9ο Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Συστήματα Παράλληλης Επεξεργασίας

Αναφορά 3ης Εργαστηριακής Άσκησης: Παράλληλη επίλυση εξίσωσης θερμότητας

parlab20

Ζάρα Στέλλα, 03117154
Λιάγκα Αικατερίνη, 03117208

Ανακάλυψη Παραλληλισμού

Μέθοδος Jacobi

Ο αλγόριθμος υπολογισμού της θερμότητας με την μέθοδο Jacobi φαίνεται παρακάτω.

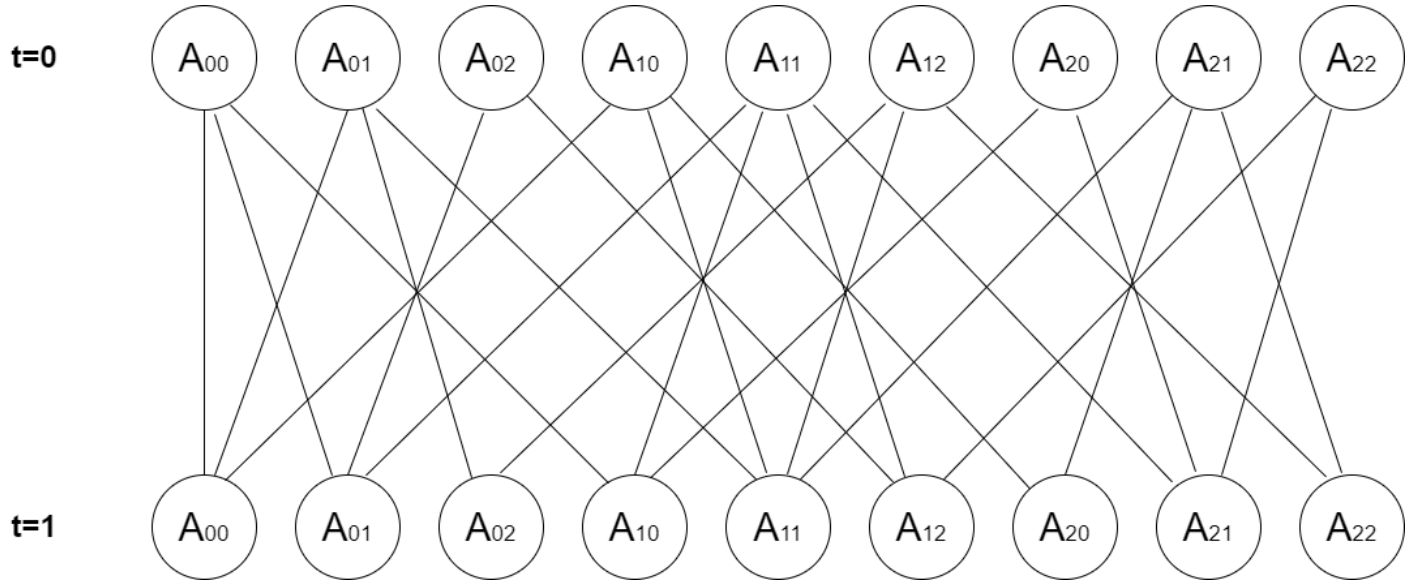
```
for (t = 0; t < T && !converged; t++) {
   for (i = 1; i < X - 1; i++)
      for (j = 1; j < Y - 1; j++)
         U[t+1][i][j]=(1/4)*(U[t][i-1][j]+U[t][i][j-1]+U[t][i+1][j]+U[t][i][j+1]);
   converged=check_convergence(U[t+1],U[t])
}
```

Παρατηρούμε ότι τα δύο εσωτερικά for loops είναι ανεξάρτητα και μπορούν να παραλληλοποιηθούν πλήρως. Παρατηρείται ότι η μόνη εξάρτηση βρίσκεται στο εξωτερικό loop και συγκεκριμένα είναι ότι για τον υπολογισμό ενός κελιού είναι αναγκαίο να γνωρίζουμε την τιμή που είχαν τα γειτονικά σε αυτό κελιά **στον προηγούμενο πίνακα**. Άρα είναι αναγκαίο να έχουμε υπολογίσει πλήρως τον προηγούμενο πίνακα, εξού και το γεγονός ότι το εξωτερικό for loop δεν μπορεί να παραλληλοποιηθεί.

Αναλυτικότερα, έστω ότι έχουμε έναν πίνακα A με θερμότητες μεγέθους 3X3, ο οποίος παρουσιάζεται παρακάτω

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

Για να γίνει καλύτερη κατανόηση των εξαρτήσεων στη μέθοδο Jacobi κατασκευάζεται ένας task graph για τον παραπάνω πίνακα, θεωρώντας ως ένα task τον υπολογισμό της θερμότητας ενός κελιού.



Επομένως επιβεβαιώνεται ότι όντως οι εξαρτησεις αφορούν μόνο το εξωτερικό loop. Η γενική ιδέα για την παραλληλοποίηση είναι η εξής:
Ο αρχικός πίνακας χωρίζεται σε μικρότερους, ίσου μεγέθους, υποπίνακες με βάση τον αριθμό των διεργασιών που διαθέτουμε. Αν μάλιστα δεν μπορεί ο αρχικός πίνακας να χωριστεί ακριβώς προστίθεται padding (το οποίο διαχειριζόμαστε κατάλληλα όταν γίνονται οι υπολογισμοί των κελιών) δεξιά (east) και κατω (south) για να μπορούμε να δουλέψουμε ομοιόμορφα με τις MPI διεργασίες. Αρχικά η root διεργασία διαχωρίζει τον πίνακα με τα δεδομένα της θερμότητας σε υποπίνακες και στέλνει σε όλες τις επιμέρους διεργασίες τους υποπίνακες current και previous που τους αντιστοιχούν. Στην συνέχεια κάθε διεργασία επικοινωνεί με τις γειτονικές της, εφόσον αυτές υπάρχουν, για να ανταλλάξει (ασύγχρονα) με τις εξής πληροφορίες:

- Στον βόρειο γείτονα (north) στέλνει την πρώτη της γραμμή και λαμβάνει την τελευταία γραμμή αυτού (την οποία τοποθετεί πριν την πρώτη γραμμή του πίνακα της)
- Στον νότιο γείτονα (south) στέλνει την τελευταία της γραμμή και λαμβάνει την πρώτη γραμμή αυτού (την οποία τοποθετεί μετά την τελευταία γραμμή του πίνακα της)
- Στον δυτικό γείτονα (west) στέλνει την πρώτη της στήλη και λαμβάνει την τελευταία στήλη αυτού (την οποία τοποθετεί πριν την πρώτη στήλη του πίνακα της)
- Στον ανατολικό γείτονα (east) στέλνει την τελευταία της στήλη και λαμβάνει την πρώτη στήλη αυτού (την οποία τοποθετεί μετά την τελευταία στήλη του πίνακα της)

Κάθε διεργασία αφού τελειώσει η επικοινωνία με τις υπόλοιπες διεργασίες εκτελεί, για τον υποπίνακα που της έχει ανατεθεί, τον αλγόριθμο Jacobi. Τέλος, όταν ο αλγόριθμος εκτελέσει και τις 256 ή συγκλίνει, η διεργασία root μαζεύει όλους τους υποπίνακες από τις υπόλοιπες διεργασίες και τους συνενώνει στον αρχικό πίνακα.

Πιο αναλυτικά η σύνδεση των βηματων της παραλληλοποίησης με τα MPI εργαλεία που χρησιμοποιήθηκαν εξηγείται παρακάτω που παρατίθεται και ο αντίστοιχος κώδικας.

Μέθοδος Gauss-Seidel SOR

Ο αλγόριθμος υπολογισμού της θερμότητας με τη μέθοδο Gauss-Seidel SOR (Successive Over-Relaxation) παρατίθεται παρακάτω:

```
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]+U[t+1][i-1][j]+U[t][i+1][j]+U[t+1][i][j-1]+U[t][i
][j+1]-4*U[t][i][j]*omega/4.0);

   converged=check_convergence(U[t+1],U[t])
}
```

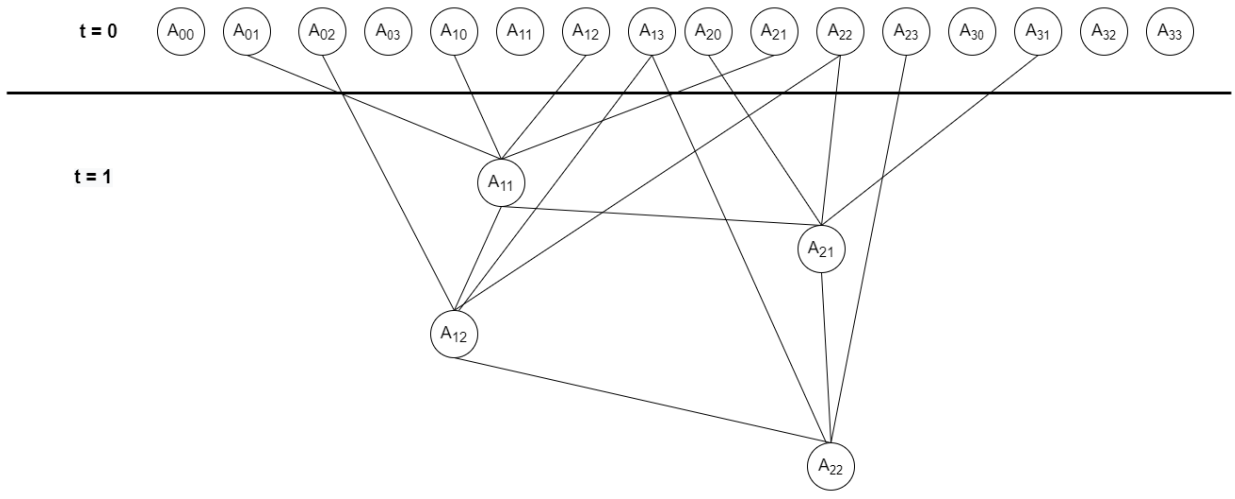
Από το παραπάνω παρατηρούμε ότι ο υπολογισμός της θερμότητας σε ένα τετράγωνο εξαρτάται από τις τιμές των γύρω τετραγώνων στο παρόν στάδιο αλλά και κατά το προηγούμενο. Πιο συγκεκριμένα, χρειαζόμαστε τις τιμές που προέκυψαν από το παρόν στάδιο υπολογισμών για τα τετράγωνα που βρίσκονται πάνω (north) και αριστερά (west), ενώ χρησιμοποιούμε τις τιμές από το προηγούμενο στάδιο υπολογισμών για τα τετράγωνα που βρίσκονται κάτω (south) και δεξιά (east). Συνεπώς, για τον υπολογισμό του τετραγώνου χρειάζεται εκτός από τον πίνακα με τους προηγούμενους υπολογισμούς και τα στοιχεία των πάνω αριστερά blocks.

