Τελική Αναφορά 2ης Άσκησης

Βιτάλιος Σαλής - 03115751
Φίλιππος Μαλανδράκης - 03116200

Εισαγωγή

Στην άσκηση μας δίνονται 3 σειριακές μέθοδοι επίλυσης του προβλήματος διάδοσης της θερμότητας σε δισδιάστατο χωρίο. Ακόμη, έχουμε έναν mpi σκελετό παραλληλοποίησης, τον οποίο καλούμαστε να διαμορφώσουμε ανάλογα, με στόχο τη δημιουργία παράλληλων εκδόσεων των παραπάνω μεθόδων. Στη συνέχεια, πρέπει να λάβουμε μετρήσεις για διάφορους συνδυασμούς μεγεθών ταμπλό - processes και να κατασκευάσουμε συγκριτικά γραφήματα.

Υλοποίηση

Αρχικά, θα παραθέσουμε τα κοινά κομμάτια κώδικα και των 3 παράλληλων υλοποιήσεών μας. Σημειώνεται ότι αξιοποιήσαμε τις βοηθητικές δομές CART_COMM, global_block και local_block. Όλες οι εισαγωγές κώδικα έγιναν στα ενδεδειγμένα σημεία. Για το διαμοιρασμό του padded πίνακα U από το process με rank 0 στα υπόλοιπα processes εισάγαμε στο αρχείο mpi_skeleton.c τα εξής:

```
if(rank == 0) {
        for(i = 1; i < grid[0] * grid[1]; i++) {
                MPI_Send(&(U[0][0]) + scatteroffset[i], 1, global_block, i, i,
CART_COMM);
        }
        for(i = 0; i < local[0]; i++) {
            for(j = 0; j < local[1]; j++) {
                u_current[i + 1][j + 1] = U[i][j];
            }
        }

}
else {
    MPI_Recv(&(u_current[1][1]), 1, local_block, 0, rank, CART_COMM, &status);
}

copy2d(u_current, u_previous, local[0] + 2, local[1] + 2);
```

Η βοηθητική συνάρτηση copy2d ορίστηκε συμπληρωματικά στο utils.c ως εξής:

```
void copy2d(double **arr1, double **arr2, int dimX, int dimY) {
    int i, j;
    for(i = 0; i < dimX; i++) {
        for(j = 0; j < dimY; j++) {
            arr2[i][j] = arr1[i][j];
        }
    }
}
```

Ορίσαμε τα MPI_Datatypes row και column για την μεταφορά των ghost rows και columns αντίστοιχα ως εξής:

```
MPI_Datatype column, row;

MPI_Type_vector(local[0] + 1, 1, local[1] + 2, MPI_DOUBLE, &column);
MPI_Type_commit(&column);

MPI_Type_contiguous(local[1] + 2, MPI_DOUBLE, &row);
MPI_Type_commit(&row);
```

Για την εύρεση των γειτόνων του κάθε process στο καρτεσιανό grid χρησιμοποιήσαμε την συνάρτηση MPI_Cart_shift:

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

Για τον προσδιορισμό των iteration ranges του κάθε process αξιοποιήσαμε την ύπαρξη ή μη γειτόνων σε κάθε προσανατολισμό ως εξής:

```
int i_min,i_max,j_min,j_max;
if(north > -1)
    i_min = 1;
else
    i_min = 2;

if(south > -1)
    i_max = local[0] +1;
else
    i_max = local[0];

if(west > -1)
    j_min = 1;
else
    j_min = 2;

if(east > -1)
    j_max = local[1] + 1;
else
    j_max = local[1];
```

Για τον έλεγχο του convergence, αρχικά διαμορφώσαμε τη βοηθητική συνάρτηση converge(από τα utils.c, αλλάζοντας κατάλληλα και τα ορίσματά της στο utils.h) ως εξής:

```
int converge(double ** u_previous, double ** u_current, int Xm, int Ym, int X,
int Y) {
    int i,j;
    for (i=Xm;i<X;i++) {
        for (j=Ym;j<Y;j++) {
            if (fabs(u_current[i][j]-u_previous[i][j])>e) {
                return 0;
            }
        }
    }
    return 1;
}
```

Στη συνέχεια, εισάγαμε το εξής:

```
gettimeofday(&tconvs, NULL);
#ifdef TEST_CONV
if (t%C==0) {
    /*Test convergence*/
    converged = converge(u_previous, u_current, i_min, j_min, i_max, j_max);
    MPI_Allreduce(&converged, &global_converged, 1, MPI_INT,
        MPI_LAND, CART_COMM);
}
    #endif
gettimeofday(&tconvf, NULL);
tconv += (tconvf.tv_sec - tconvs.tv_sec) + (tconvf.tv_usec - tconvs.tv_usec) *
0.000001;
```

Όπως φαίνεται, έχουν οριστεί μεταβλητές και για τον υπολογισμό του συνολικού convergence time ανά process. Με το πέρας του computational core, υπολογίζουμε το μέγιστο convergence time που σημειώθηκε ως εξής:

```
MPI_Reduce(&tconv, &conv_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

Το μόνο που απομένει πλέον είναι η επανασυγκέντρωση των subdomains του padded πίνακα από όλα τα processes, πράγμα που υλοποιήσαμε ως εξής:

```
if(rank == 0) {
    for(i = 1; i < grid[0] * grid[1]; i++) {
        MPI_Recv(&(U[0][0]) + scatteroffset[i], 1, global_block, i, i,
CART_COMM, &status);
    }
    for(i = 0; i < local[0]; i++) {
        for(j = 0; j < local[1]; j++) {
            U[i][j] = u_current[i + 1][j + 1];
        }
    }
}
else {
    MPI_Send(&(u_current[1][1]), 1, local_block, 0, rank, CART_COMM);
}
```

Τώρα θα καταπιαστούμε με τους διαφορετικούς τρόπους υλοποίησης της ανταλλαγής των ghost cells στο computational core ανά παράλληλη έκδοση.

Jacobi

Η συνάρτηση Jacobi απαιτεί να υπάρχουν ***πριν*** την κλήση της τα ghost cells από τους πίνακες u_previous. Το κάνουμε ως εξής:

```
MPI_Request requests[8];
int requests_cnt = 0;

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

```
requests_cnt = 0;
if(north > -1) {
    MPI_Irecv(&(u_previous[0][0]), 1, row, north, north * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_previous[1][0]), 1, row, north, rank * 10 + north, CART_COMM,
&requests[requests_cnt++]);
}
if(south > -1) {
    MPI_Irecv(&(u_previous[local[0] + 1][0]), 1, row, south, south * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_previous[local[0]][0]), 1, row, south, rank * 10 + south,
CART_COMM, &requests[requests_cnt++]);
}
if(west > -1) {
    MPI_Irecv(&(u_previous[0][0]), 1, column, west, west * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_previous[0][1]), 1, column, west, rank * 10 + west, CART_COMM,
&requests[requests_cnt++]);
}
if(east > -1) {
    MPI_Irecv(&(u_previous[0][local[1] + 1]), 1, column, east, east * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_previous[0][local[1]]), 1, column, east, rank * 10 + east,
CART_COMM, &requests[requests_cnt++]);
}
MPI_Waitall(requests_cnt, requests, MPI_STATUSES_IGNORE);
```

```
gettimeofday(&tcs, NULL);
Jacobi(u_previous, u_current, i_min, i_max, j_min, j_max);
gettimeofday(&tcf, NULL);
```

Έχουμε ορίσει ακόμα έναν πίνακα στον οποίο αποθηκεύουμε τα requests και μια μεταβλητή requests_cnt, για τις ανάγκες του πρώτου ορίσματος της συνάρτησης MPI_Waitall. Ακόμη, πραγματοποιούμε και το swap μεταξύ u_current και u_previous πριν την ανταλλαγή των ghost cells. Αυτά είναι και τα τελευταία κοινά χαρακτηριστικά των 3 υλοποιήσεων.

