**Παράλληλη Επεξεργασία**

**2023 - 2024**

Χαράλαμπος Γιαννέλης ΑΜ : 1093341

Γιώργος Σωτηρόπουλος ΑΜ : 1072541

Παναγιώτης Πέττας ΑΜ : 1093480

Αναστασία Καπελλάκη ΑΜ : 1072492

**Πίνακας περιεχομένων**

[**Εισαγωγή 3**](#_heading=h.gjdgxs)

[*Variables Explanation 3*](#_heading=h.30j0zll)

[*Rosenbrock Function 3*](#_heading=h.1fob9te)

[*Problem Parameters 4*](#_heading=h.3znysh7)

[*MDS Parameters 4*](#_heading=h.2et92p0)

[*Points and Results 4*](#_heading=h.tyjcwt)

[*Best Point Information 5*](#_heading=h.3dy6vkm)

[*Local Variables 5*](#_heading=h.1t3h5sf)

[*Multistart Trials 5*](#_heading=h.4d34og8)

[**Ερώτημα 1 : OpenMP 6**](#_heading=h.2s8eyo1)

[**Ερώτημα 2 : OpenMP Tasks 11**](#_heading=h.17dp8vu)

[**Ερώτημα 3 : MPI 14**](#_heading=h.3rdcrjn)

[*MPI enviroment 14*](#_heading=h.26in1rg)

[*Work Distribution 14*](#_heading=h.lnxbz9)

[*MPI\_Reduce 18*](#_heading=h.35nkun2)

[*Print Results 19*](#_heading=h.1ksv4uv)

[**Αποτελέσματα 19**](#_heading=h.44sinio)

[*Αρχιτεκτονική CPU 19*](#_heading=h.2jxsxqh)

[*Μετρήσεις 20*](#_heading=h.z337ya)

[OpenMP 21](#_heading=h.3j2qqm3)

[OpenMP Tasks 21](#_heading=h.1y810tw)

[MPI 22](#_heading=h.4i7ojhp)

[*Σύγκριση Μεθόδων 22*](#_heading=h.2xcytpi)

[**Συμπεράσματα 23**](#_heading=h.1ci93xb)

# Εισαγωγή

Παρακάτω θα αναλύσουμε την χρήση της κάθε μεταβλητής κρίσιμο κομμάτι για να μπορέσουμε να αξιολογήσουμε ποιες μεταβλητές θα πρέπει να προσέξουμε κατά την διαδικασία της παραλληλοποίησης.

## Variables Explanation

* **MAXVARS**: Maximum αριθμός μεταβλητών (dimensions) τις οποίες μπορεί να διαχειριστεί ο κώδικας.
* **EPSMIN**: Minimum μέγεθος convergence δηλαδή βήματος σύγκλισης, η οποία εκφράζει την ακρίβεια της τελικής λύσης.
* **funevals**: global variable η οποία παρακολουθεί τον συνολικό αριθμό των Evaluation Function που πραγματοποιήθηκαν κατά τη διάρκεια επίλυσης του προβλήματος.

## Rosenbrock Function

**f(double \*x, int n)**: Υπολογίζει την τιμή της συνάρτησης Rosenbrock για ένα

n-διάστατο σημείο x.

* **x**: To input το οποίο αντιπροσωπεύει το σημείο στον Simplex.
* **n**: O αριθμός των διαστάσεων (dimensions).
* **fv**: Η τιμή η οποία υπολογίζεται από την Rosenbrock function.
* **funevals++**: Αυξάνει τον global μετρητή ο οποίος μετρά των συνολικό αριθμό υπολογισμού της evaluation function.
* **usleep(1)**: Προσθέτει μια καθυστέρηση για κάθε αξιολόγηση συνάρτησης.

## Problem Parameters

* **nvars**: Αριθμός διαστάσεων (dimensions) του προβλήματος.
* **ntrials**: Αριθμός τυχαίων σημείων εκκίνησης για τη Multistart μέθοδο.
* **lower - upper**: Arrays τα οποία καθορίζουν τα κάτω και άνω όρια του simplex για κάθε μεταβλητή.

## MDS Parameters

* **eps**: κριτήριο σύγκλισης. Ο αλγόριθμος σταματά όταν το μέγεθος βήματος είναι μικρότερο από αυτή την τιμή.
* **maxfevals**: maximum αριθμός για function evaluations..
* **maxiter**: maximum αριθμός για iterations.
* **mu**: συντελεστής επέκτασης (Για την πράξη expansion του MDS).

* **theta**: συντελεστής Contraction (Για την πράξη constraction του MDS).
* **delta**: δηλώνει το βήμα για τα vertices στον πολύτοπο simplex.

## Points and Results

* **startpt**: Array για την αποθήκευση του starting point για τον MDS.
* **endpt**: Array για την αποθήκευση του final point για τον MDS.
* **fx**: Τιμή της συνάρτησης Rosenbrock στο τέλος της εκτέλεσης.
* **nt - nf**: αριθμός iterations και αριθμός function evaluations οι οποίοι χρησιμοποιήθηκαν κατά την εκτέλεση του MDS.