Η συγκεκριμένη μέθοδος στοχεύει στο να προσφέρει καλύτερο χρόνο σύγκλισης από τη μέθοδο Jacobi αφού αξιοποιεί τις νέες τιμές που υπολογίζονται σε συνδυασμό με αυτές που υπάρχουν από το προηγούμενο στάδιο υπολογισμού.

Έστω πως έχουμε έναν πίνακα με θερμότητες που αποτελείται από 16 block, όπως φαίνεται παρακάτω:

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}

Τότε, αν θεωρήσουμε πως ο υπολογισμός ενός block είναι ένα task, προκύπτει το παρακάτω task graph:



Από το παραπάνω γράφημα φαίνονται πιο καθαρά οι εξαρτήσεις που αφορούν τον υπολογισμό των στοιχείων.

Όπως και με τη μέθοδο Jacobi, ο αρχικός πίνακας χωρίζεται σε μικρότερους, ίσου μεγέθους, υποπίνακες με βάση τον αριθμό των διεργασιών που διαθέτουμε και αν χρειαστεί προστίθεται padding όπως εξηγήθηκε παραπάνω. Ύστερα, η root διεργασία διαχωρίζει τον πίνακα με τα δεδομένα της θερμότητας σε υποπίνακες και στέλνει σε όλες τις επιμέρους διεργασίες τους υποπίνακες current και previous που τους αντιστοιχούν. Στην συνέχεια κάθε διεργασία επικοινωνεί με τις γειτονικές της, εφόσον αυτές υπάρχουν, για να ανταλλάξει (ασύγχρονα) με τις εξής πληροφορίες:

- Στον βόρειο γείτονα (north) στέλνει την πρώτη της γραμμή από τον πίνακα previous, που περιέχει τις προηγούμενες τιμές και λαμβάνει την τελευταία γραμμή του πίνακα current αυτού, που περιέχει τις νεότερες τιμές που υπολογίστηκαν
- Στον δυτικό γείτονα (west) στέλνει την πρώτη της στήλη από τον πίνακα previous και λαμβάνει την τελευταία στήλη από τον πίνακα current αυτού
- Από τον ανατολικό γείτονα (east) λαμβάνει την πρώτη στήλη από τον πίνακα previous
- Από τον νότιο γείτονα (south) λαμβάνει την πρώτη γραμμή από τον πίνακα previous

Με τους γείτονες που βρίσκονται ανατολικά και νότια ανταλλάσσονται, για την ακρίβεια στην αρχή λαμβάνονται, μόνο στοιχεία των πινάκων previous αφού ο υπολογισμός της παρούσας κατάσταση ξεκινάει από πάνω και αριστερά (δυτικοί και βόρειοι γείτονες), οπότε κάθε φορά τα νέα στοιχεία λαμβάνονται από εκείνους του γείτονες, όπως εξηγήθηκε και παραπάνω.

Οι διεργασίες αφου ολοκληρώσουν τις κατάλληλες επικοινωνίες και λάβουν όλα τα στοιχεία τα οποία χρειάζονται για τους υπολογισμούς τους, κάνουν τον υπολογισμό (καλείται η συνάρτηση που υλοποιεί τον αλγόριθμο Gauss-Seidel SOR). Έπειτα, θα πρέπει να στείλουμε όμως τις παλιές τιμές, των πινάκων previous, στα blocks που βρίσκονται στα νότια και ανατολικά ώστε να μπορούν να είναι ενημερωμένες σωστά για το επόμενο στάδιο των υπολογισμών.

Τέλος, όπως και προηγουμένως, όταν ο αλγόριθμος εκτελέσει και τις 256 επαναλήψεις ή συγκλίνει, η διεργασία root μαζεύει όλους τους υποπίνακες από τις υπόλοιπες διεργασίες και τους συνενώνει στον αρχικό πίνακα.

Παράλληλο πρόγραμμα με τη βοήθεια της βιβλιοθήκης MPI

Μέθοδος Jacobi

Παρακάτω παρατίθενται μόνο τα τμήματα του κώδικα που προσθέσαμε ή αλλάξαμε στον mpi_skeleton.c που μας δόθηκε για να παραλληλοποιηθεί το πρόγραμμα Jacobi και δίνεται μια σύντομη εξήγηση για καθένα από αυτά.

Αρχικοποιούμε την βοηθητική μεταβλητή u_start που ορίσαμε ως το 1ο στοιχείο του πίνακα.

```
u_start = &(U[0][0]);
```

Η root διεργασία στέλνει σε όλες τις υπόλοιπες διεργασίες τους πίνακες u_current και u_previous ώστε να μπορούν εκείνες να εκτελέσουν το τμήμα κώδικα που τους αντιστοιχεί.

```
/*Make sure u_current and u_previous are both initialized*/
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);
```

Δημιουργούμε τύπους δεδομένων (datatypes) για τις γραμμές και τις στήλες ώστε να μπορεί να γίνεται εύκολα η επικοινωνία και η ανταλλαγή δεδομένων μεταξύ των διεργασιών.

```
//----Define datatypes or allocate buffers for message passing----//
MPI_Datatype column;
MPI_Type_vector(local[0], 1, local[1]+2, MPI_DOUBLE, &column);
MPI_Type_commit(&column);



MPI_Datatype row;
MPI_Type_contiguous(local[1], MPI_DOUBLE, &row);
MPI_Type_commit(&row);
```

Αρχικοποιούμε τους 4 γείτονες της εν λόγω διεργασίας (north, south, east, west). Σε περίπτωση που δεν υπάρχει σε κάποιο σύνορο γείτονας τότε στην συγκεκριμένη μεταβλητή θα αποθηκευτεί τιμή MPI_PROC_NULL.

```
int north, south, east, west;

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

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

```
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);

//In case of a non-existing neighbour MPI_Cart_shift returns MPI_PROC_NULL
```

Αρχικοποιούμε τα όρια στα οποία θα τρέξει ο αλγόριθμος Jacobi για κάθε διεργασία. Συγκεκριμένα υπάρχουν 2 πράγματα που πρέπει να προσέξουμε, αν υπάρχει γείτονας από εκείνο το σύνορο, και αν ο υποπίνακας έχει padding.

1. Για κάθε σύνορο ελέγχουμε αν υπάρχει γείτονας με το αν η τιμή της μεταβλητής είναι MPI_PROC_NULL. Αν δεν έχουμε γείτονα παρατηρείται ότι τα i_min είναι κατα ένα μεγαλύτερα ενώ τα i_max κατά ένα μικρότερα ώστε να μην συμπεριληφθεί στον υπολογισμό η μη υπάρχουσα γειτονική στήλη/γραμμη.
2. Από την άλλη padding μπορούμε να παρατηρήσουμε όπως αναφέρθηκε και παραπάνω στα σύνορα east και south, επομένως για αυτά τα σύνορα αφαιρούμε από το μέγεθος του υποπίνακα το padding με την πράξη (global_padded - global), η οποία αν δεν υπάρχει Padding βγάζει αποτέλεσμα 0, επομένως δεν επηρεάζει το αποτέλεσμα.

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

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

if (north == MPI_PROC_NULL) {
       i_min = 2;
}
else {
       i_min = 1;
}

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

if (west == MPI_PROC_NULL) {
       j_min = 2;
}
else {
```

```
        j_min = 1;
 }

 if (east == MPI_PROC_NULL) {
        j_max = local[1] - (global_padded[1] - global[1]);
 }
 else {
        j_max = local[1] + 1;
 }
```

Αρχικοποιούμε πριν μπούμε στο for loop τα requests και τον counter που θα χρησιμοποιηθούν στα μηνύματα Isend και Irecv παρακάτω.

```
MPI_Request requests[8];
int counter;
```

Ο κώδικας από εδώ και κάτω είναι μέσα στο for loop του t, δηλαδή τρέχει μέχρι να συγκλίνει ή να φτάσουμε τις 256 επαναλήψεις.
Η κάθε διεργασία στέλνει στις γειτονικές της τις κατάλληλες γραμμές/στήλες του υποπίνακα της που τους αναλογούν και αντίστοιχα λαμβάνει από τις διεργασίες γείτονες τις γραμμές/στήλες που χρειάζεται για να κάνει τους υπολογισμούς της. Στην συνέχεια αφού περιμένουμε να τελειώσει η επικοινωνία με όλους τους γείτονες (με το MPI_Waitall) εκτελείται η Jacobi και υπολογίζεται το computational time.

