Συστήματα Παράλληλης Επεξεργασίας

Άσκηση 2 - Παράλληλη επίλυση εξίσωσης θερμότητας

Χειμερινό Εξάμηνο 2020-2021

Ηλιάδης Θρασύβουλος - 03115761

Πέππας Αθανάσιος - 03115749

1. Διάδοση θερμότητας σε δύο διαστάσεις

Το πρόβλημα που προσπαθούμε να επιλύσουμε είναι το πρόβλημα της διάδοσης θερμότητας σε δύο διαστάσεις, και πιο συγκεκριμένα από το σύνορο προς το εσωτερικό μιας επιφάνειας. Οι δύο μέθοδοι που χρησιμοποιούνται εδώ είναι η μέθοδος Jacobi και η μέθοδος Gauss-Seidel με Successive Over-Relaxation. Πολύ συνοπτικά μία υλοποίηση για την κάθε μία ακολουθεί:

Jacobi:

```
for (t = 0; t < T && !converged; t++) {
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            U[t+1][i][j]=(1/4)*(U[t][i-1][j]+U[t][i][j-1]+U[t][i+1][j]+U[t][i]
[j+1]);
    converged=check_convergence(U[t+1],U[t])
}
```

Gauss-Seidel SOR

```
for (t = 0; t < T && !converged; t++) {
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            U[t+1][i][j]=U[t][i][j]+(omega/4)*(U[t+1][i-1][j]+U[t+1][i][j-1]+U[t]
[i+1][j]+U[t][i][j+1]-4*U[t][i][j]);
    converged=check_convergence(U[t+1],U[t])
}
```

Με λίγα λόγια εξηγούμε τι προσπαθούν να πετύχουν οι παραπάνω μέθοδοι. Και οι δύο μέθοδοι θεωρούν N στοιχεία πάνω στην επιφάνεια που μοντελοποιούν τη θερμοκρασία της περιοχής γύρω τους. Στη μέθοδο Jacobi κάθε χρονική στιγμή t ενημερώνουμε τη θερμοκρασία κάθε στοιχείου με βάση τις 4 θερμοκρασίες που είχαν τα γειτονικά σημεία (βόρειο, νότιο, ανατολικό, δυτικό) την *προηγούμενη* χρονική στιγμή. Συνεχίζουμε κατ' αυτό είτε μέχρι να ολοκληρωθούν συγκεκριμένα χρονικά βήματα, είτε μέχρι ένα ικανοποιητικό βαθμό σύγκλισης, οπότε θα έχουμε δηλαδή μικρή διαφορά ανάμεσα στις χρονικές στιγμές.

Η μέθοδος Gauss-Seidel είναι πολύ παρόμοια, με τη διαφορά ότι χρησιμοποιούμε για τους 4 γείτονες, 2 ενημερωμένες και 2 μη ενημερωμένες τιμές. Η παρατήρηση στην οποία βασίζεται το παραπάνω είναι πως άπαξ και φτάσουμε να υπολογίσουμε το σημείο x,y στον πίνακα, τότε ήδη έχουμε υπολογίσει στην ίδια επανάληψη τις *νέες* τιμές του από πάνω και από αριστερά γείτονα, και άρα μπορούμε να τις χρησιμοποιήσουμε από τώρα. Η διαφοροποίηση αυτή βοηθάει πολύ στην ταχύτητα σύγκλισης της προσομοίωσης, και άρα και στην γενική ταχύτητά της.

Jacobi

Στην εργασία αυτή προσπαθούμε να παραλληλοποιήσουμε τις παραπάνω μεθόδους. Για αυτό χωρίζουμε τον αρχικό πίνακα από N στοιχεία σε μικρούς υποπίνακες και τους μοιράζουμε σε διαφορετικούς επεξεργαστές. Το μόνο που χρειάζεται κάθε επεξεργαστής για να μπορεί να τρέχει ανεξάρτητα από τους υπόλοιπους είναι να ξέρει τις τιμές για την προηγούμενη χρονική στιγμή για τους γείτονες όλων των σημείων του. Πρέπει δηλαδή να υπάρχει μια επικοινωνία από τα στοιχεία στο βόρειο σύνορο του υποπίνακα με τα στοιχεία του πίνακα από πάνω, στο δυτικό με τα στοιχεία του πίνακα από αριστερά κ.ο.κ. Ο παρακάτω κώδικας δείχνει αυτή μας την προσπάθεια.

