ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

Άσκηση 2: Παράλληλη επίλυση εξίσωσης θερμότητας

Χειμερινό εξάμηνο 2019-20 - Ροή Υ

Αντωνιάδης, Παναγιώτης
el15009@central.ntua.gr

Μπαζώτης, Νικόλαος
el15739@central.ntua.gr

"Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all."

– Herb Sutter, *chair of the ISO C++ standards committee*

1 Περιγραφή προβλήματος

Η εξίσωση Laplace, γνωστή και ως εξίσωση θερμότητας, περιγράφει την κατανομή της θερμότητας σε μια δοσμένη περιοχή καθώς μεταβάλλεται ο χρόνος. Ορίζεται από την παρακάτω διαφορική εξίσωση:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x, y) = 0$$

Για την επίλυση του προβλήματος της διάδοσης θερμότητας σε δύο διαστάσεις, χρησιμοποιούνται τρεις υπολογιστικοί πυρήνες, οι οποίοι αποτελούν ευρέως διαδεδομένη δομική μονάδα για την επίλυση μερικών διαφορικών εξισώσεων: η μέθοδος Jacobi, η μέθοδος Gauss-Seidel με Successive OverRelaxation και η μέθοδος Red-Black SOR, που πραγματοποιεί Red-Black ordering στα στοιχεία του υπολογιστικού χωρίου και συνδυάζει τις δύο προηγούμενες μεθόδους.

1.1 Μέθοδος Jacobi

Στην μέθοδο αυτή, η τιμή ενός στοιχείου μια χρονική στιγμή ισούται με την μέση τιμή των γειτόνων του την προηγούμενη χρονική στιγμή, δηλαδή για το στοιχείο (x, y) την χρονική στιγμή $t + 1$ έχουμε:

$$u_{x,y}^{t+1} = \frac{u_{x-1,y}^{t} + u_{x,y-1}^{t} + u_{x+1,y}^{t} + u_{x,y+1}^{t}}{4}$$

Από άποψη υλοποίησης, απαιτούνται τουλάχιστον δύο πίνακες, ένας για την τρέχουσα (t+1) κι ένας για την προηγούμενη (t) χρονική στιγμή. Στο τέλος κάθε βήματος, πραγματοποιείται έλεγχος σύγκλισης.

```
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])
}

```

1.2 Μέθοδος Gauss-Seidel SOR

Στην μέθοδο αυτή, προσπαθούμε να αντιμετωπίσουμε το γεγονός ότι η μέθοδος Jacobi έχει πολύ αργό ρυθμό σύγκλισης. Συγκεκριμένα, όταν υπολογίζεται το σημείο $U[t + 1][i][j]$ έχουν ήδη υπολογιστεί τα σημεία $U[t + 1][i - 1][j]$ και $U[t + 1][i][j - 1]$. Συνεπώς, στην ανανέωση μπορούν να χρησιμοποιηθούν οι νέες τιμές των δύο αυτών γειτόνων, το οποίο θα επιταγχύνει την όλη διαδικασία. Έτσι, προκύπτει ο παρακάτω τύπος για την μέθοδο Gauss-Seidel:

$$u_{x,y}^{t+1} = \frac{u_{x-1,y}^{t+1} + u_{x,y-1}^{t+1} + u_{x+1,y}^{t} + u_{x,y+1}^{t}}{4}$$

Για ακόμη μεγαλύτερη σύγκλιση, χρησιμοποιείται η τεχνική Successive Over-Relaxation (SOR), η οποία χρησιμοποιεί και την προηγούμενη τιμής του ίδιου του στοιχείου και καταλήγει στην εξής σχέση:

$$u_{x,y}^{t+1} = u_{x,y}^{t} + \omega \frac{u_{x-1,y}^{t+1} + u_{x,y-1}^{t+1} + u_{x+1,y}^{t} + u_{x,y+1}^{t} - 4u_{x,y}^{t}}{4}, \ \omega \in (0, 2)$$

```
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])
}

```

1.3 Μέθοδος Red-Black SOR

Η μέθοδος αυτή είναι παρόμοια με την μέθοδο Gauss-Seidel SOR, με την διαφορά ότι χωρίζει τα στοιχεία σε δύο ομάδες, στα κόκκινα και στα μαύρα. Η ανανέωση γίνεται σε 2 φάσεις: στη κόκκινη φάση υπολογίζονται τα κόκκινα στοιχεία από τα μαύρα, ενώ στη μαύρη φάση υπολογίζονται τα μαύρα στοιχεία από τα κόκκινα. Έτσι, πετυχαίνουμε μεγαλύτερη ταχύτητα σύγκλισης.

$$u_{x,y}^{t+1} = u_{x,y}^{t} + \omega \frac{u_{x-1,y}^{t} + u_{x,y-1}^{t} + u_{x+1,y}^{t} + u_{x,y+1}^{t} - 4u_{x,y}^{t}}{4},\ when\ (x+y)\%2 = 0$$

$$u_{x,y}^{t+1} = u_{x,y}^{t} + \omega \frac{u_{x-1,y}^{t+1} + u_{x,y-1}^{t+1} + u_{x+1,y}^{t+1} + u_{x,y+1}^{t+1} - 4u_{x,y}^{t}}{4},\ when\ (x+y)\%2 = 1$$

```
for (t = 0; t < T && !converged; t++) {
    //Red phase
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            if ((i+j)%2==0)
                U[t+1][i][j]=U[t][i][j]+(omega /4)*(U[t][i-1][j]+U[t][i][j-1]
                            +U[t][i+1][j]+U[t][i][j+1]-4*U[t][i][j]);
    //Black phase
    for (i = 1; i < X - 1; i++)
        for (j = 1; j < Y - 1; j++)
            if ((i+j)%2==0)
                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+1][i+1][j]+U[t+1][i][j+1]-4*U[t][i][j]);
    converged=check_convergence(U[t+1],U[t])
}

```

2 Σχεδιασμός Παραλληλοποίησης

Στην γενική περίπτωση, ο πίνακας U είναι διάστασης $N \times M$. Τον χωρίζουμε σε ορθογώνια block μεγέθους $n \times m$ και υποθέτουμε ότι $N \mod n = 0$, $M \mod m = 0$ (αν όχι ορίζεται κατάλληλο padding). Για τον σχεδιασμό των task graph, θεωρούμε $\frac{N}{n} = \frac{M}{m} = 4$. Άρα ο πίνακας χώριζεται στα παρακάτω blocks:

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

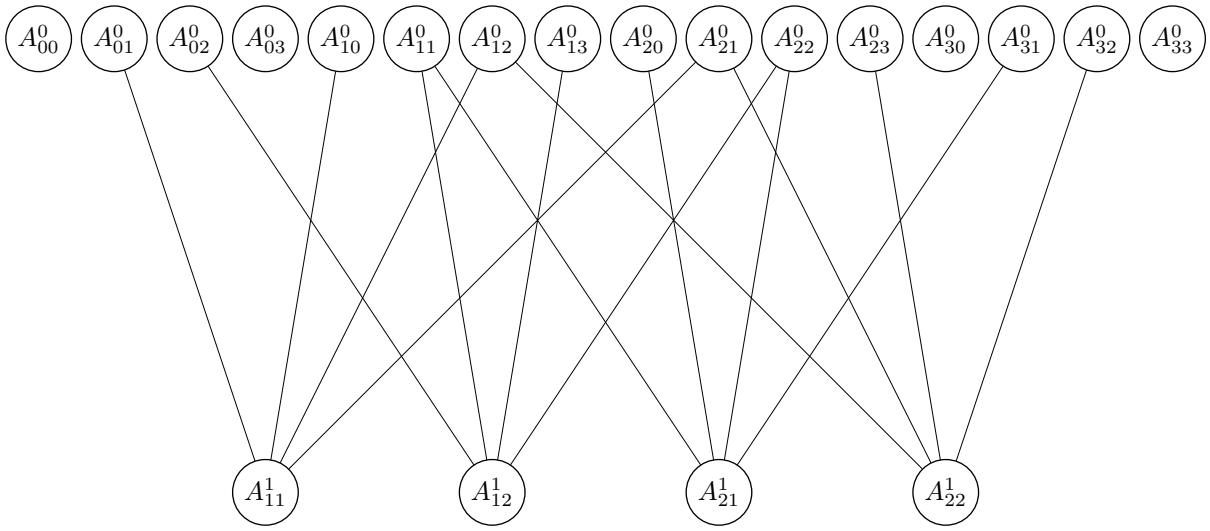
2.1 Μέθοδος Jacobi

1. **Κατανομή Υπολογισμών**

   Θεωρούμε, ως task τον υπολογισμό της θερμότητας για ένα block του δισδιάστατου πλέγματος για μία χρονική στιγμή.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

