Project Παράλληλης Επεξεργασίας 2021-22

Συμμετέχοντες ομάδας:

1) Βλάσιος Παναγιώτης Παναγιώτου , ΑΜ: 1067517
2) Σοφία Λαμπροπούλου , ΑΜ: 1072606
3) Κωνσταντίνος Παρασκευόπουλος, ΑΜ: 1072608
4) Δανάη Χαλούλου , ΑΜ: 1072596

Εισαγωγή

Για την υλοποίηση της εργασίας χρησιμοποιήθηκε υπολογιστής με 4 πυρήνες, 8 νήματα και 2,8 GHz συχνότητα. Ωστόσο επειδή η υλοποίηση των προγραμμάτων έγινε σε virtual machine χρησιμοποιήθηκαν μόνο δύο πυρήνες και 4 νήματα. Εξαιτίας του virtual machine υπήρχε περιορισμός στην χρήση της RAM στα 4 GB.

Παρατηρώντας τον σειριακό κώδικα που μας δόθηκε, η καθυστέρηση στον υπολογισμό οφείλεται στα μεγάλα for loops για όλα τα trials στην συνάρτηση main. Οπότε σε αυτό το σημείο θα γίνει το μεγαλύτερο μέρος της παραλληλοποίησης. Επίσης η υλοποίηση χρησιμοποιεί και γεννήτρια τυχαίων αριθμών η οποία εκ φύσεως της προκαλεί καθυστερήσεις στον υπολογισμό, η οποία μάλλον ωστόσο μας δίνει την δυνατότητα σε ορισμένα σημεία να μην μας ενδιαφέρει η σειρά εκτέλεσης.

Εξήγηση σειριακού κώδικα

Η συνάρτηση double f(double *x, int n) είναι ουσιαστικά η συνάρτηση f(x) (Rosenbrock) που δίνεται στην εκφώνηση, και υπολογίζει ένα σημείο Χ (το σημείο Χ αναπαρίσταται από ένα n-διάστατο διάνυσμα) στο οποίο η μη-γραμμική συνάρτηση f(x) έχει ένα τοπικό ελάχιστο.
Η συνάρτηση f(x) έχει καθολικό ελάχιστο στο σημείο (1,1,...,1) (n-διάστατο σημείο), όπου παίρνει την τιμή 0.
Για την εύρεση ενός τοπικού ελαχίστου πραγματοποιείται ευθύγραμμη αναζήτηση από το τρέχον σημείο προς συγκεκριμένες διευθύνσεις.
Εφόσον βρεθεί κάποιο νέο σημείο με καλύτερη συναρτησιακή τιμή τότε η μέθοδος αντικαθιστά το τρέχον σημείο με το νέο.
Διαφορετικά, προσαρμόζει τις παραμέτρους της ευθύγραμμης αναζήτησης και επαναλαμβάνει τη διαδικασία στο ίδιο σημείο.
Η συνάρτηση int hooke(int nvars, double startpt[MAXVARS], double endpt[MAXVARS], double rho, double epsilon, int itermax) κάνει αυτή την ευθύγραμμη αναζήτηση.

Στην αρχή, όμως, του προγράμματος , μέσω της υπορουτίνας/συνάρτησης double best_nearby() γίνεται ένας αρχικός τυχαιοκρατικός υπολογισμός ενός τυχαίου τοπικού ελαχίστου Χ.

Η συνάρτηση int hooke() ξεκινάει από αυτό το αρχικό τυχαίο τοπικό ελάχιστο και κάνει ευθύγραμμη αναζήτηση προς συγκεκριμένες διευθύνσεις που καθορίζονται μέσα στη συνάρτηση int hooke.

Αναλυτική τεκμηρίωση παράλληλων υλοποιήσεων και τυχόν βελτιστοποιήσεων

OpenMP

Αρχικά χρησιμοποιούμε την βιβλιοθήκη omp.h.

Η βελτιστοποίηση έγινε για τον κύριο βρόχο for ο οποίος προκαλεί και την μεγαλύτερη καθυστέρηση. Επίσης χρησιμοποιούμε τις μεταβλητές fx, jj, I , οι οποίες είναι private για κάθε νήμα. Και τέλος την μεταβλητή num_of_threads η οποία θα ορίζει πόσα νήματα θα δημιουργηθούν κάθε φορά. Εκτός αυτού του βρόχου μπορούσε να γίνει παραλληλοποίηση και της for που υπολογίζει με τυχαιοκρατικό τρόπο την συνάρτηση Rosenbrock με πεδίο αναζήτησης το [-5, 5). Ακόμη δημιουργήσαμε ένα critical section, ώστε να μας επιστρέφονται τα τελικά αποτελέσματα με ασφάλεια και να μην υπάρχει πιθανότητα δυο νήματα να γράψουν στην ίδια μεταβλητή όπως η best_trial. Τέλος για να βρούμε τον πραγματικό αριθμό των πράξεων πολλαπλασιάζουμε την μεταβλητή funevals με τον συνολικό αριθμό των νημάτων num_of_threads, αυτή η τακτική χρησιμοποιείται σε όλα τα πρωτόκολλα παραλληλοποίησης.