GaussSeidelSOR

Η συγκεκριμένη συνάρτηση απαιτεί να έχουν παραληφθεί ***πριν*** την κλήση της τα βόρεια και δυτικά ghost cells των πινάκων u_current. Στη συνέχεια, ***μετα*** την ολοκλήρωσή της αποστέλλουμε και τα υπόλοιπα ghost cells.

```
requests_cnt = 0;
if(north > -1) {
    MPI_Irecv(&(u_current[0][0]), 1, row, north, north * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
}

if(west > -1) {
    MPI_Irecv(&(u_current[0][0]), 1, column, west, west * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
}

MPI_Waitall(requests_cnt, requests, MPI_STATUSES_IGNORE);

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;

requests_cnt = 0;
```

```
if(north > -1) {
    MPI_Isend(&(u_current[1][0]), 1, row, north, rank * 10 + north, CART_COMM,
&requests[requests_cnt++]);
}
if(south > -1) {
    MPI_Irecv(&(u_current[local[0] + 1][0]), 1, row, south, south * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_current[local[0]][0]), 1, row, south, rank * 10 + south,
CART_COMM, &requests[requests_cnt++]);
}
if(west > -1) {
    MPI_Isend(&(u_current[0][1]), 1, column, west, rank * 10 + west, CART_COMM,
&requests[requests_cnt++]);
}
if(east > -1) {
    MPI_Irecv(&(u_current[0][local[1] + 1]), 1, column, east, east * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_current[0][local[1]]), 1, column, east, rank * 10 + east,
CART_COMM, &requests[requests_cnt++]);
}
MPI_Waitall(requests_cnt, requests, MPI_STATUSES_IGNORE);
```

RedBlackSOR

Τέλος, η RedBlackSOR απαιτεί αρχικά να υπάρχουν πριν την κλήση της RedSOR τα ghost cells από τα u_previous. Μετά το πέρας της, αναμένει τον επαναληπτικό διαμοιρασμό των ghost cells απο τα u_current και εν τέλει την κλήση της BlackSOR.

```
requests_cnt = 0;
if(north > -1) {
    MPI_Irecv(&(u_previous[0][0]), 1, row, north, north * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_previous[1][0]), 1, row, north, rank * 10 + north, CART_COMM,
&requests[requests_cnt++]);
}
if(south > -1) {
    MPI_Irecv(&(u_previous[local[0] + 1][0]), 1, row, south, south * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_previous[local[0]][0]), 1, row, south, rank * 10 + south,
CART_COMM, &requests[requests_cnt++]);
}
if(west > -1) {
```

```
    MPI_Irecv(&(u_previous[0][0]), 1, column, west, west * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_previous[0][1]), 1, column, west, rank * 10 + west, CART_COMM,
&requests[requests_cnt++]);
}
if(east > -1) {
    MPI_Irecv(&(u_previous[0][local[1] + 1]), 1, column, east, east * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_previous[0][local[1]]), 1, column, east, rank * 10 + east,
CART_COMM, &requests[requests_cnt++]);
}
MPI_Waitall(requests_cnt, requests, MPI_STATUSES_IGNORE);

gettimeofday(&tcs, NULL);

RedSOR(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;

gettimeofday(&tconvs, NULL);

requests_cnt = 0;
if(north > -1) {
    MPI_Irecv(&(u_current[0][0]), 1, row, north, north * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_current[1][0]), 1, row, north, rank * 10 + north, CART_COMM,
&requests[requests_cnt++]);
}
if(south > -1) {
    MPI_Irecv(&(u_current[local[0] + 1][0]), 1, row, south, south * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_current[local[0]][0]), 1, row, south, rank * 10 + south,
CART_COMM, &requests[requests_cnt++]);
}
if(west > -1) {
    MPI_Irecv(&(u_current[0][0]), 1, column, west, west * 10 + rank, CART_COMM,
&requests[requests_cnt++]);
    MPI_Isend(&(u_current[0][1]), 1, column, west, rank * 10 + west, CART_COMM,
&requests[requests_cnt++]);
}
if(east > -1) {
    MPI_Irecv(&(u_current[0][local[1] + 1]), 1, column, east, east * 10 + rank,
CART_COMM, &requests[requests_cnt++]);
    MPI_Isend(&(u_current[0][local[1]]), 1, column, east, rank * 10 + east,
CART_COMM, &requests[requests_cnt++]);
}
MPI_Waitall(requests_cnt, requests, MPI_STATUSES_IGNORE);
```

```
gettimeofday(&tcs, NULL);

BlackSOR(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;
```

Εκτέλεση & Μετρήσεις με Έλεγχο Σύγκλισης

Για να κάνουμε μετρήσεις χρειαζόμαστε να υποβάλουμε το πρόγραμμα μας προς εκτέλεση στο parlab queue του scirouter.

Χρησιμοποιούμε το ακόλουθο Makefile

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

all: jacobi_mpi gauss_mpi redblack_mpi

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

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

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

clean:
    rm redblack_mpi gauss_mpi jacobi_mpi
```

Για να τρέξουμε το make στο queue, χρησιμοποιούμε το ακόλουθο script make_on_queue.sh

```
#!/bin/bash

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

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

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

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmpi/1.8.3
cd /home/parallel/parlab13/HeatDiffusion
make
```

Η ορθότητα των 3 υλοποιήσεών μας ελέγχθηκε με το παρεχόμενο script test_correctness_mpi.sh.

Για να τρέξουμε τις μετρήσεις με έλεγχο σύγκλισης χρησιμοποιούμε το εξής script run_on_queue.sh:

```
#!/bin/bash

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

## Output and error files
#PBS -o run_mpi.out
#PBS -e run_mpi.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

## Start
```

```
##echo "PBS_NODEFILE = $PBS_NODEFILE"
##cat $PBS_NODEFILE

## Run the job (use full paths to make sure we execute the correct thing)
module load openmpi/1.8.3
cd /home/parallel/parlab13/HeatDiffusion

mpirun -np 64 --map-by node --mca btl self,tcp ./jacobi_mpi 1024 1024 8 8
mpirun -np 64 --map-by node --mca btl self,tcp ./gauss_mpi 1024 1024 8 8
mpirun -np 64 --map-by node --mca btl self,tcp ./redblack_mpi 1024 1024 8 8
```

Σημειώνεται ότι τόσο αυτό όσο και τα επόμενα scripts εξαγωγής μετρικών εκτελέστηκαν 3 φορές και τα αποτελέσματα που θα παρατεθούν αποτελούν τον μέσο όρο αυτών.

Επίδειξη Μετρήσεων με Έλεγχο Σύγκλισης

Παρακάτω δίνουμε τις επιδόσεις σε διάφορες παραμέτρους των 3 παράλληλων εκδοχών, σε ταμπλό 1024 x 1024.

