Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Ε.Μ.Π.

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

2016-2017

Άσκηση 3

Παράλληλη επίλυση εξίσωσης θερμότητας

Βαρηά Χρυσούλα - 03112105
Κουτσανίτη Ειρήνη - 03112135

Για την επίλυση του προβλήματος της διάδοσης θερμότητας στις δύο διαστάσεις υλοποιήθηκαν τρεις μέθοδοι: η μέθοδος Jacobi, η μέθοδος Gauss-Seidel με Successive Over Relaxation και η μέθοδος Red-Black SOR.

Ανάλυση κώδικα

Ακολουθεί μια σύντομη περιγραφή των αλλαγών στο αρχείο mpi_skeleton.c για κάθε μέθοδο. Οι υλοποιήσεις των προγραμμάτων παρουσιάζονται στο τέλος της αναφοράς.

Μέθοδος Jacobi

Αρχικά γίνεται δέσμευση και αρχικοποίηση των πινάκων u_current και u_previous, οι οποίοι χρησιμοποιούνται για την αποθήκευση των δεδομένων κάθε διεργασίας την χρονική στιγμή t και t-1 αντίστοιχα. Οι παραπάνω πίνακες έχουν δύο επιπλέον γραμμές και στήλες (σύνορα), οι οποίες χρησιμοποιούνται για την αποθήκευση των οριακών τιμών των δεδομένων γειτονικών διεργασιών.

Ο διαμοιρασμός των επιμέρους τμημάτων του πίνακα γίνεται με τη χρήση της συνάρτησης MPI_Scatterv, με την οποία η διεργασία με rank 0 αναλαμβάνει την αποστολή των δεδομένων προς τις υπόλοιπες διεργασίες. Ο τύπος των δεδομένων αποστολής από τη διεργασία root είναι global_block, ενώ ο τύπος λήψης από τις υπόλοιπες διεργασίες είναι local_block. Τα ορίσματα scattercounts και scatteroffset καθορίζουν το πλήθος των δεδομένων και την κατάλληλη θέση του πίνακα, από την οποία αρχίζει η αποστολή τους για κάθε διεργασία. Τα δεδομένα, τα οποία λαμβάνει κάθε διεργασία αποθηκεύονται τοπικά στον πίνακα u_current. Σημειώνεται ότι για να είναι επιτυχής ο διαμοιρασμός πρέπει να έχει προηγηθεί η σωστή δεικτοδότηση στον πίνακα U από τη διεργασία με rank 0. Αυτό επιτυγχάνεται με τη δέσμευση και αρχικοποίηση ενός δείκτη addr. Για την συγκέντρωση των επιμέρους τμημάτων του τελικού πίνακα γίνεται η αντίστροφη διαδικασία με χρήση της συνάρτησης MPI_Gatherv.

Η εύρεση των γειτόνων γίνεται αρχικά με τη χρήση της MPI_Cart_shift, με την οποία για συγκεκριμένη κατεύθυνση προσδιορίζει τα ranks των γειτονικών διεργασιών (north,south, east, west). Έπειτα καθορίζονται τα όρια (i_max, i_min, j_max, j_min) του κάθε τοπικού πίνακα για τον υπολογισμό των επιμέρους τιμών των δεδομένων. Κατά την παραπάνω διαδικασία εξετάζονται οι εξής περιπτώσεις:

- Η διεργασία έχει 4 γείτονες. Σε αυτή την περίπτωση εξασφαλίζεται ότι οι πίνακες u_current, u_previous περιέχουν δεδομένα σε όλες τις γραμμές και τις στήλες ανεξαρτήτως αν ο global πίνακας είναι padded ή όχι. Τα δεδομένα αυτά δεν αποτελούν αρχικές οριακές συνθήκες, επομένως σε κάθε χρονική στιγμή πρέπει να υπολογίζονται οι νέες τιμές τους.

- Η διεργασία έχει λιγότερους από 4 γείτονες. Ανάλογα με τους γείτονες της τα όρια λαμβάνουν διαφορετικές τιμές.

  Αν η διεργασία δεν έχει γείτονα north ή west, τότε περιέχει μια γραμμή ή στήλη αντίστοιχα με αρχικές οριακές συνθήκες, τις οποίες πρέπει να αφήσει αμετάβλητες. Αν η διεργασία δεν έχει γείτονα south ή east, εκτός από τις αρχικές οριακές συνθήκες, τα όρια επηρεάζονται αν ο global πίνακας είναι padded. Για παράδειγμα αν ο πίνακας είναι padded ως προς τις γραμμές, τότε οι πίνακες των διεργασίων, οι οποίες δεν έχουν γείτονα south, δεν περιέχουν δεδομένα προς υπολογισμό σε όλες τις γραμμές.

