Παράλληλη Επεξεργασία Εαρινό Εξάμηνο 2021-2022

"Παράλληλη Ολική Βελτιστοποίηση"

Ιούνιος 2022

Contents

1 Μέλη της ομάδας **3**

2 Εισαγωγή **4**

3 Στοιχεία συστήματος **5**

4 Sequential **6**
4.1 Περιγραφή Sequential 6
4.2 Μετρήσεις Sequential 6

5 OpenMP **7**
5.1 Περιγραφή OpenMP 7
5.2 6 threads 9
5.3 4 threads 11
5.4 2 threads 12
5.5 Συμπεράσματα για OpenMP 15

6 OpenMP with tasks **17**
6.1 Περιγραφή OpenMP with tasks 17
6.2 6 threads 19
6.3 4 threads 21
6.4 2 threads 23
6.5 Συμπεράσματα για OpenMP with tasks 26

7 MPI **28**
7.1 MPI Send & receive 28
7.1.1 6 threads 31
7.1.2 4 threads 33
7.1.3 2 threads 34
7.1.4 Συμπεράσματα για MPI Send & Receive 37
7.2 MPI All reduce 39
7.2.1 6 threads 41
7.2.2 4 threads 43
7.2.3 2 threads 45
7.2.4 Συμπεράσματα για MPI All reduce 47

8 Hybrid (MPI + OpenMP) **49**
8.1 6 threads 51
8.2 4 threads 53
8.3 2 threads 55
8.4 Συμπεράσματα για Hybrid 57

9 Τελικά συμπεράσματα **59**
9.1 Μετρήσεις 59

1 Μέλη της ομάδας

Ομάδα			
Όνομα	ΑΜ	Email	Έτος
Μπεχράκης Πέτρος	1040549	ceid4780@upnet.gr	13
Ξηρογιάννης Κωνσταντίνος	1047186	xirogiannis@ceid.upatras.gr	7
Πιτσιγαυδάκης Εμμανουήλ	1054405	up1054405@upnet.gr	6
Στέργιος Βασίλης	1054363	up1054363@upnet.gr	6

2 Εισαγωγή

Σκοπός της εργασίας ήταν να παραλληλοποιήσουμε το πρόγραμμα multistart hooke sequential με διάφορες τεχνικές παραλληλοποίησης ώστε να επιταχύνουμε το χρόνο εύρεσης των τοπικών ελαχίστων καθώς και την εύρεση του ολικού ελαχίστου. Πιο συγκεκριμένα χρησιμοποιήθηκαν οι τεχνικές :

Τεχνικές παραλληλοποίησης:

- Open MP
- Open MP tasks
- MPI με send και receive
- MPI με all reduce
- Hybrid υλοποίηση χρησιμοποιώντας και MPI και OpenMP

Για να μπορέσουμε να μετρήσουμε την επιτάχυνση της κάθε παραλληλοποίησης θα πραγματοποιήσουμε πειράματα με 2,4,6 threads για 64K τοπικά ελάχιστα και θα βρίσκουμε τον χρόνο της κάθε εκτέλεσης καθώς και τον συντελέστη επιτάχυνσης και την αποδοτικότητα. Τα αποτελέσματα αυτά καθώς και οι υλοποιήσεις παρουσιάζονται στα επόμενα κεφάλαια.

3 Στοιχεία συστήματος

Παραθέτουμε τα στοιχεία του υπολογιστή που έτρεξε τους κώδικες ώστε να υπάρχει ένα πλαίσιο κατανόησης σχετικά με τις μετρήσεις καθώς και την ολική επιτάχυνση στις διάφορες τεχνικές παραλληλοποίησης.

Σύστημα				
CPU	Clock	Cores	Threads	Ram
Intel I5 8600k	4.8 GHz all cores	6 cores	6 threads	12GB DDR4 @2666 MHz

4 Sequential

4.1 Περιγραφή Sequential

Στην υλοποίηση αυτή έχουμε μόνο ένα thread που υπολογίζει τα τοπικά ελάχιστα και τις συγκρίσεις μεταξύ τους καθώς δεν υπάρχει παραλληλοποίηση.Ο συγκεκριμένος κώδικας είναι που καλούμαστε να βελτιστοποιήσουμε και να επιταχύνουμε την παραλληλοποίηση του.

4.2 Μετρήσεις Sequential

Η μέτρηση του χρόνου εκτέλεσης της σειριακής υλοποίησης είναι σημαντικής βαρύτητας καθώς τη χρησιμοποίουμε ως σημείο αναφοράς για να υπολογίσουμε την επιτάχυνση (speed up) και την αποδοτικότητα (efficiency) των παράλληλων υλοποιήσεων. /

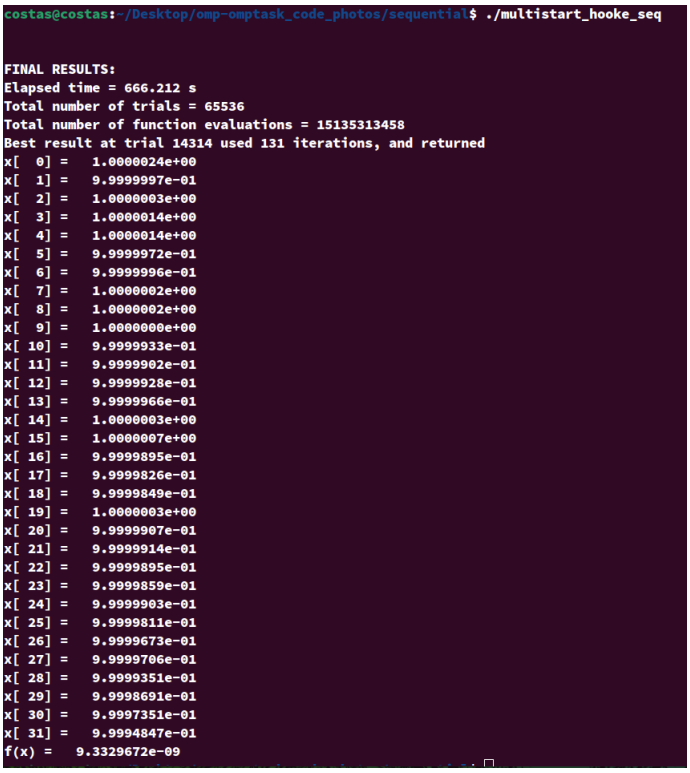

Figure 1: Χρόνος εκτέλεσης σειριακού κώδικα .

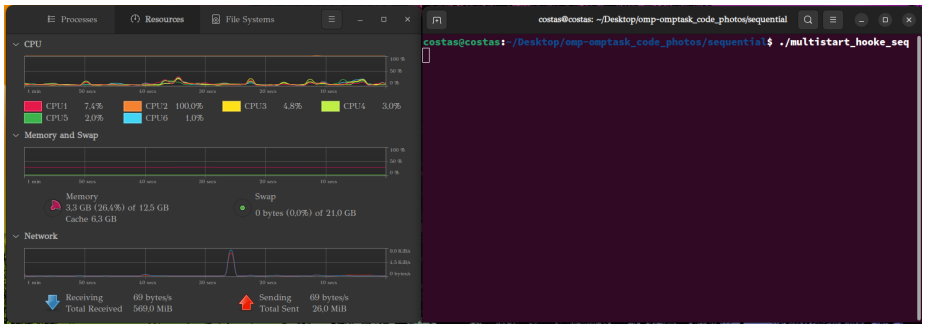

Figure 2: Cpu utilisation σειριακού κώδικα .

Παρατηρούμε ότι ο χρόνος εκτέλεσης είναι 666.212 δευτερόλεπτα δηλαδή κάτι πάνω απο 11 λεπτά , και βλέπουμε πως χρησιμοποιεί μόνο έναν απο τους 6 διαθέσιμους πυρήνες.

5 OpenMP

5.1 Περιγραφή OpenMP

Το OpenMP είναι ένα API το οποίο υποστηρίζει παραλληλοποίηση προγραμμάτων με χρήση κοινής κρυφής μνήμης.Η μέθοδος παραλληλοποίησης αυτή έχει μια αρχική διεργασία η οποία σε συγκεκριμένα σημεία του κώδικα δημιουργεί υποδιεργασίες για να μοιράσει συγκεκριμένες ενέργειες σε αυτές. Οι υποδιεργασίες αυτές τρέχουν παράλληλα και αναθέτονται σε διαφορετικούς πυρήνες.

Το κομμάτι κώδικα που θέλουμε να τρέξει παράλληλα στη συγκεκριμένη εργασία αναγνωρίζεται με κατάλληλες εντολές ώστε να γνωρίζει ο compiler πού πρέπει να δημιουργηθούν οι υποδιεργασίες.

Τέλος το OpenMP API από μόνο του δεν υποστηρίζει παραλληλοποίηση σε πάνω απο ένα υπολογιστικό σύστημα ,όπως για παράδειγμα clusters, καθώς απαιτεί τη χρήση κοινής κρυφής μνήμης.Για να πετύχει παραλληλοποίηση σε cluster θα χρειαστεί μια υβριδική προσέγγιση με MPI όπως παρουσιάζεται σε επόμενο κεφάλαιο.

Για το δικό μας πρόβλημα θα δημιουργήσουμε υπο-διεργασίες όπου η κάθε μία θα υπολογίζει ένα βέλτιστο τοπικό ελάχιστο και αυτό το πετυχαίνουμε με την εντολή :

```
#pragma omp parallel shared (trial)
```

Στη συγκέκριμένη περίπτωση επειδή παραλληλοποιούμε τα trials δίνουμε σαν όρισμα στην παραλληλοποίηση τη μεταβλητή trial. Εναλλακτικά θα μπορούσε να γίνει με την εντολή :

```
   #pragma omp parallel for
```

Όπου θα έκανε την επιλογή της μεταβλητής αυτόματα.

Παραθέτουμε των κώδικα της main όπου έγιναν οι αλλαγές για να παραλληλοποιηθεί ο κώδικας.

```
// Open MP
int main(int argc, char *argv[])
{
   double startpt[MAXVARS], endpt[MAXVARS];
   int itermax = IMAX;
   double rho = RHO_BEGIN;
   double epsilon = EPSMIN;
   int nvars;
   int trial, ntrials;
   double fx;
   int i, jj;
   double t0, t1;
    double best_fx = 1e10;
   double best_pt[MAXVARS];
   int best_trial = -1;
   int best_jj = -1;

   for (i = 0; i < MAXVARS; i++) best_pt[i] = 0.0;
   ntrials = 64*1024; /* number of trials */
   nvars = 32;   /* number of variables (problem dimension) */
   srand48(1);

   t0 = get_wtime();
   omp_set_dynamic(0); // Thetoume 0 gia na orisoume posa thread tha
       xrhsimopoihsoume me thn entolh apo katw
   omp_set_num_threads(6); //Orizoume ton arithmo thread se 6
   #pragma omp parallel shared (trial) // Ksekiname thn parallhlopoihsh
       (enallaktika #pragma omp parallel for )
   {
   for (trial = 0; trial < ntrials; trial++) {
      /* starting guess for rosenbrock test function, search space in
          [-5, 5) */

      for (i = 0; i < nvars; i++) {
         startpt[i] = 10.0*drand48()-5.0;
      }

      jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
#if DEBUG
      printf("\n\n\nHOOKE %d USED %d ITERATIONS, AND RETURNED\n", trial,
          jj);
```

```
      for (i = 0; i < nvars; i++)
         printf("x[%3d] = %15.7le \n", i, endpt[i]);
#endif
        fx = f(endpt, nvars);
        #if DEBUG
      printf("f(x) = %15.7le\n", fx);
#endif
      if (fx < best_fx) {
         best_trial = trial;
         best_jj = jj;
         best_fx = fx;
         for (i = 0; i < nvars; i++)
            best_pt[i] = endpt[i];
      }
   }}
   t1 = get_wtime();
    printf("\n\nFINAL RESULTS:\n");
   printf("Elapsed time = %.3lf s\n", t1-t0);
   printf("Total number of trials = %d\n", ntrials);
   printf("Total number of function evaluations = %ld\n", funevals);
   printf("Best result at trial %d used %d iterations, and returned\n",
       best_trial, best_jj);
   for (i = 0; i < nvars; i++) {
      printf("x[%3d] = %15.7le \n", i, best_pt[i]);
   }
   printf("f(x) = %15.7le\n", best_fx);
   return 0;
}
```

Με την εντολή :

```
omp_set_dynamic(0);
```

θέτουμε ότι δεν θα κάνει dynamic διαμοιρασμό σε thread και πως θα ορίσουμε εμείς συγκεκριμένα πόσα thread θα χρησιμοποιηθούν.