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

int main(int argc, char ** argv) {
    int rank,size;          //Rank --> id of mpi process, Size --> total number
of processes
    int global[2],local[2]; //global matrix dimensions and local matrix
dimensions (2D-domain, 2D-subdomain)
    int global_padded[2];   //padded global matrix dimensions (if padding is not
needed, global_padded=global)
    int grid[2];            //processor grid dimensions
    int i,j,t;
    int global_converged=0,converged=0; //flags for convergence, global and per
process
    MPI_Datatype dummy;     //dummy datatype used to align user-defined datatypes
in memory
    //double omega;                     //relaxation factor - useless for Jacobi

    struct timeval tts,ttf,tcs,tcf,tcvs,tcvf;   //Timers: total-> tts,ttf,
computation -> tcs,tcf
    double ttotal=0,tcomp=0,tconv=0,total_time,comp_time,conv_time;

    double ** U, ** u_current, ** u_previous, ** swap; //Global matrix, local
current and previous matrices, pointer to swap between current and previous
    double * U_start;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Status status;
    //----Read 2D-domain dimensions and process grid dimensions from stdin----//

    if (argc!=5) {
        fprintf(stderr,"Usage: mpirun .... ./exec X Y Px Py");
        exit(-1);
    }
    else {
        global[0]=atoi(argv[1]);
        global[1]=atoi(argv[2]);
        grid[0]=atoi(argv[3]);
        grid[1]=atoi(argv[4]);
```

```
    }

    //----Create 2D-cartesian communicator----//
        //----Usage of the cartesian communicator is optional----//

    MPI_Comm CART_COMM;         //CART_COMM: the new 2D-cartesian communicator
    int periods[2]={0,0};       //periods={0,0}: the 2D-grid is non-periodic
    int rank_grid[2];           //rank_grid: the position of each process on the
new communicator

    MPI_Cart_create(MPI_COMM_WORLD,2,grid,periods,0,&CART_COMM);
 //communicator creation
    MPI_Cart_coords(CART_COMM,rank,2,rank_grid);                        //rank
mapping on the new communicator


    //----Compute local 2D-subdomain dimensions----//
    //----Test if the 2D-domain can be equally distributed to all processes----//
    //----If not, pad 2D-domain----//

    for (i=0;i<2;i++) {
        if (global[i]%grid[i]==0) {
            local[i]=global[i]/grid[i];
            global_padded[i]=global[i];
        }
        else {
            local[i]=(global[i]/grid[i])+1;
            global_padded[i]=local[i]*grid[i];
        }
    }

        //Initialization of omega
    //omega=2.0/(1+sin(3.14/global[0]));

    //----Allocate global 2D-domain and initialize boundary values----//
    //----Rank 0 holds the global 2D-domain----//
    if (rank==0) {
        U=allocate2d(global_padded[0],global_padded[1]);
        init2d(U,global[0],global[1]);
    }

    //----Allocate local 2D-subdomains u_current, u_previous----//
    //----Add a row/column on each size for ghost cells----//

    u_previous=allocate2d(local[0]+2,local[1]+2);
    u_current=allocate2d(local[0]+2,local[1]+2);
    //----Distribute global 2D-domain from rank 0 to all processes----//

        //----Appropriate datatypes are defined here----//
        /*****The usage of datatypes is optional*****/

    //----Datatype definition for the 2D-subdomain on the global matrix----//

    MPI_Datatype global_block;
    MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
    MPI_Type_commit(&global_block);
```

```
    //----Datatype definition for the 2D-subdomain on the local matrix----//

    MPI_Datatype local_block;
    MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
    MPI_Type_commit(&local_block);

    //----Rank 0 defines positions and counts of local blocks (2D-subdomains) on
global matrix----//
    int * scatteroffset, * scattercounts;
    if (rank==0) {
        U_start = &(U[0][0]);
        scattercounts=(int*)malloc(size*sizeof(int));
        scatteroffset=(int*)malloc(size*sizeof(int));
        for (i=0;i<grid[0];i++)
            for (j=0;j<grid[1];j++) {
                scattercounts[i*grid[1]+j]=1;
                scatteroffset[i*grid[1]+j]=
(local[0]*local[1]*grid[1]*i+local[1]*j);
            }
    }

    printf("asdfasdfasdf\n");

    //----Rank 0 scatters the global matrix----//

    //----Rank 0 scatters the global matrix----//

        //************TODO*****************//

    /*Fill your code here*/

    MPI_Scatterv(U_start, scattercounts, scatteroffset, global_block, &
(u_current[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    MPI_Scatterv(U_start, scattercounts, scatteroffset, global_block, &
(u_previous[1][1]), 1, local_block, 0, MPI_COMM_WORLD);

    //*********************************//

    if (rank==0)
        free2d(U);

        //----Define datatypes or allocate buffers for message passing----//

        //************TODO*****************//

        /*Fill your code here*/

    MPI_Datatype col;
    MPI_Type_vector(local[0],1,local[1]+2,MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&col);
    MPI_Type_commit(&col);
        //*********************************//

    //----Find the 4 neighbors with which a process exchanges messages----//

        //************TODO*****************//
```

```
        /*Fill your code here*/

    int north, south, east, west;
    int x=1, y=0;
    MPI_Cart_shift(CART_COMM,x,1,&west,&east);
    MPI_Cart_shift(CART_COMM,y,1,&north,&south);

        /*Make sure you handle non-existing
                neighbors appropriately*/

        //***********************************//

    //---Define the iteration ranges per process-----//

    int i_min,i_max,j_min,j_max;