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

/*Compute and Communicate*/
/*Add appropriate timers for computation*/

//communicate with neighbours - send and receive all needed rows and
columns

counter = 0;
if (north != MPI_PROC_NULL) {
            MPI_Isend(&u_previous[1][1], 1, row, north, rank, CART_COMM,
&requests[counter++]);
             MPI_Irecv(&u_previous[0][1], 1, row, north, north, CART_COMM,
&requests[counter++]);
}

if (south != MPI_PROC_NULL) {
```

```
                MPI_Isend(&u_previous[local[0]][1],  1,  row,  south,  rank,
CART_COMM, &requests[counter++]);
                MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south, south,
CART_COMM, &requests[counter++]);
}

if (west != MPI_PROC_NULL) {
            MPI_Isend(&u_previous[1][1], 1, column, west, rank, CART_COMM,
&requests[counter++]);
            MPI_Irecv(&u_previous[1][0], 1, column, west, west, CART_COMM,
&requests[counter++]);
}

if (east != MPI_PROC_NULL) {
                MPI_Isend(&u_previous[1][local[1]], 1, column, east, rank,
CART_COMM, &requests[counter++]);
               MPI_Irecv(&u_previous[1][local[1]+1], 1, column, east, east,
CART_COMM, &requests[counter++]);
}

MPI_Waitall(counter, requests, MPI_STATUSES_IGNORE);


gettimeofday(&tcs, NULL);

Jacobi(u_previous, u_current, i_min, i_max, j_min, j_max);

gettimeofday(&tcf,NULL);


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

Ελέγχουμε για σύγκλιση και υπολογίζουμε το convergence time.

```
#ifdef TEST_CONV
if (t%C==0) {
    //*************TODO**************//
    /*Test convergence*/

    gettimeofday(&tcvs, NULL);
    converged = converge(u_previous, u_current, local[0], local[1]);
```

```
    gettimeofday(&tcvf, NULL);

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

}
#endif
```

Στην συνέχεια και αφού έχουν τελειώσει τον υπολογισμό και την σύγκλιση όλες οι διεργασίες υπολογίζεται ο συνολικός χρόνος για τον οποίο τρέχει το πρόγραμμα και όλοι οι χρόνοι που υπολογίσαμε για βρίσκονται στις επιμέρους διεργασίες προστίθενται και οι τιμές τους επιστρέφουν στην αρχική root διεργασία με τις μεθόδους MPI_Reduce που φαίνονται παρακάτω

```
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);
```

Τέλος, η διεργασία root μαζεύει τα αποτελέσματα για τους υποπίνακες από όλες τις διεργασίες και τα συνενώνει ξανά σε έναν μεγάλο αρχικό πίνακα.

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

Μέθοδος Gauss-Seidel SOR

Παρακάτω παρατίθενται μόνο τα τμήματα του κώδικα που προσθέσαμε ή αλλάξαμε στον mpi_skeleton.c που μας δόθηκε για να παραλληλοποιηθεί το πρόγραμμα Gauss-Seidel SOR και δίνεται μια σύντομη εξήγηση για καθένα από αυτά.

Καθώς μεγάλο οι αρχικοποιήσεις και οι δομές οι οποίες χρησιμοποιήθηκαν είναι ίδιες με αυτές για το πρόγραμμα Jacobi, παρατίθεται και σχολιάζεται αναλυτικά το κομμάτι με τον κώδικα για την επικοινωνία των διεργασιών το οποίο διαφέρει.

```
    MPI_Request requestsfirst[6];
    MPI_Request requestslast[2];
    MPI_Status statusfirst[6];
    MPI_Status statuslast[2];
    int first = 0;
    int last = 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


        swap = u_previous;
        u_previous = u_current;
        u_current = swap;
        first = 0;
        last = 0;


        /*Compute and Communicate*/
        /*Add appropriate timers for computation*/
         //communicate with neighbours - send and receive all needed rows
and columns


        if (north != MPI_PROC_NULL) {
                        MPI_Isend(&u_previous[1][1], 1, row, north, rank,
MPI_COMM_WORLD, &requestsfirst[first++]);
                        MPI_Irecv(&u_current[0][1], 1, row, north, north,
MPI_COMM_WORLD, &requestsfirst[first++]);
        }

        if (south != MPI_PROC_NULL) {
                      MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south,
south, MPI_COMM_WORLD, &requestsfirst[first++]);
        }

        if (west != MPI_PROC_NULL) {
```

```
                    MPI_Isend(&u_previous[1][1], 1, column, west, rank,
MPI_COMM_WORLD, &requestsfirst[first++]);
                     MPI_Irecv(&u_current[1][0], 1, column, west, west,
MPI_COMM_WORLD, &requestsfirst[first++]);
        }

        if (east != MPI_PROC_NULL) {
                   MPI_Irecv(&u_previous[1][local[1]+1], 1, column, east,
east, MPI_COMM_WORLD, &requestsfirst[first++]);
        }

        MPI_Waitall(first, requestsfirst, statusfirst);


        /*Compute and Communicate*/

        /*Add appropriate timers for computation*/


        gettimeofday(&tcs, NULL);

           GaussSeidel(u_previous, u_current, i_min, i_max, j_min, j_max,
omega);

        gettimeofday(&tcf,NULL);

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

        //counter = 0;
        if (south != MPI_PROC_NULL) {
                 MPI_Isend(&u_previous[local[0]][1], 1, row, south, rank,
MPI_COMM_WORLD, &requestslast[last++]);
        }

        if (east != MPI_PROC_NULL) {
                MPI_Isend(&u_previous[1][local[1]], 1, column, east, rank,
MPI_COMM_WORLD, &requestslast[last++]);
        }

        MPI_Waitall(last, requestslast, statuslast);
```

Στο παραπάνω κομμάτι κώδικα φαίνονται οι αποστολές και οι παραλαβές δεδομένων από τη μία διεργασία στην άλλη, οι οποίες ακολουθούν το μοντέλο το οποίο περιγράψαμε παραπάνω. Αυτό το οποίο αξίζει να σημειωθεί είναι ότι για την υλοποίηση της επικοινωνίας χρησιμοποιήθηκαν non-blocking συναρτήσεις ώστε τα εκάστοτε στοιχεία να μπορούν να αποσταλούν μόλις είναι έτοιμα χωρίς να χρειάζεται περαιτέρω αναμονή.

Ακόμα, αφού έχει γίνει η απαραίτητη ανταλλαγή στοιχείων χρησιμοποιήθηκε ένα MPI_Waitall για το συγχρονισμό των διεργασιών και αφού έγινε ο υπολογισμός, με τη μέθοδο Gauss-Seidel SOR, γίνονται και οι τελευταίες αποστολές στοιχείων.

Αρχεία sh και makefiles που χρησιμοποιήθηκαν

Μέθοδος Jacobi

Για την μεταγλώττιση του αρχείου χρησιμοποιήθηκε το εξής Makefile

```
CC=mpicc
CFLAGS=-O3 -lm -Wall
RES=-DPRINT_RESULTS
CONV=-DTEST_CONV

all: jacobi

jacobi: Jacobi_mpi.c utils.c
    $(CC) $(CFLAGS) $(RES) $(CONV) Jacobi_mpi.c utils.c -o jacobi

clean:
    rm jacobi
```

Σημείωση:
Για τις μετρήσεις που τρέξαμε χωρίς convergence αφαιρέθηκε το CONV= –DTEST_CONV και αντίστοιχα το $(CONV) στην εντολή της μεταγλώττισης

Το αρχείο make_on_queue_jacobi.sh που χρησιμοποιήθηκε για την μεταγλώτισση στην ουρά parlab φαίνεται παρακάτω

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_jacobi

## Output and error files
```

```
#PBS -o make_jacobi.out
#PBS -e make_jacobi.err

## Limit memory, runtime etc.
#PBS -l walltime=00:10:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=1:ppn=1


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/Jacobi
make
```

Δημιουργήθηκαν 2 αρχεία run_on_queue για την μέθοδο Jacobi. Ένα για τις μετρήσεις με σύγκλιση, το run_on_queue_jacobi_convergence.sh, και ένα για τις μετρήσεις χωρίς σύγκλιση, το run_on_queue_jacobi.sh, γιατί θέλαμε να τρέξουμε τον κώδικα για διαφορετικά μεγέθη πινάκων και διαφορετικό αριθμό διεργασιών

Παρακάτω παρατίθεται το αρχείο run_on_queue_jacobi.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_jacobi

## Output and error files
#PBS -o run_jacobi.out
#PBS -e run_jacobi.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/Jacobi

for j in 2048 4096 6144
do
    for i in 1 2 3
        do
```

```
            echo "Array Length ${j}"
            echo "Iteration ${i}"
          mpirun -np 1 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 1 1
          mpirun -np 2 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 2 1
          mpirun -np 4 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 2 2
          mpirun -np 8 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 4 2
          mpirun -np 16 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 4 4
          mpirun -np 32 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 8 4
          mpirun -np 64 --map-by node --mca btl self,tcp ./jacobi ${j} ${j} 8 8
        done
done
```

Παρακάτω παρατίθεται το run_on_queue_jacobi_convergence.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_jacobi

## Output and error files
#PBS -o run_jacobi_convergence.out
#PBS -e run_jacobi_convergence.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/Jacobi

for i in 1 2 3
do
    echo "Iteration ${i}"
    mpirun -np 64 --map-by node --mca btl self,tcp ./jacobi 1024 1024 8 8
done
```

Μέθοδος Gauss-Seidel SOR

Για την μεταγλώττιση του αρχείου χρησιμοποιήθηκε το εξής Makefile

```
CC=mpicc
CFLAGS=-O3 -lm -Wall
RES=-DPRINT_RESULTS
CONV=-DTEST_CONV

all: gssor

gssor: GSSOR_mpi.c utils.c
    $(CC) $(CFLAGS) $(RES) $(CONV) GSSOR_mpi.c utils.c -o gssor

clean:
    rm gssor
```

Σημείωση:

Για τις μετρήσεις που τρέξαμε χωρίς convergence αφαιρέθηκε το CONV= –DTEST_CONV και αντίστοιχα το $(CONV) στην εντολή της μεταγλώττισης

Το αρχείο make_on_queue_gssor.sh που χρησιμοποιήθηκε για την μεταγλώτισση στην ουρά parlab φαίνεται παρακάτω:

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_gssor

## Output and error files
#PBS -o make_gssor.out
#PBS -e make_gssor.err

## Limit memory, runtime etc.
#PBS -l walltime=00:10:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=1:ppn=1


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/GSSOR
make
```