   Οι εξαρτήσεις που υπάρχουν μεταξύ των διαφορετικών tasks παρουσιάζονται για το συγκεκριμένο παράδειγμα στο παρακάτω task graph, όπου ορίζονται οι δύο πρώτες χρονικές στιγμές μόνο. Ουσιαστικά, κάθε χρονική στιγμή για να γίνει ο υπολογισμός ενός block χρειάζεται τις προηγούμενες τιμές από τα γειτονικά του μόνο blocks.



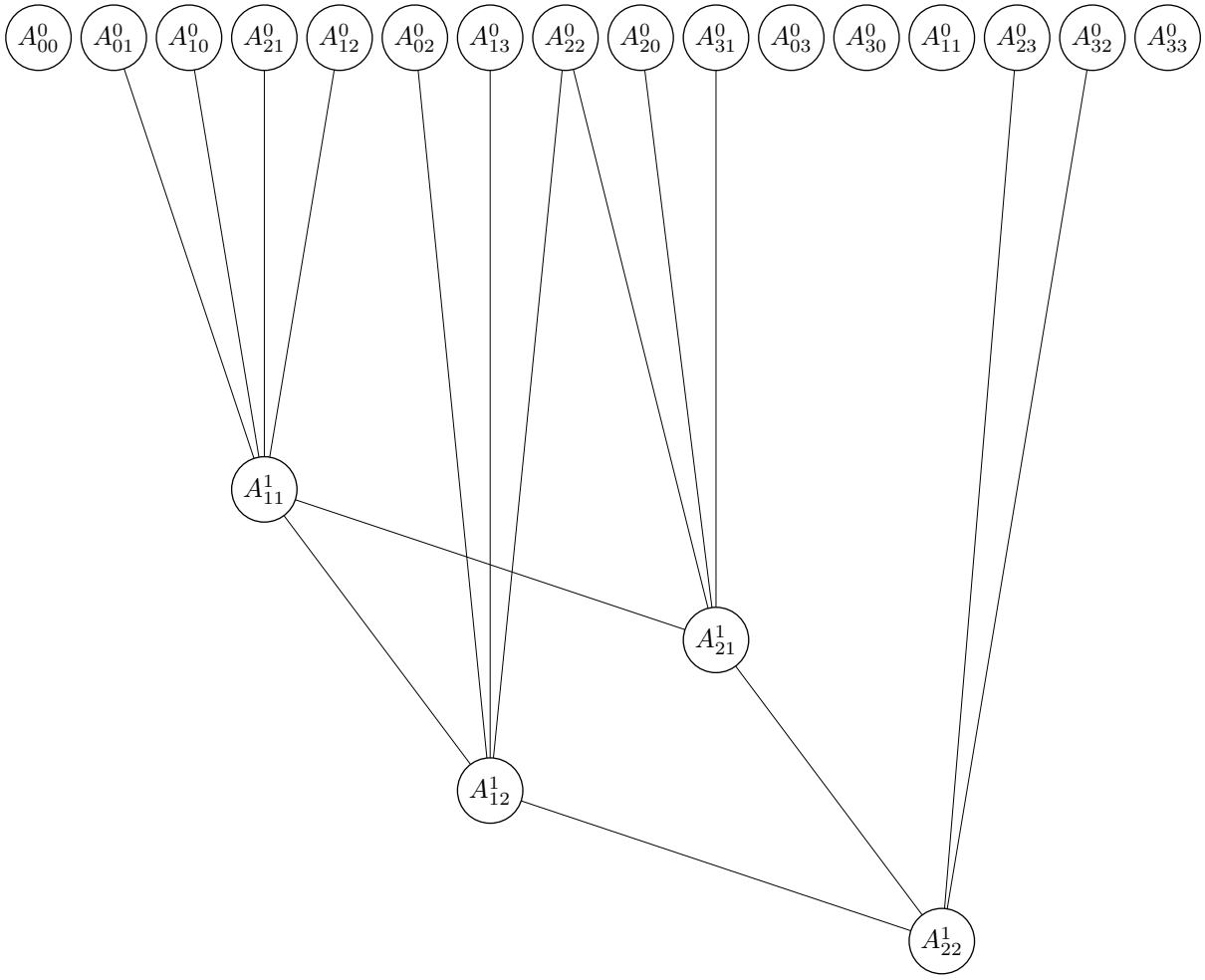
2.2 Μέθοδος Gauss-Seidel SOR

1. **Κατανομή Υπολογισμών**

   Αντίστοιχα με παραπάνω, θεωρούμε ως task τον υπολογισμό της θερμότητας για ένα block του δισδιάστατου πλέγματος για μία χρονική στιγμή.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

   Οι εξαρτήσεις εδώ διαφέρουν από αυτές του Jacobi. Κάθε χρονική στιγμή για να γίνει ο υπολογισμός ενός block χρειάζεται τις προηγούμενες τιμές από το δεξιά, το από κάτω και το ίδιο block και τις ανανεωμένες τιμές του από πάνω και αριστερά block. Έτσι, με βάση το παράδειγμα παραπάνω έχουμε το εξής task graph:



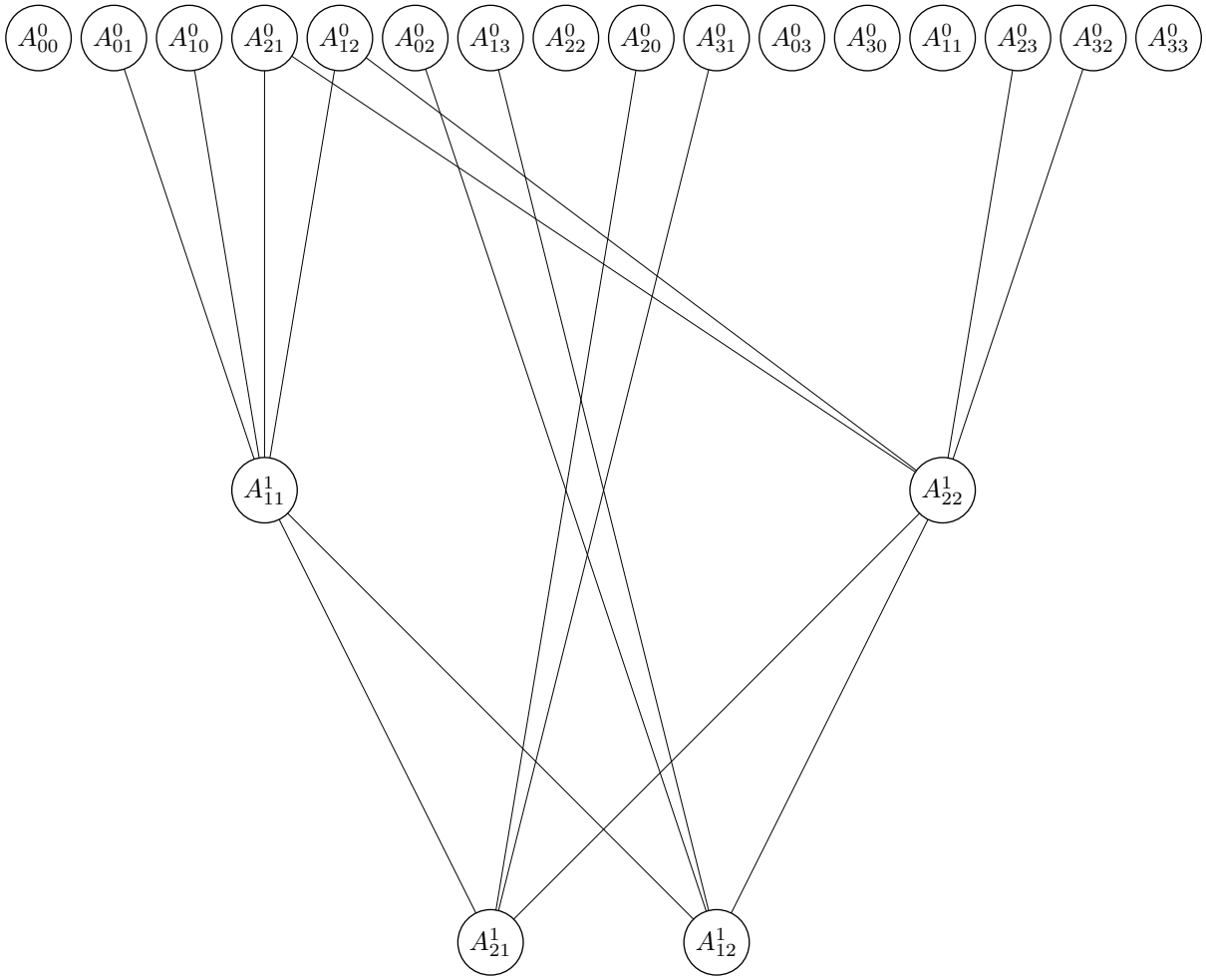
2.3 Μέθοδος Red-Black SOR

1. **Κατανομή Υπολογισμών**

   Αντίστοιχα με παραπάνω, θεωρούμε ως task τον υπολογισμό της θερμότητας για ένα block του δισδιάστατου πλέγματος για μία χρονική στιγμή.

2. **Ορισμός ορθής σειράς εκτέλεσης μέσω task graph**

   Ακολουθεί το task graph με βάση το αρχικό παράδειγμα που έχουμε:


A^0_{00}
A^0_{01}
A^0_{10}
A^0_{21}
A^0_{12}
A^0_{02}
A^0_{13}
A^0_{22}
A^0_{20}
A^0_{31}
A^0_{03}
A^0_{30}
A^0_{11}
A^0_{23}
A^0_{32}
A^0_{33}
A^1_{11}
A^1_{22}
A^1_{21}
A^1_{12}

3 Ανάπτυξη παράλληλων προγραμμάτων

Σε κάθε περίπτωση, ο αρχικός πίνακας U χωρίζεται σε block και κάθε διεργασία αναλαμβάνει από ένα block. Για κάθε επανάληψη, οι διεργασίες υπολογίζουν τις νέες τιμές τους αφού επικοινωνήσουν μεταξύ τους για να ανταλλάξουν δεδομένα. Έχουμε σύγκλιση όταν όλα τα block έχουν συγκλίνει στις τελικές τιμές τους. Στο τέλος, οι τιμές των επιμέρους block συνενώνονται στον αρχικό πίνακα. Ακολουθούν ορισμένες διαδικασίες που είναι κοινές και για τις 3 υλοποιήσεις.

1. Δημιουργία 2D καρτεσιανού communicator για την ευκολότερη διαχείριση των διεργασιών

```
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
```

2. Υπολογισμός διαστάσεων του επιμέρους πίνακα κάθε block. Σε περίπτωση που ο διαμοιρασμός των block δεν χωράει ακριβώς κάνουμε padding στον αρχικό πίνακα.

```
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];
    }
}
```

3. Διαμοιρασμός του αρχικού global πίνακα U στον local πίνακα κάθε διεργασίες

```
//-----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]);
    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);
```

4. Ορισμός μιας δομής column, που θα διευκολύνει την μεταφορά μεταξύ διεργασιών μιας στήλης ενός πίνακα.

```
/* Define a datatype that corresponds to a collumn of a local block */
MPI_Datatype column;
MPI_Type_vector(local[0], 1,local[1]+2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&column);
MPI_Type_commit(&column);
```

5. Υπολογισμός για κάθε διεργασία του rank των γειτόνων του ώστε να μπορεί να επικοινωνήσει μαζί τους.

```
int north, south, east, west;
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
```

6. Ορισμός του range πάνω στο οποίο η διεργασία θα ανανεώνει τον local πίνακά της. Εδώ, πρέπει να λάβουμε υπόψιν τόσο τα ghost cell που έχουν τοποθετηθεί για την διευκόλυνση της επικοινωνίας όσο και τα πιθανά padding cells.

```
int i_min, i_max, j_min, j_max;

/* internal process (ghost cell only) */
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;
}
```

7. Έλεγχος σύγκλισης, το οποίο συμβαίνει όταν όλα τα local block έχουνε συγκλίνει

```
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;
}
```

8. Συνένωση των επιμέρους local πινάκων στον αρχικό global πίνακα U, τον οποίο τον χειρίζεται η διεργασία με rank = 0

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

3.1 Μέθοδος Jacobi

Στην μέθοδο αυτή, κάθε σημείο για να ανανεωθεί χρειάζεται τις τιμές των γειτόνων του. Συνεπώς, κάθε φορά πριν το υπολογιστικό μέρος, κάθε block χρειάζεται να ανταλλάξει με τους γείτονές του τα σύνορά τους. Οι τιμές που έρχονται από τους γείτονες αποθηκεύονται στα ghost cells που έχουμε δεσμεύσει στην αρχή, με αποτέλεσμα να μην χρειάζεται να αλλάξει καθόλου ο βρόγχος ανανέωσης του σειριακού προγράμματος. Μόλις ολοκληρωθεί η επικοινωνία κάθε κελί θα έχει τις σωστές τιμές των γειτόνων του.

```
//----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
    swap=u_previous;
    u_previous=u_current;
    u_current=swap;
    /*Compute and Communicate*/
    // Communicate with north
    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){
```

```
22          MPI_Sendrecv(&u_previous[local[0]][1], local[1], MPI_DOUBLE, south, 0, &u_previous[
    local[0]+1][1],
23                               local[1], MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &status );
24      }
25      // Communicate with east
26      if (east != MPI_PROC_NULL){
27          MPI_Sendrecv(&u_previous[1][local[1]], 1, column, east, 0, &u_previous[1][local
    [1]+1],
28                               1, column, east, 0, MPI_COMM_WORLD, &status );
29      }
30      // Communicate with west
31      if (west != MPI_PROC_NULL){
32          MPI_Sendrecv(&u_previous[1][1], 1, column, west, 0, &u_previous[1][0],
33                               1, column, west, 0, MPI_COMM_WORLD, &status );
34      }
35      /*Add appropriate timers for computation*/
36      gettimeofday(&tcs, NULL);
37
38      for (i=i_min;i<i_max;i++)
39          for (j=j_min;j<j_max;j++)
40              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;
41
42      gettimeofday(&tcf, NULL);
43      tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
44  #ifdef TEST_CONV
45      if (t%C==0) {
46          /*Test convergence*/
47          gettimeofday(&tcvs, NULL);
48          converged=converge(u_previous,u_current,local[0],local[1]);
49          MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
50          gettimeofday(&tcvf, NULL);
51          tconv += (tcvf.tv_sec-tcvs.tv_sec)+(tcvf.tv_usec-tcvs.tv_usec)*0.000001;
52      }
53  #endif
54  }
55  gettimeofday(&ttf,NULL);
56  ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;
57
58  MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
59  MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
60  MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
```

3.2 Μέθοδος Gauss-Seidel SOR

Στην μέθοδο αυτή, η υλοποίηση της επικοινωνίας δεν είναι τόσο απλή. Κάθε block πρέπει:

- Πριν τον υπολογισμό, να στείλει στον πάνω και στον αριστερά τα δεδομένα της προηγούμενης χρονικής στιγμής.
- Πριν τον υπολογισμό, να λάβει από τον πάνω και τον αριστερά τα δεδομένα της τωρινής χρονικής στιγμής.
- Πριν τον υπολογισμό, να λάβει από τον δεξιά και τον κάτω τα δεδομένα της προηγούμενης χρονικής στιγμής.