        /*Fill your code here*/

    i_min = 1;
    i_max = local[0] + 1;

    /* boundary process - no possible padding */
    if (north == MPI_PROC_NULL) {
        i_min = 2;  // ghost cell + boundary
    }

    /* boundary process and padded global array */
    if (south == MPI_PROC_NULL){
        i_max -= (global_padded[0] - global[0]) + 1;
    }

    /* internal process (ghost cell only) */
    j_min = 1;
    j_max = local[1] + 1;

    /* boundary process - no possible padding */
    if (west == MPI_PROC_NULL) {
        j_min = 2;  //ghost cell + boundary
    }

    /* boundary process and padded global array */
    if (east == MPI_PROC_NULL){
        j_max -= (global_padded[1] - global[1]) + 1;
    }

        /*Three types of ranges:
                -internal processes
                -boundary processes
                -boundary processes and padded global array
        */

        //***********************************//

        //----Computational core----//
        gettimeofday(&tts, NULL);
#ifdef TEST_CONV
    for (t=0;t<T && !global_converged;t++) {
#endif
```

```
#ifndef TEST_CONV
#undef T
#define T 256
    for (t=0;t<T;t++) {
#endif

                /*Fill your code here*/
                /*Compute and Communicate*/
        swap=u_previous;
        u_previous=u_current;
        u_current=swap;

        if (north != MPI_PROC_NULL){
            MPI_Sendrecv(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0,
&u_previous[0][1],
                            local[1], MPI_DOUBLE, north, 0, MPI_COMM_WORLD,
&status );
        }

                // Communicate with south
        if (south != MPI_PROC_NULL){
            MPI_Sendrecv(&u_previous[local[0]][1], local[1], MPI_DOUBLE, south,
0, &u_previous[local[0]+1][1],
                            local[1], MPI_DOUBLE, south, 0, MPI_COMM_WORLD,
&status );
        }

                // Communicate with east
        if (east != MPI_PROC_NULL){
            MPI_Sendrecv(&u_previous[1][local[1]], 1, col, east, 0,
&u_previous[1][local[1]+1],
                             1, col, east, 0, MPI_COMM_WORLD, &status );
        }

                // Communicate with west
        if (west != MPI_PROC_NULL){
            MPI_Sendrecv(&u_previous[1][1], 1, col, west, 0, &u_previous[1][0],
                             1, col, west, 0, MPI_COMM_WORLD, &status );
        }


                /*Add appropriate timers for computation*/
        gettimeofday(&tcs, NULL);

        for (i=i_min;i<i_max;i++)
                for (j=j_min;j<j_max;j++)
                        u_current[i][j]=(u_previous[i-1][j]+u_previous[i+1]
[j]+u_previous[i][j-1]+u_previous[i][j+1])/4.0;

        gettimeofday(&tcf, NULL);

        tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;


#ifdef TEST_CONV
        if (t%C==0) {
                        /*Test convergence*/
                gettimeofday(&tcvs, NULL);
```

```
                converged=converge(u_previous,u_current,local[0],local[1]);

                MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN,
MPI_COMM_WORLD);
                gettimeofday(&tcvf, NULL);
                tconv += (tcvf.tv_sec-tcvs.tv_sec)+(tcvf.tv_usec-
tcvs.tv_usec)*0.000001;
        }
#endif

    }
    gettimeofday(&ttf,NULL);

    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;

    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

    //----Rank 0 gathers local matrices back to the global matrix----//
    if (rank==0) {
            U=allocate2d(global_padded[0],global_padded[1]);
    }

    MPI_Gatherv(&u_current[1][1], 1, local_block, U_start, scattercounts,
scatteroffset, global_block, 0, MPI_COMM_WORLD);

        //----Printing results----//
    if (rank==0) {
        printf("Jacobi X %d Y %d Px %d Py %d Iter %d ComputationTime %lf
TotalTime %lf midpoint
%lf\n",global[0],global[1],grid[0],grid[1],t,comp_time,total_time,U[global[0]/2]
[global[1]/2]);

        #ifdef PRINT_RESULTS
        char * s=malloc(50*sizeof(char));

 sprintf(s,"resJacobiMPI_%dx%d_%dx%d",global[0],global[1],grid[0],grid[1]);
        fprint2d(s,U,global[0],global[1]);
        free(s);
        #endif

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

Gauss-Seidel SOR

Στην υλοποίηση της μεθόδου Jacobi αρκούσε *πριν* την εκτέλεση των υπολογισμών κάθε υποπίνακας του global αρχικού U να ανταλλάξει με τους γειτονικούς πίνακες τα κατάλληλα στοιχεία (δηλαδή τα cells που συνορεύουν με τον κάθε υποπίνακα).

Στη μέθοδο Gauss-Seidel SOR υπάρχουν διαφορές. Κάθε block χρειάζεται να λάβει *πριν* την εκτέλεση των υπολογισμών τις κατάλληλες τιμές για τους γείτονές του, με τη διαφορά ότι χρειάζεται την τιμή της *τωρινής* χρονικής στιγμής από τον πάνω και τον αριστερά γείτονα και της *προηγούμενης* χρονικής στιγμής από τον κάτω και τον δεξιά.