Δημιουργήθηκαν 2 αρχεία run_on_queue για την μέθοδο Gauss-Seidel SOR. Ένα για τις μετρήσεις με σύγκλιση, το run_on_queue_gssor_convergence.sh, και ένα για τις μετρήσεις χωρίς σύγκλιση, το run_on_queue_gssor.sh, γιατί θέλαμε να τρέξουμε τον κώδικα για διαφορετικά μεγέθη πινάκων και διαφορετικό αριθμό διεργασιών

Παρακάτω παρατίθεται το αρχείο run_on_queue_gssor.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_gssor

## Output and error files
#PBS -o run_gssor.out
#PBS -e run_gssor.err

## Limit memory, runtime etc.
#PBS -l walltime=01:30:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/GSSOR

for j in 2048 4096 6144
do
    for i in 1 2 3
        do
            echo "Array Length ${j}"
            echo "Iteration ${i}"
            mpirun -np 1 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 1 1
          mpirun -np 2 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 2 1
          mpirun -np 4 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 2 2
          mpirun -np 8 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 4 2
          mpirun -np 16 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 4 4
          mpirun -np 32 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 8 4
          mpirun -np 64 --map-by node --mca btl self,tcp ./gssor ${j} ${j} 8 8
        done
done
```

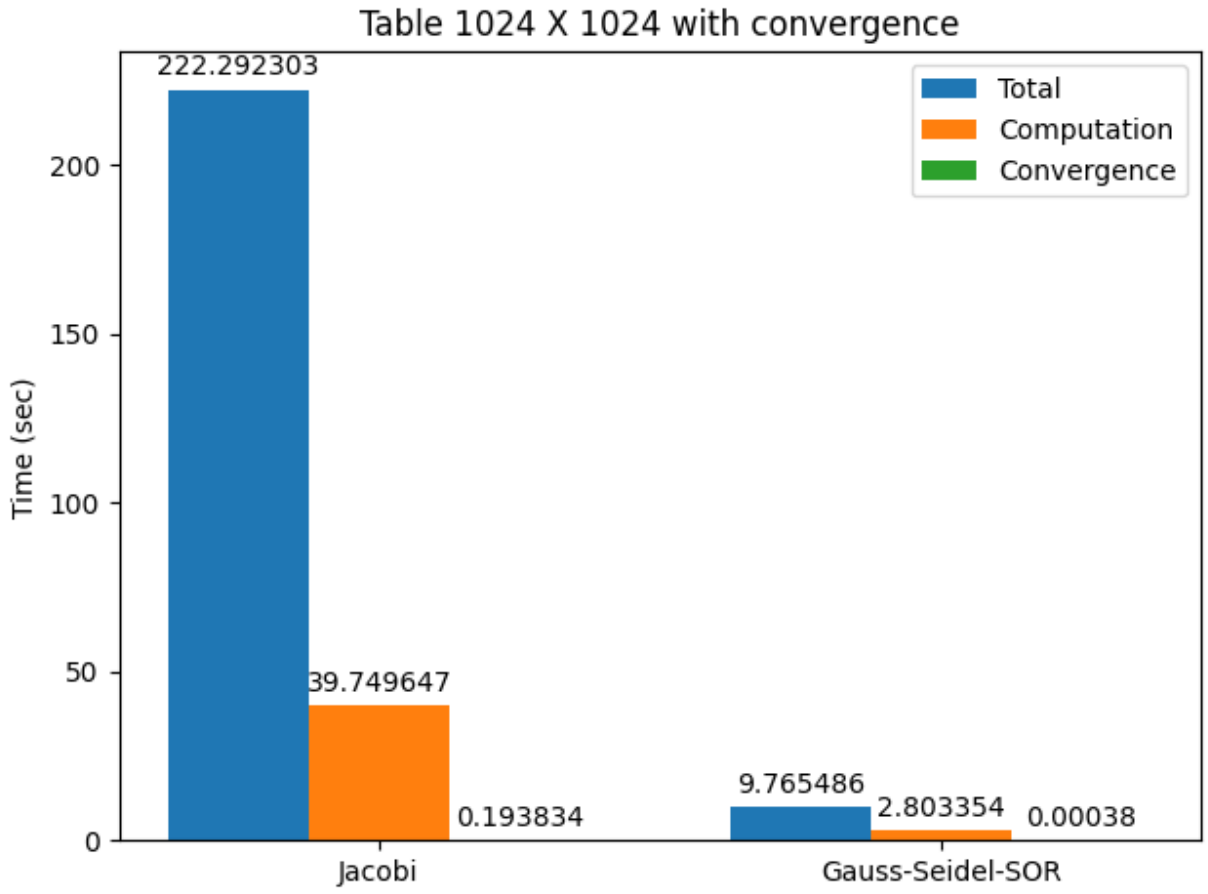
Παρακάτω παρατίθεται το run_on_queue_jacobi_convergence.sh

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_gssor

## Output and error files
#PBS -o run_gssor_convergence.out
#PBS -e run_gssor_convergence.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

## How many nodes:processors_per_node should we get?
#PBS -l nodes=8:ppn=8


module load openmpi/1.8.3
cd /home/parallel/parlab20/a3/solution/GSSOR

for i in 1 2 3
do
    echo "Iteration ${i}"
    mpirun -np 64 --map-by node --mca btl self,tcp ./gssor 1024 1024 8 8
done
```

Αποτελέσματα μετρήσεων επίδοσης βάσει του σεναρίου μετρήσεων

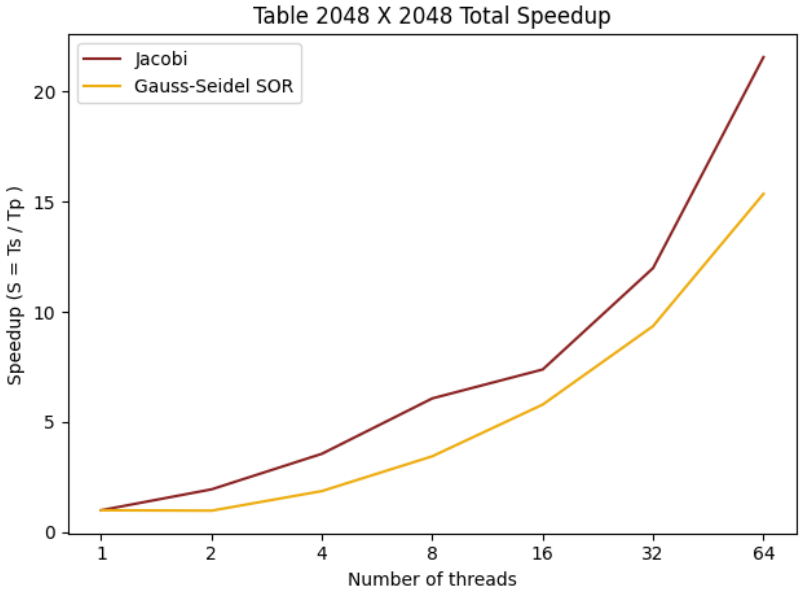
Με έλεγχο σύγκλισης

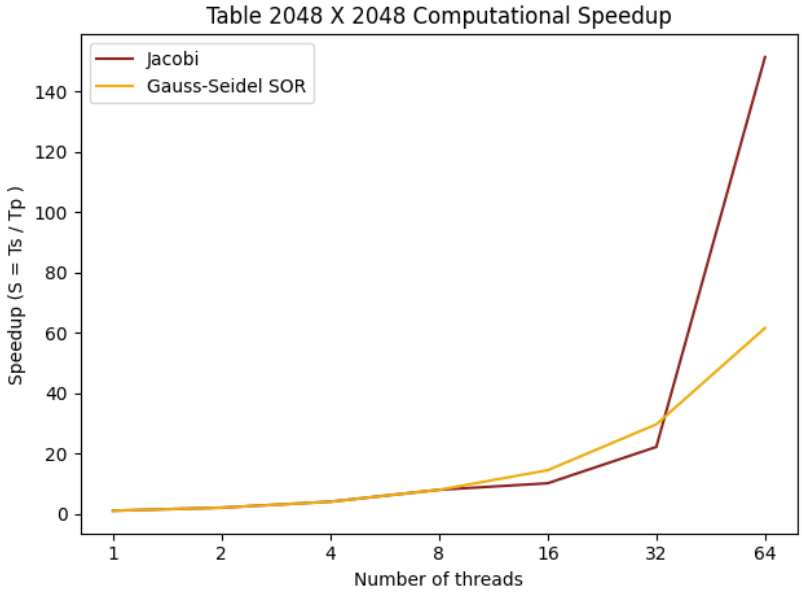
Παρακάτω φαίνεται το διάγραμμα χρόνων για πίνακα 1024 X 1024 με έλεγχο σύγκλισης για 64 MPI διεργασίες και για τις δυό μας μεθόδους