Με την εντολή :

```
   omp_set_num_threads(6);
```

Θέτουμε ότι θα χρησιμοποιήσει 6 νήματα (Threads)

5.2 6 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP με 6 threads:

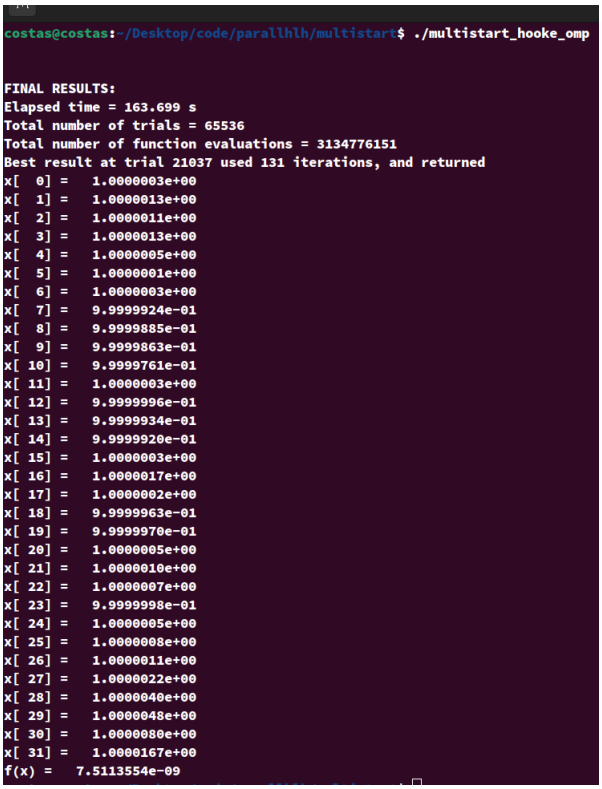

Figure 3: Χρόνος εκτέλεσης OPEN MP με 6 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 163.699 δευτερόλεπτα δηλαδή 2.7 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 75.5% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(6) = \frac{666.212}{163.699} = 4.068 \tag{1}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(6) = \frac{6}{1} = 6 \tag{2}$$

Απόδοση :

$$E(6) = \frac{S(6)}{6}\% = \frac{4.068}{6}\% = 67.8\% \tag{3}$$

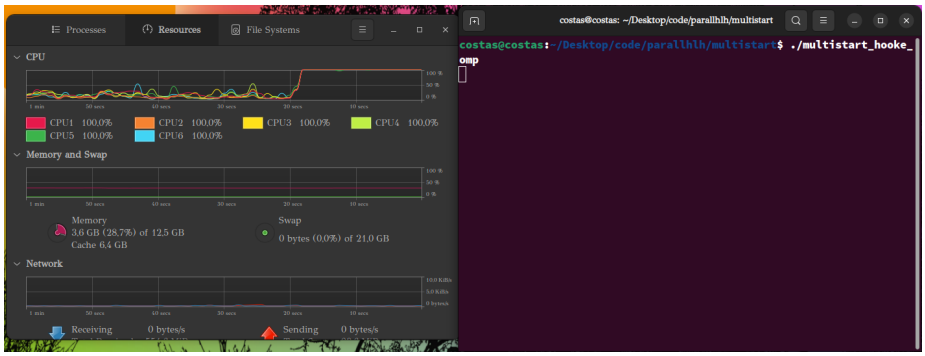

Figure 4: Cpu utilisation OPEN MP με 6 threads.

5.3 4 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP με 4 threads: Η μόνη εντολή που άλλαξε με τον κώδικα που έχουμε παραθέσει ήδη είναι η ακόλουθη:

```
omp_set_num_threads(4);
```

Όπου θέτουμε τον αριθμό των thread σε 4.

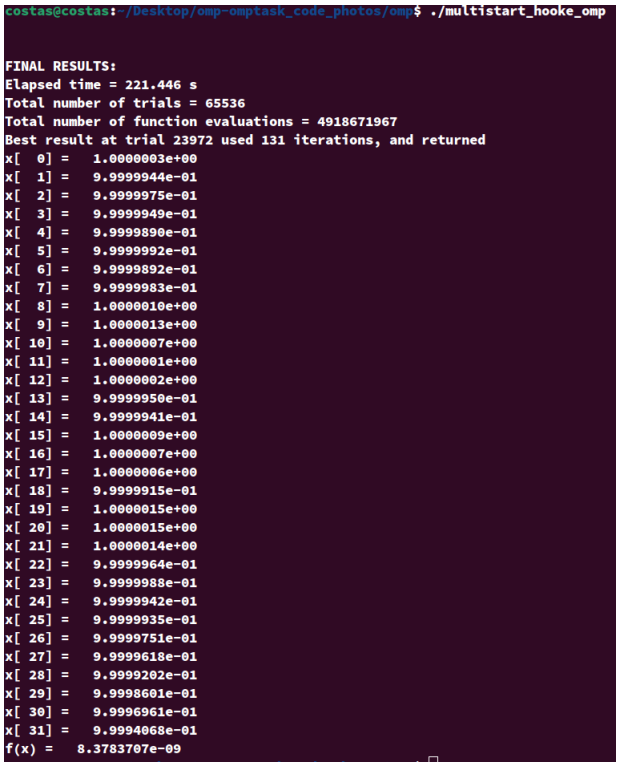

Figure 5: Χρόνος εκτέλεσης OPEN MP με 4 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 221.446 δευτερόλεπτα δηλαδή 3.69 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 66.8% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(4) = \frac{666.212}{221.446} = 3.008 \tag{4}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(4) = \frac{4}{1} = 4 \tag{5}$$

Απόδοση :

$$E(4) = \frac{S(4)}{4}\% = \frac{3.008}{4}\% = 75.2\% \tag{6}$$

5.4 2 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP με 2 threads: Η μόνη εντολή που άλλαξε με τον κώδικα που έχουμε παραθέσει ήδη είναι η ακόλουθη:

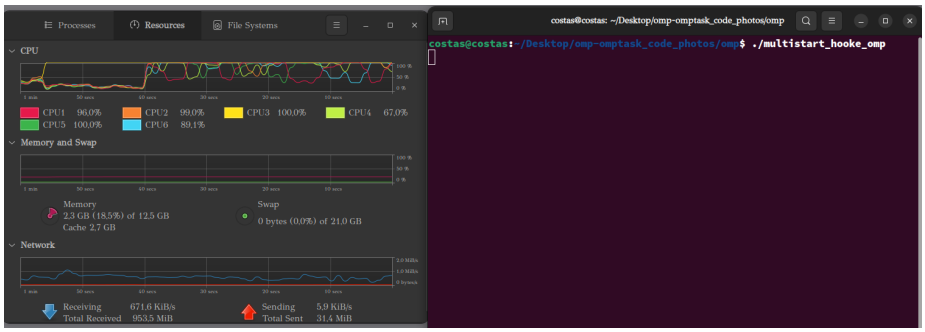

Figure 6: Cpu utilisation OPEN MP με 4 threads.

```
omp_set_num_threads(2);
```

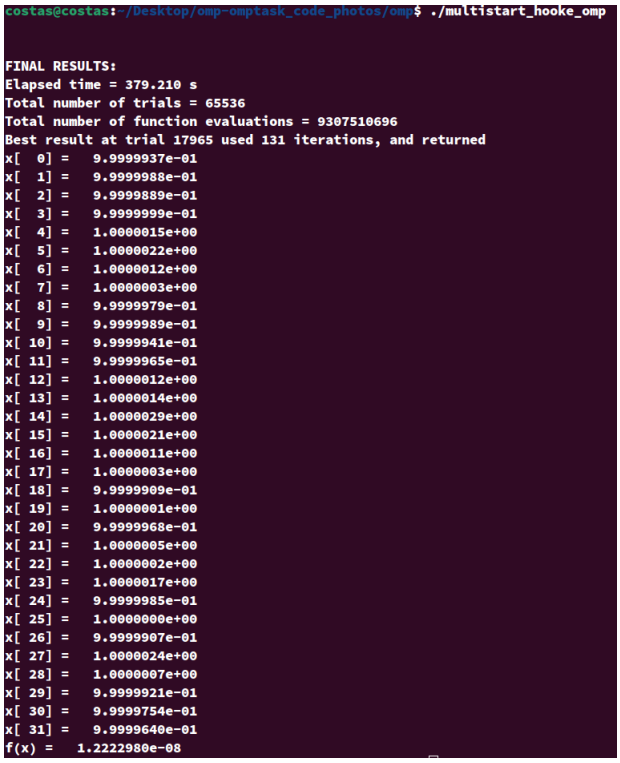

Figure 7: Χρόνος εκτέλεσης OPEN MP με 2 threads.

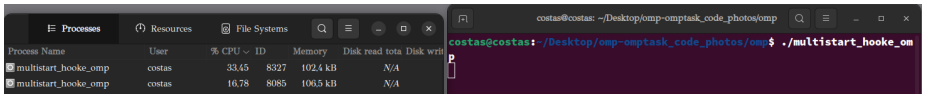

Figure 8: Cpu utilisation OPEN MP με 2 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 379.21 δευτερόλεπτα δηλαδή 6.32 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 43% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(2) = \frac{666.212}{379.21} = 1.756 \tag{7}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(2) = \frac{2}{1} = 2 \tag{8}$$

Απόδοση :

$$E(2) = \frac{S(2)}{2}\% = \frac{1.756}{2}\% = 87.8\% \tag{9}$$

5.5 Συμπεράσματα για OpenMP

Παρατηρούμε με την αύξηση του αριθμού των thread μια μείωση της απόδοσης αλλά παρόλα αυτά ο χρόνος εκτέλεσης μειώνεται.Αυτό συμβαίνει λόγω της μεγαλύτερης ανάγκης να διευθετυθούν οι συγκρούσεις των thread στη κοινή κρυφή μνήμη.

Open MP				
#Threads	Χρόνος σε seconds	Μειώση σε (%)	Efficiency	Speed up (measured)
6	163.699s	75.5%	67.8%	4.068
4	221.446s	66.8%	75.2%	3.008
2	379.21s	43%	87.8%	1.756
Σειριακό	666.212	0	100%	1.000

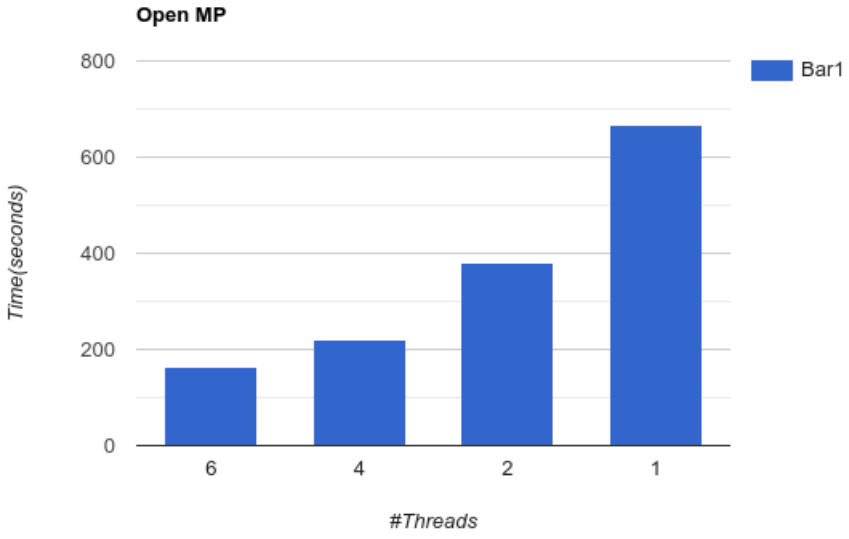

Figure 9: Bar graph of time in relation with threads (lower is better).

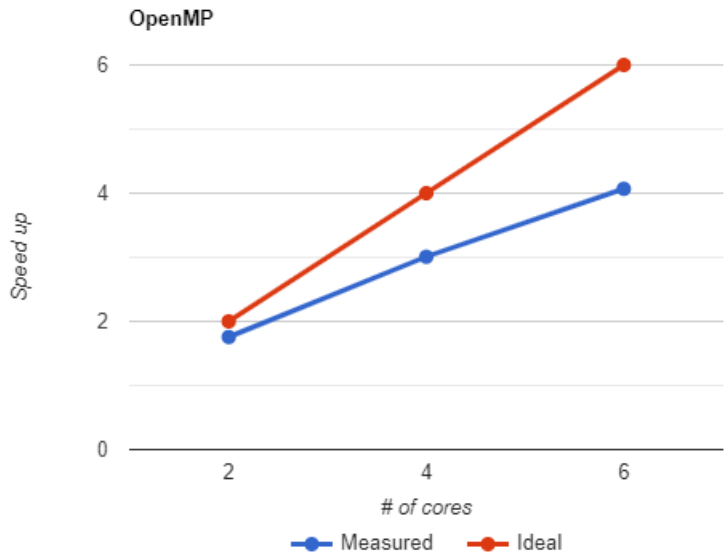

Figure 10: Line graph of theoretical speed up against measured speed up

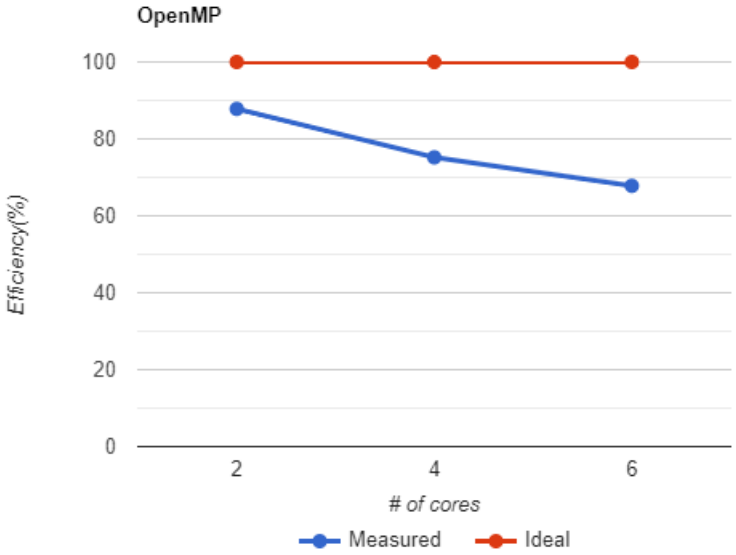

Figure 11: Line graph of theoretical efficiency against measured efficiency

6 OpenMP with tasks

6.1 Περιγραφή OpenMP with tasks

Στο OpenMP API όταν επιθυμούμε να έχουμε μεγαλύτερο έλεγχο στα thread που δημιουργούνται καθώς και να δημιουργήσουμε εμφωλευμένα threads τότε χρησιμοποιούμε την τεχνική των task. Στη δικιά μας υλοποίηση απομονώσαμε σαν πιο υπολογιστικά κοστοβόρα διαδικασία του υπολογισμού του αποτελέσματος της συνάρτησης hooke και την ανάθεση της στην μεταβλητή jj. Για αυτό το λόγο δημιουργήσαμε ένα task για τον υπολογισμό του τοπικού ελαχίστου με τη μέθοδο του Hooke-Jeeves. Παραθέτουμε των κώδικα της main όπου έγιναν οι αλλαγές για να παραλληλοποιηθεί ο κώδικας.

```
// Open MP with tasks
int main(int argc, char *argv[])
{
   double startpt[MAXVARS], endpt[MAXVARS];
   int itermax = IMAX;
   double rho = RHO_BEGIN;
   double epsilon = EPSMIN;
   int nvars;
   int trial, ntrials;
   double fx;
   int i, jj;
   double t0, t1;

   double best_fx = 1e10;
   double best_pt[MAXVARS];
   int best_trial = -1;
   int best_jj = -1;
   omp_set_dynamic(0); // THESE 0 GIA NA ALLA3EIS POSA THREAD FTIAXNEI
   omp_set_num_threads(6);
   for (i = 0; i < MAXVARS; i++) best_pt[i] = 0.0;

   ntrials = 64*1024; /* number of trials */
   nvars = 32;   /* number of variables (problem dimension) */
   srand48(1);

   t0 = get_wtime();
   #pragma omp parallel shared(trial)
   {
   for (trial = 0; trial < ntrials; trial++) {
      /* starting guess for rosenbrock test function, search space in
          [-5, 5) */

      for (i = 0; i < nvars; i++) {
         startpt[i] = 10.0*drand48()-5.0;
      }
```

```
      #pragma omp task //edw dhmiourgoume to task
      {
      jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
      }

#if DEBUG
      printf("\n\n\nHOOKE %d USED %d ITERATIONS, AND RETURNED\n", trial,
          jj);

      for (i = 0; i < nvars; i++)
        printf("x[%3d] = %15.7le \n", i, endpt[i]);
#endif

      fx = f(endpt, nvars);
#if DEBUG
      printf("f(x) = %15.7le\n", fx);
#endif
      #pragma omp taskwait
      if (fx < best_fx) {
        best_trial = trial;
        best_jj = jj;
        best_fx = fx;
        for (i = 0; i < nvars; i++)
          best_pt[i] = endpt[i];
      }
   } }
   t1 = get_wtime();

   printf("\n\nFINAL RESULTS:\n");
   printf("Elapsed time = %.3lf s\n", t1-t0);
   printf("Total number of trials = %d\n", ntrials);
   printf("Total number of function evaluations = %ld\n", funevals);
   printf("Best result at trial %d used %d iterations, and returned\n",
       best_trial, best_jj);
   for (i = 0; i < nvars; i++) {
      printf("x[%3d] = %15.7le \n", i, best_pt[i]);
   }
   printf("f(x) = %15.7le\n", best_fx);

   return 0;
}
```