- Μετά τον υπολογισμό, να στείλει στον δεξιά και στον κάτω τα δεδομένα της τωρινής χρονική στιγμής.

Ουσιαστικά, αυτό που θα συμβεί είναι ότι αρχικά όλες οι διεργασίες, εκτός από αυτές που βρίσκονται πάνω και αριστερά στο ταμπλό, θα μπλοκάρουν περιμένοντας τα τωρινά δεδομένα από τον πάνω και τον αριστερά τους. Οι διεργασίες που δεν έχουν γείτονα πάνω και αριστερά θα υπολογίσουν τις νέες τιμές και θα τις στείλουν στον κάτω και δεξιά γείτονά τους. Η διαδικασία αυτή θα επαναληφθεί μέχρι όλα τα block να ανανεωθούν κ.ο.κ.. Για να υλοποιήσουμε αυτή την διαδικασία, χρησιμοποιήσαμε non-blocking send και receive συναρτήσεις και θέσαμε στα σημεία όπου χρειαζόμαστε τις γειτονικές τιμές το καταάλληλο wait, όπως φαίνεται και στον κώδικα παρακάτω.

```
MPI_Request before_req[6];
MPI_Status before_status[6];
MPI_Request after_req[2];
MPI_Status after_status[2];
int before_req_len = 0;
int after_req_len = 0;

//----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

    before_req_len = 0;
    after_req_len = 0;

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

    /* Send data to north */
    if (north != MPI_PROC_NULL){
        MPI_Isend(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0, MPI_COMM_WORLD, &
    before_req[before_req_len]);
        MPI_Irecv(&u_current[0][1], local[1], MPI_DOUBLE, north, 0, MPI_COMM_WORLD, &
    before_req[before_req_len + 1]);
        before_req_len += 2;
    }

    /* Send data to west */
    if (west != MPI_PROC_NULL){
        MPI_Isend(&u_previous[1][1], 1, column, west, 0, MPI_COMM_WORLD, &before_req[
    before_req_len]);
        MPI_Irecv(&u_current[1][0], 1, column, west, 0, MPI_COMM_WORLD, &before_req[
    before_req_len + 1]);
            before_req_len += 2;
    }

    /* Get data from south */
    if (south != MPI_PROC_NULL){
```

```
42          MPI_Irecv(&u_previous[local[0]+1][1], local[1], MPI_DOUBLE, south, 0, MPI_COMM_WORLD
    , &before_req[before_req_len]);
43          before_req_len ++;
44      }
45      /* Get data from east */
46      if (east != MPI_PROC_NULL){
47          MPI_Irecv(&u_previous[1][local[1]+1], 1, column, east, 0, MPI_COMM_WORLD, &
    before_req[before_req_len]);
48          before_req_len ++;
49      }
50      MPI_Waitall(before_req_len, before_req, before_status);
51
52      /*Compute and Communicate*/
53
54      /*Add appropriate timers for computation*/
55      gettimeofday(&tcs, NULL);
56      for (i=i_min;i<i_max;i++)
57          for (j=j_min;j<j_max;j++)
58              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]);
59
60      gettimeofday(&tcf, NULL);
61      tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
62
63      /* Send data to south */
64      if (south != MPI_PROC_NULL){
65          MPI_Isend(&u_current[local[0]][1], local[1], MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &
    after_req[after_req_len]);
66          after_req_len ++;
67      }
68
69      /* Send data to east */
70      if (east != MPI_PROC_NULL){
71          MPI_Isend(&u_current[1][local[1]], 1, column, east, 0, MPI_COMM_WORLD, &after_req[
    after_req_len]);
72          after_req_len ++;
73      }
74      MPI_Waitall(after_req_len, after_req, after_status);
75
76  #ifdef TEST_CONV
77      if (t%C==0) {
78          /*Test convergence*/
79          gettimeofday(&tcvs, NULL);
80          converged=converge(u_previous,u_current,local[0],local[1]);
81          MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
82          gettimeofday(&tcvf, NULL);
83          tconv += (tcvf.tv_sec-tcvs.tv_sec)+(tcvf.tv_usec-tcvs.tv_usec)*0.000001;
84      }
85  #endif
86  }
87  gettimeofday(&ttf,NULL);
88  ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;
89
90  MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
91  MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
92  MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
```

3.3 Μέθοδος Red-Black SOR

Τέλος, στην μέθοδο Red-Black SOR έχουμε δύο φάσεις υπολογισμού. Στην πρώτη φάση, τα κόκκινα στοιχεία πρέπει να γνωρίζουν τους γείτονές τους και στην δεύτερη φάση τα μαύρα στοιχεία αντίστοιχα. Από την στιγμή, που τα κόκκινα και τα μαύρα στοιχεία τοποθετούνται εναλλάξ στις άρτιες και περιττές θέσεις, και στις δύο φάσεις όλα τα block χρειάζεται να επικοινωνήσουν με τους γείτονές τους. Έτσι, καταλήγουμε σε ένα αντίστοιχο μοντέλο με αυτό του Jacobi, με την διαφορά ότι τώρα θα γίνει 2 φορές η επικοινωνία, μία για να ενημερωθεί ο πίνακας u_previous και να υπολογιστούν τα κόκκινα και μία για να ενημερωθεί ο πίνακας u_current και να υπολογιστούν τα μαύρα στοιχεία.

```
//----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

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

    // Communicate with north
    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, column, east, 0, &u_previous[1][local
    [1]+1],
                            1, column, east, 0, MPI_COMM_WORLD, &status );
    }

    // Communicate with west
    if (west != MPI_PROC_NULL){
        MPI_Sendrecv(&u_previous[1][1], 1, column, west, 0, &u_previous[1][0],
                            1, column, 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++)
```

```
            if ((i+j)%2==0)
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_previous[i-1][j]+u_previous[
    i+1][j]+u_previous[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;

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

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

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

    gettimeofday(&tcs, NULL);

    for (i=i_min;i<i_max;i++)
        for (j=j_min;j<j_max;j++)
            if ((i+j)%2==1)
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_current[i-1][j]+u_current[i
    +1][j]+u_current[i][j-1]+u_current[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;

#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);
```

4 Μετρήσεις επίδοσης

4.1 Μετρήσεις με έλεγχο σύγκλισης

Εδώ, πραγματοποιήσαμε μετρήσεις με έλεγχο σύγκλισης για τα παράλληλα προγράμματά που αναπτύχθηκαν σε MPI, για μέγεθος πίνακα 1024x1024. Γι' αυτό το μέγεθος πίνακα, λάβαμε μετρήσεις (συνολικός χρόνος, χρόνος υπολογισμών, χρόνος ελέγχου σύγκλισης) για 64 MPI διεργασίες. Οι μετρήσεις αυτές συνοψίζονται στον παρακάτω πίνακα:

Algorithm	Grid Size	Iterations	Compute	Converge	Communicate	Total
Jacobi	16x4	798201	36.06	23.57	536.5	599.13
GaussSeidelSOR	16x4	3201	0.47	0.23	2.58	3.28
RedBlackSOR	16x4	2501	0.29	0.14	3.39	3.82
Jacobi	4x16	798201	38.9	28.23	418.86	485.99
GaussSeidelSOR	4x16	3201	0.44	0.22	2.22	2.88
RedBlackSOR	4x16	2501	0.29	0.11	2.7	3.1
Jacobi	8x8	798201	40.28	9.49	217.01	266.78
GaussSeidelSOR	8x8	3201	0.48	0.13	1.33	1.94
RedBlackSOR	8x8	2501	0.31	0.08	1.38	1.77