Γενικά η παραλληλοποίηση

```
#pragma omp parallel num_threads(num_of_threads) firstprivate(fx, jj, i) /*Εκκίνηση παραλληλισμού και δημιουργία νημάτων με την παράμετρο num_threads(int),
επίσης χρησιμοποιούμε την μέθοδο firstprivate για να ορίσουμε τις μεταβλητές που δεν θέλουμε να διαμοιράζονται μεταξύ των threads*/
{
    #pragma omp for /*Παραλληλοποίηση βρόχου*/
       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
       #pragma omp critical /*mutex περιοχή για τις μεταβλητές που μας επιστρέφουν το τελικό αποτέλεσμα, λ.χ. αν έχουμε 2 νέα best ταυτόχρονα θα υπάρξει πρόβλημα*/
       {
           if (fx < best_fx) {
               best_trial = trial;
               best_jj = jj;
               best_fx = fx;
               for (i = 0; i < nvars; i++)
                   best_pt[i] = endpt[i];
           }
       }
   }
}
```

Παραλληλοποίηση βρόχου

```
#pragma omp for /*Παραλληλοποίηση βρόχου*/
   for (trial = 0; trial < ntrials; trial++) {
```

Critical Section

```
#pragma omp critical /*mutex περιοχή για τις μεταβλητές που μας επιστρέφουν το τελικό αποτέλεσμα, λ.χ. αν έχουμε 2 νέα best ταυτόχρονα θα υπάρξει πρόβλημα*/
{
       if (fx < best_fx) {
           best_trial = trial;
           best_jj = jj;
           best_fx = fx;
           for (i = 0; i < nvars; i++)
               best_pt[i] = endpt[i];
       }
   }
```

Εύρεση συνολικού αριθμού υπολογισμών συνάρτησης

```
printf("Total number of function evaluations = %ld\n", funevals * num_of_threads); /*Πολλαπλασιασμό με αριθμό threads για να βρούμε το πραγματικό αριθμό των πράξεων (αφού το αρχικό ποσό μοιράζεται σε n threads*/
```

OpenMP Tasks

Αρχικά χρησιμοποιούμε την βιβλιοθήκη omp.h.

Η βελτιστοποίηση έγινε για τον κύριο βρόχο for ο οποίος προκαλεί και την μεγαλύτερη καθυστέρηση. Επίσης χρησιμοποιούμε τις μεταβλητές fx, jj, I , οι οποίες είναι private για κάθε νήμα. Και τέλος την μεταβλητή num_of_threads η οποία θα ορίζει πόσα νήματα θα δημιουργηθούν κάθε φορά. Εκτός αυτού του βρόχου μπορούσε να γίνει παραλληλοποίηση και της for που υπολογίζει με τυχαιοκρατικό τρόπο την συνάρτηση Rosenbrock με πεδίο αναζήτησης το [-5, 5). Εντός του κύριου βρόχου for γράψαμε ένα block κώδικα που μπορεί να χρησιμοποιηθεί από οποιοδήποτε νήμα δίχως να περιμένει να τελειώσουν τα υπόλοιπα, το οποίο μας βοηθάει σε αυτήν την περίπτωση διότι βρισκόμαστε εντός ενός for loop περιορισμένου αριθμού επαναλήψεων. Ακόμη, εντός του κύριου βρόχου for, δημιουργήσαμε μια περιοχή δυαδικού σημαφόρου (mutex area) για τις μεταβλητές που μας δίνουν το τελικό αποτέλεσμα ώστε να εξασφαλιστεί η άρτια εκτέλεση του προγράμματος.

