Κουτρούμπας Αθανάσιος 03116073
Αραπίδης Μανώλης 03116071

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ
2η ΑΣΚΗΣΗ

Παράλληλη Επίλυση Εξίσωσης Θερμότητας
Ακ. έτος 2020-2021, 9ο Εξάμηνο, Σχολή ΗΜ&ΜΥ

ΖΗΤΟΥΜΕΝΑ

2.1

Ανακαλύψτε τον παραλληλισμό του αλγορίθμου και σχεδιάστε την παραλληλοποίησή του σε αρχιτεκτονικές κατανεμημένης μνήμης με μοντέλο ανταλλαγής μηνυμάτων.

Παραλληλισμός Jacobi

```
for (t = 0; t < T && !converged; t++)
    for (i = X_min; i < X_max; i++)
        for (j = Y_min; j < Y_max; j++)
            u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] +
u_previous[i][j-1] +
                               u_previous[i][j+1]) / 4.0;
```

Αλγόριθμος Jacobi

Παρατηρήσαμε για τον αλγόριθμο του Jacobi, ότι τα δύο εσωτερικά loops είναι πλήρως ανεξάρτητα, άρα και παραλληλοποιήσιμα. Αυτό συμβαίνει, διότι οι τιμές στα κελιά current, εξαρτώνται μόνο από τα γειτονικά τους κελιά στο πίνακα previous, και δεν υπάρχει καμία εξάρτηση μεταξύ τους.

Χρησιμοποιήσαμε το **MPI** για την επικοινωνία μεταξύ των κόμβων (μέσω ανταλλαγής μηνυμάτων).

Αρχικά χωρίζουμε τον πίνακα εισόδου, σε ισομεγέθη χωρία, τα οποία είναι ίσα με το πλήθος των διεργασιών μας. Η κάθε διεργασία εκτελεί τον αλγόριθμο στο κάθε κελί του χωρίου που

έχει παραλάβει. Για την ορθή λειτουργία του αλγορίθμου, χρειάζεται να ανταλλάξει δεδομένα με γειτονικές διεργασίες, πιο συγκεκριμένα στήλες και γραμμές.

Αναλυτικά η διαδικασία ανταλλαγής και επεξεργασίας δεδομένων φαίνεται παρακάτω:

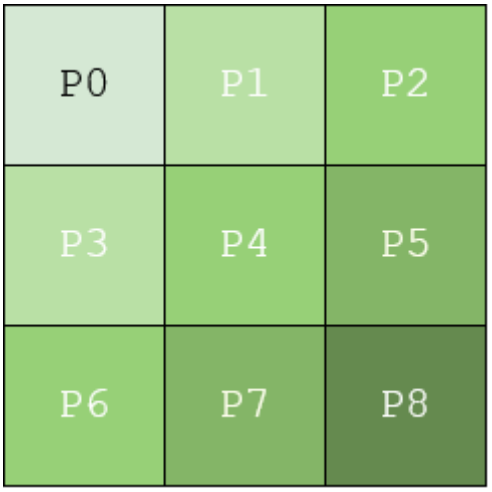
- Επιλέγουμε ως **root** την διεργασία 0.
    - Η διεργασία root είναι υπεύθυνη για το αρχικό διαχωρισμό του πίνακα σε χωρία και τον διαμοιρασμό αυτών στις υπόλοιπες διεργασίες.
- Η κάθε διεργασία χρειάζεται:
    - Να **παραλάβει** (ασύγχρονα):
        - Από τον **βόρειο** γείτονά της, την τελευταία του γραμμή.
        - Από τον **νότιο** γείτονά της, την πρώτη του γραμμή.
        - Από τον **δυτικό** γείτονά της, την τελευταία του στήλη (δεξιά στήλη).
        - Από τον **ανατολικό** γείτονά της, την πρώτη του στήλη (αριστερή στήλη).
    - Να **στείλει** (ασύγχρονα):
        - Στον **βόρειο** γείτονά της, την πρώτη του γραμμή.
        - Στον **νότιο** γείτονά της, την τελευταία του γραμμή.
        - Στον **δυτικό** γείτονά της, την πρώτη του στήλη (αριστερή στήλη).
        - Στον **ανατολικό** γείτονά της, την τελευταία του στήλη (δεξιά στήλη).
    - Να εκτελέσει τον αλγόριθμο σε κάθε κελί της με τα δεδομένα που παρέλαβε από τους γείτονές της για τα κελιά της περιφέρειας.
    - Η διεργασίες με χωρία που δεν έχουν όλους τους γείτονες, στέλνουν και παραλαμβάνουν δεδομένα μόνο για τους υπάρχοντες.
- Τέλος, όταν ο αλγόριθμος συγκλίνει ή φτάσει έως ένα συγκεκριμένο αριθμό επαναλήψεων, τότε η διεργασία **root** αναλαμβάνει να μαζέψει το χωρίο από κάθε διεργασία και να το συνενώσει στον αρχικό πίνακα.

Παραλληλισμός Gauss-Seidel με SOR

```
for (t = 0; t < T && !converged; t++)
    for (i = X_min; i < X_max; i++)
        for (j = Y_min; j < Y_max; j++)
            u_current[i][j] = u_previous[i][j] + (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]) * omega /
4.0;
```

Παρατηρήσαμε για τον αλγόριθμο του Gauss-Seidel, ότι ο υπολογισμός του current εξαρτάται από τωρινές γραμμές όπως και προηγούμενος. Οπότε δεν παραλληλοποείται όπως ο Jacobi. Κάθε διεργασία για να προχωρήσει στην εκτέλεση του αλγορίθμου, χρειάζεται να παραλάβει τις τρέχουσες γραμμές και στήλες από τον **βόρειο** και **δυτικό** γείτονα της. Ενώ όπως και στον Jacobi χρειάζεται να παραλάβει τις προηγούμενες γραμμές και στήλες από τον **νότιο** και τον **ανατολικό** γείτονα της, οπότε έχουμε ανεξαρτησία μεταξύ αυτών των παραλαβών.
Άρα οι διεργασίες που μόλις έχουν παραλάβει από τους βόρειους και δυτικούς γείτονες τις απαραίτητες γραμμές και στήλες, δεν έχουν κάποια εξάρτηση μεταξύ τους και μπορούν να εκτελεστούν παράλληλα. Εκεί βρίσκεται η παραλληλία του αλγόριθμου Gauss-Seidel.