```
Jacobi X 1024 Y 1024 Px 8 Py 8 Iter 798201 ComputationTime 39.742329 Convergence
Time 7.524570 TotalTime 221.080193 midpoint 5.431022 processes 64

GaussSeidelSOR X 1024 Y 1024 Px 8 Py 8 Iter 3201 ComputationTime 0.602642
Convergence Time 0.143842 TotalTime 2.266213 midpoint 5.642998 processes 64

RedBlackSOR X 1024 Y 1024 Px 8 Py 8 Iter 2501 ComputationTime 0.308155
ConvergenceTime 0.096920 TotalTime 1.506307 midpoint 5.642974 processes 64
```

Συμπεράσματα

Ο τρομακτικά μεγάλος αριθμός iterations που απαιτεί η μέθοδος Jacobi αναμενόμενα εκτοξεύει τόσο τον χρόνο που σπαταλάται στο έλεγχο της σύγκλισης, όσο και στο καθαρά υπολογιστικό μέρος. Μην αναφερθούμε δε στην ανταλλαγή των ghost cells, η οποία (και συνδυαστικά με τα υπόλοιπα) μας δίνει συνολικό χρόνο εκτέλεσης άνω των 4,5 λεπτών! Επίσης, το midpoint της είναι ελαφρώς χειρότερο σε σχέση με των 2 επόμενων υλοποιήσεων.

Οι άλλες 2 μέθοδοι φαινομενικά είναι πιο κοντά, όμως εν τέλει η RedBlackSOR είναι αυτή που ξεχωρίζει και προφανώς θα επιλεχθεί ως βέλτιστη. Οι διαφορές τους στο midpoint που δίνουν είναι αμελητέες.

Μετρήσεις χωρίς Έλεγχο Σύγκλισης

Αρχικά, μετατρέψαμε σε σχόλιο την ακόλουθη σειρά του Makefile, ώστε να μην ελέγχουμε για σύγκλιση των u_current, u_previous.

```
#CONV=-DTEST_CONV
```

Έτσι, τα επόμενα scripts μας έτρεξαν με σταθερό αριθμό iterations, συγκεκριμένα 256.

Για να τρέξουμε τις ζητούμενες μετρήσεις, χρησιμοποιήσαμε το παρεχόμενο script test_scalability_mpi.sh (διορθώσαμε απλά τα path και τα ονόματα των executables, καθώς και τα μεγέθη ταμπλό σε 2048, 4096 και 6144).

Επίδειξη Μετρήσεων χωρίς Έλεγχο Σύγκλισης(Α Μέρος)

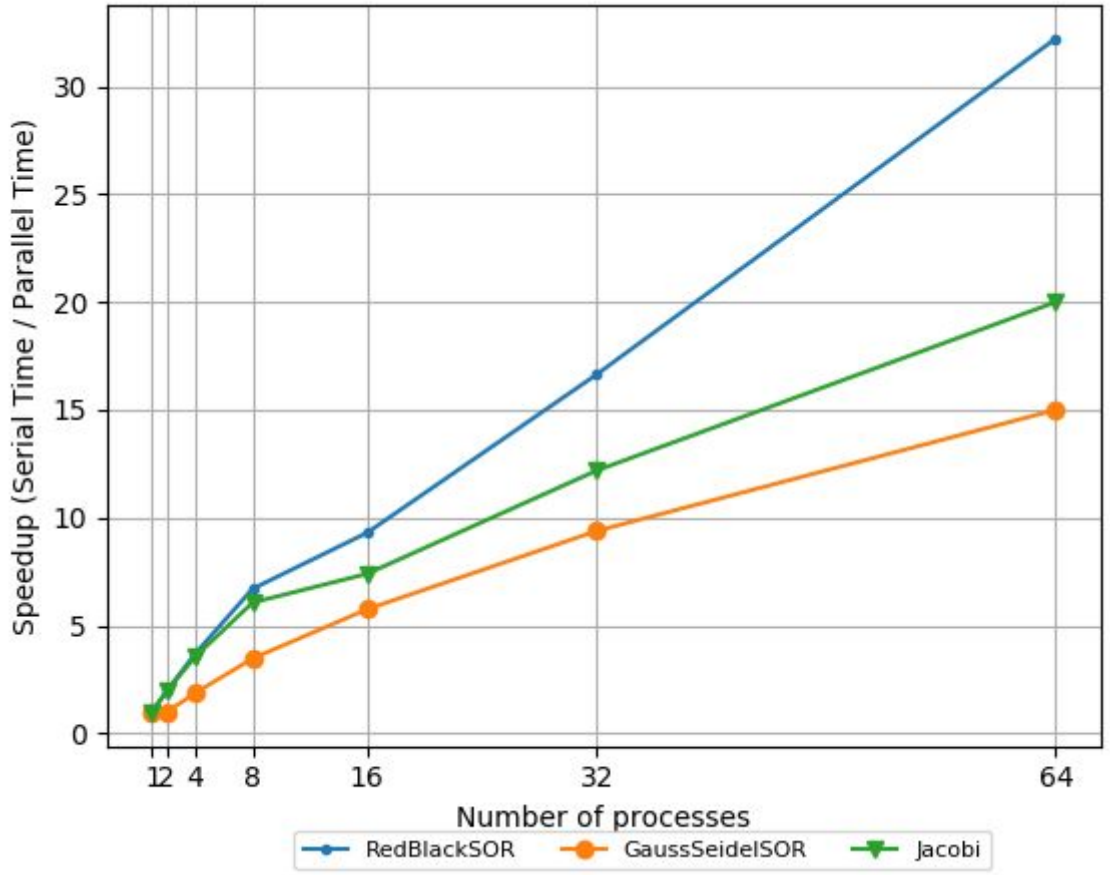
Για να δημιουργήσουμε γραφήματα speedup, υλοποιήσαμε το ακόλουθο python script.

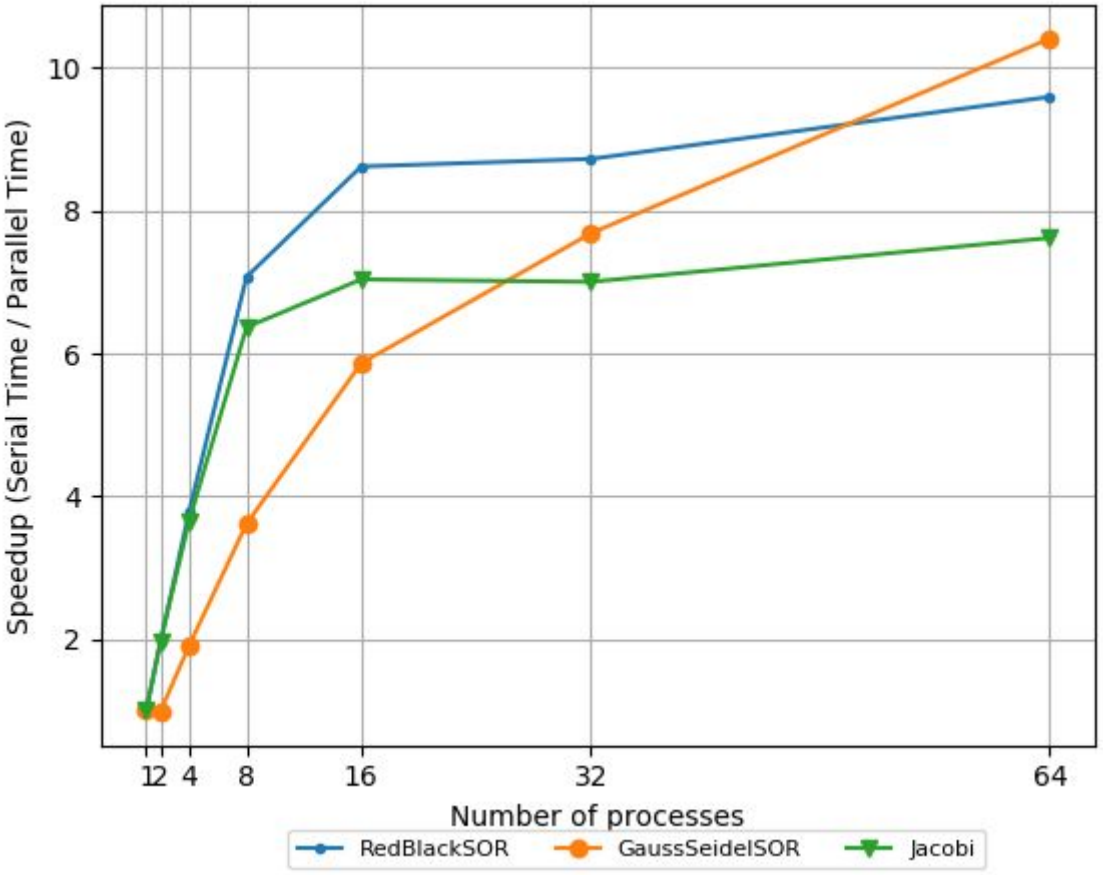
```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import sys