## Best Point Information

* **best\_pt**: για την αποθήκευση των συντεταγμένων του καλύτερου σημείου που βρέθηκε κατά τη διαδικασία Multistart.
* **best\_fx**: Τιμή συνάρτησης στο καλύτερο σημείο.
* **best\_trial**: Index του βήματος που βρέθηκε το καλύτερο σημείο.
* **best\_nt - best\_nf**: Αριθμός επαναλήψεων και υπολογισμού Evaluation Function για την καλύτερη δοκιμή.

## Local Variables

* **trial , i**: Μετρητές επαναλήψεων.
* **t0 , t1**: Μετρητές χρόνου.

## Multistart Trials

O παρακάτω κώδικας χρησιμοποιείται για την εκτέλεση πολλαπλών δοκιμών του αλγορίθμου MDS από διαφορετικά σημεία εκκίνησης και την επιλογή της καλύτερης λύσης που βρέθηκε μεταξύ αυτών των δοκιμών. Το for loop επαναλαμβάνει τον αριθμό δοκιμών που καθορίζεται από το ntrials. Κάθε επανάληψη αντιπροσωπεύει μια νέα δοκιμή με διαφορετικό σημείο εκκίνησης για τον αλγόριθμο MDS.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

**srand48**:σκοπός του srand48 είναι να τροφοδοτεί τη γεννήτρια τυχαίων αριθμών που χρησιμοποιείται από το drand48, με τον τρέχοντα αριθμό του trial. H χρήση του trial εξασφαλίζει ότι σε κάθε επανάληψη θα έχουμε διαφορετικό σημείο εκκίνησης. Είναι σημαντικό να τονίσουμε ότι η χρήση του ίδιου seed για την γεννήτρια τυχαίων αριθμών παράγει την ίδια ακολουθία.

Στη συνέχεια παράγεται τυχαίο σημείο εκκίνησης εντός του search space [-2,2] για την τρέχουσα δοκιμή. Η εντολή **drand48()** παράγει έναν τυχαίο αριθμό κινητής υποδιαστολής ομοιόμορφα κατανεμημένο στην περιοχή [0.0, 1.0].

Ο τύπος **lower[i] + (upper[i] - lower[i]) \* drand48()** προσαρμόζει τον τυχαίο αριθμό στο επιθυμητό εύρος [lower[i], upper[i]). Εφόσον στην συγκεκριμένη περίπτωση έχουμε nvars ίσο του 4 για αυτό παράγουμε ένα σημείο με 4 συντεταγμένες αφού ο simplex έχει 4 διαστάσεις.

Στη συνέχεια μέσω της συνάρτησης **initialize\_simplex()** δημιουργούνται και τα υπόλοιπα σημεία.

**For nvars = 4**

**Initial Point**: startpt = [a, b, c, d].

**Simplex Vertices**:

* Vertex 0: [a, b, c, d]
* Vertex 1: [a + delta, b, c, d]
* Vertex 2: [a, b + delta, c, d]
* Vertex 3: [a, b, c + delta, d]
* Vertex 4: [a, b, c, d + delta]

# Ερώτημα 1 : OpenMP

**funevals**

Παρατηρούμε ότι στην συνάρτηση f υπάρχει ο μετρητής funevals στον οποίο προσθέτουμε την εντολή # pragma omp atomic. Η funevals όπως αναφέραμε είναι μια global μεταβλητή η οποία μετρά τον συνολικό αριθμό υπολογισμού της Evaluation Function. Έτσι κάθε thread θα αυξήσει την συγκεκριμένη μεταβλητή κάθε φορά που καλείται η συνάρτηση f.

Αν δεν υπάρχει κάποιος συγχρονισμός τότε θα δημιουργηθεί race condition καθώς πολλά threads μπορεί να επιχειρήσουν να αυξήσουν τα funevals ταυτόχρονα. Εδώ λοιπόν μπορεί να χάσουμε αυξήσεις. Αν για παράδειγμα η μεταβλητή έχει την τιμή 10 και 2 threads επιχειρήσουν ταυτόχρονα να την αυξήσουν, είναι πιθανό να χαθεί η μία αύξηση και γενικότερα στον συνολικό μετρητή εν τέλη να δούμε αισθητά μειωμένη τιμή. Επίσης αυτό συνεπάγεται ότι τη δεδομένη στιγμή που κάποιο thread επιχειρήσει να αυξήσει την μεταβλητή, η τιμή της να είναι ασυνεπής.Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

**omp\_get\_time()**

Παρατηρούμε ότι αντικαταστήσαμε την συνάρτηση μέτρησης του χρόνου η οποία χρησιμοποιεί την gettimeofday() με την εντολή omp\_get\_time(). Τα πλεονεκτήματα είναι τα εξής :

* Η omp\_get\_wtime() προσφέρει μεγαλύτερη ακρίβεια και υψηλότερη ανάλυση σε σχέση με την gettimeofday() γεγονός πολύ σημαντικό στην παράλληλη επεξεργασία.
* Η οmp\_get\_time έχει σχεδιαστεί ειδικά για την μέτρηση της απόδοσης σε παράλληλα προγράμματα και εξασφαλίζει συνεπή μέτρηση με τη χρήστη διαφορετικών treads. Αντίθετα η gettimeofday() μπορεί να εμφανίσει ασυνέπειες ρολογιού. Έτσι θα λέγαμε ότι η συγκεκριμένα συνάρτηση είναι thread safe και συνιστάται στην παράλληλη βελτιστοποίηση.