Με την εντολή :

```
omp_set_dynamic(0);
```

Θέτουμε ότι δεν θα κάνει dynamic διαμοιρασμό σε thread και πως θα ορίσουμε εμείς συγκεκριμένα πόσα thread θα χρησιμοποιηθούν.

Με την εντολή :

```
omp_set_num_threads(6);
```

Θέτουμε ότι θα χρησιμοποιήσει 6 νήματα (Threads)

Με την εντολή :

```
#pragma omp parallel shared (trial)
```

Ξεκινάμε την παραλληλοποίηση με κοινή μεταβλητή την trial.
Με την εντολή :

```
#pragma omp task
  {
  jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
  }
```

Θέτουμε σαν task την ανάθεση της μεταβλητής jj με τα αποτελέσματα της συνάρτησης hooke.
Με την εντολή :

```
#pragma omp taskwait
```

Όριζουμε πως πρέπει να περιμένουμε να τελειώσει το task πριν συνεχίσει η εκτέλεση του κώδικα.

6.2 6 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP with tasks με 6 threads:

```
costas@costas:~/Desktop/code/parallhlh/multistart$ ./multistart_hooke_omp_task

FINAL RESULTS:
Elapsed time = 163.849 s
Total number of trials = 65536
Total number of function evaluations = 3207435835
Best result at trial 56805 used 131 iterations, and returned
x[  0] =   1.0027755e+00
x[  1] =   1.0013816e+00
x[  2] =   1.0006880e+00
x[  3] =   1.0003436e+00
x[  4] =   1.0001714e+00
x[  5] =   1.0000853e+00
x[  6] =   1.0000422e+00
x[  7] =   1.0000202e+00
x[  8] =   1.0000085e+00
x[  9] =   1.0000037e+00
x[ 10] =   1.0000013e+00
x[ 11] =   9.9999965e-01
x[ 12] =   9.9999801e-01
x[ 13] =   9.9999778e-01
x[ 14] =   9.9999323e-01
x[ 15] =   9.9998916e-01
x[ 16] =   9.9997990e-01
x[ 17] =   9.9996289e-01
x[ 18] =   9.9992674e-01
x[ 19] =   9.9985468e-01
x[ 20] =   9.9970743e-01
x[ 21] =   9.9941389e-01
x[ 22] =   9.9882002e-01
x[ 23] =   9.9763534e-01
x[ 24] =   9.9525964e-01
x[ 25] =   9.9051088e-01
x[ 26] =   9.8104161e-01
x[ 27] =   9.6230660e-01
x[ 28] =   8.9900957e-01
x[ 29] =   7.9263106e-01
x[ 30] =   6.1712290e-01
x[ 31] =   3.6871219e-01
f(x) =   1.1328658e-08
```

Figure 12: Χρόνος εκτέλεσης OPEN MP with tasks με 6 threads.

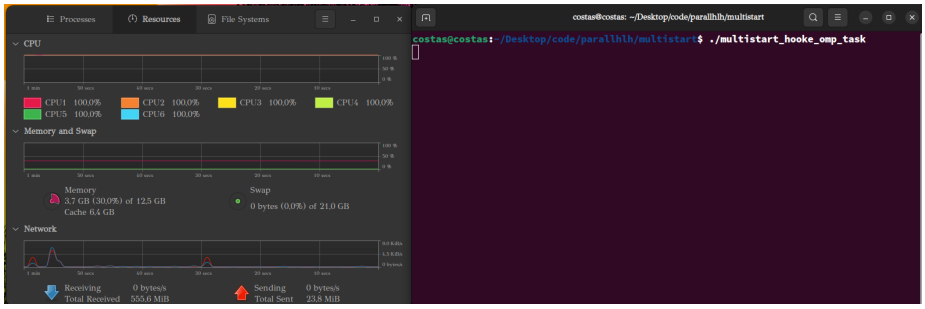

Figure 13: Cpu utilisation OPEN MP with tasks με 6 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 163.849 δευτερόλεπτα δηλαδή 2.73 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 75.3% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(6) = \frac{666.212}{163.849} = 4.066 \tag{10}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(6) = \frac{6}{1} = 6 \tag{11}$$

Απόδοση :

$$E(6) = \frac{S(6)}{6}\% = \frac{4.066}{6}\% = 67.7\% \tag{12}$$

6.3 4 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP with tasks με 4 threads: Η μόνη εντολή που άλλαξε με τον κώδικα που έχουμε παραθέσει ήδη είναι η ακόλουθη:

```
omp_set_num_threads(4);
```

```
costas@costas:~/Desktop/omp-omptask_code_photos/omp_task/4 threads$ ./multistart_hooke_omptask

FINAL RESULTS:
Elapsed time = 212.593 s
Total number of trials = 65536
Total number of function evaluations = 4823022236
Best result at trial 54000 used 131 iterations, and returned
x[  0] =   1.0043388e+00
x[  1] =   1.0022905e+00
x[  2] =   1.0014038e+00
x[  3] =   1.0012316e+00
x[  4] =   1.0016800e+00
x[  5] =   1.0034879e+00
x[  6] =   1.0070672e+00
x[  7] =   1.0034177e+00
x[  8] =   1.0015001e+00
x[  9] =   1.0003425e+00
x[ 10] =   9.9936130e-01
x[ 11] =   9.9805851e-01
x[ 12] =   9.9577497e-01
x[ 13] =   9.9137504e-01
x[ 14] =   9.8268308e-01
x[ 15] =   9.6550814e-01
x[ 16] =   9.3194387e-01
x[ 17] =   8.6801450e-01
x[ 18] =   7.5240024e-01
x[ 19] =   5.6376920e-01
x[ 20] =   3.1189699e-01
x[ 21] =   1.0840716e-01
x[ 22] =   2.1975106e-02
x[ 23] =   1.0591088e-02
x[ 24] =   1.0216889e-02
x[ 25] =   1.0209345e-02
x[ 26] =   1.0209409e-02
x[ 27] =   1.0208560e-02
x[ 28] =   1.0208922e-02
x[ 29] =   1.0204112e-02
x[ 30] =   1.0002275e-02
x[ 31] =   9.9387319e-05
f(x) =   9.2830681e-09
```

Figure 14: Χρόνος εκτέλεσης OPEN MP with tasks με 4 threads.

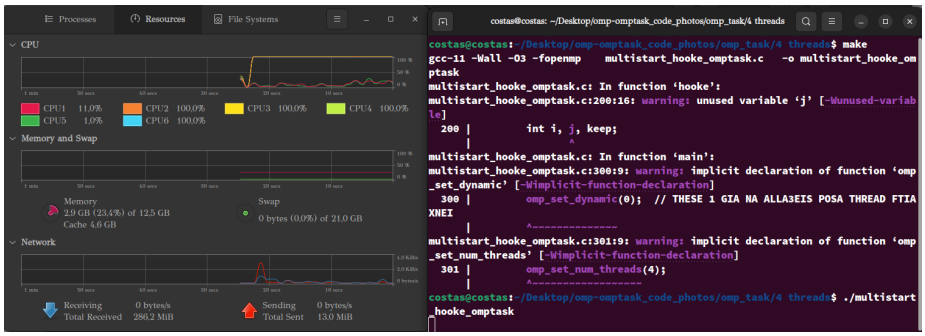

Figure 15: Cpu utilisation OPEN MP with tasks με 4 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 212.593 δευτερόλεπτα δηλαδή 3.54 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 68% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(4) = \frac{666.212}{212.593} = 3.133 \tag{13}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(4) = \frac{4}{1} = 4 \tag{14}$$

Απόδοση :

$$E(4) = \frac{S(4)}{4}\% = \frac{3.133}{4}\% = 78.3\% \tag{15}$$

6.4 2 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με OPEN MP with tasks με 2 threads: Η μόνη εντολή που άλλαξε με τον κώδικα που έχουμε παραθέσει ήδη είναι η ακόλουθη:

```
omp_set_num_threads(2);
```

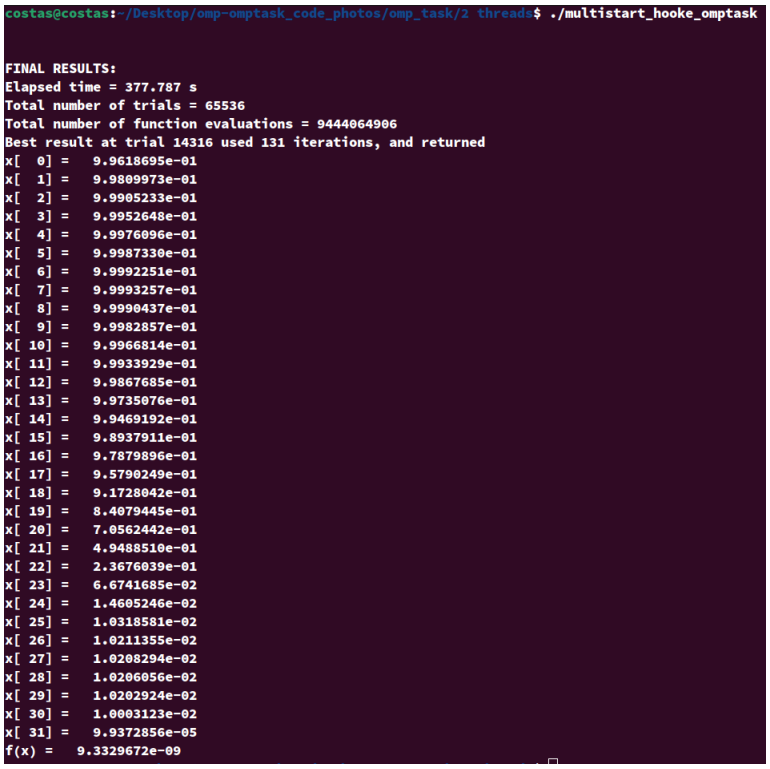

Figure 16: Χρόνος εκτέλεσης OPEN MP with tasks με 2 threads.

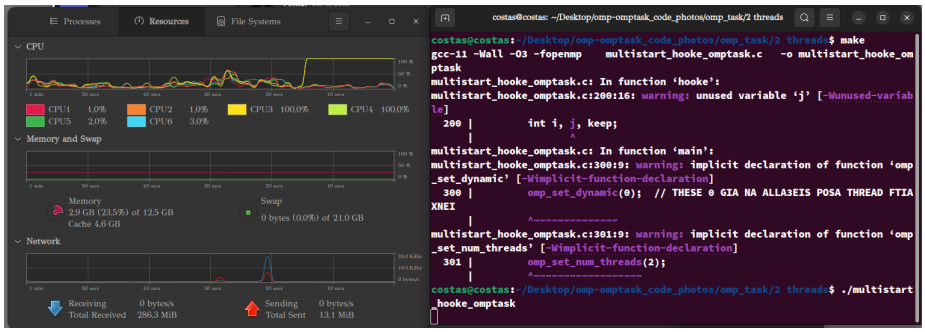

Figure 17: Cpu utilisation OPEN MP with tasks με 2 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 377.787 δευτερόλεπτα δηλαδή 6.2 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 43.3% σε σχέση με τη σειριακή υλοποίηση. Παράγοντας επιτάχυνσης :

$$S(2) = \frac{666.212}{377.787} = 1.763 \tag{16}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(2) = \frac{2}{1} = 2 \tag{17}$$

Απόδοση :

$$E(2) = \frac{S(2)}{2}\% = \frac{1.763}{2}\% = 88.1\% \tag{18}$$

6.5 Συμπεράσματα για OpenMP with tasks

Παρατηρούμε ότι η αποδόσεις είναι παρόμοιες με αυτές της απλής OpenMP αλλά έχουμε μια αύξηση στην απόδοση στις υλοποιήσεις που χρησιμοποιούν 2 και 4 thread.