def parse_file(fname):
        with open(fname) as f:
                lines = f.readlines()

        i = 1
        size_stats = {}
        for line in lines:
                # line of the form:
```

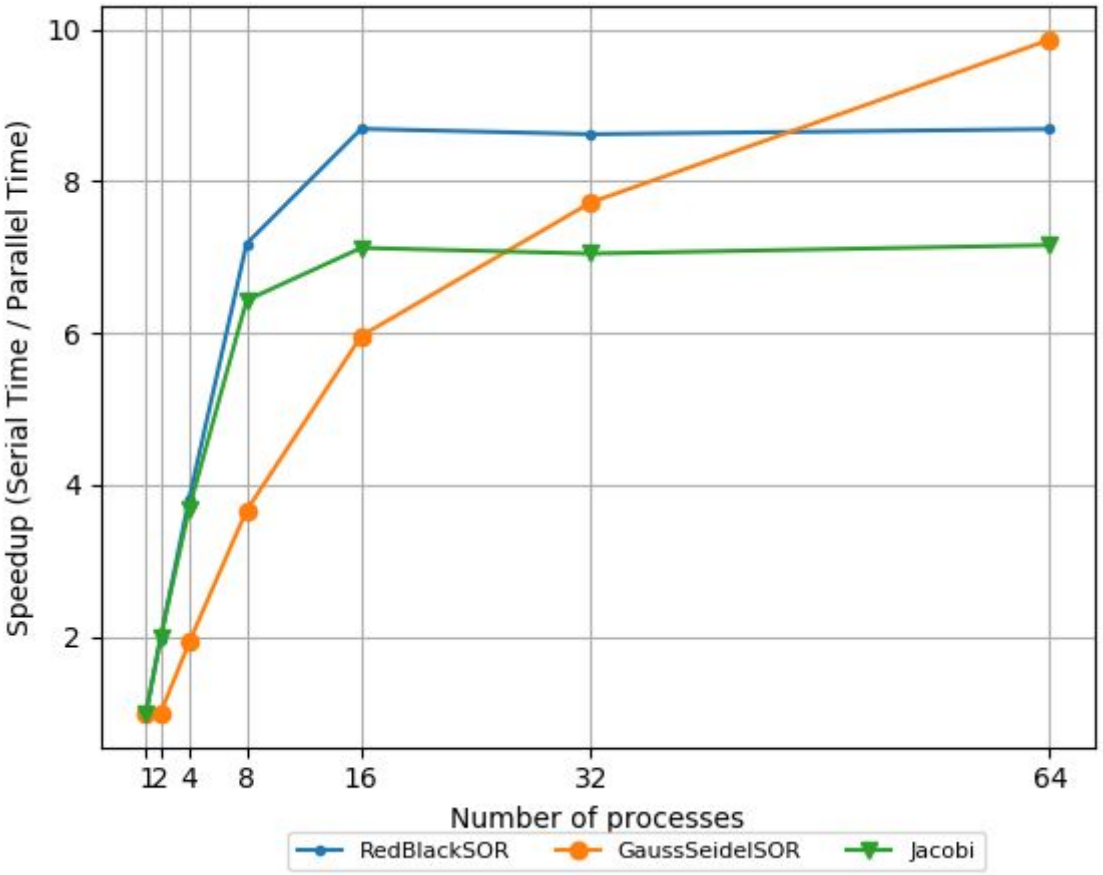
```
                # executable X Y Px Py Iter ComputationTime TotalTime midpoint
processes
                splitted = line.split()
                # executable
                executable = splitted[0].strip()
                # elapsed time
                elapsed_time = splitted[14].strip()
                # processes number
                process_num = splitted[18].strip()
                if not size_stats.get(executable, None):
                    size_stats[executable] = []

                size_stats[executable].append({"elapsed": elapsed_time,
"processes": process_num})
        return size_stats

if len(sys.argv) < 2:
    print ("Usage parse_stats.py <input_file>")
    exit(-1)

stats_by_size = parse_file(sys.argv[1])
markers = ['.', 'o', 'v', '*', 'D', 'X']

serial_time={}


x_ticks = [1, 2, 4, 8, 16, 32, 64]
fig = plt.figure(1)
plt.grid(True)
ax = plt.subplot(111)
ax.set_xlabel("Number of processes")
ax.set_ylabel("Speedup (Serial Time / Parallel Time)")
ax.xaxis.set_ticks(x_ticks)
ax.xaxis.set_ticklabels(map(str, x_ticks))

for j, size in enumerate(sorted(stats_by_size.keys(), key=lambda x: sorted(x))):
    stats = stats_by_size[size]
    y_axis = [0 for _ in range(len(x_ticks))]

    for stat in stats:
        if int(stat["processes"]) == 1:
            serial_time[size] = float(stat["elapsed"])


    for stat in stats:
        pos = x_ticks.index(int(stat["processes"]))
        y_axis[pos] = serial_time[size] / float(stat["elapsed"])