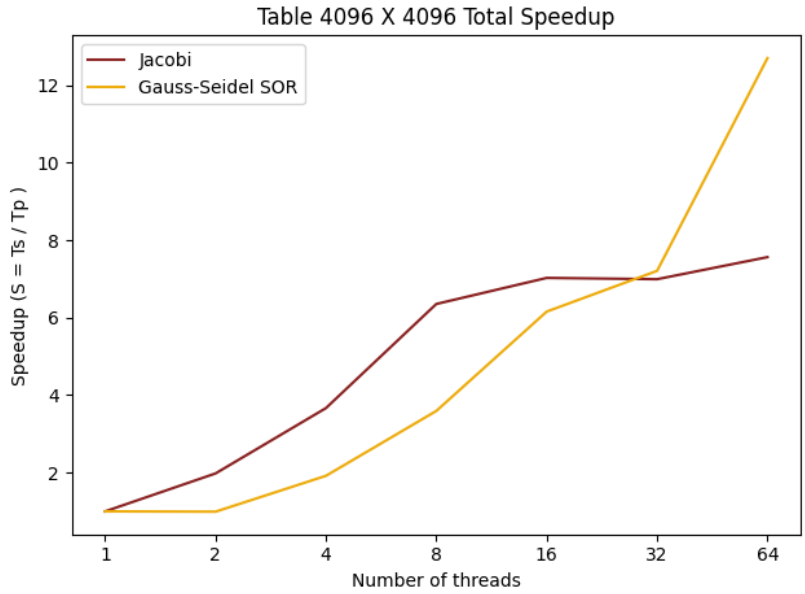


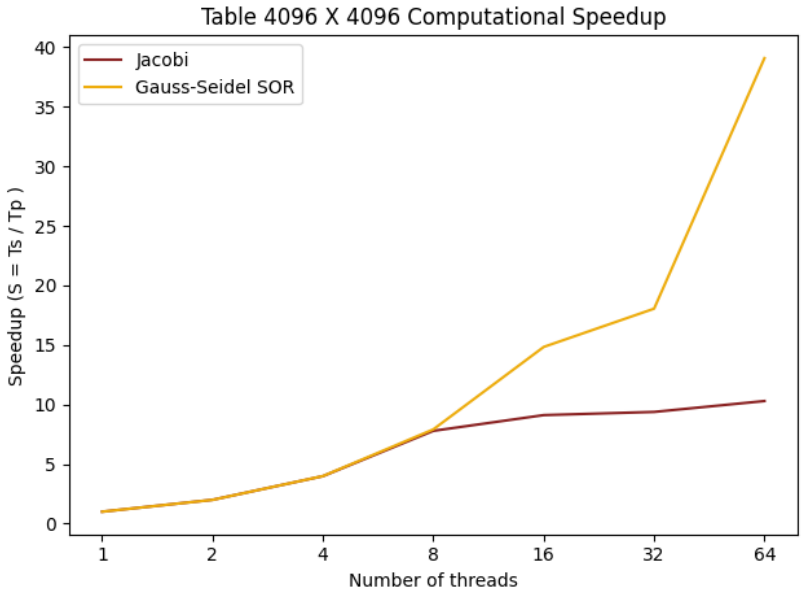
Χωρίς έλεγχο σύγκλισης

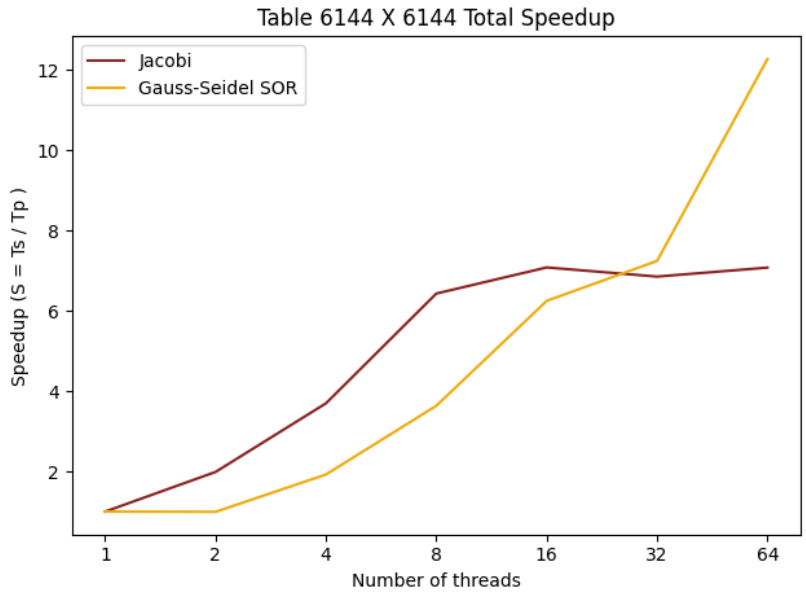
Αρχικά παρατίθενται τα διαγράμματα με τα Speedup που ζητούνταν

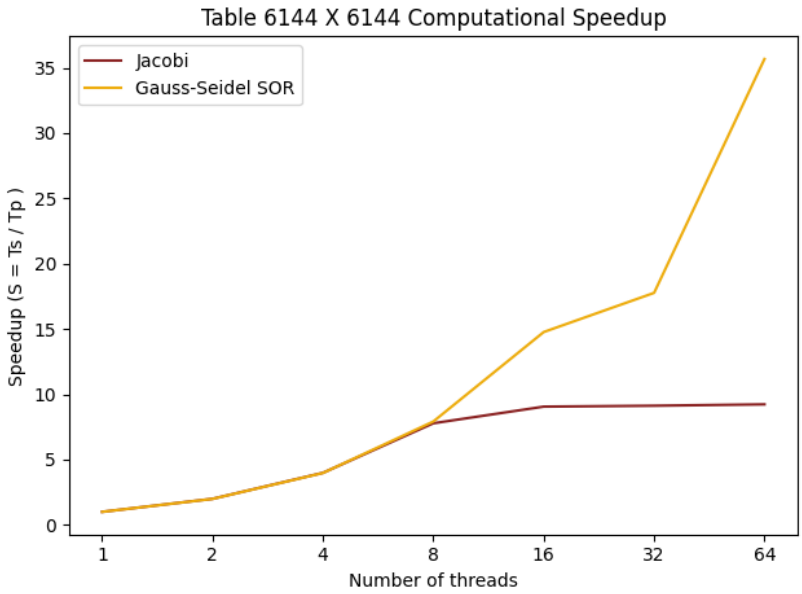






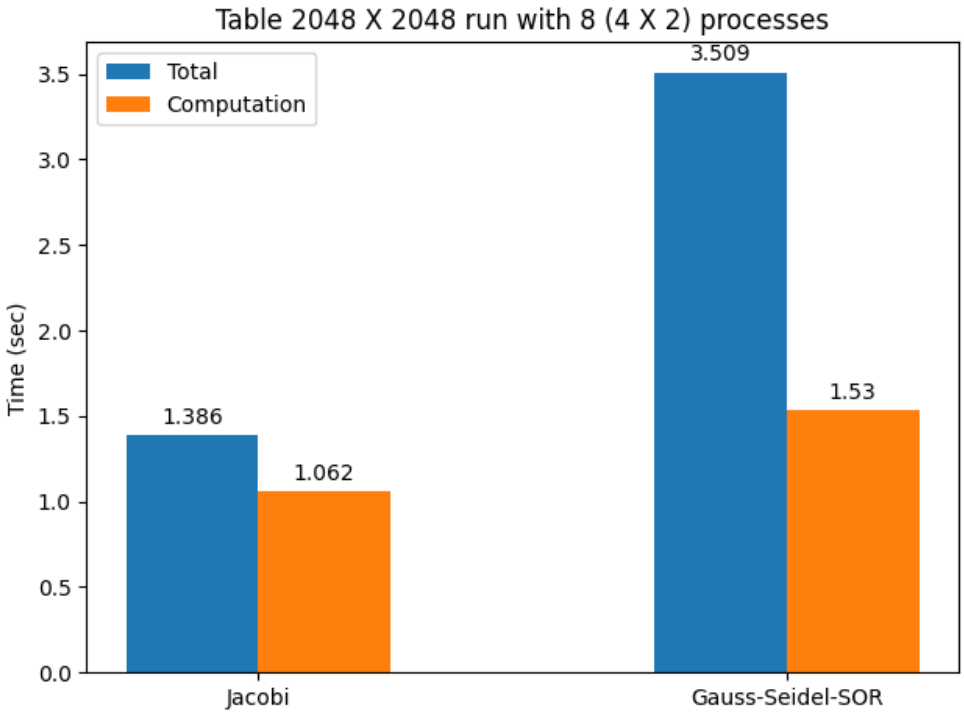






Στην συνέχεια φαίνονται τα διαγράμματα με μπάρες του χρόνου για μεγέθη πίνακα 2048, 4096, 6144 και μόνο για 8, 16, 32 και 64 MPI διεργασίες. Σημειώνεται ότι τα διαγράμματα παρουσιάζονται με την σειρά για 8, 16, 32, 64 MPI διεργασίες, κάτι το οποίο φαίνεται και στον τίτλο τους.

1. 2048 X 2048



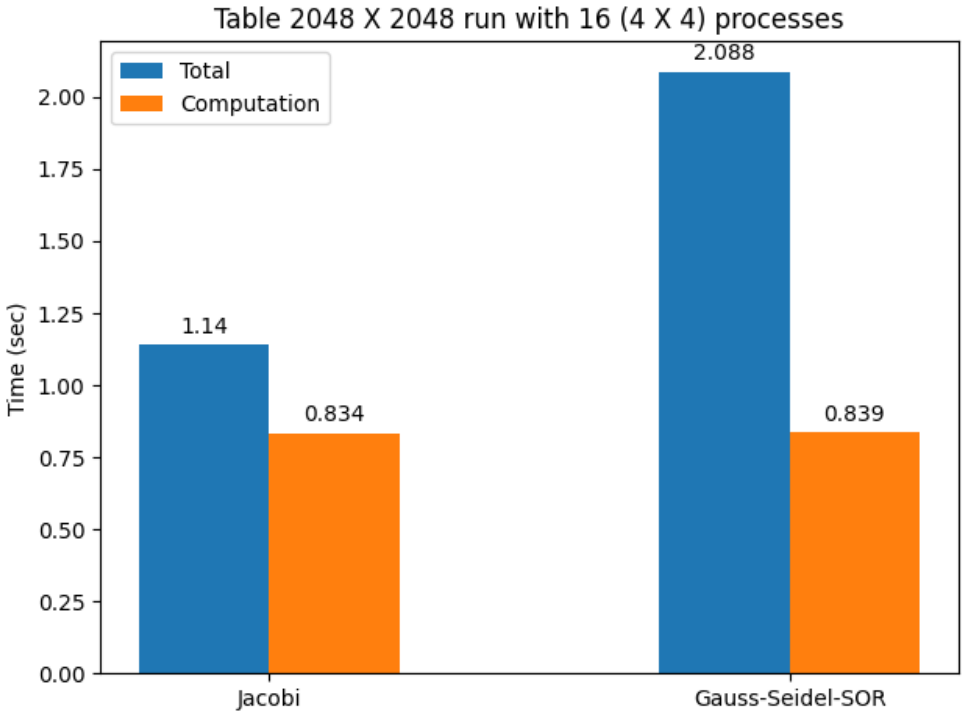


Table 2048 X 2048 run with 16 (4 X 4) processes
Total
Computation
Time (sec)
2.088
1.14
0.834
0.839
Jacobi
Gauss-Seidel-SOR

