ΠΑΡΑΛΛΗΛΗ ΕΠΕΞΕΡΓΑΣΙΑ

03/07/2022

Τριαντάφυλλος Πράππας 1067504

Εισαγωγή

PC Specs

CPU	**Ryzen 7 3700x**
RAM	16GB 3200Mhz
OS	Windows 10 Home

Οι παρακάτω κώδικες υλοποιήθηκαν στο Visual Studio Code με την χρήση του WSL: Ubuntu 20.04 καθώς και την δυνατότητα των Windows 10, Windows Subsystem for Linux

Για να τρέξουμε τον κώδικα OpenMP:
gcc -fopenmp -O3 -o omp.c
./omp

Για να τρέξουμε τον κώδικα OpenMP Task:
gcc -fopenmp -O3 -o omp_tasks.c
./omp_tasks

Για να τρέξουμε τον κώδικα MPI:
mpicc -g -O3 -o mpi mpi.c
mpirun -c 6 mpi

Για να τρέξουμε τον κώδικα MPI + OpenMP:
mpicc -g -O3 -o mpi_omp mpi_omp.c
mpirun -c 6 mpi_omp

Κώδικες

OpenMP:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h> /*Openmp Library*/

#define MAXVARS     (250)   /* max # of variables        */
#define RHO_BEGIN   (0.5)   /* stepsize geometric shrink */
#define EPSMIN      (1E-6)  /* ending value of stepsize  */
#define IMAX        (5000)  /* max # of iterations       */

/* global variables */
unsigned long funevals = 0;


/* Rosenbrocks classic parabolic valley ("banana") function */
double f(double *x, int n)
{
    double fv;
    int i;

    funevals++;
    fv = 0.0;
    for (i=0; i<n-1; i++)   /* rosenbrock */
        fv = fv + 100.0*pow((x[i+1]-x[i]*x[i]),2) + pow((x[i]-1.0),2);

    return fv;
}

/* given a point, look for a better one nearby, one coord at a time */
double best_nearby(double delta[MAXVARS], double point[MAXVARS], double
prevbest, int nvars)
{
    double z[MAXVARS];
    double minf, ftmp;
    int i;
    minf = prevbest;
    for (i = 0; i < nvars; i++)
        z[i] = point[i];
    for (i = 0; i < nvars; i++) {
        z[i] = point[i] + delta[i];
        ftmp = f(z, nvars);
        if (ftmp < minf)
            minf = ftmp;
        else {
            delta[i] = 0.0 - delta[i];
            z[i] = point[i] + delta[i];
            ftmp = f(z, nvars);
            if (ftmp < minf)
                minf = ftmp;
            else
                z[i] = point[i];
        }
    }
    for (i = 0; i < nvars; i++)
        point[i] = z[i];

    return (minf);
}


int hooke(int nvars, double startpt[MAXVARS], double endpt[MAXVARS],
double rho, double epsilon, int itermax)
{
    double delta[MAXVARS];
    double newf, fbefore, steplength, tmp;
    double xbefore[MAXVARS], newx[MAXVARS];
    int i, j, keep;
    int iters, iadj;

    for (i = 0; i < nvars; i++) {
        newx[i] = xbefore[i] = startpt[i];
        delta[i] = fabs(startpt[i] * rho);
        if (delta[i] == 0.0)
            delta[i] = rho;
    }
    iadj = 0;
    steplength = rho;
    iters = 0;
    fbefore = f(newx, nvars);
    newf = fbefore;
    while ((iters < itermax) && (steplength > epsilon)) {
        iters++;
        iadj++;
#if DEBUG
        printf("\nAfter %5d funevals, f(x) =  %.4le at\n", funevals,
fbefore);
        for (j = 0; j < nvars; j++)
            printf("    x[%2d] = %.4le\n", j, xbefore[j]);
#endif
        /* find best new point, one coord at a time */
        for (i = 0; i < nvars; i++) {
            newx[i] = xbefore[i];
        }
        newf = best_nearby(delta, newx, fbefore, nvars);
        /* if we made some improvements, pursue that direction */
        keep = 1;
        while ((newf < fbefore) && (keep == 1)) {
            iadj = 0;
            for (i = 0; i < nvars; i++) {
                /* firstly, arrange the sign of delta[] */
                if (newx[i] <= xbefore[i])
                    delta[i] = 0.0 - fabs(delta[i]);
                else
                    delta[i] = fabs(delta[i]);
                /* now, move further in this direction */
                tmp = xbefore[i];
                xbefore[i] = newx[i];
                newx[i] = newx[i] + newx[i] - tmp;
            }
            fbefore = newf;
            newf = best_nearby(delta, newx, fbefore, nvars);
            /* if the further (optimistic) move was bad.... */
            if (newf >= fbefore)
                break;

            /* make sure that the differences between the new */
            /* and the old points are due to actual */
            /* displacements; beware of roundoff errors that */
            /* might cause newf < fbefore */
            keep = 0;
            for (i = 0; i < nvars; i++) {
                keep = 1;
                if (fabs(newx[i] - xbefore[i]) > (0.5 *
fabs(delta[i])))
                    break;
                else
                    keep = 0;
            }
        }
        if ((steplength >= epsilon) && (newf >= fbefore)) {
            steplength = steplength * rho;
            for (i = 0; i < nvars; i++) {
                delta[i] *= rho;
            }
        }
    }
    for (i = 0; i < nvars; i++)
        endpt[i] = xbefore[i];

    return (iters);
```

```
}


double get_wtime(void)
{
    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv_sec + (double)t.tv_usec*1.0e-6;
}

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();
    int num_threads = 16;
    #pragma omp parallel num_threads(num_threads) firstprivate(fx, jj,
i) /*Starts Parallelization and Creates Threads with num_threads(int)
parameter, we use firstprivate to initialize our values*/
    {
         #pragma omp for /*Parallelization*/
            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;
                }