Για παράδειγμα, εάν έχουμε τον πίνακα εισόδου που φαίνεται παρακάτω:



Παράδειγμα πίνακα εισόδου 3 x 3

Παρατηρούμε ότι οι διεργασίες που μπορούν να εκτελεστούν παράλληλα είναι οι παρακάτω και με την εξής σειρά:

- P_0
- P_1, P_3
- P_2, P_4, P_6
- P_5, P_7
- P_8

Χρησιμοποιήσαμε το **MPI** για την επικοινωνία μεταξύ των κόμβων (μέσω ανταλλαγής μηνυμάτων).

Αρχικά, όπως και στον αλγόριθμο Jacobi, χωρίζουμε τον πίνακα εισόδου σε ισομεγέθη χωρία, τα οποία είναι ίσα με το πλήθος των διεργασιών μας. Η κάθε διεργασία εκτελεί τον αλγόριθμο στο κάθε κελί του χωρίου που έχει παραλάβει. Για την ορθή λειτουργία του αλγορίθμου, χρειάζεται να ανταλλάξει δεδομένα με γειτονικές διεργασίες, πιο συγκεκριμένα στήλες και γραμμές.

Αναλυτικά η διαδικασία ανταλλαγής και επεξεργασίας δεδομένων φαίνεται παρακάτω:

- Επιλέγουμε ως **root** την διεργασία 0.
    - Η διεργασία root είναι υπεύθυνη για το αρχικό διαχωρισμό του πίνακα σε χωρία και τον διαμοιρασμό αυτών στις υπόλοιπες διεργασίες.
- Η κάθε διεργασία έχει δύο φάσεις:
    - **1η Φάση** :
        - Να **παραλάβει** (ασύγχρονα):
            - Από τον **νότιο** γείτονά της, την πρώτη γραμμή του previous.
            - Από τον **ανατολικό** γείτονά της, την πρώτη στήλη του previous.
        - Να **στείλει** (ασύγχρονα):
            - Στον **βόρειο** γείτονά της, την πρώτη γραμμή του previous.
            - Στον **δυτικό** γείτονά της, την πρώτη στήλη του previous.
    - **2η Φάση**:
        - Περιμένει να **παραλάβει** (ασύγχρονα):
            - Από τον **βόρειο** γείτονά της, την τελευταία γραμμή του current.
            - Από τον **δυτικό** γείτονά της, την τελευταία στήλη του current.
        - Εκτελεί τον αλγόριθμο Gauss-Seidel στο χωρίο της.
        - Να **στείλει** (ασύγχρονα):
            - Στον **νότιο** γείτονά της, την τελευταία γραμμή του current.
            - Στον **ανατολικό** γείτονά της, την τελευταία στήλη του current.
    - Η διεργασίες με χωρία που δεν έχουν όλους τους γείτονες, στέλνουν και παραλαμβάνουν δεδομένα μόνο για τους υπάρχοντες.
- Τέλος, όταν ο αλγόριθμος συγκλίνει ή φτάσει έως ένα συγκεκριμένο αριθμό επαναλήψεων, τότε η διεργασία **root** αναλαμβάνει να μαζέψει το χωρίο από κάθε διεργασία και να το συνενώσει στον αρχικό πίνακα.

Παραλληλισμός Red-Black με SOR

```
// RedSOR Phase
for (i = X_min; i < X_max; i++)
    for (j = Y_min; j < Y_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]);
```

```
// BlackSOR Phase
for (i=X_min;i<X_max;i++)
    for (j=Y_min;j<Y_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]);
```

Παρατηρήσαμε για τον αλγόριθμο του Red-Black , ότι ο υπολογισμός του current χωρίζεται σε δύο φάσεις.

Κόκκινα κελιά : `(i+j)%2 = 0`
Μαύρα κελιά : `(i+j)%2 = 1`

Στην **πρώτη φάση** γίνεται ο υπολογισμός των κοκκινων κελιών του πίνακα current σε κάθε block, τα οποία εξαρτώνται από τα μαύρα του πίνακα previous του ίδιου του block καθώς και των γειτόνων του. Επομένως αφού τα κόκκινα κελιά εξαρτώνται μόνο από τον πίνακα previous, είναι ανεξάρτητα μεταξύ τους και μπορούν να εκτελεστούν παράλληλα.

Στην **δεύτερη φάση** γίνεται ο υπολογισμός των μαύρων κελιών σε κάθε block, τα οποία εξαρτώνται από τα κόκκινα του πίνακα current του ίδιου του block καθώς και των γειτόνων του. Επομένως πρέπει ο υπολογισμός τους να γίνεται μετά των κόκκινων του current και επειδή εξαρτάται αποκλειστικά από αυτά, είναι ανεξάρτητα μεταξύ τους και μπορούν να εκτελεστούν παράλληλα.

Οπότε η παραλληλία του αλγορίθμου είναι ότι τα κόκκινα κελια υπολογίζονται πρώτα και είναι ανεξάρτητα μεταξύ τους, άρα και ο υπολογισμός είναι παραλληλοποιήσιμος. Έπειτα ακολουθεί με την ίδια λογική, ο παράλληλος υπολογισμός των μαύρων ανεξάρτητων κελιών.