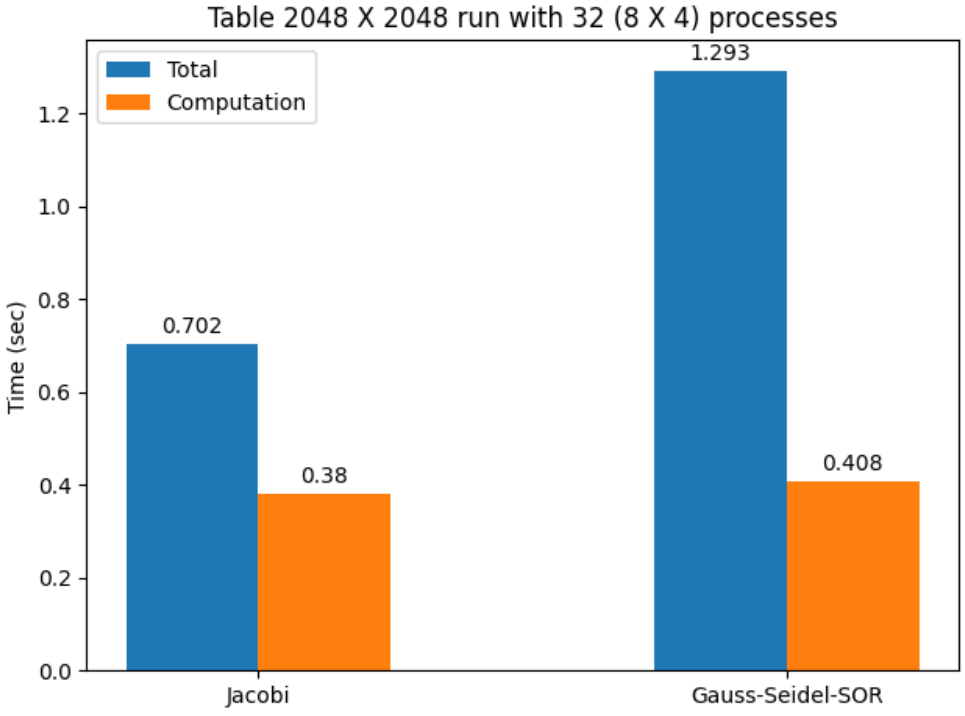


Table 2048 X 2048 run with 32 (8 X 4) processes
Total
Computation
Time (sec)
1.293
0.702
0.38
0.408
Jacobi
Gauss-Seidel-SOR

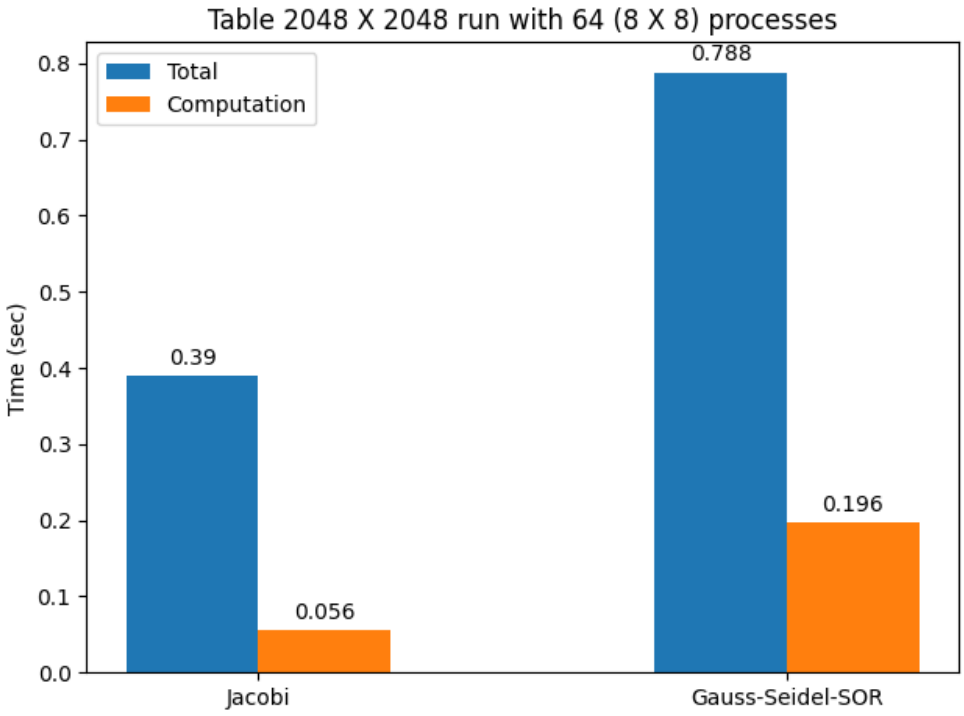


Table 2048 X 2048 run with 64 (8 X 8) processes
Total
Computation
Time (sec)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.39
0.056
0.788
0.196
Jacobi
Gauss-Seidel-SOR

2. 4096 X 4096

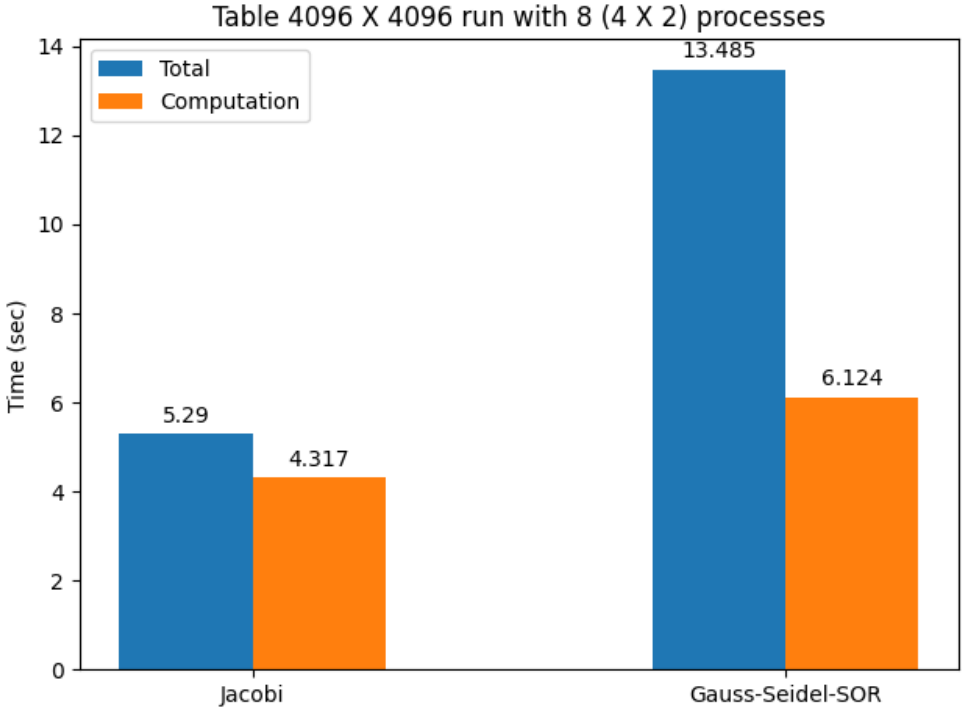


Table 4096 X 4096 run with 8 (4 X 2) processes
Total
Computation
Time (sec)
0
2
4
6
8
10
12
14
5.29
4.317
13.485
6.124
Jacobi
Gauss-Seidel-SOR

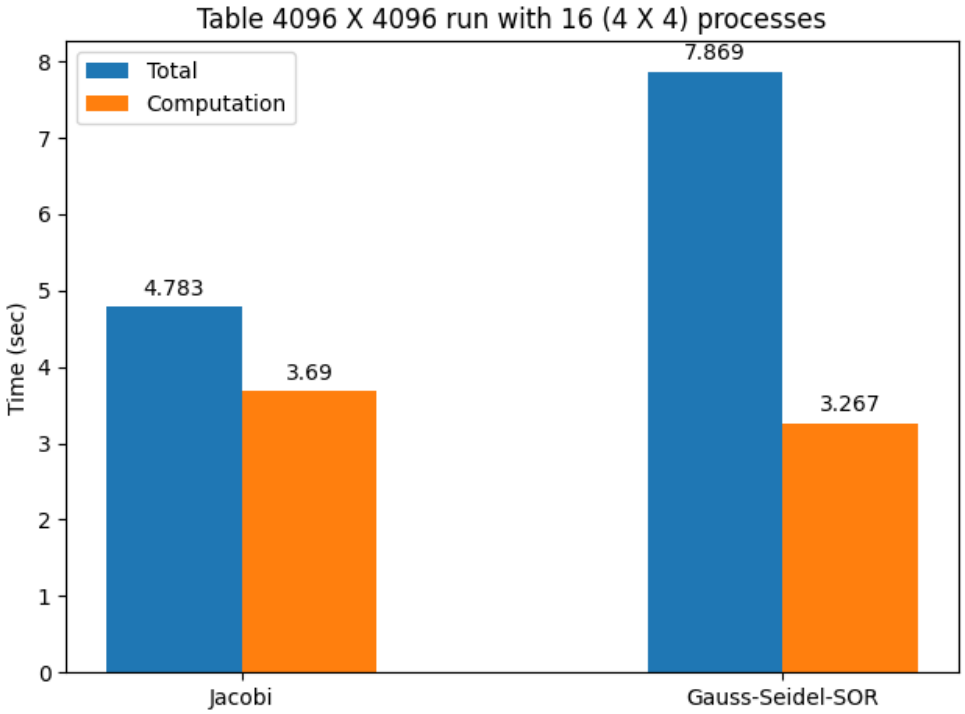


Table 4096 X 4096 run with 16 (4 X 4) processes
Total
Computation
Time (sec)
0
1
2
3
4
5
6
7
8
4.783
3.69
7.869
3.267
Jacobi
Gauss-Seidel-SOR

