ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ροή Υ : Συστήματα Παράλληλης Επεξεργασίας

3η Άσκηση (Τελική Αναφορά)

Παράλληλη επίλυση εξίσωσης θερμότητας σε Αρχιτεκτονικές Κατανεμημένης Μνήμης με μοντέλο ανταλλαγής μηνυμάτων

Κριθαρούλας Διονύσιος 03117875

Ομάδα:Parlab18

Εξάμηνο: 9ο

Εισαγωγή

Στην συγκεκριμένη εργαστηριακή άσκηση ζητήθηκε η παραλληλοποίηση τριών υπολογιστικών πυρήνων οι οποίοι επιλύουν το πρόβλημα της διάδοσης της θερμότητας. Οι πυρήνες αυτοί αποτελούν ευρέως διαδεδομένη δομική μονάδα για την επίλυση μερικών διαφορικών εξισώσεων: η μέθοδος Jacobi, η μέθοδος Gauss-Seidel με Successive Over Relaxation και η μέθοδος Red-Black SOR, που πραγματοποιεί Red-Black ordering στα στοιχεία του υπολογιστικού χωρίου και συνδυάζει τις δύο προηγούμενες μεθόδους.

Στην συνέχεια αναλύεται η κάθε μέθοδος ξεχωριστά:

1.1 Μέθοδος Jacobi

Η μέθοδος Jacobi βρίσκει σε κάθε χρονικό βήμα την τιμή ενός στοιχείου του χωρίου υπολογίζοντας τον μέσο όρο των τιμών των τεσσάρων γειτονικών στοιχείων του. Οι τέσσερις γειτονικές τιμές αναφέρονται στην προηγούμενη χρονική στιγμή. Έτσι ο υπολογισμός της τιμής ενός σημείου x,y την χρονική στιγμή t+1 υπολογίζεται με βάση τον ακόλουθο τύπο:

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

Η υλοποίηση της συγκεκριμένης μεθόδου δίνεται στην συνέχεια:

```
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 σε κάθε χρονικό βήμα υπολογισμού χρειαζόμαστε την τιμή των τεσσάρων γειτονικών στοιχείων την προηγούμενη χρονική στιγμή. Για τον λόγο αυτό χρησιμοποιούνται δύο πίνακες, ένας για την τρέχουσα χρονική στιγμή (current array) και ένας για την προηγούμενη χρονική στιγμή (previous array). Στο τέλος κάθε χρονικού βήματος πραγματοποιείται έλεγχος σύγκλισης (ο έλεγχος βέβαια αυτός μπορεί να απενεργοποιηθεί και ο αλγόριθμος να εκτελεστεί για σταθερό αριθμό βημάτων).

1.2 Μέθοδος Gauss-Seidel SOR

Η μέθοδος Gauss-Seidel SOR χρησιμοποιείται προκειμένου να επιταχύνει τον αργό ρυθμό σύγκλισης που παρουσιάζει η μέθοδος Jacobi. Η κύρια ιδέα της μεθόδου αυτής είναι πως όταν φθάσει η ώρα υπολογισμού του στοιχείου u[i][j] για το χρονικό βήμα t έχουν ήδη υπολογισθεί οι τιμές των στοιχείων u[i-1][j] και u[i][j-1] για το ίδιο χρονικό βήμα t (επάνω και αριστερά γείτονας). Αντίθετα για τα στοιχεία u[i+1][j] και u[i][j+1] δεν έχουν ακόμα υπολογισθεί οι νέες τιμές για το χρονικό βήμα t. Έτσι χρησιμοποιώντας στην ανανέωση του u[i][j] τις νέες τιμές των u[i-1][j] και u[i][j-1] μπορούμε να οδηγηθούμε πιο γρήγορα σε σύγκλιση. Η ανανέωση λοιπόν των στοιχείων σύμφωνα με την μέθοδο Gauss-Seidel ακολουθεί τον παρακάτω τύπο:

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

Για ακόμα μεγαλύτερη επιτάχυνση του ρυθμού σύγκλισης χρησιμοποιείται μαζί με την μέθοδο Gauss-Seidel η τεχνική Successive Over-Relaxation (SOR) η οποία χρησιμοποιεί και την τιμή του ίδιου του στοιχείου u[i][j] την προηγούμενη χρονική στιγμή. Ο τελικός λοιπόν τύπος που χρησιμοποιείται από την μέθοδο Gauss-Seidel SOR για την ανανέωση των στοιχείων ενός πίνακα σε κάθε χρονικό βήμα είναι ο ακόλουθος:

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

Η υλοποίηση της συγκεκριμένης μεθόδου δίνεται στην συνέχεια:

```
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])
 }
```

Παρατηρούμε ότι και στην συγκεκριμένη μέθοδο υπάρχει η ανάγκη χρήσης δύο πινάκων, έναν για την τωρινή και έναν για την προηγούμενη χρονική στιγμή. Όπως

και στην μέθοδο Jacobi έτσι και εδώ στο τέλος κάθε χρονικού βήματος πραγματοποιείται έλεγχος σύγκλισης (ο έλεγχος και σε αυτήν την περίπτωση μπορεί να απενεργοποιηθεί και ο αλγόριθμος να εκτελεστεί για σταθερό αριθμό βημάτων).

1.3 Μέθοδος Red-Black SOR

Το Red-Black SOR είναι μία μέθοδος που σκοπό έχει να επιταχύνει ακόμα περισσότερο τον ρυθμό σύγκλισης. Η μέθοδος αυτή λειτουργεί σε δύο φάσεις. Στην πρώτη φάση υπολογίζονται οι νέες τιμές των στοιχείων τα οποία βρίσκονται στις θέσεις [i][j] για τις οποίες ισχύει (i+j)%2=0. Η ανανέωση των στοιχείων αυτών γίνεται με χρήση της μεθόδου Gauss-Seidel SOR χρησιμοποιώντας τις τιμές των τεσσάρων γειτονικών τους στοιχείων (black στοιχεία) από την προηγούμενη χρονική στιγμή. Τα στοιχεία αυτά είναι τα κόκκινα στοιχεία (Red) και ο τύπος ανανέωσης των τιμών τους για τον χρονικό βήμα t+1 δίνεται από τον ακόλουθο τύπο:

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

Στην δεύτερη φάση υπολογίζονται οι νέες τιμές των στοιχείων τα οποία βρίσκονται στις θέσεις [i][j] για τις οποίες ισχύει (i+j)%2=1. Η ανανέωση των στοιχείων αυτών γίνεται με χρήση της μεθόδου Gauss-Seidel SOR χρησιμοποιώντας όμως τις τιμές των τεσσάρων γειτονικών τους στοιχείων (red στοιχεία) από την τωρινή χρονική στιγμή τα οποία έχουν υπολογισθεί στην πρώτη φάση. Τα στοιχεία αυτά είναι τα μαύρα στοιχεία (Black) και ο τύπος ανανέωσης των τιμών τους για τον χρονικό βήμα t+1δίνεται από τον ακόλουθο τύπο:

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

Με βάση την παραπάνω μέθοδο πετυχαίνουμε λοιπόν καλύτερη ταχύτητα σύγκλισης. Η υλοποίηση της συγκεκριμένης μεθόδου δίνεται στην συνέχεια:

```
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])
}
```

Στην παραπάνω υλοποίηση φαίνεται καθαρά η ύπαρξη δύο φάσεων (Red και Black Phase) τις οποίες αναλύσαμε προηγουμένως ενώ όπως και στις προηγούμενες δύο μεθόδους παρατηρείται η ανάγκη χρήσης δύο πινάκων, έναν για την τωρινή και έναν για την προηγούμενη χρονική στιγμή όπου και πάλι στο τέλος κάθε χρονικού βήματος πραγματοποιείται έλεγχος σύγκλισης (ο έλεγχος και σε αυτήν την περίπτωση μπορεί να απενεργοποιηθεί και ο αλγόριθμος να εκτελεστεί για σταθερό αριθμό βημάτων).

2. Ζητούμενα

2.1

Στο συγκεκριμένο βήμα της εργασίας ανακαλύπτουμε τον παραλληλισμό του καθενός από τους τρεις παραπάνω αλγόριθμους και σχεδιάζουμε την παραλληλοποίηση του σε αρχιτεκτονικές κατανεμημένης μνήμης με μοντέλο ανταλλαγής μηνυμάτων. Αρχικά και για τις τρεις μεθόδους θεωρούμε τα ακόλουθα:

Ο πίνακας U είναι γενικά διάστασης NxM. **Θεωρούμε ως task τον υπολογισμό των στοιχείων ενός block διάστασης nxm του πίνακα U για ένα χρονικό βήμα t (κατανομή υπολογισμών).** Για απλότητα στην σχεδίαση της παραλληλοποίησης θεωρούμε πως N mod n = 0 , N/n = 4 και αντίστοιχα πως M mod m = 0 , M/m = 4. Με άλλα λόγια θεωρούμε πως οι διαστάσεις nxm του κάθε block διαιρούν ακριβώς τις διαστάσεις NxM του πίνακα U και μάλιστα πως το αποτέλεσμα της διαίρεσης είναι 4. Σε διαφορετική περίπτωση προβλέπεται padding του πίνακα U προκειμένου κάθε block να έχει τις ίδιες διαστάσεις. Με βάση την παραπάνω υπόθεση ο πίνακας U έχει την ακόλουθη μορφή:

$$\begin{bmatrix} U_{00} & U_{01} & U_{02} & U_{03} \\ U_{10} & U_{11} & U_{12} & U_{13} \\ U_{20} & U_{21} & U_{22} & U_{23} \\ U_{30} & U_{31} & U_{32} & U_{33} \end{bmatrix}$$

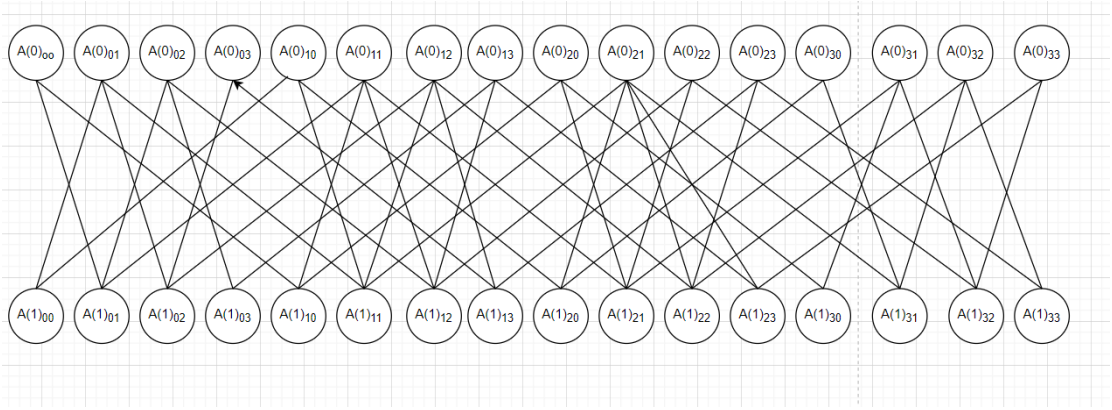
Όπου U_{ij} αντιστοιχεί στο block ij του πίνακα U.

Στην συνέχεια σχεδιάζουμε το task-graph για κάθε μία από τις τρεις μεθόδους με βάση τις παραπάνω υποθέσεις:

Μέθοδος Jacobi

Ορισμός σωστής σειράς εκτέλεσης μέσω task-graph

Στην μέθοδο Jacobi όπως είδαμε και στην αρχική παρουσίαση της μεθόδου κάθε στοιχείο ανανεώνει την τιμή του σε κάθε χρονικό βήμα χρησιμοποιώντας τις τιμές που έχουν τα 4 γειτονικά του στοιχεία (βόρειο , νότιο , ανατολικό , δυτικό) στο προηγούμενο χρονικό βήμα. Επομένως κάθε block προκειμένου να υπολογίσει τις νέες τιμές των στοιχείων του χρειάζεται τις τιμές των γειτονικών του block στο προηγούμενο χρονικό βήμα. Έτσι προκύπτει το ακόλουθο task-graph για την μέθοδο Jacobi:



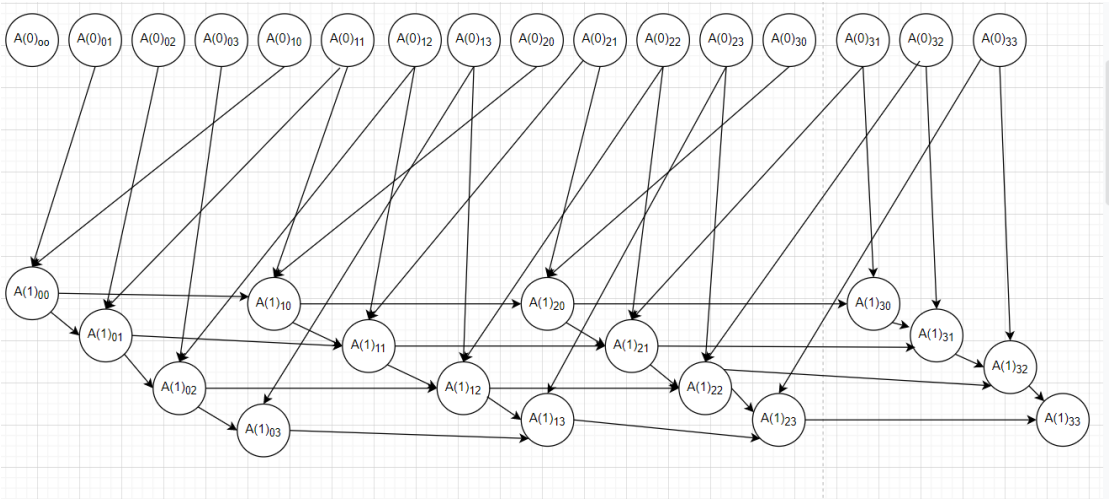
Σημείωση: Το task_graph αυτό επεκτείνεται με τον ίδιο τρόπο και για παραπάνω χρονικά βήματα εκτέλεσης. Το νούμερο μέσα στην παρένθεση δείχνει το χρονικό βήμα εκτέλεσης ενώ ο δείκτης την θέση του block στον δισδιάστατο πίνακα U.

Σύμφωνα και με το παραπάνω task-graph παρατηρούμε πως μετά τις απαραίτητες ανταλλαγές τιμών μεταξύ των γειτονικών block ο υπολογισμός των νέων τιμών σε κάθε block μπορεί να γίνει παράλληλα για ένα χρονικό βήμα εκτέλεσης. Δεν μπορεί όμως να υπάρξει παραλληλία μεταξύ των χρονικών βημάτων καθώς σε κάθε νέο χρονικό βήμα χρειαζόμαστε τιμές από το προηγούμενο χρονικό βήμα προκειμένου να πραγματοποιήσουμε την ανανέωση.

Μέθοδος Gauss-Seidel SOR

Ορισμός σωστής σειράς εκτέλεσης μέσω task-graph

Στην μέθοδο Gauss-Seidel SOR όπως είδαμε και στην αρχική παρουσίαση της μεθόδου οι εξαρτήσεις μεταξύ των στοιχείων του πίνακα U είναι πιο περίπλοκες. Κάθε στοιχείο προκειμένου να ανανεώσει την τιμή του σε ένα χρονικό βήμα χρειάζεται τις τιμές του βόρειου (πάνω) και δυτικού (αριστερού) γείτονα του στο ίδιο χρονικό βήμα καθώς και τις τιμές του νότιου (κάτω) και ανατολικού (αριστερά) γείτονα του από το προηγούμενο χρονικό βήμα υπολογισμού αλλά και την τιμή που είχε το ίδιο το στοιχείο στο προηγούμενο χρονικό βήμα. Έτσι λοιπόν και κάθε block προκειμένου να υπολογίσει τις νέες τιμές των στοιχείων του χρειάζεται τις τωρινές (ανανεωμένες) τιμές του βόρειου και δυτικού block με το οποίο συνορεύει καθώς και τις τιμές της προηγούμενης χρονικής στιγμής του νότιου και ανατολικού block με το οποίο συνορεύει. Έτσι προκύπτει το ακόλουθο task-graph για την μέθοδο Gauss-Seidel SOR:



Σημείωση: Το task_graph αυτό επεκτείνεται με τον ίδιο τρόπο και για παραπάνω χρονικά βήματα εκτέλεσης. Το νούμερο μέσα στην παρένθεση δείχνει το χρονικό βήμα εκτέλεσης ενώ ο δείκτης την θέση του block στον δισδιάστατο πίνακα. Επιπλέον έχουν χρησιμοποιηθεί κατευθυνόμενες ακμές για να φαίνεται ξεκάθαρα η ροή των δεδομένων που μεταφέρονται μεταξύ των διεργασιών.

Και στην μέθοδο αυτή λοιπόν με βάση και το παραπάνω task-graph παρατηρείται πως δεν μπορεί να υπάρξει παράλληλή εκτέλεση μεταξύ των χρονικών βημάτων καθώς σε κάθε χρονικό βήμα χρειαζόμαστε τιμές τόσο από το προηγούμενο όσο και από το τρέχων χρονικό βήμα. Παράλληλα βλέπουμε πως η επικοινωνία μεταξύ των διεργασιών έχει γίνει πιο περίπλοκη σε σχέση με την μέθοδο Jacobi καθώς οι εξαρτήσεις που πρέπει να ικανοποιηθούν έχουν αλλάξει.

Μέθοδος Red-Black SOR

Ορισμός σωστής σειράς εκτέλεσης μέσω task-graph

Στην μέθοδο Red-Black SOR όπως είδαμε και στην αρχική παρουσίαση της μεθόδου τα στοιχεία ανάλογα με την θέση [i][j] στην οποία βρίσκονται έχουν διαφορετικές εξαρτήσεις. Πιο συγκεκριμένα έχουμε τις δύο ακόλουθες περιπτώσεις:

- Τα στοιχεία για τα οποία ισχύει (i+j)%2 = 0 (Red) χρειάζονται τις τιμές των $4^{ων}$ γειτονικών τους στοιχείων (Black) από το προηγούμενο χρονικό βήμα προκειμένου να ανανεώσουν την τιμή τους.
- Τα στοιχεία για τα οποία ισχύει (i+j)%2 = 1 (Black) χρειάζονται τις τιμές των $4^{ων}$ γειτονικών τους στοιχείων (Red) από το τρέχων χρονικό βήμα προκειμένου να ανανεώσουν την τιμή τους.

Παρατηρούμε λοιπόν πως ουσιαστικά πριν από κάθε φάση υπολογισμού πρέπει ένα block να λάβει τα απαραίτητα στοιχεία από όλους τους γείτονες του όπως ακριβώς έκανε και στην μέθοδο Jacobi. Μόνο που πριν την πρώτη φάση τα δεδομένα που θα λάβει θα αφορούν την προηγούμενη χρονική στιγμή ενώ πριν την δεύτερη φάση θα αφορούν την τρέχουσα χρονική στιγμή. Έτσι λοιπόν κάθε block σε ένα χρονικό βήμα για να υπολογίσει τα στοιχεία του που βρίσκονται σε άρτιες θέσεις χρειάζεται να πάρει τα απαραίτητα στοιχεία του προηγούμενου χρονικού βήματος από τους γείτονες τους ενώ για να υπολογίσει τα στοιχεία του που βρίσκονται σε περιττές θέσεις χρειάζεται να πάρει τα απαραίτητα στοιχεία του τρέχοντος χρονικού βήματος από τους γείτονες τους.

Και στην μέθοδο αυτή λοιπόν παρατηρείται πως δεν μπορεί να υπάρξει παράλληλη εκτέλεση μεταξύ των χρονικών βημάτων καθώς σε κάθε χρονικό βήμα χρειαζόμαστε τιμές τόσο από το προηγούμενο όσο και από το τρέχων χρονικό βήμα. Η επικοινωνία σε αυτήν την περίπτωση λοιπόν δεν είναι πιο περίπλοκη από αυτήν που πραγματοποιήθηκε στην μέθοδο Jacobi αλλά χρειάζεται να γίνει δύο φορές.

Παρατήρηση: Στα παραπάνω task-graph έχει παραληφθεί η εξάρτηση που υπάρχει μεταξύ ενός block στο χρονικό βήμα 1 και της τιμής του στο χρονικό βήμα 0. Επειδή και στις τρεις μεθόδους κάθε block για να ανανεώσει τις τιμές των στοιχείων του στο χρονικό βήμα 1 χρειάζεται τις τιμές των στοιχείων του στο προηγούμενο χρονικό βήμα η εξάρτηση αυτή υπονοείται και δεν προστέθηκε για πιο καθαρά αποτελέσματα.

Στην συνέχεια έχουμε τα τελευταία δύο βήματα του σχεδιασμού της παραλληλοποίησης τα οποία είναι κοινά και για τις τρεις μεθόδους:

Οργάνωση Πρόσβασης στα δεδομένα

Εφόσον δουλεύουμε σε αρχιτεκτονική κατανεμημένης μνήμης η λογική απόφαση είναι σε κάθε task (διεργασία) να ανατίθενται τα δεδομένα του block στο οποίο και εκτελεί τους υπολογισμούς. Έτσι η επικοινωνία μεταξύ των tasks (μέσω μοντέλου ανταλλαγής μηνυμάτων) θα γίνεται προκειμένου να ανταλλαχθούν τα απαραίτητα δεδομένα και να καλυφθούν οι εξαρτήσεις που προκύπτουν σε κάθε μία μέθοδο και οι οποίες παρουσιάστηκαν προηγουμένως.

Ανάθεση Εργασιών σε οντότητες εκτέλεσης

Στο συγκεκριμένο στάδιο γίνεται η επιλογή ο αριθμός των block στα οποία χωρίζεται ο αρχικός πίνακας U να είναι ίσος με τον αριθμό των οντοτήτων εκτέλεσης – νημάτων τα οποία διαθέτουμε. Έτσι κάθε νήμα αναλαμβάνει τον υπολογισμό των τιμών ενός συγκεκριμένου block για όλα τα χρονικά βήματα εκτέλεσης. Εφόσον όπως είδαμε δεν μπορεί να υπάρξει παραλληλία μεταξύ των χρονικών βημάτων εκτέλεσης σε καμία από τις τρεις μεθόδους μία τέτοια απόφαση είναι λογική. Ο αριθμός λοιπόν των block στα οποία χωρίζεται ο πίνακας U εξαρτάται από τον αριθμό των οντοτήτων εκτέλεσης-νημάτων που διαθέτουμε.

2. Ανάπτυξη παράλληλων προγραμμάτων

Στο συγκεκριμένο βήμα αναλύουμε τις παράλληλες υλοποιήσεις μας για την κάθε μία από τις τρεις μεθόδους.

Αρχικά επεξηγούμε το πρώτο κομμάτι του κώδικα το οποίο είναι κοινό και στις τρεις υλοποιήσεις και το οποίο μας είχε δοθεί έτοιμο ως σκελετός. Έτσι έχουμε:

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

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

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

Στο συγκεκριμένο κομμάτι κώδικα ορίζουμε τις βασικές μεταβλητές οι οποίες θα χρησιμοποιηθούν στην συνέχεια της υλοποίησης. Επεξηγούμε τις πιο κύριες:

- *rank:* ο μοναδικός βαθμός (αναγνωριστικό) που θα αποδοθεί σε κάθε διεργασία που αποτελεί μέρος του communicator MPI_COMM_WORLD.
- *size:* ο αριθμός των διεργασιών που συμμετέχουν στον communicator COMM_WORLD.

- *global[2]:* Οι διαστάσεις του global πίνακα U.
- *local[2]:* Οι διαστάσεις του κάθε block το οποίο αναλαμβάνει μία διεργασία.
- *global_padded[2]:* Οι διαστάσεις του global πίνακα U μετά από padding το οποίο χρειάζεται στις περιπτώσεις που ο global πίνακας U δεν μπορεί να χωριστεί σε block ίδιων διαστάσεων ώστε να μοιραστούν στις διεργασίες.
- *grid[2]:* Οι διαστάσεις του grid όπου χωρίζεται ο global πίνακας U. Το γινόμενο grid[0]*grid[1] δίνει το σύνολο των νημάτων εκτέλεσης.
- *global_converged=0:* Η μεταβλητή αυτή θα ισούται με 1 (True) στην περίπτωση σύγκλισης όλου του χωρίου.
- *converged=0:* Η μεταβλητή αυτή θα ισούται με 1 (True) στην περίπτωση σύγκλισης του τοπικού block το οποίο επεξεργάζεται η κάθε διεργασία.
- *MPI_Datatype dummy:* Νέο Datatype που θα χρησιμοποιηθεί για να κάνει align user-defined datatypes στην μνήμη (χρειάζεται στις περιπτώσεις χρήσης collective operations όπως Gather , Scatter κ.λ.π.).
- *double omega:* Το ω που είδαμε στους τύπους των μεθόδων Gauss-Seidel SOR και Red-Black SOR.
- *struct timeval tts:* time total start. Η χρονική στιγμή όπου αρχίζει η εκτέλεση των υπολογισμών και της επικοινωνίας μίας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *struct timeval ttf:* time total finish. Η χρονική στιγμή όπου τελειώνει η εκτέλεση των υπολογισμών και της επικοινωνίας μιας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *struct timeval tcs:* time computational start. Η χρονική στιγμή όπου αρχίζει η εκτέλεση μόνο των υπολογισμών μιας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *struct timeval tcf:* time computational finish. Η χρονική στιγμή όπου τελειώνει η εκτέλεση μόνο των υπολογισμών μιας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *struct timeval tcvs (προστέθηκε από εμάς):* time converged start. Η χρονική στιγμή όπου αρχίζει ο έλεγχος σύγκλισης για μια διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *struct timeval tcvf (προστέθηκε από εμάς):* time converged finish. Η χρονική στιγμή όπου τελειώνει ο έλεγχος σύγκλισης για διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *double ttotal:* time total. Η χρονική διάρκεια που κράτησαν τόσο οι υπολογισμοί όσο και η επικοινωνία αλλά και ο έλεγχος σύγκλισης μιας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *double tcomp:* time computational. Η χρονική διάρκεια που κράτησαν οι υπολογισμοί μιας διεργασίας (τοπική μεταβλητή της κάθε διεργασίας).
- *double tconv (προστέθηκε από εμάς):* time converged. Η χρονική διάρκεια που κράτησε ο έλεγχος σύγκλισης για μια διεργασία (τοπική μεταβλητή της κάθε διεργασίας).

- *double total_time:* Η χρονική διάρκεια που κράτησαν οι υπολογισμοί , η επικοινωνία και ο έλεγχος σύκλισης για τον υπολογισμό των τελικών τιμών του πίνακα U (λαμβάνεται η max τιμή από όλες τις ttotal τοπικές μεταβλητές).
- *double comp_time:* Η χρονική διάρκεια που κράτησαν οι υπολογισμοί του global πίνακα U (λαμβάνεται η max τιμή από όλες τις tcomp τοπικές μεταβλητές).
- *double conv_time (προστέθηκε από εμάς):* Η χρονική διάρκεια που κράτησε ο έλεγχος σύγκλισης για όλον τον global πίνακα U (λαμβάνεται η max τιμή από όλες τις tconv τοπικές μεταβλητές).

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

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

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

Στο συγκεκριμένο πρόγραμμα αρχικοποιούμε το περιβάλλον εκτέλεσης του MPI μέσω της κλίσης MPI_Init. Έπειτα μέσω της κλίσης MPI_Comm_size και MPI_Comm_rank δίνουμε στις μεταβλητές size και rank τις τιμές που εξηγήσαμε προηγουμένως. Τέλος οι μεταβλητές global[0] , global[1] , grid[0] , grid[1] αποκτούν τις τιμές που δίνονται κατά την κλήση του εκτελέσιμου.

```
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
```

Στο συγκεκριμένο κομμάτι κώδικα μέσω της κλήσης MPI_Cart_create δημιουργούμε έναν νέο Καρτεσιανό Communicator CART_COMM για την ευκολότερη επικοινωνία μεταξύ των διεργασιών κάθε μία από τις οποίες θα αναλάβει ένα συγκεκριμένο block του πίνακα U. Επιπλέον μέσω της κλίσης MPI_Cart_coords βρίσκουμε από το rank της διεργασίας στον communicator MPI_COMM_WORLD , τον βαθμό της διεργασίας rank_grid στον Καρτεσιανό communicator CART_COMM στον οποίο ανήκει. Ο νέος βαθμός rank_grid αποτελείται πλέον από μία δυάδα αριθμών που αντιστοιχούν στην

θέση της συγκεκριμένης διεργασίας στο καρτεσιανό πλέγμα που αναπαριστά τον πίνακα U.

```
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];
    }
}
```

Στο συγκεκριμένο κομμάτι κώδικα υπολογίζονται οι διαστάσεις του κάθε block που αναλαμβάνει μία διεργασία καθώς και οι νέες διαστάσεις του global πίνακα U σε περίπτωση που χρειαστεί padding. Πιο συγκεκριμένα εάν οι διαστάσεις grid[0] , grid[1] διαιρούν τέλεια τις αντίστοιχες διαστάσεις global[0],global[1] του πίνακα U τότε ισχύει local[i] = global[i] /grid[i] και δεν απαιτείται padding. Σε αντίθετη περίπτωση σε όποια διάσταση παρατηρηθεί υπόλοιπο από την διαίρεση τότε ισχύει local[i] = global[i]/grid[i] (πηλίκο της διαίρεσης) + 1 ενώ οι διαστάσεις του global πίνακα U στον οποίο πλέον έχει πραγματοποιηθεί padding είναι global_padded[i] = local[i]*grid[i].

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

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

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

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

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

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

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

Στο συγκεκριμένο κομμάτι κώδικα θέτουμε την τιμή που θα πάρει η μεταβλητή omega. Έπειτα μόνο η διεργασία με rank = 0 (main thread) κάνει allocate τον δισδιάστατο global πίνακα U μέσω της κλίσης της συνάρτησης allocate2d (η υλοποίηση της βρίσκεται στο αρχείο utils.c το οποίο μας δίνεται). Οι διαστάσεις του πίνακα που θα γίνει allocate στην μνήμη ορίζονται ως global_padded[0] , global_padded[1] συμπεριλαμβάνοντας και την περίπτωση όπου απαιτείται padding. Αντίθετα αρχικοποιείται μόνο το κομμάτι του πίνακα το οποίο δεν αποτελεί μέρος του padding οπότε και καλείται η συνάρτηση init2d (και αυτή ορίζεται στο αρχείο utils.c) με ορίσματα όμως τα global[0] και global[1] (και πάλι μόνο από την διεργασία με rank = 0).

Τέλος η κάθε διεργασία κάνει allocate δύο πίνακες u_previous και u_current οι οποίοι αναπαριστούν το block που αντιστοιχεί στην κάθε διεργασία διαστάσεων (local[0] +2) x (local[1] +2). Το u_previous χρησιμοποιείται για να περιέχει τις τιμές των στοιχείων του block που υπολογίστηκαν στο προηγούμενο χρονικό βήμα ενώ το u_current περιέχει τις τιμές των στοιχείων του block που υπολογίζονται στο τρέχων χρονικό βήμα κατά την διάρκεια εκτέλεσης του αλγορίθμου.

Παρατήρηση : Παρατηρούμε πως οι διαστάσεις των u_previous και u_current δεν είναι local[0] x local[1] αλλά αντίθετα είναι (local[0]+2)x(local[1]+2). Αυτό συμβαίνει καθώς σε κάθε block αντιστοιχούν δύο επιπλέον γραμμές (μία στην αρχή του block και μία στο τέλος) καθώς και δύο επιπλέον στήλες (μία στην αρχή του block και μία στο τέλος). Οι γραμμές και στήλες αυτές περιέχουν τα λεγόμενα ghost cells τα οποία χρησιμεύουν για να αποθηκεύουν τις τιμές των γειτονικών block τις οποίες χρειάζεται κάθε block προκειμένου να πραγματοποιήσει τους υπολογισμούς του (περισσότερα για τα ghost cells στην συνέχεια).

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

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

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

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

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

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

Το συγκεκριμένο κομμάτι κώδικα δημιουργεί δύο νέα MPI Datatypes.

Το πρώτο datatype ονομάζεται global block και αναφέρεται στο κάθε block του global πίνακα U. Δημιουργείται μέσω της κλήσης MPI_Type_Vector σύμφωνα με τα ορίσματα της οποίας ενώνονται local[0] δεδομένα (όσος και ο αριθμός των γραμμών που αντιστοιχούν στο block μίας διεργασίας) μεγέθους local[1] (όσος και ο αριθμός των στηλών που αντιστοιχούν στο block μίας διεργασίας) τα οποία έχουν μεταξύ τους stride global_padded[1] (όσος δηλαδή και ο αριθμός των στηλών του global πίνακα U μετά και από πιθανό padding). Τα δεδομένα που ενώνονται είναι τύπου MPI_DOUBLE.

Με αντίστοιχο τρόπο δημιουργείται και το Datatype local_block όπου μέσω της κλήσης MPI_Type_Vector ενώνονται local[0] δεδομένα (όσος και ο αριθμός των γραμμών που αντιστοιχούν στο block μίας διεργασίας) μεγέθους local[1] (όσος και ο αριθμός των στηλών που αντιστοιχούν στο block μιας διεργασίας) τα οποία έχουν μεταξύ τους stride local[1] +2 καθώς μετριούνται και τα δύο ghost cells (ένα στην αρχή και ένα στο τέλος της γραμμής του block της διεργασίας).

Παρατήρηση: Στην πραγματικότητα μέσω της κλίσης MPI_Type_Vector δημιουργούμε το Datatype dummy και στην συνέχεια καλούμε την MPI_Type_create_resized η οποία δημιουργεί τα Datatypes global_block και local_block αντίστοιχα. Κάτι τέτοιο γίνεται για τον ακόλουθο λόγο:

> Το MPI_Type_Vector δημιουργεί έναν τύπο δεδομένων MPI με την αντιγραφή αρχικά ενός υπάρχοντα τύπου MPI_Datatype ορισμένες φορές σε ένα “block” (το block εδώ το φανταζόμαστε ως συνεχόμενες θέσεις μνήμης και δεν έχει σχέση με τα datatypes local_block και global_block που δημιουργούνται). Ο

νέος τύπος που δημιουργείται θα περιέχει έναν ορισμένο αριθμό από τέτοια “block” διαχωριζόμενα με την καθορισμένη σταθερή απόσταση stride.

Όταν ένα τέτοιο Datatype (που έχει ορισθεί δηλαδή από την MPI_Type_Vector) επαναλαμβάνεται ως στιγμιότυπο μίας Collective συνάρτησης MPI (π.χ. Gather , Scatter) η οποία είτε θα μοιράσει στις διάφορες διεργασίες είτε θα μαζέψει από τις διάφορες διεργασίες δεδομένα του συγκεκριμένου τύπου χρησιμοποιείται ένας vector extent ο οποίος καθορίζει την αρχική θέση του συγκεκριμένου Datatype για την κάθε επόμενη διεργασία. Αυτό το vector extent υπολογίζεται για κάθε διεργασία ως η απόσταση από την αρχή του πρώτου block μέχρι το τέλος του τελευταίου block της προηγούμενης από αυτήν διεργασίας. Αυτό σημαίνει για παράδειγμα πως το Datatype της δεύτερης διεργασίας ξεκινάει μόνο μετά το τελευταίο block της πρώτης. Στην περίπτωση μας όμως που θέλουμε κάθε διεργασία να λαμβάνει ένα Datatype τύπου local_block (block ενός πίνακα) κάτι τέτοιο θα δημιουργούσε πρόβλημα. Για τον λόγο αυτό εκτελούμε τις εντολές που αναφέραμε παραπάνω.