Αναλυτικά η διαδικασία ανταλλαγής και επεξεργασίας δεδομένων φαίνεται παρακάτω:

- Επιλέγουμε ως **root** την διεργασία 0.
    - Η διεργασία root είναι υπεύθυνη για το αρχικό διαχωρισμό του πίνακα σε χωρία και τον διαμοιρασμό αυτών στις υπόλοιπες διεργασίες.
- Η κάθε διεργασία έχει δύο φάσεις:
    - **1η Φάση**: Υπολογισμός κόκκινων κελιών
        - Να **παραλάβει** (ασύγχρονα):
            - Από τον **βόρειο** γείτονά της, την τελευταία γραμμή του previous.
            - Από τον **νότιο** γείτονά της, την πρώτη γραμμή του previous.
            - Από τον **δυτικό** γείτονά της, την τελευταία στήλη του previous.
            - Από τον **ανατολικό** γείτονά της, την πρώτη στήλη του previous.
        - Να **στείλει** (ασύγχρονα):
            - Στον **βόρειο** γείτονά της, την πρώτη γραμμή του previous.
            - Στον **νότιο** γείτονά της, την τελευταία γραμμή του previous.
            - Στον **δυτικό** γείτονά της, την πρώτη στήλη του previous.
            - Στον **ανατολικό** γείτονά της, την τελευταία στήλη του previous.
        - Εκτελεί τον αλγόριθμο **RedSor** στο χωρίο της.
    - **2η Φάση**: Υπολογισμός μαύρων κελιών
        - Να **παραλάβει** (ασύγχρονα):
            - Από τον **βόρειο** γείτονά της, την τελευταία γραμμή του current.
            - Από τον **νότιο** γείτονά της, την πρώτη γραμμή του current.
            - Από τον **δυτικό** γείτονά της, την τελευταία στήλη του current.
            - Από τον **ανατολικό** γείτονά της, την πρώτη στήλη του current.
        - Να **στείλει** (ασύγχρονα):
            - Στον **βόρειο** γείτονά της, την πρώτη γραμμή του current.
            - Στον **νότιο** γείτονά της, την τελευταία γραμμή του current.
            - Στον **δυτικό** γείτονά της, την πρώτη στήλη του current.
            - Στον **ανατολικό** γείτονά της, την τελευταία στήλη του current.
        - Εκτελεί τον αλγόριθμο **BlackSor** στο χωρίο της.
    - Η διεργασίες με χωρία που δεν έχουν όλους τους γείτονες, στέλνουν και παραλαμβάνουν δεδομένα μόνο για τους υπάρχοντες.
- Τέλος, όταν ο αλγόριθμος συγκλίνει ή φτάσει έως ένα συγκεκριμένο αριθμό επαναλήψεων, τότε η διεργασία **root** αναλαμβάνει να μαζέψει το χωρίο από κάθε διεργασία και να το συνενώσει στον αρχικό πίνακα.

2.2

Αναπτύξτε παράλληλο πρόγραμμα στο μοντέλο ανταλλαγής μηνυμάτων με τη βοήθεια της βιβλιοθήκης MPI. Στο αρχείο `mpi_skeleton.c` σας δίνεται σκελετός υλοποίησης σε MPI, στον οποίο καλείστε να συμπληρώσετε τον κώδικά σας.

Πρόγραμμα Jacobi

Το πλήρες πρόγραμμα βρίσκεται στο αρχείο `jacobi_mpi.c`

Χρησιμοποιήσαμε το αρχείο `mpi_skeleton.c`, και προσθέσαμε ότι χρειαζόταν για να λειτουργήσει ο αλγόριθμος Jacobi.
Αρχικά, κάνουμε scatter τον πίνακα U μέσω του pointer `u=&U[0][0]`, όπου `scattercounts` είναι πίνακας που δηλώνει το πλήθος των global block που στέλνουμε, πιο συγκεκριμένα 1 σε κάθε διεργασία και `scatteroffset` δηλώνει από που αρχίζει να εφαρμόζεται στον πίνακα η δομή global block. Στον παραλήπτη αποθηκεύεται μία (1) δομή local block στον πίνακα `u_previous` αρχίζοντας από την θέση (1,1).

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

Έπειτα αρχικοποιούμε τον πίνακα `u_current` να είναι ίσος με τον `u_previous`:

```
for(i=0; i<local[0]+2; i++)
    for(j=0; j<local[1]+2; j++)
        u_current[i][j]=u_previous[i][j];
```

Ορίζουμε δύο δομές δεδομένων για την αποστολή και παραλαβή γραμμής και στήλης από και σε γειτονικές διεργασίες.

```
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);
```

Βρίσκουμε τους τέσσερις γείτονες για κάθε διεργασία, μέσω της 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);
```

Κάνουμε έλεγχο για την ύπαρξη γειτόνων σε κάθε κατεύθυνση και ορίζουμε τα αρχικά και τελικά όρια με τα οποία θα τρέξει ο Jacobi στο συγκεκριμένο block.

```
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];
```

Έπειτα μπαίνουμε στο επαναληπτικό κομμάτι του κώδικα, το οποίο εκτελείται μέχρι να γίνει η σύγκλιση (ή μόλις φτάσει τις T επαναλήψεις) ή αν δεν έχουμε ορίσει σύγκλιση τρέχει για 256 επαναλήψεις. Μέσα σε αυτό το κομμάτι γίνονται οι αποστολές/λήψεις δεδομένων μεταξύ των διεργασιών όπως περιγράψαμε στο προηγούμενο ερώτημα. Όλες αυτές οι κλήσεις είναι ασύγχρονες και ύστερα από την κλήση τους όλες περιμένουν μέσω τις `MPI_Waitall`, την ολοκλήρωση όλων των μεταφορών. Τέλος καλούμε την συνάρτηση Jacobi για να γίνει το υπολογιστικό μέρος του αλγόριθμου για το block της διεργασίας.

```
#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

 /*Compute and Communicate*/

swap=u_previous;
 u_previous=u_current;
 u_current=swap;
counts=0;
```

```
	if(north >= 0){
		MPI_Isend(&u_previous[1][1], 1, row, north, rank, CART_COMM, &requests[counts++]);
		MPI_Irecv(&u_previous[0][1], 1, row, north, north, CART_COMM, &requests[counts++]);
	}

	if(south >= 0){
		MPI_Isend(&u_previous[local[0]][1], 1, row, south, rank, CART_COMM,
&requests[counts++]);
		MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south, south, CART_COMM,
&requests[counts++]);
	}

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

	if(east >=0){
		MPI_Isend(&u_previous[1][local[1]], 1, column, east, rank, CART_COMM,
&requests[counts++]);
		MPI_Irecv(&u_previous[1][local[1]+1], 1, column, east, east, CART_COMM,
&requests[counts++]);
	}

	MPI_Waitall(counts, 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;
```

Έπειτα, ελέγχουμε κάθε 100 επαναλήψεις εάν έχουμε φτάσει στην σύγκλιση μέσω της συνάρτησης `j_converge`, η οποία είναι η δοσμένη `converge` απλά με τα αρχικά και τελικά όρια προσαρμοσμένα όπως υπολογίσαμε παραπάνω. Έπειτα κάνουμε `MPI_Allreduce`, την τιμή που επιστρέφει η `j_converge` για να ελέγξουμε ότι όλες οι διεργασίες έχουν συγκλίνει.