Επίσης το κάθε block χρειάζεται να στέλνει δεδομένα στους γείτονές του. Για να λειτουργεί ο αλγόριθμος πρέπει το block να στέλνει τόσο πριν όσο και μετά τον υπολογισμό στους κατάλληλους γείτονες. Συγκεκριμένα θα πρέπει πριν εκτελέσει τους υπολογισμούς να στείλει τις τιμές που έχει στον πάνω και τον αριστερά γείτονα (στέλνοντας τους δηλαδή τις τιμές της *προηγούμενης* χρονικής στιγμής), και μετά την εκτέλεση να στείλει τα νέα δεδομένα κάτω και δεξιά (την *τωρινή* δηλαδή χρονική στιγμή).

Αν αυτό ακούγεται λίγο μπερδεμένο, είναι επειδή είναι :). Ελπίζουμε με τον κώδικα να ξεκαθαρίσουμε λίγο τα πράγματα.

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



int main(int argc, char ** argv) {
    int rank,size;
    int global[2],local[2]; //global matrix dimensions and local matrix
dimensions (2D-domain, 2D-subdomain)
    int global_padded[2];   //padded global matrix dimensions (if padding is not
needed, global_padded=global)
    int grid[2];            //processor grid dimensions
    int i,j,t;
    int global_converged=0,converged=0; //flags for convergence, global and per
process
    MPI_Datatype dummy;     //dummy datatype used to align user-defined datatypes
in memory
    double omega;                       //relaxation factor - useless for Jacobi

    struct timeval tts,ttf,tcs,tcf,tcvs,tcvf;   //Timers: total-> tts,ttf,
computation -> tcs,tcf
    double ttotal=0,tcomp=0,tconv=0,total_time,comp_time,conv_time;

    double ** U, ** u_current, ** u_previous, ** swap; //Global matrix, local
current and previous matrices, pointer to swap between current and previous
    double *U_start;


    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    //----Read 2D-domain dimensions and process grid dimensions from stdin----//

    if (argc!=5) {
        fprintf(stderr,"Usage: mpirun .... ./exec X Y Px Py");
        exit(-1);
    }
    else {
```

```
        global[0]=atoi(argv[1]);
        global[1]=atoi(argv[2]);
        grid[0]=atoi(argv[3]);
        grid[1]=atoi(argv[4]);
    }

    //----Create 2D-cartesian communicator----//
        //----Usage of the cartesian communicator is optional----//

    MPI_Comm CART_COMM;         //CART_COMM: the new 2D-cartesian communicator
    int periods[2]={0,0};       //periods={0,0}: the 2D-grid is non-periodic
    int rank_grid[2];           //rank_grid: the position of each process on the
new communicator

    MPI_Cart_create(MPI_COMM_WORLD,2,grid,periods,0,&CART_COMM);
 //communicator creation
    MPI_Cart_coords(CART_COMM,rank,2,rank_grid);                        //rank
mapping on the new communicator

    //----Compute local 2D-subdomain dimensions----//
    //----Test if the 2D-domain can be equally distributed to all processes----//
    //----If not, pad 2D-domain----//

    for (i=0;i<2;i++) {
        if (global[i]%grid[i]==0) {
            local[i]=global[i]/grid[i];
            global_padded[i]=global[i];
        }
        else {
            local[i]=(global[i]/grid[i])+1;
            global_padded[i]=local[i]*grid[i];
        }
    }

        //Initialization of omega
    omega=2.0/(1+sin(3.14/global[0]));

    //----Allocate global 2D-domain and initialize boundary values----//
    //----Rank 0 holds the global 2D-domain----//
    if (rank==0) {
        U=allocate2d(global_padded[0],global_padded[1]);
        init2d(U,global[0],global[1]);
    }

    //----Allocate local 2D-subdomains u_current, u_previous----//
    //----Add a row/column on each size for ghost cells----//

    u_previous=allocate2d(local[0]+2,local[1]+2);
    u_current=allocate2d(local[0]+2,local[1]+2);

    //----Distribute global 2D-domain from rank 0 to all processes----//

    //----Datatype definition for the 2D-subdomain on the global matrix----//

    MPI_Datatype global_block;
    MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
    MPI_Type_commit(&global_block);
```

```
    //----Datatype definition for the 2D-subdomain on the local matrix----//

    MPI_Datatype local_block;
    MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
    MPI_Type_commit(&local_block);

    //----Datatype definition for a column in the local block----//

    MPI_Datatype col;
    MPI_Type_vector(local[0],1,local[1]+2,MPI_DOUBLE,&dummy);
    MPI_Type_create_resized(dummy,0,sizeof(double),&col);
    MPI_Type_commit(&col);

