Τμημα Μηχανικων Η/Υ & Πληροφορικης,
Πανεπιστημιο Πατρων

Αναφορα Εργαστηριακης Ασκησης

Εαρινο Εξαμηνο

# Παράλληλη Επεξεργασία

Μπουραντάς Κωνσταντίνος
bourantas@ceid.upatras.gr

# Περιεχόμενα



# Κατάλογος Σχημάτων

# Ευρετήριο Κώδικα

# 1 Ερώτημα 1

## 1.1 Ερώτημα α: Βελτιστοποίηση Αντιγραφών

Ξεκινάμε την βελτίωση των αντιγραφών αποθηκεύοντας τις τιμές του πίνακα u και των πράξεων που παραμένουν σταθερά σε κάθε επανάληψη έτσι ώστε να μην τα υπολογίζουμε εξ αρχής κάθε φορά που τα χρησιμοποιούμε. Επίσης , ενσωματώνουμε των έλεγχο του πίνακα μέσα στον βρόγχο i και έτσι εξ αλείφουμε έναν βρόγχο. Επίσης όλες οι αλλαγές μέσα στον βρόγχο γίνονται πάνω στον uplus και στο τέλος του χρησιμοποιούμε την εντολή memcpy() για την αντιγραφή των τιμών από τον πίνακα uplus στον πίνακα u , η οποία προσφέρει έναν πολύ πιο αποδοτικό τρόπο για την αντιγραφή θέσεων μνήμης.

Απόσπασμα Κώδικα 1: Αναδιοργάνωση αντιγραφών σειριακού προγράμματος

```
1   double sum;
2   double temp;
3   int step;
4
5   for (it = 0; it < itime; it++) {
6     for (i = 0; i < n; i++) {
7       sum = 0.0;
8       temp = u[i];
9       step = i * n;
10      uplus[i] = temp + dt * (mu - temp);
11
12      for (j = 0; j < n; j++) {
13        sum += sigma[step+ j] * (u[j] - temp);
14      }
15      uplus[i] += dt * sum / divide;
16    }
17
18      uplus[i]+= dt * (sum - semi_sum*temp) / divide;
19      // temp = uplus[i];
20
21      // temp_u[i] = uplus[i];
22      if ( uplus[i] > uth) {
23        uplus[i] = 0.0;
24
25        if (it >= ttransient) {
26          omega1[i] += 1.0;
27        }
28      }
29    }
30
31    //
32    // for (i = 0; i < n; i++) {
```

```
33                //  u[i] = uplus[i];
34                //  if (u[i] > uth) {
35                //   u[i] = 0.0;
36                //   /*
37                //   * Calculate omega's.
38                //   */
39                //   if (it >= ttransient) {
40                //    omega1[i] += 1.0;
41                //   }
42                //  }
43                // }
44
45    memcpy(u, uplus, n * sizeof *u);
```

## 1.2 Ερώτημα β: Αναδιοργάνωση Πράξεων

Στο συγκεκριμένο ερώτημα ξεκινάμε την αναδιοργάνωση των πράξεων ξεκινώντας από τον υπολογισμό της μεταβλητής sum . Όπως βλέπουμε παρακάτω στον κώδικα υπολογίζουμε μέρος του sum το οποίο παραμένει σταθερό , έξω από τους εμφωλευμένους βρόγχους και αποθηκεύουμε το αποτέλεσμα στην μεταβλητή semi_sum. Με τον τρόπο αυτό μέσα στον βρόγχο j υπολογίζουμε μόνο το ημι άθροισμα του γινομένου sigma[step + j] * u[j] και έπειτα πολλαπλασιάζουμε την semi_sum με το υ[i] το οποίο μένει σε κάθε επανάληψη σταθερό. Με τον τρόπο αυτό δεν χρειάζεται να υπολογίζουμε σε κάθε επανάληψη του βρόγχου j το σταθερό αυτό ήμι άθροισμα μειώνοντας έτσι την πολυπλοκότητα της συγκεκριμένης πράξης. Επίσης αποθηκεύουμε διάφορα γινόμενα και περιεχόμενα πινάκων τις οποίες χρησιμοποιούμε συχνά μέσα στους βρόγχους έτσι ώστε να μην χρειάζεται σε κάθε επανάληψη να τα υπολογίζουμε κάθε φορά που θα τα χρειαστούμε πχ. το γινόμενο i*n,το περιεχόμενο του πίνακα u κλπ. Τέλος αντικαθιστούμε τον βρόγχο για την αντιγραφή του πίνακα uplus στον πίνακα u με την εντολή memcpy() η οποία αποτελεί έναν πολύ πιο γρήγορο και αποτελεσματικό τρόπο για την αντιγραφή θέσεων μνήμης.

Απόσπασμα Κώδικα 2: Αναδιοργάνωση πράξεων και βελτίωση κώδικα σειριακού προγράμματος

```
1  for (i = 0; i < n;i++) {
2       semi_sum+=sigma[i];
3   }
4
5
6    for (it = 0; it < itime; it++) {
7      for (i = 0; i < n; i++) {
8        sum = 0.0;
9        temp = u[i];
10       step = i * n;
11       uplus[i] = temp + dt * (mu - temp);
```

```
12
13      for (j = 0; j < n;j++) {
14        sum += sigma[step + j] * u[j];
15      }
16
17
18      uplus[i]+= dt * (sum - semi_sum*temp) / divide;
19      // temp = uplus[i];
20
21      // temp_u[i] = uplus[i];
22      if ( uplus[i] > uth) {
23        uplus[i] = 0.0;
24
25        if (it >= ttransient) {
26          omega1[i] += 1.0;
27        }
28      }
29
30
31    }
32
33    memcpy(u, uplus, n * sizeof *u);
```

## 1.3 Ερώτημα γ: Χρήση της Βιβλιοθήκης BLAS

Προχωρώντας , θα αντικαταστήσουμε ολόκληρο τον βρόγχο που υπολογίζει την τιμή του sum με την συνάρτηση cblas_dgemv() η οποία ανήκει στην βιβλιοθήκη BLAS. Μια σύντομη περιγραφή της βιβλιοθήκης:

> Η Basic Linear Algebra Subroutine γνωστή και με την συντομογραφία BLAS είναι μια ντεφάκτο σύμβαση για δημιουργία προγραμματιστικών διεπαφών (APIs: Application Programmers interfaces) για προγραμματιστικές βιβλιοθήκες οι οποίες υλοποιούν αλγόριθμους γραμμικής άλγεβρας που συσχετίζονται με διανυσματικές πράξεις και πίνακες (π.χ. πολλαπλασιασμό πινάκων). (πηγή: Wikipedia)

Επομένως για να χρησιμοποιήσουμε την συνάρτηση , κάναμε τις παρακάτω αλλαγές:

Απόσπασμα Κώδικα 3: Χρήση του BLAS στο σειριακό πρόγραμμα

```
1 for (it = 0; it < itime; it++) {
2     cblas_dgemv(CblasRowMajor, CblasNoTrans, n, n,
3     1.0, sigma, n, u, 1, 0.0, temp_u, 1);
4
5     for (i = 0; i < n; i++) {
6       // temp = u[i];
```

```
7       step = i * n;
8       // uplus[i] = temp + dt * (mu - temp);
9
10      // sum = 0.0;
11      // for (j = 0; j < n;j++) {
12      // sum += sigma[step + j] * u[j];
13      // }
14
15      // printf("%f\n",temp_u[i] );
16
17      uplus[i] =( u[i] + dt * (mu - u[i]))+\\
18      dt * (temp_u[i] - semi_sum*u[i]) / divide;
19      // temp = uplus[i];
20
21      // temp_u[i] = uplus[i];
22      if ( uplus[i] > uth) {
23        uplus[i] = 0.0;
24
25        if (it >= ttransient) {
26          omega1[i] += 1.0;
27        }
28      }
29
30    }
```