Γενικά η παραλληλοποίηση

```
#pragma omp parallel number_of_threads(number_of_threads) // έναρξη του παραλληλισμού και δημιουργία νημάτων μεσω της ακέραιας (int) παραμετρου number_of_threads. Κανουμε χρηση της συναρτησης/μεθοδου firstprivate ωστε
//                                                        // να ορισουμε τις μεταβλητες που δεν επιθυμουμε να διαμοιραζονται μεταξυ των νηματων
{
{
    #pragma omp single nowait // σε αυτο το σημειο λεμε οτι το ακολουθο block απο κωδικα δυναται να γινει απο οποιοδηποτε νημα διχως να αναμενει να τελειωσουν και τα αλλα (σχολιο: αυτο ειναι εξαιρετικα βοηθητικο σε loops
    //                                                  //  που δεν εχουν ορισμενο καποιον συγκεκριμενο αριθμο επαναληψεων (repetitions)   )
        for (trial = 0; trial < ntrials; trial++) {
            #pragma omp task firstprivate(fx, jj, i) // // εναρξη tasks ωστε να πραγματοποιηθει υπολογισμος της συναρτησης
            {
                /* 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
                #pragma omp critical /* mutex area (περιοχη δυαδικου σημαφορου) για τις μεταβλητες που μας δινουν το τελικο αποτελεσμα , για παραδειγμα αν εχουμε 2 νεα best ταυτοχρονως, τοτε θα εμφανιστει θεμα/προβλημα
                                    στην εκτελεση του προγραμματος */
                {
                    if (fx < best_fx) {
                        best_trial = trial;
                        best_jj = jj;
                        best_fx = fx;
                        for (i = 0; i < nvars; i++)
                            best_pt[i] = endpt[i];
                    }
                }
            }
        }
}
```

Έναρξη παραλληλισμού και δημιουργία νημάτων μέσω της ακέραιας παραμέτρου number_of_threads

```
#pragma omp parallel number_of_threads(number_of_threads)
//                                                      // να
{
```

Έναρξη block κώδικα που δύναται να γίνει από οποιοδήποτε νήμα δίχως να αναμένει να τελειώσουν και τα άλλα

```
#pragma omp single nowait // α
```

Χρήση της συνάρτησης/μεθόδου firstprivate ώστε να ορίσουμε τις μεταβλητές που δεν επιθυμούμε να διαμοιράζονται μεταξύ των νημάτων και έναρξη tasks ώστε να πραγματοποιηθεί υπολογισμός της συνάρτησης

```
for (trial = 0; trial < ntrials; trial++) {
    #pragma omp task firstprivate(fx, jj, i) // // εναρξη tasks ωστε
```

Critical section

```
#pragma omp critical /* mutex area
                        στην εκτε
{
        if (fx < best_fx) {
```

Εύρεση συνολικού αριθμού υπολογισμών συνάρτησης

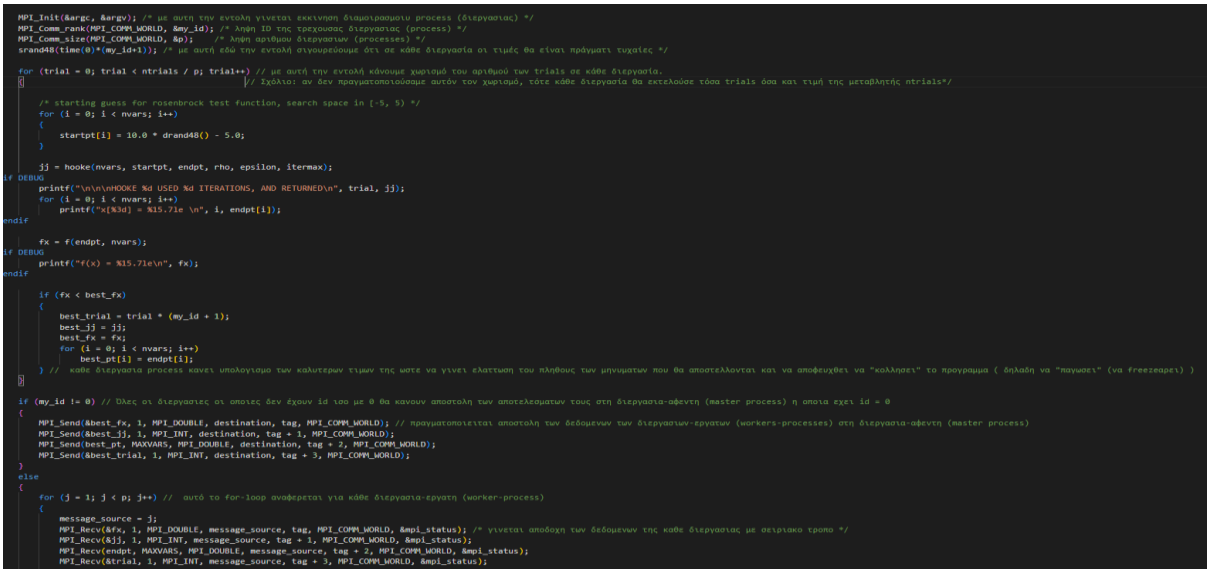
```
printf("Total number of function evaluations = %ld\n", funevals * number_of_threads); //  πραγματοπ
```

MPI

Αρχικά χρησιμοποιούμε την βιβλιοθήκη mpi.h.