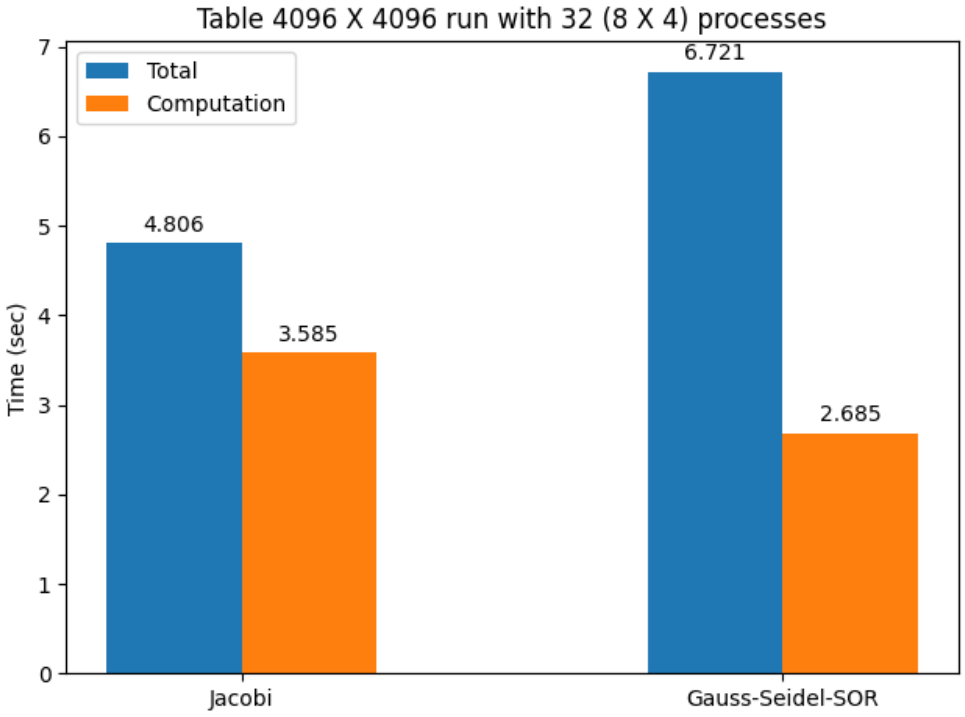


Table 4096 X 4096 run with 32 (8 X 4) processes
Total
Computation
Time (sec)
0
1
2
3
4
5
6
7
4.806
3.585
6.721
2.685
Jacobi
Gauss-Seidel-SOR

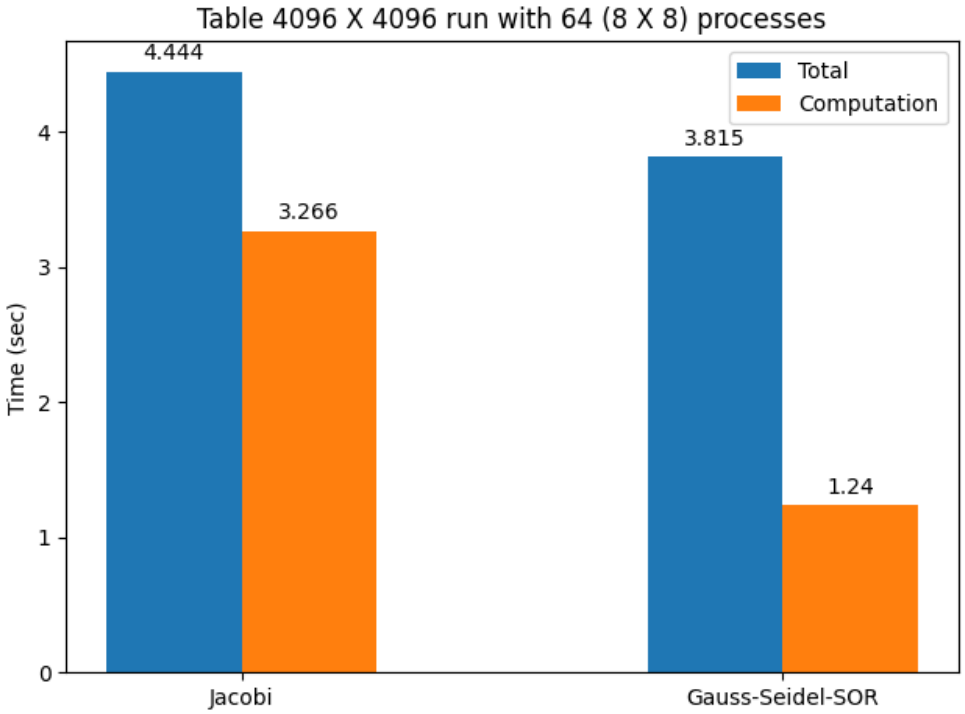


Table 4096 X 4096 run with 64 (8 X 8) processes
Total
Computation
4.444
3.266
3.815
1.24
Time (sec)
0
1
2
3
4
Jacobi
Gauss-Seidel-SOR

3. 6144 X 6144

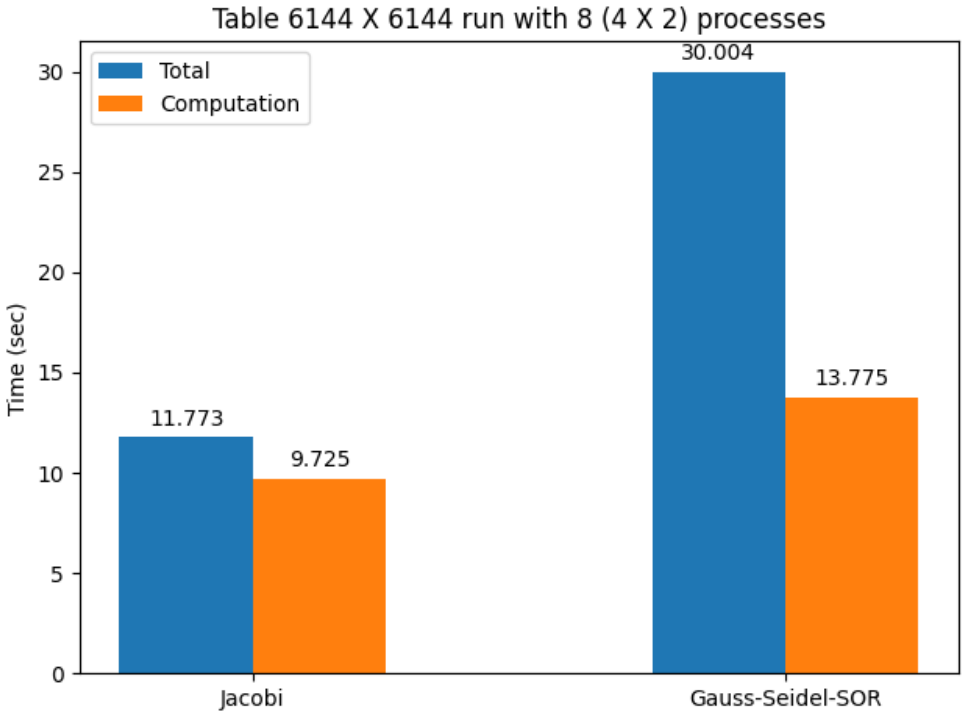


Table 6144 X 6144 run with 8 (4 X 2) processes
Total
Computation
11.773
9.725
30.004
13.775
Time (sec)
0
5
10
15
20
25
30
Jacobi
Gauss-Seidel-SOR

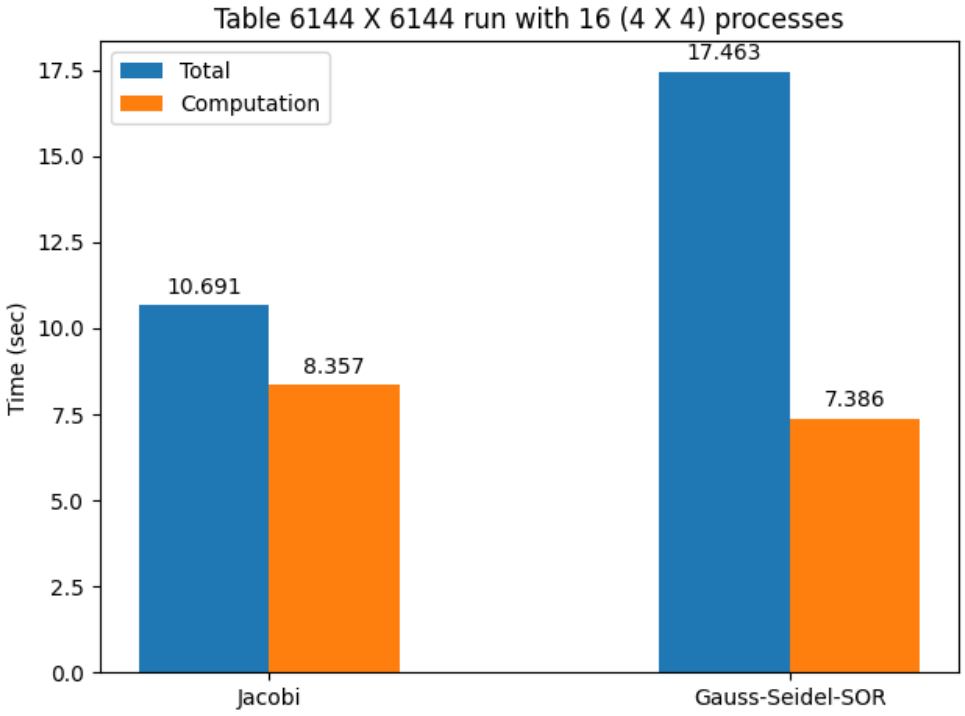


Table 6144 X 6144 run with 16 (4 X 4) processes
Total
Computation
Time (sec)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10.691
8.357
17.463
7.386
Jacobi
Gauss-Seidel-SOR