                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 /*In case we have 2 best_fx at the
same time*/
            {
                    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 *
num_threads); /*Multiply with num_threads to find the number of
evaluations (it is shared in n threads)*/
    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;
}
```

OpenMP Tasks:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h> /*εισάγουμε την βιβλιοθήκη του Openmp*/

#define MAXVARS     (250)   /* max # of variables        */
#define RHO_BEGIN   (0.5)   /* stepsize geometric shrink */
#define EPSMIN      (1E-6)  /* ending value of stepsize  */
#define IMAX        (5000)  /* max # of iterations       */

/* global variables */
unsigned long funevals = 0;


/* Rosenbrocks classic parabolic valley ("banana") function */
double f(double *x, int n)
{
    double fv;
    int i;

    funevals++;
    fv = 0.0;
    for (i=0; i<n-1; i++)   /* rosenbrock */
        fv = fv + 100.0*pow((x[i+1]-x[i]*x[i]),2) + pow((x[i]-1.0),2);

    return fv;
}

/* given a point, look for a better one nearby, one coord at a time */
double best_nearby(double delta[MAXVARS], double point[MAXVARS], double
prevbest, int nvars)
{
    double z[MAXVARS];
    double minf, ftmp;
    int i;
    minf = prevbest;
    for (i = 0; i < nvars; i++)
        z[i] = point[i];
    for (i = 0; i < nvars; i++) {
        z[i] = point[i] + delta[i];
        ftmp = f(z, nvars);
        if (ftmp < minf)
            minf = ftmp;
        else {
            delta[i] = 0.0 - delta[i];
            z[i] = point[i] + delta[i];
            ftmp = f(z, nvars);
            if (ftmp < minf)
                minf = ftmp;
            else
                z[i] = point[i];
        }
    }
    for (i = 0; i < nvars; i++)
        point[i] = z[i];

    return (minf);
}


int hooke(int nvars, double startpt[MAXVARS], double endpt[MAXVARS],
double rho, double epsilon, int itermax)
{
    double delta[MAXVARS];
    double newf, fbefore, steplength, tmp;
    double xbefore[MAXVARS], newx[MAXVARS];
    int i, j, keep;
    int iters, iadj;

    for (i = 0; i < nvars; i++) {
        newx[i] = xbefore[i] = startpt[i];
        delta[i] = fabs(startpt[i] * rho);
        if (delta[i] == 0.0)
            delta[i] = rho;
    }
    iadj = 0;
    steplength = rho;
    iters = 0;
    fbefore = f(newx, nvars);
    newf = fbefore;
    while ((iters < itermax) && (steplength > epsilon)) {
        iters++;
        iadj++;
#if DEBUG
        printf("\nAfter %5d funevals, f(x) =  %.4le at\n", funevals,
fbefore);
        for (j = 0; j < nvars; j++)
            printf("   x[%2d] = %.4le\n", j, xbefore[j]);
#endif
        /* find best new point, one coord at a time */
        for (i = 0; i < nvars; i++) {
            newx[i] = xbefore[i];
        }
        newf = best_nearby(delta, newx, fbefore, nvars);
        /* if we made some improvements, pursue that direction */
        keep = 1;
        while ((newf < fbefore) && (keep == 1)) {
            iadj = 0;
            for (i = 0; i < nvars; i++) {
                /* firstly, arrange the sign of delta[] */
                if (newx[i] <= xbefore[i])
                    delta[i] = 0.0 - fabs(delta[i]);
                else
                    delta[i] = fabs(delta[i]);
                /* now, move further in this direction */
                tmp = xbefore[i];
                xbefore[i] = newx[i];
                newx[i] = newx[i] + newx[i] - tmp;
            }
            fbefore = newf;
            newf = best_nearby(delta, newx, fbefore, nvars);
            /* if the further (optimistic) move was bad.... */
            if (newf >= fbefore)
                break;

            /* make sure that the differences between the new */
            /* and the old points are due to actual */
            /* displacements; beware of roundoff errors that */
            /* might cause newf < fbefore */
            keep = 0;
            for (i = 0; i < nvars; i++) {
                keep = 1;
                if (fabs(newx[i] - xbefore[i]) > (0.5 *
fabs(delta[i])))
                    break;
                else
                    keep = 0;
            }
        }
        if ((steplength >= epsilon) && (newf >= fbefore)) {
            steplength = steplength * rho;
            for (i = 0; i < nvars; i++) {
                delta[i] *= rho;
            }
        }
    }
    for (i = 0; i < nvars; i++)
        endpt[i] = xbefore[i];

    return (iters);
}
```

```c
double get_wtime(void)
{
    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv_sec + (double)t.tv_usec*1.0e-6;
}

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();
    int num_threads = 16;
    #pragma omp parallel num_threads(num_threads)  /*Starts
Parallelization and Creates Threads with num_threads(int) parameter, we
use firstprivate to initialize our values*/
    {
         #pragma omp single nowait /*The code below can be processed by
any thread without waiting for the other threads to end their
processes*/
            for (trial = 0; trial < ntrials; trial++) {
                #pragma omp task firstprivate(fx, jj, i) /*Tasks
initialization to calculate the function*/
                {
                    /* starting guess for rosenbrock test function,
search space in [-4, 4) */
                    for (i = 0; i < nvars; i++) {
                        startpt[i] = 4.0*drand48()-4.0;
                    }

                    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 /*In case we have 2 best_fx at the
same time*/
                {
                        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 *
num_threads); /*Multiply with num_threads to find the number of
evaluations (it is shared in n threads)*/
    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;
}
```

MPI:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define MAXVARS (250)    /* max # of variables         */
#define RHO_BEGIN (0.5) /* stepsize geometric shrink */
#define EPSMIN (1E-6)    /* ending value of stepsize  */
#define IMAX (5000)      /* max # of iterations       */

/* global variables */
unsigned long funevals = 0;

/* Rosenbrocks classic parabolic valley ("banana") function */
double f(double *x, int n)
{
    double fv;
    int i;

    funevals++;
    fv = 0.0;
    for (i = 0; i < n - 1; i++) /* rosenbrock */
        fv = fv + 100.0 * pow((x[i + 1] - x[i] * x[i]), 2) + pow((x[i]
- 1.0), 2);

    return fv;
}

/* given a point, look for a better one nearby, one coord at a time */
double best_nearby(double delta[MAXVARS], double point[MAXVARS], double
prevbest, int nvars)
{
    double z[MAXVARS];
    double minf, ftmp;
    int i;
    minf = prevbest;
    for (i = 0; i < nvars; i++)
        z[i] = point[i];
    for (i = 0; i < nvars; i++)
    {
        z[i] = point[i] + delta[i];
        ftmp = f(z, nvars);
        if (ftmp < minf)
            minf = ftmp;
        else
        {
            delta[i] = 0.0 - delta[i];
            z[i] = point[i] + delta[i];
            ftmp = f(z, nvars);
            if (ftmp < minf)
                minf = ftmp;
            else
                z[i] = point[i];
        }
    }
    for (i = 0; i < nvars; i++)
        point[i] = z[i];

    return (minf);
}

int hooke(int nvars, double startpt[MAXVARS], double endpt[MAXVARS],
double rho, double epsilon, int itermax)
{
    double delta[MAXVARS];
    double newf, fbefore, steplength, tmp;
    double xbefore[MAXVARS], newx[MAXVARS];
    int i, j, keep;
    int iters, iadj;

    for (i = 0; i < nvars; i++)
    {
        newx[i] = xbefore[i] = startpt[i];
        delta[i] = fabs(startpt[i] * rho);
        if (delta[i] == 0.0)
            delta[i] = rho;
    }
    iadj = 0;
    steplength = rho;
    iters = 0;
    fbefore = f(newx, nvars);
    newf = fbefore;
    while ((iters < itermax) && (steplength > epsilon))
    {
        iters++;
        iadj++;
#if DEBUG
        printf("\nAfter %5d funevals, f(x) =  %.4le at\n", funevals,
fbefore);
        for (j = 0; j < nvars; j++)
            printf("   x[%2d] = %.4le\n", j, xbefore[j]);
#endif
```