Η βελτιστοποίηση έγινε για τον κύριο βρόχο for ο οποίος προκαλεί και την μεγαλύτερη καθυστέρηση. Ορίσαμε κατάλληλες μεταβλητές για την εποπτεία του rank και του size της διεργασίας, για την πηγή και το tag των μηνυμάτων, καθώς και για τον προορισμό και την κατάσταση των μηνυμάτων. Έπειτα, μέσω της δήλωσης 4 κατάλληλων εντολών εξασφαλίσαμε την άρτια διαχείριση και αλληλεπίδραση των διεργασιών. Τέλος, εντός του βρόχου for πραγματοποιούμε χωρισμό του αριθμού των trials σε κάθε διεργασία, και εντός αυτού του for loop υπάρχουν ακόμα 2 εμφωλευμένα for loops, δομές if-else και κατάλληλες εντολές MPI_Send και MPI_Recv για τον υπολογισμό από κάθε διεργασία των καλύτερων τιμών, την αποστολή των δεδομένων των workers-processes στις master-processes και την αποδοχή των δεδομένων της κάθε διεργασίας με σειριακό τρόπο.

Γενικά η παραλληλοποίηση



```
for (j = 1; j < p; j++) //  αυτό το for-loop αναφερεται για κάθε διεργασια-εργατη (worker-process)
{
    message_source = j;
    MPI_Recv(&fx, 1, MPI_DOUBLE, message_source, tag, MPI_COMM_WORLD, &mpi_status); /* γινεται αποδοχη των δεδομενων της καθε διεργασιας με σειριακο τροπο */
    MPI_Recv(&jj, 1, MPI_INT, message_source, tag + 1, MPI_COMM_WORLD, &mpi_status);
    MPI_Recv(endpt, MAXVARS, MPI_DOUBLE, message_source, tag + 2, MPI_COMM_WORLD, &mpi_status);
    MPI_Recv(&trial, 1, MPI_INT, message_source, tag + 3, MPI_COMM_WORLD, &mpi_status);
    if (fx < best_fx) //  κατ'αρχην κανουμε συγκριση των δεδομενων της διεργασιας-αφεντη (master process) με τη διεργασια που εχει id=1 και στη συνεχεια σειριακα με τις υπολοιπες διεργασιες
    {
        best_trial = trial;
        best_jj = jj;
        best_fx = fx;
        for (i = 0; i < nvars; i++)
            best_pt[i] = endpt[i];
    }
}
```

Μεταβλητές και εντολές για την εκτέλεση του MPI

```
int my_id, p; // δύο μεταβλητές MPI ώστε να έχουμε εποπτεία του rank και του size του process (διεργασίας)
int message_source;  // αυτή τη μεταβλητή την δηλώσαμε ώστε να ορίζουμε την πηγή των μηνυμάτων
int tag = 1234; // μεταβλητή ώστε να ορίσουμε το tag των μηνυμάτων
int destination = 0; // Μεταβλητή για να ορίζουμε τον προορισμό των μηνυμάτων
MPI_Status mpi_status; // Μεταβλητή για να γνωρίζουμε την κατάσταση των μηνυμάτων
```

```
MPI_Init(&argc, &argv); /* με αυτη την εντολη γινεται εκκινηση διαμοιρασμου process (διεργασιας) */
MPI_Comm_rank(MPI_COMM_WORLD, &my_id); /* ληψη ID της τρεχουσας διεργασιας (process) */
MPI_Comm_size(MPI_COMM_WORLD, &p);    /* ληψη αριθμου διεργασιων (processes) */
srand48(time(0)*(my_id+1)); /* με αυτή εδώ την εντολή σιγουρεύουμε ότι σε κάθε διεργασία οι τιμές θα είναι πράγματι τυχαίες */
```

```
if (my_id != 0) // Όλες οι διεργασιες οι οποιες δεν έχουν id ισο με 0 θα κανουν αποστολη των αποτελεσματων τους στη διεργασια-αφεντη (master process) η οποια εχει id = 0
{
    MPI_Send(&best_fx, 1, MPI_DOUBLE, destination, tag, MPI_COMM_WORLD); // πραγματοποιειται αποστολη των δεδομενων των διεργασιων-εργατων (workers-processes) στη διεργασια-αφεντη (master process)
    MPI_Send(&best_jj, 1, MPI_INT, destination, tag + 1, MPI_COMM_WORLD);
    MPI_Send(best_pt, MAXVARS, MPI_DOUBLE, destination, tag + 2, MPI_COMM_WORLD);
    MPI_Send(&best_trial, 1, MPI_INT, destination, tag + 3, MPI_COMM_WORLD);
}
```

```
else
{
    for (j = 1; j < p; j++) //  αυτό το for-loop αναφερεται για κάθε διεργασια-εργατη (worker-process)
    {
        message_source = j;
        MPI_Recv(&fx, 1, MPI_DOUBLE, message_source, tag, MPI_COMM_WORLD, &mpi_status); /* γινεται αποδοχη των δεδομενων της καθε διεργασιας με σειριακο τροπο */
        MPI_Recv(&jj, 1, MPI_INT, message_source, tag + 1, MPI_COMM_WORLD, &mpi_status);
        MPI_Recv(endpt, MAXVARS, MPI_DOUBLE, message_source, tag + 2, MPI_COMM_WORLD, &mpi_status);
        MPI_Recv(&trial, 1, MPI_INT, message_source, tag + 3, MPI_COMM_WORLD, &mpi_status);
        if (fx < best_fx) //  κατ'αρχην κανουμε συγκριση των δεδομενων της διεργασιας-αφεντη (master process) με τη διεργασια που εχει id=1 και στη συνεχεια σειριακα με τις υπολοιπες διεργασιες
        {
            best_trial = trial;
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
                best_pt[i] = endpt[i];
        }
    }
```