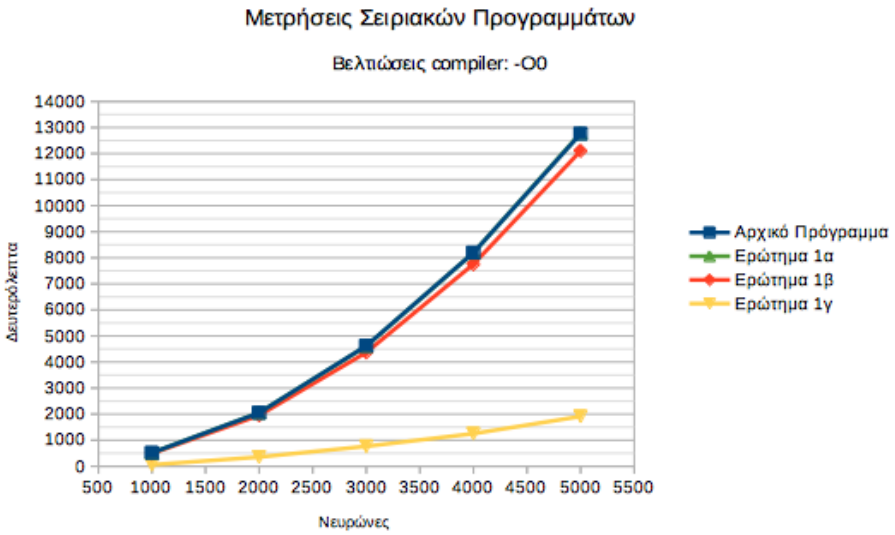
Όπως βλέπουμε υπολογίζουμε μια φορά σε κάθε επανάληψη του βρόγχου it μέσω της συγκεκριμένης συνάρτησης το άθροισμα sum για κάθε επανάληψη του βρόγχου i αποθηκεύοντας το στον πίνακα temp_sum. Έτσι μπορούμε πλέον να εξαλείψουμε τελείως τον υπολογισμό μέσα στον βρόγχου j , εξοικονομώντας σημαντικό ποσοστό χρόνο κατά την εκτέλεση του προγράμματος.

## 1.4 Μετρήσεις και Σχολιασμός Αποτελεσμάτων

Στην συνέχεια θα αναλύσουμε τα αποτελέσματα των μετρήσεων για τις παραπάνω περιπτώσεις. Οι μετρήσεις γίνανε για τους παρακάτω αριθμούς νευρώνων: 1000, 2000, 3000, 4000, 5000. Μετρήσαμε την απόδοση των αλλαγών μας καθώς και του δοθέντος προγράμματος με την χρήση βελτιώσεων κατά την μεταγλώττιση : -O0 και -O3. Ας ξεκινήσουμε με τις μετρήσεις των προγραμμάτων με την χρήση βελτιώσεων -O0:

| FOR -O0 | 1000 | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|---|
| Αρχικό Πρόγραμμα | | | | | |
| Calculations | 513.793736 | 2052.246026 | 4621.323415 | 8196.479971 | 12760.53613 |
| I/O | 0.014879 | 0.025518 | 0.039071 | 0.0482 | 0.058145 |
| Total Time | 513.808615 | 2052.271544 | 4621.362486 | 8196.528171 | 12760.59427 |
| | | | | | |
| Ερώτημα 1α | | | | | |
| Calculations | 514.185335 | 2053.282164 | 4611.978764 | 8190.999061 | 12780.83634 |
| I/O | 0.01401 | 0.02273 | 0.040081 | 0.050627 | 0.061812 |
| Total Time | 514.199345 | 2053.304894 | 4612.018845 | 8191.049688 | 12780.89815 |
| | | | | | |
| Ερώτημα 1β | | | | | |
| Calculations | 488.177452 | 1951.365854 | 4368.45198 | 7755.386595 | 12101.12411 |
| I/O | 0.013627 | 0.025807 | 0.038721 | 0.04875 | 0.062221 |
| Total Time | 488.191079 | 1951.391661 | 4368.490701 | 7755.435345 | 12101.18633 |
| | | | | | |
| Ερώτημα 1γ | | | | | |
| Calculations | 55.857532 | 353.170416 | 764.58173 | 1249.682144 | 1911.38745 |
| I/O | 0.017691 | 0.031101 | 0.044712 | 0.062394 | 0.075113 |
| Total Time | 55.875223 | 353.201517 | 764.626442 | 1249.744538 | 1911.462563 |

Σχήμα 1: Συνολικές μετρήσεις σειριακών προγραμμάτων με -O0




Σχήμα 2: Διάγραμμα χρονοβελτιώσεων σειριακών προγραμμάτων με -O0

Όπως παρατηρούμε οι βελτιώσεις των αντιγραφών μεταξύ των u και uplus προσέφεραν ελάχιστη χρονοβελτίωση και παρατηρώντας το διάγραμμα βλέπουμε ότι οι χρόνοι του αρχικού προγράμματος και το ερωτήματος α οριακά συμπίπτουν. Έπειτα με την αναδιοργάνωση των πράξεων στο ερώτημα β βλέπουμε αρκετά καλύτερους χρόνους ειδικά όταν ο αριθμός των νευρώνων αυξάνονται κάτι που είναι αναμενόμενο καθώς έχουμε αφαιρέσει ένα μέρος των υπολογισμών από τον βρόγχο j.

Τέλος βλέπουμε τεράστια βελτίωση του χρόνου όταν χρησιμοποιούμε την βιβλιοθήκη BLAS κάτι που είναι εξίσου αναμενόμενο καθώς έχουμε αφαίρεση τελείως τον βρόγχο με τον πολυπλοκότερο υπολογισμό , βρόγχο j.

| FOR -O3 | | | | | |
|---|---|---|---|---|---|
| default.c | 1000 | 2000 | 3000 | 4000 | 5000 |
| Calculations | 176.085507 | 745.962153 | 1647.859965 | 2891.843178 | 12777.706998 |
| I/O | 0.015583 | 0.02824 | 0.040488 | 0.053373 | 0.074224 |
| Total Time | 176.10109 | 745.990393 | 1647.900453 | 2891.896551 | 12777.781222 |
| | | | | | |
| lif1d_a.c | | | | | |
| Calculations | 177.303234 | 765.851278 | 1684.921095 | 2935.400329 | 4557.918281 |
| I/O | 0.014475 | 0.027631 | 0.04272 | 0.049712 | 0.05229 |
| Total Time | 177.317709 | 765.878909 | 1684.963815 | 2935.450041 | 4557.970571 |
| | | | | | |
| lif1d_ab.c | | | | | |
| Calculations | 180.119228 | 752.126271 | 1655.700104 | 2905.276871 | 4499.852374 |
| I/O | 0.017549 | 0.030269 | 0.038225 | 0.048466 | 0.057837 |
| Total Time | 180.136777 | 752.15654 | 1655.738329 | 2905.325337 | 4499.910211 |
| | | | | | |
| lif1d_c.c | | | | | |
| Calculations | 23.395758 | 313.185642 | 648.415996 | 1047.434439 | 1597.075659 |
| I/O | 0.338915 | 0.45587 | 0.667955 | 0.666899 | 0.780284 |
| Total Time | 23.734673 | 313.641512 | 649.059745 | 1048.101338 | 1597.855943 |