    //----Rank 0 defines positions and counts of local blocks (2D-subdomains) on
global matrix----//
    int * scatteroffset, * scattercounts;
    if (rank==0) {
        U_start = &(U[0][0]);
        scatteroffset=(int*)malloc(size*sizeof(int));
        scattercounts=(int*)malloc(size*sizeof(int));
        for (i=0;i<grid[0];i++)
            for (j=0;j<grid[1];j++) {
                scattercounts[i*grid[1]+j]=1;
                scatteroffset[i*grid[1]+j]=
(local[0]*local[1]*grid[1]*i+local[1]*j);
            }
    }

    //----Rank 0 scatters the global matrix----//
    MPI_Scatterv(U_start, scattercounts, scatteroffset, global_block, &
(u_previous[1][1]), 1, local_block, 0, MPI_COMM_WORLD);
    MPI_Scatterv(U_start, scattercounts, scatteroffset, global_block, &
(u_current[1][1]), 1, local_block, 0, MPI_COMM_WORLD);


    if (rank==0)
        free2d(U);
        //************************************//
    //----Find the 4 neighbors with which a process exchanges messages----//
        //*************TODO******************//
    int north, south, east, west;
    int x=1, y=0;
    MPI_Cart_shift(CART_COMM,x,1,&west,&east);
    MPI_Cart_shift(CART_COMM,y,1,&north,&south);

        /*Make sure you handle non-existing
                neighbors appropriately*/
        //************************************//
    //---Define the iteration ranges per process-----//

    int i_min,i_max,j_min,j_max;

    i_min = 1;
    i_max = local[0] + 1;

    if (north == MPI_PROC_NULL)
```

```
        i_min = 2;

    if (south == MPI_PROC_NULL)
        i_max -= (global_padded[0] - global[0]) + 1;

    j_min = 1;
    j_max = local[1] + 1;

    if (west == MPI_PROC_NULL)
        j_min = 2;

    if (east == MPI_PROC_NULL)
        j_max -= (global_padded[1] - global[1]) + 1;

        /*Three types of ranges:
                -internal processes
                -boundary processes
                -boundary processes and padded global array
        */
        //----Computational core----//

    MPI_Request before_request[6];
    MPI_Status before_status[6];
    MPI_Request after_request[2];
    MPI_Status after_status[2];

    int len_before_request = 0;
    int len_after_request = 0;
    gettimeofday(&tts, NULL);
    #ifdef TEST_CONV
    for (t=0;t<T && !global_converged;t++) {
    #endif
    #ifndef TEST_CONV
    #undef T
    #define T 256
    for (t=0;t<T;t++) {
    #endif

        len_before_request = 0;
        len_after_request = 0;

        swap = u_previous;
        u_previous = u_current;
        u_current = swap;

        /*
         * We need to send data to north and west, and get data from south and
east
         */

        /* Great White North */
         if (north != MPI_PROC_NULL){
            MPI_Isend(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0,
MPI_COMM_WORLD, &before_request[len_before_request]);
            MPI_Irecv(&u_current[0][1], local[1], MPI_DOUBLE, north, 0,
MPI_COMM_WORLD, &before_request[len_before_request + 1]);
            len_before_request += 2;
        }
```

```
        /* The West is the Best (?) */
        if (west != MPI_PROC_NULL){
            MPI_Isend(&u_previous[1][1], 1, col, west, 0, MPI_COMM_WORLD,
&before_request[len_before_request]);
            MPI_Irecv(&u_current[1][0], 1, col, west, 0, MPI_COMM_WORLD,
&before_request[len_before_request + 1]);
            len_before_request += 2;
        }


        /* Get data from south */
        if (south != MPI_PROC_NULL){
            MPI_Irecv(&u_previous[local[0]+1][1], local[1], MPI_DOUBLE, south, 0,
MPI_COMM_WORLD, &before_request[len_before_request]);
            len_before_request++;
        }
        /* Yeast */
        if (east != MPI_PROC_NULL){
            MPI_Irecv(&u_previous[1][local[1]+1], 1, col, east, 0,
MPI_COMM_WORLD, &before_request[len_before_request]);
            len_before_request++;
        }


        MPI_Waitall(len_before_request, before_request, before_status);


                /*Compute and Communicate*/

                /*Add appropriate timers for computation*/

        gettimeofday(&tcs, NULL);

        for (i=i_min;i<i_max;i++)
                for (j=j_min;j<j_max;j++)
                        u_current[i][j]=u_previous[i][j] + (omega/4.0)*
(u_current[i-1][j]+u_previous[i+1][j]+u_current[i][j-1]+u_previous[i]
[j+1]-4*u_previous[i][j]);

        gettimeofday(&tcf, NULL);

        tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;


        /* *SEND* to south */
        if (south != MPI_PROC_NULL){
            MPI_Isend(&u_current[local[0]][1], local[1], MPI_DOUBLE, south, 0,
MPI_COMM_WORLD, &after_request[len_after_request]);
            len_after_request++;
        }

        /* *SEND* data to east */
        if (east != MPI_PROC_NULL){
            MPI_Isend(&u_current[1][local[1]], 1, col, east, 0, MPI_COMM_WORLD,
&after_request[len_after_request]);
            len_after_request++;
        }
```