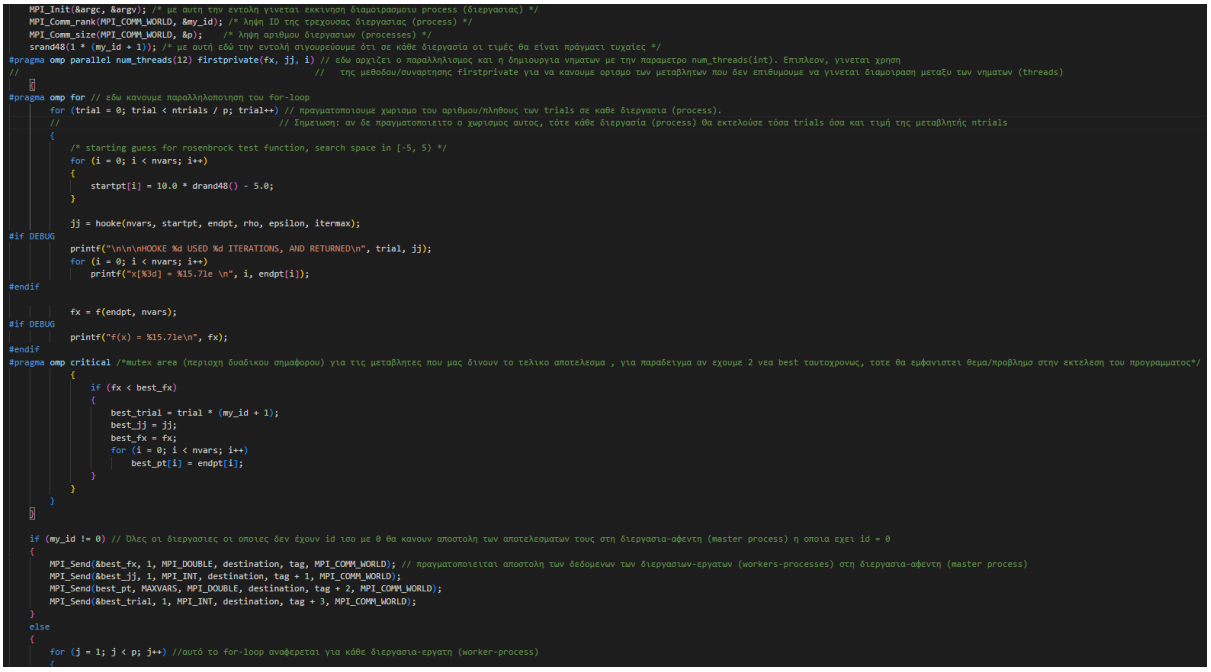
Εύρεση συνολικού αριθμού υπολογισμών συνάρτησης

```
printf("Total number of function evaluations = %ld\n", funevals * p); //  οπως και στο OpenMP , ετσι και σε αυτη εδω τη περιπτωση κανουμε πολλαπλασιασμο του αριθμου των πράξεων
                                                                      // με τον αριθμό των διεργασιων (processes) προκειμενου να υπολογισουμε τον πραγματικό αριθμό των πράξεων που γινονται
```

OpenMP + MPI --- hybrid implementation

Συνδυάσαμε τις 2 παραπάνω υλοποιήσεις, ουσιαστικά σπάμε το μεγάλο loop σε p κομμάτια ενώ χρησιμοποιούμε και το OpenMP ώστε να παραλληλοποιήσουμε τον βρόχο όπως εξηγήσαμε παραπάνω προσέχοντας βέβαια και το critical section στο οποίο χρησιμοποιούμε mutex προκειμένου να εξασφαλίσουμε ότι δε θα γράψουν 2 νήματα ταυτόχρονα στην ίδια μεταβλητή. Έξω από το loop χρησιμοποιώντας τη συνάρτηση MPI_Send στέλνουμε τα αποτελέσματα στον "master" πυρήνα ο οποίος θα λαμβάνει μέσω της εντολής MPI_Recv. Τέλος, κλείνουμε το MPI με την εντολή MPI_Finalize.