OpenMP with tasks				
#Threads	Χρόνος σε seconds	Μειώση σε (%)	Efficiency	Speed up (measured)
6	163.849s	75.3%	67.7%	4.066
4	212.593s	68%	78.3%	3.133
2	377.787s	43.3%	88.1 %	1.763
Σειριακό	666.212	0	100%	1.000

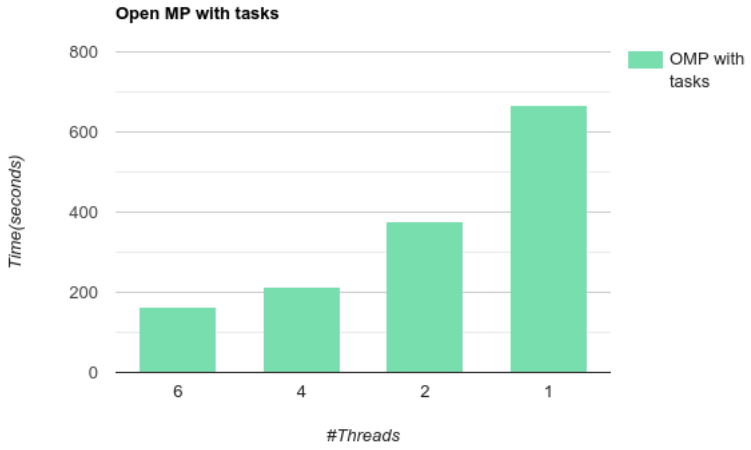

Figure 18: Bar graph of time in relation with threads (lower is better).

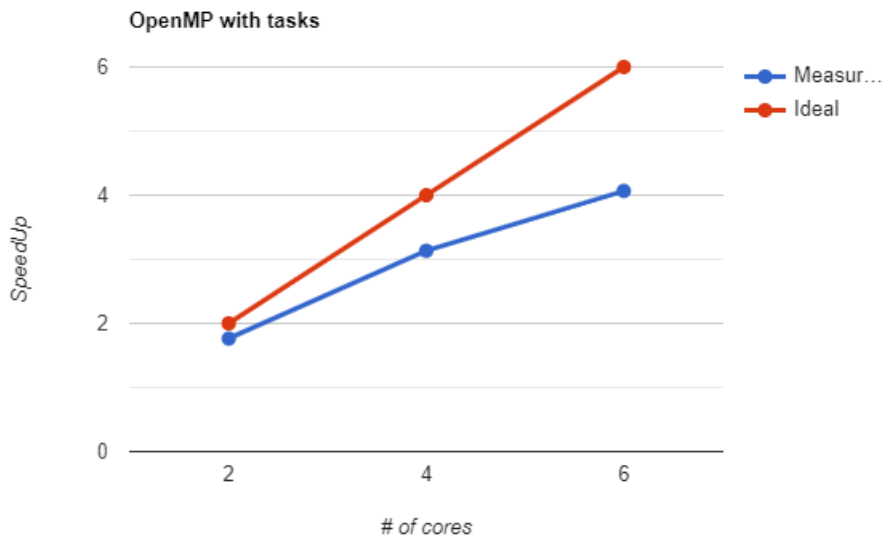

Figure 19: Line graph of theoretical speed up against measured speed up

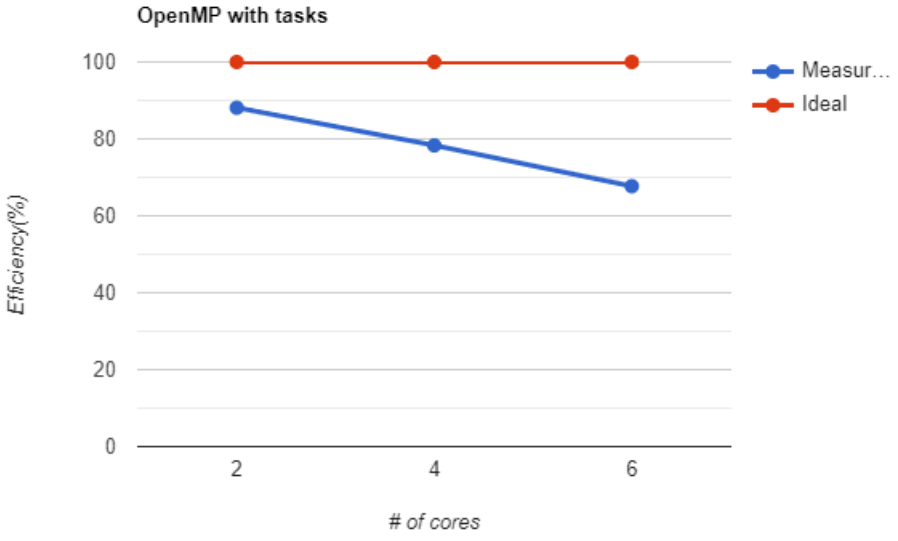

Figure 20: Line graph of theoretical efficiency against measured efficiency

7 MPI

Το MPI είναι ενα πρωτόκολλο επικοινωνίας μεταξύ διεργασιών με την χρήση μηνυμάτων με σκοπό την παραλληλοποίηση ενός προγράμματος. Μέσω αυτού επιτρέπεται να επικοινωνήσουν διεργασίες μεταξύ τους που δεν βρίσκονται απαραίτητα στο ίδιο υπολογιστικό σύστημα αφού δεν απαιτείται κοινή κρυφή μνήμη όπως στην υλοποίηση με το OpenMP API. Η παραλληλοποίηση με το MPI δεν μπορεί να γίνει με αυτόματο τρόπο , όπως γίνεται στο OpenMP , αλλά πρέπει εμείς να επιλέξουμε πότε οι διεργασίες θα επικοινωνήσουν μεταξύ τους. Τέλος σε αντίθεση με το OpenMP όπου εμείς ξεκινάμε ένα thread και αυτό αρχικοποιεί άλλα, στο MPI είναι ανεξάρτητες διαδικασίες που τρέχουν ολόκληρο τον κώδικα και όχι μεμονωμένα κομμάτια του.

7.1 MPI Send & receive

Με τις εντολές :

`MPI_Send() , MPI_Recv()`

Συγκεντρώνουμε στο thread με rank 0 τα αποτελέσματα από κάθε μία διεργασία και τα συγκρίνουμε ώστε να βρούμε το ολικό ελάχιστο και στη συνέχεια το εμφανίζουμε.

Παραθέτουμε των κώδικα της main όπου έγιναν οι αλλαγές για να παραλληλοποιηθεί ο κώδικας.

```
// MPI Send & Receive
int main(int argc, char *argv[])
{
   MPI_Status status;
   double startpt[MAXVARS], endpt[MAXVARS];
   int itermax = IMAX;
   double rho = RHO_BEGIN;
   double epsilon = EPSMIN;
   int nvars;
   int trial, ntrials;
   double fx;
   int i, jj;
   double t0, t1;
   int rank;
   int size;
   double best_fx = 1e10;
   double best_pt[MAXVARS];
   int best_trial = -1;
   int best_jj = -1;

   for (i = 0; i < MAXVARS; i++) best_pt[i] = 0.0;

   ntrials = 64*1024; /* number of trials */
```

```
   nvars = 32;   /* number of variables (problem dimension) */
   srand48(1);

   t0 = get_wtime();
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   double const step = (int)(ntrials+0.51) / size;
   // do just one piece on each rank
   unsigned long start = rank * step;
   unsigned long end = (rank+1) * step;

   for (trial = start; trial < end; trial++) {
      /* starting guess for rosenbrock test function, search space in
          [-5, 5) */

      for (i = 0; i < nvars; i++) {
         startpt[i] = 10.0*drand48()-5.0;
      }

      jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
#if DEBUG
      printf("\n\n\nHOOKE %d USED %d ITERATIONS, AND RETURNED\n", trial,
          jj);
      for (i = 0; i < nvars; i++)
        printf("x[%3d] = %15.7le \n", i, endpt[i]);
#endif

      fx = f(endpt, nvars);
#if DEBUG
      printf("f(x) = %15.7le\n", fx);
#endif
      if (fx < best_fx) {
         best_trial = trial;
         best_jj = jj;
         best_fx = fx;
         for (i = 0; i < nvars; i++)
            best_pt[i] = endpt[i];
      }
   }
t1 = get_wtime();
//printf("Elapsed time = %.3lf s\n""recv:%d", t1-t0,i);
    if (rank == 0) {
        for (i = 1;i<size;++i){

            MPI_Recv(&fx, 1, MPI_DOUBLE,i, 770, MPI_COMM_WORLD,&status);
            MPI_Recv(&jj, 1, MPI_INT,i, 771, MPI_COMM_WORLD,&status);
            MPI_Recv(&trial, 1, MPI_INT,i, 772, MPI_COMM_WORLD,&status);
            MPI_Recv(&endpt, nvars, MPI_DOUBLE,i, 773,
                MPI_COMM_WORLD,&status);
```

```
        //printf("recv:%d",i);
           if (fx < best_fx) {
            best_trial = trial;
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
               best_pt[i] = endpt[i];
         }
        }
    }else{
        //printf("recv:%d",rank);
           MPI_Send(&best_fx, 1, MPI_DOUBLE,0, 770, MPI_COMM_WORLD);
           MPI_Send(&best_jj, 1, MPI_INT,0, 771, MPI_COMM_WORLD);
           MPI_Send(&best_trial, 1, MPI_INT,0, 772, MPI_COMM_WORLD);
           MPI_Send(&best_pt, nvars, MPI_DOUBLE,0, 773, MPI_COMM_WORLD);
    }

t1 = get_wtime();
if(rank == 0 ){
   printf("\n\nFINAL RESULTS:\n");
   printf("Elapsed time = %.3lf s\n", t1-t0);
   printf("Total number of trials = %d\n", ntrials);
   printf("Total number of function evaluations = %ld\n", funevals);
   printf("Best result at trial %d used %d iterations, and returned\n",
       best_trial, best_jj);
   for (i = 0; i < nvars; i++) {
      printf("x[%3d] = %15.7le \n", i, best_pt[i]);
   }
   printf("f(x) = %15.7le\n", best_fx);
}
MPI_Finalize();
   return 0;
}
```

Εξήγηση εντολών: Η εντολή:

```
MPI_Send()
```

Στέλνει με τη χρήση ενός buffer δεδομένα προς μια άλλη διεργασία εντός του επικοινωνητή MPI_COMM_WORLD ενώ η εντολή :

```
MPI_Recv()
```

Δέχεται τα δεδομένα από την διεργασία που έκανε το MPI_Send και τα αποθηκεύει σε μία μεταβλητή.

7.1.1 6 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI (send & receive) με 6 threads:

```
costas@costas:~/Desktop/omp-omptask_code_photos/mpi$ mpirun -np 6 ./MPI_SEND_RECV


FINAL RESULTS:
Elapsed time = 129.400 s
Total number of trials = 65536
Total number of function evaluations = 2504000764
Best result at trial 3483 used 131 iterations, and returned
x[  0] =   9.9999892e-01
x[  1] =   9.9999892e-01
x[  2] =   1.0000004e+00
x[  3] =   9.9999960e-01
x[  4] =   1.0000018e+00
x[  5] =   1.0000005e+00
x[  6] =   1.0000001e+00
x[  7] =   9.9999977e-01
x[  8] =   9.9999982e-01
x[  9] =   1.0000003e+00
x[ 10] =   9.9999964e-01
x[ 11] =   9.9999938e-01
x[ 12] =   9.9999947e-01
x[ 13] =   9.9999947e-01
x[ 14] =   9.9999773e-01
x[ 15] =   1.0000004e+00
x[ 16] =   9.9999895e-01
x[ 17] =   9.9999907e-01
x[ 18] =   9.9999873e-01
x[ 19] =   9.9999881e-01
x[ 20] =   9.9999851e-01
x[ 21] =   9.9999988e-01
x[ 22] =   1.0000004e+00
x[ 23] =   9.9999956e-01
x[ 24] =   9.9999998e-01
x[ 25] =   1.0000000e+00
x[ 26] =   1.0000014e+00
x[ 27] =   1.0000027e+00
x[ 28] =   1.0000072e+00
x[ 29] =   1.0000184e+00
x[ 30] =   1.0000369e+00
x[ 31] =   1.0000722e+00
f(x) =   1.1365244e-08
```

Figure 21: Χρόνος εκτέλεσης MPI Send & Receive με 6 threads.

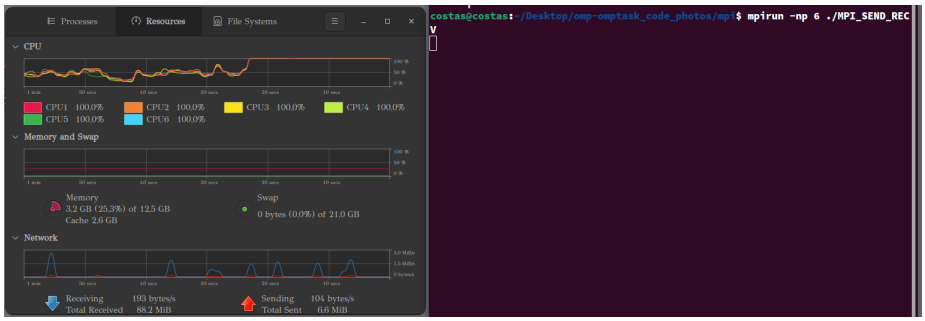

Figure 22: Cpu utilisation MPI Send & Receive με 6 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 129.4 δευτερόλεπτα δηλαδή 2.15 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 80% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(6) = \frac{666.212}{129.4} = 5.148 \tag{19}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(6) = \frac{6}{1} = 6 \tag{20}$$

Απόδοση :

$$E(6) = \frac{S(6)}{6}\% = \frac{3.908}{6}\% = 85.8\% \tag{21}$$

7.1.2 4 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI (send & receive) με 4 threads:

```
costas@costas:~/Desktop/omp-omptask_code_photos/mpi$ mpirun -np 4 ./MPI_SEND_RECV


FINAL RESULTS:
Elapsed time = 170.432 s
Total number of trials = 65536
Total number of function evaluations = 3775067890
Best result at trial 14314 used 131 iterations, and returned
x[  0] =   1.0000024e+00
x[  1] =   9.9999997e-01
x[  2] =   1.0000003e+00
x[  3] =   1.0000014e+00
x[  4] =   1.0000014e+00
x[  5] =   9.9999972e-01
x[  6] =   9.9999996e-01
x[  7] =   1.0000002e+00
x[  8] =   1.0000002e+00
x[  9] =   1.0000000e+00
x[ 10] =   9.9999933e-01
x[ 11] =   9.9999902e-01
x[ 12] =   9.9999928e-01
x[ 13] =   9.9999966e-01
x[ 14] =   1.0000003e+00
x[ 15] =   1.0000007e+00
x[ 16] =   9.9999895e-01
x[ 17] =   9.9999826e-01
x[ 18] =   9.9999849e-01
x[ 19] =   1.0000003e+00
x[ 20] =   9.9999907e-01
x[ 21] =   9.9999914e-01
x[ 22] =   9.9999895e-01
x[ 23] =   9.9999859e-01
x[ 24] =   9.9999903e-01
x[ 25] =   9.9999811e-01
x[ 26] =   9.9999673e-01
x[ 27] =   9.9999706e-01
x[ 28] =   9.9999351e-01
x[ 29] =   9.9998691e-01
x[ 30] =   9.9997351e-01
x[ 31] =   9.9994847e-01
f(x) =   9.3329672e-09
```

Figure 23: Χρόνος εκτέλεσης MPI Send & Receive με 4 threads.