    ax.plot(x_ticks, tuple(y_axis), label=str(size), marker=markers[j])
```

```
lgd = ax.legend(ncol=len(stats_by_size.keys()), bbox_to_anchor=(0.9, -0.1),
prop={'size':8})
plt.savefig("heat-diffusion-6144-speedup.png", bbox_extra_artists=(lgd,),
bbox_inches='tight')
```

Ακολουθούν τα γραφήματα του speedup για μεγέθη ταμπλό 2048 x 2048, 4096 x 4096 και 6144 x 6144 αντίστοιχα.




Speedup (Serial Time / Parallel Time)
10
8
6
4
2
1
2
4
8
16
32
64
Number of processes
RedBlackSOR
GaussSeidelSOR
Jacobi


Speedup (Serial Time / Parallel Time)
10
8
6
4
2
1
2
4
8
16
32
64
Number of processes
RedBlackSOR
GaussSeidelSOR
Jacobi

Επιμέρους Συμπεράσματα

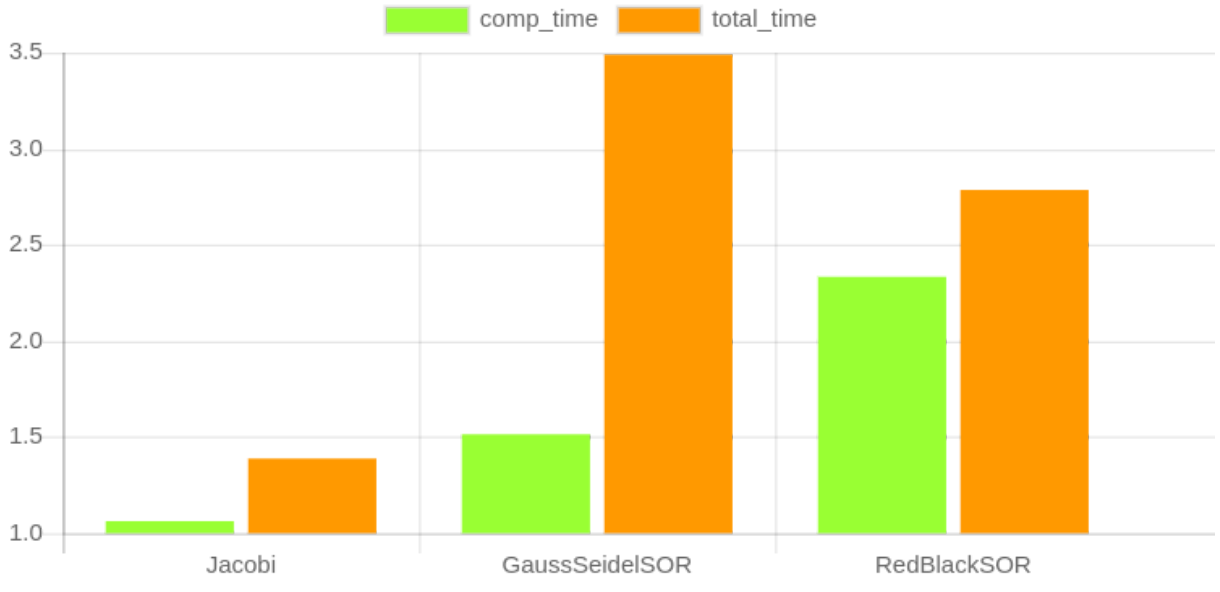
Στο μικρότερο ταμπλό φαίνεται η δύναμη του RedBlackSOR, ο οποίος κλιμακώνει πολύ καλύτερα από τις 2 άλλες υλοποιήσεις. Σε μεγαλύτερα ταμπλό ωστόσο, αυτό παύει να ισχύει, με τον GaussSeidelSOR να φτάνει μάλιστα να υπερισχύει (σε κλιμάκωση) στα 