```
        /* find best new point, one coord at a time */
        for (i = 0; i < nvars; i++)
        {
            newx[i] = xbefore[i];
        }
        newf = best_nearby(delta, newx, fbefore, nvars);
        /* if we made some improvements, pursue that direction */
        keep = 1;
        while ((newf < fbefore) && (keep == 1))
        {
            iadj = 0;
            for (i = 0; i < nvars; i++)
            {
                /* firstly, arrange the sign of delta[] */
                if (newx[i] <= xbefore[i])
                    delta[i] = 0.0 - fabs(delta[i]);
                else
                    delta[i] = fabs(delta[i]);
                /* now, move further in this direction */
                tmp = xbefore[i];
                xbefore[i] = newx[i];
                newx[i] = newx[i] + newx[i] - tmp;
            }
            fbefore = newf;
            newf = best_nearby(delta, newx, fbefore, nvars);
            /* if the further (optimistic) move was bad.... */
            if (newf >= fbefore)
                break;

            /* make sure that the differences between the new */
            /* and the old points are due to actual */
            /* displacements; beware of roundoff errors that */
            /* might cause newf < fbefore */
            keep = 0;
            for (i = 0; i < nvars; i++)
            {
                keep = 1;
                if (fabs(newx[i] - xbefore[i]) > (0.5 *
fabs(delta[i])))
                    break;
                else
                    keep = 0;
            }
        }
        if ((steplength >= epsilon) && (newf >= fbefore))
        {
            steplength = steplength * rho;
            for (i = 0; i < nvars; i++)
```

```c
            {
                delta[i] *= rho;
            }
        }
    }
    for (i = 0; i < nvars; i++)
        endpt[i] = xbefore[i];

    return (iters);
}

double get_wtime(void)
{
    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv_sec + (double)t.tv_usec * 1.0e-6;
}

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;
    int i, j, jj;
    double t0, t1, fx;

    double best_fx = 1e10;
    double best_pt[MAXVARS];
    int best_trial = -1;
    int best_jj = -1;

    int myid, p; /*MPI values for rank and size*/
    int source; /*Variable to initialize the source of messages*/
    int tag = 1024; /*Variable to initialize tag*/
    int dest = 0; /*Variable to initialize the destination of
messages*/
    MPI_Status status; /*Vatiable for message status*/
    for (i = 0; i < MAXVARS; i++)
        best_pt[i] = 0.0;

    ntrials = 64 * 1024; /* number of trials */
    nvars = 32;           /* number of variables (problem dimension) */
    t0 = get_wtime();
    MPI_Init(&argc, &argv); /*Starts Share Process*/
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes */
    srand48(1*(myid+1)); /*each process gets random values*/
    for (trial = 0; trial < ntrials / p; trial++) /*Split the number of
trials for each process (if not every process would have ntrials)*/
    {

        /* starting guess for rosenbrock test function, search space in
[-4, 4) */
        for (i = 0; i < nvars; i++)
        {
            startpt[i] = 4.0 * drand48() - 4.0;
        }

        jj = hooke(nvars, startpt, endpt, rho, epsilon, itermax);
#if DEBUGαριθμός!
        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 * (myid + 1);
            best_jj = jj;
            best_fx = fx;
            for (i = 0; i < nvars; i++)
                best_pt[i] = endpt[i];
        } /*Calculates best value to reduce number of messages*/
    }