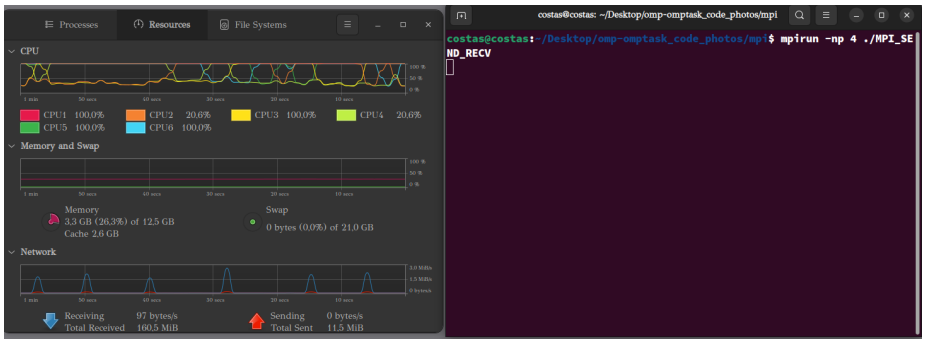

Figure 24: Cpu utilisation MPI Send & Receive με 4 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 170.432 δευτερόλεπτα δηλαδή 2.84 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 74.4% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(4) = \frac{666.212}{170.432} = 3.908 \tag{22}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(4) = \frac{4}{1} = 4 \tag{23}$$

Απόδοση :

$$E(4) = \frac{S(4)}{4}\% = \frac{3.908}{4}\% = 97.7\% \tag{24}$$

7.1.3 2 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI (send & receive) με 2 threads:

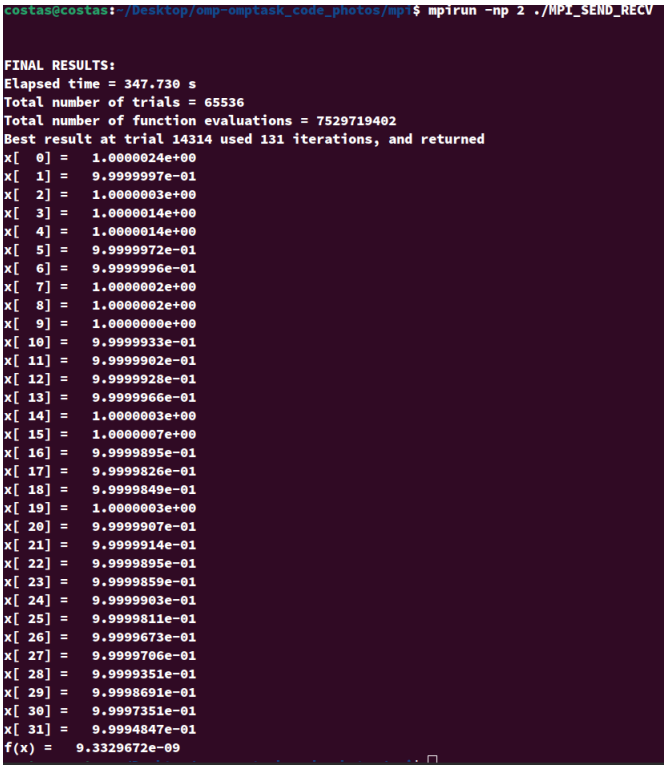

Figure 25: Χρόνος εκτέλεσης MPI Send & Receive με 2 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 347.730 δευτερόλεπτα δηλαδή 5.78 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 47.8% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(2) = \frac{666.212}{347.730} = 1.915 \tag{25}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(2) = \frac{2}{1} = 2 \tag{26}$$

Απόδοση :

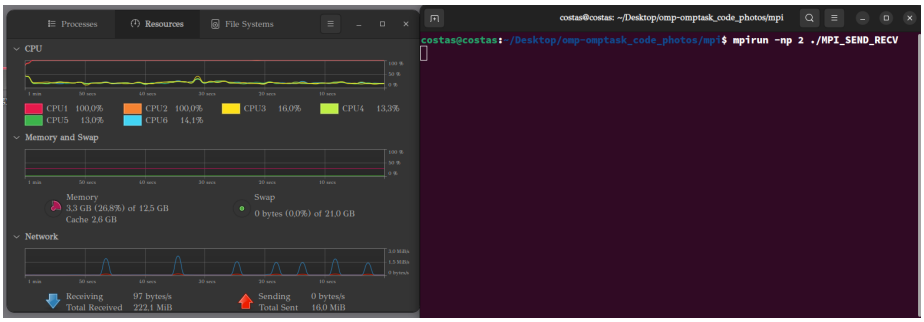

Figure 26: Cpu utilisation MPI Send & Receive με 2 threads.

$$E(2) = \frac{S(2)}{2}\% = \frac{3.908}{6}\% = 95.7\% \tag{27}$$

7.1.4 Συμπεράσματα για MPI Send & Receive

Παρατηρούμε οτι οι αποδόσεις είναι καλύτερες απο αυτές της OpenMP και αυτό οφείλεται στο γεγονός ότι δεν υπάρχουν συγκρούσεις στην κοινή μνήμη, αφού δεν υπάρχει κοινή μνήμη στην ουσία, καθώς οι διεργασίες είναι ανεξάρτητες μεταξύ τους. Μόνο στο τέλος υπάρχει μία επικοινωνία με τις εντολές MPI_Send και MPI_Recv.

MPI Send & Receive				
#Threads	Χρόνος σε seconds	Μειώση σε (%)	Efficiency	Speed up (measured)
6	129.4s	80%	85.8%	5.148
4	170.432s	74.4%	97.7%	3.908
2	347.730s	47.8%	95.7 %	1.915
Σειριακό	666.212	0	100%	1.000

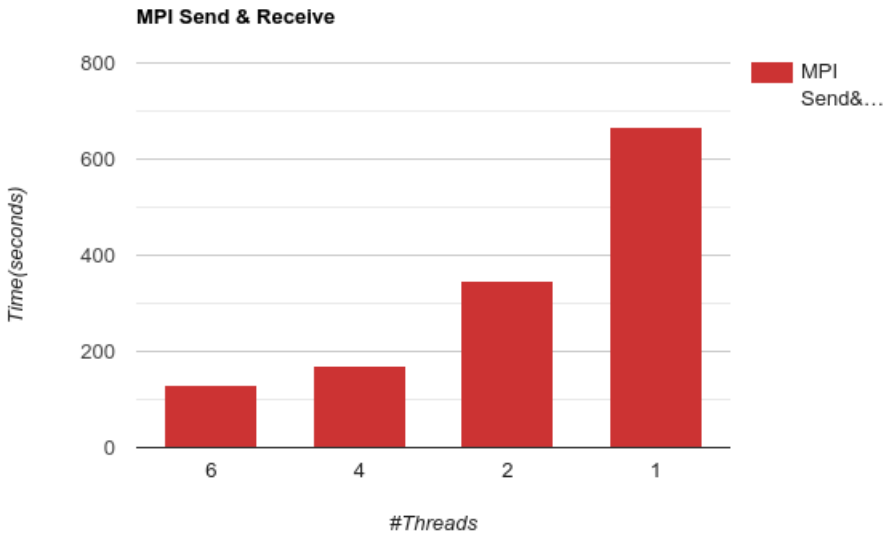

Figure 27: Bar graph of time in relation with threads (lower is better).

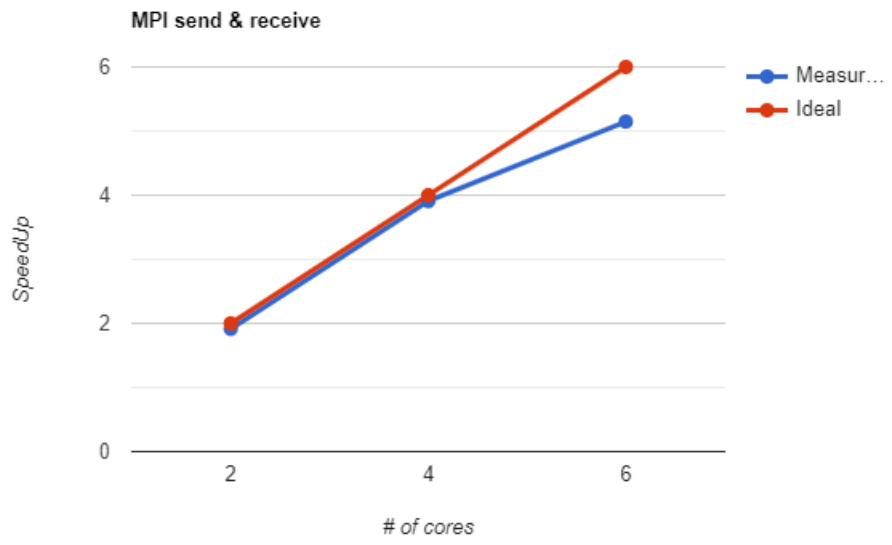

Figure 28: Line graph of theoretical speed up against measured speed up

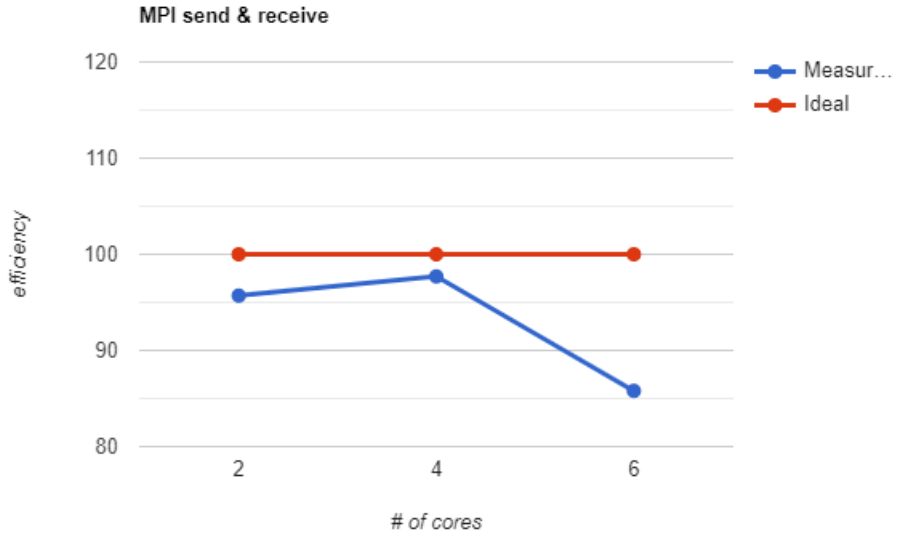

Figure 29: Line graph of theoretical efficiency against measured efficiency

7.2 MPI All reduce

Με την εντολή

```
MPI_Allreduce(&in,&out,1, MPI_FLOAT_INT, MPI_MINLOC, MPI_COMM_WORLD);
```

Με τη χρήση του operator MPI_MINLOC όλες οι διεργασίες γνωρίζουνε ποιό το καλύτερο ελάχιστο και σε ποιά διεργασία βρέθηκε, στη συνέχεια η διεργασία αυτή τυπώνει το τελικό αποτέλεσμα.

Παραθέτουμε των κώδικα της main όπου έγιναν οι αλλαγές για να παραλληλοποιηθεί ο κώδικας.

```
// MPI All reduce
int main(int argc, char *argv[])
{
   MPI_Status status;
   double startpt[MAXVARS], endpt[MAXVARS];
   int itermax = IMAX;
   double rho = RHO_BEGIN;
   double epsilon = EPSMIN;
   int nvars;
   int trial, ntrials;
   double fx;
   int i, jj;
   double t0, t1;
   int rank;
   int size;
   double best_fx = 1e10;
   double best_pt[MAXVARS];
   int best_trial = -1;
   int best_jj = -1;
    struct {
           float val;
           int   rank;
           }in,out;

   for (i = 0; i < MAXVARS; i++) best_pt[i] = 0.0;

   ntrials = 64*1024 ; /* number of trials */
   nvars = 32;   /* number of variables (problem dimension) */
   srand48(1);
   MPI_Init(&argc, &argv);
   t0 = get_wtime();

   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   double const step = (int)(ntrials+0.51) / size;
   // do just one piece on each rank
   unsigned long start = rank * step;
   unsigned long end = (rank+1) * step;
```

```
   for (trial = start; trial < end; trial++) {
      /* starting guess for rosenbrock test function, search space in
          [-5, 5) */

      for (i = 0; i < nvars; i++) {
         startpt[i] = 10.0*drand48()-5.0;
      }

      jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
#if DEBUG
      printf("\n\n\nHOOKE %d USED %d ITERATIONS, AND RETURNED\n", trial,
          jj);
      for (i = 0; i < nvars; i++)
         printf("x[%3d] = %15.7le \n", i, endpt[i]);
#endif

      fx = f(endpt, nvars);
#if DEBUG
      printf("f(x) = %15.7le\n", fx);
#endif
      if (fx < best_fx) {
         best_trial = trial;
         best_jj = jj;
         best_fx = fx;
         for (i = 0; i < nvars; i++)
            best_pt[i] = endpt[i];
      }
   }
t1 = get_wtime();
//printf("Elapsed time = %.3lf s\n""recv:%d", t1-t0,i);
in.val = best_fx;
in.rank = rank;
int best_rank;
MPI_Allreduce(&in,&out,1, MPI_FLOAT_INT, MPI_MINLOC, MPI_COMM_WORLD);
t1 = get_wtime();
best_rank = out.rank;
if(rank == best_rank ){
   printf("\n\nFINAL RESULTS:\n");
   printf("Elapsed time = %.3lf s\n", t1-t0);
   printf("Total number of trials = %d\n", ntrials);
   printf("Total number of function evaluations = %ld\n", funevals);
   printf("Best result at trial %d used %d iterations, and returned\n",
       best_trial, best_jj);
   for (i = 0; i < nvars; i++) {
      printf("x[%3d] = %15.7le \n", i, best_pt[i]);
   }
   printf("f(x) = %15.7le\n", best_fx);
}
MPI_Finalize();
```

```
   return 0;
}
```

Η εντολή:

```
MPI_Allreduce(&in,&out,1, MPI_FLOAT_INT, MPI_MINLOC, MPI_COMM_WORLD);
```

Μαζέβει από όλες τις διεργασίες και βρίσκει την ελάχιστη τιμή του buffer in και επιστρέφει σε όλες τις διεργασίες εντός του επικοινωνητή MPI_COMM_WORLD τόσο το ελάχιστο όσο και το rank της διεργασίας που το έχει.