Σχήμα 3: Συνολικές μετρήσεις σειριακών προγραμμάτων με -O3


Σχήμα 4: Διάγραμμα χρονοβελτιώσεων σειριακών προγραμμάτων με -O3

Για τις βελτιώσεις -O3 βλέπουμε ότι το αρχικό πρόγραμμα καθυστερεί πάρα πολύ για τους 5000 νευρώνες. Επίσης παρατηρούμε σημαντική βελτίωση στους

συνολικούς χρόνους εκτέλεσης των υπόλοιπων προγραμμάτων κάτι που ήταν αναμενόμενο λόγω των βελτιώσεων του επιπέδου 3 που προσφέρει ο μεταγλωττιστής.

# 2 Ερώτημα 2

## 2.1 Ερώτημα α: Παραλληλοποίηση Προγράμματος

### 2.1.1 Παραλληλοποίηση βρόγχου i

Ξεκινώντας την παραλληλοποίηση του προγράμματος θα επιχειρήσουμε την παραλληλοποίηση του βρόγχου i. Για να το επιτύχουμε αυτό θα κάνουμε χρήση της δομής του OpenMp , #pragma omp for.

Απόσπασμα Κώδικα 4: Παραλληλοποίηση βρόγχου j

```
#pragma omp parallel private(it,i,j,sum,temp,step)
firstprivate(n,sigma,dt,mu,semi_sum)
 {
  for (it = 0; it < itime; it++) {
    #pragma omp for schedule(static,8)
    for (i = 0; i < n; i++) {
      sum = 0.0;
      #pragma omp atomic read
      temp = u[i];
      step = i * n;
      #pragma omp atomic write
      uplus[i] = temp + dt * (mu - temp);
      for (j = 0; j < n;j++) {

        sum += sigma[step + j] * u[j];
      }

      #pragma omp atomic update
      uplus[i]+= dt * (sum - semi_sum*temp) / divide;
      // temp = uplus[i];

      // // temp_u[i] = uplus[i];
      if (uplus[i] > uth) {
        #pragma omp atomic write
        uplus[i] = 0.0;

        if (it >= ttransient) {
          #pragma omp atomic
          omega1[i] += 1.0;
        }
      }

```

```
33    }
34
35   #pragma omp barrier
```

Ξεκινώντας , δημιουργούμε έξω από τους ένθετους βρόγχους την παράλληλη περιοχή και καθορίζουμε την εμβέλεια των μεταβλητών οι οποίες εμπεριέχονται σε αυτή. Έπειτα παραλληλοποιούμε τον συγκεκριμένο βρόγχο και χρησιμοποιούμε το schedule τύπου static το οποίο χωρίζει τις επαναλήψεις του βρόγχου σε ίσα μέρη και τα κατανέμει στα νήματα που έχουμε δημιουργήσει (ανά 8). Από τις δοκιμές που έγιναν η επιλογή αυτής της τακτικής ήταν η αποδοτικότερη . Έπειτα χρησιμοποιούμε ατομικές εντολές προκείμενου να εξασφαλίσουμε ότι τα νήματα δεν θα έχουν ταυτόχρονη πρόσβαση στις κοινές μεταβλητές και προκύψουν προβλήματα όπως πχ. race conditions. Στο τέλος του βρόγχου χρησιμοποιούμε την δομή barrier ώστε να εξασφαλίσουμε ότι πριν συνεχίσει η ροή του προγράμματος , όλα τα νήματα θα έχουν τελειώσει τους υπολογισμούς τους.

### 2.1.2 Παραλληλοποίηση βρόγχου j

Στην συνέχεια προχωράμε στην παραλληλοποίηση του βρόγχου j χρησιμοποιώντας την ίδια λογική με προηγουμένως. Η μόνη διαφορά είναι ότι στην αποθήκευση των τιμών στους πίνακες κάνουμε χρήση της εντολής #pragma omp single η οποία μας εξασφαλίζει ότι μόνο ένα νήμα θα εκτελέσει τις εντολές στην συγκεκριμένη περιοχή.

Απόσπασμα Κώδικα 5: Παραλληλοποίηση βρόγχου j

```
1  #pragma omp parallel private(i,j,it)
2  firstprivate(n,sigma,dt,mu,semi_sum)
3  {
4
5   for (it = 0; it < itime; it++) {
6     for (i = 0; i < n; i++) {
7       sum = 0.0;
8       temp = u[i];
9       step = i * n;
10      #pragma omp atomic write
11      uplus[i] = temp + dt * (mu - temp);
12
13      #pragma omp for firstprivate(sum)
14      for (j = 0; j < n;j++) {
15        sum = sum + sigma[step + j] * u[j];
16
17      }
18
19      #pragma omp single
20      {
21
```

```
22      #pragma omp atomic
23      uplus[i]+= dt * (sum - semi_sum*temp) / divide;
24
25      if (uplus[i] > uth) {
26        #pragma omp atomic write
27        uplus[i] = 0.0;
28
29        if (it >= ttransient) {
30          #pragma omp atomic
31          omega1[i] += 1.0;
32        }
33      }
34    }
35    }
```

### 2.1.3 Παραλληλοποίηση βρόγχου it

Η συγκεκριμένη περίπτωση παρουσιάζει ορισμένες ιδιαιτερότητες συγκριτικά με τις δύο προηγούμενες. Αυτό συμβαίνει λόγω των εξαρτήσεων που παρουσιάζουν τα δεδομένα μας . Επομένως , η παραλληλοποίηση του βρόγχου it δεν μπορεί να γίνει εύκολα με την χρήση της εντολής #pragma omp for , οπότε θα επιχειρίσουμε να κάνουμε χρήση της δομής #pragma omp tasks. Οι αλλαγές οι οποίες έγιναν είναι οι παρακάτω:

Απόσπασμα Κώδικα 6: Παραλληλοποίηση βρόγχου it

```
1   #pragma omp parallel
2   {
3     #pragma omp single nowait
4     {
5     for (it = 0; it < itime; it++) {
6       #pragma omp task private(i,j) depend(out:sum)
7       {
8         // printf("Task1: Thread : %d it: %d i: %d\n",
              ↪ omp_get_thread_num(), it, i);
9       for (i = 0; i < n; i++) {
10
11      sum = 0.0;
12
13      // #pragma omp atomic read
14      temp = u[i]*semi_sum;
15      step = i * n;
16      // #pragma omp atomic write
17      uplus[i] = u[i] + dt * (mu - u[i]);
18
19
```

```
20      for (int j = 0; j < n;j++) {
21        sum += sigma[step + j] * u[j];
22      }
23
24      // printf("sum = %f\n",sum );
25
26   // #pragma omp taskwait
27      // #pragma omp atomic update
28         sum -= temp;
29          please = sum / divide;
30        uplus[i]+= dt * please;
31
32          if (uplus[i] > uth) {
33            // #pragma omp atomic write
34            uplus[i] = 0.0;
35
36            if (it >= ttransient) {
37              // #pragma omp atomic
38              omega1[i] += 1.0;
39            }
40          }
41
42   }
43
44
45     memcpy(u, uplus, n * sizeof * u);
46    }//task1
47
48
49 #if !defined(ALL_RESULTS)
50     if (it % ntstep == 0) {
51 #endif
52      // printf("Thread: %d\n",omp_get_thread_num());
53      gettimeofday(&IO_start, NULL);
54
55      #pragma omp task private(i) depend(in:sum)
56      {
57        printf("Time is %ld\n", it);
58
59      fprintf(output1, "%ld\t", it);
60
61      for (i = 0; i < n; i++) {
62        // printf("Thread: %d\n",omp_get_thread_num());
63        // printf("write to ouput1 : %d\n",i);
64        fprintf(output1, "%19.15f", u[i]);
65      }
```

```
66      fprintf(output1, "\n");
67     }
68
69      time = (double)it * dt;
70
71      #pragma omp task private(j) depend(in:sum)
72      {
73      fprintf(output2, "%ld\t", it);
74
75      for (i = 0; i < n; i++) {
76        omega[i] = 2.0 * M_PI * omega1[i] / (time - ttransient *
              ↪ dt);
77        fprintf(output2, "%19.15f", omega[i]);
78      }
79      fprintf(output2, "\n");
80    }
```

Όπως βλέπουμε ξεκινάμε την παραλληλοποίηση δημιουργώντας 3 tasks. Το ένα task περιλαμβάνει ολόκληρο τον βρόγχο του i ενώ τα άλλα δύο περιλαμβάνουν του βρόγχους ενημέρωσης των αρχείων . Συγχρονίζοντας τα tasks μας κατάλληλα εξασφαλίζουμε ότι θα εκτελεστούν από τα νήματα με την σωστή σειρά. Τα δύο τελευταία tasks εκτελούνται ταυτόχρονα μόλις τελειώσει το πρώτο. Η συγκεκριμένη υλοποίηση δεν αποτελεί τον πιο αποδοτικό τρόπο παραλληλοποίησης λόγω της εξάρτησης μεταξύ των tasks αλλά μας προσφέρει έναν σταθερό και αρκετά λιγότερο συνολικό χρόνο εκτέλεσης.

## 2.2 Ερώτημα β: Χρήση της Βιβλιοθήκης BLAS

Από τις παραπάνω τρεις υλοποιήσεις , με βάση τις μετρήσεις που έγιναν , η πιο γρήγορη και αποδοτική ήταν η παραλληλοποίηση του βρόγχου i. Επομένως προβήκαμε στις παρακάτω αλλαγές με σκοπό να ενσωματώσουμε και την χρήση της βιβλιοθήκης BLAS για την πραγματοποίηση ορισμένων κοστοβόρων υπολογισμών.

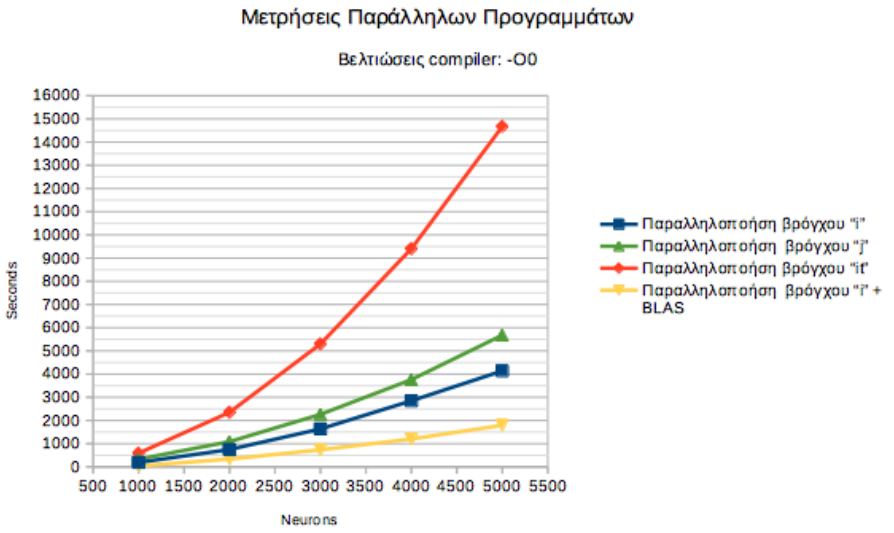
Απόσπασμα Κώδικα 7: Παραλληλοποίηση βρόγχου i και χρήση BLAS

```
1  double sum;
2  double *final_sigma = (double *)calloc(n, sizeof(double));
3  int total_threads;
4
5  #pragma omp parallel private(it,i,j,step) firstprivate(n,sigma,
       ↪ dt,mu,semi_sum)
6  {
7
8    double *temp_u;
9    int thread_id = omp_get_thread_num();
10   int array_pos,position;
11
```

```
12      total_threads = omp_get_num_threads();
13      position = (n / total_threads) * n * thread_id;
14      array_pos = (n / total_threads) * thread_id;
15      temp_u = (double *)calloc(n / total_threads, sizeof(double))
            ↪ ;
16
17    for (it = 0; it < itime; it++) {
18
19        cblas_dgemv(CblasRowMajor, CblasNoTrans, n / total_threads
              ↪ , n, 1.0, sigma + position, n, u, 1 , 0.0, temp_u,
              ↪ 1);
20        memcpy( final_sigma + array_pos, temp_u, n / total_threads
              ↪  * sizeof * final_sigma );
21
22
23      #pragma omp barrier
24
25
26      #pragma omp for schedule(static,8)
27      for (i = 0; i < n; i++) {
28        step = i * n;
29
30        #pragma omp atomic write
31        uplus[i] = ( u[i] + dt * (mu - u[i])) + dt * (final_sigma[
              ↪ i] - semi_sum * u[i]) / divide;
32        // temp = uplus[i];
33
34        // temp_u[i] = uplus[i];
35        if ( uplus[i] > uth) {
36          #pragma omp atomic write
37          uplus[i] = 0.0;
38
39          if (it >= ttransient) {
40            #pragma omp atomic
41            omega1[i] += 1.0;
42          }
43        }
44      }
45
46
47      #pragma omp barrier
48
49
50      #pragma omp single
51      {
52        memcpy(u, uplus, n * sizeof * u);
```