Γενικά η παραλληλοποίηση

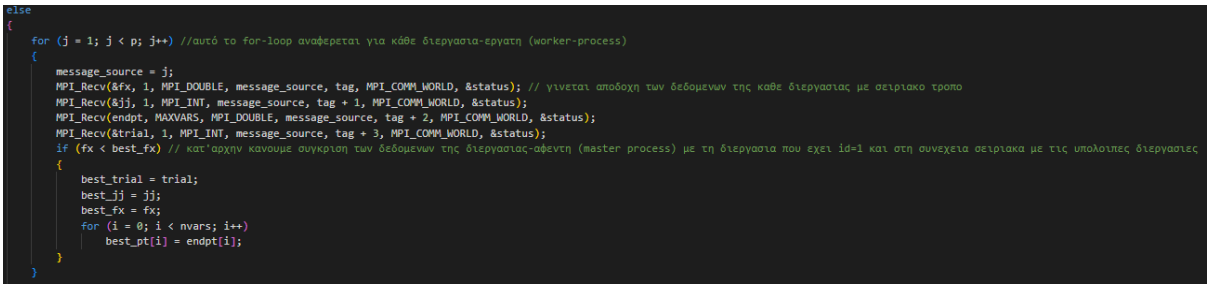


```
    MPI_Init(&argc, &argv); /* με αυτη την εντολη γινεται εκκινηση διαμοιρασμου process (διεργασιας) */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id); /* ληψη ID της τρεχουσας διεργασιας (process) */
    MPI_Comm_size(MPI_COMM_WORLD, &p);     /* ληψη αριθμου διεργασιων (processes) */
    srand48(1 * (my_id + 1)); /* με αυτή εδώ την εντολή σιγουρεύουμε ότι σε κάθε διεργασία οι τιμές θα είναι πράγματι τυχαίες */
#pragma omp parallel num_threads(12) firstprivate(fx, jj, i) // εδω αρχιζει ο παραλληλισμος και η δημιουργια νηματων με την παραμετρο num_threads(int). Επιπλεον, γινεται χρηση
//                                                           //   της μεθόδου/συναρτησης firstprivate για να κανουμε ορισμο των μεταβλητων που δεν επιθυμουμε να γινεται διαμοιραση μεταξυ των νηματων (threads)
    {
#pragma omp for // εδω κανουμε παραλληλοποιηση του for-loop
        for (trial = 0; trial < ntrials / p; trial++) // πραγματοποιουμε χωρισμο του αριθμου/πληθους των trials σε καθε διεργασια (process).
        //                                            // Σημειωση: αν δε πραγματοποιειτο ο χωρισμος αυτος, τότε κάθε διεργασία (process) θα εκτελούσε τόσα trials όσα και τιμή της μεταβλητής ntrials
        {
            /* 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
#pragma omp critical /*mutex area (περιοχη δυαδικου σημαφορου) για τις μεταβλητες που μας δινουν το τελικο αποτελεσμα , για παραδειγμα αν εχουμε 2 νεα best ταυτοχρονως, τοτε θα εμφανιστει θεμα/προβλημα στην εκτελεση του προγραμματος*/
            {
                if (fx < best_fx)
                {
                    best_trial = trial * (my_id + 1);
                    best_jj = jj;
                    best_fx = fx;
                    for (i = 0; i < nvars; i++)
                        best_pt[i] = endpt[i];
                }
            }
        }
    }

    if (my_id != 0) // Ολες οι διεργασιες οι οποιες δεν έχουν id ισο με 0 θα κανουν αποστολη των αποτελεσματων τους στη διεργασια-αφεντη (master process) η οποια εχει id = 0
    {
        MPI_Send(&best_fx, 1, MPI_DOUBLE, destination, tag, MPI_COMM_WORLD); // πραγματοποιειται αποστολη των δεδομενων των διεργασιων-εργατων (workers-processes) στη διεργασια-αφεντη (master process)
        MPI_Send(&best_jj, 1, MPI_INT, destination, tag + 1, MPI_COMM_WORLD);
        MPI_Send(best_pt, MAXVARS, MPI_DOUBLE, destination, tag + 2, MPI_COMM_WORLD);
        MPI_Send(&best_trial, 1, MPI_INT, destination, tag + 3, MPI_COMM_WORLD);
    }
    else
    {
        for (j = 1; j < p; j++) //αυτό το for-loop αναφέρεται για κάθε διεργασία-εργάτη (worker-process)
        {
```



```
else
{
    for (j = 1; j < p; j++) //αυτό το for-loop αναφέρεται για κάθε διεργασία-εργάτη (worker-process)
    {
        message_source = j;
        MPI_Recv(&fx, 1, MPI_DOUBLE, message_source, tag, MPI_COMM_WORLD, &status); // γίνεται αποδοχη των δεδομενων της καθε διεργασιας με σειριακο τροπο
        MPI_Recv(&jj, 1, MPI_INT, message_source, tag + 1, MPI_COMM_WORLD, &status);
        MPI_Recv(endpt, MAXVARS, MPI_DOUBLE, message_source, tag + 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&trial, 1, MPI_INT, message_source, tag + 3, MPI_COMM_WORLD, &status);
        if (fx < best_fx) // κατ'αρχην κανουμε συγκριση των δεδομενων της διεργασιας-αφεντη (master process) με τη διεργασια που εχει id=1 και στη συνεχεια σειριακα με τις υπολοιπες διεργασιες
        {
            best_trial = trial;
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
                best_pt[i] = endpt[i];
        }
    }
```