7.2.1 6 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI All reduce με 6 threads:

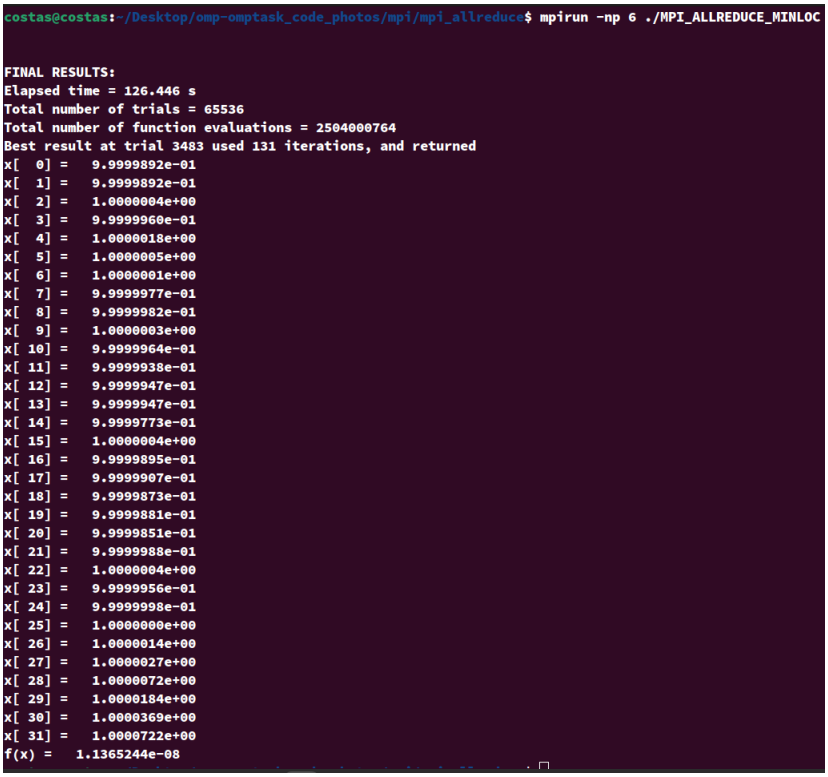

Figure 30: Χρόνος εκτέλεσης MPI All reduce με 6 threads.

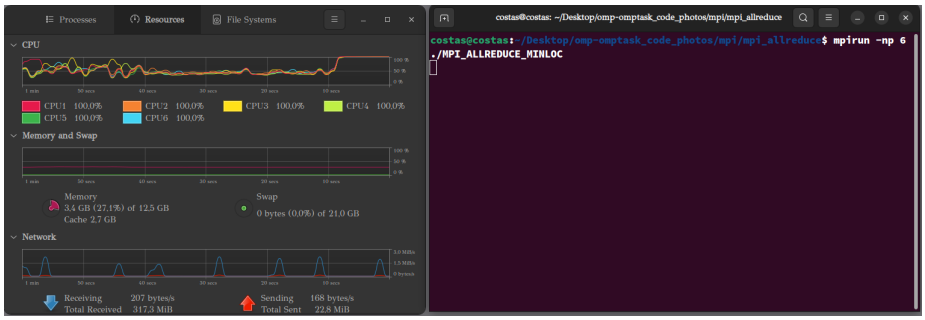

Figure 31: Cpu utilisation MPI All reduce με 6 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 126.446 δευτερόλεπτα δηλαδή 2.10 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 81% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(6) = \frac{666.212}{126.446} = 5.268 \tag{28}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(6) = \frac{6}{1} = 6 \tag{29}$$

Απόδοση :

$$E(6) = \frac{S(6)}{6}\% = \frac{5.268}{6}\% = 87.8\% \tag{30}$$

7.2.2 4 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI All reduce με 4 threads:

```
costas@costas:~/Desktop/omp-omptask_code_photos/mpi/mpi_allreduce$ mpirun -np 4 ./MPI_ALLREDUCE_MINLOC


FINAL RESULTS:
Elapsed time = 170.926 s
Total number of trials = 65536
Total number of function evaluations = 3775067890
Best result at trial 14314 used 131 iterations, and returned
x[  0] =   1.0000024e+00
x[  1] =   9.9999997e-01
x[  2] =   1.0000003e+00
x[  3] =   1.0000014e+00
x[  4] =   1.0000014e+00
x[  5] =   9.9999972e-01
x[  6] =   9.9999996e-01
x[  7] =   1.0000002e+00
x[  8] =   1.0000002e+00
x[  9] =   1.0000000e+00
x[ 10] =   9.9999933e-01
x[ 11] =   9.9999902e-01
x[ 12] =   9.9999928e-01
x[ 13] =   9.9999966e-01
x[ 14] =   1.0000003e+00
x[ 15] =   1.0000007e+00
x[ 16] =   9.9999895e-01
x[ 17] =   9.9999826e-01
x[ 18] =   9.9999849e-01
x[ 19] =   1.0000003e+00
x[ 20] =   9.9999907e-01
x[ 21] =   9.9999914e-01
x[ 22] =   9.9999895e-01
x[ 23] =   9.9999859e-01
x[ 24] =   9.9999903e-01
x[ 25] =   9.9999811e-01
x[ 26] =   9.9999673e-01
x[ 27] =   9.9999706e-01
x[ 28] =   9.9999351e-01
x[ 29] =   9.9998691e-01
x[ 30] =   9.9997351e-01
x[ 31] =   9.9994847e-01
f(x) =   9.3329672e-09
```

Figure 32: Χρόνος εκτέλεσης MPI All reduce με 4 threads.

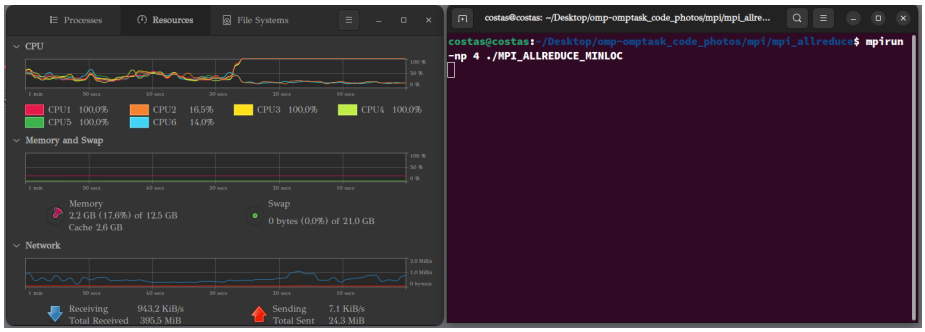

Figure 33: Cpu utilisation MPI All reduce με 4 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 170.926 δευτερόλεπτα δηλαδή 2.84 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 74.3% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(4) = \frac{666.212}{170.926} = 3.897 \tag{31}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(4) = \frac{4}{1} = 4 \tag{32}$$

Απόδοση :

$$E(4) = \frac{S(4)}{4}\% = \frac{3.897}{4}\% = 97.4\% \tag{33}$$

7.2.3 2 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση με MPI All reduce με 2 threads:

```
costas@costas:~/Desktop/omp-omptask_code_photos/mpi/mpi_allreduce$ mpirun -np 2 ./MPI_ALLREDUCE_MINLOC


FINAL RESULTS:
Elapsed time = 347.444 s
Total number of trials = 65536
Total number of function evaluations = 7529719402
Best result at trial 14314 used 131 iterations, and returned
x[  0] =   1.0000024e+00
x[  1] =   9.9999997e-01
x[  2] =   1.0000003e+00
x[  3] =   1.0000014e+00
x[  4] =   1.0000014e+00
x[  5] =   9.9999972e-01
x[  6] =   9.9999996e-01
x[  7] =   1.0000002e+00
x[  8] =   1.0000002e+00
x[  9] =   1.0000000e+00
x[ 10] =   9.9999933e-01
x[ 11] =   9.9999902e-01
x[ 12] =   9.9999928e-01
x[ 13] =   9.9999966e-01
x[ 14] =   1.0000003e+00
x[ 15] =   1.0000007e+00
x[ 16] =   9.9999895e-01
x[ 17] =   9.9999826e-01
x[ 18] =   9.9999849e-01
x[ 19] =   1.0000003e+00
x[ 20] =   9.9999907e-01
x[ 21] =   9.9999914e-01
x[ 22] =   9.9999895e-01
x[ 23] =   9.9999859e-01
x[ 24] =   9.9999903e-01
x[ 25] =   9.9999811e-01
x[ 26] =   9.9999673e-01
x[ 27] =   9.9999706e-01
x[ 28] =   9.9999351e-01
x[ 29] =   9.9998691e-01
x[ 30] =   9.9997351e-01
x[ 31] =   9.9994847e-01
f(x) =   9.3329672e-09
```

Figure 34: Χρόνος εκτέλεσης MPI All reduce με 2 threads.

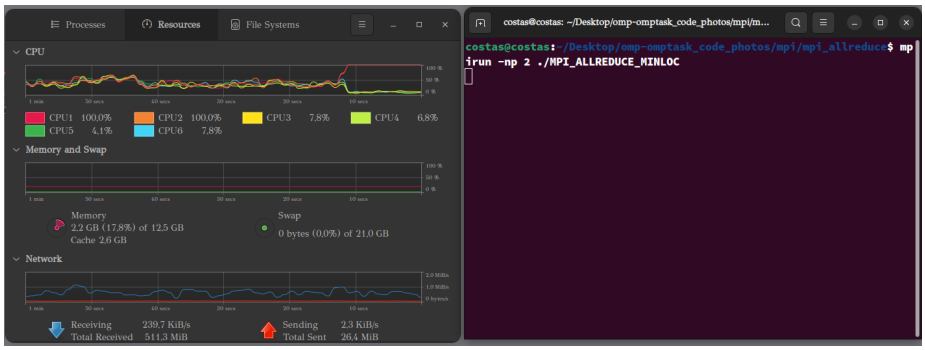

Figure 35: Cpu utilisation MPI All reduce με 2 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 347.444 δευτερόλεπτα δηλαδή 5.79 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 47.8% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(2) = \frac{666.212}{347.444} = 1.917 \tag{34}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(2) = \frac{2}{1} = 2 \tag{35}$$

Απόδοση :

$$E(2) = \frac{S(2)}{2}\% = \frac{1.917}{2}\% = 95.8\% \tag{36}$$

7.2.4 Συμπεράσματα για MPI All reduce

Παρατηρούμε ελαφρώς καλύτερες μετρικές σε σχέση με τη μέθοδο MPI Send & Recv καθώς η All reduce έχει καλύτερο optimisation από τον compiler και γενικότερα προσθέτει λιγότερο overhead από Send Recv μέσα σε μια For Loop.

MPI All reduce				
#Threads	Χρόνος σε seconds	Μειώση σε (%)	Efficiency	Speed up (measured)
6	126.446s	81%	87.8%	5.268
4	170.926s	74.3%	97.4%	3.897
2	347.444s	47.8%	95.8 %	1.917
Σειριακό	666.212	0	100%	1.000

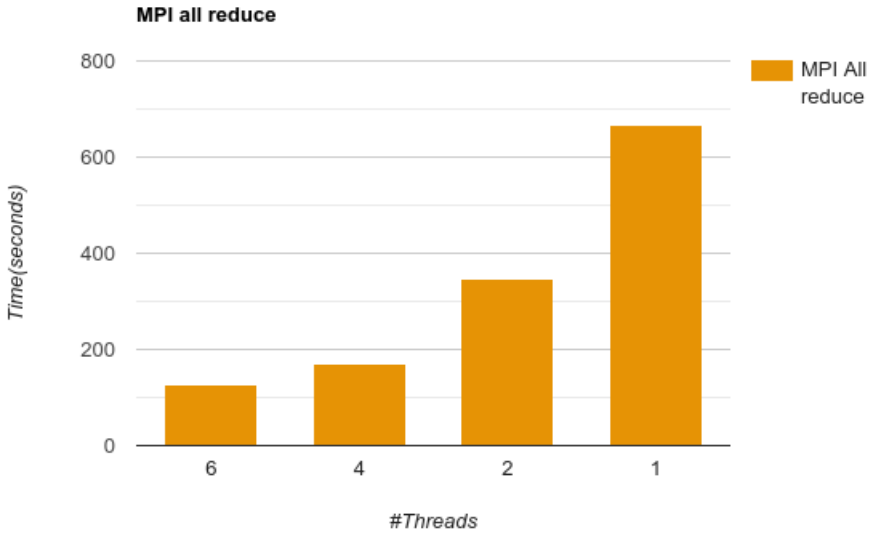

Figure 36: Bar graph of time in relation with threads (lower is better).