```
//----Rank 0 defines positions and counts of local blocks (2D-subdomains) on global matrix----//
int * scatteroffset, * scattercounts;
if (rank==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);
        }
    }
}
```

Στο συγκεκριμένο κομμάτι κώδικα υπολογίζονται οι τιμές των πινάκων scattercounts και scatteroffset. Οι πίνακες αυτοί είναι μονοδιάστατοι και έχουν μήκος όσος είναι ο αριθμός των διεργασιών, δηλαδή grid[0] x gird[1]. Χωρίζοντας δηλαδή τον πίνακα U σε block διαστάσεων local[0] x local[1] όπως έχουμε αναφέρει και διασχίζοντας τα block αυτά κατά γραμμές κάθε ένα block (και άρα η διεργασία στην οποία θα αποδοθεί) αντιστοιχεί σε μία θέση των πινάκων scattercounts και scatteroffset. Για τους πίνακες αυτούς ισχύει:

- Η θέση k του πίνακα scattercounts περιέχει τον αριθμό των στοιχείων που θα αποκτήσει η διεργασία k. Εφόσον η κάθε διεργασία αποκτά ένα στοιχείο τύπου MPI_Datatype local_block η τιμή της κάθε θέσης ισούται με 1.
- Κάθε θέση k του πίνακα scatteroffset περιέχει το offset από το στοιχείο U[0][0] του πρώτου στοιχείου του block που θα αποδοθεί στην διεργασία k. Το offset αυτό μετριέται σε στοιχεία MPI_DOUBLE του πίνακα U και υπολογίζονται διασχίζοντας τα στοιχεία κατά γραμμές.

Η χρησιμότητα των δύο αυτών πινάκων γίνεται φανερή στο ακριβώς επόμενο κομμάτι κώδικα.

Στην συνέχεια προχωράμε στην επεξήγηση του κώδικα τον οποίο υλοποιήσαμε. Αρχικά παραθέτουμε τα πρώτα κομμάτια του κώδικα τα οποία είναι κοινά και για τις τρεις μεθόδους. Έτσι έχουμε:

```
if (rank==0) start = &(U[0][0]);
MPI_Scatterv(start,scattercounts,scatteroffset,global_block,&(u_previous[1][1]),1,local_block,0,MPI_COMM_WORLD)
MPI_Scatterv(start,scattercounts,scatteroffset,global_block,&(u_current[1][1]),1,local_block,0,MPI_COMM_WORLD);
```

Στο συγκεκριμένο κομμάτι κώδικα πρέπει αρχικά να μοιράσουμε τον αρχικοποιημένο πίνακα U στις διάφορες διεργασίες. Η κάθε διεργασία όπως αναφέραμε θα πάρει ένα block του global πίνακα U διαστάσεων local[0] x local[1]. Το block αυτό της κάθε διεργασίας θέλουμε να αποθηκευτεί τόσο στον πίνακα u_previous όσο και στον πίνακα u_current έτσι ώστε να είναι και οι δύο σωστά αρχικοποιημένοι. Σύμφωνα με το παραπάνω κομμάτι κώδικα αρχικά η διεργασία με rank = 0 (master thread) ορίζει έναν δείκτη με όνομα start ο οποίος δείχνει στο πρώτο στοιχείο U[0][0] του global πίνακα U. Στην συνέχεια έχουμε τις ακόλουθες δύο κλίσεις:

- *MPI_Scatterv(start,scattercounts,scatteroffset,global_block,&(u_previous[1][1]),1,local_block,0,MPI_COMM_WORLD) :* Η MPI_Scatterv χωρίζει τον πίνακα U ,η αρχική διεύθυνση του οποίου είναι η start, δίνοντας με την σειρά σε κάθε διεργασία scattercounts[i] (στην περίπτωση μας είναι όλα 1) δεδομένα τύπου global_block (datatype τον ορισμό του οποίου εξηγήσαμε προηγουμένως). Το datatype global_block που θα δοθεί στην διεργασία i ξεκινάει από την θέση scatteroffset[i]. Τα δεδομένα της κάθε διεργασίας θα αποθηκευτούν στον πίνακα u_previous ξεκινώντας από την διεύθυνση u_previous[1][1]. Τα δεδομένα αυτά είναι 1 το πλήθος και τύπου local_block. Η διεργασία που μοιράζει τα δεδομένα έχει rank = 0 και η επικοινωνία γίνεται μέσω του communicator MPI_COMM_WORLD.

  Παρατήρηση : Παρατηρούμε πως για κάθε διεργασία η αρχική διεύθυνση από την οποία θα αρχίσει η αποθήκευση των δεδομένων είναι η &(u_previous[1][1]) και όχι η &(u_previous[0][0]). Αυτό οφείλεται στην ύπαρξη των ghost cells τα οποία θα χρησιμοποιηθούν όπως είπαμε για να αποθηκεύσουν τα δεδομένα που θα λαμβάνει μία διεργασία από τα γειτονικά της block και στα οποία δεν θέλουμε να αποθηκευτούν δεδομένα του ίδιου του block της διεργασίας.

- MPI_Scatterv(start,scattercounts,scatteroffset,global_block,&(u_current[1][1]),1,local_block,0,MPI_COMM_WORLD): Η συγκεκριμένη κλίση λειτουργεί με τον ίδιο ακριβώς τρόπο που περιγράψαμε προηγουμένως με την διαφορά πως η αρχική διεύθυνση για την αποθήκευση των δεδομένων της κάθε

διεργασίας είναι πλέον η &(u_current[1][1]) ώστε να πραγματοποιηθεί και η αρχικοποίηση του πίνακα u_current.

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


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

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


    /*Fill your code here*/

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);
```

Στο συγκεκριμένο κομμάτι του κώδικα η διεργασία με rank = 0 (master thread) απελευθερώνει το κομμάτι της μνήμης που είχε δεσμεύσει για τον πίνακα U μίας και τα δεδομένα του έχουν πλέον μοιραστεί κατάλληλα στις διάφορες διεργασίες όπως είδαμε προηγουμένως.

Έπειτα η κάθε διεργασία δημιουργεί ένα νέο MPI_Datatype το οποίο ονομάζεται column και το οποίο αναπαριστά μία στήλη του block της συγκεκριμένης διεργασίας. Το Datatype αυτό δημιουργείται μέσω της κλίσης της συνάρτησης MPI_Type_vector στην οποία ενώνουμε local[0] δεδομένα (όσος και ο αριθμός των γραμμών του εκάστοτε block) μεγέθους 1 και τύπου MPI_DOUBLE τα οποία απέχουν μεταξύ τους stride local[0] +2 (το +2 προκύπτει και πάλι λόγο των ghost cells). Και πάλι χρησιμοποιείται η κλίση MPI_create_resized για τους λόγους που εξηγήσαμε και σε παραπάνω περίπτωση.

Παρατήρηση: Το συγκεκριμένο MPI_Datatype column θα χρησιμοποιηθεί για την μεταφορά ολόκληρων στηλών ενός block σε γειτονικά του, κάτι το οποίο απαιτείται όπως θα δούμε ώστε να εξασφαλιστεί η ορθότητα εκτέλεσης του αλγορίθμου.

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

    //*************TODO******************//
int north, south, east, west;



    /*Fill your code here*/


    /*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);
```

Στο συγκεκριμένο κομμάτι κώδικα κάθε διεργασία βρίσκει το rank που έχουν οι τέσσερις γειτονικές της διεργασίες (north , south , east , west) στον communicator MPI_COMM_WORLD με την βοήθεια του καρτεσιανού communicator CART_COMM. Η κάθε διεργασία χρειάζεται τα ranks αυτά καθώς θα χρειαστεί να επικοινωνήσει με τις γειτονικές της διεργασίες προκειμένου να ανταλλάξει δεδομένα. Για τον σκοπό αυτό χρησιμοποιεί όπως βλέπουμε και στον κώδικα τις ακόλουθες 2 κλίσεις:

- *MPI_Cart_shift(CART_COMM,0,1,&north,&south):* Με την κλίση αυτή αποθηκεύονται στις μεταβλητές north και south τα ranks του νότιου και βόρειου γείτονα της κάθε διεργασίας αντίστοιχα όπως αυτοί ορίζονται από το καρτεσιανό πλέγμα που αναπαριστά τις διεργασίες.
- *MPI_Cart_shift(CART_COMM,1,1,&west,&east):* Με την κλίση αυτή αποθηκεύονται στις μεταβλητές west και east τα ranks του δυτικού και ανατολικού γείτονα της κάθε διεργασίας αντίστοιχα όπως αυτοί ορίζονται από το καρτεσιανό πλέγμα που αναπαριστά τις διεργασίες.

Σημείωση: Εάν μία διεργασία δεν έχει κάποιον ή κάποιους από τους παραπάνω γείτονες τότε στις αντίστοιχες μεταβλητές (north , south , west , east) θα δοθεί η τιμή MPI_PROC_NULL.

```
//---Define the iteration ranges per process-----//
    //*************TODO******************//

int i_min,i_max,j_min,j_max;



    /*Fill your code here*/


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

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

if(north == MPI_PROC_NULL){
    i_min++;
}

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

if(west == MPI_PROC_NULL){
    j_min++;
}

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

Στο συγκεκριμένο κομμάτι κώδικα βρίσκουμε τα άνω (i_max , j_max) και κάτω όρια (i_min και j_min) των μεταβλητών i και j οι οποίες διατρέχουν τις γραμμές και τις στήλες του block της κάθε διεργασίας προκειμένου να ανανεώσουν τις τιμές του σε κάθε χρονικό βήμα (παρατηρώντας μάλιστα κανείς την υλοποίηση της κάθε μεθόδου την οποία παραθέσαμε παραπάνω βλέπει την ύπαρξη δύο nested loops (το εξωτερικό τρέχει πάνω στο i και το εσωτερικό πάνω στο j) τα οποία διασχίζουν κατά γραμμές τον πίνακα (το block για την κάθε διεργασία) που πρέπει να ανανεωθεί σε ένα χρονικό βήμα).

Σημείωση: Η τιμή i_max και j_max ποτέ δεν αποκτάται από τους iterators i και j. Πάντα φθάνουν μέχρι την τιμή i_max -1 και j_max -1.

Ο υπολογισμός των ορίων αυτών χωρίζεται στις ακόλουθες περιπτώσεις:

- Στην περίπτωση που η διεργασία είναι εσωτερική (internal) δηλαδή το block που έχει αναλάβει έχει 4 γειτονικά block (north , south , east , west) τότε ισχύουν τα εξής:

    1. i_min = 1. Δεν λαμβάνεται υπόψιν η πρώτη γραμμή (i=0) η οποία αποτελείται από ghost cells.
    2. i_max = local[0] +1. Δεν λαμβάνεται υπόψιν η τελευταία γραμμή (i = local[0] +1) η οποία επίσης αποτελείται από ghost cells. Έτσι οι γραμμές που λαμβάνονται υπόψιν εκτείνονται από την 1 μέχρι και την local[0].
    3. j_min = 1. Δεν λαμβάνεται υπόψιν η πρώτη στήλη (j = 0) η οποία επίσης αποτελείται από ghost cells.
    4. j_max = local[1] +1. Δεν λαμβάνεται υπόψιν η τελευταία στήλη (j = local[1] +1) η οποία επίσης αποτελείται από ghost cells. Έτσι οι στήλες που λαμβάνονται υπόψιν εκτείνονται από την 1 μέχρι και την local[1].

- Στην περίπτωση που η διεργασία δεν έχει βόρειο γείτονα πρέπει επίσης να αποκλειστεί η δεύτερη γραμμή καθώς τα στοιχεία της γραμμής αυτής δεν διαθέτουν βόρειο γείτονα. Έτσι το i_min αυξάνεται κατά ένα σε σχέση με την πρώτη περίπτωση. Παράλληλα η μη ύπαρξη βόρειου γείτονα δεν συνοδεύεται από κάποιο πιθανό padding το οποίο θα οδηγούσε στον αποκλεισμό επιπλέον γραμμών ή στηλών. Το padding αφορά μόνο διεργασίες που δεν έχουν ανατολικό (δεξιό) ή νότιο (κάτω) γείτονα.

- Στην περίπτωση που η διεργασία δεν έχει νότιο γείτονα πρέπει επίσης να αποκλειστεί η προτελευταία γραμμή καθώς τα στοιχεία της γραμμής αυτής δεν διαθέτουν νότιο γείτονα. Όμως σε περίπτωση όπου υπάρχει padding σε αυτήν την κατεύθυνση χρειάζεται να αποκλειστούν και οι επιπλέον γραμμές που αντιστοιχούν σε αυτό. Έτσι τελικά το i_max μειώνεται από local[0] +1 σε i_max – (global_padded[0]-global[0]) -1 όπου στην περίπτωση που δεν υπάρχει padding ισχύει global_padded[0] -global[0] =0.

- Στην περίπτωση που η διεργασία δεν έχει δυτικό γείτονα πρέπει επίσης να αποκλειστεί η δεύτερη στήλη καθώς τα στοιχεία της στήλης αυτής δεν διαθέτουν δυτικό γείτονα. Έτσι το j_min αυξάνεται κατά ένα σε σχέση με την πρώτη περίπτωση. Παράλληλα η μη ύπαρξη δυτικού γείτονα δεν συνοδεύεται από κάποιο πιθανό padding το οποίο θα οδηγούσε στον αποκλεισμό επιπλέον γραμμών ή στηλών. Το padding αφορά μόνο διεργασίες που δεν έχουν ανατολικό (δεξιό) ή νότιο (κάτω) γείτονα.

- Στην περίπτωση που η διεργασία δεν έχει ανατολικό γείτονα πρέπει επίσης να αποκλειστεί η προτελευταία στήλη καθώς τα στοιχεία της στήλης αυτής δεν διαθέτουν ανατολικό γείτονα. Όμως σε περίπτωση όπου υπάρχει padding σε αυτήν την κατεύθυνση χρειάζεται να αποκλειστούν και οι επιπλέον στήλες που αντιστοιχούν σε αυτό. Έτσι τελικά το j_max μειώνεται από local[1] +1 σε j_max – (global_padded[1]-global[1]) -1 όπου στην περίπτωση που δεν υπάρχει padding ισχύει global_padded[1] -global[1] =0.