```
	#ifdef TEST_CONV
	if (t%C==0) {
		gettimeofday(&tcvs,NULL);
		converged=j_converge(u_previous,u_current,i_min, i_max, j_min, j_max);
		gettimeofday(&tcvf,NULL);
		 MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
		tconv+=(tcvf.tv_sec-tcvs.tv_sec)+(tcvf.tv_usec-tcvs.tv_usec)*0.000001;
	 }
	 #endif
```

Εφόσων έχει προκύψει η σύγκλιση (ή ο μέγιστος αριθμός επαναλήψεων) τότε ο root κάνει Reduce επιλέγοντας το μέγιστο από, τον συνολικό χρόνο εκτέλεσης, τον χρόνο υπολογισμού (ή και τον χρόνο σύγκλισης).

```
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 root διεργασία κάνει Gather τους υποπίνακες από όλες τις διεργασίες και τους ενώνει στον αρχικό πίνακα U.

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

Πρόγραμμα Gauss-Seidel με SOR

Το πλήρες πρόγραμμα βρίσκεται στο αρχείο `seidelsor_mpi.c`

Για την υλοποίηση του Gauss-Seidel, χρησιμοποιήσαμε στην ουσία τον ίδιο κώδικα του Jacobi απλά αλλάζοντας το κομμάτι των ανταλλαγών δεδομένων όπως φαίνεται παρακάτω. Απλά υλοποιούμε σε κώδικα την διαδικασία που περιγράψαμε στο προηγούμενο ερώτημα:

```
if(north >= 0){
    MPI_Isend(&u_previous[1][1], 1, row, north, rank, CART_COMM, &pre_requests[pre_counts++]);
    MPI_Irecv(&u_current[0][1], 1, row, north, north, CART_COMM, &pre_requests[pre_counts++]);
}

if(west >=0){
    MPI_Isend(&u_previous[1][1], 1, column, west, rank, CART_COMM, &pre_requests[pre_counts++]);
    MPI_Irecv(&u_current[1][0], 1, column, west, west, CART_COMM, &pre_requests[pre_counts++]);
}

if(south >= 0){
    MPI_Irecv(&u_previous[local[0]+1][1], 1, row, south, south, CART_COMM,
&pre_requests[pre_counts++]);
}

if(east >=0){
    MPI_Irecv(&u_previous[1][local[1]+1], 1, column, east, east, CART_COMM,
&pre_requests[pre_counts++]);
}

MPI_Waitall(pre_counts, pre_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;

if(south >= 0){
    MPI_Isend(&u_current[local[0]][1], 1, row, south, rank, CART_COMM,
&posts_requests[post_counts++]);
}

if(east >=0){
    MPI_Isend(&u_current[1][local[1]], 1, column, east, rank, CART_COMM,
&posts_requests[post_counts++]);
}

MPI_Waitall(post_counts, posts_requests, MPI_STATUSES_IGNORE);
```

Πρόγραμμα Red-Black με SOR

Το πλήρες πρόγραμμα βρίσκεται στο αρχείο `redblacksor_mpi.c`

Για την υλοποίηση του Red-Black, χρησιμοποιήσαμε στην ουσία τον ίδιο κώδικα του Jacobi απλά αλλάζοντας το κομμάτι των ανταλλαγών δεδομένων όπως φαίνεται παρακάτω.
Απλά υλοποιούμε σε κώδικα την διαδικασία που περιγράψαμε στο προηγούμενο ερώτημα:

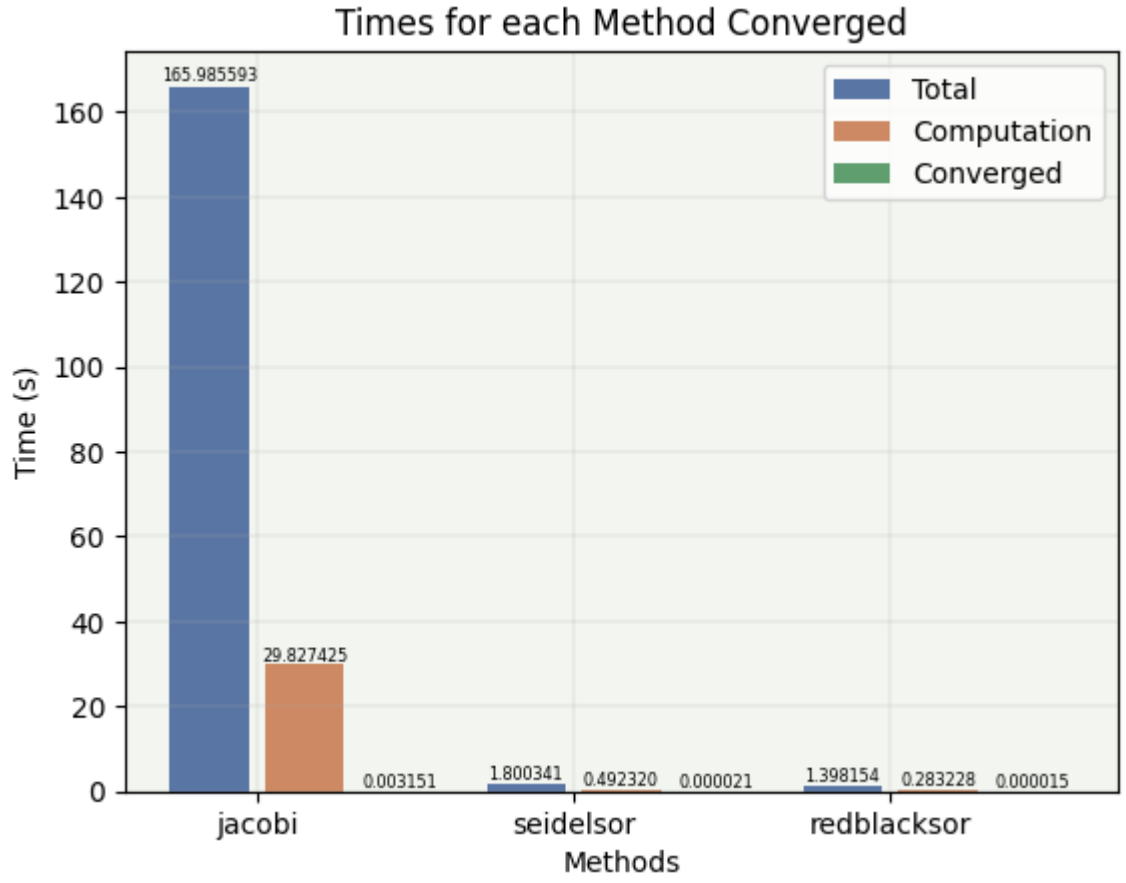
```
if(north >= 0){
      MPI_Isend(&u_previous[1][1], 1, row, north, rank, CART_COMM, &requests[counts++]);
      MPI_Irecv(&u_previous[0][1], 1, row, north, north, CART_COMM, &requests[counts++]);
}
if(west >=0){
      MPI_Isend(&u_previous[1][1], 1, column, west, rank, CART_COMM, &requests[counts++]);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

MPI_Waitall(counts, 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;

counts=0;

if(north >= 0){
      MPI_Isend(&u_current[1][1], 1, row, north, rank, CART_COMM, &requests[counts++]);
      MPI_Irecv(&u_current[0][1], 1, row, north, north, CART_COMM, &requests[counts++]);
}
if(west >=0){
      MPI_Isend(&u_current[1][1], 1, column, west, rank, CART_COMM, &requests[counts++]);
      MPI_Irecv(&u_current[1][0], 1, column, west, west, CART_COMM, &requests[counts++]);
}
if(south >= 0){
      MPI_Isend(&u_current[local[0]][1], 1, row, south, rank, CART_COMM, &requests[counts++]);
      MPI_Irecv(&u_current[local[0]+1][1], 1, row, south, south, CART_COMM, &requests[counts++]);
}
```

```
if(east >=0){
       MPI_Isend(&u_current[1][local[1]], 1, column, east, rank, CART_COMM, &requests[counts++]);
       MPI_Irecv(&u_current[1][local[1]+1], 1, column, east, east, CART_COMM, &requests[counts++]);
}

MPI_Waitall(counts, 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;
```

2.3

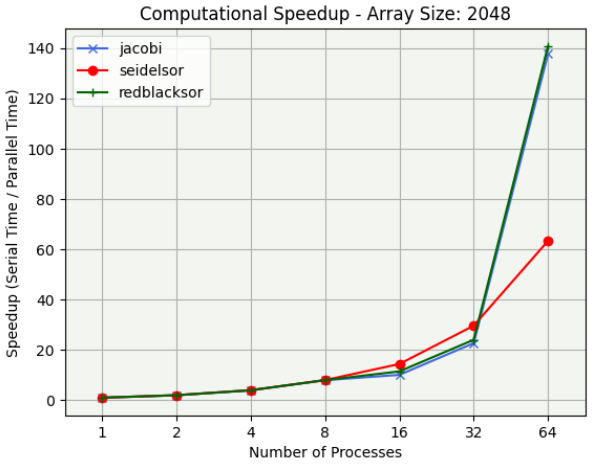
Πραγματοποιείστε μετρήσεις επίδοσης με βάση συγκεκριμένο σενάριο που θα σας δοθεί στο μάθημα.