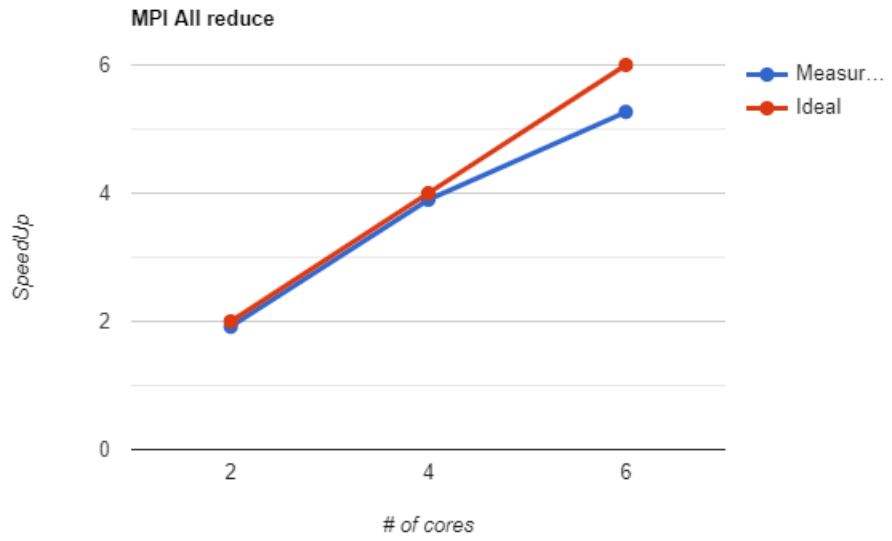

Figure 37: Line graph of theoretical speed up against measured speed up

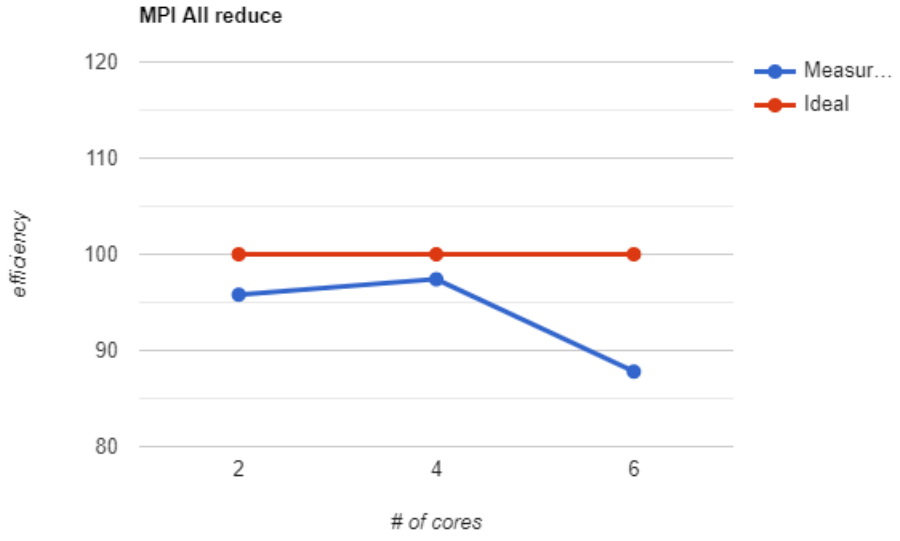

Figure 38: Line graph of theoretical efficiency against measured efficiency

8 Hybrid (MPI + OpenMP)

Σε αυτή την υλοποίηση χρησομοποιήσαμε τόσο την MPI όσο και την OpenMP για να επιτύχουμε την καλύτερη απόδοση. Αυτή η υβριδική προσέγγιση χρησιμοποιείται κυριώς σε clusters αλλά ακόμη και σε ένα υπολογιστικό σύστημα με ένα πολυπήρυνο επεξεργαστή είναι πιο γρήγορη από το απλό MPI η το απλό OpenMP.

Παραθέτουμε των κώδικα της main όπου έγιναν οι αλλαγές για να παραλληλοποιηθεί ο κώδικας.

```
// Hybrid
int main(int argc, char *argv[])
{
   MPI_Status status;
   double startpt[MAXVARS], endpt[MAXVARS];
   int itermax = IMAX;
   double rho = RHO_BEGIN;
   double epsilon = EPSMIN;
   int nvars;
   int trial, ntrials;
   double fx;
   int i, jj;
   double t0, t1;
   int rank;
   int size;
   double best_fx = 1e10;
   double best_pt[MAXVARS];
   int best_trial = -1;
   int best_jj = -1;
    struct {
           float val;
           int   rank;
           }in,out;

   for (i = 0; i < MAXVARS; i++) best_pt[i] = 0.0;

   ntrials = 64*1024 ; /* number of trials */
   nvars = 32;   /* number of variables (problem dimension) */
   srand48(1);
   int provided;
   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED,&provided);
   t0 = get_wtime();


   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   double const step = (int)(ntrials+0.51) / size;
   // do just one piece on each rank
   unsigned long start = rank * step;
```

```
   unsigned long end = (rank+1) * step;
#pragma omp parallel for //dhmiourgia thread kai moirasmos twn
    epanalhpsewn se auta.
   for (trial = start; trial < end; trial++) {
     /* starting guess for rosenbrock test function, search space in
         [-5, 5) */

     for (i = 0; i < nvars; i++) {
        startpt[i] = 10.0*drand48()-5.0;
     }

     jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
#if DEBUG
     printf("\n\n\nHOOKE %d USED %d ITERATIONS, AND RETURNED\n", trial,
         jj);
     for (i = 0; i < nvars; i++)
        printf("x[%3d] = %15.7le \n", i, endpt[i]);
#endif

     fx = f(endpt, nvars);
#if DEBUG
     printf("f(x) = %15.7le\n", fx);
#endif
     if (fx < best_fx) {
        best_trial = trial;
        best_jj = jj;
        best_fx = fx;
        for (i = 0; i < nvars; i++)
           best_pt[i] = endpt[i];
     }
   }
t1 = get_wtime();
//printf("Elapsed time = %.3lf s\n""recv:%d", t1-t0,i);
in.val = best_fx;
in.rank = rank;
int best_rank;
MPI_Allreduce(&in,&out,1, MPI_FLOAT_INT, MPI_MINLOC, MPI_COMM_WORLD);
t1 = get_wtime();
best_rank = out.rank;
if(rank == best_rank ){
   printf("\n\nFINAL RESULTS:\n");
   printf("Elapsed time = %.3lf s\n", t1-t0);
   printf("Total number of trials = %d\n", ntrials);
   printf("Total number of function evaluations = %ld\n", funevals);
   printf("Best result at trial %d used %d iterations, and returned\n",
       best_trial, best_jj);
   for (i = 0; i < nvars; i++) {
      printf("x[%3d] = %15.7le \n", i, best_pt[i]);
   }
   printf("f(x) = %15.7le\n", best_fx);
```

```
}
MPI_Finalize();
   return 0;
}
```

8.1 6 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση Hybrid με 6 threads:

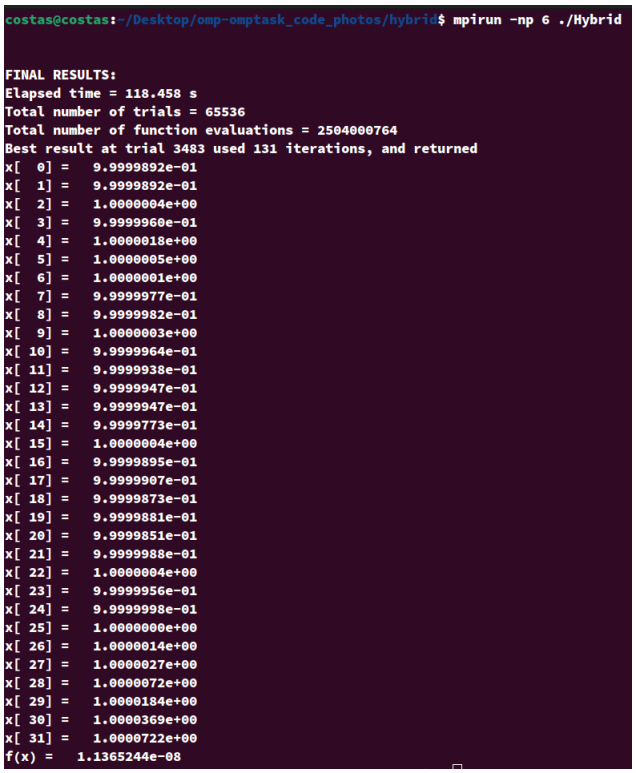

Figure 39: Χρόνος εκτέλεσης Hybrid με 6 threads.

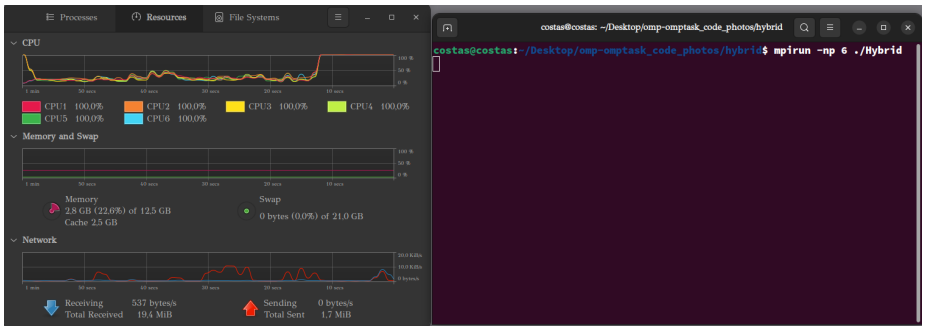

Figure 40: Cpu utilisation Hybrid με 6 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 118.458 δευτερόλεπτα δηλαδή 1.97 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 82.2% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(6) = \frac{666.212}{118.458} = 5.624 \tag{37}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(6) = \frac{6}{1} = 6 \tag{38}$$

Απόδοση :

$$E(6) = \frac{S(6)}{6}\% = \frac{5.624}{6}\% = 93.7\% \tag{39}$$

8.2 4 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση Hybrid με 4 threads:

```
costas@costas:~/Desktop/omp-omptask_code_photos/hybrid$ mpirun -np 4 ./Hybrid


FINAL RESULTS:
Elapsed time = 172.068 s
Total number of trials = 65536
Total number of function evaluations = 3775067890
Best result at trial 14314 used 131 iterations, and returned
x[  0] =   1.0000024e+00
x[  1] =   9.9999997e-01
x[  2] =   1.0000003e+00
x[  3] =   1.0000014e+00
x[  4] =   1.0000014e+00
x[  5] =   9.9999972e-01
x[  6] =   9.9999996e-01
x[  7] =   1.0000002e+00
x[  8] =   1.0000002e+00
x[  9] =   1.0000000e+00
x[ 10] =   9.9999933e-01
x[ 11] =   9.9999902e-01
x[ 12] =   9.9999928e-01
x[ 13] =   9.9999966e-01
x[ 14] =   1.0000003e+00
x[ 15] =   1.0000007e+00
x[ 16] =   9.9999895e-01
x[ 17] =   9.9999826e-01
x[ 18] =   9.9999849e-01
x[ 19] =   1.0000003e+00
x[ 20] =   9.9999907e-01
x[ 21] =   9.9999914e-01
x[ 22] =   9.9999895e-01
x[ 23] =   9.9999859e-01
x[ 24] =   9.9999903e-01
x[ 25] =   9.9999811e-01
x[ 26] =   9.9999673e-01
x[ 27] =   9.9999706e-01
x[ 28] =   9.9999351e-01
x[ 29] =   9.9998691e-01
x[ 30] =   9.9997351e-01
x[ 31] =   9.9994847e-01
f(x) =   9.3329672e-09
```

Figure 41: Χρόνος εκτέλεσης Hybrid με 4 threads.

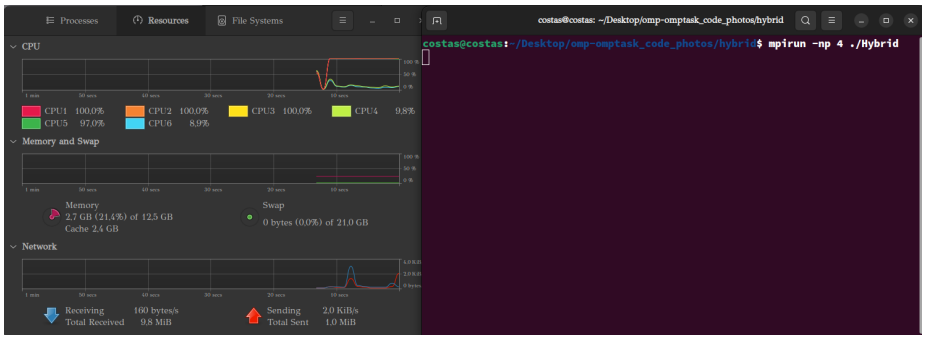

Figure 42: Cpu utilisation Hybrid με 4 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 172.068 δευτερόλεπτα δηλαδή 2.86 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 74.1% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(4) = \frac{666.212}{172.068} = 3.871 \tag{40}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(4) = \frac{4}{1} = 4 \tag{41}$$

Απόδοση :

$$E(4) = \frac{S(4)}{4}\% = \frac{5.624}{6}\% = 96.7\% \tag{42}$$

8.3 2 threads

Παραθέτουμε χρόνους για την παράλληλη υλοποίηση Hybrid με 2 threads:

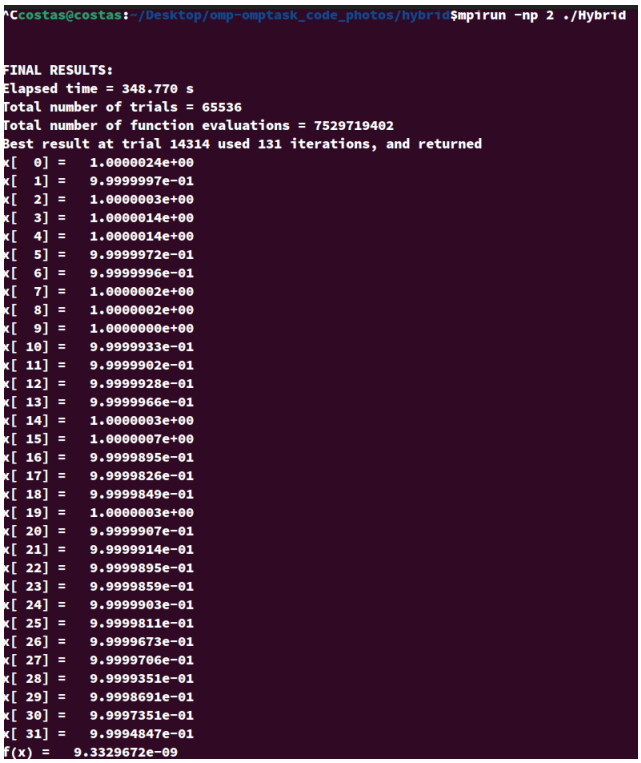

Figure 43: Χρόνος εκτέλεσης Hybrid με 2 threads.