Στην συνέχεια ο κώδικας διαφοροποιείται ανάλογα με την κάθε μέθοδο οπότε και αναλύεται ξεχωριστά:

2.1 Μέθοδος Jacobi

Για την μέθοδο Jacobi πραγματοποιήθηκαν δύο υλοποιήσεις. Οπότε και αναλύουμε την κάθε υλοποίηση ξεχωριστά:

1η Υλοποίηση Μέθοδος Jacobi

Παραθέτουμε τον κώδικα της 1ης Υλοποίησης για την μέθοδο Jacobi και στην συνέχεια τον επεξηγούμε:

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

gettimeofday(&tts, NULL);
MPI_Status status;
#ifdef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
```

```
#undef T
#define T 256
for (t=0;t<256;t++) {
#endif
        //*************TODO*******************//
        /*Fill your code here*/

       /*Compute and Communicate*/
 gettimeofday(&tts, NULL);
 MPI_Status status;
```

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

        //*************TODO*******************//
        /*Fill your code here*/

        /*Compute and Communicate*/
```

```
swap=u_previous;

u_previous=u_current;

u_current=swap;

if (north != MPI_PROC_NULL){


        MPI_Sendrecv(&(u_previous[1][1]),local[1],MPI_DOUBLE,north,0,
        &(u_previous[0
        ][1]),local[1],MPI_DOUBLE,north,0,MPI_COMM_WORLD,&status);

}

if (south != MPI_PROC_NULL){


        MPI_Sendrecv(&(u_previous[local[0]][1]),local[1],MPI_DOUBLE,so
        uth,0,&(u_previous[local[0]+1][1]),local[1],MPI_DOUBLE,south,0,
        MPI_COMM_WORLD,&status);

}
```

```
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);

}


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++){
            u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] + u_previous[i][j-1] + u_previous[i][j+1])/4.0;
        }
    }
    gettimeofday(&tcf,NULL);
    tcomp = tcomp + (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001;
    #ifdef TEST_CONV
    if (t%C==0) {
        //*************TODO**************//
        /*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 = 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);
   #ifdef TEST_CONV

MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
   #endif
   //----Rank 0 gathers local matrices back to the global matrix----//
   if (rank==0) {
         U=allocate2d(global_padded[0],global_padded[1]);
   }
/*Fill your code here*/
    if (rank==0) start = &(U[0][0]);
```

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

Στην συνέχεια επεξηγούμε κάθε φάση της υλοποίησης ξεχωριστά:

Αρχική χρονική στιγμή πριν την έναρξη της επαναληπτικής υλοποίησης.

Αρχικά παρατηρούμε πως καλείται η gettimeofday(&tts , NULL) προκειμένου να αποθηκευτεί στην μεταβλητή ttfs η χρονική στιγμή που αρχίζει η επαναληπτική διαδικασία της συγκεκριμένης παράλληλης υλοποίησης για την κάθε διεργασία.

Επικοινωνία μεταξύ των διεργασιών.

Στην μέθοδο Jacobi όπως αναλύσαμε και στην σχεδίαση της παραλληλοποίησης κάθε block χρειάζεται τις τιμές των γειτονικών του block από το προηγούμενο χρονικό βήμα προκειμένου να προχωρήσει στην ανανέωση των τιμών του στο τρέχων χρονικό βήμα. Οι τιμές που χρειάζεται λοιπόν βρίσκονται στον πίνακα u_previous της κάθε διεργασίας γείτονα με την οποία συνορεύει. Πιο συγκεκριμένα σε κάθε χρονικό βήμα προκειμένου να ξεκινήσει η ανανέωση των τιμών των block από τις διεργασίες απαιτείται:

- Εφόσον μία διεργασία έχει βόρειο γείτονα (north) χρειάζεται να του στείλει την δεύτερη γραμμή του πίνακα u_previous η οποία ξεκινάει από την διεύθυνση &u_previous[1][1]. Αντίστοιχα χρειάζεται να αποθηκεύσει στην πρώτη γραμμή του πίνακα u_previous η οποία αποτελείται από ghost cells όπως έχουμε αναφέρει τα δεδομένα που θα σταλθούν από τον βόρειο γείτονα. Η πρώτη γραμμή ξεκινάει από την διεύθυνση &u_previous[0][1].
- Εφόσον μία διεργασία έχει νότιο γείτονα (south) χρειάζεται να του στείλει την προτελευταία γραμμή του πίνακα u_previous η οποία ξεκινάει από την διεύθυνση &u_previous[local[0]][1]. Αντίστοιχα χρειάζεται να αποθηκεύσει στην τελευταία γραμμή του πίνακα u_previous η οποία αποτελείται από ghost cells όπως έχουμε αναφέρει τα δεδομένα που θα σταλθούν από τον νότιο γείτονα. Η τελευταία γραμμή ξεκινάει από την διεύθυνση &u_previous[local[0]+1][1].
- Εφόσον μία διεργασία έχει δυτικό γείτονα (west) χρειάζεται να του στείλει την δεύτερη στήλη του πίνακα u_previous η οποία ξεκινάει από την διεύθυνση &u_previous[1][1]. Αντίστοιχα χρειάζεται να αποθηκεύσει στην πρώτη στήλη του πίνακα u_previous η οποία αποτελείται από ghost cells όπως έχουμε αναφέρει τα δεδομένα που θα σταλθούν από τον δυτικό γείτονα. Η πρώτη γραμμή ξεκινάει από την διεύθυνση &u_previous[1][0].
- Εφόσον μία διεργασία έχει ανατολικό γείτονα (east) χρειάζεται να του στείλει την προτελευταία στήλη του πίνακα u_previous η οποία ξεκινάει από την διεύθυνση &u_previous[1][local[1]]. Αντίστοιχα χρειάζεται να αποθηκεύσει στην τελευταία στήλη του πίνακα u_previous η οποία αποτελείται από ghost cells όπως έχουμε αναφέρει τα δεδομένα που θα σταλθούν από τον ανατολικό γείτονα. Η τελευταία στήλη ξεκινάει από την διεύθυνση &u_previous[1][local[1]+1].

Στην συγκεκριμένη υλοποίηση όπως βλέπουμε λοιπόν σε κάθε νέο χρονικό βήμα αντιγράφεται αρχικά ο πίνακας u_current στον u_previous. Έπειτα ελέγχεται αν η

διεργασία έχει βόρειο , νότιο , ανατολικό και δυτικό γείτονα. Σε περιπτώσεις που υπάρχει ο αντίστοιχος γείτονας εκτελείται η διαδικασία που περιγράψαμε παραπάνω. Η αποστολή και λήψη των κατάλληλων δεδομένων που αναλύσαμε προηγουμένως γίνεται με την κλήση της MPI_Sendrecv όπου στην περίπτωση που χρειάζεται να σταλθούν και να ληφθούν γραμμές του block (βόρειος και νότιος γείτονας) πραγματοποιείται αποστολή και λήψη local[1] δεδομένων τύπου MPI_DOUBLE ενώ στην περίπτωση που χρειάζεται να σταλθούν και να ληφθούν στήλες του block (δυτικός και ανατολικός γείτονας) πραγματοποιείται αποστολή και λήψη ενός δεδομένου με MPI_Datatype column. Καταλαβαίνουμε λοιπόν σε αυτό το σημείο την σημασία του Datatype column το οποίο ορίσαμε προηγουμένως στην επικοινωνία μεταξύ ορισμένων διεργασιών.

Η MPI_Sendrecv είναι blocking συνάρτηση με αποτέλεσμα το πρόγραμμα να μπλοκάρει και να μην συνεχίζεται η εκτέλεση του ενώσω γίνεται η αποστολή και λήψη του μηνύματος. Ο λόγος που χρησιμοποιείται blocking συνάρτηση είναι πως για την ορθότητα εκτέλεσης του αλγορίθμου απαιτείται σε κάθε χρονικό βήμα να πραγματοποιείται πρώτα η ανταλλαγή των δεδομένων που αναφέρθηκαν προηγουμένως και έπειτα να προχωρήσει η κάθε διεργασία στην ανανέωση του block της.

Διαδικασία Υπολογισμών (Ανανέωσης Τιμών).

Μετά το στάδιο της επικοινωνίας ακολουθεί το στάδιο της ανανέωσης για ένα χρονικό βήμα εκτέλεσης. Πριν την έναρξη των υπολογισμών καλείται η gettimeofday(&tcs , NULL) προκειμένου να αποθηκευτεί στην μεταβλητή tcs η χρονική στιγμή που αρχίζει η διαδικασία των υπολογισμών μίας διεργασίας για ένα συγκεκριμένο χρονικό βήμα. Έπειτα πραγματοποιείται η ανανέωση του block της κάθε διεργασίας ακριβώς όπως πραγματοποιείται και στην σειριακή περίπτωση της μεθόδου με την διαφορά ότι πλέον χρησιμοποιούμε τα i_min , i_max , j_min , j_max ως όριο των δυο nested loops προκειμένου να έχουμε τα κατάλληλα όρια του block για τους υπολογισμούς μας. Μετά το τέλος του εμφολευμένου βρόγχου καλείται η gettimeofday(&tcf , NULL) προκειμένου να αποθηκευτεί στην μεταβλητή tcf η χρονική στιγμή που τελειώνει η διαδικασία των υπολογισμών μίας διεργασίας για ένα συγκεκριμένο χρονικό βήμα. Τέλος σε κάθε χρονικό βήμα προστίθεται στην μεταβλητή tcomp η παράσταση (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001 η οποία υπολογίζει την χρονική διάρκεια που διήρκησε η ανανέωση του block της διεργασίας για ένα χρονικό βήμα. Στο τέλος των χρονικών βημάτων η μεταβλητή tcomp θα περιέχει την χρονική διάρκεια που καταναλώθηκε στην ανανέωση του block μίας διεργασίας για το σύνολο των χρονικών βημάτων εκτέλεσης (διάρκεια υπολογισμών μίας διεργασίας).

Έλεγχος Σύγκλισης (Τοπικός αλλά και για όλο το χωρίο).

Μετά το τέλος του σταδίου ανανέωσης του block πραγματοποιείται από κάθε διεργασία σε κάθε χρονικό βήμα ο έλεγχος σύγκλισης σε περίπτωση που τον έχουμε ενεργοποιημένο. Αρχικά καλείται η gettimeofday(&tcvs , NULL) προκειμένου να αποθηκευτεί στην μεταβλητή tcvs η χρονική στιγμή που αρχίζει η διαδικασία του ελέγχου σύγκλισης μίας διεργασίας για ένα συγκεκριμένο χρονικό βήμα. Έπειτα καλείται η συνάρτηση converge (η υλοποίηση της δίνεται έτοιμη στο αρχείο utils.c) με την οποία η κάθε διεργασία ελέγχει αν έχει πραγματοποιηθεί σύγκλιση στο block της. Η τιμή επιστροφής της συνάρτησης αυτής αποθηκεύεται στην τοπική μεταβλητή converged της κάθε διεργασίας. Σε περίπτωση που είναι 1 αυτό σημαίνει πως έχει επέλθει σύγκλιση στο block της διεργασίας διαφορετικά εάν είναι 0 τότε δεν έχει επέλθει ακόμα σύγκλιση.

Στην συνέχεια όμως χρειάζεται να ελεγχθεί εάν έχει επέλθει σύγκλιση σε όλο το χωρίο και όχι μόνο τοπικά κάθε διεργασία να εξετάζει την σύγκλιση του δικού της block. Για τον λόγο αυτό κάθε διεργασία χρειάζεται να έχει γνώση για το αν παρατηρήθηκε σύγκλιση σε όλες τις υπόλοιπες διεργασίες. Σε περίπτωση που και σε όλες τις υπόλοιπες διεργασίες έχει παρατηρηθεί σύγκλιση τότε πρέπει να μην προχωρήσει σε επόμενο χρονικό βήμα υπολογισμού. Για το λόγο αυτό καλείται η συνάρτηση MPI_Allreduce(&converged,&global_converged,1,MPI_INT,MPI_MIN, MPI_COMM_WORLD) σύμφωνα με την οποία όλες οι διεργασίες στέλνουν σε όλες τις υπόλοιπες διεργασίες την τιμή της τοπικής τους μεταβλητής converged. Έπειτα η κάθε διεργασία κρατάει το ελάχιστο από όλες αυτές τις τιμές και το αποθηκεύει στην μεταβλητή global_converged. Εάν έστω και σε μία διεργασία δεν είχε παρατηρηθεί σύγκλιση τότε global_converged = 0 και η διαδικασία θα επαναληφθεί και για το επόμενο χρονικό βήμα. Αντίθετα εάν global_converged =1 σημαίνει ότι για όλες τις διεργασίες ισχύει converged = 1. Επομένως έχει επέλθει σύγκλιση σε κάθε block του πίνακα U και η διαδικασία δεν πρέπει να προχωρήσει σε επόμενο χρονικό βήμα.

Τελική χρονική στιγμή μετά την λήξη της επαναληπτικής υλοποίησης.

Μετά το τέλος των χρονικών βημάτων λοιπόν που απαιτούνται και αφού κάθε διεργασία έχει υπολογίσει τις τελικές τιμές των στοιχείων του block που της έχει ανατεθεί καλείται από την κάθε διεργασία η gettimeofday(&ttf , NULL) προκειμένου να αποθηκευτεί στην μεταβλητή ttf η χρονική στιγμή που τελειώνει όλη αυτή η επαναληπτική διαδικασία για κάθε διεργασία. Έτσι υπολογίζεται τέλος η χρονική διάρκεια που διήρκησε όλη η επαναληπτική διαδικασία για κάθε διεργασία η οποία ισούται με ttotal = (ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001.

Υπολογισμός του συνολικού χρόνου υπολογισμων , συνολικού χρόνου ελέγχου σύγκλισης και συνολικού χρόνου όλης της διαδικασίας.

Στην συνέχεια απαιτείται να υπολογισθεί ο συνολικός χρόνος υπολογισμών , ελέγχου σύγκλισης καθώς και ο συνολικός χρόνος όλης της διαδικασίας που χρειάστηκε προκειμένου να υπολογισθούν οι τελικές τιμές του πίνακα U. Ο συνολικός χρόνος εκτέλεσης όπως υποδεικνύεται αφορά το υπολογιστικό μέρος του αλγορίθμου και όχι αρχικοποιήσεις , αρχικές αποστολές και τελική συλλογή δεδομένων. Για τον λόγο αυτό και ο υπολογισμός του (μαζί με τον υπολογισμού του συνολικού χρόνου υπολογισμών και συνολικού χρόνου ελέγχου σύγκλισης) γίνεται πριν συλλεχθούν τα αποτελέσματα όλων των διεργασιών.

Και για τους τρεις χρόνους χρησιμοποιείται η συνάρτηση MPI_Reduce όπου η διεργασία με rank =0 (master thread) συλλέγει τους αντίστοιχους χρόνους της κάθε διεργασίας και από αυτούς κρατάει τον μέγιστο (MPI_MAX). Έτσι ο συνολικός χρόνος υπολογισμών (tcomp) ορίζεται ως ο μέγιστος μεταξύ των χρόνων υπολογισμών της κάθε διεργασίας. Αντίστοιχη είναι η λογική και για τον συνολικό χρόνο σύγκλισης (tconv) όσο και για τον συνολικό χρόνο ολόκληρης της επαναληπτικής διαδικασίας (ttotal).

Συλλογή των τελικών αποτελεσμάτων και πάλι στον πίνακα U.

Τέλος συγκεντρώνουμε τα αποτελέσματα όλων των διεργασιών και πάλι στον πίνακα U για τον οποίο η διεργασία με rank =0 δεσμεύει εκ νέου μνήμη. Η συλλογή των επιμέρους block γίνεται με την συνάρτηση Gatherv η οποία έχει παρόμοια λογική με την συνάρτηση Scatterv που χρησιμοποιήθηκε για τον διαμοιρασμό των block στις διεργασίες. Και πάλι χρησιμοποιούνται ο πίνακας scattercounts και scatteroffset τους οποίους είχαμε υπολογίσει στην αρχή ενώ τα επιμέρους block τοποθετούνται στον πίνακα U μέσω της διεργασίας με rank = 0.