64 processes. Αυτό πιθανότατα πιστώνεται στο γεγονός ότι ο RBSOR έχει να επικοινωνήσει διπλάσιο αριθμό ghost rows και columns μεταξύ των processes (συγκριτικά με τις άλλες 2 υλοποιήσεις), το μέγεθος των οποίων αυξάνεται στα μεγαλύτερα ταμπλό.

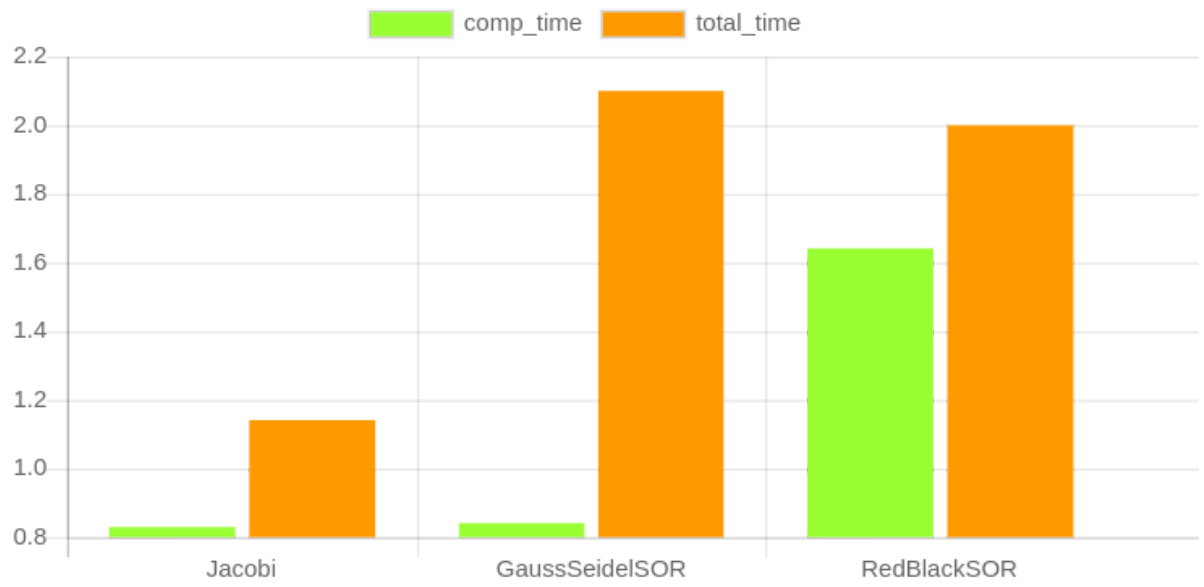
Ο Jacobi κλιμακώνει πολύ καλά μέχρι τα 8 processes, ωστόσο πέραν αυτών η αύξηση των communication times για 16+ processes φαίνεται να αντισταθμίζει πλήρως την μείωση των computation times.

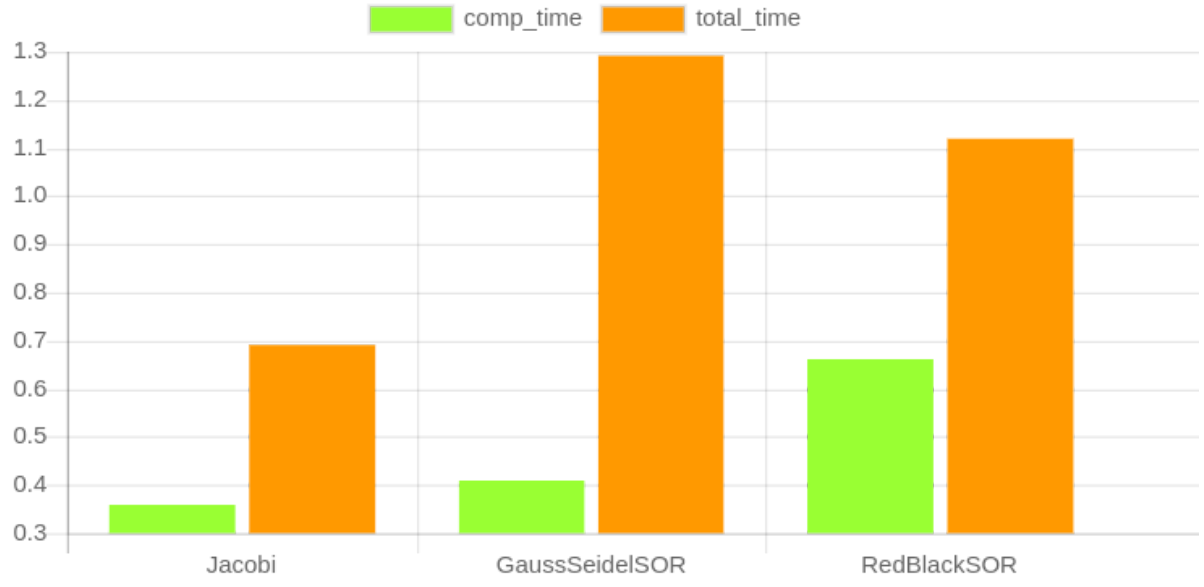
Η πιο ενδιαφέρουσα συμπεριφορά είναι αυτή του GaussSeidelSOR, ο οποίος παρουσιάζει παραπλήσια κλιμάκωση ανεξάρτητα των 3 ταμπλό.

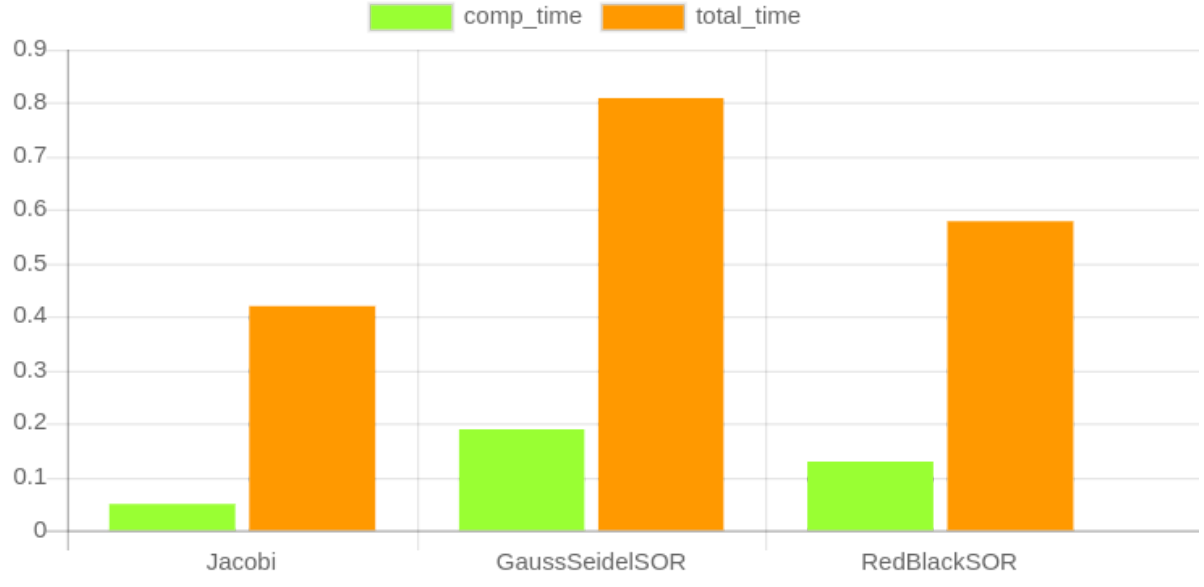
Επίδειξη Μετρήσεων χωρίς Έλεγχο Σύγκλισης(Β Μέρος)

Σε όλα τα παρακάτω γραφήματα, ο άξονας y δίνει χρόνο σε second. Αρχικά, δίνουμε τα αποτελέσματά μας για ταμπλό 2048 x 2048 και αριθμό processes 8, 16, 32 και 64 αντίστοιχα.