```
        MPI_Waitall(len_after_request, after_request, after_status);

#ifdef TEST_CONV
        if (t%C==0) {
            gettimeofday(&tcvs, NULL);
            converged=converge(u_previous,u_current,local[0],local[1]);

            MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN,
MPI_COMM_WORLD);
            gettimeofday(&tcvf, NULL);
            tconv += (tcvf.tv_sec-tcvs.tv_sec)+(tcvf.tv_usec-
tcvs.tv_usec)*0.000001;
        }
#endif
    }
    gettimeofday(&ttf,NULL);

    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;

    MPI_Reduce(&ttotal, &total_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&tcomp, &comp_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&tconv, &conv_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);


    //----Rank 0 gathers local matrices back to the global matrix----//

    if (rank==0) {
            U=allocate2d(global_padded[0],global_padded[1]);
            U_start = &(U[0][0]);
    }
        //*************TODO******************//
    MPI_Gatherv(&u_current[1][1], 1, local_block, U_start, scattercounts,
scatteroffset, global_block, 0, MPI_COMM_WORLD);
     //************************************//
        //----Printing results----//

    if (rank==0) {
        /*printf("MPI - GaussSeidelSOR X %d Y %d Px %d Py %d Iter %d
ComputationTime %lf TotalTime %lf midpoint
%lf\n",global[0],global[1],grid[0],grid[1],t,comp_time,total_time,U[global[0]/2]
[global[1]/2]);
        */
        printf("Jacobi X %d Y %d Px %d Py %d Iter %d ComputationTime %lf
TotalTime %lf Convergence Time %lf  midpoint
%lf\n",global[0],global[1],grid[0],grid[1],t,comp_time,total_time,conv_time,
U[global[0]/2][global[1]/2]);
        #ifdef PRINT_RESULTS
        char * s=malloc(50*sizeof(char));
        sprintf(s,"MPI-
resGaussSeidelSOR_%dx%d_%dx%d",global[0],global[1],grid[0],grid[1]);
        fprint2d(s,U,global[0],global[1]);
        free(s);
        #endif