    if (myid != 0) /*Every process with id != 0 send their result to
master process*/
    {
        MPI_Send(&best_fx, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
/*Sends results of processes to master process*/
        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++) /*Worker-process*/
        {
            source = j;
            MPI_Recv(&fx, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
&status); /*Receives results of each process serially*/
            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) /*Compares starting data of master
process with process with id = 0*/
            {
                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
* p); /*Multiply with number of processes to get number of function
evalutations*/
        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;
}
```

MPI + OpenMP:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h> /*εισάγουμε την βιβλιοθήκη του Openmp*/
#include <mpi.h>

#define MAXVARS (250)   /* max # of variables        */
#define RHO_BEGIN (0.5) /* stepsize geometric shrink */
#define EPSMIN (1E-6)   /* ending value of stepsize  */
#define IMAX (5000)     /* max # of iterations       */

/* global variables */
unsigned long funevals = 0;

/* Rosenbrocks classic parabolic valley ("banana") function */
double f(double *x, int n)
{
    double fv;
    int i;

    funevals++;
    fv = 0.0;
    for (i = 0; i < n - 1; i++) /* rosenbrock */
        fv = fv + 100.0 * pow((x[i + 1] - x[i] * x[i]), 2) + pow((x[i]
- 1.0), 2);

    return fv;
}

/* given a point, look for a better one nearby, one coord at a time */
double best_nearby(double delta[MAXVARS], double point[MAXVARS], double
prevbest, int nvars)
{
    double z[MAXVARS];
    double minf, ftmp;
    int i;
    minf = prevbest;
    for (i = 0; i < nvars; i++)
        z[i] = point[i];
    for (i = 0; i < nvars; i++)
    {
        z[i] = point[i] + delta[i];
        ftmp = f(z, nvars);
        if (ftmp < minf)
            minf = ftmp;
        else
        {
            delta[i] = 0.0 - delta[i];
            z[i] = point[i] + delta[i];
            ftmp = f(z, nvars);
            if (ftmp < minf)
                minf = ftmp;
            else
                z[i] = point[i];
        }
    }
    for (i = 0; i < nvars; i++)
        point[i] = z[i];

    return (minf);
}

int hooke(int nvars, double startpt[MAXVARS], double endpt[MAXVARS],
double rho, double epsilon, int itermax)
{
    double delta[MAXVARS];
    double newf, fbefore, steplength, tmp;
    double xbefore[MAXVARS], newx[MAXVARS];
    int i, j, keep;
    int iters, iadj;

    for (i = 0; i < nvars; i++)
    {
        newx[i] = xbefore[i] = startpt[i];
        delta[i] = fabs(startpt[i] * rho);
        if (delta[i] == 0.0)
            delta[i] = rho;
    }
    iadj = 0;
    steplength = rho;
    iters = 0;
    fbefore = f(newx, nvars);
    newf = fbefore;
    while ((iters < itermax) && (steplength > epsilon))
    {
        iters++;
        iadj++;
#if DEBUG
        printf("\nAfter %5d funevals, f(x) =  %.4le at\n", funevals,
fbefore);
        for (j = 0; j < nvars; j++)
```

```c
            printf("   x[%2d] = %.4le\n", j, xbefore[j]);
#endif
        /* find best new point, one coord at a time */
        for (i = 0; i < nvars; i++)
        {
            newx[i] = xbefore[i];
        }
        newf = best_nearby(delta, newx, fbefore, nvars);
        /* if we made some improvements, pursue that direction */
        keep = 1;
        while ((newf < fbefore) && (keep == 1))
        {
            iadj = 0;
            for (i = 0; i < nvars; i++)
            {
                /* firstly, arrange the sign of delta[] */
                if (newx[i] <= xbefore[i])
                    delta[i] = 0.0 - fabs(delta[i]);
                else
                    delta[i] = fabs(delta[i]);
                /* now, move further in this direction */
                tmp = xbefore[i];
                xbefore[i] = newx[i];
                newx[i] = newx[i] + newx[i] - tmp;
            }
            fbefore = newf;
            newf = best_nearby(delta, newx, fbefore, nvars);
            /* if the further (optimistic) move was bad.... */
            if (newf >= fbefore)
                break;

            /* make sure that the differences between the new */
            /* and the old points are due to actual */
            /* displacements; beware of roundoff errors that */
            /* might cause newf < fbefore */
            keep = 0;
            for (i = 0; i < nvars; i++)
            {
                keep = 1;
                if (fabs(newx[i] - xbefore[i]) > (0.5 *
fabs(delta[i])))
                    break;
                else
                    keep = 0;
            }
        }
        if ((steplength >= epsilon) && (newf >= fbefore))
        {
            steplength = steplength * rho;
            for (i = 0; i < nvars; i++)
            {
                delta[i] *= rho;
            }
        }
    }
    for (i = 0; i < nvars; i++)
        endpt[i] = xbefore[i];

    return (iters);
}

double get_wtime(void)
{
    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv_sec + (double)t.tv_usec * 1.0e-6;
}

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, j;
    double t0, t1;

    double best_fx = 1e10;
    double best_pt[MAXVARS];
    int best_trial = -1;
    int best_jj = -1;

    int myid, p; /*MPI values for rank and size*/
    int source; /*Variable to initialize the source of messages*/
    int tag = 1024; /*Variable to initialize tag*/
    int dest = 0; /*Variable to initialize the destination of
messages*/
    MPI_Status status; /*Variable for message status*/

    for (i = 0; i < MAXVARS; i++)
        best_pt[i] = 0.0;
    ntrials = 64 * 1024; /* number of trials */
    nvars = 32;           /* number of variables (problem dimension) */
    t0 = get_wtime();
    int provided;
    MPI_Init(&argc, &argv); /*Starts Share Process*/
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes */
    srand48(1 * (myid + 1)); /*each process gets random values*/
#pragma omp parallel num_threads(12) firstprivate(fx, jj, i) /*Starts
Parallelization and Creates Threads with num_threads(int) parameter, we
use firstprivate to initialize our values*/
    {
#pragma omp for /*Parallelization*/
        for (trial = 0; trial < ntrials / p; trial++) /*Split the
number of trials for each process (if not every process would have
ntrials)*/
        {
            /* starting guess for rosenbrock test function, search
space in [-4, 4) */
            for (i = 0; i < nvars; i++)
            {
                startpt[i] = 4.0 * drand48() - 4.0;
            }

            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 /*In case we have 2 best_fx at the same time*/
            {
                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];
                }
            }
        }
    }

    if (myid != 0) /*Every process with id != 0 send their result to
master process*/
    {
        MPI_Send(&best_fx, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
/*Sends results of processes to master process*/
        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++) /*Worker-process*/
        {
            source = j;
            MPI_Recv(&fx, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
&status); /*Receives results of each process serially*/
            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) /*Compares starting data of master
process with process with id = 0*/
            {
                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
* p); /*Multiply with number of processes to get number of function
evalutations*/
        printf("Best result at trial %d used %d iterations, and
returned\n", best_trial, best_jj);
```