Εικόνα που περιέχει κείμενο, γραμματοσειρά, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματα

**#pragma omp parallel reduction(+:funevals)**

Στη συνέχεια με την παραπάνω εντολή δημιουργούμαι την παράλληλη περιοχή όπου τα threads θα εκτελέσουν τον κώδικα ο οποίος υπάρχει μέσα σε αυτό το block.

Επιπλέον με την εντολή του reduction υποδεικνύουμε στο OpenMP να δημιουργήσει είναι private αντίγραφο των funevals σε κάθε νήμα. Στο τέλος της εκτέλεσης αυτά τα αντίγραφα συνδυάζονται και μάλιστα προστίθενται ( + : ) για να αναπαριστούμε με ακρίβεια το συνολικό αριθμό της μεταβλητής funevals.Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό πολυμέσων, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

**Private Variables**

Το private clause υποδηλώνει ότι οι συγκεκριμένες μεταβλητές θα πρέπει να είναι ιδιωτικές ανά νήμα και το καθένα να έχει το δικό του αντίγραφο αυτών των variables. Με αυτό τον τρόπο θα αποτρέψουμε race conditions των μεταβλητών αυτών.

* **trial:** κάθε νήμα πρέπει να έχει το δικό του trial για να μην έχουμε race conditions.
* **i:** δείκτης loop for που χρησιμοποιείται εντός παράλληλης περιοχής.
* **startpt:** κάθε thread θα πρέπει να έχει θα δικά του σημεία έναρξης.
* **endpt:** κάθε thread θα πρέπει να έχει θα δικά του σημεία λήξης.
* **fx:** τιμή της συνάρτησης Rosenbrock στο τελικό σημείο του MDS.
* **nt,nf:** counters επαναλήψεων και evaluation functions που χρησιμοποιούνται από τον MDS.

Αν αυτές οι μεταβλητές δεν ήταν private τότε τα threads θα τις μοιράζονταν, θα είχαμε λοιπόν data races καθώς τα threads θα αντικαθιστούν τιμές των υπολοίπων με αποτέλεσμα λανθασμένα αποτελέσματα.

**randBuffer**

O πινακας randBuffer χρησιμοποιείται για την αποθήκευση του seed για την συνάρτηση erand48(). O αρχικοποιείται με 3 ακεραίους χωρίς πρόσημο. Πιο συγκεκριμένα, τα randBuffer[0] και randBuffer[1] αρχικοποιούνται με την τιμή 0 για λόγους απλότητας ενώ ο randBuffer[2] τίθεται σε μία μοναδική τιμή για κάθε thread προσθέτοντας τις μεταβλητές tseed + ntrials + omp\_get\_thread\_num().

Το ntrials το οποίο είναι οι επαναλήψεις του προβλήματος είναι 64 και προσθέτει μεγαλύτερη μεταβλητότητα ενώ το οmp\_get\_thread\_num() εξασφαλίζει ότι το κάθε νήμα θα κάνει διαφορετικό seeding.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

Στον παραπάνω κώδικα παρατηρούμε ότι κάθε νήμα χρησιμοποιεί το δικό του randBuffer για τη δημιουργία τυχαίων αριθμών με το erand48() , διασφαλίζοντας ότι δεν υπάρχουν παρεμβολές μεταξύ των threads. Επιπλέον κάθε thread παράγει μοναδικό σημείο εκκίνησης για το πρόβλημα βελτιστοποίησης το οποίο πρέπει να λυθεί.

**Σημασία χρήσης κατάλληλου seeding**

Το seeding το οποίο θα ακολουθήσουμε είναι ζωτικής σημασίας καθώς εάν τα seeds δεν διαφοροποιούνται επαρκώς τότε θα λάβουμε μεγαλύτερο Fx δηλαδή δεν θα έχουμε αποτελεσματικότητα στη βελτιστοποίηση multistart. Επιπρόσθετα για παράδειγμα δεν χρησιμοποιήσαμε seeding της μορφής time(NULL) με σκοπό να εξασφαλίσουμε το reproducibility. Πιο συγκεκριμένα θέλαμε να αποφύγουμε την τυχαιότητα καθώς με την χρήστη της time(NULL) στο seeding παρατηρούσαμε μεγάλες μεταβολές της συνάρτησης και της εύρεσης τοπικού ελαχίστου από εκτέλεση σε εκτέλεση.

**Scheduling**

H αλήθεια είναι ότι μεταξύ χρήσης dynamic ή static scheduling δεν παρατηρήθηκαν διαφορές στον χρόνο εκτέλεσης. Αυτό σημαίνει ότι ο φόρτος εργασίας τον οποίο αναλαμβάνει το κάθε thread είναι ο ίδιος και σημαίνει ότι το πρόβλημα μπορεί να παραλληλοποιηθεί αποτελεσματικά χωρίς να απαιτείται ρύθμιση της στρατηγικής που θα ακολουθηθεί στον χρονοπρογραμματισμό.

Ωστόσο επιλέξαμε τον dynamic καθώς εξ ορισμού αν ένα thread ολοκληρώσει το chunk του τότε δυναμικά αναλαμβάνει ένα άλλο chunk μέχρι να ολοκληρωθούν οι επαναλήψεις. Έτσι διασφαλίζουμε καλύτερη εξισορρόπηση φορτίου και επίσης είναι πιο αποδοτικό αν κάποιες επαναλήψεις διαρκέσουν λιγότερο από άλλες δηλαδή αν ο MDS ο οποίος καλείται είναι non – uniform σαν συνάρτηση.

