλεσης με OpenMp+Vectorization

Σημειώνεται πως με μεταβολή του λόγου των I, p, π.χ. για $I = 128$, $p = 2$, η υλοποίηση του vectorization μπορεί να επιτύχει ακόμη καλύτερους χρόνους καθώς υπάρχουν περισσότερες επαναλήψεις σε στήλες για να κάνει unroll άρα και να βελτιστοποιήσει, περαιτέρω τους χρόνους.

4 Σύγκριση Υλοποιήσεων

4.1 Απεικόνιση αποτελεσμάτων

Στον πίνακα 4.1, εμφανίζονται συγκεντρωτικά τα αποτελέσματα της επιτάχυνσης όλων των υλοποιήσεων έναντι της σειριακής, πίνακας 2.6.

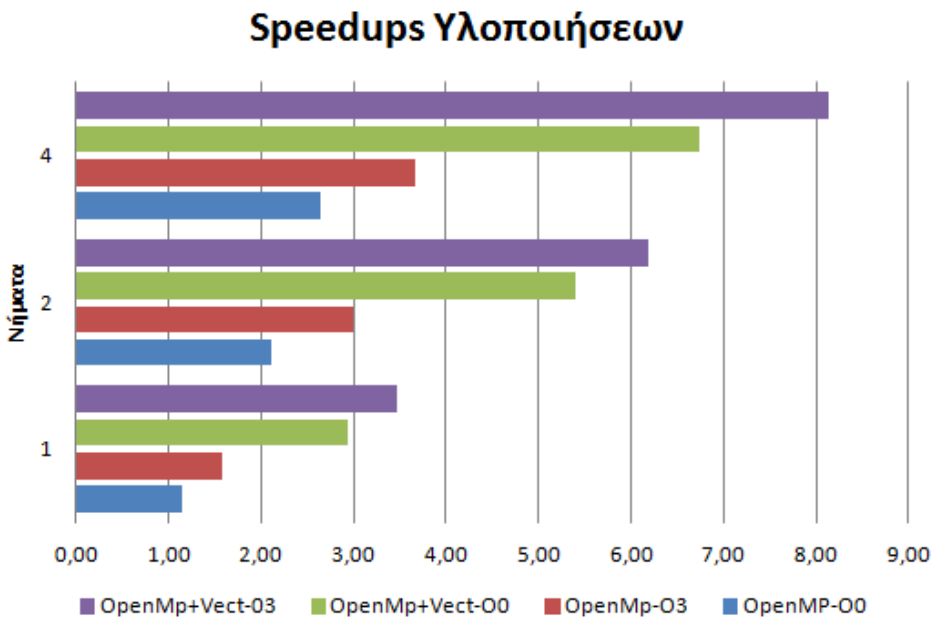


Σχήμα 4.12 : Χρόνοι εκτέλεσης υλοποιήσεων

Algorithm / Threads	1	2	4
OpenMp-O0	1,15	2,12	2,63
OpenMp-O3	1,58	3,00	3,67
OpenMp+Vect-O0	2,93	5,41	6,74
OpenMp+Vect-O3	3,48	6,18	8,14

Πίνακας 4: Speedups Υλοποιήσεων

Παρατηρείται πως η υλοποίηση OpenMp-O0, για $N = 1$ νήματα, έχει πολύ μικρή χρονοβελτίωση σε σχέση με την αρχική. Το κόστος δημιουργίας και εκτέλεσης των task, επιφέρει επιπρόσθετο βάρος στην εκτέλεση. Βέβαια η μείωση του αποθηκευτικού χώρου και η πλοκαδοποίηση ωφελεί και πάλι τον χρόνο εκτέλεσης.



Σχήμα 4.13 : Speedups υλοποιήσεων

Επίσης η υλοποίση OpenMp+Vect-O0, με $N = 1$, προσεγγίζει την βελτιστοποιημένη OpenMp-03, με $N = 4$. Αυτό υποδεικνύει πόσο χρήσιμη είναι η διανυσματοποίηση στην επίτευξη καλύτερης παραλληλίας, επιτυγχάνοντας μέγιστο $speedup = 8.14$, για $N = 4$ στην OpenMp+Vect-O3.

Βιβλιογραφία

[1]. Reference: Scalasca Home Page

[2]. Reference: PAPI Home Page

[3]. Reference: Intel SSE Intrinsics Guide