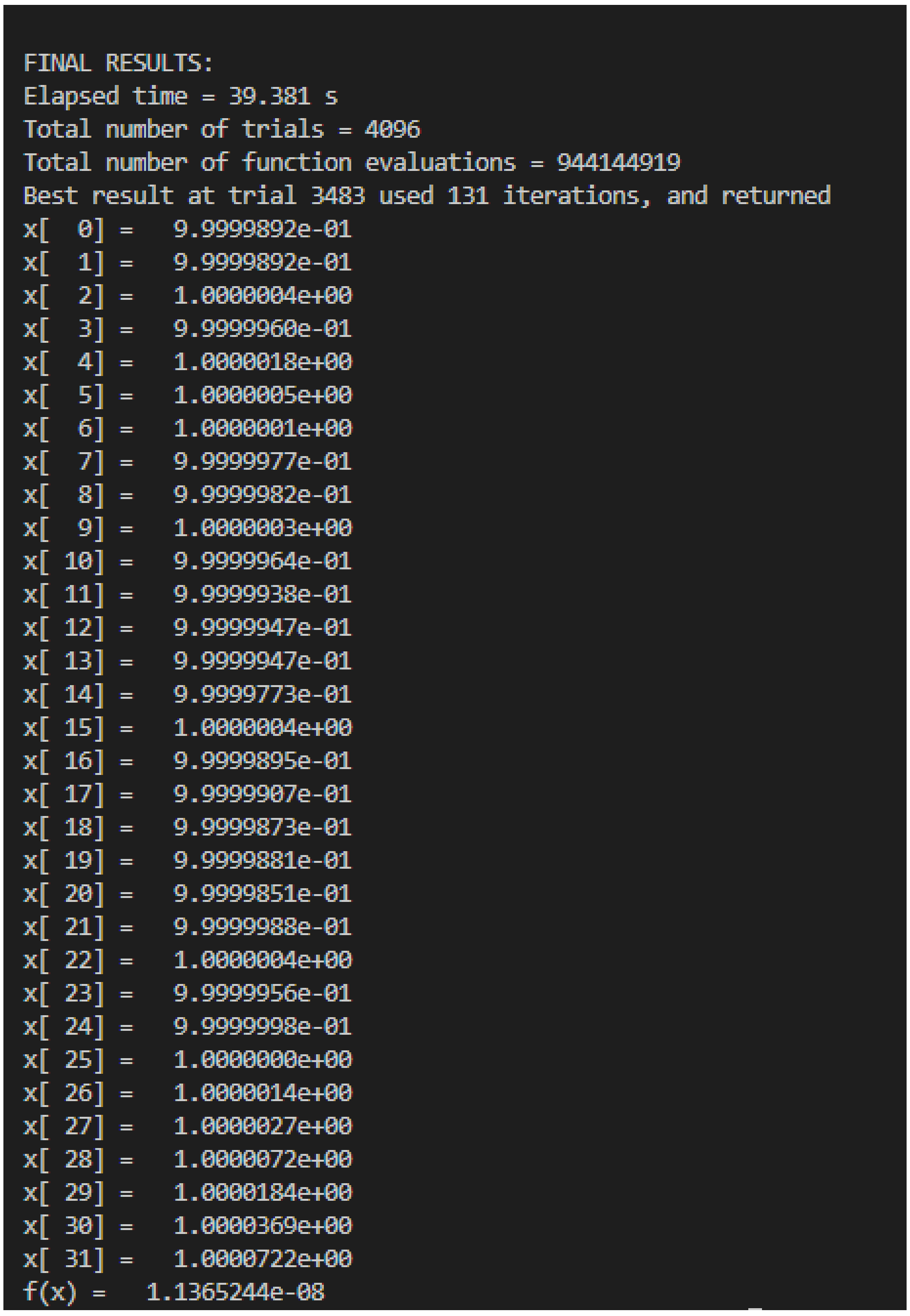
```
        for (i = 0; i < nvars; i++)
        {
            printf("x[%3d] = %15.7le \n", i, best_pt[i]);
        }
        printf("f(x) = %15.7le\n", best_fx);
    }

    MPI_Finalize();

    return 0;
}
```

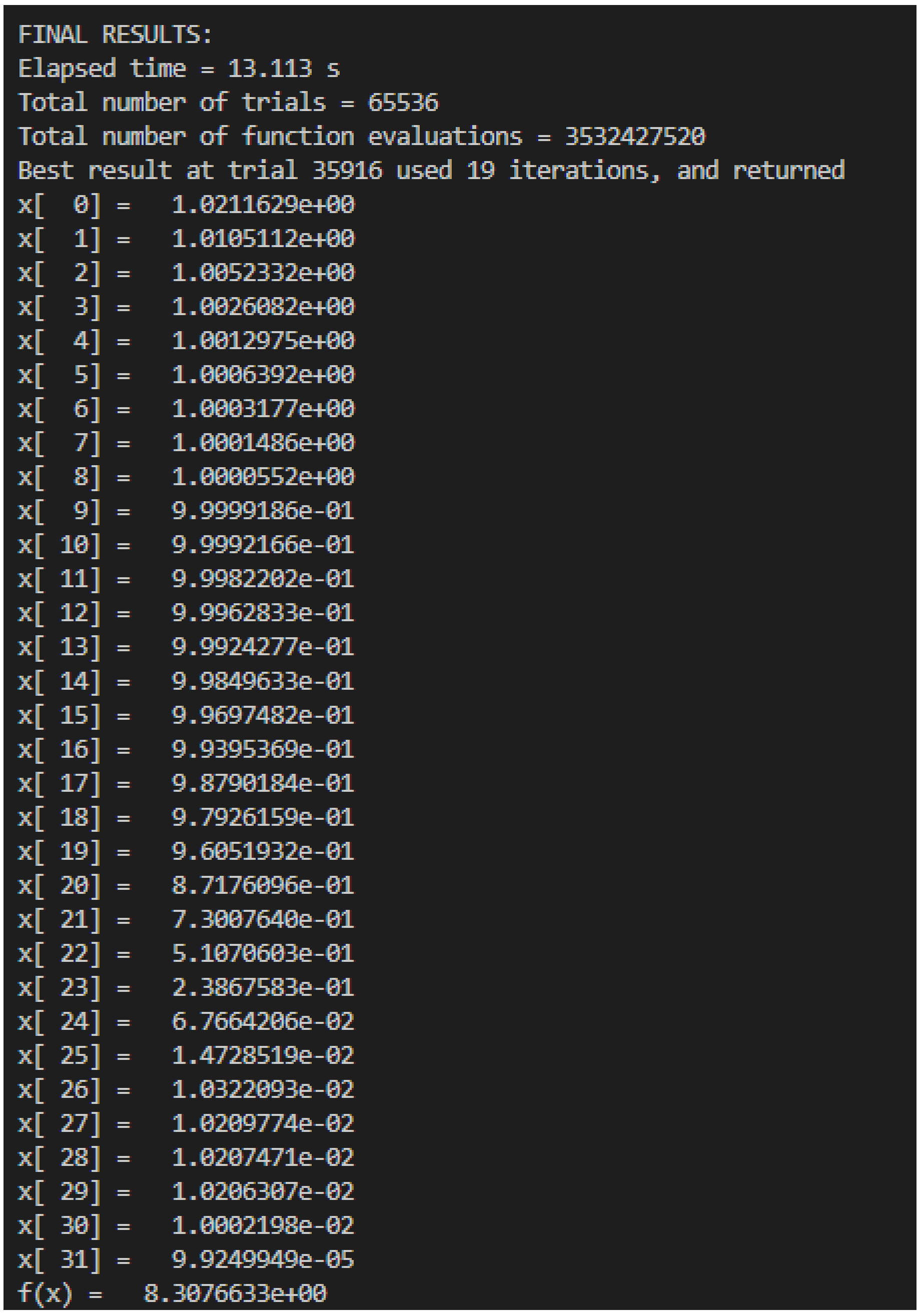
Αποτελέσματα

Αποτελέσματα αρχικού κώδικα:



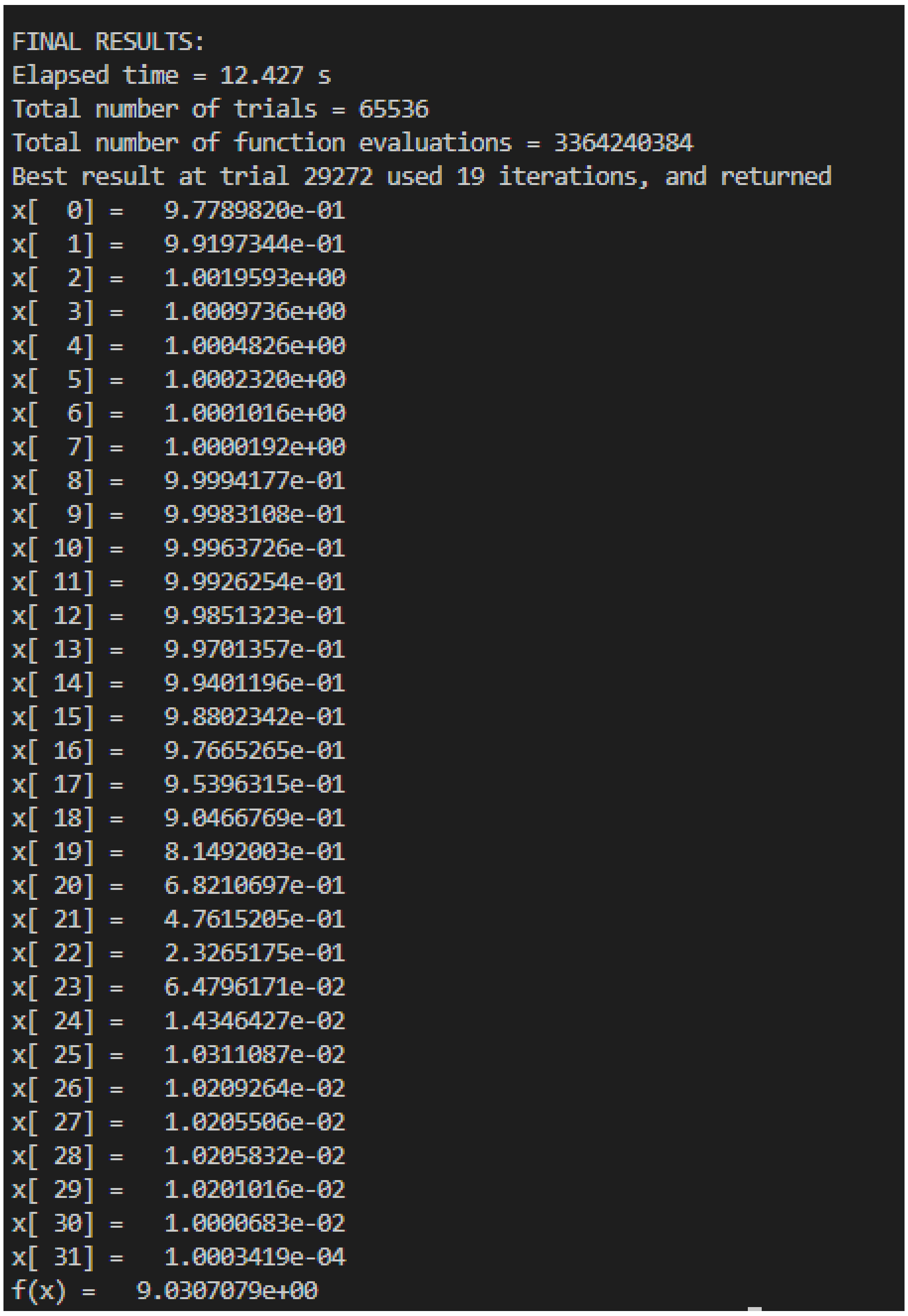
```
FINAL RESULTS:
Elapsed time = 39.381 s
Total number of trials = 4096
Total number of function evaluations = 944144919
Best result at trial 3483 used 131 iterations, and returned
x[  0] =   9.9999892e-01
x[  1] =   9.9999892e-01
x[  2] =   1.0000004e+00
x[  3] =   9.9999960e-01
x[  4] =   1.0000018e+00
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
```

Αποτελέσματα κώδικα με OpenMP:



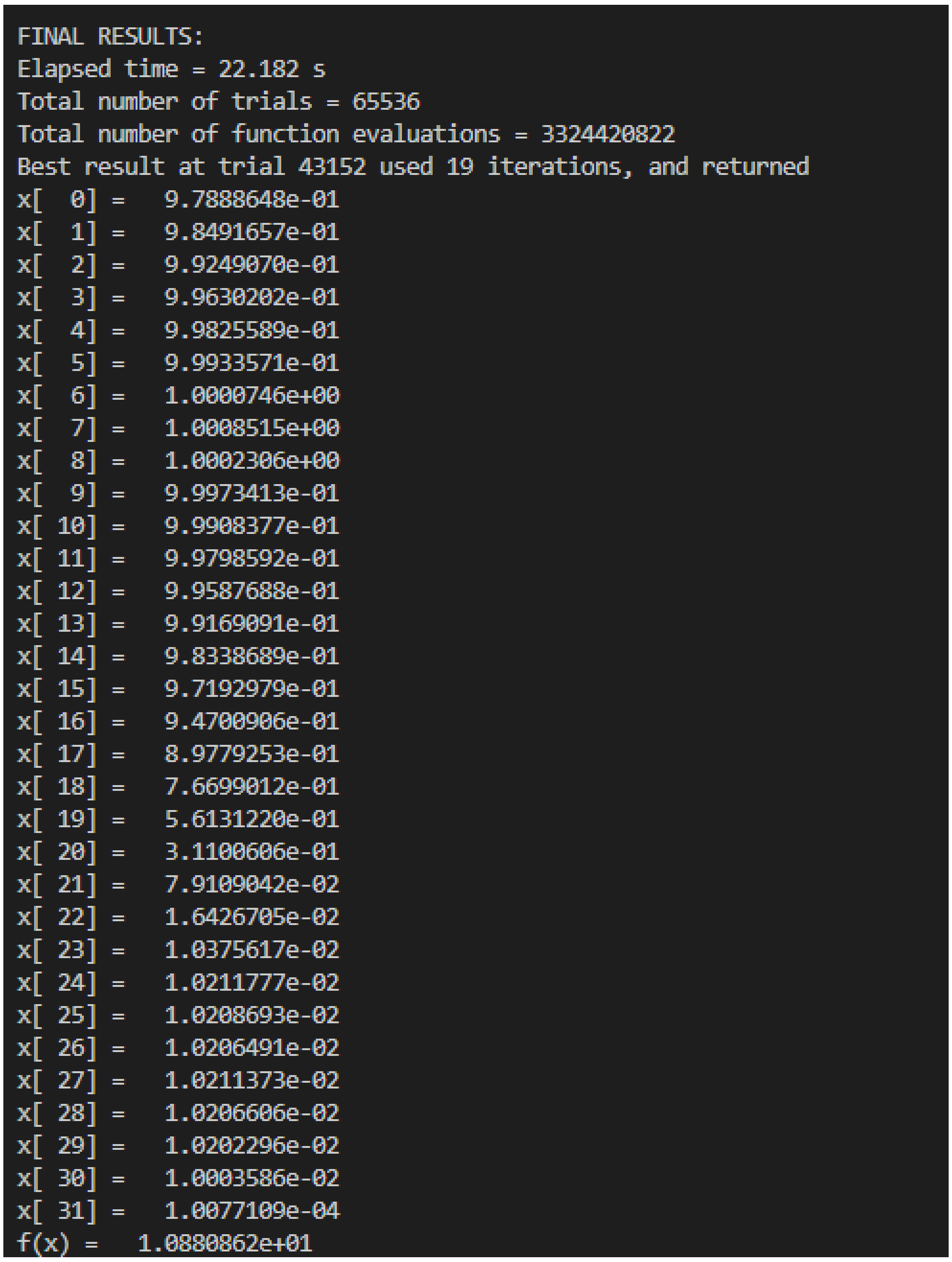
```
FINAL RESULTS:
Elapsed time = 13.113 s
Total number of trials = 65536
Total number of function evaluations = 3532427520
Best result at trial 35916 used 19 iterations, and returned
x[  0] =   1.0211629e+00
x[  1] =   1.0105112e+00
x[  2] =   1.0052332e+00
x[  3] =   1.0026082e+00
x[  4] =   1.0012975e+00
x[  5] =   1.0006392e+00
x[  6] =   1.0003177e+00
x[  7] =   1.0001486e+00
x[  8] =   1.0000552e+00
x[  9] =   9.9999186e-01
x[ 10] =   9.9992166e-01
x[ 11] =   9.9982202e-01
x[ 12] =   9.9962833e-01
x[ 13] =   9.9924277e-01
x[ 14] =   9.9849633e-01
x[ 15] =   9.9697482e-01
x[ 16] =   9.9395369e-01
x[ 17] =   9.8790184e-01
x[ 18] =   9.7926159e-01
x[ 19] =   9.6051932e-01
x[ 20] =   8.7176096e-01
x[ 21] =   7.3007640e-01
x[ 22] =   5.1070603e-01
x[ 23] =   2.3867583e-01
x[ 24] =   6.7664206e-02
x[ 25] =   1.4728519e-02
x[ 26] =   1.0322093e-02
x[ 27] =   1.0209774e-02
x[ 28] =   1.0207471e-02
x[ 29] =   1.0206307e-02
x[ 30] =   1.0002198e-02
x[ 31] =   9.9249949e-05
f(x) =   8.3076633e+00
```

Αποτελέσματα κώδικα OpenMP Tasks:



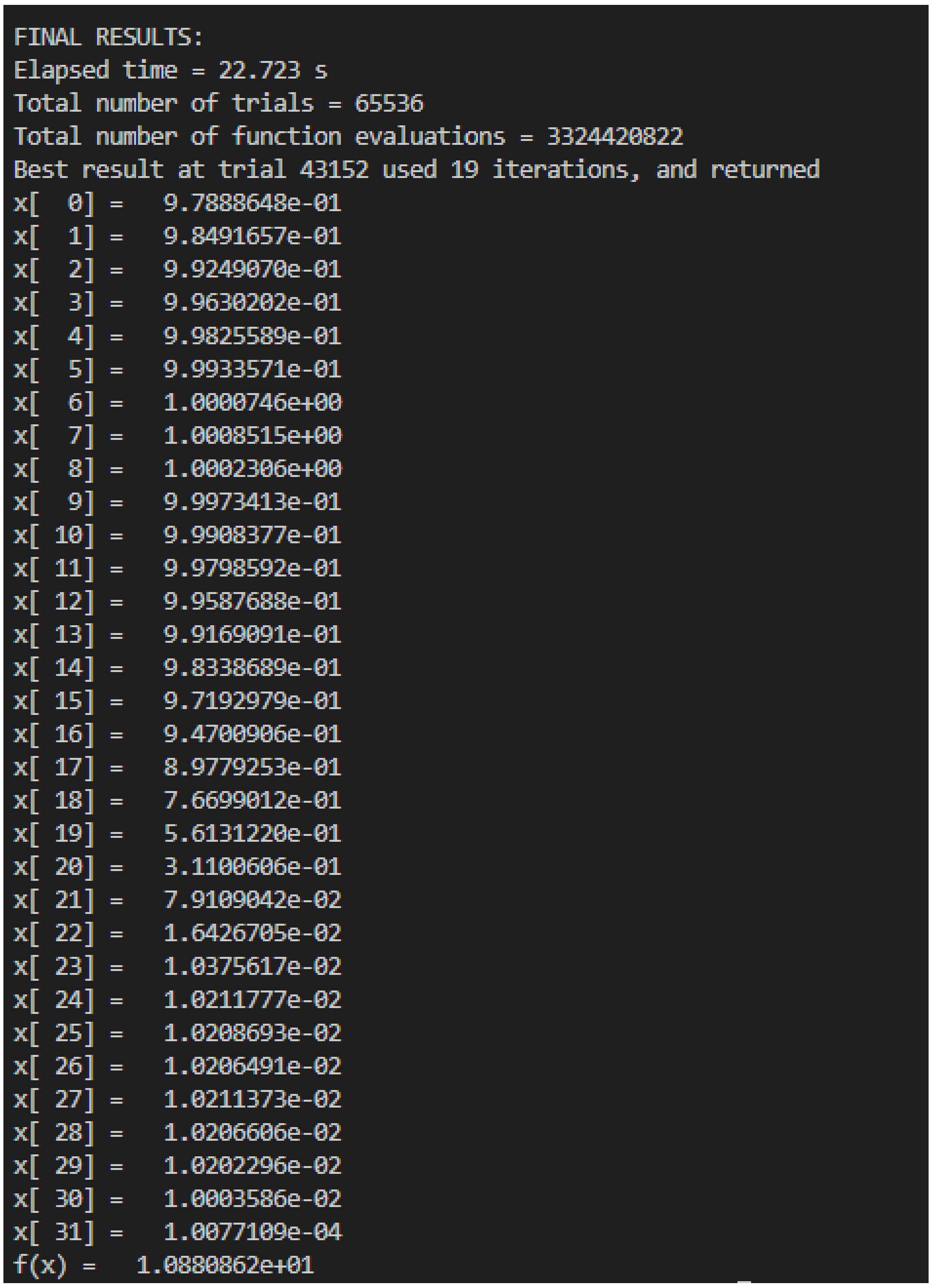
```
FINAL RESULTS:
Elapsed time = 12.427 s
Total number of trials = 65536
Total number of function evaluations = 3364240384
Best result at trial 29272 used 19 iterations, and returned
x[  0] =   9.7789820e-01
x[  1] =   9.9197344e-01
x[  2] =   1.0019593e+00
x[  3] =   1.0009736e+00
x[  4] =   1.0004826e+00
x[  5] =   1.0002320e+00
x[  6] =   1.0001016e+00
x[  7] =   1.0000192e+00
x[  8] =   9.9994177e-01
x[  9] =   9.9983108e-01
x[ 10] =   9.9963726e-01
x[ 11] =   9.9926254e-01
x[ 12] =   9.9851323e-01
x[ 13] =   9.9701357e-01
x[ 14] =   9.9401196e-01
x[ 15] =   9.8802342e-01
x[ 16] =   9.7665265e-01
x[ 17] =   9.5396315e-01
x[ 18] =   9.0466769e-01
x[ 19] =   8.1492003e-01
x[ 20] =   6.8210697e-01
x[ 21] =   4.7615205e-01
x[ 22] =   2.3265175e-01
x[ 23] =   6.4796171e-02
x[ 24] =   1.4346427e-02
x[ 25] =   1.0311087e-02
x[ 26] =   1.0209264e-02
x[ 27] =   1.0205506e-02
x[ 28] =   1.0205832e-02
x[ 29] =   1.0201016e-02
x[ 30] =   1.0000683e-02
x[ 31] =   1.0003419e-04
f(x) =   9.0307079e+00
```