    }
    MPI_Finalize();
    return 0;
```

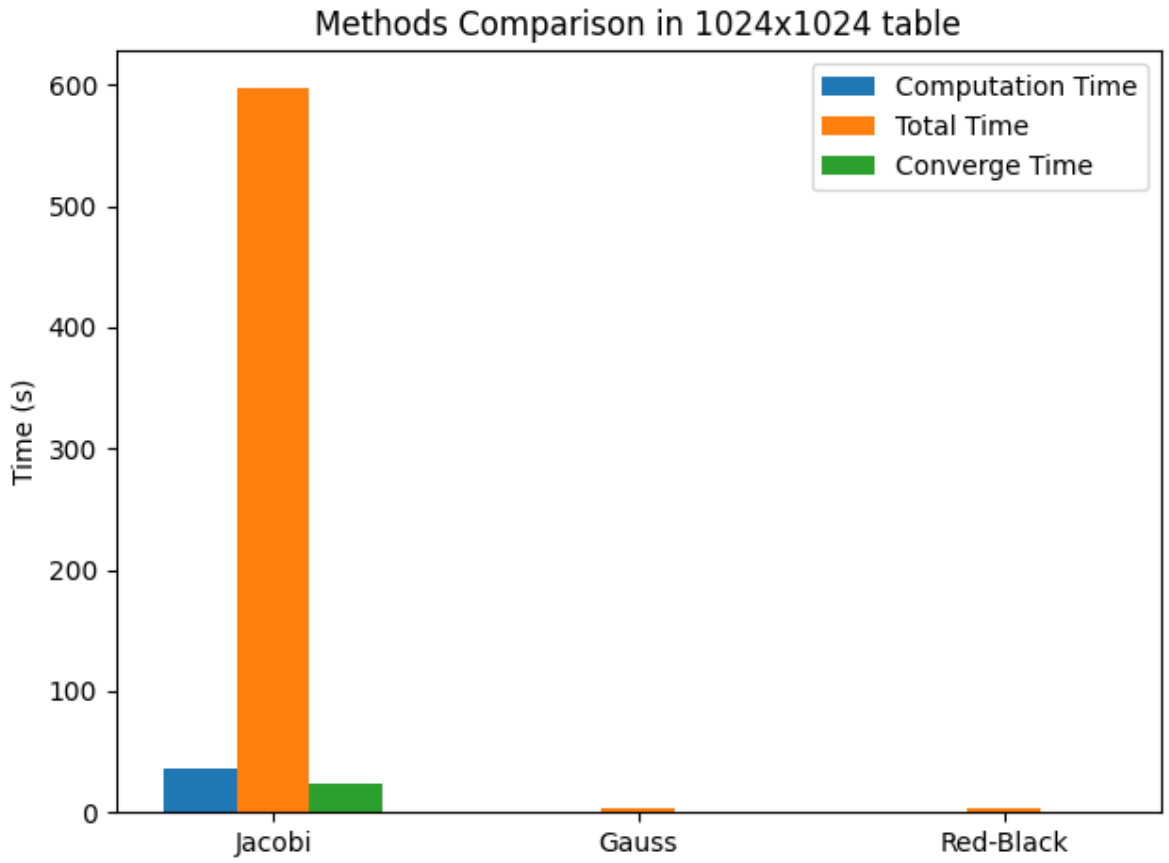
```
}
```

Μετρήσεις

1. Μετρήσεις με έλεγχο σύγκλισης

Σε αυτό το σενάριο τρέξαμε τους τρεις αλγορίθμους με ενεργοποιημένο τον έλεγχο σύγκλισης, με 64 MPI διεργασίες και τετραγωνικό μέγεθος πίνακα 1024.

Οι χρόνοι φαίνονται παρακάτω

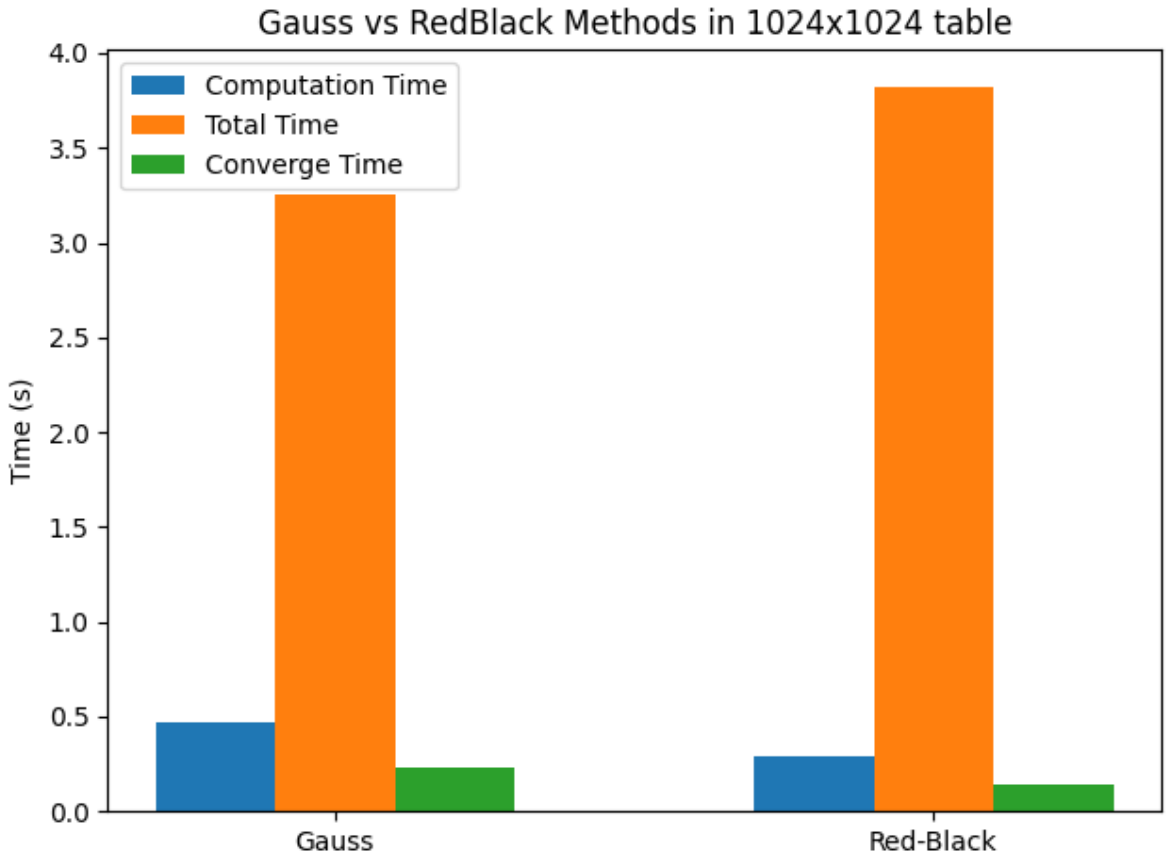


Είναι προφανές πώς ο αλγόριθμος Jacobi είναι κατα πολύ πιο χρονοβόρος από τους άλλους δύο. Σε σημείο μάλιστα που σε κοινή κλίμακα οι χρόνοι των μεθόδων GaussSor και RedBlackSor δεν διακρίνονται.

Χαρακτηριστικό είναι το γεγονός πως ο χρόνος σύγκλισης του Jacobi είναι μεγαλύτερος από τους συνολικούς χρόνους εκτέλεσης των άλλων δυο αλγορίθμων. Συνεπώς σε καμία περίπτωση κάποιος δε θα επιλέξει τον αλγόριθμο Jacobi για την εκτέλεση της συγκεκριμένης εφαρμογής σε περιβάλλον κατανεμημένης μνήμης.

Τώρα μένει να συγκρίνουμε τους χρόνους των GaussSor και RedBlackSor ώστε να βγάλουμε ενας ασφαλές συμπέρασμα για τον καλύτερο αλγόριθμο.

Εποπτικά έχουμε



Εδώ διακρίνουμε κάπως ανάμεικτα αποτελέσματα. Από τη μία ο RedBlackSor έχει πιο μικρούς χρόνους σύγκλισης και υπολογισμού αλλά το GaussSor έχει μικρότερο συνολικό χρόνο εκτέλεσης. Επειδή εν γένει αυτό που θέλουμε είναι ο αλγοριθμός μας να έχει οσο γίνεται μικρότερο χρόνο εκτέλεσης ο GaussSor υπερτερεί έναντι του RedBlackSor και συνεπώς είναι ο καταλληλότερος αλγόριθμος για αρχιτεκτονικές κατανεμημένης μνήμης.