Συνοψίζουμε τις παραπάνω μετρήσεις, στα διαγράμματα που ακολουθούν:

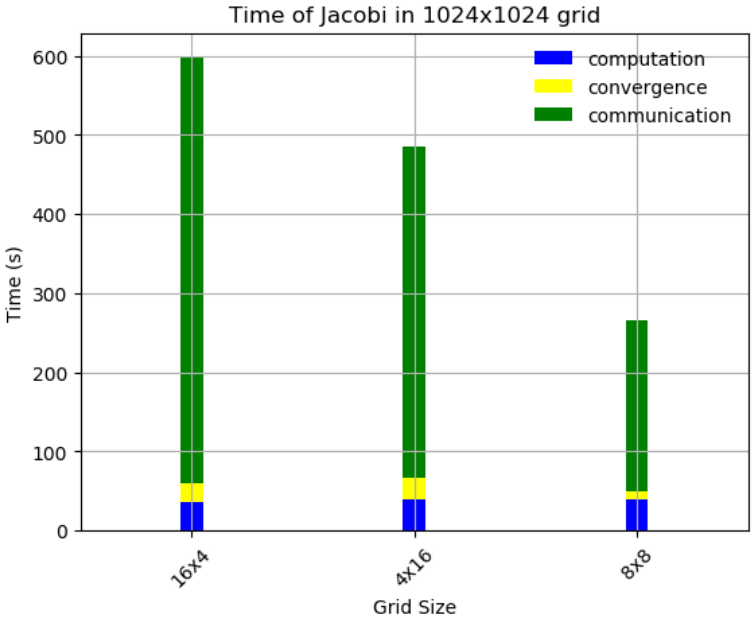
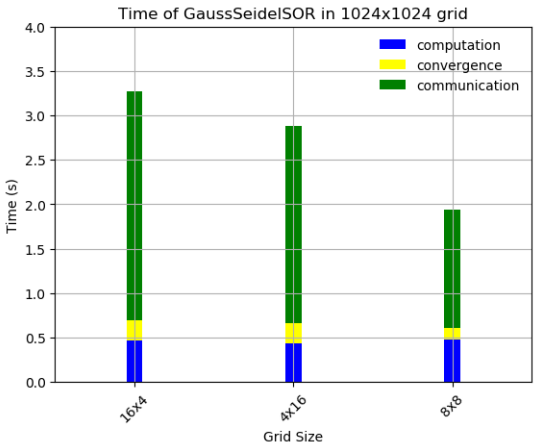
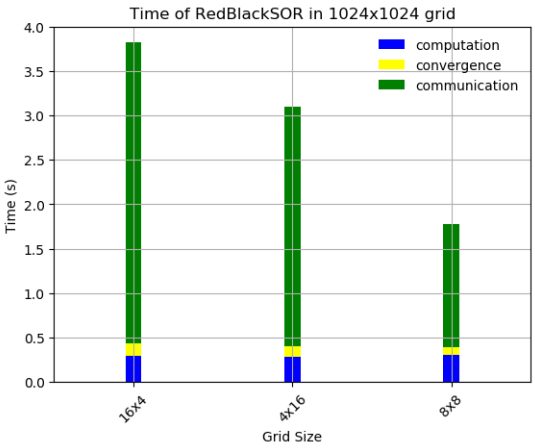


Figure 1: Barplot για Jacobi



(a) Barplot για GaussSeidelSOR



(b) Barplot για RedBlackSOR

Αρχικά, παρατηρούμε ότι σε όλες τις περιπτώσεις μας συμφέρει να ορίσουμε grid size 8x8, καθώς έτσι έχουμε μεταφορά ίσου μεγέθους δεδομένων μεταξύ των διεργασιών, Στις άλλες περιπτώσεις, η μεταφορά των δεδομένων στην μεγαλύτερη πλευρά ενός block καθυστερούσε όλη την διαδικασία. Ως προς το είδος του αλγορίθμου τα συμπεράσματα από τα πειράματα ήταν και αυτά που αναμέναμε θεωρητικά. Δηλαδή:

- ✓ Ο Jacobi είναι πολύ πιο αργός από τους άλλους δύο αλγορίθμους. Αυτό συμβαίνει διότι έχει πάρα πολύ αργό ρυθμό σύγκλισης. Αν δούμε τον παραπάνω πίνακα, συγκλίνει μετά από 798201 επαναλήψεις αριθμός 200-πλάσιος από τις επαναλήψεις των άλλων δύο. Συνεπώς, δεν έχει καμία σημασία ο χρόνος υπολογισμού, επικοινωνίας και σύγκλισης του Jacobi σε μία επανάληψη από την στιγμή που οι επαναλήψεις του είναι τόσο πολλές.

- ✓ Ανάμεσα στους άλλους δύο, είναι λίγο πιο γρήγορος ο RedBlackSOR. Ουσιαστικά, η διαφορά των δύο αλγορίθμων είναι ότι από την μία ο Gauss έχει μικρότερο χρόνο επικοινωνίας και υπολογισμού ανά επανάληψη αλλά από την άλλη ο RedBlack συκγλίνει πιο γρήγορα (700 επαναλήψεις λιγότερο). Συνεπώς, οι λιγότερες επαναλήψεις του δεύτερου υπερχισχύουν του καλύτερου χρόνου ανά επανάληψη που έχει ο πρώτος. Για τον λόγο αυτό, βλέπουμε τις πράσινες περιοχές σχέδον ίσες αλλά τις κίτρινες και μπλε μικρότερες στον RedBlackSOR.

- ✓ Παρατηρούμε την μεγάλη σημασία που έχει στα συστήματα κατανεμημένης μνήμης το δίκτυο διασύνδεσης καθώς σε όλες τις περιπτώσεις ο χρόνος επικοινωνίας είναι ο μεγαλύτερος (μη ξεχνάμε ότι έχουμε 64 MPI διεργασίες που σημαίνει επικοινωνία μεταξύ 8 nodes).

Συνολικά, για την επίλυση του προβλήματος στο συγκεκριμένο σύστημα κατανεμημένης μνήμης θα επιλέγαμε την μέθοδο RedBlackSOR. Ωστόσο, στην γενική περίπτωση, η απάντηση δεν είναι τόσο απόλυτη. Σε περίπτωση για παράδειγμα που υπάρχει υψηλή κίνηση στο δίκτυο διασύνδεσης ίσως προτιμούσαμε την μέθοδο GaussSeidel. Αυτό γιατί ο RedBlackSOR έχει υψηλότερο ρυθμό επικοινωνίας και η απόδοσή του θα επηρεαζόταν περισσότερο από την απόδοση του GaussSeidel.

4.2 Μετρήσεις χωρίς έλεγχο σύγκλισης

Εδώ, πραγματοποιήσαμε μετρήσεις, απενεργοποιώντας τον έλεγχο σύγκλισης, για σταθερό αριθμό επαναλήψεων (T=256), για μεγέθη πίνακα 2048x2048, 4096x4096, 6144x6144, για 1,2,4,8,16,32 και 64

MPI διεργασίες. Αρχικά, παρουσιάζεται ένα διάγραμμα επιτάχυνσης για κάθε μέγεθος πίνακα (άξονας x: MPI διεργασίες, άξονας y: speedup) που απεικονίζει τις τρεις μεθόδους (Jacobi, Gauss-Seidel SOR και Red-Black SOR).