**Critical Section**

Το critical section χρησιμοποιείται για να διασφαλίσουμε ότι κάθε φορά ένα μόνο νήμα θα ενημερώνει τις μεταβλητές best\_fx, best\_trial, best\_nt, best\_nf και best\_pt. Αυτές οι μεταβλητές περιέχουν την καλύτερη λύση που έχει βρεθεί μέχρι στιγμής και το να επιτρέπεται σε πολλά νήματα να τις ενημερώνουν ταυτόχρονα θα μπορούσε να οδηγήσει σε λανθασμένα αποτελέσματα. Όταν πολλά treads προσπαθήσουν να ενημερώσουν τις συγκεκριμένες μεταβλητές θα δημιουργηθούν race conditions. Ωστόσο θα πρέπει να σημειώσουμε ότι τα critical sections εφόσον εκτελούνται από ένα thread τη φορά επιβαρύνουν τον συνολικό χρόνο εκτέλεσης.Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό πολυμέσων, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

# Ερώτημα 2 : OpenMP Tasks

**Ανάλυση**

Με τη χρήση OpenMP Tasks μας δίνεται επιπλέον ευελιξία σχετικά  με τη παραλληλοποίηση το προβλήματος. Κάθε task δημιουργεί overhead για τη δημιουργία, διαχείριση και ολοκλήρωση του, συνεπώς με τη χρήση πολλών task το overhead μπορεί να ξεπεράσει τα οφέλη της παράλληλης εκτέλεσης. Για την αποφυγή του παραπάνω προβλήματος χρησιμοποιήθηκε το εργαλείο ´gprof' για την χρονική ανάλυση το σειριακού κώδικα.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, μαύρο

Περιγραφή που δημιουργήθηκε αυτόματα

Το εργαλείο μας δείχνει πως η συνάρτηση Rosenbrock, δηλαδή η συνάρτηση f, καταναλώνει το μεγαλύτερο μέρος του χρόνου που απαιτείται για την εκτέλεση του προγράμματος. Σύμφωνα με τα παραπάνω μπορούμε να αποφανθούμε πως η παραλληλοποίηση θα γίνει στην συνάρτηση f.

**Υλοποίηση**

 Η μεθοδολογία του OpenMp Tasks είναι παρόμοια με το πρώτο ερώτημα σχετικά με την αρχικοποίηση και το seeding οπότε θα αναλυθούν οι διαφορές τους.



**#pragma omp parallel**

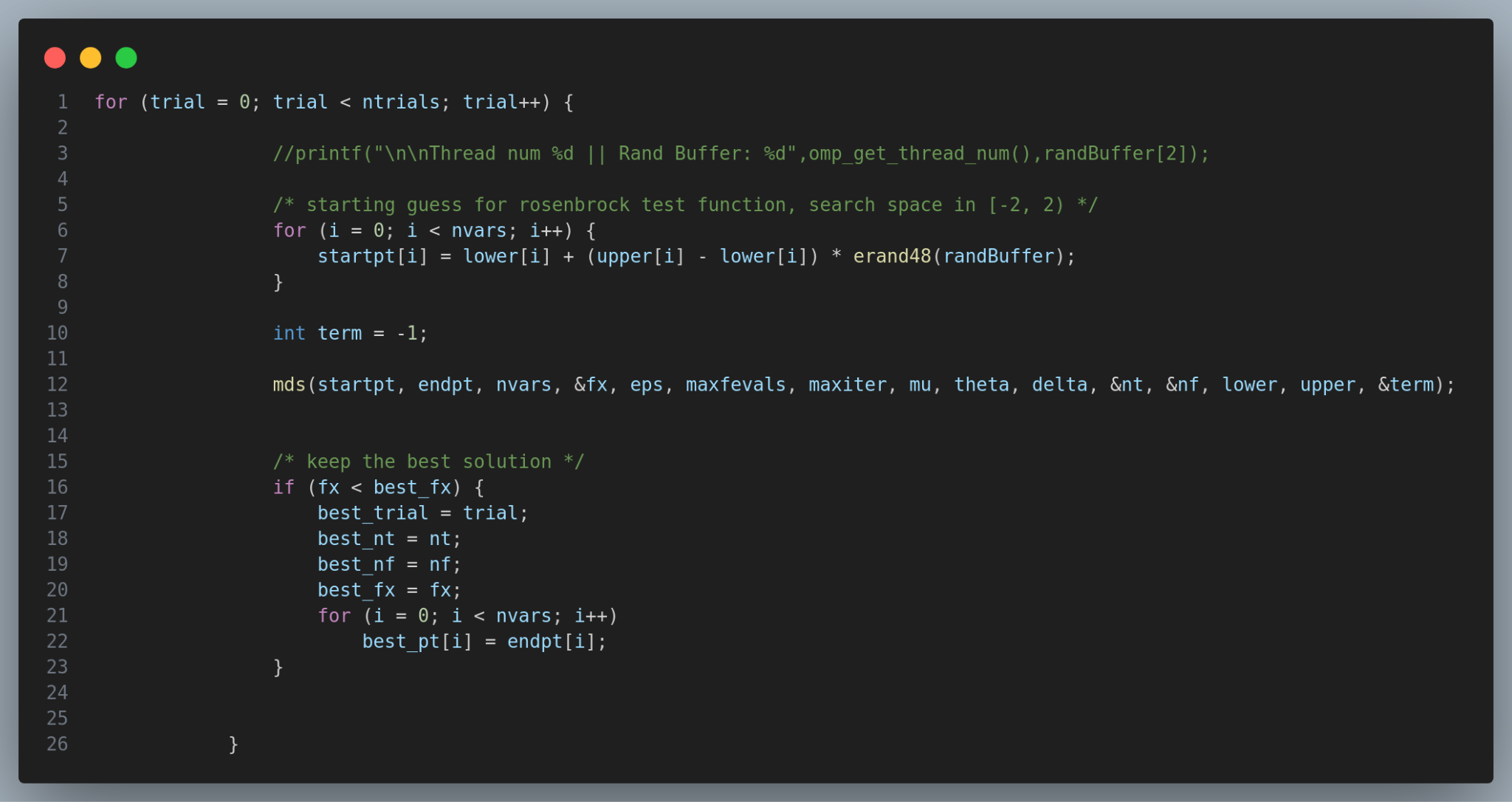
Δημιουργεί μια ομάδα threads για παράλληλη εκτέλεση (ορίζει την παράλληλη περιοχή).