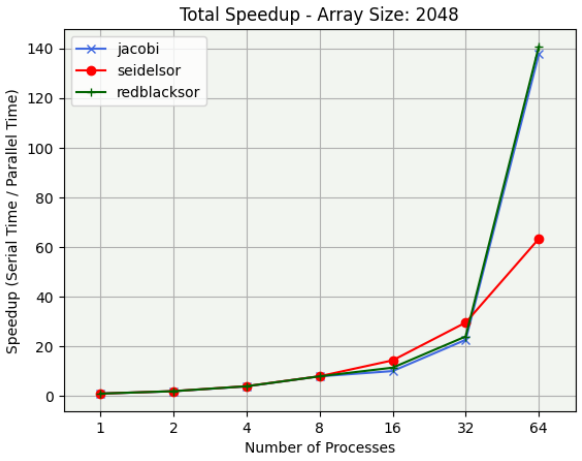
Μετρήσεις με έλεγχο σύγκλισης

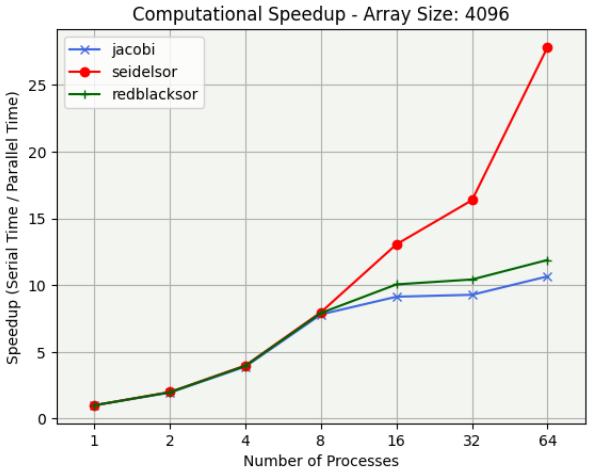


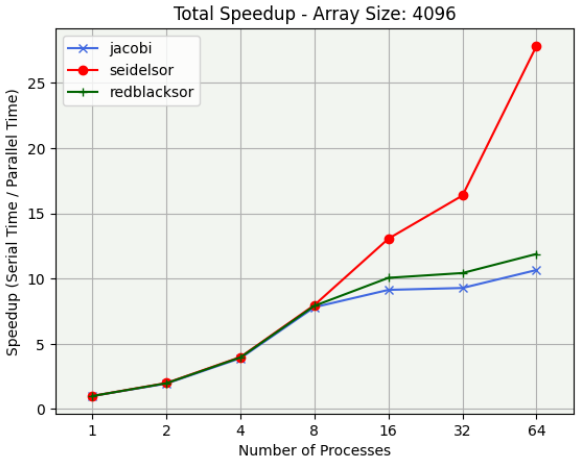
Μετρήσεις χωρίς έλεγχο σύγκλισης

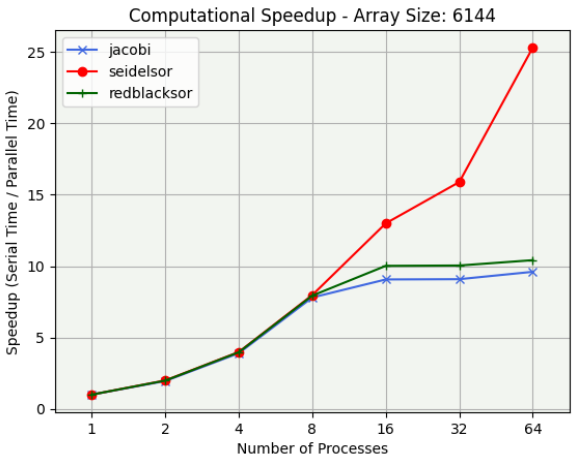
Διαγράμματα Speedup (Computational και Total)