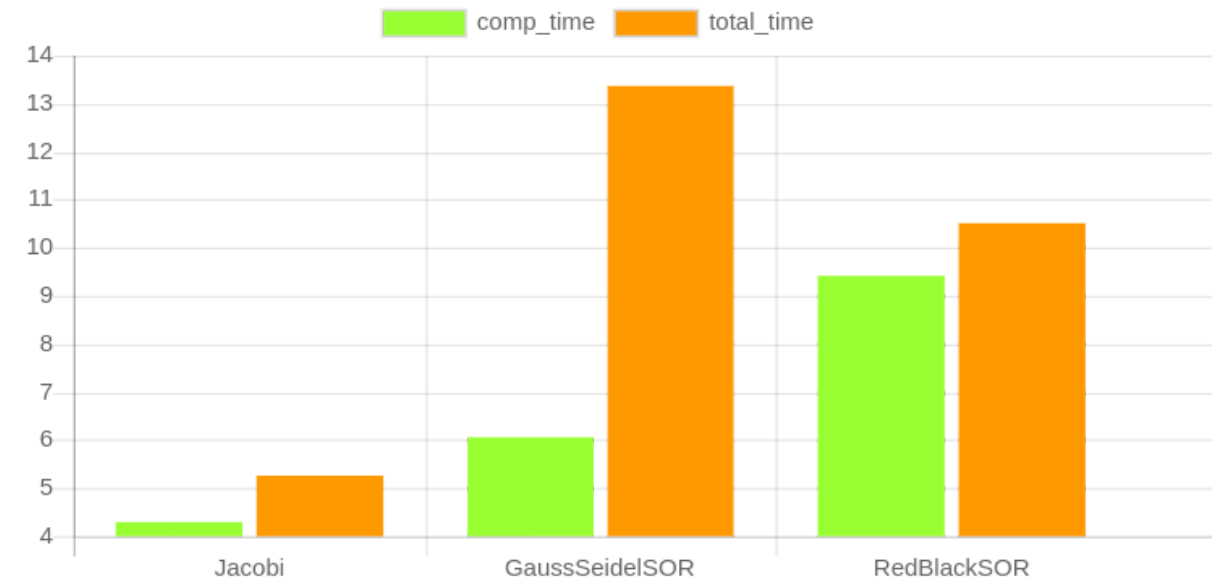


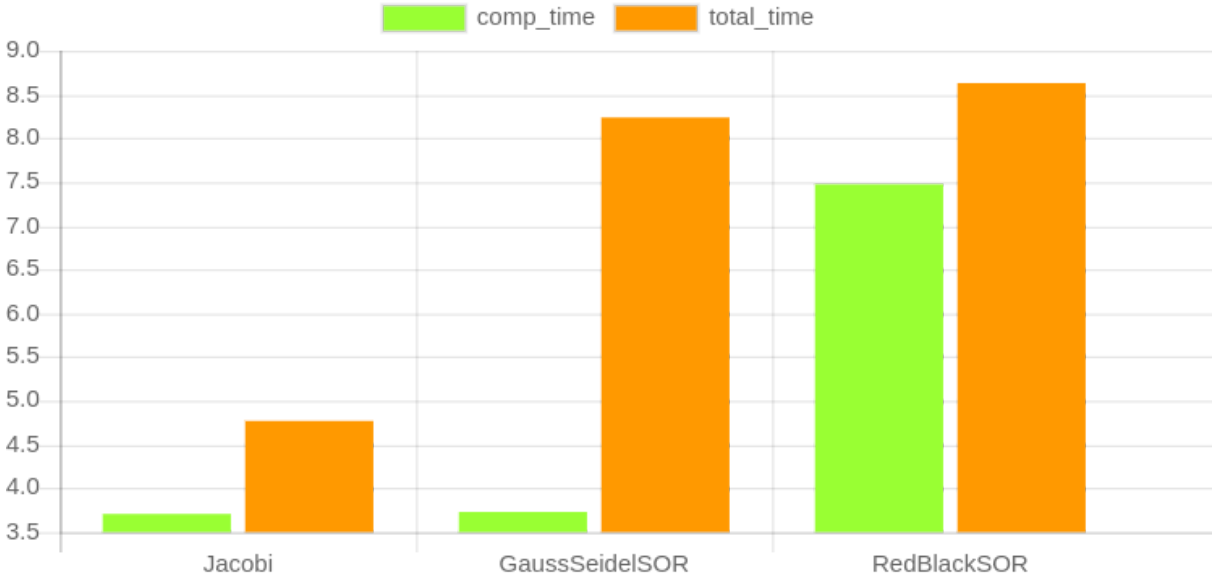

comp_time
total_time
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
Jacobi
GaussSeidelSOR
RedBlackSOR

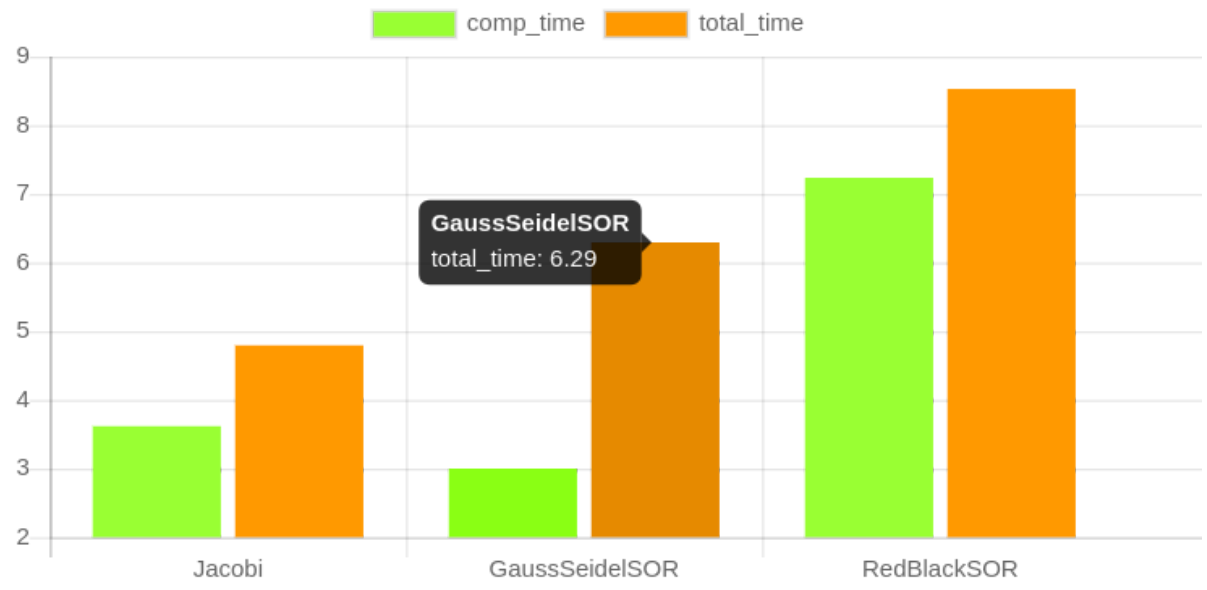

comp_time
total_time
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Jacobi
GaussSeidelSOR
RedBlackSOR

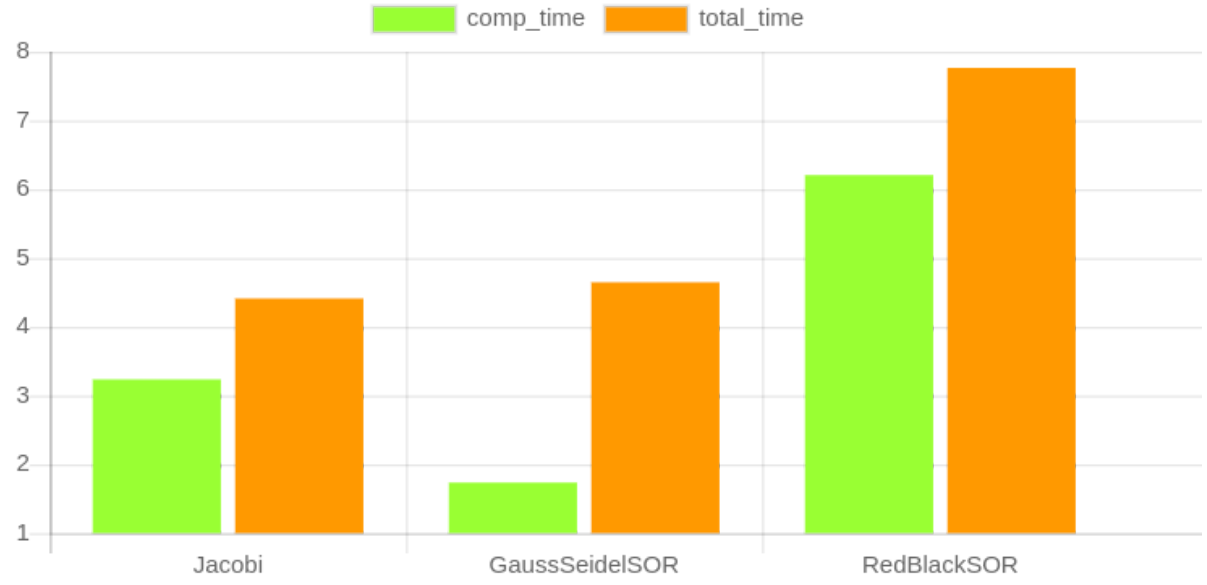

comp_time
total_time
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Jacobi
GaussSeidelSOR
RedBlackSOR

Στη συνέχεια, δίνουμε τα αποτελέσματά μας για ταμπλό 4096 x 4096 και αριθμό processes 8, 16, 32 και 64 αντίστοιχα.

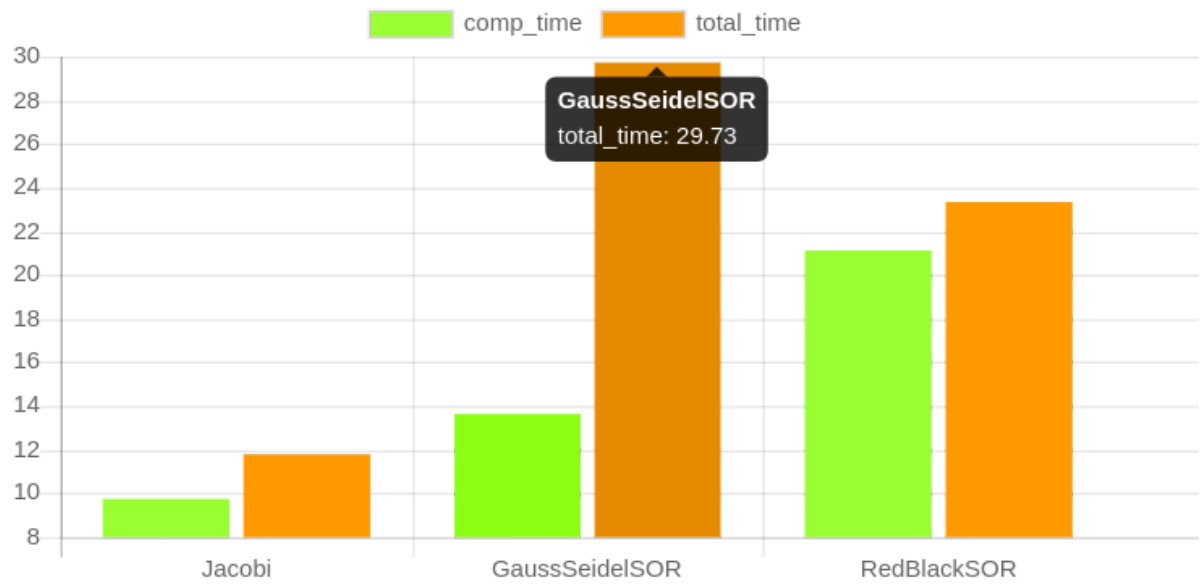


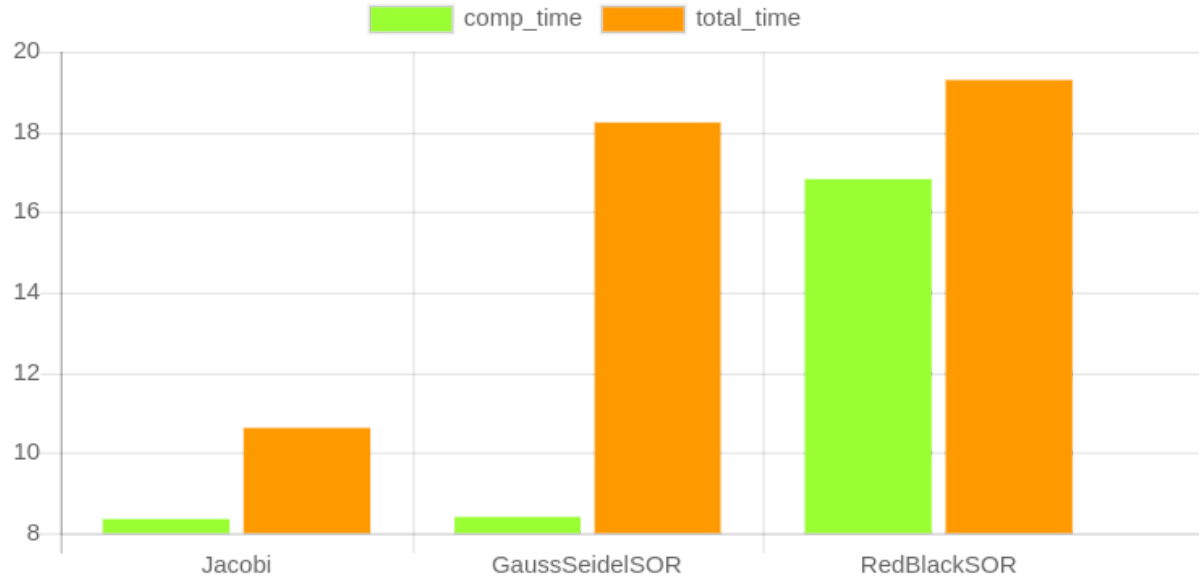


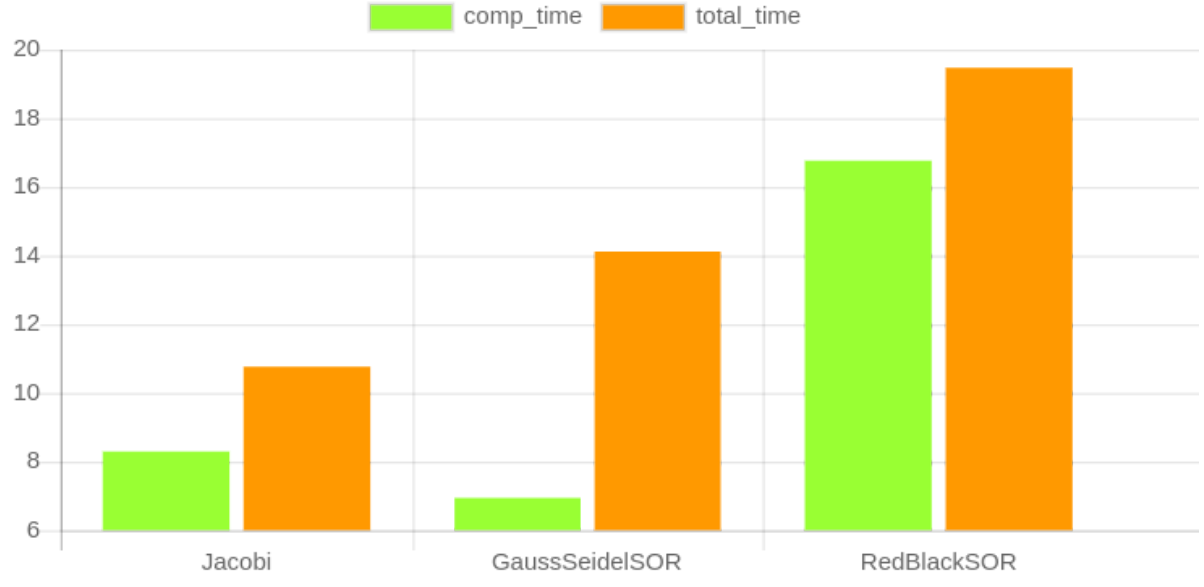




Τέλος, δίνουμε τα αποτελέσματά μας για ταμπλό 6144 x 6144 και αριθμό processes 8, 16, 32 και 64 αντίστοιχα.


comp_time
total_time
30
28
26
24
22
20
18
16
14
12
10
8
GaussSeidelSOR
total_time: 29.73
Jacobi