**#pragma omp single**

Σύμφωνα με τη θεωρία για τη χρήση του OpenMP tasks πρέπει η mds να εκτελεστεί από ένα νήμα και αυτό μας το προσφέρει η “omp single”. Με αυτό το τρόπο τα υπόλοιπα νήματα μένουν ‘ελεύθερα’ έως ότου να καλεστούν από ένα task.

**Επαναληπτική δομή για τον mds**

Από τη στιγμή που η επαναληπτική δομή περικλείεται από τη εντολή που αναλύθηκε προηγούμενος ( “omp single” ) και εκτελείται από ένα νήμα δεν χρείαζεται κάποια τροποποίηση από τον σειριακό. Για παράδειγμα η ‘omp critical’ που χρησιμοποιήθηκε πριν για τον υπολογισμό των best μεταβλητών αφαιρέθηκε μιάς και δεν θα προσπελαστεί απο παραπάνω από ένα νήμα.



**#pragma omp task**

Τέλος μπορούμε να δημιουργούμε tasks στα σημεία του κώδικα που το χρειάζονται. Στη περίπτωση μας καταλλήξαμε πως παραλληλοποίηση χρειάζεται η Rosenbrock οπότε ορίσαμε tasks για κάθε κάλεσμα της στον mds.

Εικόνα που περιέχει κείμενο, λογισμικό, λογισμικό πολυμέσων, πολυμέσα

Περιγραφή που δημιουργήθηκε αυτόματα

Παραπάνω βλέπουμε μία εφαρμογή του  task. H ‘omp task’ δημιουργεί μια μονάδα εργασίας που μπορεί να εκτελεστεί από οποιοδήποτε thread στην ομάδα OpenMp. Η firstprivate(i) ορίζεται την ιδιωτικότητα του (i) για κάθε thread. Το untied clause καθορίζει ότι το task μπορεί να το αναλάβει και να το εκτελέσει οποιοδήποτε διαθέσιμο thread ανεξάρτητα απο το πιο thread το δημιούργησε. Ομοίως υλοποιήθηκαν και τα υπόλοιπα σημεία που καλείται η Rosenbrock.

**#pragma omp taskwait**

Η συγκεκριμένη εντολή μας εξασφαλίζει την ολοκλήρωση εκτέλεσης του  task. Από την στιγμή που η ‘fec’ θα ξαναχρησιμοποιήσει  πρέπει να εξασφαλιστεί πως η Rosenbrock έχει επιστρέψει πρώτου συνεχίσει η εκτέλεση. Βέβαια η ‘fec’ θα χρησιμοποιηθεί μετά την ολοκλήρωση της αρχικής  επαναληπτικής δομής for γι’ αυτό το taskwait μπήκε μετά. Έτσι δημιουργούνται task για κάθε i απο 1 έως n-1 και αφού χρησιμοποιούν διαφορετικά κομμάτια του πίνακα ‘ec’ δεν έχουμε race conditions. Ειδικότερα, η εμφωλευμένη for θα υπολογίσει το κομμάτι του πίνακα ‘ec’ το οποίο θα χρησιμοποιήσει η Rosenbrock στο κάθε i. Μόλις ολοκληρωθεί αυτός ο υπολογισμός ορίζεται task για να υπολογίσει το fec[i] χρησιμοποιώντας το κομμάτι του πίνακα ‘ec’ που μόλις υπολογίστηκε. Έπειτα, δεν χρειάζεται να περιμένει να ολοκληρωθεί το task οπότε προχωράει στο επόμενο ‘i’.