Για την ανταλλαγή δεδομένων μεταξύ γειτονικών διεργασιών κάθε χρονική στιγμή, γίνεται χρήση των non-blocking MPI_Isend και MPI_Irecv. Για να εξασφαλιστεί ότι οι διεργασίες λαμβάνουν ό,τι δεδομένα χρειάζονται για τον υπολογισμό της επόμενης χρονικής στιγμής, χρησιμοποιείται

η MPI_Waitall. Με αυτόν τον τρόπο αποφεύγεται μια διεργασία να γίνεται block σε κάθε αίτημα αποστολής ή λήψης δεδομένων περιμένοντας να ολοκληρωθεί μεμονωμένα η μεταφορά τους. Αντίθετα η κάθε διεργασία πρώτα στέλνει και λαμβάνει δεδομένα από όλους τους γείτονες της και έπειτα μπλοκάρει για να εξασφαλίσει ότι έχουν ολοκληρωθεί όλες οι παραπάνω ανταλλαγές. Επισημαίνεται ότι για την ανταλλαγή δεδομένων στήλης ορίζεται και χρησιμοποιείται η δομή vertical_border.

Η εύρεση της τοπικής σύγκλισης κάθε διεργασίας επιτυγχάνεται μέσω της δοθείσας συνάρτησης converge. Αντίστοιχα για την εύρεση της ολικής σύγκλισης γίνεται χρήση της MPI_Allreduce με τον τελεστή AND. Αν όλες οι διεργασίες συγκλίνουν τοπικά, τότε το αποτέλεσμα της λογικής σύζευξης είναι μονάδα, αλλιώς μηδέν. Μέχρι να καθοριστεί το παραπάνω αποτέλεσμα, όλες οι διεργασίες μπλοκάρουν.

Επίσης ο υπολογισμός του πίνακα γίνεται σύμφωνα με τις αντίστοιχες διαφάνειες της παρουσίασης, χρησιμοποιώντας τη συνάρτηση Jacobi. Η χρονική εξάρτηση των δεδομένων αντιμετωπίζεται με τη χρήση των πινάκων u_current, u_previous, οι οποίοι εναλλάσσονται σε κάθε επανάληψη.

Τέλος ο ολικός χρόνος και ο χρόνος των υπολογισμών βρίσκεται με τη χρήση των μεταβλητών ttotal και tcomp. Από τους παραπάνω χρόνους, οι οποίοι υπολογίζονται για κάθε διεργασία ξεχωριστά, επιλέγονται οι μέγιστοι με χρήση του MPI_Reduce (με όρισμα τελεστή MPI_MAX).

Μέθοδος Gauss-Seidel SOR

Η υλοποίηση της μεθόδου Gauss-Seidel είναι όμοια με αυτή της μεθόδου Jacobi. Η διαφορά έγκειται στον τρόπο ανταλλαγής των δεδομένων σε κάθε επανάληψη. Επειδή ο υπολογισμός κάθε στοιχείου του τοπικού πίνακα εξαρτάται από τιμές τόσο της προηγούμενης χρονικής στιγμής όσο και της παρούσας, κάθε διεργασία λαμβάνει τα μισά δεδομένα πριν τον υπολογισμό των νέων τιμών (από τους γείτονες north, west αν υπάρχουν) και τα άλλα μισά μετά από αυτόν. Η ίδια έπειτα στέλνει τις νέες τιμές, τις οποίες υπολόγισε σε όλους τους γείτονες της.

Μέθοδος Red-Black SOR

Η υλοποίηση της μεθόδου Red-Black είναι όμοια με τις δύο προηγούμενες, επομένως αναλύονται μόνο οι τροποποιήσεις, οι οποίες γίνονται.

Αρχικά ορίζονται δύο δομές για την ανταλλαγή δεδομένων μεταξύ γειτονικών διεργασιών (vertical_border, horizontal_border). Η επικοινωνία γίνεται σε δύο μέρη, πρώτα οι διεργασίες στέλνουν και λαμβάνουν τα γειτονικά κελιά που βρίσκονται σε θέσεις με περιττό άθροισμα γραμμής και στήλης (στη συνέχεια θα αναφέρονται σαν περιττές θέσεις) για τον υπολογισμό της πρώτης φάσης (Red) και μετά από αυτό τον υπολογισμό στέλνονται και λαμβάνονται τα γειτονικά κελιά που βρίσκονται σε θέσεις με άρτιο άθροισμα γραμμής και στήλης (στη συνέχεια θα αναφέρονται σαν άρτιες θέσεις) για τον υπολογισμό του υπόλοιπου πίνακα.