Παρατήρηση: *Οι φάσεις Αρχική χρονική στιγμή πριν την έναρξη της επαναληπτικής υλοποίησης , Έλεγχος Σύγκλισης , Υπολογισμός του συνολικού χρόνου υπολογισμών , συνολικού χρόνου ελέγχου σύγκλισης και συνολικού χρόνου όλης της διαδικασίας και Συλλογή των τελικών αποτελεσμάτων και πάλι στον πίνακα U υλοποιούνται με τον ίδιο ακριβώς τρόπο σε όλες τις υλοποιήσεις που ακολουθούν και γι' αυτό δεν ξανααναφέρονται. Οι φάσεις της επικοινωνίας και των υπολογισμών αντίθετα διαφοροποιούνται σε μεγάλο βαθμό και γι' αυτό επεξηγούνται αναλυτικά σε κάθε μία από τις επόμενες υλοποίηση.*

2η Υλοποίηση Μέθοδος Jacobi

Παραθέτουμε τον κώδικα της 2ης Υλοποίησης για την μέθοδο Jacobi και στην συνέχεια τον επεξηγούμε:

```
/----Computational core----//
 gettimeofday(&tts, NULL);
  MPI_Status status;
  #ifdef TEST_CONV
  for (t=0;t<T && !global_converged;t++) {
  #endif
  #ifndef TEST_CONV
  #undef T
  #define T 256
  for (t=0;t<256;t++) {
  #endif
//*************TODO*******************//
          /*Fill your code here*/
```

```
          /*Compute and Communicate*/
     swap=u_previous;
     u_previous=u_current;
     u_current=swap;
     req_len = 0;
     if (north != MPI_PROC_NULL){

               MPI_Isend(&(u_previous[1][1]),local[1],MPI_DOUBLE,north,0,MPI
               _COMM_WORLD,&req[req_len]);
                req_len++;

               MPI_Irecv(&(u_previous[0][1]),local[1],MPI_DOUBLE,north,0,MPI_
               COMM_WORLD,&req[req_len]);
               req_len++;
      }
```

```
if (south != MPI_PROC_NULL){

        MPI_Isend(&(u_previous[local[0]][1]),local[1],MPI_DOUBLE,south,
        0,MPI_COMM_WORLD,&req[req_len]);

        req_len++;

        MPI_Irecv(&(u_previous[local[0]+1][1]),local[1],MPI_DOUBLE,sout
        h,0,MPI_COMM_WORLD,&req[req_len]);

        req_len++;

}

if (east != MPI_PROC_NULL){

        MPI_Isend(&(u_previous[1][local[1]]),1,column,east,0,MPI_COMM
        _WORLD,&req[req_len]);
```

```
        MPI_Irecv(&(u_previous[1][local[1]+1]),1,column,east,0,MPI_COM
        M_WORLD,&req[req_len]);

        req_len++;

}

if (west != MPI_PROC_NULL){

        MPI_Isend(&(u_previous[1][1]),1,column,west,0,MPI_COMM_WO
        RLD,&req[req_len]);

        req_len++;

        MPI_Irecv(&(u_previous[1][0]),1,column,west,0,MPI_COMM_WO
        RLD,&req[req_len]);

        req_len++;

}
```

```
gettimeofday(&tcs,NULL);
for (i=i_min+1;i<i_max-1;i++){
    for (j=j_min+1;j<j_max-1;j++){
        u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] + u_previous[i][j-1] + u_previous[i][j+1])/4.0;
    }
}
gettimeofday(&tcf,NULL);
tcomp = tcomp + (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001;

MPI_Waitall(req_len , req , status_req);
```

```
/*Add appropriate timers for computation*/
gettimeofday(&tcs,NULL);
 for (j=j_min;j<j_max;j++){
    u_current[i_min][j] = (u_previous[i_min-1][j] + u_previous[i_min+1][j] + u_previous[i_min][j-1] + u_previous[i_min][j+1])/4.0;
    u_current[i_max-1][j] = (u_previous[i_max-1-1][j] + u_previous[i_max-1+1][j] + u_previous[i_max-1][j-1] + u_previous[i_max-1][j+1])/4.0;
 }
 for (i=i_min;i<i_max;i++){
    u_current[i][j_min] = (u_previous[i-1][j_min] + u_previous[i+1][j_min] + u_previous[i][j_min-1] + u_previous[i][j_min+1])/4.0;
    u_current[i][j_max-1] = (u_previous[i-1][j_max-1] + u_previous[i+1][j_max-1] + u_previous[i][j_max-1-1] + u_previous[i][j_max-1+1])/4.0;
 }
```

```
    gettimeofday(&tcf,NULL);
    tcomp = tcomp + (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001;
     #ifdef TEST_CONV
    if (t%C==0) {
            //*************TODO**************//
            /*Test convergence*/
            gettimeofday(&tcvs,NULL);
            converged = converge(u_previous , u_current , local[0] , local[1]);

              MPI_Allreduce(&converged,&global_converged,1,MPI_INT,MPI_M
              IN,MPI_COMM_WORLD);
              gettimeofday(&tcvf,NULL);
```

```
       tconv = tconv + (tcvf.tv_sec - tcvs.tv_sec) + (tcvf.tv_usec -
       tcvs.tv_usec)*0.000001;
      }
    #endi
     //***********************************//
  }
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_WOR
LD);

MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WO
RLD);
  #ifdef TEST_CONV
```

```
MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WOR
LD);
   #endif
   //----Rank 0 gathers local matrices back to the global matrix----//
   if (rank==0) {
         U=allocate2d(global_padded[0],global_padded[1]);
   }
      //*************TODO*******************//
      /*Fill your code here*/
    if (rank==0) start = &(U[0][0]);
MPI_Gatherv(&(u_current[1][1]),1,local_block,start,scattercounts,scatteroffset,gl
obal_block,0,MPI_COMM_WORLD);
```

Η δεύτερη υλοποίηση προσπαθεί να εκμεταλλευτεί το γεγονός πως τα εσωτερικά στοιχεία κάθε block μίας διεργασίας (αυτά δηλαδή που δεν ανήκουν σε ακριανές γραμμές ή στήλες) μπορούν να υπολογισθούν χωρίς να χρειάζεται να έχουν ληφθεί τα απαραίτητα δεδομένα από τις γειτονικές διεργασίες. Έτσι ο υπολογισμός των στοιχείων αυτών μπορεί να γίνει παράλληλα με την διαδικασία της επικοινωνίας μειώνοντας τον συνολικό χρόνο που χρειάζεται η διεργασία σε κάθε βήμα προκειμένου να ανανεώσει τις τιμές των στοιχείων της. Αντίθετα για τα ακριανά στοιχεία χρειάζεται και πάλι να εξασφαλισθεί πως πρώτα θα έχουν ληφθεί τα απαραίτητα δεδομένα από τις γειτονικές διεργασίες και έπειτα θα πραγματοποιηθεί η ανανέωση τους. Για τον λόγο αυτό στην φάση της επικοινωνίας αντί για την συνάρτηση MPI_Sendrecv η οποία χρησιμοποιήθηκε στην πρώτη υλοποίηση χρησιμοποιούνται οι συναρτήσεις MPI_Isend και MPI_Irecv οι οποίες είναι non-blocking με αποτέλεσμα η εκτέλεση του προγράμματος να συνεχίζεται αμέσως μετά την κλίση τους και παράλληλα με την αποστολή και λήψη των κατάλληλων δεδομένων. Κατά τα άλλα η λογική της επικοινωνίας παραμένει η ίδια όπως και στην πρώτη υλοποίηση. Έπειτα και καθώς η επικοινωνία βρίσκεται σε εξέλιξη υπολογίζονται οι τιμές των εσωτερικών στοιχείων του block. Για το λόγο αυτό τα όρια του nested for loop έχουν αλλάξει και πλέον το i εκτείνεται από i_min +1 έως i_max -1 και αντίστοιχα το j εκτείνεται από j_min +1 έως j_max -1. Μετά τον υπολογισμό των

εσωτερικών στοιχείων καλείται η συνάρτηση MPI_Waitall η οποία περιμένει να ολοκληρωθούν όλες οι αποστολές και λήψεις που έχουν προηγηθεί στην φάση της επικοινωνίας προκειμένου να συνεχιστεί η εκτέλεση του κώδικα. Αφού λοιπόν έχει εξασφαλισθεί ότι κάθε διεργασία έχει λάβει τα απαραίτητα δεδομένα από τους γείτονες της η εκτέλεση συνεχίζεται με την ανανέωση των τιμών για τα ακριανά στοιχεία. Στην συνέχεια η διαδικασία αυτή επαναλαμβάνεται για το επόμενο χρονικό βήμα (εφόσον βέβαια δεν έχει επέλθει σύγκλιση ή έχει ολοκληρωθεί ο σταθερός αριθμός χρονικών βημάτων που έχουν ορισθεί).

2.2 Μέθοδος Gauss-Seidel SOR

Στην συνέχεια παραθέτουμε τον κώδικα υλοποίησης της Gauss-Seidel SOR και αναλύουμε το κομμάτι των υπολογισμών αλλά και της επικοινωνίας μεταξύ των διεργασιών:

```
//----Computational core----//
  gettimeofday(&tts, NULL);


// number of requests before and after computation. Init to zero
int req_num_bef = 0;
int req_num_after = 0;


// arrays with mpi requests and status before and after computation
MPI_Request req_bef[6];
MPI_Status status_bef[6];
MPI_Request req_after[6];
MPI_Status status_after[6];


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

```
for (t=0;t<256;t++) {
#endif
        //*************TODO*******************//
        /*Fill your code here*/
        /*Compute and Communicate*/
        /*Add appropriate timers for computation*/
        req_num_bef = 0;
        req_num_after = 0;
        swap = u_previous;
        u_previous = u_current;
        u_current = swap;
        /*Send previous data to north and receive current data from north*/
        if (north != MPI_PROC_NULL){
            MPI_Isend(&u_previous[1][1], local[1] , MPI_DOUBLE , north , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
            MPI_Irecv(&u_current[0][1], local[1] , MPI_DOUBLE , north , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
        }
        /*Send previous data to west and receive current data from west*/
        if (west != MPI_PROC_NULL){
            MPI_Isend(&u_previous[1][1], 1 , column , west , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
            MPI_Irecv(&u_current[1][0], 1 , column , west , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
        }
```

```
        /*Receive previous data from south*/
        if (south != MPI_PROC_NULL){
            MPI_Irecv(&u_previous[local[0]+1][1] , local[1] , MPI_DOUBLE , south , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
        }
        /*Receive previous data from east*/
        if (east !=MPI_PROC_NULL){
            MPI_Irecv(&u_previous[1][local[1]+1] , 1 , column , east , 0 , MPI_COMM_WORLD , &req_bef[req_num_bef]);
            req_num_bef++;
        }
        /*Wait for all the requests to be completed*/
        MPI_Waitall(req_num_bef , req_bef , status_bef);
        gettimeofday(&tcs,NULL);
        for (i = i_min ; i<i_max ; i++){
            for (j = j_min; j<j_max; j++){
                u_current[i][j] = u_previous[i][j] + (omega/4.0)*(u_current[i-1][j] + u_current[i][j-1] + u_previous[i][j+1] + u_previous[i+1][j] - 4*u_previous[i][j]);
            }
        }
        gettimeofday(&tcf,NULL);

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

        /*Send current data to south*/
        if (south != MPI_PROC_NULL){
            MPI_Isend(&u_current[local[0]][1] , local[1] , MPI_DOUBLE ,south , 0 , MPI_COMM_WORLD , &req_after[req_num_after]);
            req_num_after++;
```

```
        }
        /*Send current data to east*/
        if (east != MPI_PROC_NULL){
            MPI_Isend(&u_current[1][local[1]], 1 , column , east , 0 , MPI_COMM_WORLD , &req_after[req_num_after]);
            req_num_after++;
        }
        /* Wait for all the requests to be completed */
        MPI_Waitall(req_num_after , req_after , status_after);
    #ifdef TEST_CONV
    if (t%C==0) {
            //*************TODO**************//
            /*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 = 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);
  #ifdef TEST_CONV
```

```
MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD)
;
  #endif
  //----Rank 0 gathers local matrices back to the global matrix----//
  if (rank==0) {
      U=allocate2d(global_padded[0],global_padded[1]);
      U_start = &(U[0][0]);
  }
    //*************TODO*******************//
    /*Fill your code here*/
    MPI_Gatherv(&u_current[1][1] , 1 , local_block , U_start , scattercounts , scatteroffset , global_block , 0 , MPI_COMM_WORLD);
```

Όπως μπορούμε να δούμε και από τον παραπάνω κώδικα αλλά και από τις αρχικές μας παρατηρήσεις για την συγκεκριμένη μέθοδο η επικοινωνία μεταξύ των διεργασιών στην περίπτωση της υλοποίησης της μεθόδου Gauss-Seidel SOR διαφέρει σημαντικά σε σχέση με την υλοποίηση της μεθόδου Jacobi. Στην συγκεκριμένη περίπτωση έχουμε τις εξής απαιτήσεις:

<u>Πριν από τους υπολογισμούς:</u>

- Κάθε διεργασία πρέπει να λάβει από τον βόρειο (north) και δυτικό (west) γείτονα της (ένα αυτοί υπάρχουν) τα απαραίτητα δεδομένα του τρέχοντος χρονικού βήματος. Τα δεδομένα αυτά πρέπει ήδη να έχουν ανανεωθεί από την βόρεια και δυτική διεργασία για το συγκεκριμένο χρονικό βήμα και να έχουν αποθηκευτεί στον πίνακα u_current.
- Κάθε διεργασία πρέπει να στείλει στον βόρειο (north) και δυτικό (west) γείτονα της (εάν αυτοί υπάρχουν) τα απαραίτητα δεδομένα του προηγούμενου χρονικού βήματος. Τα δεδομένα αυτά έχουν ήδη υπολογισθεί από την κάθε διεργασία στο προηγούμενο χρονικό βήμα και έχουν αποθηκευτεί στον πίνακα u_previous.
- Κάθε διεργασία πρέπει να λάβει από τον νότιο (south) και ανατολικό (east) γείτονα της (εάν αυτοί υπάρχουν) τα απαραίτητα δεδομένα της προηγούμενης χρονικής στιγμής. Τα δεδομένα αυτά έχουν ήδη υπολογισθεί από τις δύο αυτές δυο διεργασίες στο προηγούμενο χρονικό βήμα και έχουν αποθηκευτεί στον πίνακα u_previous.

<u>Μετά από τους υπολογισμούς:</u>

- Κάθε διεργασία πρέπει να στείλει στον νότιο (south) και ανατολικό (east) γείτονα της (εάν αυτοί υπάρχουν) τα απαραίτητα δεδομένα του τρέχοντος χρονικού βήματος. Τα δεδομένα αυτά έχουν ήδη υπολογισθεί από την κάθε διεργασία στο τρέχων χρονικό βήμα και έχουν αποθηκευτεί στον πίνακα u_current.

Για μεγαλύτερη κατανόηση αναλύουμε το τι πραγματικά συμβαίνει σε ένα νέο χρονικό βήμα:

Μπαίνοντας λοιπόν σε ένα νέο χρονικό βήμα κάθε διεργασία έχει υπολογισμένο μόνο τον πίνακα u_previous με τις τιμές των στοιχείων του block της για το προηγούμενο χρονικό βήμα. Όλες οι διεργασίες λοιπόν στέλνουν στον βόρειο και δυτικό γείτονα τους τα απαραίτητα δεδομένα που βρίσκονται στον πίνακα u_previous. Επιπλέον περιμένουν από τον βόρειο και δυτικό γείτονα τους τα δεδομένα του τρέχοντος χρονικού βήματος. Όλες λοιπόν οι διεργασίες μπλοκάρουν περιμένοντας τα δεδομένα αυτά εκτός από την διεργασία που δεν έχει ούτε βόρειο ούτε δυτικό γείτονα. Η διεργασία αυτή θα προχωρήσει στην ανανέωση των τιμών του block της και στην συνέχεια θα στείλει τις ανανεωμένες τιμές στην ανατολική και νότια διεργασία της. Η ανατολική και νότια διεργασία λοιπόν έχουν πλέον τα δεδομένα που χρειάζονται προκειμένου να προχωρήσουν στην ανανέωση των στοιχείων τους και με την σειρά τους να στείλουν στην δικιά τους νότια και ανατολική διεργασία τα στοιχεία που χρειάζονται. Η διαδικασία αυτοί επαναλαμβάνεται μέχρι όλες οι διεργασίες να ανανεώσουν τις τιμές των block τους. Έπειτα η διαδικασία θα ξαναεπαναληφθεί για το επόμενο χρονικό βήμα κ.ο.κ

Η ανανέωση του block της κάθε διεργασίας πραγματοποιείται όπως ακριβώς και στην σειριακή περίπτωση της μεθόδου με την διαφορά ότι πλέον χρησιμοποιούμε τα i_min , i_max , j_min , j_max ως όρια των δυο nested loops προκειμένου να έχουμε τα κατάλληλα όρια του block για τους υπολογισμούς μας. Επιπλέον για τα δεδομένα του πίνακα που αναφέρονται στην προηγούμενη χρονική στιγμή χρησιμοποιείται ο πίνακας u_previous ενώ για τα δεδομένα που αναφέρονται στην τωρινή χρονική στιγμή χρησιμοποιείται ο πίνακας u_current.

Παρατήρηση: Τα δεδομένα που ανταλλάσσονται μεταξύ γειτονικών διεργασιών αποτελούν πάλι ακριανές στήλες και γραμμές των block της κάθε διεργασίας όπως και στην περίπτωση της μεθόδου Jacobi. Επιπλέον και στην συγκεκριμένη υλοποίηση γίνεται η χρήση των ghost cells για την αποθήκευση των δεδομένων που λαμβάνονται από τις γειτονικές διεργασίες.

2.3 Μέθοδος Red-Black SOR

Στην συνέχεια παραθέτουμε τον κώδικα υλοποίησης της Red-Black SOR και αναλύουμε το κομμάτι των υπολογισμών αλλά και της επικοινωνίας μεταξύ των διεργασιών:

```
gettimeofday(&tts, NULL);

MPI_Status status;
```

```
#ifdef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#undef T
#define T 256
for (t=0;t<256;t++) {
#endif
        //*************TODO*******************//
        /*Fill your code here*/
        /*Compute and Communicate*/
    swap=u_previous;
    u_previous=u_current;
    u_current=swap;
    if (north != MPI_PROC_NULL){

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

MPI_Sendrecv(&(u_previous[local[0]][1]),local[1],MPI_DOUBLE,south,0,&(u_previou
s[local[0]+1][1]),local[1],MPI_DOUBLE,south,0,MPI_COMM_WORLD,&status);
    }
    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);
    }
    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 = tcomp + (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001;

    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);
    }
    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);
    }
    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);
    }
```

```
    if (west != MPI_PROC_NULL){

MPI_Sendrecv(&(u_current[1][1]),1,column,west,0,&(u_current[1][0]),1,column,wes
t,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==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 = tcomp + (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec-tcs.tv_usec)*0.000001;
    #ifdef TEST_CONV
    if (t%C==0) {
            //**************TODO**************//
            /*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 = 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_WORL
D);
    #ifdef TEST_CONV

MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD)
;
    #endif
    //----Rank 0 gathers local matrices back to the global matrix----//
    if (rank==0) {
          U=allocate2d(global_padded[0],global_padded[1]);
    }
        //*************TODO******************//
        /*Fill your code here*/
     if (rank==0) start = &(U[0][0]);
MPI_Gatherv(&(u_current[1][1]),1,local_block,start,scattercounts,scatteroffset,glob
al_block,0,MPI_COMM_WORLD);
```

Όπως αναφέραμε και στην ανάλυση της σειριακής υλοποίησης της μεθόδου η Red-Black SOR αποτελείται από δύο φάσεις υπολογισμού για κάθε χρονικό βήμα. Στην πρώτη φάση ανανεώνονται οι τιμές των κόκκινων στοιχείων (άρτιες θέσεις) κάθε block με βάσει τις τιμές των τεσσάρων γειτονικών τους στοιχείων που είχαν αποκτήσει στο τέλος του προηγούμενου χρονικού βήματος. Στην δεύτερη φάση ανανεώνονται οι τιμές των μαύρων στοιχείων (περιττές θέσεις) ενός block με βάσει τις τιμές των τεσσάρων γειτονικών τους στοιχείων που έχουν ανανεώσει στο τρέχων χρονικό βήμα. Ο τύπος της ανανέωσης δείχθηκε στην αρχική παρουσίαση της μεθόδου. Συμπεραίνουμε λοιπόν πως ουσιαστικά πριν από κάθε φάση υπολογισμού πρέπει ένα block να λάβει τα απαραίτητα στοιχεία από όλους τους γείτονες του όπως ακριβώς έκανε και στην μέθοδο Jacobi. Μόνο που πριν την πρώτη φάση τα δεδομένα που θα λάβει θα αφορούν την προηγούμενη χρονική στιγμή ενώ πριν την δεύτερη φάση θα αφορούν την τρέχουσα χρονική στιγμή. Έτσι λοιπόν πριν από την πρώτη φάση τα δεδομένα θα μεταφέρονται μεταξύ των πινάκων u_previous των διεργασιών ενώ πριν από την δεύτερη φάση τα δεδομένα θα μεταφέρονται μεταξύ των πινάκων u_current των διεργασιών

σ4.Μετρήσεις βάση σεναρίου

Για τις μετρήσεις έχουμε στην διάθεση μας 8 κόμβους της συστοιχίας clones (ουρά parlab) , που διαθέτουν 8 πυρήνες ο καθένας (συνολικά 64 πυρήνες)

Μετρήσεις με έλεγχο σύγκλισης για μέγεθος πίνακα U 1024x1024:

Αρχικά πραγματοποιήθηκαν μετρήσεις με έλεγχο σύγκλισης για τις 4 παράλληλες υλοποιήσεις που αναλύσαμε παραπάνω για μέγεθος πίνακα U 1024x1024. Για το συγκεκριμένο μέγεθος πίνακα λήφθηκαν μετρήσεις για τον αριθμό των επαναλήψεων , τον συνολικό χρόνο υπολογισμών , τον συνολικό χρόνο για τον έλεγχο σύγκλισης καθώς και για τον συνολικό χρόνο της υλοποίησης (αφορά το υπολογιστικό μέρος του αλγορίθμου και όχι αρχικοποιήσεις , αρχικές αποστολές και τελική συλλογή δεδομένων.). Επιπλέον υπολογίσθηκε και ο συνολικός χρόνος επικοινωνίας ως η διαφορά του συνολικού χρόνου εκτέλεσης και του αθροίσματος των χρόνων υπολογισμού και ελέγχου σύγκλισης. Οι μετρήσεις αυτές πραγματοποιήθηκαν για 64 MPI διεργασίες ενώ δοκιμάστηκαν διαφορετικές διαστάσεις για το Grid (Grid Size). Έτσι προέκυψαν τα ακόλουθα αποτελέσματα:

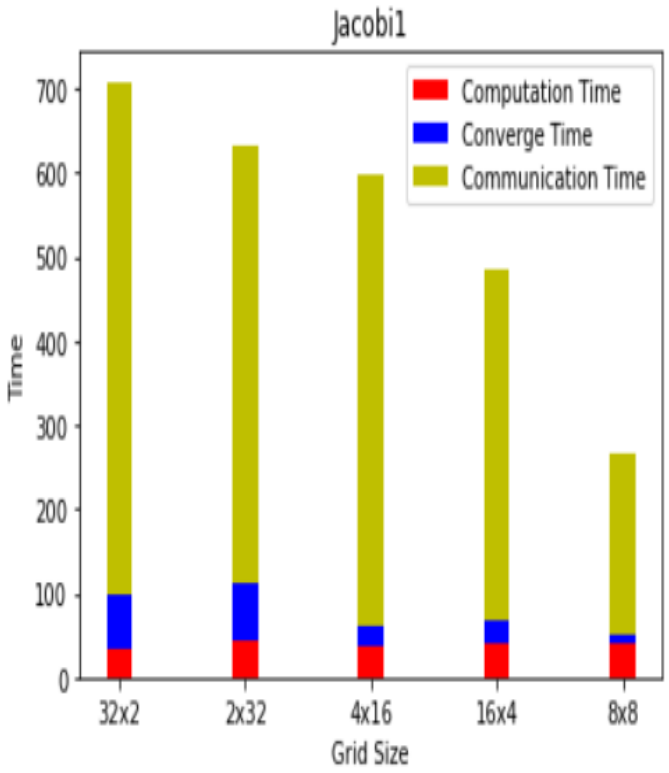
Υλοποίηση	Grid Size	Iterations	Compute Time	Converge Time	Communication Time	Total Time

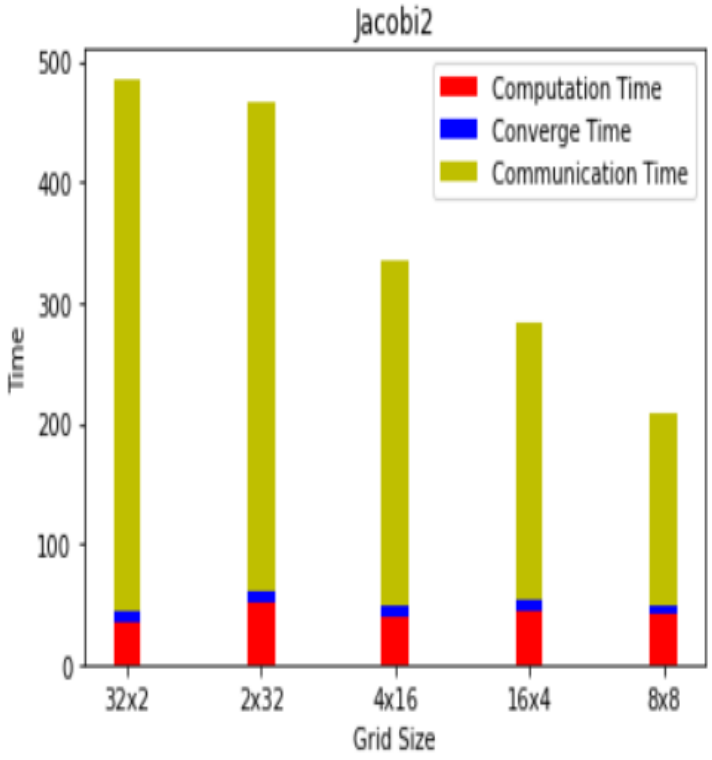
Jacobi Version1	32x2	798201	34.764019	62.132093	611.44225	708.339462
Jacobi Version2	32x2	798201	35.218310	8.435057	441.983654	485.637021
GaussSeidelSOR	32x2	3201	0.530207	0.476545	2.884232	3.890984
RedBlackSOR	32x2	2501	0.464734	0.206941	3.795734	4.467409
Jacobi Version1	2x32	798201	44.080145	68.231766	518.650588	630.962499
Jacobi Version2	2x32	798201	51.210110	8.582161	406.361526	466.153797
GaussSeidelSOR	2x32	3201	0.490884	0.474115	2,768337	3.733336
RedBlackSOR	2x32	2501	0.287103	0.223895	3.364892	3.875890
Jacobi Version1	16x4	798201	36.005642	23.676173	539.81211	599.493925
Jacobi Version2	16x4	798201	40.102799	7.840399	287.325736	335.268934
GaussSeidelSOR	16x4	3201	0.544386	0.250169	2.673706	3.468261
RedBlackSOR	16x4	2501	0.292481	0.140398	3.382311	3.815190
Jacobi Version1	4x16	798201	39.016268	28.334052	419.051414	486.401734
Jacobi Version2	4x16	798201	44.833261	8.150043	230.602537	283.585841
GaussSeidelSOR	4x16	3201	0.512607	0.237817	2.351711	3.102135
RedBlackSOR	4x16	2501	0.289587	0.110522	2.679918	3.080027
Jacobi Version1	8x8	798201	40.325666	9.555106	217.722814	267.603586
Jacobi Version2	8x8	798201	41.818123	7.538981	159.149928	208.507032
GaussSeidelSOR	8x8	3201	0.547604	0.138299	1.370066	2.055969
RedBlackSOR	8x8	2501	0.310279	0.080629	1.366476	1.757384

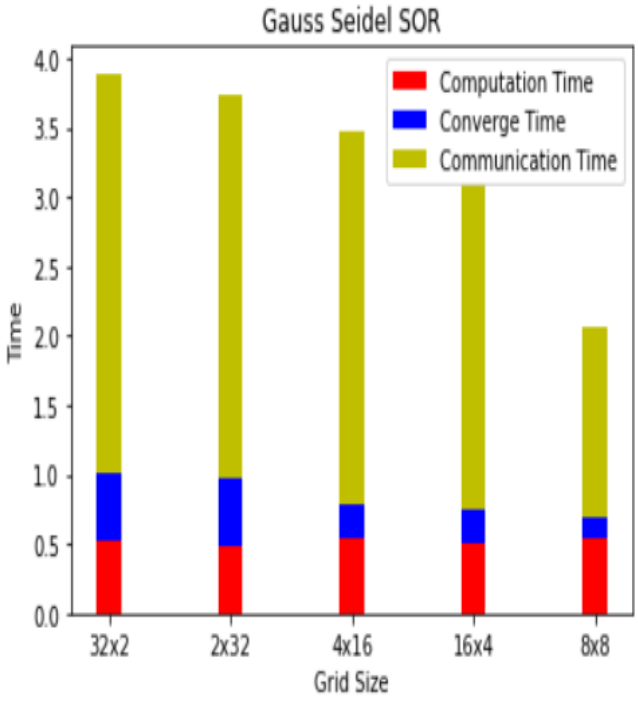
Από τις παραπάνω μετρήσεις μπορούμε να συμπεράνουμε αρχικά πως η καλύτερη επιλογή grid Size είναι 8x8. Η επιλογή αυτή μειώνει σε μεγάλο βαθμό τον χρόνο που

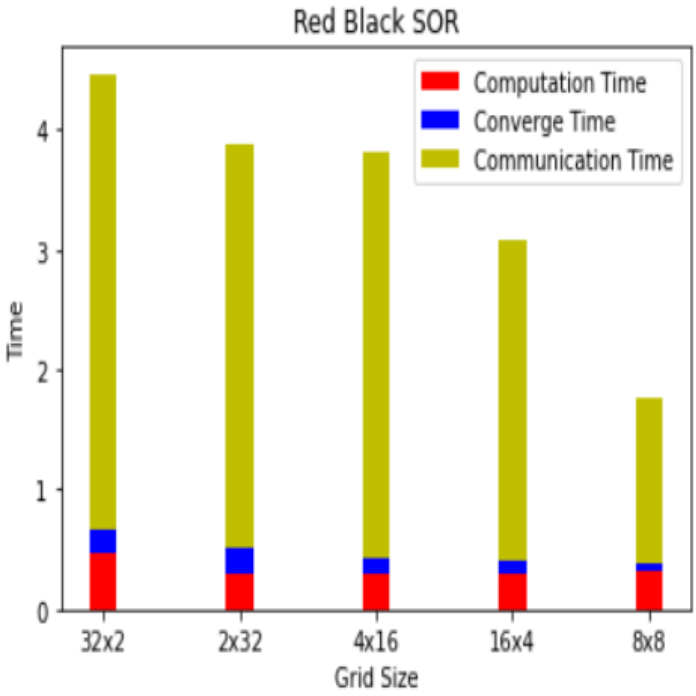
καταναλώνεται στην επικοινωνία και άρα και τον συνολικό χρόνο εκτέλεσης. Ο χρόνος επικοινωνίας μειώνεται καθώς με ίδιες τιμές του grid και στις δύο διαστάσεις μεταφέρεται ο ίδιος αριθμός δεδομένων από κάθε πλευρά του block της κάθε διεργασίας. Σε αντίθετη περίπτωση υπάρχει ανισοκατανομή φορτίου με την μία πλευρά να μεταφέρει μεγαλύτερο όγκο δεδομένων. Κάτι τέτοιο καθυστερεί την διαδικασία της επικοινωνίας.

Στην συνέχεια αναπαριστούμε τις παραπάνω μετρήσεις στις ακόλουθες γραφικές αναπαραστάσεις μπαρών για να έχουμε καλύτερη εικόνα. Κάθε γραφική παράσταση αναπαριστά τους χρόνους Υπολογισμού , Ελέγχου σύγκλισης και Επικοινωνίας για κάθε μία μέθοδο και για όλα τα μεγέθη Grid ου φαίνονται και στον παραπάνω πίνακα:









Σύμφωνα με τις μετρήσεις αλλά και τις γραφικές παραστάσεις που ακολούθησαν έχουμε τις ακόλουθες παρατηρήσεις:

- Επιβεβαιώνουμε και από τις γραφικές αναπαραστάσεις το αρχικό μας συμπέρασμα για το καλύτερο Grid Size (8x8).
- Παρατηρούμε ότι σε όλες τις μεθόδους το μεγαλύτερο μέρος του χρόνου καταναλώνεται στην επικοινωνία μεταξύ των διεργασιών δείχνοντας έτσι πόσο σημαντικό κομμάτι παίζει η επικοινωνία μεταξύ των διεργασιών σε ένα κατανεμημένο σύστημα μνήμης για την απόδοση μίας υλοποίησης.