**Rosenbrock**

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, υπολογιστής

Περιγραφή που δημιουργήθηκε αυτόματα

Μέσα στην συνάρτηση καλείται η usleep ή οποία έχει σκοπό να προσθέσει ένα τεχνητό φόρτο εργασίας. Μπορούμε να την συσχετίσουμε με ένα φόρτο το οποίο δεν σχετίζεται με τα προηγούμενα. Πιο συγκεκριμένα, αν υποθέσουμε πως δεν επηρεάζει τον υπολογισμό του ´fv´ μπορούμε να της ορίσουμε ένα δικό της task στο οποίο θα δώσουμε και εδώ το clause untied. Επομένως, η συνάρτηση ‘f’ επιστρέψει το fv χωρίς να περιμένει την ολοκλήρωση του usleep.

# Ερώτημα 3 : MPI

## MPI enviroment

Στο συγκεκριμένο ερώτημα θα χρησιμοποιήσουμε MPI το οποίο αποτελεί ένα σύστημα μηνυμάτων σε αρχιτεκτονικές παράλληλων υπολογιστών. Συγκεκριμένα επιτρέπει σε πολλαπλές διεργασίες να επικοινωνούν μεταξύ τους αποστέλλοντας και λαμβάνοντας μηνύματα. Το MPI λειτουργεί με την αρχικοποίηση πολλαπλών διεργασιών (processes) σε ένα σύστημα για παράδειγμα όπως και στην προκειμένη περίπτωση με πολλούς πυρήνες δηλαδή σε multi-core συστήματα.

**MPI\_Init(&argc,&argv):** αρχικοποιεί το MPI περιβάλλον.

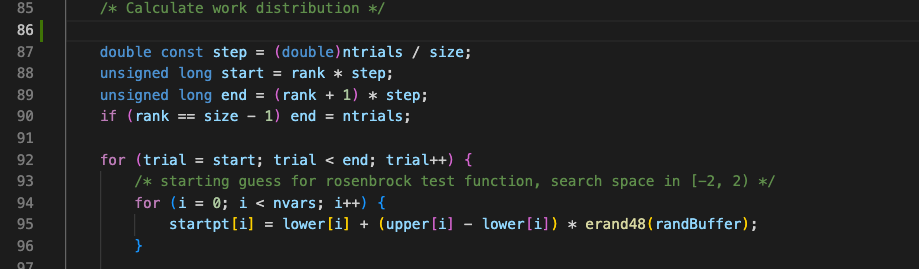
**MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank):** η συγκεκριμένη εντολή χρησιμοποιείται για να καθορίσει το rank για την κάθε διεργασία στον communicator. Στην προκειμένη περίπτωση communicator είναι ο MPI\_COMM\_WORLD.Αν για παράδειγμα τρέξουμε το executable με 8 processes τότε θα έχουμε ranks 0, 1, 2, 3, 4, 5, 6 και 7.

**MPI\_Comm\_size(MPI\_COMM\_WORLD,&size):** Καθορίζει το συνολικό αριθμό διεργασιών στον communicator MPI\_COMM\_WORLD. Στη συγκεκριμένη περίπτωση θα έχουμε size 8.

**MPI\_Finalize():** τερματίζει το MPI περιβάλλον.Εικόνα που περιέχει κείμενο, λογισμικό, λογισμικό πολυμέσων, λογισμικό γραφικών

Περιγραφή που δημιουργήθηκε αυτόματα

## Work Distribution



* Ο συνολικός αριθμός των επαναλήψεων (ntrials) διαιρείται με τον αριθμό των διεργασιών (size) για να καθοριστεί πόσα trials θα διαχειριστεί η κάθε διεργασία (process).
* Το **step** αντιπροσωπεύει τον αριθμό των trials όπου θα αναλάβει το κάθε process.
* Η μεταβλητή **start** είναι ο δείκτης για το αρχικό trial του κάθε process.
* Η μεταβλητή **end** είναι ο δείκτης για το τελευταίο trial του κάθε process.
* Η τελευταία διεργασία (rank size - 1) εξασφαλίζει ότι καλύπτει τυχόν υπολειπόμενες δοκιμές εάν το ntrials δεν είναι απόλυτα διαιρετό με το size.

**Example**

If ntrials = 64 and size = 8

step = 64 / 8 = 8

* **Process 0 (rank 0)**:
  + start = 0 \* 8 = 0
  + end = 1 \* 8 = 8
  + This process handles trials 0 to 7.
* **Process 1 (rank 1)**:
  + start = 1 \* 8 = 8
  + end = 2 \* 8 = 16
  + This process handles trials 8 to 15.
* **Process 7 (rank 7)**:
  + start = 7 \* 8 = 56
  + end = 8 \* 8 = 64
  + This process handles trials 56 to 63.

Αξίζει να αναφέρουμε ότι στον συγκεκριμένο κώδικα το process με rank 0 συμμετέχει κανονικά στους υπολογισμούς. Εφόσον ολοκληρώσει λοιπόν το rank 0 συγκεντρώνει τα αποτελέσματα από όλες τις διεργασίες και καθορίζει τον συνολικό best result.

Έτσι λοιπόν πετυχαίνουμε αξιοποίηση όλων των διαθέσιμων πόρων που οδηγεί σε μείωση του συνολικού χρόνου υπολογισμού. Τέλος αποφεύγουμε την αδράνεια του rank 0 από το να συλλέγει απλά τα αποτελέσματα χρησιμοποιώντας έτσι αποδοτικά την διαθέσιμη υπολογιστική ισχύς.