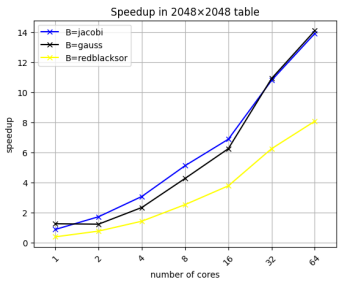
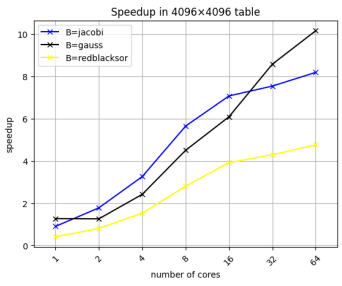
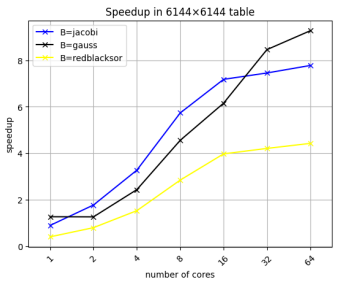
2. Μετρήσεις χωρίς έλεγχο σύγκλισης

Σε αυτό το σενάριο, απενεργοποιήσαμε τον έλεγχο σύγκλισης ώστε να βγάλουμε ασφαλή συμπεράσματα για την επιτάχυνση.

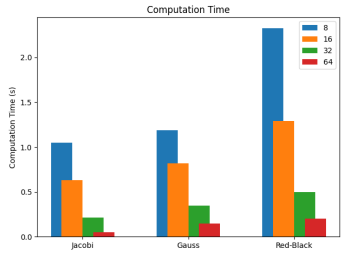
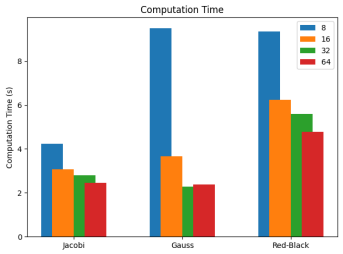
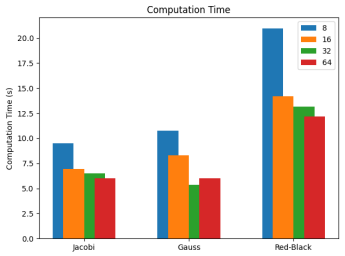
Τρέξαμε τους τρεις αλγορίθμους για σταθερό T=256, 1,2,4,8,16,32,64 MPI διεργασίες και για τετραγωνικούς πίνακες 2048, 4096, 6144

Τα αποτελέσματά μας είναι τα εξης

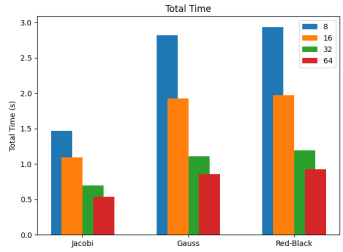
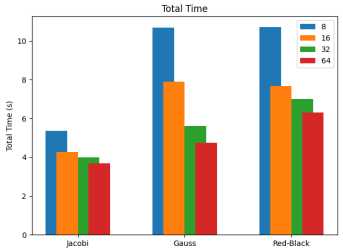
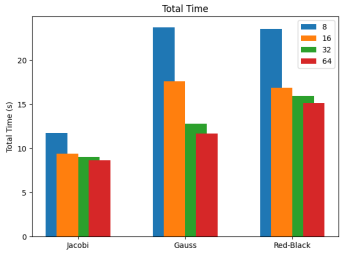
Speedup

2048x2048	4096x4096	6144x6144
		

Computation Time

2048x2048	4096x4096	6144x6144
		

Total Time

2048x2048	4096x4096	6144x6144
		

Από τα παραπάνω έχουμε τις εξής παρατηρήσεις

- Για σταθερό αριθμό επαναλήψεων ο Jacobi είναι πιο γρήγορος διότι εχει λιγότερο κόστος επικοινωνίας από τον Gauss και λιγότερο υπολογιστικό κόστος από τον RedBlack
- Ο αλγόριθμος που κλιμακώνει καλύτερα είναι ο Gauss.