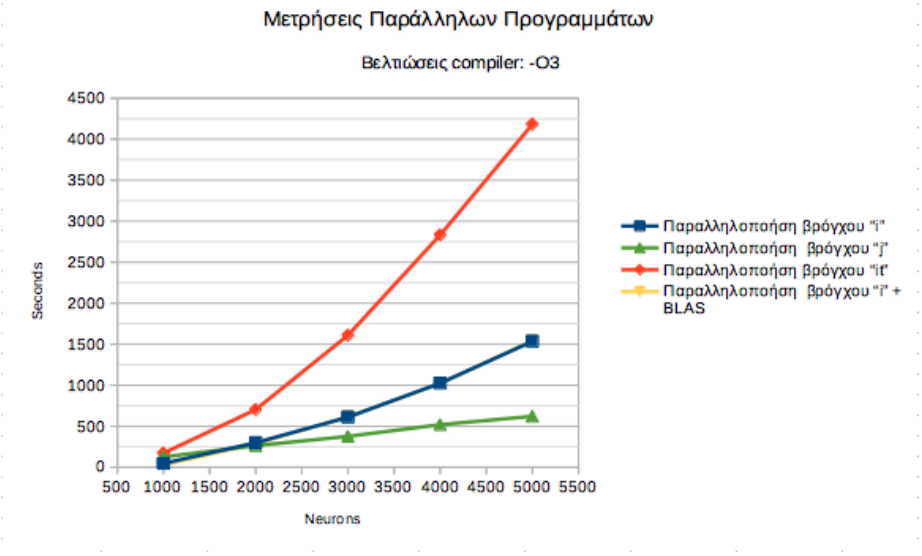
## 2.3 Μετρήσεις και Σχολιασμός Αποτελεσμάτων

Τα αποτελέσματα των μετρήσεων που κάναμε παρουσιάζουν ιδιαίτερο ενδιαφέρον. Αρχικά η αρχική εκτίμηση ότι η παραλληλοποίηση του βρόγχου j , ο οποίος εκτελεί τις πιο κοστοβόρες πράξεις θα οδηγούσε στον καλύτερο χρόνο εκτέλεσης , ήταν εν μέρη εσφαλμένη . Αυτό συνέβη διότι η δουλεία που ανατίθεται κατά την παραλληλοποίηση του στα threads είναι σχετικά μικρή σε σχέση με το κόστος που έχουμε για την εκτέλεση της παραλληλοποιήσης (overhead). Επίσης , άλλη μια σημαντική επισήμανση σχετικά με τα αποτελέσματα που έχουμε είναι η διαφορά της επίδοσης των διαφορών προγραμμάτων μας ανάλογα με τις βελτιώσεις που χρησιμοποιήθηκαν κατά την μεταγλώττιση -O0 και -O3. Ας δούμε λοιπόν πιο λεπτομερώς τα αποτελέσματα:



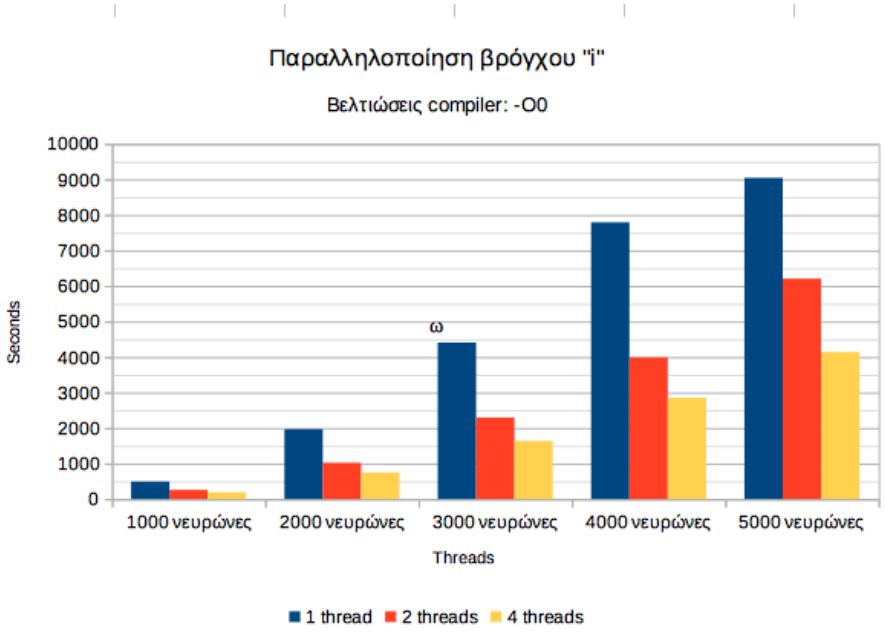

Σχήμα 5: Συνολικές μετρήσεις παράλληλων προγραμμάτων με -O0 για 4 νήματα

Όπως αναμέναμε βλέπουμε ότι καλύτερο χρόνο εκτέλεσης είχε η παραλληλοποιήση του βρόγχου i με την χρήση της βιβλιοθήκης BLAS. Παρατηρούμε επίσης ότι η παραλληλοποιήση του βρόγχου it με την χρήση των tasks είναι σημαντικά πιο αργή από τις υπόλοιπες. Αυτό ήταν κάτι το οποίο αναμέναμε από τον τρόπο που παραλληλοποιήσαμε τον συγκεκριμένο βρόγχο καθώς τα tasks μας ήταν εξαρτημένα συνεπώς δεν υπήρχε καμία ουσιαστική παραλληλοποίηση στο πρόγραμμα και όσο και να αυξάναμε τον αριθμό των νημάτων , ο χρόνος θα παρέμενε σχετικά σταθερός καμιά ουσιαστική βελτίωση. Ένα ακόμα ενδιαφέρον εύρημα στις μετρήσεις υπήρξε κατά την χρησιμοποίηση των βελτιώσεων -O3. Ας τις δούμε παρακάτω:

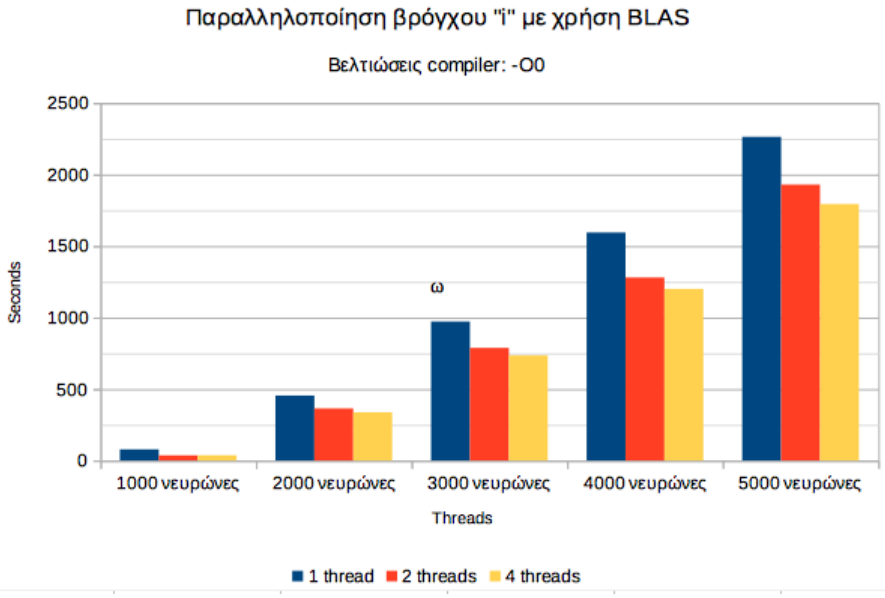



Σχήμα 6: Συνολικές μετρήσεις παράλληλων προγραμμάτων με -O3 για 4 νήματα

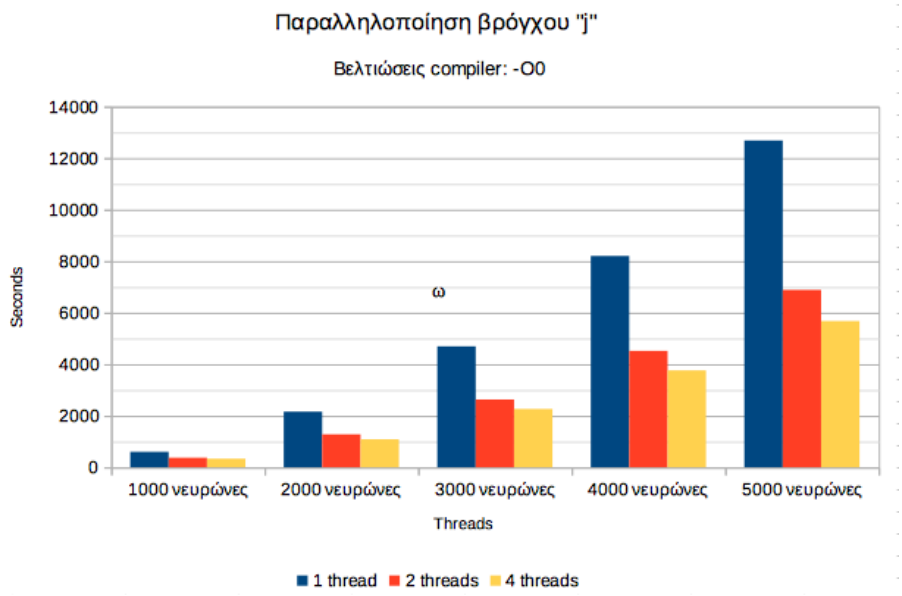
Κοιτώντας τις μετρήσεις βλέπουμε ότι στην προκείμενη περίπτωση η ταχύτερη υλοποίηση είναι αυτή της παραλληλοποίησης του βρόγχου j ενώ η παραλλήλοποιήση του βρόγχου i με ή χωρίς την χρήση της BLAS είναι παρόμοια . Αυτό οφείλεται στις βελτιώσεις που πραγματοποιεί ο μεταγλωττιστής στους βρόγχους από μόνος του στην προκειμένη περίπτωση. Στην συνέχεια ας δούμε πως διακυμάνθηκαν οι χρόνοι για κάθε υλοποίηση ξεχωριστά ανάλογα με τον αριθμό των νημάτων που χρησιμοποιήσαμε.



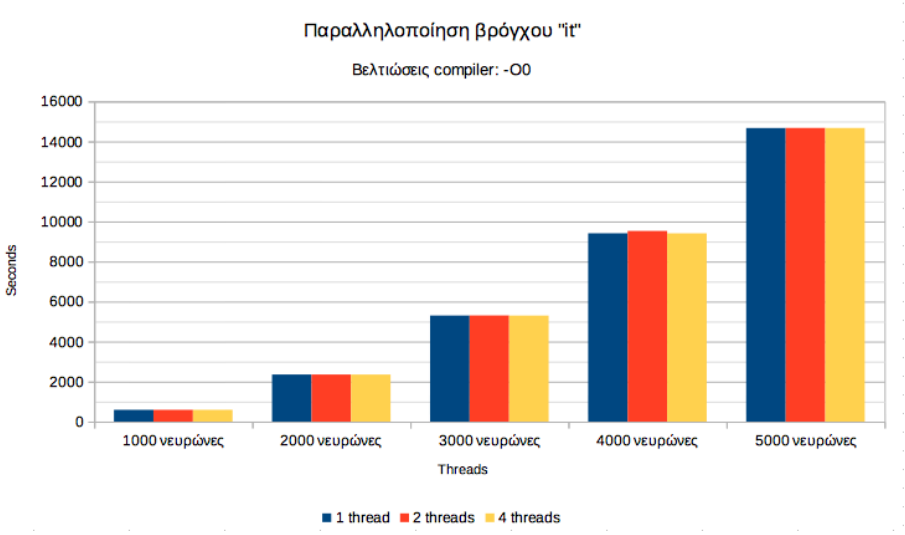

Σχήμα 7: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου i με -O0.



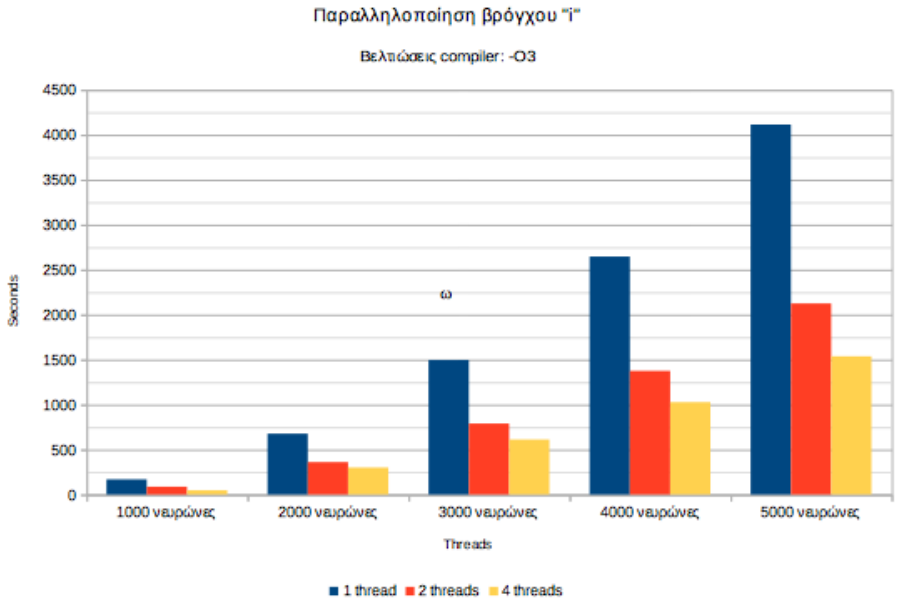

Σχήμα 8: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου i & BLAS με -O0.



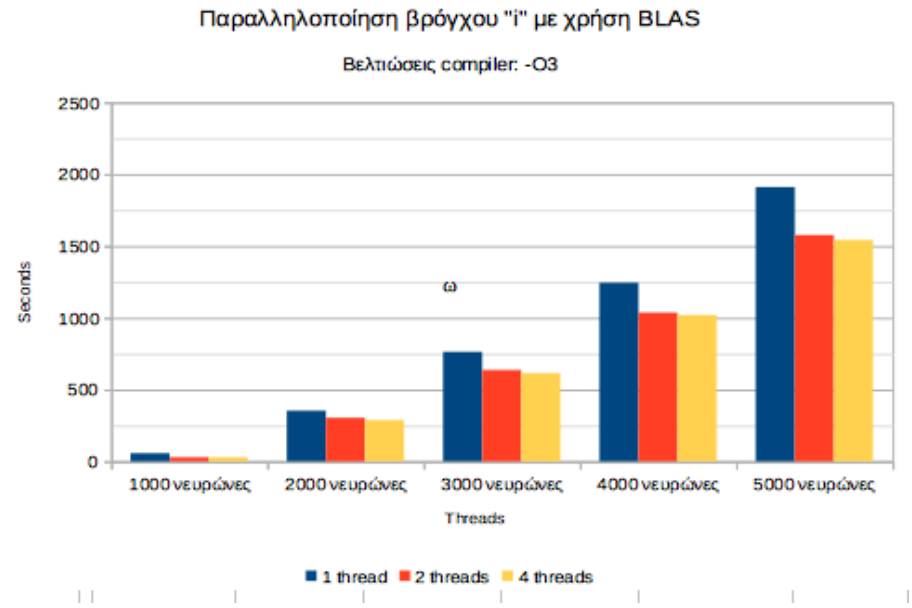

Σχήμα 9: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου j με -O0.