To παρακάτω block κώδικα εκτελείται από κάθε process και ενημερώνει το τοπικά καλύτερο αποτέλεσμά της εάν βρεθεί καλύτερο :Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

Για να διαχωρίσουμε καλύτερα τα local με τα global αποτελέσματα ορίζουμε επιπλέον και διαχωρίζουμε global από local μεταβλητές :

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Στην συνέχεια παρατηρούμε το παρακάτω block κώδικα το οποίο διαχωρίζει τις αρμοδιότητες του master rank = 0 και των υπολοίπων. Παρατηρούμε ότι αρχικοποιούνται οι global μεταβλητές με τις local μεταβλητές οι οποίες βρέθηκαν στο rank 0. Αυτό είναι σημαντικό βήμα καθώς θα πρέπει να συμπεριλάβουμε και τα αποτελέσματα του master process :Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Στη συνέχεια το rank 0 κάνει recieve τα local results από όλα τα υπόλοιπα processes και ενημέρώνει αντίστοιχα τις global μεταβλητές, υπολογίζοντας το καλύτερο αποτέλεσμα.

Συγκεκριμένα το rank 0 λαμβάνει πληροφορίες σχετικά με την συνάρτηση fx , το trial στο οποίο βρέθηκε η βέλτιστη fx καθώς και τον αριθμό των επαναλήψεων nt και nf. Τέλος γίνονται recieve και πληροφορίες σχετικά με τις συντεταγμένες του βέλτιστου σημείου pt.Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

**Rank 0**

**MPI\_Recv(&recv\_fx,1,MPI\_DOUBLE,p,100,MPI\_COMM\_WORLD,&status)**

* **recv\_fx:** Η μεταβλητή στην οποία θα αποθηκευτούν τα δεδομένα της fx.
* **1:** Αριθμός των στοιχείων που πρέπει να ληφθούν.
* **MPI\_DOUBLE:** Τύπος δεδομένων των στοιχείων.
* **p:** Rank του process το οποίο έστειλε το μήνυμα.
* **100:** tag μηνύματος για την αναγνώριση του μηνύματος.
* **MPI\_COMM\_WORLD:** Communicator.
* **&status:** μεταβλητή κατάστασης σχετικά με το ληφθέν μήνυμα.

Αντίστοιχα τα υπόλοιπα processes αποστέλλουν τις πληροφορίες στο rank 0. Επίσης έχουμε ορίσει αντίστοιχα το send tag ανάλογα με την μεταβλητή σε 100 ,101 ,102 κ.ο.κ. τα οποία αντιστοιχίζονται με τις Receive εντολές :Εικόνα που περιέχει στιγμιότυπο οθόνης, κείμενο, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

## MPI\_Reduce

**MPI\_Reduce(&local\_funevals, &global\_funevals, 1, MPI\_UNSIGNED\_LONG, MPI\_SUM, 0, MPI\_COMM\_WORLD)**

Συνδυάζει τα local\_funevals από όλες τις διεργασίες σε global\_funevals χρησιμοποιώντας την πράξη αθροίσματος.

* **local\_funevals:** buffer εισόδου (δεδομένα προς reduction).
* **global\_funevals:** buffer εξόδου (αποτέλεσμα της reduction).
* **1:** Αριθμός στοιχείων στον buffer εισόδου.
* **MPI\_UNSIGNED\_LONG:** Τύπος δεδομένων των στοιχείων.
* **MPI\_SUM:** Πράξη προς εκτέλεση (άθροισμα).
* **0:** rank της διεργασίας που θα λάβει το αποτέλεσμα.
* **MPI\_COMM\_WORLD:** Communicator.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματα

## Print Results

To rank 0 εκτυπώνει τα τελικά αποτελέσματα :Εικόνα που περιέχει κείμενο, λογισμικό πολυμέσων, λογισμικό, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματα

# Αποτελέσματα

Το μηχάνημα στο οποίο έγιναν οι μετρήσεις είναι ένα Macbook Air 2022 το οποίο έχει τον M2 επεξεργαστή με τα παρακάτω χαρακτηριστικά :

## Αρχιτεκτονική CPU

CPU: 8-Core με συνδυασμό τεσσάρων πυρήνων υψηλής απόδοσης (power-cores) και τεσσάρων efficiency cores. Αυτή η αρχιτεκτονική επιτρέπει την αποτελεσματική παράλληλη εκτέλεση εργασιών και την εξισορρόπηση φορτίου μεταξύ των πυρήνων.

**system\_profiler SPHardwareDataType**

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

**sysctl -a | grep machdep.cpu** 

Εικόνα που περιέχει στιγμιότυπο οθόνης, πράσινο, κύκλωμα

Περιγραφή που δημιουργήθηκε αυτόματα

## Μετρήσεις

Αρχικά υλοποιήθηκε script στο οποίο δοκιμάζουμε την εκτέλεση της παραλληλοποίησης σε 4 ,8 ,12 και 16 threads. Εκτελούμε το python αρχείο thread\_plots.py στο οποίο παρατηρούμε ότι πραγματοποιούμε τις παρακάτω εκτελέσεις :

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Στη συνέχεια στα αντίστοιχα json files γράφουμε τα αποτελέσματα των εκτελέσεων έτσι ώστε αν thread selections να δούμε που παρατηρείται το μεγαλύτερο speedup.