Μεταβλητές και εντολές για την εκτέλεση του OpenMP + MPI

```
int my_id, p; // δύο μεταβλητές MPI ώστε να έχουμε εποπτεία του rank και του size του process(διεργασίας)
int message_source; // αυτή τη μεταβλητή την δηλώσαμε ώστε να ορίζουμε την πηγή των μηνυμάτων
int tag = 1234; // μεταβλητή ώστε να ορίσουμε το tag των μηνυμάτων
int destination = 0; // Μεταβληρή για να ορίζουμε τον προορισμό των μηνυμάτων
MPI_Status status; // Μεταβληρή για να γνωρίζουμε την κατάσταση των μηνυμάτων
```

```
int provided;
MPI_Init(&argc, &argv); /* με αυτη την εντολη γινεται εκκινηση διαμοιρασμου process (διεργασιας) */
MPI_Comm_rank(MPI_COMM_WORLD, &my_id); /* ληψη ID της τρεχουσας διεργασιας (process) */
MPI_Comm_size(MPI_COMM_WORLD, &p);     /* ληψη αριθμου διεργασιων (processes) */
```

Έναρξη παραλληλισμού OpemMP , δημιουργία νημάτων με την παράμετρο num_threads(int) και χρήση της μεθόδου/συνάρτησης firstprivate για να κάνουμε ορισμό των μεταβλητών που δεν επιθυμούμε να γίνεται διαμοίραση μεταξύ των νημάτων (threads)

```
#pragma omp parallel num_threads(12) firstprivate(fx, jj, i)
```

Παραλληλοποίηση βρόχου

```
#pragma omp for // εδω κανουμε παραλληλοποιηση του for-
        for (trial = 0; trial < ntrials / p; trial++) /
        //                                            //
        {
```

Critical Section

```
#pragma omp critical /*mutex area (περιοχη δυαδικου σ
        {
            if (fx < best_fx)
            {
                best_trial = trial * (my_id + 1);
                best_jj = jj;
                best_fx = fx;
                for (i = 0; i < nvars; i++)
                    best_pt[i] = endpt[i];
            }
        }
```