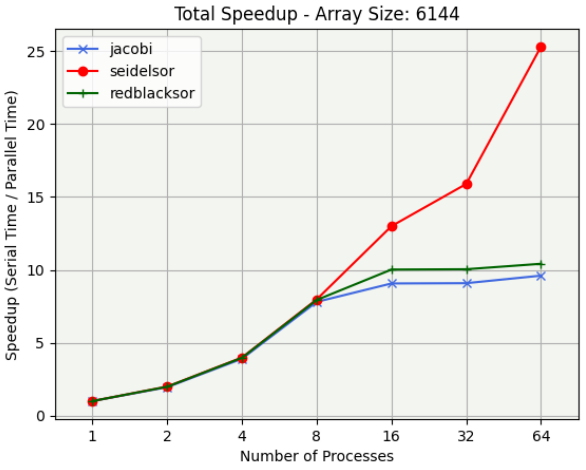




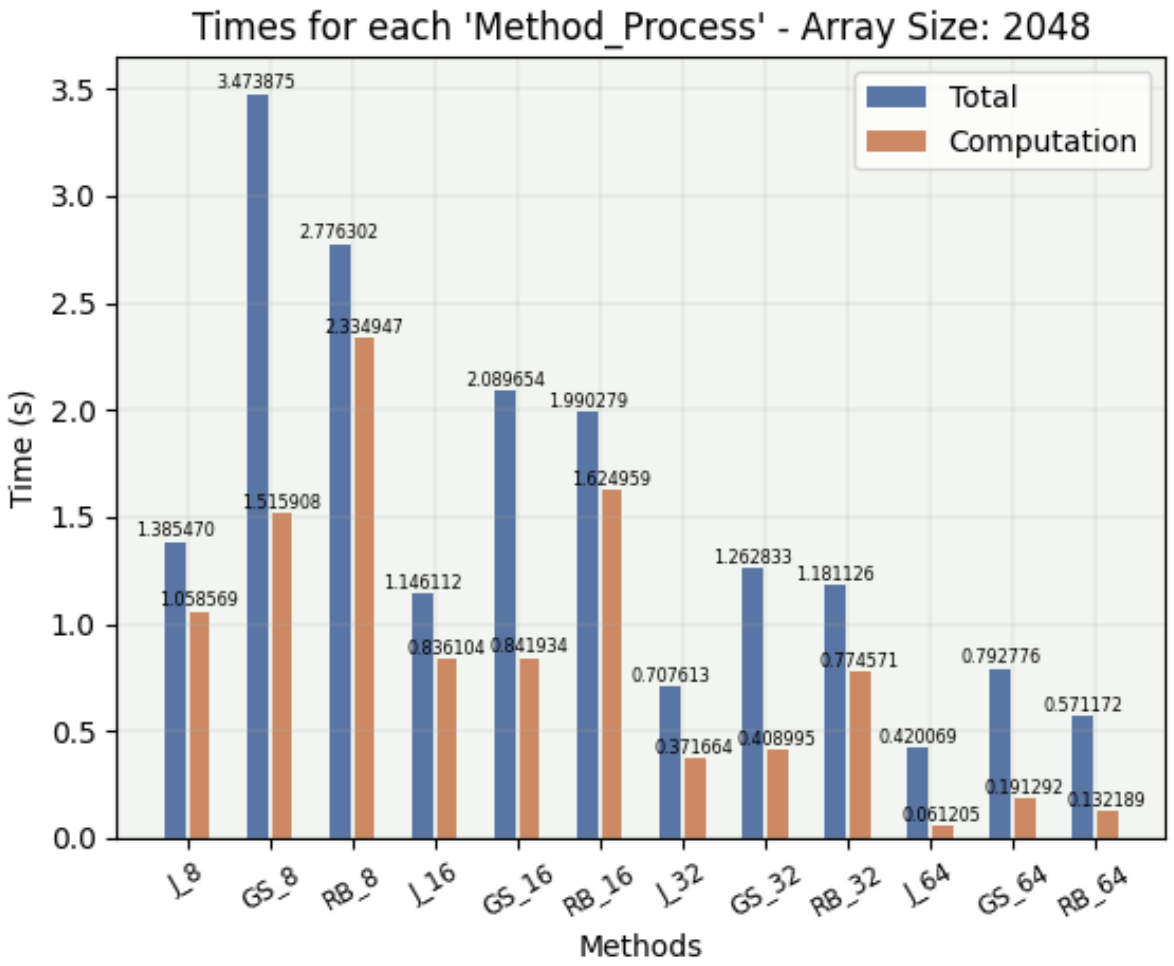


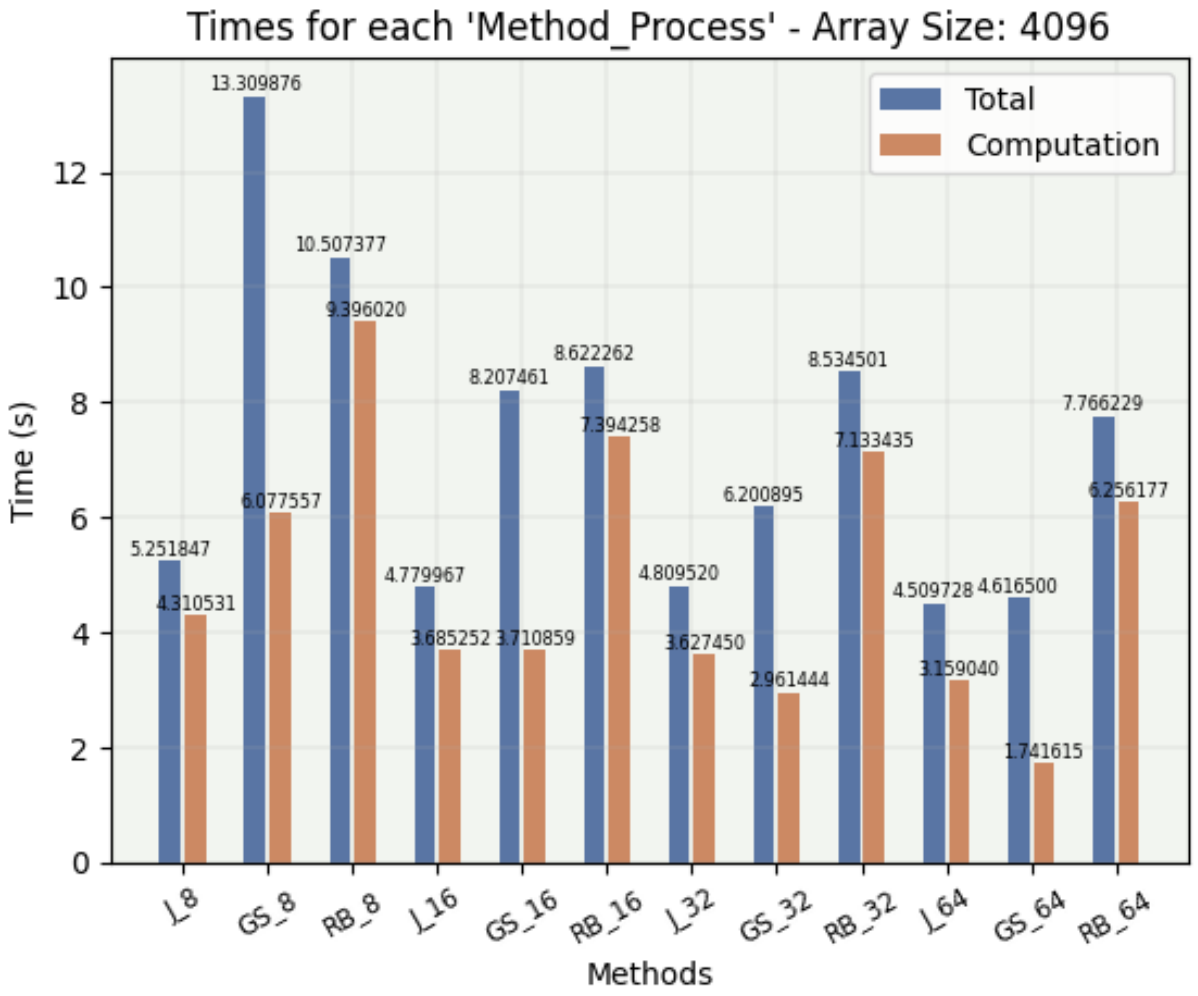


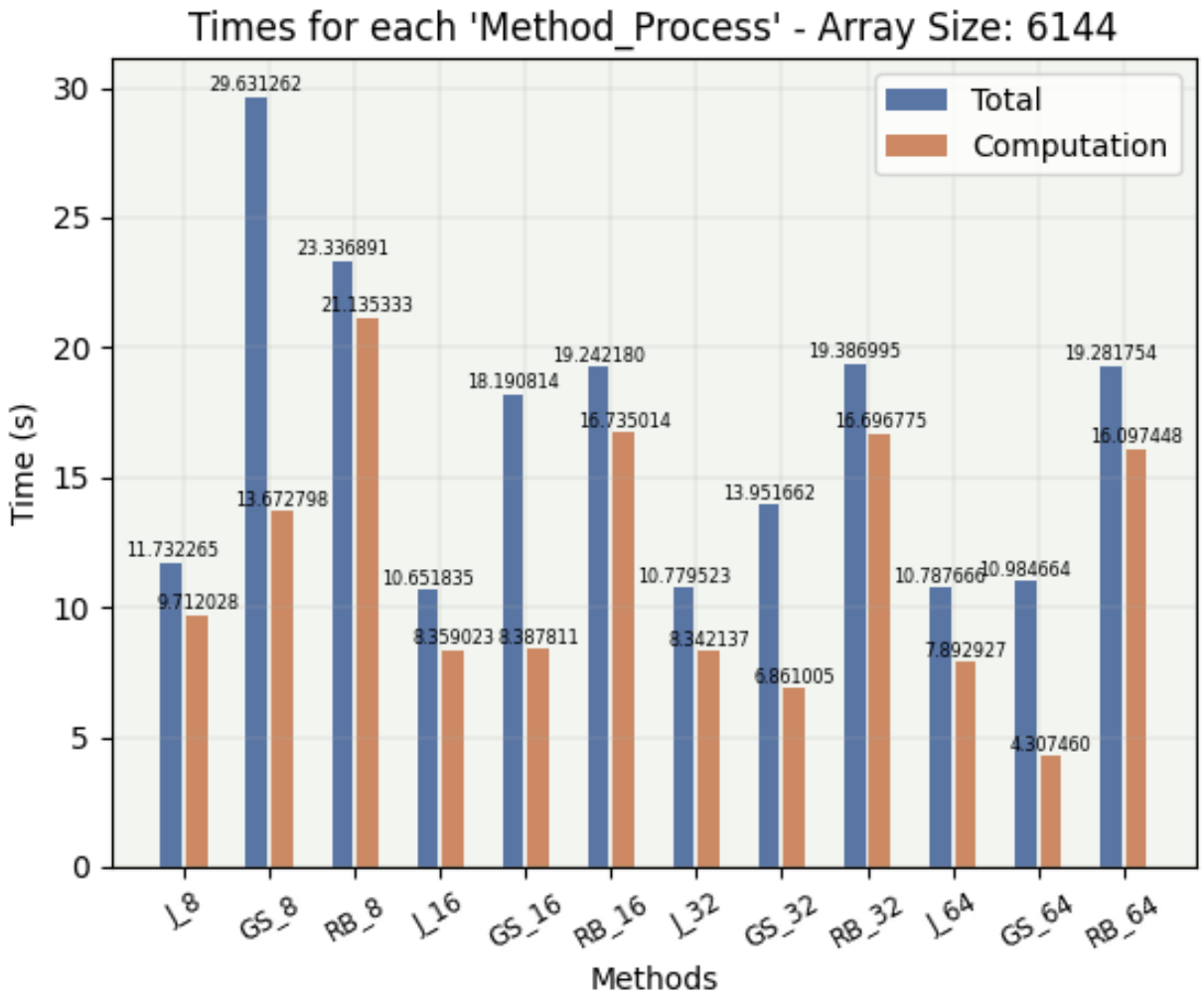




Διαγράμματα Χρόνου




Times for each 'Method_Process' - Array Size: 4096
Total
Computation
Time (s)
Methods
0
2
4
6
8
10
12
J_8
GS_8
RB_8
J_16
GS_16
RB_16
J_32
GS_32
RB_32
J_64
GS_64
RB_64
5.251847
4.310531
13.309876
6.077557
10.507377
9.396020
4.779967
3.685252
8.207461
3.710859
8.622262
7.394258
4.809520
3.627450
6.200895
2.961444
8.534501
7.133435
4.509728
3.159040
4.616500
1.741615
7.766229
6.256177


Times for each 'Method_Process' - Array Size: 6144
Total
Computation
Time (s)
Methods
0
5
10
15
20
25
30
11.732265
9.712028
29.631262
13.672798
23.336891
21.135333
10.651835
8.359023
18.190814
8.387811
19.242180
16.735014
10.779523
8.342137
13.951662
6.861005
19.386995
16.696775
10.787666
7.892927
10.984664
4.307460
19.281754
16.097448
J_8
GS_8
RB_8
J_16
GS_16
RB_16
J_32
GS_32
RB_32
J_64
GS_64
RB_64

2.4

Συγκεντρώστε τα αποτελέσματα, τις συγκρίσεις και τα σχόλιά σας στην Τελική Αναφορά.

Παρατηρήσεις με έλεγχο σύγκλισης

Παρατηρούμε ότι, ο Gauss-Seidel και ο Red-Black έχουν πολύ ταχύτερους χρόνους σύγκλισης από τον Jacobi. Πιο συγκεκριμένα είναι 2 τάξης μεγέθους καλύτεροι οι χρόνοι σύγκλισεις αυτών των υλοποιήσεων.
Σε σχέση με τους Gauss-Seidel, Red-Black, βλέπουμε ότι ο δεύτερος πετυχαίνει καλύτερους χρόνους σε σχέση με τον πρώτο.

Παρατηρήσεις χωρίς έλεγχο σύγκλισης

Αρχικά διαπιστώνουμε ότι το speedup σε συνολικό αλλά και υπολογιστικό χρόνο έχει την ίδια συμπεριφορά.

Παρατηρούμε αρχικά ότι, όλοι οι αλγόριθμοι από 8 διεργασίες και κάτω, έχουν την ίδια κλιμάκωση για όλα τα μεγέθη πίνακα.