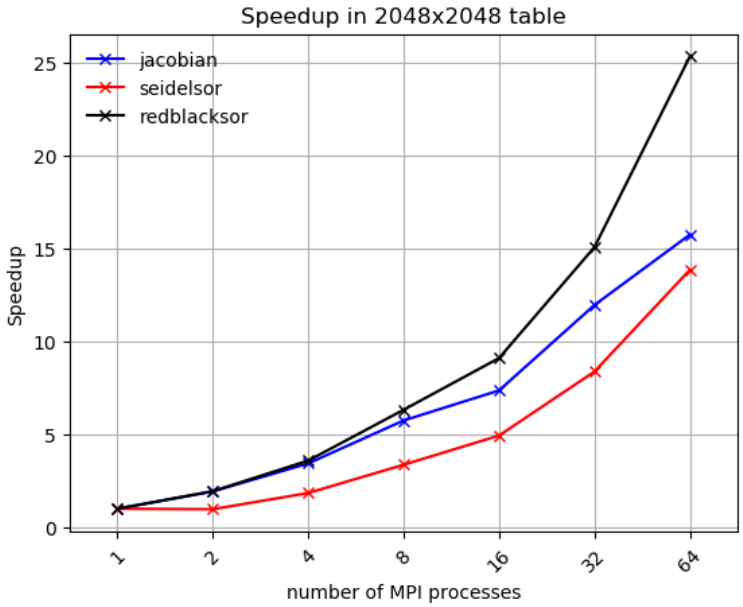


Figure 3: Speedup σε 2048x2048

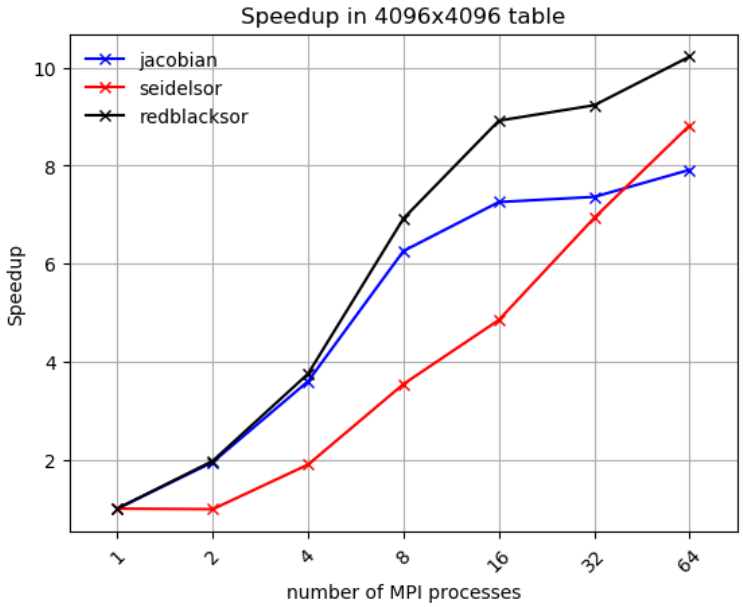


Figure 4: Speedup σε 4096x4096

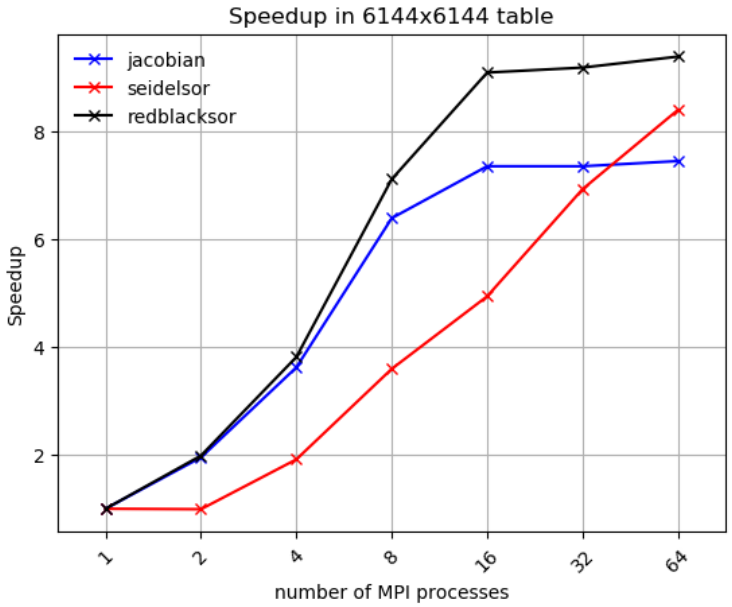
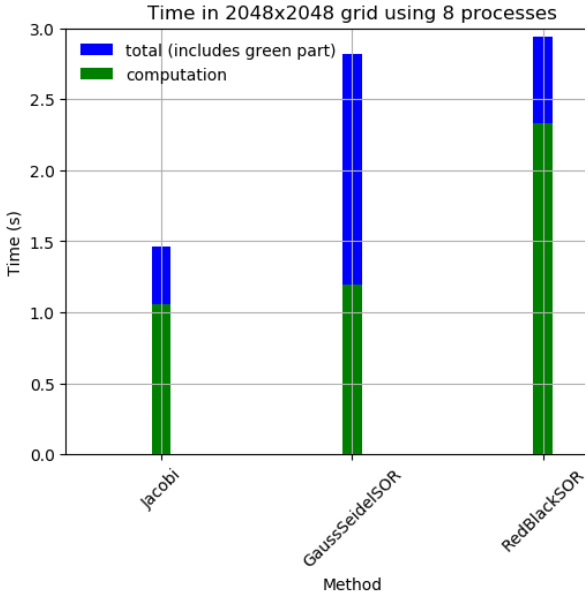


Figure 5: Speedup σε 6144x6144

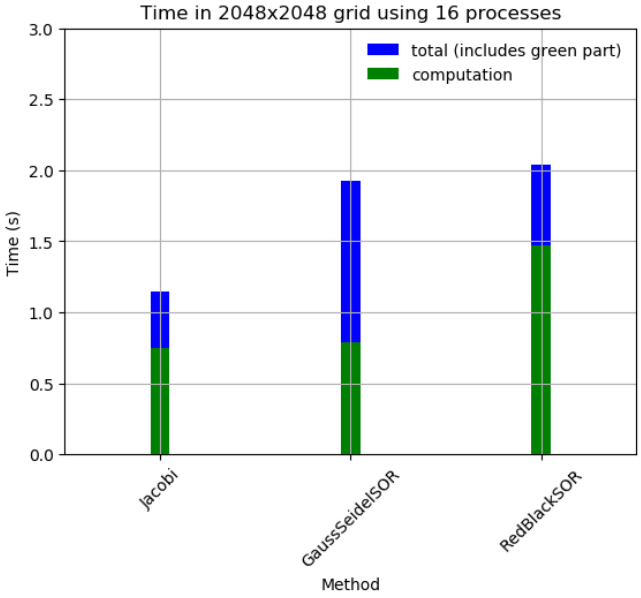
Επίσης, για καλύτερη ανάλυση των μετρήσεων, κατασκευάζουμε διαγράμματα με μπάρες (1 για κάθε

μέγεθος πίνακα και αριθμό επεξεργαστών) που θα απεικονίζουν το συνολικό χρόνο εκτέλεσης και το χρόνο υπολογισμού για κάθε μία από τις τρεις μεθόδους (άξονας x: μέθοδος, άξονας y: χρόνος). Για λόγους καλύτερης εποπτείας, κατασκευάσαμε διαγράμματα μόνο για 8, 16, 32 και 64 MPI διεργασίες, κρατώντας κοινή κλίμακα στον άξονα y ανά μέγεθος πίνακα.