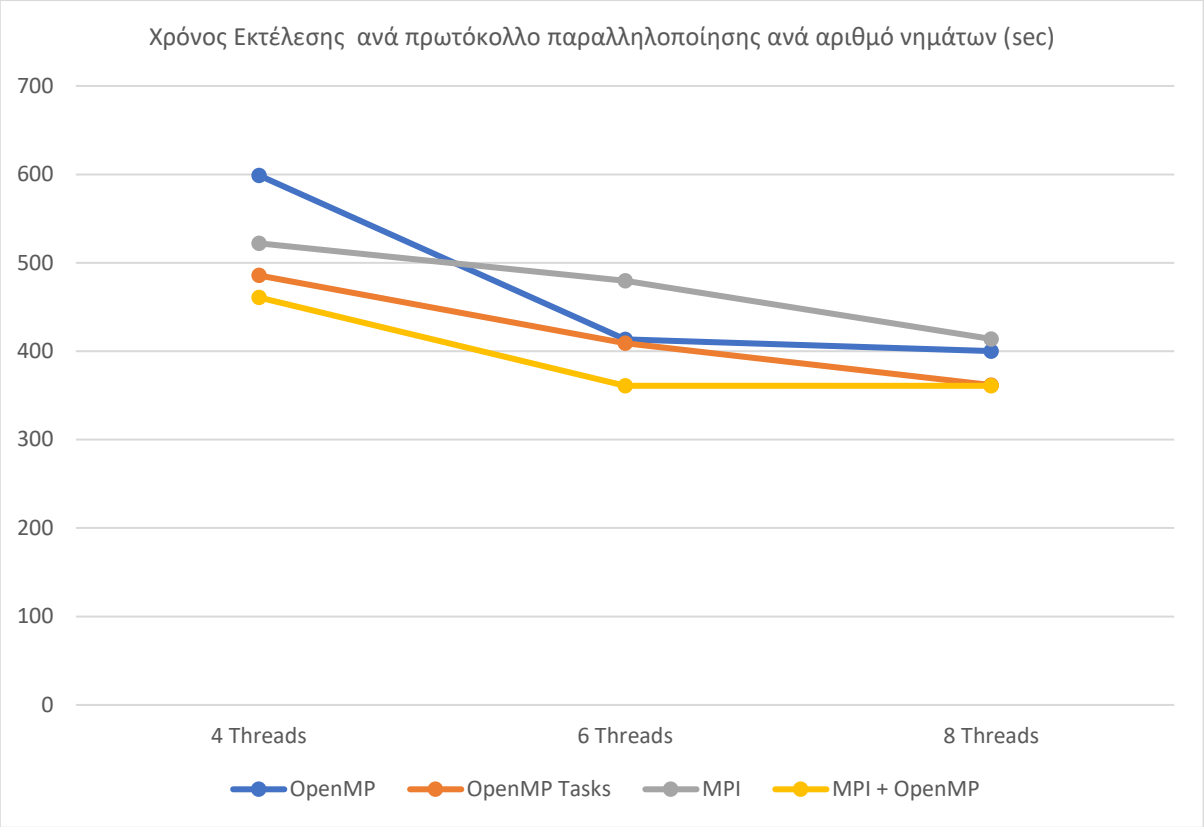
Εύρεση συνολικού αριθμού υπολογισμών συνάρτησης

```
printf("Total number of function evaluations = %ld\n", funevals * p); // κανουμε πολ
```

Αποτελέσματα

Εκτελώντας τις παραπάνω υλοποιήσεις σε 3 διαφορετικούς αριθμούς τμημάτων λαμβάνουμε τα εξής αποτελέσματα:

	4 Threads	6 Threads	8 Threads
OpenMP	598,866	413,422	400,153
OpenMP Tasks	485,654	409,211	361,715
MPI	522,024	479,714	413,86
MPI + OpenMP	460,756	360,937	360,867



*** Στον κατακόρυφο άξονα, φαίνεται ο χρόνος εκτέλεσης σε δευτερόλεπτα ενώ στον οριζόντιο το νήμα της υλοποίησης.*

Συμπεράσματα

Αρχικά, παρατηρούμε ότι για 4 νήματα το MPI + OpenMP είναι ο πιο γρήγορος τρόπος παραλληλοποίησης το οποίο είναι λογικό αφού χρησιμοποιεί 2 είδη παραλληλοποιήσεων. Στη συνέχεια το OpenMP Tasks που είναι πιο γρήγορο από το MPI και το OpenMP καθώς παραλληλοποιεί χρησιμοποιώντας tasks που είναι μια αρκετά πιο ελαφριά μορφή διεργασίας. Στη συνέχεια, εκτελώντας στα 6 νήματα παρατηρούμε ότι αυτή η κατάταξη συνεχίζει ως ήταν ωστόσο το MPI αυτή τη φορά είναι πιο αργό από το OpenMP και είναι ουσιαστικά ο πιο αργός τρόπος παραλληλοποίησης. Τέλος, στα 8 νήματα παρατηρούμε ότι τα 4 είδη παραλληλοποίησης έχουν πολύ μικρές διαφορές με τη σειρά να είναι MPI + OpenMP το γρηγορότερο, στη συνέχεια ακολουθούν OpenMP Tasks (με διαφορά μόλις 1 δευτερόλεπτο), OpenMP και τελευταίο το MPI.

**Οι παράξενοι χρόνοι εκτέλεσης του MPI ίσως οφείλονται στο ότι χρησιμοποιήσαμε διαφορετικό υπολογιστή για την εκτέλεσή του.*

Γενικότερα οι αναμενόμενες παρατηρήσεις θεωρούμε πως είναι MPI + OpenMP να είναι πάντα το γρηγορότερο, MPI και OpenMP Tasks να ακολουθούν με αυτή τη σειρά και τελευταίο το OpenMP.

Speed-Up 4	Speed-Up 6	Speed-Up 8
2,306526	3,341138	3,45193
2,844206	3,37552	3,818752
2,646047	2,879424	3,337602
2,997899	3,826984	3,827726

Τα αντίστοιχα speedup είναι τα παραπάνω και υπολογίζονται από τον τύπο t_1/t_p , όπου t_1 ο χρόνος εκτέλεσης της διαδικασίας σειριακά (1381,300 sec) και t_p ο αντίστοιχος χρόνος εκτέλεσης για κάθε αριθμό νημάτων.