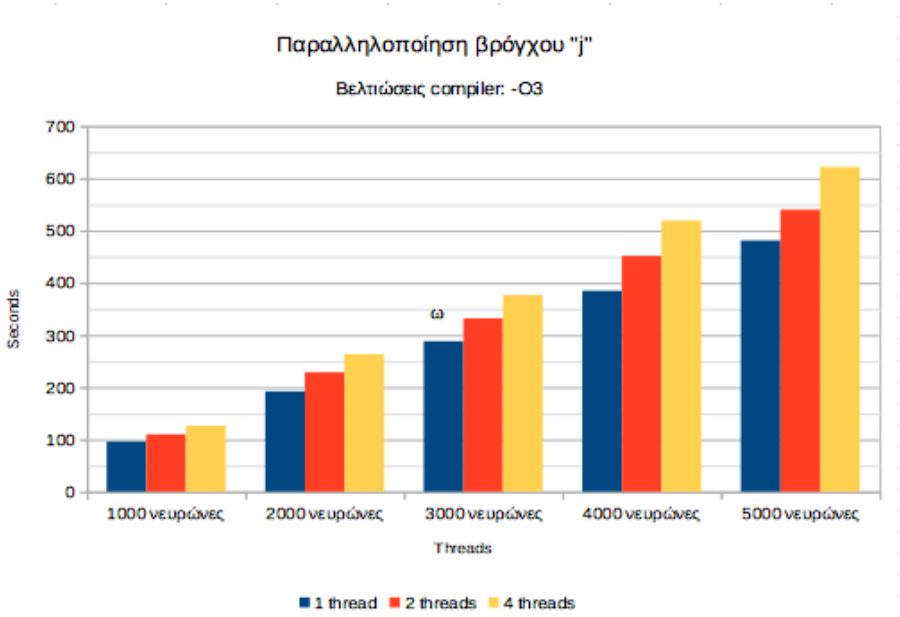

Σχήμα 10: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου it με -O0.



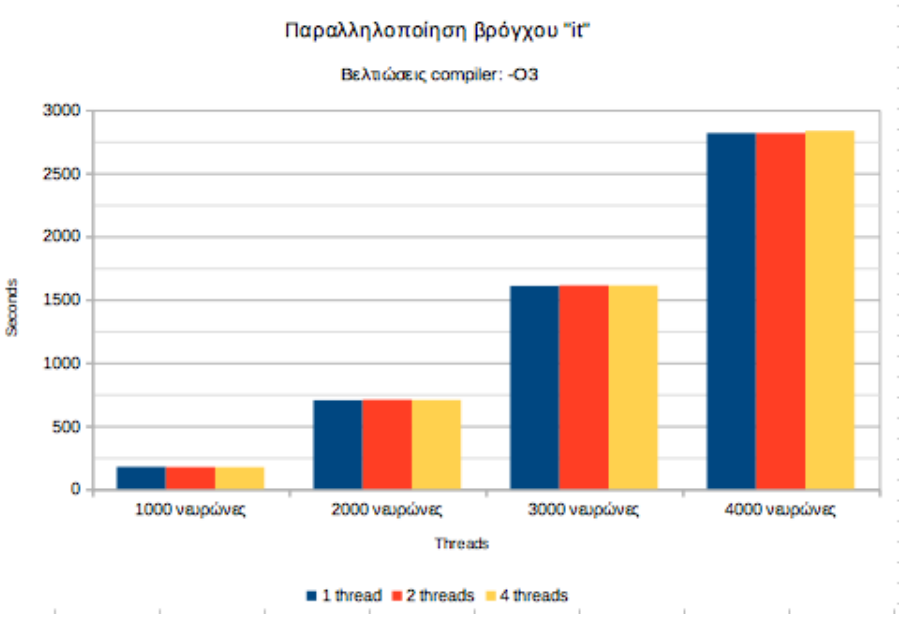

Σχήμα 11: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου i με -O3.




Σχήμα 12: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου i & BLAS με -O3.




Σχήμα 13: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου j με -O3.




Σχήμα 14: Συγκριτικές μετρήσεις παραλληλοποίησης βρόγχου it με -O3.

# 3 Ερώτημα 3: Βελτιστοποίηση Εγγραφής Αρχείων

Χρησιμοποιώντας την εντολή -DALL_RESULTS σε κάθε επανάληψη του εξωτερικού βρόγχου το πρόγραμμα μας προβαίνει στην συνεχόμενη εγγραφή των τιμών στα αρχεία μας κάτι που κοστίζει σημαντικά για μεγάλο αριθμό επαναλήψεων. Προκειμένου να διατηρήσουμε το πρόγραμμα μας αποδοτικό για όλες τις περιπτώσεις χρήσης προβήκαμε στις παρακάτω αλλαγές. Αρχικά χρησιμοποιήσαμε δυο πίνακες τύπου double τους spacetime_buf και omega_buf , οι οποίο θα κρατάνε όλες τις τιμές για τα ανάλογα αρχεία μέχρι το τέλος του προγράμματος. Επίσης δημιουργήσαμε άλλον έναν πίνακα τύπου int ο οποίος θα κρατάει τις τιμές της μεταβλητής time . Στην συνέχεια ενώσαμε τους δύο βρόγχους που υπήρχαν για την εγγραφή των τιμών σε έναν και μέσα σε αυτόν ενημερώνουμε κάθε φορά τους πινακές μας. Μόλις τελειώσουν οι επαναλήψεις του βρόγχου it , δημιουργούμε αρχικά δύο tasks και έπειτα γράφουμε παράλληλα στα αρχεία μας τα περιεχόμενα των πινάκων.