Για τη σωστή μεταφορά των δεδομένων είναι απαραίτητος ο έλεγχος του πλήθους των γραμμών και των στηλών των τοπικών πινάκων u_current και u_previous. Όταν οι διαστάσεις local[0] ή local[1] έχουν περιττό μέγεθος, το πρώτο κελί των πινάκων u_current και u_previous δεν αντιστοιχεί σε κελί άρτιας θέσης του global πίνακα, για όλες τις διεργασίες. Γι' αυτό το λόγο γίνεται χρήση των μεταβλητών fix_red, fix_black, οι οποίες χρησιμοποιούνται ως ορίσματα στις συναρτήσεις RedSOR και BlackSOR. Επιπλέον, ορίζονται και οι μεταβλητές rec_nw, send_nw, rec_s, send_s, rec_e και send_e με αντίστοιχο τρόπο για την ανταλλαγή των σωστών δεδομένων σε κάθε επανάληψη (καθορίζουν από ποια θέση του πίνακα u_previous ξεκινά η αποστολή και λήψη δεδομένων από τις γειτονικές διεργασίες).

Μετρήσεις με έλεγχο σύγκλισης

Οι ζητούμενες μετρήσεις για μέγεθος πίνακα 1024 x 1024 και 64 διεργασίες παρουσιάζονται παρακάτω:

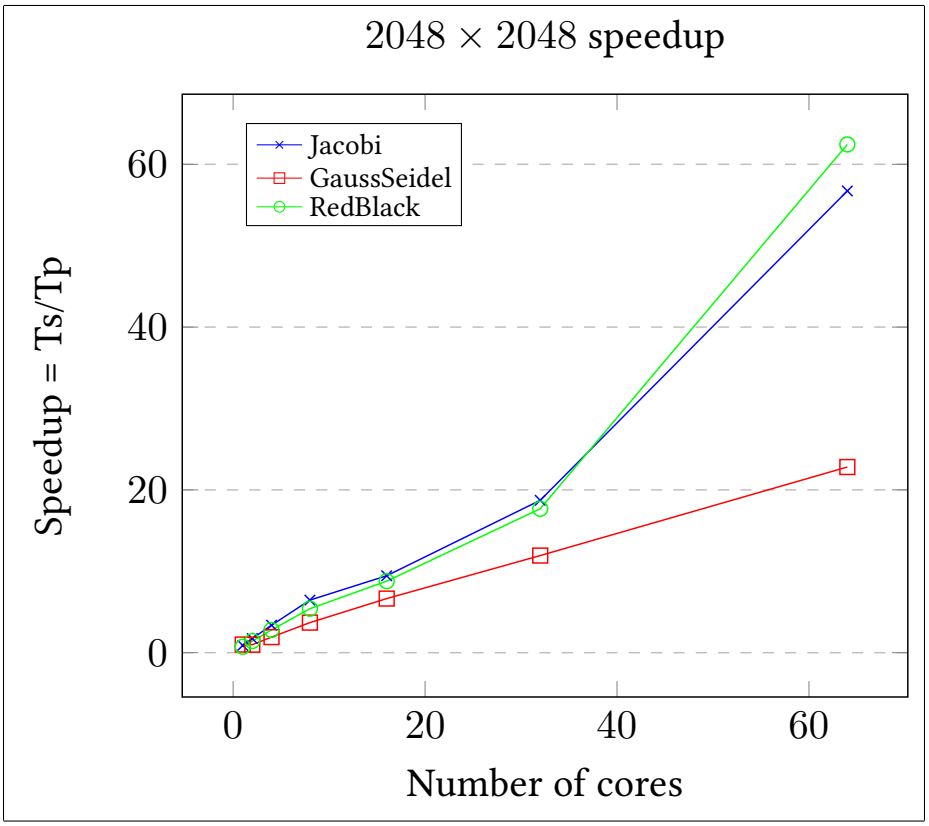
	Computation	Communication	Converge	Total	Iteration
Jacobi	36.638	175.015	7.673	222.326	798200
Gauss - Seidel	0.602	1.359	0.143	2.105	3200
Red Black	0.369	0.662	0.017	1.049	2500