- Παρατηρούμε πως οι δύο υλοποιήσεις της μεθόδου Jacobi έχουν πολύ μεγαλύτερο συνολικό χρόνο εκτέλεσης από τις άλλες δύο υλοποιήσεις. Κάτι τέτοιο οφείλεται στον πολύ αργό ρυθμό σύγκλισης που έχει σαν μέθοδο η Jacobi. Ο αργός ρυθμός σύγκλισης φαίνεται καθαρά από τον αριθμό των χρονικών βημάτων όπου για τις δύο υλοποιήσεις της μεθόδου Jacobi απαιτούνται 798201 ενώ για τις άλλες δύο μεθόδους απαιτούνται 3201 (Gauss Seidel SOR) και 2501 (Red Black SOR). Παρατηρούμε λοιπόν ότι η διαφορά των απαιτούμενων χρονικών βημάτων είναι πάρα πολύ μεγάλη. Έτσι στην συγκεκριμένη περίπτωση δεν μπορούμε να συγκρίνουμε τους χρόνους υπολογισμού , ελέγχου σύγκλισης και επικοινωνίας για ένα συγκεκριμένο χρονικό βήμα μεταξύ των υλοποιήσεων σε Jacobi και των άλλων υλοποιήσεων καθώς η διαφορά στα απαιτούμενα χρονικά βήματα είναι πάρα πολύ μεγάλη.

- Παρατηρούμε ότι η δεύτερη υλοποίηση της Μεθόδου Jacobi πράγματι βελτιώνει τον χρόνο εκτέλεσης σε σχέση με την πρώτη. Ο χρόνος υπολογισμού και των δύο υλοποιήσεων κινείται στα ίδια επίπεδα κάτι λογικό καθώς και οι δύο υλοποιήσεις υπολογίζουν με τον ίδιο τρόπο την διάδοση της θερμότητας. Επίσης και ο χρόνος που απαιτείται στον έλεγχο σύγκλισης δεν παρουσιάζει μεγάλη διαφορά. Η μεγάλη διαφορά παρατηρείται στον χρόνο επικοινωνίας όπου η δεύτερη υλοποίηση έχει σαφώς μικρότερες τιμές. Κάτι τέτοιο οφείλεται όπως έχουμε εξηγήσει και παραπάνω στο γεγονός πως στην δεύτερη υλοποίηση η ανανέωση των τιμών των εσωτερικών στοιχείων ενός block γίνεται παράλληλα με την επικοινωνία των διεργασιών. Κάτι τέτοιο λοιπόν μειώνει τον συνολικό χρόνο που απαιτείται για την επικοινωνία των διεργασιών αφού αυτός επικαλύπτεται με τον χρόνο υπολογισμών.
- Ανάμεσα στις δύο υλοποιήσεις για την Gauss Seidel SOR και την Red Black SOR παρατηρούμε πως η υλοποίηση της Red Black SOR οδηγεί σε λίγο μικρότερο συνολικό χρόνο εκτέλεσης. Αυτό οφείλεται στο γεγονός πως η Red Black SOR έχει τον καλύτερο ρυθμό σύγκλισης κάτι που φαίνεται από τον αριθμό των απαιτούμενων χρονικών βημάτων τα οποία είναι 3201 έναντι 1501 της Gauss Seidel SOR .
- Παρατηρούμε ότι ο χρόνος ελέγχου σύγκλισης δεν παίζει καθοριστικό ρόλο στην απόδοση των υλοποιήσεων. Ο χρόνος αυτός παρατηρείται να είναι μεγαλύτερος όσο το Grid Size είναι περισσότερο ανισοκατανεμμημένο στις δύο του διαστάσεις. Ο λόγος που κάτι τέτοιο συμβαίνει είναι παρόμοιος με τον λόγο που κάτι τέτοιο παρατηρείται και στον χρόνο επικοινωνία όπως εξηγήσαμε προηγουμένως. Επειδή για να ελεγχθεί η συνολική σύγκλιση σε κάθε χρονικό βήματα χρειάζεται η κάθε διεργασία να κάνει broadcast σε όλες τις υπόλοιπες διεργασίες την μεταβλητή converged ώστε να τους πληροφορήσει εάν βρίσκεται σε κατάσταση σύγκλισης ή όχι , η επικοινωνία αυτή επιβαρύνεται με μία τέτοια ανισοκατανομή.

Σαν μέθοδο με βάση τις παραπάνω μετρήσεις και αφού δεν μπορούμε να συγκρίνουμε ακόμα σχετικά καλά τους αλγορίθμους μεταξύ τους καθώς κύριο ρόλο φαίνεται να παίζει ο ρυθμός σύγκλισης τους θα επιλέγαμε την Red Black SOR η οποία παρουσιάζει τον ελάχιστο συνολικό χρόνο εκτέλεσης.

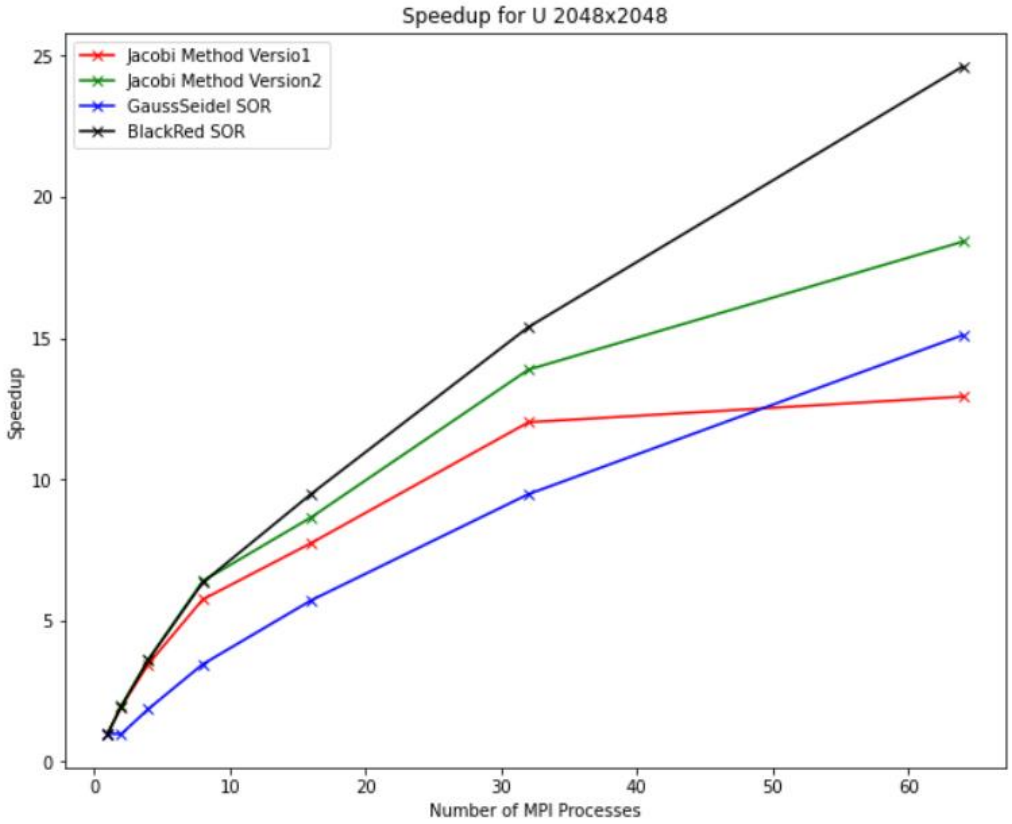
Στην συνέχεια απενεργοποιούμε τον έλεγχο σύγκλισης και προχωράμε σε επιπλέον μετρήσεις για καλύτερα συμπεράσματα.

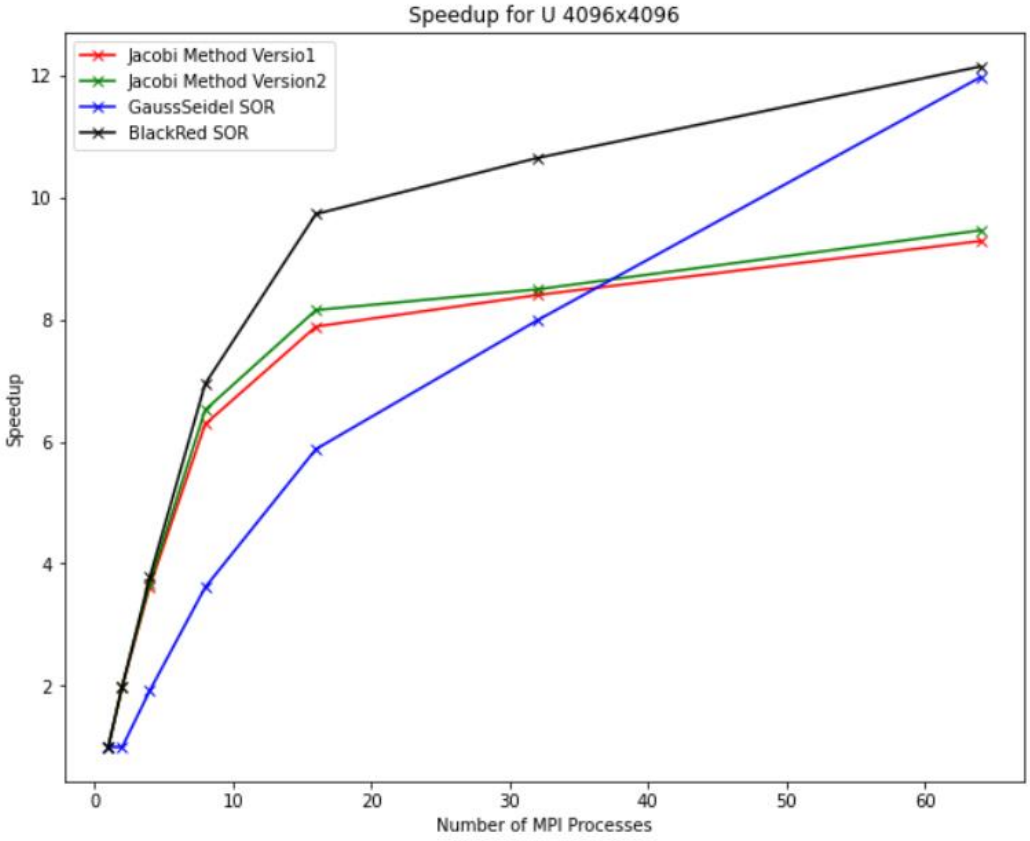
Μετρήσεις χωρίς έλεγχο σύγκλισης

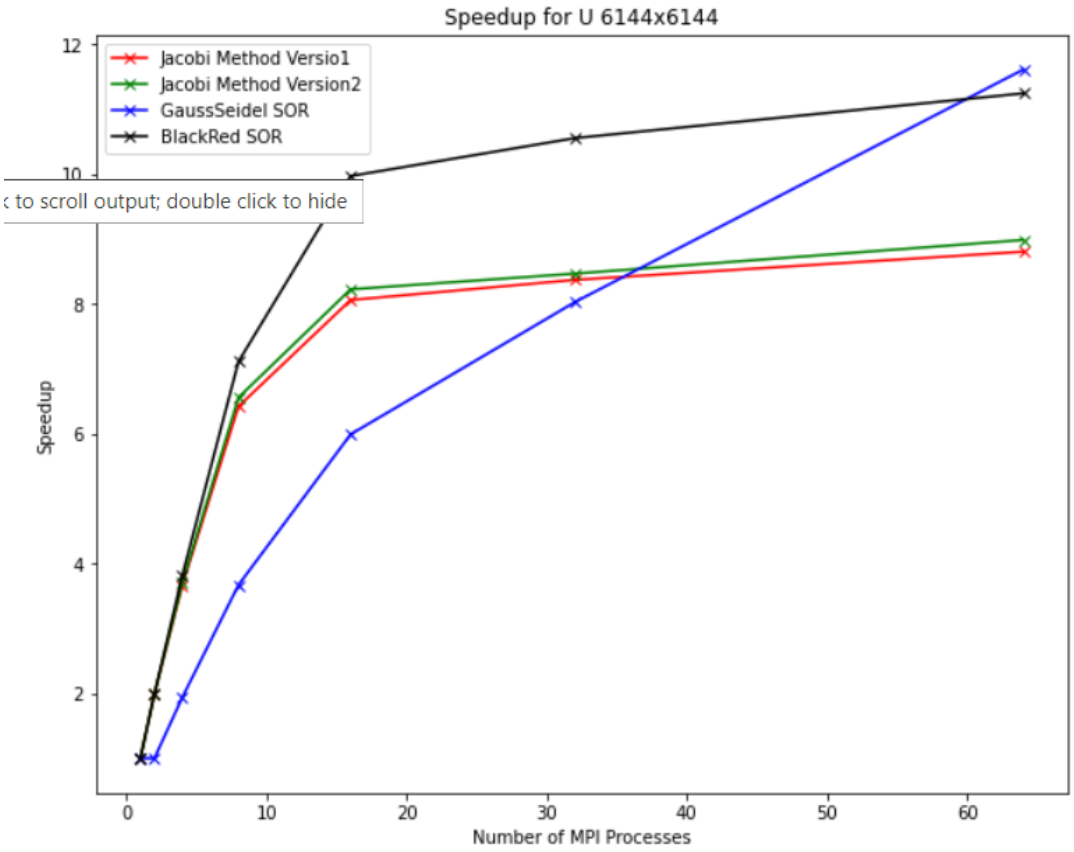
Στην συνέχεια λοιπόν προχωράμε σε μετρήσεις χωρίς έλεγχο σύγκλισης με σταθερό αριθμό χρονικών βημάτων T = 256. Στο συγκεκριμένο κομμάτι πάρθηκαν μετρήσεις για μεγέθη πίνακα U 2048x2048 , 4096x4096 και 6144x6144 για 1, 2, 4, 8, 16, 32, 64

διεργασίες MPI. Και πάλι οι μετρήσεις που πάρθηκαν αφορούσαν , τον συνολικό χρόνο υπολογισμών καθώς και τον συνολικό χρόνο της υλοποίησης. (αφορά το υπολογιστικό μέρος του αλγορίθμου και όχι αρχικοποιήσεις , αρχικές αποστολές και τελική συλλογή δεδομένων.). Επιπλέον υπολογίσθηκε και ο συνολικός χρόνος επικοινωνίας ως η διαφορά του συνολικού χρόνου εκτέλεσης από τον συνολικό χρόνο υπολογισμών.

Αρχικά λοιπόν κατασκευάσαμε ένα διάγραμμα επιτάχυνσης (speedup) για κάθε μέγεθος πίνακα. Το κάθε διάγραμμα παρουσιάζει το speedup και των τριών μεθόδων. Το speedup υπολογίζεται ως Ts / Tp όπου Ts ο σειριακός χρόνος εκτέλεσης και Tp ο χρόνος εκτέλεσης του παράλληλου αλγορίθμου. Έτσι έχουμε:



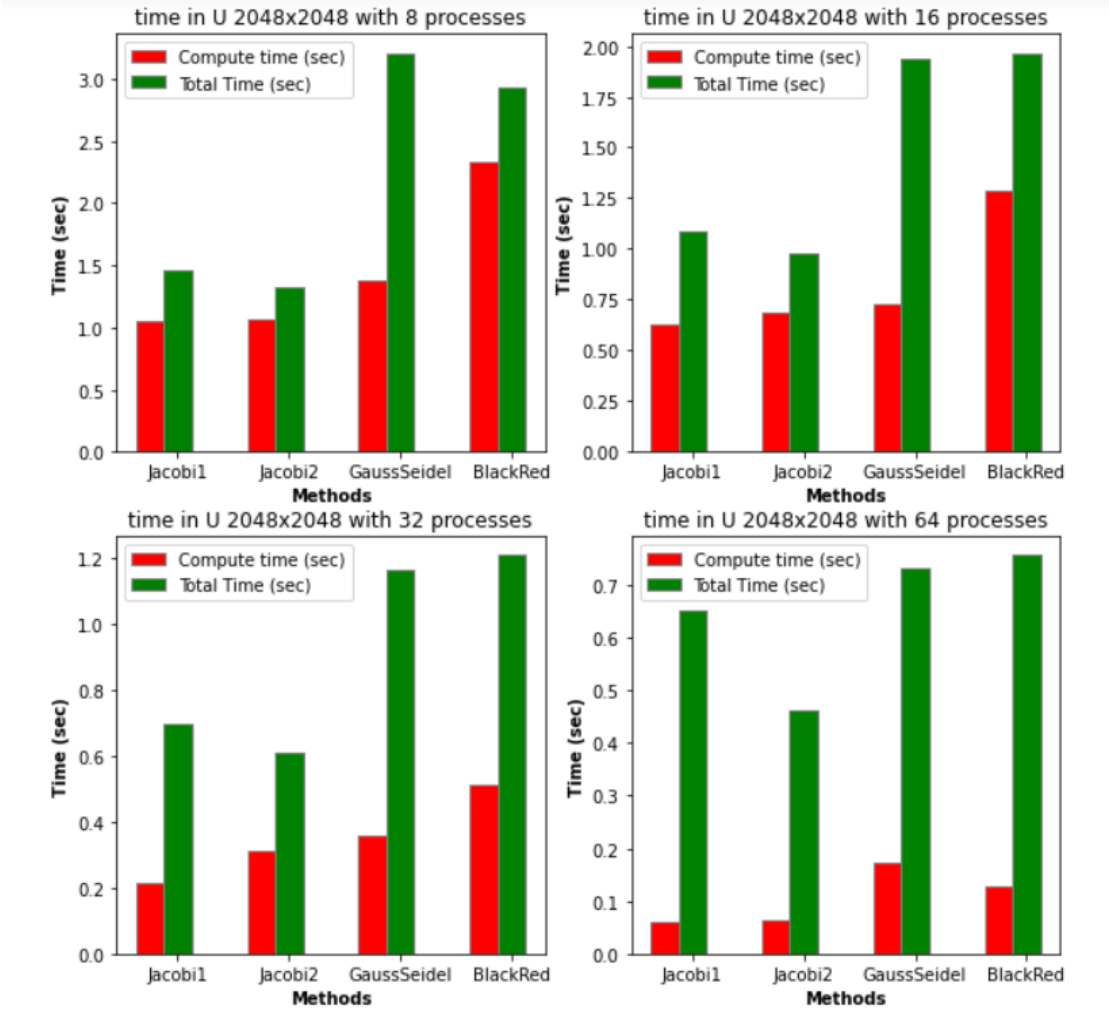

Speedup for U 4096x4096
Jacobi Method Versio1
Jacobi Method Version2
GaussSeidel SOR
BlackRed SOR
Speedup
Number of MPI Processes
0
10
20
30
40
50
60
2
4
6
8
10
12

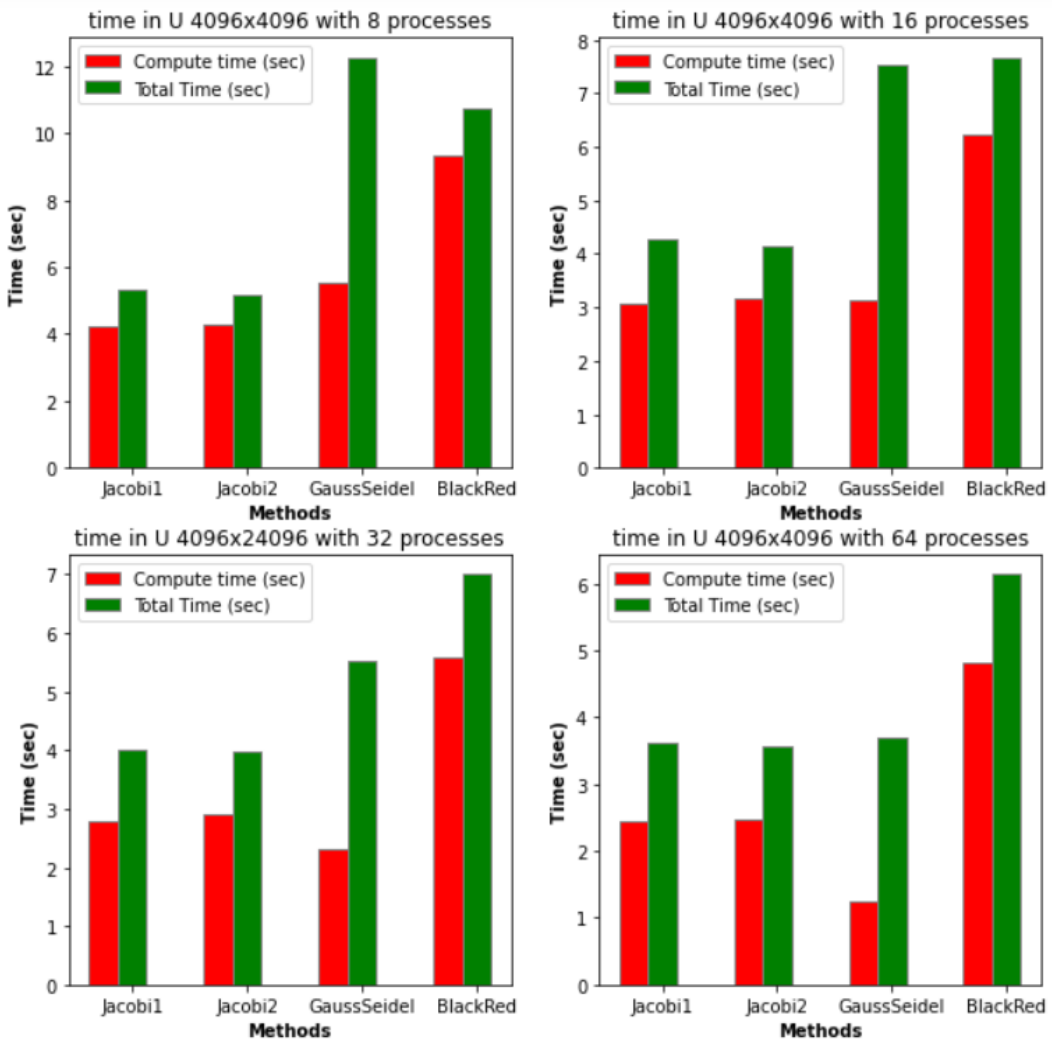


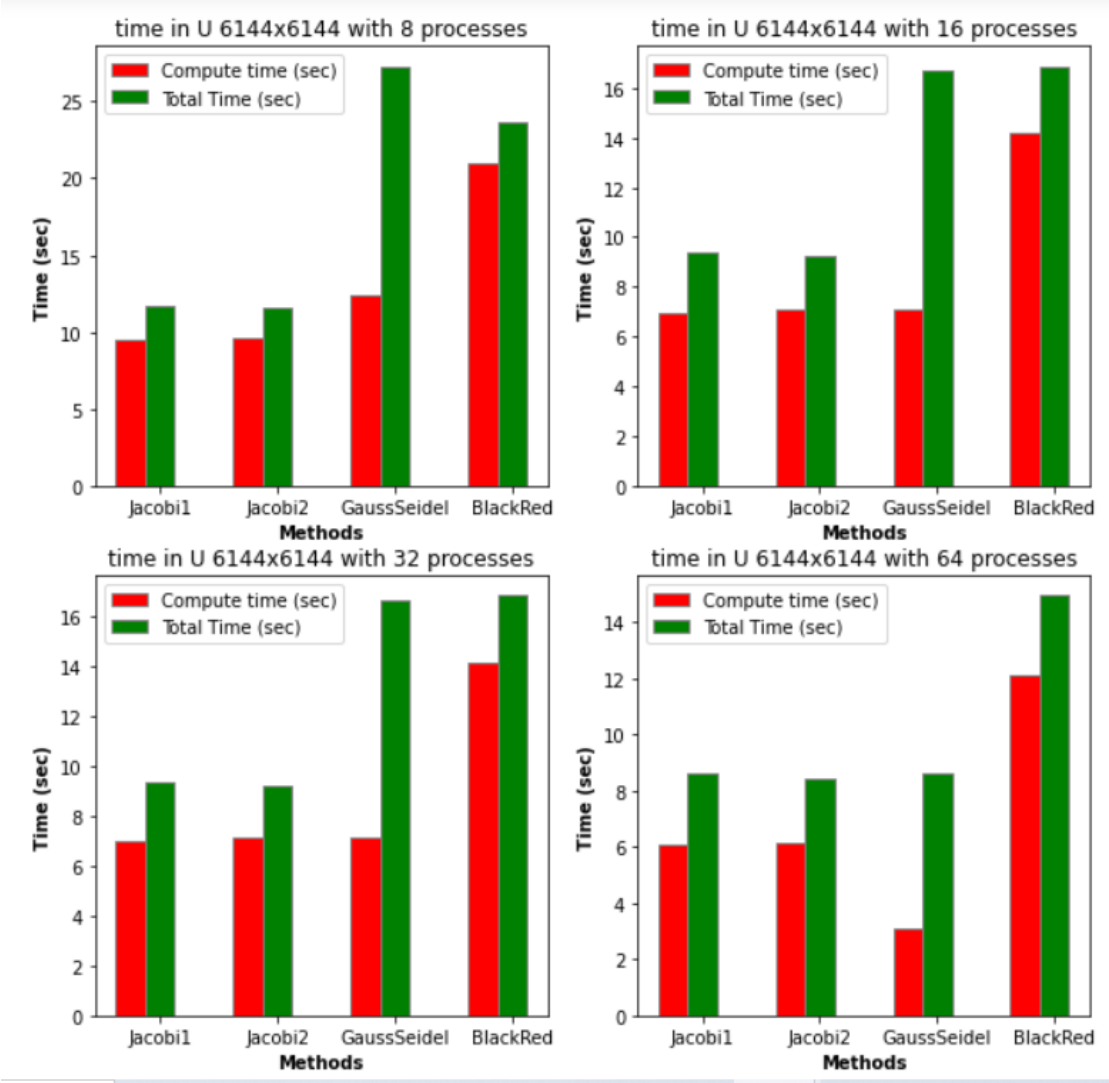
Παρατηρήσεις:

- Για μέγεθος πίνακα 2048x2048 η Gauss Seidel SOR και η Red Black SOR παρουσιάζουν σχεδόν γραμμικό speedup. Αντίθετα οι υλοποιήσεις που αφορούν την μέθοδο Jacobi χάνουν το γραμμικό speedup για παραπάνω από 8 MPI διεργασίες.
- Για μεγαλύτερα μεγέθη πίνακα παύει πλέον να ισχύει το γραμμικό speedup και για τις Gauss Seidel SOR και Red Black SOR. Μάλιστα το speedup που «χαλάει περισσότερο» είναι αυτό της Black Red SOR σε σχέση με την Gauss Seidel SOR με αποτέλεσμα μάλιστα το speedup της Gauss Seidel SOR να ξεπερνάει αυτό της Red Black SOR για μέγεθος πίνακα 6144x6144.
- Παρατηρούμε πως τα speedup των δύο υλοποιήσεων της μεθόδου Jacobi παρουσιάζουν την χειρότερη συμπεριφορά καθώς εμφανώς παύουν να είναι γραμμικά για πάνω από 8 MPI διεργασίες. Μάλιστα η συμπεριφορά αυτή παρατηρείται ακόμα και για μέγεθος πίνακα 2048x2048, Παρόλα αυτά μέχρι τις 32 MPI διεργασίες συνεχίζουν να έχουν μεγαλύτερο speedup από τις άλλες δύο μεθόδους και για τα τρία μεγέθη πίνακα. Αυτή που έχει το μεγαλύτερο είναι η έκδοση 2.

Στην συνέχεια κατασκευάζουμε διαγράμματα με μπάρες (1 για κάθε μέγεθος πίνακα και αριθμό επεξεργαστών) τα οποία απεικονίζουν τον συνολικό χρόνο εκτέλεσης και τον χρόνο υπολογισμού για κάθε μία από τις τέσσερις υλοποιήσεις. Για λόγους εποπτείας κατασκευάσαμε διαγράμματά μόνο για 8 , 16 , 32 και 64 MPI διεργασίες. Έτσι έχουμε:




time in U 4096x4096 with 8 processes
Compute time (sec)
Total Time (sec)
Time (sec)
Methods
Jacobi1
Jacobi2
GaussSeidel
BlackRed
time in U 4096x4096 with 16 processes
time in U 4096x24096 with 32 processes
time in U 4096x4096 with 64 processes



Παρατηρήσεις:

- Παρατηρούμε πως τον μικρότερο συνολικό χρόνο εκτέλεσης τον πετυχαίνει η δεύτερη υλοποίηση της μεθόδου Jacobi για όλα τα μεγέθη του πίνακα. Η διαφορά βέβαια με την πρώτη υλοποίηση της μεθόδου Jacobi γίνεται ελάχιστη καθώς αυξάνεται το μέγεθος του πίνακα. Το γεγονός ότι η μέθοδος Jacobi παρουσιάζει τον μικρότερο χρόνο για T = 256 χρονικά βήματα δεν πρέπει να μας κάνει να θεωρήσουμε πως αποτελεί την καλύτερη μέθοδο καθώς όπως είδαμε και προηγουμένως θα έχει κάνει την μικρότερη πρόοδο ως προς την σύγκλιση των τιμών του πίνακα μέχρι αυτό το σημείο. Ο λόγος που παρουσιάζει τον καλύτερο συνολικό χρόνο είναι εμφανώς ο χρόνος επικοινωνίας που είναι ο μικρότερος. Κάτι τέτοιο οφείλεται στο γεγονός πως οι γείτονες ανταλλάσσουν τις απαραίτητες τιμές που είχαν υπολογίσει από τις προηγούμενες χρονικές στιγμές. Αντίθετα στην Gauss Seidel όπως είδαμε η επικοινωνία είναι πιο περίπλοκη ενώ αντίστοιχα στην Red Black SOR

πραγματοποιείται διπλή επικοινωνία μεταξύ των γειτονικών διεργασιών (μία για κάθε φάση).

- Παρατηρούμε πως η αύξηση των διεργασιών για σταθερό μέγεθος πίνακα μειώνει τον χρόνο υπολογισμού και αυξάνει τον χρόνο επικοινωνίας για όλες τις μεθόδους. Κάτι τέτοιο όπως γνωρίζουμε είναι συχνό στα συστήματα κατανεμμημένης μνήμης καθώς η αύξηση του αριθμού των διεργασιών οδηγεί στην ανάγκη επικοινωνίας όλο και περισσότερων διεργασιών και άρα στην μεγαλύτερη συμφόρηση στο δίκτυο διασύνδεσης.
- Παρατηρούμε ότι για τον ίδιο αριθμό διεργασιών η αύξηση του μεγέθους του πίνακα αυξάνει τον χρόνο υπολογισμών μειώνοντας το χάσμα μεταξύ χρόνου υπολογισμών και χρόνου επικοινωνίας. Κάτι τέτοιο είναι λογικό καθώς για σταθερό αριθμό διεργασιών δεν αλλάζουν οι απαιτήσεις επικοινωνίας. Η αύξηση του μεγέθους του πίνακα βέβαια οδηγεί στην μεταφορά όλο και μεγαλύτερου όγκου δεδομένων μεταξύ των διεργασιών.
- Παρατηρούμε ότι για όλα τα μεγέθη του πίνακα και για ίδιο αριθμό διεργασιών ο συνολικός χρόνος υπολογισμών της Red Black SOR είναι μεγαλύτερος από όλους τους άλλους. Κάτι τέτοιο οφείλεται στο γεγονός πως σε κάθε χρονικό βήμα η μέθοδος αυτή κάνει δύο περάσματα στον κάθε block προκειμένου να ανανεώσει τις τιμές του (2 φάσης υπολογισμών).
- Παρατηρούμε ότι για το ίδιο μέγεθος πίνακα η αύξηση του αριθμού των διεργασιών οδηγεί σε μεγαλύτερη μείωση του συνολικού χρόνου εκτέλεσης της Gauss Seidel SOR έναντι της Red Black SOR. Κάτι τέτοιο οφείλεται στο γεγονός πως στην Red Black SOR οι διεργασίες επικοινωνούν με όλους τους τους γείτονες δύο φορές σε ένα χρονικό βήμα. Η αύξηση λοιπόν του αριθμού των διεργασιών οδηγεί στην μεγαλύτερη επιβάρυνση στον χρόνο επικοινωνίας της Red Black SOR παρόλο που ο τρόπος επικοινωνίας της Gauss Seidel SOR είναι πιο περίπλοκος. Έτσι τελικά για 64 MPI διεργασίες ο συνολικός χρόνος εκτέλεσης της Gauss Seidel SOR γίνεται μικρότερος από αυτόν της Red Black SOR.

Γενικά λοιπόν συμπεράσματα για τις τρεις μεθόδους:

- Με έλεγχο σύγκλισης είδαμε πως η μέθοδος Jacobi παρουσιάζει πολύ μεγαλύτερο χρόνο εκτέλεσης εξαιτίας του πολύ αργού ρυθμού σύγκλισης που παρουσιάζει. Η μέθοδος που παρουσιάζει τον μικρότερο συνολικό χρόνο εκτέλεσης είναι η μέθοδος Red Blacks SOR η οποία έχει και τον γρηγορότερο ρυθμό σύγκλισης.
- Απενεργοποιώντας τον έλεγχο σύγκλισης και κρατώντας σταθερό τον αριθμό των χρονικών βημάτων είδαμε πως η Jacobi παρουσιάζει τον μικρότερο χρόνο εκτέλεσης εξαιτίας του μικρότερου χρόνου επικοινωνίας που απαιτείται σε αυτήν. Παρατηρήσαμε όμως πως οι Gauss Seidel SOR και η Red Black SOR κλιμακώνουν καλύτερα για μεγαλύτερο αριθμό διεργασιών.

- Για σταθερό αριθμό βημάτων είδαμε επίσης πως για μεγάλο αριθμό διεργασιών η Gauss Seidel SOR παρουσιάζει καλύτερα αποτελέσματα από την Red Black SOR εξαιτίας του καλύτερου τρόπου ανανέωσης των τιμών (1 πέρασμα αντί για 2) αλλά και του γεγονότος ότι το κόστος επικοινωνίας της επιβαρύνεται λιγότερο από την αύξηση των διεργασιών. Μάλιστα για 64 MPI διεργασίες η Red Black SOR φτάνει τον συνολικό χρόνο εκτέλεσης της Jacobi. Κάτι τέτοιο σαφώς και οφείλεται στον non blocking τρόπο επικοινωνίας μεταξύ των διεργασιών έναντι του blocking τρόπου επικοινωνίας των άλλων δύο μεθόδων.
- Η αύξηση του μεγέθους του πίνακα επιβαρύνει όλες τις μεθόδους λόγο του μεγάλου φόρτου που δίνει στην επικοινωνία μεταξύ των διεργασιών.

Ως καλύτερη επιλογή μάλλον θα επιλέγαμε τελικά την Gauss Seidel SOR καθώς παρόλο που η Red Black SOR παρουσιάζει καλύτερο τελικό χρόνο εκτέλεσης με τον μεγαλύτερο ρυθμό σύγκλισης η Gauss Seidel SOR παρουσιάζει μεγαλύτερη κλιμακωσιμότητα , στοιχείο πολύ σημαντικό στα συστήματα κατανεμμημένης μνήμης όπου οι καλά κλιμακώσιμες υλοποιήσεις μπορούν να εκμεταλλευτούν καλύτερα τα οφέλη που προσφέρει ένα κατανεμμημένο σύστημα μνήμης. Ένα βέβαια κάποιος ενδιαφερόταν αποκλειστικά να πετύχει τον χαμηλότερο χρόνο εκτέλεσης θα μπορούσε να επιλέξει την Red Black SOR.

Σημειώσεις και οδηγίες

- Η ορθότητα των παραπάνω παράλληλων MPI διεργασιών ελέγχθηκε συγκρίνοντας τα αποτελέσματα συγκριτικά με την αντίστοιχη σειριακή περίπτωση η υλοποίηση της οποίας μας είχε δοθεί.
- Στο παραδοτέο έχουμε συμπεριλάβει τα .c αρχεία Jacobi1.c , Jacobi2.c , Gauss-Seidel_SOR.c , Red-Black_SOR.c τα οποία περιέχουν τους κώδικες των υλοποιήσεων μας. Επιπλέον έχουμε συμπεριλάβει και ένα έτοιμο Makefile για την μεταγλώττιση τους. Τα εκτελέσιμα που προκύπτουν είναι αντίστοιχα τα jacobi_mpi , jacobi2_mpi , gaussSeidel_mpi , redblack_mpi.
- Η μεταγλώττιση έγινε με -O3 για εκμετάλλευση των optimizations του επεξεργαστή.