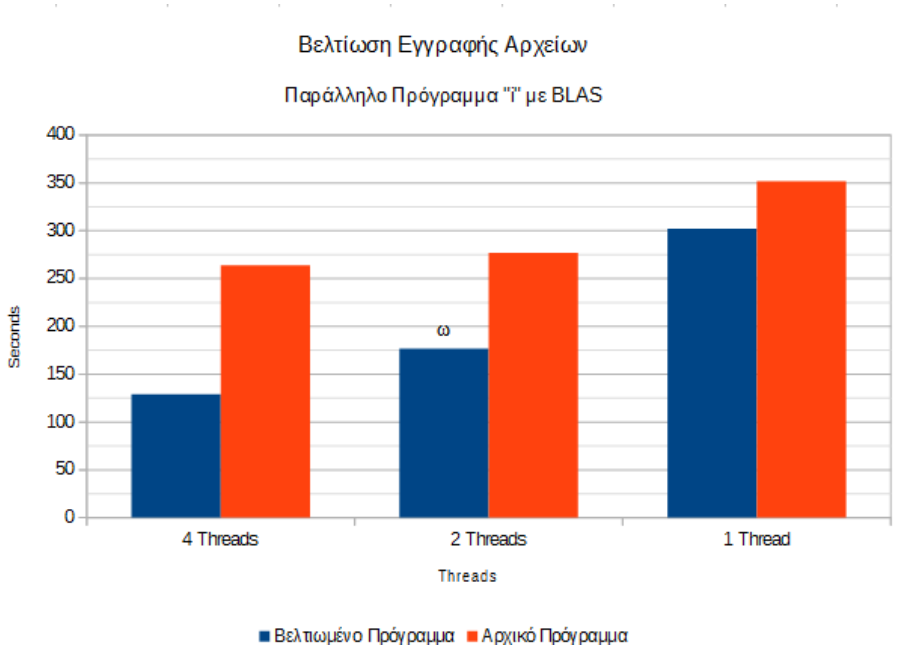
Απόσπασμα Κώδικα 8: Αναδιοργάνωση εγγραφών σε αρχεία στην παραλληλοποίηση του βρόγχου i + BLAS

```
1  int end ;
2  int index = 0;
3  int time_index = 0;
4  double *spacetime_buf;//buffer gia to spacetime
5  spacetime_buf = (double *)calloc(n * itime, sizeof(double));
6  double *omega_buf;//buffer gia to omega
7  omega_buf = (double *)calloc(n * itime, sizeof(double));
8  long int *time_buf;
9  time_buf = (long int *)calloc(itime, sizeof(long int));
10
11
12 double *final_sigma = (double *)calloc(n, sizeof(double));
13 int total_threads;
14
15 #pragma omp parallel private(it,i,j,step) firstprivate(n,sigma,
       ↪ dt,mu,semi_sum)
16 {
17
18   double *temp_u;
19   int thread_id = omp_get_thread_num();
20   int array_pos, position;
21
22   total_threads = omp_get_num_threads();
23   position = (n / total_threads) * n * thread_id;
24   array_pos = (n / total_threads) * thread_id;
25   temp_u = (double *)calloc(n / total_threads, sizeof(double));
26
```

```
27
28    for (it = 0; it < itime; it++) {
29
30      cblas_dgemv(CblasRowMajor, CblasNoTrans, n / total_threads,
            ↪ n, 1.0, sigma + position, n, u, 1 , 0.0, temp_u, 1);
31      memcpy( final_sigma + array_pos, temp_u, n / total_threads *
            ↪  sizeof * final_sigma );
32
33      #pragma omp for schedule(static,8)
34      for (i = 0; i < n; i++) {
35        step = i * n;
36
37        #pragma omp atomic write
38        uplus[i] = ( u[i] + dt * (mu - u[i])) + dt * (final_sigma[
              ↪ i] - semi_sum * u[i]) / divide;
39
40        if ( uplus[i] > uth) {
41          #pragma omp atomic write
42          uplus[i] = 0.0;
43
44          if (it >= ttransient) {
45            #pragma omp atomic
46            omega1[i] += 1.0;
47          }
48        }
49      }
50
51      #pragma omp barrier
52
53      #pragma omp single
54      memcpy(u, uplus, n * sizeof * u);
55
56
57 #if !defined(ALL_RESULTS)
58      if (it % ntstep == 0) {
59 #endif
60        #pragma omp master
61        {
62          printf("Time␣is␣%ld\n", it);
63          gettimeofday(&IO_start, NULL);
64          // fprintf(output1, "%ld\t", it);
65          time_buf[time_index] = it;
66          time_index++;
67          // printf("time buf: %ld index: %d\n",time_buf[index],
                ↪ index);
68          time = (double)it * dt;
```

```
69
70          for (i = 0; i < n; i++) {
71            // printf("index: %d\n",index);
72            // fprintf(output1, "%19.15f", u[i]);
73            omega[i] = 2.0 * M_PI * omega1[i] / (time - ttransient
                  ↪  * dt);
74            spacetime_buf[index] = u[i];
75            omega_buf[index] = omega[i];
76            index++;
77          }
78
79          gettimeofday(&IO_end, NULL);
80          IO_usec += ((IO_end.tv_sec - IO_start.tv_sec) *
                ↪ 1000000.0 + (IO_end.tv_usec - IO_start.tv_usec));
81        }//master end
82 #if !defined(ALL_RESULTS)
83       }
84 #endif
85     }
86
87     #pragma omp single
88     {
89
90       fprintf(output1, "%ld\t", time_buf[0]);
91       fprintf(output2, "%ld\t", time_buf[0]);
92
93       #pragma omp task firstprivate(index)
94       {
95         int time_index1 = 1;
96         // printf("thread to space: %d\n", omp_get_thread_num());
97         for (int i = 0; i < index; i++) {
98
99           if ((i % (n) == 0) && i != 0) {
100            fprintf(output1, "\n%ld\t", time_buf[time_index1]);
101            time_index1++;
102          }
103          fprintf(output1, "%19.15f", spacetime_buf[i]);
104        }
105      }
106
107      #pragma omp task firstprivate(index)
108      {
109        int time_index2 = 1;
110        // printf("thread to omega: %d\n", omp_get_thread_num());
111        for (int i = 0; i < index; i++) {
112
```

```
113           if ((i % (n) == 0) && i != 0) {
114             fprintf(output2, "\n%ld\t", time_buf[time_index2]);
115             time_index2++;
116           }
117           fprintf(output2, "%19.15f", omega_buf[i]);
118 
119         }
120       }
121     }
122   }//omp parallel
123 
124   gettimeofday(&global_end, NULL);
125   global_usec = ((global_end.tv_sec - global_start.tv_sec) *
      ↪ 1000000.0 + (global_end.tv_usec - global_start.tv_usec));
126 
127   printf("Time␣for␣calculations␣=␣%13.6f␣sec\n", (global_usec -
      ↪ IO_usec) / 1000000.0);
128   printf("Time␣for␣I/O␣␣␣␣␣␣␣␣␣␣=␣%13.6f␣sec\n", IO_usec /
      ↪ 1000000.0);
129   printf("Total␣execution␣time␣␣=␣%13.6f␣sec\n", global_usec /
      ↪ 1000000.0);
130 
131   fclose(output1);
132   fclose(output2);
133 
134   return 0;
135 }
```




Σχήμα 15: Συγκριτικές μετρήσεις βελτίωσης εγγραφής αρχείων παραλληλοποίησης βρόγχου i με -O3.

Για την διεξαγωγή των μετρήσεων χρησιμοποιήσαμε ως το αρχικό πρόγραμμα την πιο γρήγορη παραλληλοποιήση από τα προηγούμενα ερωτήματα , την παραλληλοποίηση του βρόγχου i με την χρήση BLAS , με βελτιώσεις από τον μεταγλωττιστή -O3. Παρατηρούμε από τις μετρήσεις μια σημαντική μείωση στον συνολικό χρόνο εκτέλεσης κάτι που οφείλεται κυρίως στους πολύ μικρούς χρόνους για I/O. Βλέπουμε επίσης ότι για την εκτέλεση για 1 νήμα ο χρόνος εκτέλεσης αυξάνεται κάτι που περιμέναμε καθώς για ένα νήμα δεν εκμεταλλευόμαστε την παραλληλία στην εγγραφή των αρχείων μας. Τρέξαμε τα δύο αρχεία για 1000 νευρώνες , με την χρήση 4 , 2 και 1 νημάτων. Τα σύστημα δεν μας επέτρεπε να τρέξουμε το πρόγραμμά μας για περισσότερους νευρώνες καθώς λόγω της παραγωγής πολύ μεγάλων αρχείων η διεργασία του προγράμματος σταματούσε από το λειτουργικό σύστημα.