Για περισσότερες από 8 διεργασίες, αρχίζει και διαφοροποιείται η συμπεριφορά του κάθε αλγορίθμου.
Παρατηρούμε ότι, για το μικρό μέγεθος πίνακα, ο Jacobi και ο Red-Black έχουν το ίδιο speedup και για τις 64 διεργασίες πετυχαίνουν πολύ καλύτερη απόδοση από του Gauss-Seidel, ο οποίος όμως για 16, 32 διεργασίες έχει ελάχιστα καλύτερη κλιμάκωση από τους άλλους δύο.
Όμως, για τα μεγαλύτερα μεγέθη πινάκων, βλέπουμε ότι ο Jacobi και ο Red-Black δεν κλιμακώνουν καλά μετά τις 16 διεργασίες, ενώ ο Gauss-Seidel σταδιακά αυξανόμενη κλιμάκωση. Επίσης, σημειωτέον είναι ότι ο Gauss-Seidel στην αύξηση των διεργασιών από 32 σε 64, πετυχαίνει την καλύτερη κλιμάκωση από οποιαδήποτε άλλο διπλασιασμό διεργασιών.

Σχετικά με τους χρόνους εκτέλεσης των αλγορίθμων, παρόλο που ο Jacobi δεν είχε την καλύτερη κλιμάκωση, επιτυγχάνει τους καλύτερους συνολικούς χρόνους σε όλα τα μεγέθη και ειδικά για μικρότερο πλήθος διεργασιών. Παρόλα αυτά, ο Gauss-Seidel πετυχαίνει γρηγορότερους χρόνους υπολογισμού κυρίως για τα μεγαλύτερα πλήθη διεργασιών. Επίσης στα μεγαλύτερα πλήθη διεργασιών πετυχαίνει σχεδόν ίδιους συνολικούς χρόνους με τον Jacobi.

Συνολικά, δεδομένου της σημαντικά καλύτερης κλιμακωσιμότητας και υπολογιστικού χρόνου που πετυχαίνει ο Gauss-Seidel, μπορούμε να τον διαλέξουμε ως την βέλτιστη επιλογή. Παρατηρώντας τις μετρήσεις βλέπουμε ότι το bottleneck του, βρίσκεται στην επικοινωνία μεταξύ των διεργασιών καθώς έχει τους καλύτερους υπολογιστικούς χρόνους, αλλά όχι πάντα και τους συνολικούς. Με κάποιο άλλο πρότυπο ή βελτίωση του ήδη υπάρχοντος στο κομμάτι της επικοινωνίας μεταξύ των διεργασιών ή και βελτίωση του bandwidth και του πρωτοκόλλου επικοινωνίας θα μπορούσαμε να εκμεταλλευτούμε πιο αποδοτικά τα παραπάνω χαρακτηριστικά του αλγορίθμου.