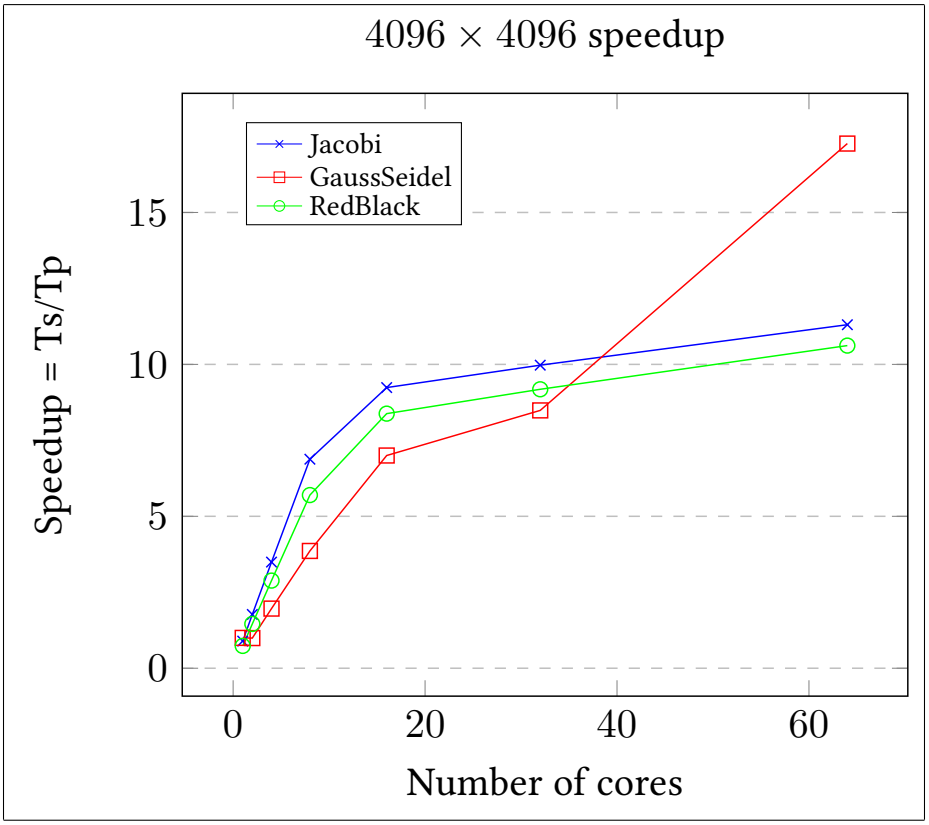
Η μέθοδος Jacobi παρουσιάζει τον υψηλότερο χρόνο εκτέλεσης (όλοι οι χρόνοι σύγκλισης, επικοινωνίας, υπολογισμού είναι υψηλότεροι) σε σχέση με τις υπόλοιπες μεθόδους και ακολουθούν η μέθοδος Gauss - Seidel και Red Black. Η μεγάλη διαφορά στους χρόνους, κυρίως ανάμεσα στη μέθοδο Jacobi και τις άλλες δύο, φαίνεται να οφείλεται κατά κύριο λόγο στον αριθμό των επαναλήψεων, μέχρι να συγκλίνουν οι τιμές του πίνακα.

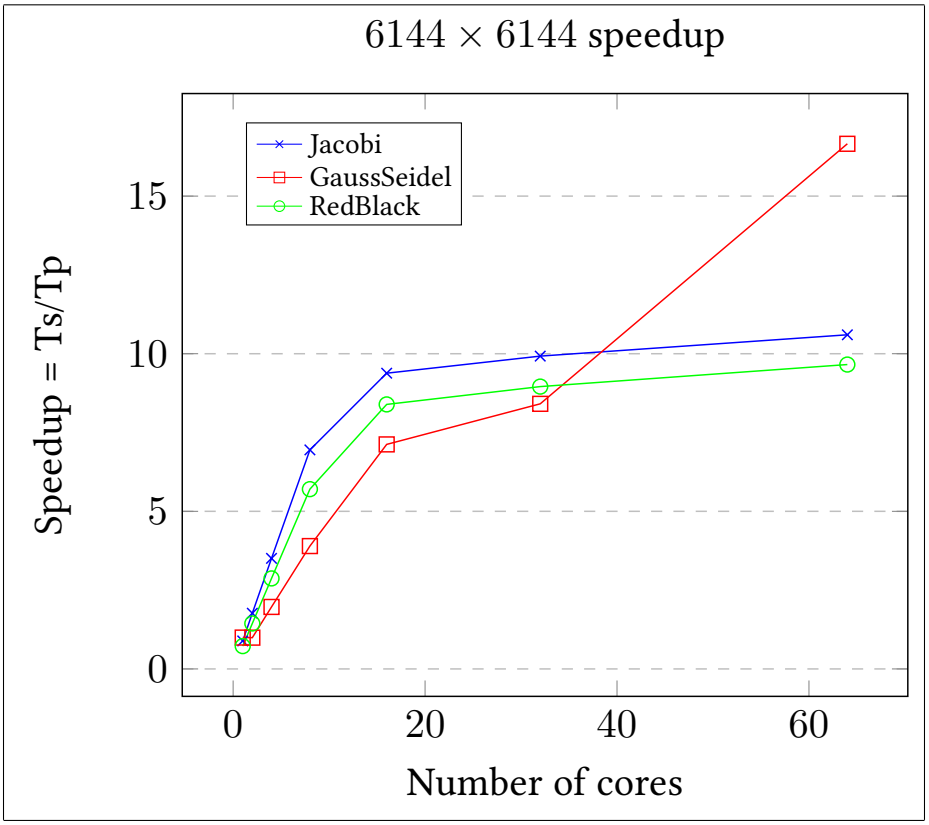
Η καλύτερη μέθοδος, για ένα σύστημα κατανεμημένης μνήμης, με είσοδο κοντά στο μέγεθος της εκφώνησης και 64 διεργασίες, είναι η μέθοδος Red Black. Είναι η πιο γρήγορη στον υπολογισμό των δεδομένων και έχει τον μικρότερο χρόνο επικοινωνίας.