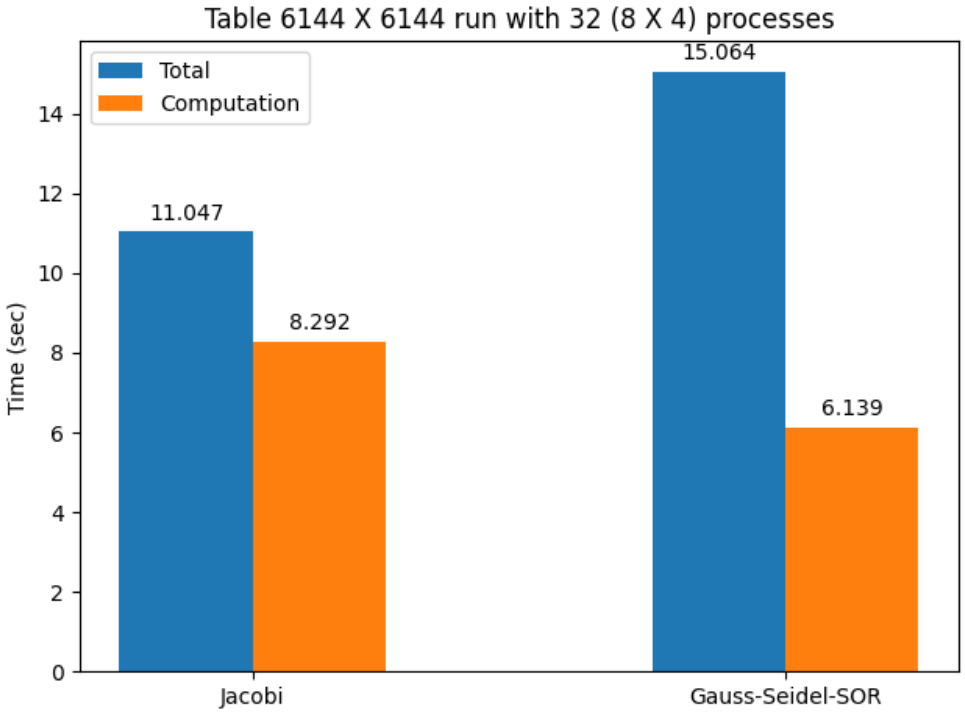
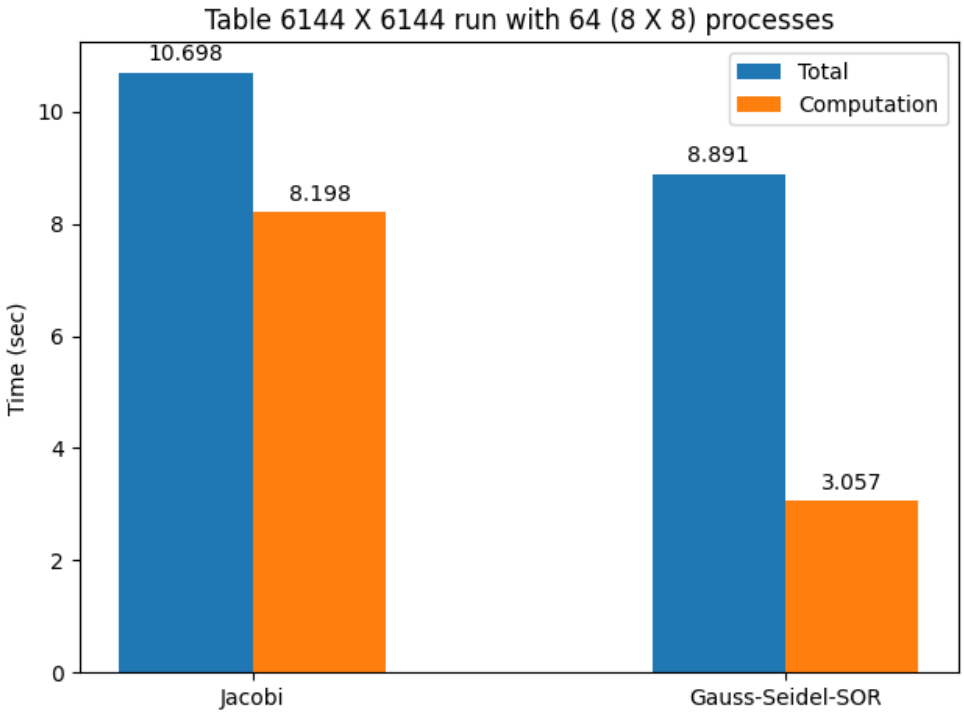


Table 6144 X 6144 run with 32 (8 X 4) processes
Total
Computation
Time (sec)
14
12
10
8
6
4
2
0
11.047
8.292
15.064
6.139
Jacobi
Gauss-Seidel-SOR



Γενικά Συμπεράσματα και Παρατηρήσεις

Με έλεγχο σύγκλισης

Παρατηρείται ότι προτιμότερος από τους δύο αλγορίθμους για ένα σύστημα κατανεμημένης μνήμης είναι με διαφορά ο Gauss-Siedel SOR, αφού έχει εμφανώς καλύτερο χρόνο σε σχέση με τον Jacobi, γεγονός αναμενόμενο αφού γνωρίζαμε ότι ο ο Gauss-Siedel SOR συγκλίνει πολύ πιο γρήγορα. Συγκεκριμένα, ο Jacobi χρειάζεται 798.201 επαναλήψεις για να συγκλίνει, ενώ ο Gauss-Siedel SOR συγκλίνει μετά από μόλις 14.801 επαναλήψεις. Η διαφορά είναι αισθητή και αποτυπώνεται έντονα και στους τελικούς χρόνους, για αυτό και δεν έχει νόημα να συγκριθούν οι επιμέρους χρόνοι σύγκλισης, υπολογισμού και συνολικός, αφού η διαφορα στις επαναλήψεις είναι τόσο μεγάλη που είναι αυτή που καθορίζει το αποτέλεσμα. Αυτό που ίσως αξίζει να αναφερθεί είναι ότι και στις 2 μεθόδους ο μεγαλύτερος χρόνος είναι αυτός της επικοινωνίας μεταξύ των διεργασιών [(total time)-(computational time)].

Χωρίς έλεγχο σύγκλισης

Παρατηρείται ότι για μικρό αριθμό διεργασιών, μέχρι 8 ή και 16 διεργασίες τα speedup και για τους δύο αλγόριθμους κλιμακώνουν αρκετά καλα και με παρόμοιο τρόπο. Ωστόσο ο Jacobi για μεγαλύτερο αριθμό διεργασιών δεν παρουσιάζει αύξηση του speedup, γεγονός που προφανώς σημαίνει ότι ο χρόνος επικοινωνίας αυξάνεται τόσο που αντισταθμίζει την μείωση του χρόνου υπολογισμού, γεγονός που γίνεται περισσότερο αντιληπτό σε μεγάλα μεγέθη πίνακα. Από την άλλη ο αλγόριθμος Gauss-Siedel SOR παρουσιάζει πολύ καλύτερη κλιμάκωση για μεγαλύτερο αριθμό διεργασιών, με καλύτερη από όλες να είναι η κλιμάκωση του speedup από τις 32 στις 64 διεργασίες.

Όσον αφορά τους χρόνους εκτέλεσης και συνολικά (διαγράμματα μπαρών) για 8 διεργασίες και πάνω παρατηρείται ότι μικρότερους χρόνους στις περισσότερες περιπτώσεις έχει η μέθοδος Jacobi, παρόλο που αυτή εμφάνιζε χειρότερη κλιμάκωση speedup από την Gauss-Siedel SOR. Συγκεκριμένα παρατηρείται ότι ακόμα και στις περιπτώσεις που οι χρόνοι εκτέλεσης είναι παρόμοιοι για τις δύο μεθόδους ο συνολικός χρόνος για την Gauss-Siedel SOR είναι πάντα μεγαλύτερος, γεγονός που οφείλεται στην επικοινωνία των διεργασιών, η οποία είναι πιο σύνθετη στην μέθοδο Gauss-Siedel SOR και αυξάνει κατα πολύ τον συνολικό χρόνο. Για μικρά μεγέθη πίνακα, λοιπόν, η μέθοδος Jacobi έχει πάντα καλύτερους χρόνους, λόγω του μικρότερου κόστους επικοινωνίας. Ωστόσο παρατηρούμε ότι για τα μεγάλα μεγέθη πίνακα και για μεγάλο αριθμό διεργασιών (64 MPI διεργασίες) η αύξηση του speedup είναι τόσο μεγάλη για την Gauss-Siedel SOR, με αποτέλεσμα ο χρόνος εκτέλεσης να μειώνεται τόσο πολύ, που τελικά ο συνολικός χρόνος είναι μικρότερος σε σχέση με την Jacobi, παρόλο που ο χρόνος επικοινωνίας είναι αισθητά μεγαλύτερος.

Επομένως συμπεραίνει κανείς ότι στην περίπτωση που δεν έχουμε σύγκλιση η μέθοδος Jacobi δίνει γενικά καλύτερα αποτελέσματα, κυρίως λόγω της απλότητας της επικοινωνίας των διεργασιών της. Ωστόσο αν διαθέτουμε πόρους που μας επιτρέπουν την χρήση πολλών MPI διεργασιών και πραγματευόμαστε μεγάλα μεγέθη πινάκων θα προτιμηθεί η μέθοδος Gauss-Siedel SOR που δίνει πολύ πιο γρήγορους χρόνους εκτέλεσης (που ισοσταθμίζουν και την λίγο πιο σύνθετη και χρονοβόρα επικοινωνία των διεργασιών της).