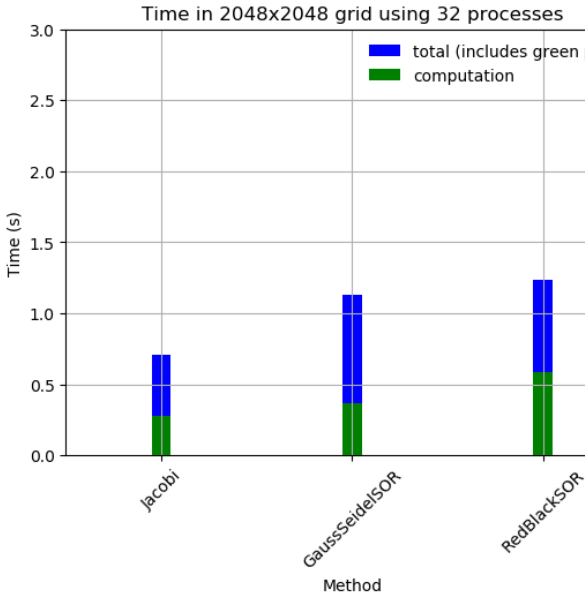
- 2048x2048



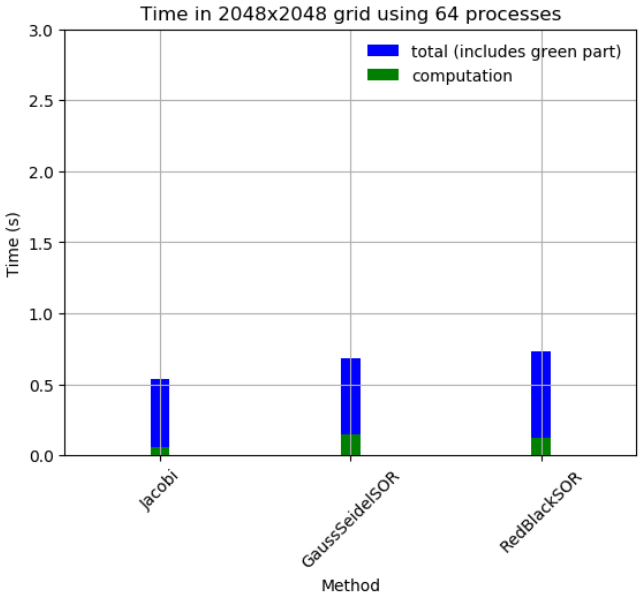
(a) Barplot για 8 processes



(b) Barplot για 16 processes

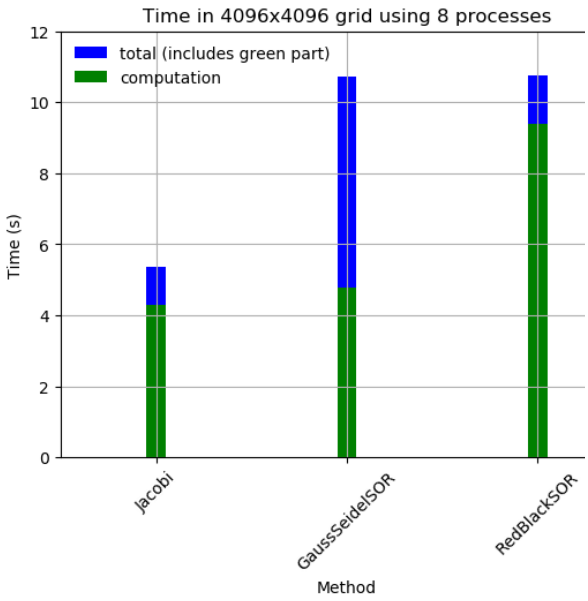


(a) Barplot για 32 processes

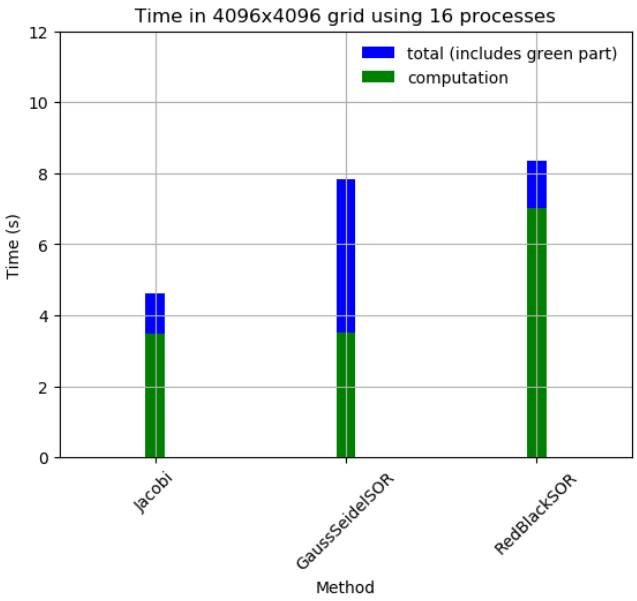


(b) Barplot για 64 processes

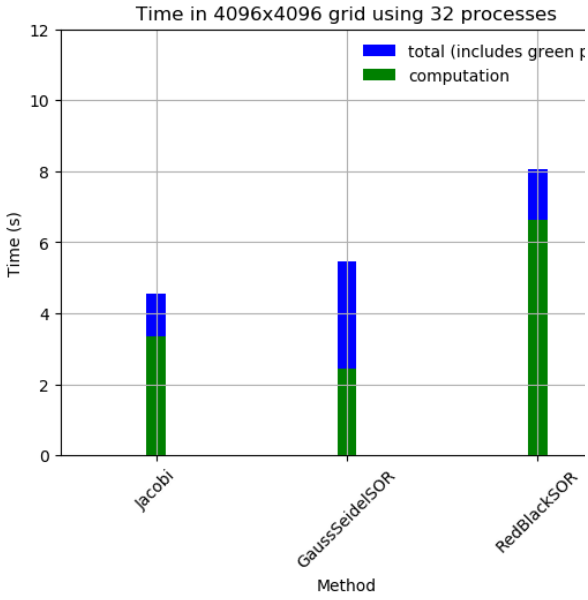
- 4096x4096



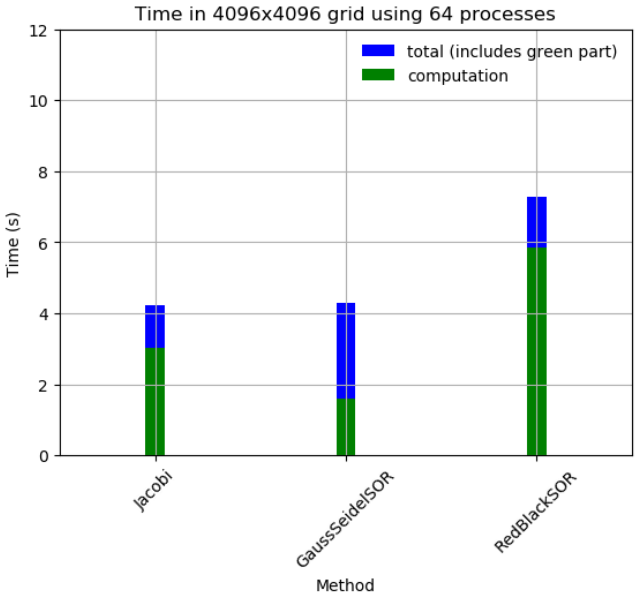
(a) Barplot για 8 processes



(b) Barplot για 16 processes

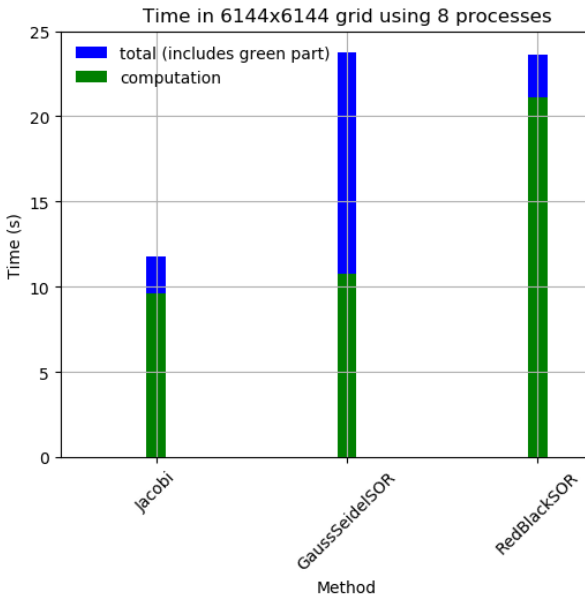


(a) Barplot για 32 processes

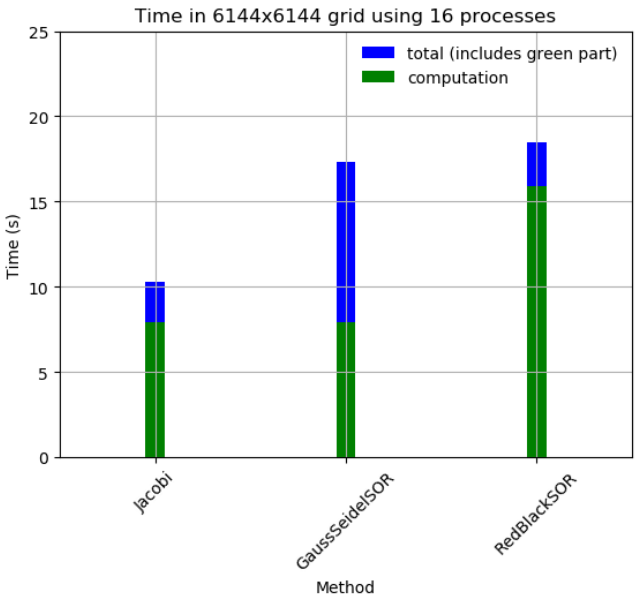


(b) Barplot για 64 processes

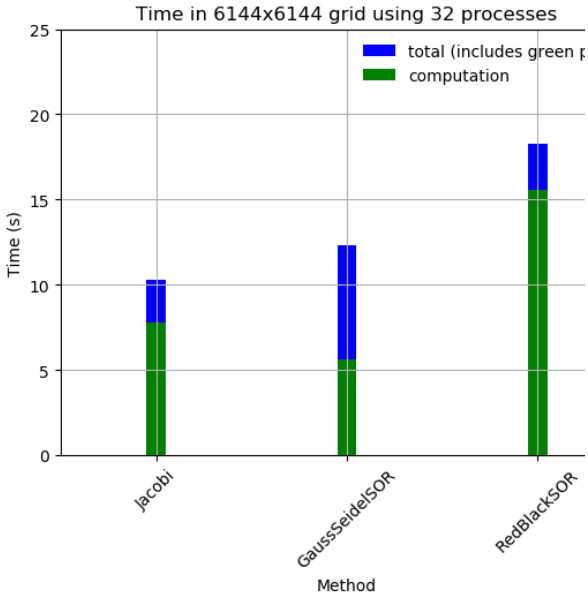
- 6144x6144



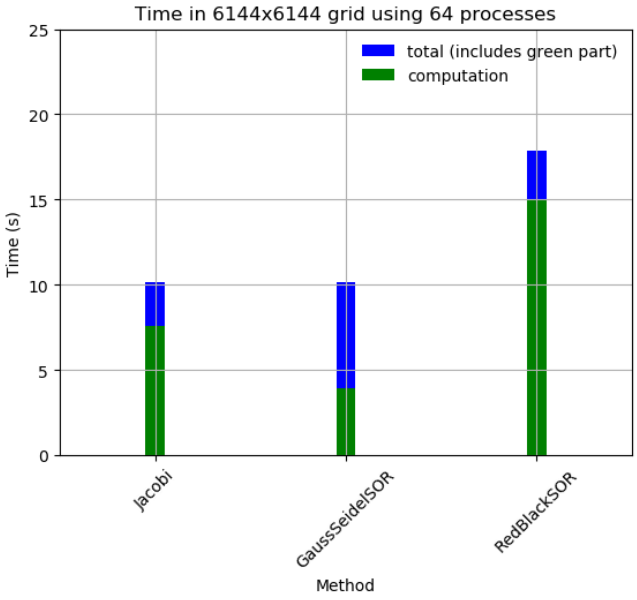
(a) Barplot για 8 processes



(b) Barplot για 16 processes



(a) Barplot για 32 processes



(b) Barplot για 64 processes

Έχουμε τις εξής παρατηρήσεις ως προς την κλιμακωσιμότητα των αλγορίθμων μας:

- ✓ Για σταθερό αριθμό επαναλήψεων, ο Jacobi είναι πιο γρήγορος από τους δύο άλλους αλγορίθμους καθώς έχει λιγότερο κόστος επικοινωνίας από τον Gauss και λιγότερο υπολογιστικό κόστος από τον RebBlack.

- ✓ Αν εξαιρέσουμε το μικρότερο μέγεθος πίνακα, όπου κλιμακώνουν όλοι οι αλγόριθμοι, αυτός που κλιμακώνει καλύτερα για τα μεγαλύτερα μεγέθη είναι ο GaussSeidel. Ο RedBlackSOR μπορεί να έχει συνεχώς μεγαλύτερο speedup από τους άλλους δύο αλλά δεν πρέπει να παραβλέψουμε ότι ο χρόνος για ένα πυρήνα του RedBlackSOR είναι σχεδόν διπλάσιος από τους άλλους δύο, το οποίο οφείλεται στο διπλάσιο υπολογιστικό κόστος. Έτσι, μπορεί ο RedBlack να έχει μεγάλο speedup αλλά δεν κλιμακώνει από ένα σημείο και μετά λόγω του υπολογιστικού κόστους που πρέπει να καλύπτει κάθε φορά.