Μετρήσεις χωρίς έλεγχο σύγκλισης

Οι μετρήσεις για αυτό το ερώτημα παρουσιάζονται αναλυτικά στο τέλος της ενότητας. Τα ζητούμενα διαγράμματα επιτάχυνσης παραθέτονται στη συνέχεια:



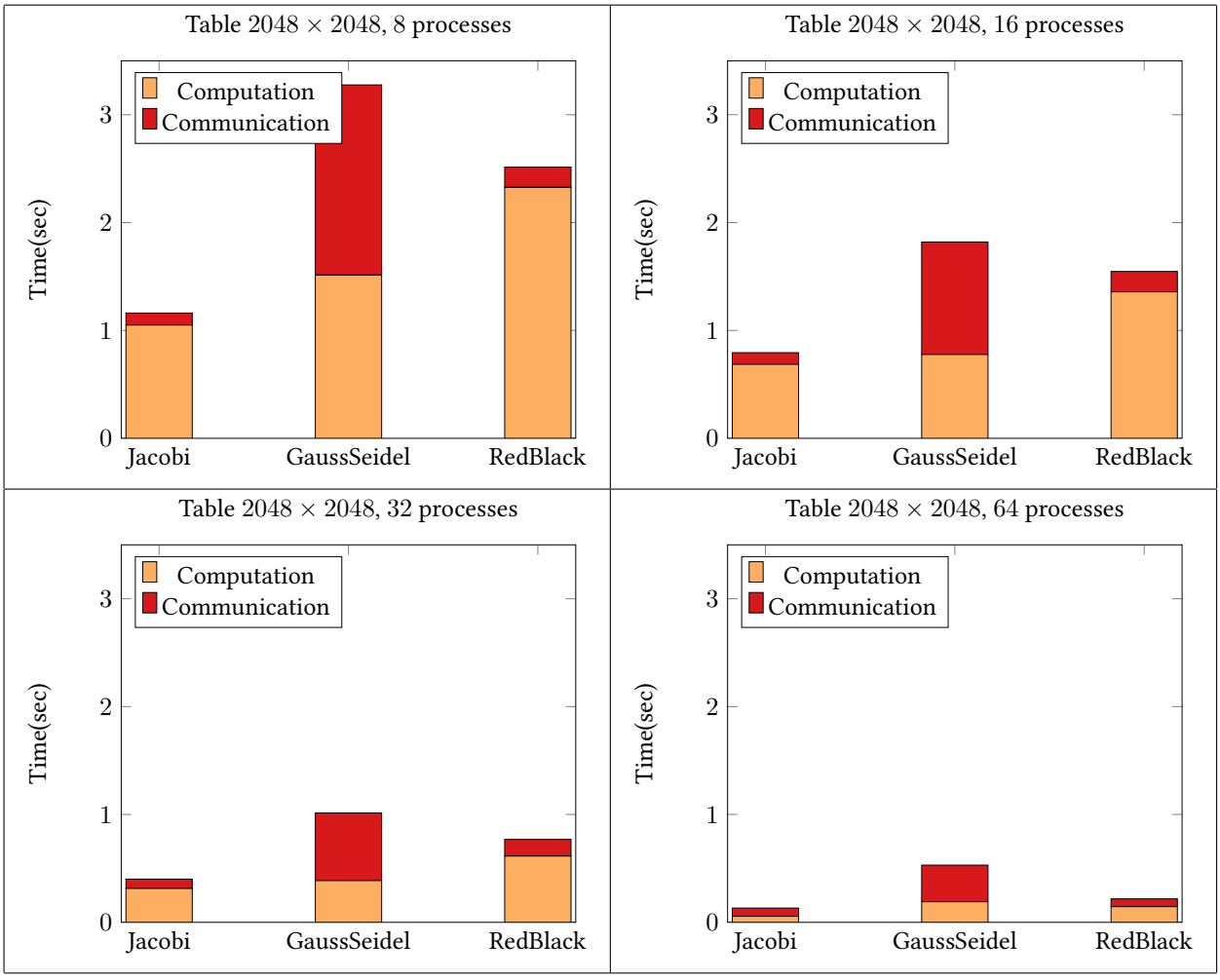

4096 × 4096 speedup
Jacobi
GaussSeidel
RedBlack
Speedup = Ts/Tp
0
5
10
15
0
20
40
60
Number of cores


6144 × 6144 speedup
Jacobi
GaussSeidel
RedBlack
Speedup = Ts/Tp
0
5
10
15
0
20
40
60
Number of cores

Παρατηρήσεις

- Τα διαγράμματα των μεθόδων Jacobi και Red Black παρουσιάζουν περίπου την ίδια επιτάχυνση για όλα τα μεγέθη πινάκων. Για τον πίνακα 2048×2048 κλιμακώνουν σχεδόν γραμμικά, όμως για τα μεγέθη 4096×4096 και 6144×6144 δεν υπάρχει σημαντική αύξηση της επιτάχυνσης για πλήθος επεξεργαστών μεγαλύτερο από 16. Σύμφωνα με τις παραπάνω μετρήσεις, αυτό οφείλεται κυρίως στον χρόνο υπολογισμού και όχι στο χρόνο επικοινωνίας όπως θα περιμέναμε.
- Αντίθετα η μέθοδος Gauss - Seidel ενώ παρουσιάζει μικρότερη επιτάχυνση από τις άλλες δύο για μέγεθος 2048×2048 φαίνεται να μη χειροτερεύει όσο οι άλλες για μεγαλύτερου μεγέθους πίνακες και περισσότερους από 16 πυρήνες (δεν παρατηρείται κορεσμός).
- Από τους πίνακες με τις αναλυτικές μετρήσεις, παρατηρείται επίσης, ότι ο κύριος λόγος, για τον οποίο δεν κλιμακώνουν καλά οι μέθοδοι Jacobi και Red Black στους μεγάλους πίνακες, για περισσότερους από 16 πυρήνες είναι οι χρόνοι των υπολογισμών, καθώς δεν υποδιπλασιάζονται κάθε φορά που διπλασιάζεται το πλήθος των διεργασιών. Ο χρόνος επικοινωνίας επίσης σε αυτές τις δύο μεθόδους δεν αποτελεί μεγάλο ποσοστό του συνολικού χρόνου, όπως φαίνεται και στα επόμενα διαγράμματα.

Στη συνέχεια παρουσιάζονται τα ζητούμενα διαγράμματα με μπάρες:



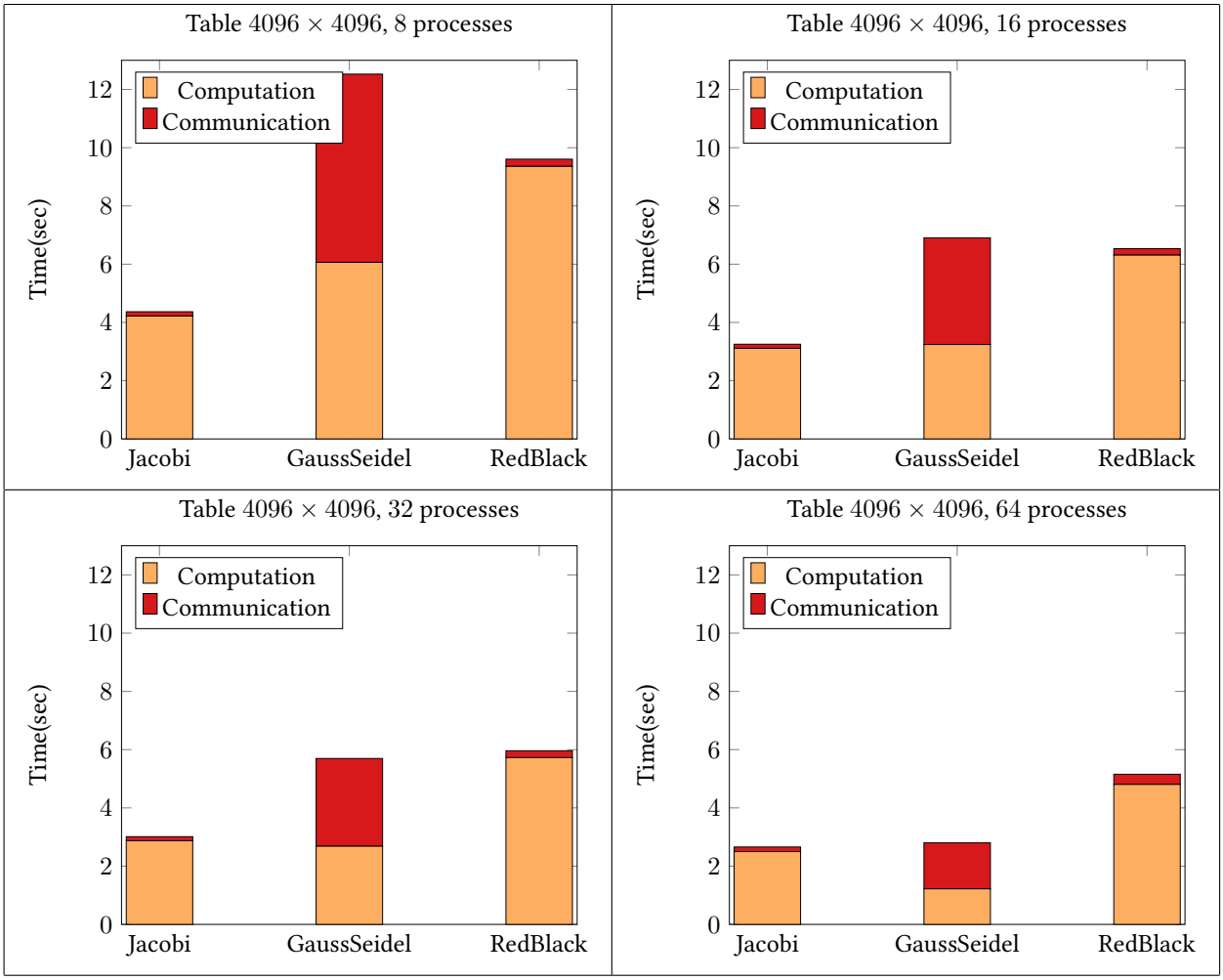