### OpenMP

Στο μοντέλο OpenMP παρατηρείται το μεγαλύτερο speedup στα 16 threads παρόλο που στο σύστημα μας έχουμε 8 threads.

Εικόνα που περιέχει γραμμή, γράφημα, διάγραμμα, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

### OpenMP Tasks

Παρατηρούμε στα tasks ότι λαμβάνουμε καλύτερους χρόνους χρησιμοποιώντας 8 threads.

Εικόνα που περιέχει γραμμή, γράφημα, αριθμός, διάγραμμα

Περιγραφή που δημιουργήθηκε αυτόματα

### MPI

Εδώ στο σύστημα μας μπορούμε να χρησιμοποιήσουμε έως 8 processes οπότε πραγματοποιούμε τις μετρήσεις για 4 και 8 processes.

Εικόνα που περιέχει γραμμή, γράφημα, διάγραμμα

Περιγραφή που δημιουργήθηκε αυτόματα

## Σύγκριση Μεθόδων

Για την σύγκριση μεταξύ των μεθόδων υλοποιήθηκε το script plots.py σε python με σκοπό την σχεδίαση των διαγραμμάτων. Για να εξασφαλιστούν ορθές τιμές η κάθε υλοποίηση παραλληλοποίησης εκτελείτε πολλαπλές φορές (5) και κρατάει τον μέσο όρο για την σχεδίαση των τιμών για τη δημιουργία των διαγραμμάτων. Για να έχουμε καλύτερα αποτελέσματα το seeding  σε κάθε υλοποίηση έχει διαμορφωθεί με αυτο το σκοπό. Το tseed και ntrial είναι σταθεροί παράγοντες στην εκάστοτε υλοποίηση αλλά στο openmp προστίθεται το thread num ενώ στην mpi προστίθεται το rank. Με αυτό τον τρόπο παρατηρήσαμε ότι λαμβάνουμε το καλύτερο δυνατό f(x).

Εικόνα που περιέχει γραμμή, διάγραμμα, γράφημα, παράλληλα

Περιγραφή που δημιουργήθηκε αυτόματα

# Συμπεράσματα

Αρχικά βλέπουμε πως η παραλληλοποίηση μας επιστρέφει σημαντική βελτίωση στην χρονική απόδοση του συστήματος. Είναι σημαντικό να αναφερθεί πως η παραλληλοποίηση διαφέρει ανάλογα με το σύστημα στο οποίο χρησιμοποιείται (ex. αρχιτεκτονική, single core speed, total threads). Πιο συγκεκριμένα, στη δική μας περίπτωση τον περισσότερο υπολογιστικό χρόνο τον δεσμεύει η usleep οπότε όταν υλοποιήθηκε η παραλληλοποίηση με tasks δόθηκε περισσότερο βάρος εκεί.

Συγκρίνοντας το χρόνο μεταξύ των τριών υλοποιήσεων παραλληλοποίησης βλέπουμε παραπλήσια χρονική απόδοση συνεπώς στο συγκεκριμένο πρόβλημα είναι εφικτή η χρήση και των τριών. Βέβαια OpenMP Tasks χρειάστηκε ανάλυση του κώδικα ώστε να εντοπιστεί το πιο χρονοβόρο σημείο το οποίο μπορεί να μην εφικτό σε μεγαλύτερα προβλήματα ή τουλάχιστον να μην προσφέρει σημαντική βελτίωση σε σχέση με το χρόνο που χρειάζεται για να παραλληλοποιηθούν μεμονωμένα σημεία. Από την άλλη πλευρά, το openmp προσφέρει το καλύτερο χρόνο και σε ένα πρόβλημα μεγαλύτερης κλίμακας αλλά ίδιας φύσης μπορεί η διαφορά μεταξύ των υλοποιήσεων να ήταν μεγαλύτερη και να η σωστή επιλογή να ήταν η χρήση του.

Συγκρίνοντας το f(x) βλέπουμε το καλύτερο αποτέλεσμα στο OpenMP tasks οπότε σε ένα πρόβλημα που το error είναι ο κύριος παράγοντας θα ήταν προτιμότερη χρήση του. Βέβαια παρατηρώντας το efficiency το OpenMP δεν χρησιμοποιεί τους πόρους του συστήματος τόσο αποδοτικά. Δηλαδή η αύξηση του χρόνου που προσφέρει δεν είναι αρκετή για τους πόρους που δεσμεύει. Σε ένα σύστημα που οι πόροι είναι περιορισμένοι ίσως να μην είναι η καλύτερη επιλογή.

Συνοψίζοντας, κάθε υλοποίηση έχει προτερήματα και μειονεκτήματα το οποίο επηρεάζονται σημαντικά από αρκετούς παράγοντες. Είναι σημαντικό σε κάθε πρόβλημα να εξετάζεται τι θα αποφέρει τα καλύτερα αποτελέσματα ανάλογα με τις απαιτήσεις. Στο δικό μας πρόβλημα και οι 3 λύσεις μας επιστρέφουν πολύ συγκρίσιμα αποτελέσματα και δεν μπορούν να αποφανθούμε πως μια υλοποίηση υπερτερεί μιας άλλης γιατί αυτό εξαρτάται από το ποιος παράγοντας έχει περισσότερη βαρύτητα ( execution time, efficiency, best f(x)),