GaussSeidelSOR
RedBlackSOR


comp_time
total_time
20
18
16
14
12
10
8
Jacobi
GaussSeidelSOR
RedBlackSOR


comp_time
total_time
20
18
16
14
12
10
8
6
Jacobi
GaussSeidelSOR
RedBlackSOR

Επιμέρους Συμπεράσματα

Όπως παρατηρήθηκε και πριν, στο μικρότερο ταμπλό και οι 3 υλοποιήσεις παρουσιάζουν σχετική κλιμάκωση. Κάνοντας focus τώρα στα 8+ processes, βλέπουμε ότι ο RBSOR μειώνει σημαντικά τα comp times του αλλά όχι και τα communication times του, ο Jacobi μειώνει σχετικά τα comp times του αλλά όχι και τα comm times του, ενώ ο GSSOR μειώνει σημαντικά και τα 2. Δηλαδή ο GSSOR κλιμακώνει καλύτερα στα 8+ processes, σε σχέση με έως τα 8 processes.

Στα άλλα 2 ταμπλό, οι Jacobi και RBSOR εξουδετερώνονται σε μεγάλο βαθμό, με τον GSSOR να παραμένει απτόητος στην κλιμάκωσή του για 8+ processes. Πιθανότατα, αν ο GSSOR κατόρθωνε να έχει λιγότερα συνολικά iterations με ενεργοποιημένο το corvengence test, να ήταν αυτός ο βέλτιστος αλγόριθμός μας.