Πολυτεχνική Σχολή

Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών & Πληροφορικής

CEID

Παράλληλη Επεξεργασία 2020-2021

Εργασία

"Παράλληλη Ολική Βελτιστοποίηση"

ΑΒΡΑΜΟΠΟΥΛΟΣ ΜΙΧΑΗΛ 1067451 | ΚΑΡΑΓΙΑΝΝΗΣ ΑΛΕΞΑΝΔΡΟΣ 1067452 | ΠΑΝΤΑΖΑΡΑΣ ΕΥΣΤΡΑΤΙΟΣ 1067490

Περιεχόμενα

A) Εισαγωγή

Κοιτώντας τον κώδικα που δίνεται για την υλοποίηση του αλγορίθμου Multistart για τη συνάρτηση Rosenbrock παρατηρούμε δύο σημεία:

- το μεγαλύτερο μέρος της σειριακής καθυστέρησης προέρχεται από το for loop για όλα τα trials (περίπου 128000) στην συνάρτηση main
- η υλοποίηση χρησιμοποιεί random number generator, άρα σε κάποια σημεία δεν έχει σημασία η σειρά εκτέλεσης

Επίσης για την συνάρτηση Rosenbrock ξέρουμε πως έχει καθολικό ελάχιστο στο σημείο (1,1,...,1), όπου παίρνει την τιμή 0, χρήσιμο για να ελέγχουμε κάθε φορά την ορθότητα της υλοποίησης.

Για την εργασία χρησιμοποιήσαμε το πρόγραμμα cygwin για εξομοίωση συστήματος Linux (για compile & run των αρχείων c). Για την συγγραφή του κώδικα χρησιμοποιήσαμε το IDE Visual Studio Code. Ο υπολογιστής που χρησιμοποιήσαμε έχει cpu με 6 cores και 12 threads στα 3.9Ghz max.

B) Αναλυτική τεκμηρίωση παράλληλων υλοποιήσεων και τυχόν βελτιστοποιήσεων

Σε κάθε περίπτωση ξεκινώντας από τον αρχικό κώδικα κάνουμε στοχευμένες αλλαγές για να παραλληλοποιήσουμε το πρόγραμμα (δείχνουμε μόνο τις αλλαγές για λόγους συντομίας).

OpenMP:

Αρχικά κάνουμε include <omp.h>. Μετά μέσα στην main λέμε στην OpenMP να βελτιστοποιήσει τον κεντρικό βρόχο των ntrials:

```
    int num_threads = 12;
#pragma omp parallel num_threads(num_threads) firstprivate(fx, jj, i) /*Εκκίνηση παραλλη
    {
#pragma omp for /*Παραλληλοποίηση βρόχου*/
        for (trial = 0; trial < ntrials; trial++)
        {
            /* starting guess for rosenbrock test function, search space in [-4, 4) */
            for (i = 0; i < nvars; i++)
            {
                startpt[i] = 4.0 * drand48() - 4.0;
```


ΑΒΡΑΜΟΠΟΥΛΟΣ ΜΙΧΑΗΛ 1067451 | ΚΑΡΑΓΙΑΝΝΗΣ ΑΛΕΞΑΝΔΡΟΣ 1067452 | ΠΑΝΤΑΖΑΡΑΣ ΕΥΣΤΡΑΤΙΟΣ 1067490

Όπως βλέπουμε αυτό γίνεται με τη δημιουργία ενός omp parallel block και ορίζοντας ποιες μεταβλητές θέλουμε να είναι private σε κάθε αυτόνομο thread. Περικλείω επίσης και τον βρόχο σε ένα omp for block. Σημαντικό είναι εδώ να μην ξεχάσουμε στο σημείο του βρόχου που ελέγχει για νέο minimum να χρησιμοποιήσουμε omp critical για να μην χάσουμε κανένα νέο ελάχιστο:

```
#pragma omp critical /*mutex περιοχή για τις μετ
            {
                if (fx < best_fx)
                {
                    best_trial = trial;
                    best_jj = jj;
                    best_fx = fx;
                    for (i = 0; i < nvars; i++)
                        best_pt[i] = endpt[i];
                }
            }
        }
```

Προσέχουμε τέλος μια λεπτομέρεια για να εμφανίζεται στο τέλος σωστά το νούμερο των function evaluations (χρειάζεται να πολλαπλασιάσουμε με τον αριθμό των threads):

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

OpenMP tasks:

Παρόμοια υλοποίηση με την κανονική OpenMP, όμως με τη χρήση Tasks. Πρακτικά αλλάζει μόνο το σημείο πάνω από το loop όπου ουσιαστικά ορίζουμε ένα task (με τις ίδιες private variables) για κάθε iteration του βρόχου:

```
#pragma omp parallel num_threads(num_threads)  /*Εκκίνηση
{
    #pragma omp single nowait /*Παραλληλοποίηση βρόχου*/
        for (trial = 0; trial < ntrials; trial++) {
            #pragma omp task firstprivate(fx, jj, i)
            {
                /* starting guess for rosenbrock test func
                for (i = 0; i < nvars; i++) {
```

Ουσιαστικά λέμε με το omp single nowait στην OpenMP να τρέξει με thread τα tasks που δημιουργούνται από το loop, χωρίς να περιμένει να τελειώσει το κάθε ένα. Είναι ισοδύναμη υλοποίηση με την παραπάνω.

MPI:

Αρχικά κάνουμε include <mpi.h>. Ορίζουμε κάποιες μεταβλητές που θα χρειαστούμε:

```
int myid, p; /* MPI μεταβλητ
int source;
int tag = 1234;
int dest = 0;
MPI_Status status;
```

Και τώρα πρέπει να παραλληλοποιήσουμε τον βρόχο. Η ιδέα μας ήταν πως αφού δεν απαιτείται κάποια σειρά στις τυχαίες δοκιμές του αλγορίθμου, απλώς τον «σπάμε» σε όσα cores μας δίνονται και μετά συλλέγουμε τα αποτελέσματα από κάθε core στο “master” core με rank 0 που γίνεται και η τελική σύγκριση. Πριν από το for loop αρχικοποιούμε το MPI (και το random number generator!)

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes */
srand48(time(0)*(myid+1));
for (trial = 0; trial < ntrials / p; trial++)
{
```

Με αυτά το κάθε core ξέρει το id του και ξέρει τον συνολικό αριθμό από cores. Τρέχουμε το βρόχο ntrials/p φορές (αφού έχουμε p cores). Αφού τελειώσουν όλα τα cores το «master» συλλέγει τα υπόλοιπα:

```
if (myid != 0)
{
    MPI_Send(&best_fx, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD); /* Στέλνουμε τα δεδομένα
    MPI_Send(&best_jj, 1, MPI_INT, dest, tag + 1, MPI_COMM_WORLD);
    MPI_Send(best_pt, MAXVARS, MPI_DOUBLE, dest, tag + 2, MPI_COMM_WORLD);
    MPI_Send(&best_trial, 1, MPI_INT, dest, tag + 3, MPI_COMM_WORLD);
}
else
{
    for (j = 1; j < p; j++)
    {
        source = j;
        MPI_Recv(&fx, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&jj, 1, MPI_INT, source, tag + 1, MPI_COMM_WORLD, &status);
        MPI_Recv(endpt, MAXVARS, MPI_DOUBLE, source, tag + 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&trial, 1, MPI_INT, source, tag + 3, MPI_COMM_WORLD, &status);
        if (fx < best_fx)
        {
            best_trial = trial;
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
                best_pt[i] = endpt[i];
        }
    }
```


ΑΒΡΑΜΟΠΟΥΛΟΣ ΜΙΧΑΗΛ 1067451 | ΚΑΡΑΓΙΑΝΝΗΣ ΑΛΕΞΑΝΔΡΟΣ 1067452 | ΠΑΝΤΑΖΑΡΑΣ ΕΥΣΤΡΑΤΙΟΣ 1067490

Στην «αποστολή» και «παραλαβή» μηνυμάτων, εκτός από τους pointers στα δεδομένα δίνουμε:

- τον τύπο του δεδομένου (έχει κάποια defined η MPI και μπορούμε να ορίσουμε και custom)
- το id του destination core (μόνο για αποστολές)
- το id του source core (μόνο για παραλαβές)
- ένα tag που διαχωρίζει όμοια μηνύματα σε πιο περίπλοκες υλοποιήσεις (εμάς μας είναι άχρηστο)
- το communicator που θέλουμε να χρησιμοποιήσουμε
- μια μεταβλητή MPI Status για να αποθηκευτεί το status

Τέλος όπως και πριν προσέχουμε να πολλαπλασιάσουμε το τελικό function evaluations με τον αριθμό των cores:

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

Κάτω-κάτω τερματίζουμε το MPI:

```
MPI_Finalize();
```

MPI+OpenMP:

Όπως λέει και ο τίτλος, εδώ έχουμε στοιχεία και από τις δύο επιμέρους υλοποιήσεις. Προφανώς κάνουμε include και τις δύο βιβλιοθήκες. Πάλι ορίζουμε μεταβλητές MPI:

```
int myid, p; /* MPI μεταβλ
int source;
int tag = 1234;
int dest = 0;
MPI_Status status;
```

Αρχικοποιούμε το MPI και σπάμε το loop σε p κομμάτια:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes */
srand48(time(0) * (myid + 1));
```

```
for (trial = 0; trial < ntrials / p; trial++)
```

Μόνο που αυτή τη φορά λέμε και στην OMP να παραλληλοποιήσει τον βρόχο όπως νωρίτερα:

```
#pragma omp parallel num_threads(num_threads) firstprivate(fx, jj, i) /*Εκκ
    {
#pragma omp for /*Παραλληλοποίηση βρόχου*/
        for (trial = 0; trial < ntrials / p; trial++)
        {
```

Προσέχουμε το critical section:

```
#pragma omp critical /*mutex περιοχή για τις μεταβλητές
            {
                if (fx < best_fx)
                {
                    best_trial = trial * (myid + 1);
                    best_jj = jj;
                    best_fx = fx;
                    for (i = 0; i < nvars; i++)
                        best_pt[i] = endpt[i];
                }
            }
```

Έξω από το loop και πάλι στέλνουμε τα αποτελέσματα στο “master” core:

```
if (myid != 0)
{
    MPI_Send(&best_fx, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD); /* Στέλνουμε τα δεδομένα τ
    MPI_Send(&best_jj, 1, MPI_INT, dest, tag + 1, MPI_COMM_WORLD);
    MPI_Send(best_pt, MAXVARS, MPI_DOUBLE, dest, tag + 2, MPI_COMM_WORLD);
    MPI_Send(&best_trial, 1, MPI_INT, dest, tag + 3, MPI_COMM_WORLD);
}
else
{
    for (j = 1; j < p; j++)
    {
        source = j;
        MPI_Recv(&fx, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&jj, 1, MPI_INT, source, tag + 1, MPI_COMM_WORLD, &status);
        MPI_Recv(endpt, MAXVARS, MPI_DOUBLE, source, tag + 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&trial, 1, MPI_INT, source, tag + 3, MPI_COMM_WORLD, &status);
        if (fx < best_fx)
        {
            best_trial = trial;
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
                best_pt[i] = endpt[i];
        }
    }
```

Και το print:

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

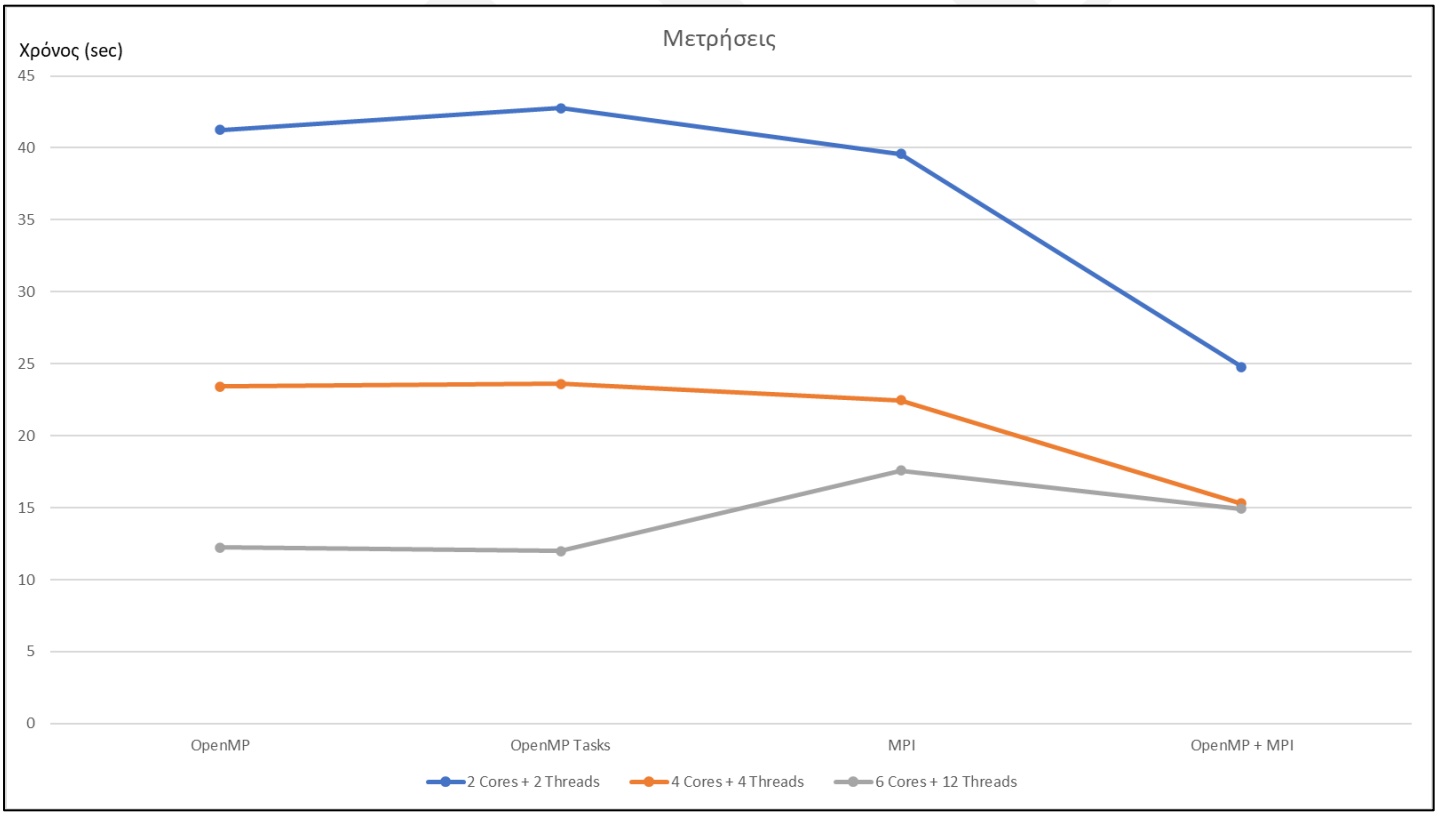
Κλείνουμε το MPI:

```
MPI_Finalize();
```

Γ) Αποτελέσματα

Εκτελώντας τις 4 εναλλακτικές υλοποιήσεις για τις ακόλουθες 3 διαφορετικές ρυθμίσεις cores/threads λαμβάνουμε τα αποτελέσματα:

- 2cores-2threads
- 4cores-4threads
- 6cores-12threads



Στον κατακόρυφο άξονα φαίνεται ο χρόνος εκτέλεσης σε δευτερόλεπτα, ενώ στον οριζόντιο το είδος της υλοποίησης. Βλέπουμε πως οι διαδοχικές καμπύλες απέχουν όλο και λιγότερο όσο πλησιάζουμε το «όριο» της παραλληλοποίησης

Δ) Συμπεράσματα

Παρατηρούμε πως στις «λογικές» περιπτώσεις όπου δεν χρησιμοποιούμε 100% των πυρήνων και threads υπάρχει μια καθαρή διαφορά μεταξύ των υλοποιήσεων: Η OpenMP είναι σαφώς γρηγορότερη από την σειριακή, η OpenMP-Tasks είναι σχεδόν ίδια με την OpenMP, η MPI είναι ακόμα γρηγορότερη από αυτές και τέλος η «υβριδική» OpenMP+MPI είναι σημαντικά καλύτερη από όλες τις υπόλοιπες. Αν τις διατάξουμε με σειρά απόδοσης δηλαδή φαίνεται πως:

OpenMP+MPI > MPI > OpenMP-Tasks = OpenMP >> Serial

Στην τελευταία δοκιμή που χρησιμοποιούμε 100% των διαθέσιμων πόρων στο μηχάνημά μας (6 πυρήνες 12 νήματα) τα στατιστικά αρχίζουν και αλλάζουν λίγο. Παρατηρούμε πως η διάταξη γίνεται:

OpenMP-Tasks = OpenMP > OpenMP+MPI > MPI >> Serial

Αυτό που μάλλον συμβαίνει σε αυτήν την περίπτωση είναι πως επειδή πλησιάζουμε το σημείο «κορεσμού» των παράλληλων λειτουργιών της μηχανής με τα 12 thread, η παραλληλοποίηση με cores χρήσει του MPI έχει σημαντικό performance overhead, τόσο που δεν αξίζει η χρήση της βιβλιοθήκης «μόνο» για 128000 iterations.