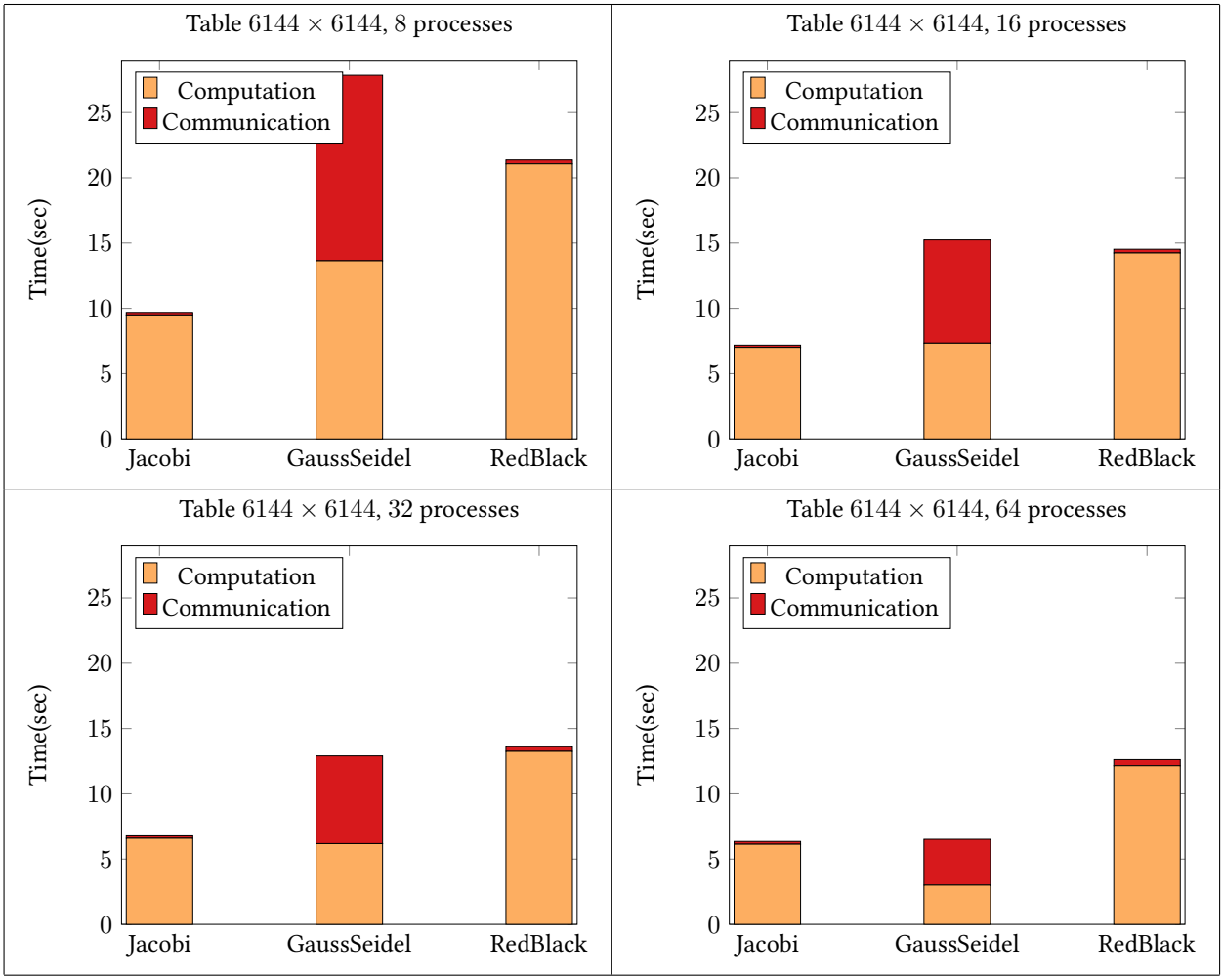