- ✓ Η καλύτερη κλιμάκωση του Gauss φαίνεται και στα τελευταία barplot όπου καταφέρνει να πετύχει ίσους χρόνους με τον Jacobi, παρά το γεγονός ότι προφανώς ο χρόνος επικοινωνίας είναι μεγαλύτερος.

- ✓ Συνολικά, ο GaussSeidelSOR, παρά το κόστος επικοινωνίας που έχει, κλιμακώνει καλύτερα μειώνοντας συνεχώς το συνολικό του χρόνο, το οποίο οφείλεται σίγουρα και στην non-blocking διαχείριση της επικοινωνίας. Από την άλλη, ο RedBlackSOR και ο Jacobi δεν κλιμακώνουν τόσο καλά διότι τα blocking SendRecv εμποδίζουν την επιπλέον μείωση του χρόνου επικοινωνίας.

5 Σημειώσεις

- Όλες οι υλοποιήσεις μεταγλωτίστηκαν με όρισμα -O3 για compiler optimizations.
- Η ορθότητα κάθε υλοποίησης εξετάστηκε συγκρίνοντας το output που έδινε με το output του αντίστοιχου σειριακού αλγορίθμου.
- Μια πιθανή βελτιστοποίηση στο Jacobi που δεν προλάβαμε να υλοποιήσουμε είναι, καθώς γίνεται non-blocking επικοινωνία με τους γείτονες, να υπολογίζονται οι νέες τιμές στο εσωτερικό του πίνακα όπου δεν απαιτείται η γνώση τιμών που ανήκουν σε άλλες διεργασίες.
- Όπου αναφέρεται χρόνος επικοινωνίας, έχει προκύψει από την διαφορά του συνολικού χρόνου από τον χρόνο σύγκλισης κα τον χρόνο υπολογισμού. Συνεπώς, περιλαμβάνει τον χρόνο για το swap το οποίο θεωρείται αμελητέο.
- Όλες οι μετρήσεις έγιναν από 3 φορές και κρατήσαμε κάθε φορά τον μέσο όρο τους για μεγαλύτερη ακρίβεια.

Αναφορές

[1] "Σημειώσεις του μαθήματος" http://www.cslab.ntua.gr/courses/pps/notes.go

[2] "MPI documentation" https://www.open-mpi.org/doc/

[3] "Heat equation" https://en.wikipedia.org/wiki/Heat_equation