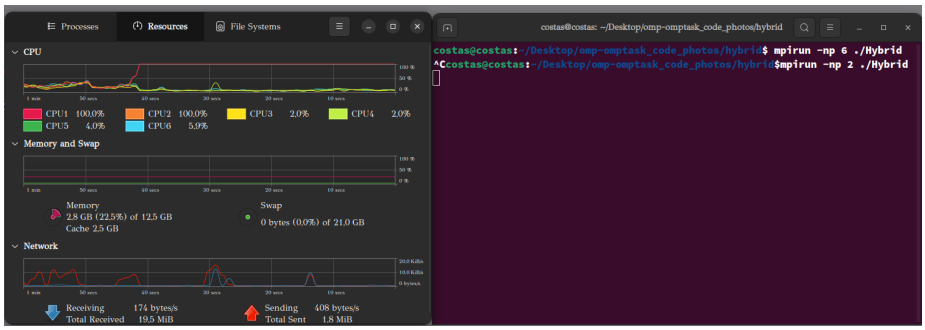

Figure 44: Cpu utilisation Hybrid με 2 threads.

Παρατηρούμε πώς ο χρόνος εκτέλεσης είναι 348.770 δευτερόλεπτα δηλαδή 5.81 λεπτά. Αυτό σημαίνει πώς ο χρόνος μειώθηκε κατα 47.6% σε σχέση με τη σειριακή υλοποίηση.

Παράγοντας επιτάχυνσης :

$$S(2) = \frac{666.212}{348.770} = 1.910 \tag{43}$$

Θεωριτικός παράγοντας επιτάχυνσης:

$$S(2) = \frac{2}{1} = 2 \tag{44}$$

Απόδοση :

$$E(2) = \frac{S(2)}{2}\% = \frac{1.910}{2}\% = 95.5\% \tag{45}$$

8.4 Συμπεράσματα για Hybrid

Παρατηρούμε πώς οι μετρικές είναι καλύτερες από όλες τις άλλες υλοποιήσεις με τη μεγαλύτερη βελτίωση σε efficiency όπου αυξήθηκε κατα 5% στην εκτέλεση με 6 threads σε σχέση με τη προηγούμενη καλύτερη που ήταν η MPI.

Hybrid				
#Threads	Χρόνος σε seconds	Μειώση σε (%)	Efficiency	Speed up (measured)
6	118.458s	82.2%	93.7%	5.624
4	172.068s	74.1%	96.7%	3.871
2	348.770s	47.6%	95.5 %	1.910
Σειριακό	666.212	0	100%	1.000

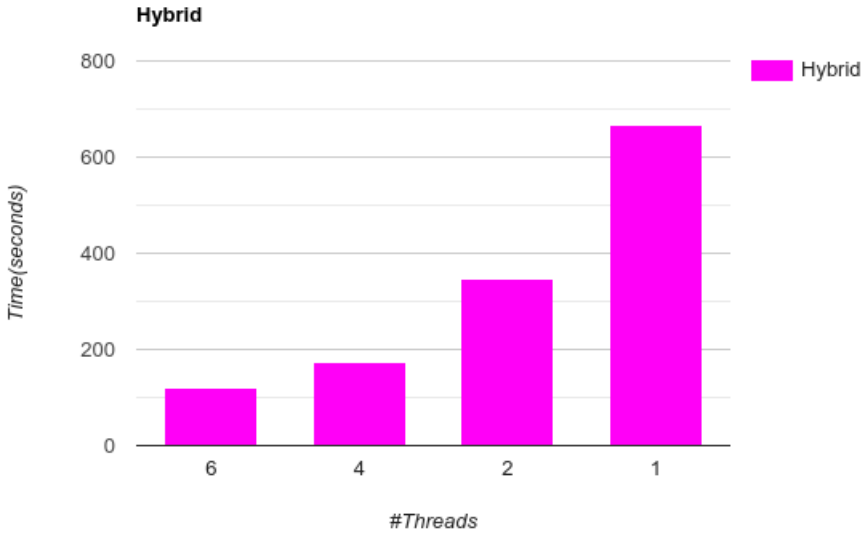

Figure 45: Bar graph of time in relation with threads (lower is better).

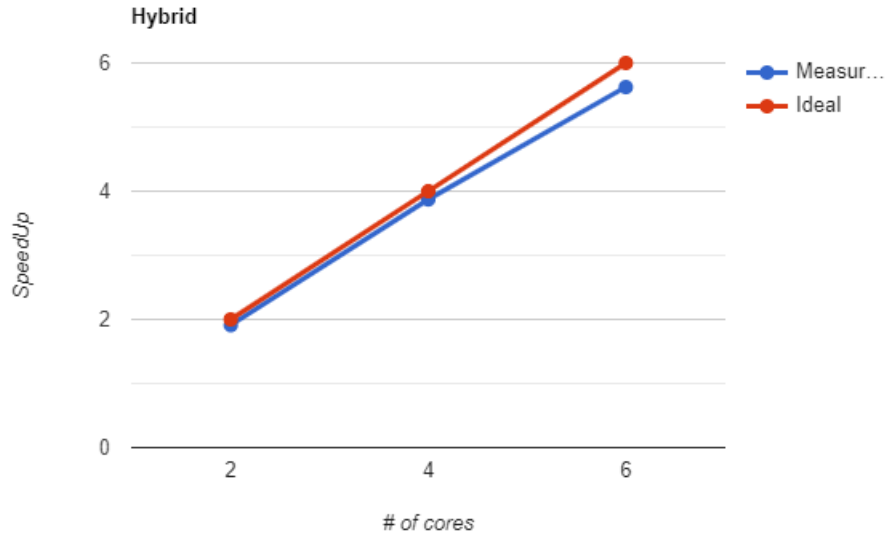

Figure 46: Line graph of theoretical speed up against measured speed up

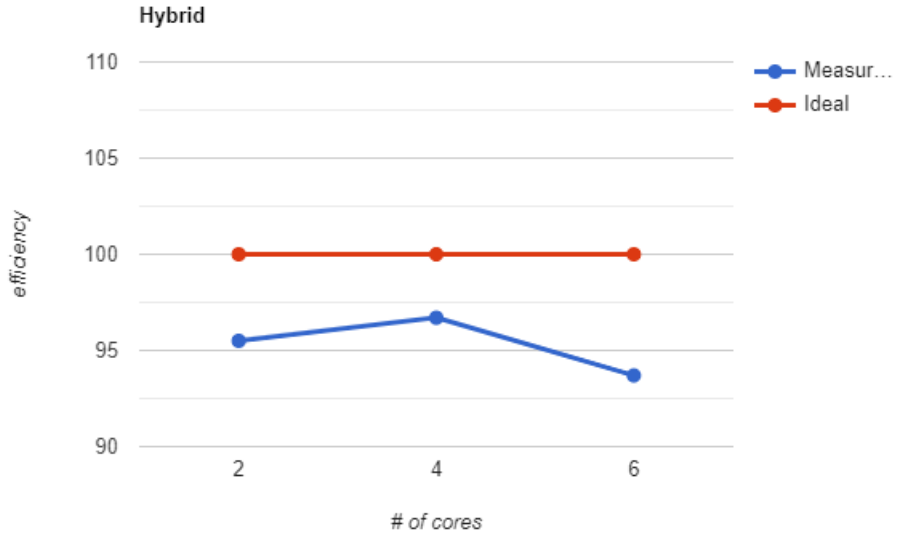

Figure 47: Line graph of theoretical efficiency against measured efficiency

9 Τελικά συμπεράσματα

Πιο έυκολη στην υλοποίηση ήταν η OpenMP άλλα ήταν επίσης η πιο αργή σε σχέση με τις υπόλοιπες υλοποιήσεις καθώς επίσης δεν υποστηρίζει τη χρήση clusters.

Η χρήση των task στην OpenMP βελτίωσε ελάχιστα τους χρόνους.Η υλοποίηση της παραλληλοποίησης με MPI απαιτούσε αλλαγές στο σειριακό κώδικα αλλά είχε καλύτερη επίδοση από την OpenMP ενώ επιτρέπει και μεγάλο scaleability με τη χρήση clusters. Τέλος η καλύτερη αλλά πιο πολύπλοκη υλοποίηση ήταν η υβριδική προσσέγγιση χρησιμοποιώντας και MPI και OpenMP προσφέροντας τους καλύτερους χρόνους.

9.1 Μετρήσεις

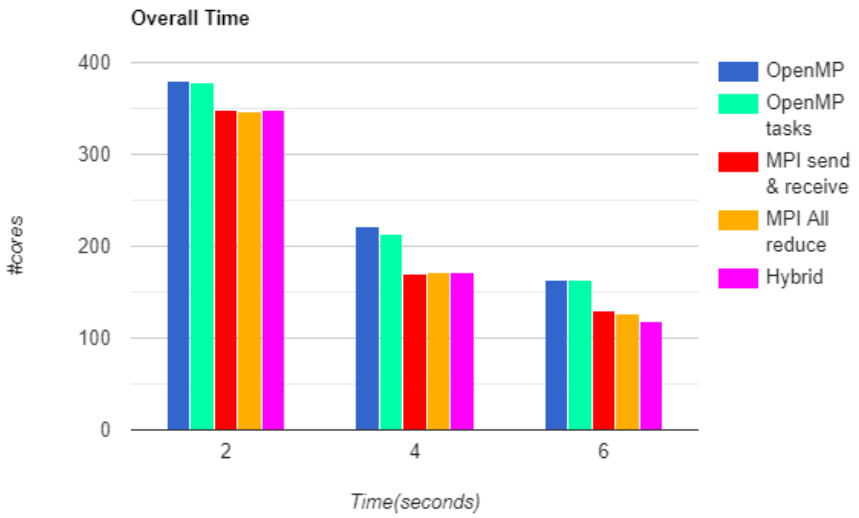

Figure 48: Bar graph of time in relation with threads (lower is better).

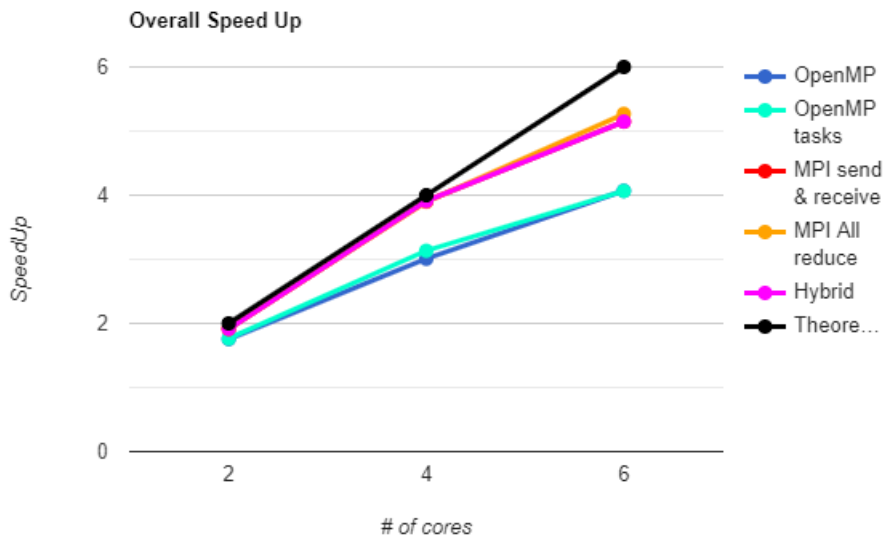

Figure 49: Line graph of theoretical speed up against measured speed up

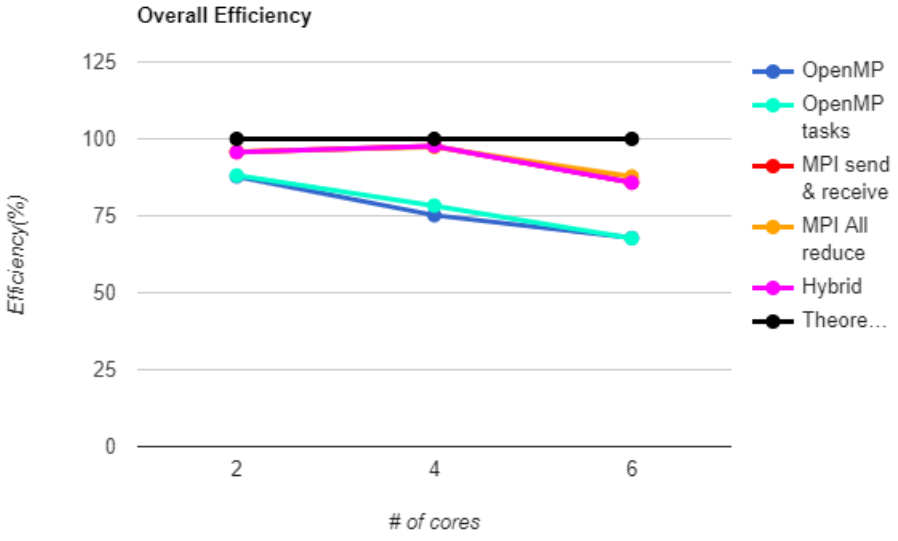

Figure 50: Line graph of theoretical efficiency against measured efficiency