Αποτελέσματα κώδικα MPI:



```
FINAL RESULTS:
Elapsed time = 22.182 s
Total number of trials = 65536
Total number of function evaluations = 3324420822
Best result at trial 43152 used 19 iterations, and returned
x[  0] =   9.7888648e-01
x[  1] =   9.8491657e-01
x[  2] =   9.9249070e-01
x[  3] =   9.9630202e-01
x[  4] =   9.9825589e-01
x[  5] =   9.9933571e-01
x[  6] =   1.0000746e+00
x[  7] =   1.0008515e+00
x[  8] =   1.0002306e+00
x[  9] =   9.9973413e-01
x[ 10] =   9.9908377e-01
x[ 11] =   9.9798592e-01
x[ 12] =   9.9587688e-01
x[ 13] =   9.9169091e-01
x[ 14] =   9.8338689e-01
x[ 15] =   9.7192979e-01
x[ 16] =   9.4700906e-01
x[ 17] =   8.9779253e-01
x[ 18] =   7.6699012e-01
x[ 19] =   5.6131220e-01
x[ 20] =   3.1100606e-01
x[ 21] =   7.9109042e-02
x[ 22] =   1.6426705e-02
x[ 23] =   1.0375617e-02
x[ 24] =   1.0211777e-02
x[ 25] =   1.0208693e-02
x[ 26] =   1.0206491e-02
x[ 27] =   1.0211373e-02
x[ 28] =   1.0206606e-02
x[ 29] =   1.0202296e-02
x[ 30] =   1.0003586e-02
x[ 31] =   1.0077109e-04
f(x) =   1.0880862e+01
```

Αποτελέσματα MPI + OpenMP:



```
FINAL RESULTS:
Elapsed time = 22.723 s
Total number of trials = 65536
Total number of function evaluations = 3324420822
Best result at trial 43152 used 19 iterations, and returned
x[  0] =   9.7888648e-01
x[  1] =   9.8491657e-01
x[  2] =   9.9249070e-01
x[  3] =   9.9630202e-01
x[  4] =   9.9825589e-01
x[  5] =   9.9933571e-01
x[  6] =   1.0000746e+00
x[  7] =   1.0008515e+00
x[  8] =   1.0002306e+00
x[  9] =   9.9973413e-01
x[ 10] =   9.9908377e-01
x[ 11] =   9.9798592e-01
x[ 12] =   9.9587688e-01
x[ 13] =   9.9169091e-01
x[ 14] =   9.8338689e-01
x[ 15] =   9.7192979e-01
x[ 16] =   9.4700906e-01
x[ 17] =   8.9779253e-01
x[ 18] =   7.6699012e-01
x[ 19] =   5.6131220e-01
x[ 20] =   3.1100606e-01
x[ 21] =   7.9109042e-02
x[ 22] =   1.6426705e-02
x[ 23] =   1.0375617e-02
x[ 24] =   1.0211777e-02
x[ 25] =   1.0208693e-02
x[ 26] =   1.0206491e-02
x[ 27] =   1.0211373e-02
x[ 28] =   1.0206606e-02
x[ 29] =   1.0202296e-02
x[ 30] =   1.0003586e-02
x[ 31] =   1.0077109e-04
f(x) =   1.0880862e+01
```

Συμπεράσματα

Αφού τρέξαμε τους κώδικες αρκετές φορές πήραμε τα τελικά αποτελέσματα που αναφέραμε πιο πάνω με τα αντίστοιχα screenshot. Παρατηρούμε ότι τα OpenMP και OpenMP Tasks έχουνε μεγάλη διαφορά (περίπου 10 δευτερολέπτων) στον χρόνο εκτέλεσης συγκριτικά με τα MPI και MPI + OpenMP. Η διαφορά αυτή δικαιολογείται με το ότι το OpenMP σε γενικές γραμμές είναι γρηγορότερο σε σχέση με το MPI. Βέβαια μπορεί η διαφορά αυτή να οφείλεται στο ότι ο κώδικας που υλοποιήσαμε να μην είναι και ο βέλτιστος. Καθώς επίσης και σε «προβλήματα» που θα μπορούσε να έχει ο επεξεργαστής του συστήματος που πήραμε τα αποτελέσματα.
Γενικά όμως μπορούμε να παρατηρήσουμε ότι οι παράλληλοι κώδικες εκτελούνται πολύ πιο γρήγορα σε σχέση με τον σειριακό κώδικα. Στις περιπτώσεις των OpenMP και OpenMP Tasks παρατηρούμε διαφορά περίπου 30 δευτερολέπτων. Ενώ στα MPI και MPI + OpenMP διαφορά περίπου 20 δευτερολέπτων.