Table 4096 × 4096, 8 processes
Table 4096 × 4096, 16 processes
Table 4096 × 4096, 32 processes
Table 4096 × 4096, 64 processes
Computation
Communication
Time(sec)
0
2
4
6
8
10
12
Jacobi
GaussSeidel
RedBlack



Παρατηρήσεις

- Αρχικά επισημαίνεται ότι για όλα τα μεγέθη του πίνακα με την αύξηση του πλήθους των διεργασιών, βελτιώνεται ο χρόνος εκτέλεσης του προγράμματος. Αυτό είναι αναμενόμενο, καθώς το πρόγραμμα παραλληλοποιείται σε μεγαλύτερο βαθμό (δηλαδή μικρότερα τμήματα υπολογισμών διανέμονται σε περισσότερες διεργασίες).

- Τον μεγαλύτερο χρόνο επικοινωνίας παρουσιάζει η μέθοδος Gauss – Seidel. Στη συγκεκριμένη μέθοδο, σε κάθε επανάληψη απαιτείται η ανταλλαγή στοιχείων μεταξύ διεργασιών πριν και μετά τον υπολογισμό του τοπικού πίνακα. Αυτό συνεπάγεται ότι οι διεργασίες είναι αναγκαίο να συγχρονίζονται σε δύο σημεία του προγράμματος, το οποίο μπορεί να δικαιολογήσει την αύξηση του χρόνου επικοινωνίας. Σημειώνεται ότι ενώ και στη μέθοδο Red Black ανταλλάσσονται στοιχεία δύο φορές σε κάθε επανάληψη, δεν ανταλλάσσεται το ίδιο πλήθος (μόνο οι μισές τιμές , στις άρτιες ή περιττές θέσεις αντίστοιχα), επομένως η καθυστέρηση του συγχρονισμού είναι μικρότερη.

- Επίσης παρατηρείται ότι ο χρόνος υπολογισμού της μεθόδου Red Black είναι περίπου διπλάσιος από αυτόν της μεθόδου Jacobi. Στη Red Black μέθοδο σε κάθε επανάληψη γίνεται δύο φορές υπολογισμός των στοιχείων του πίνακα (μια φορά ανανεώνονται οι τιμές των άρτιων θέσεων και μια φορά των περιττών θέσεων). Ο χρόνος επικοινωνίας είναι επίσης περίπου διπλάσιος, το οποίο δεν είναι αναμενόμενο, διότι χρησιμοποιούνται ειδικά datatypes (στη μέθοδο Red Black),

με τα οποία δεν ανταλλάσσονται περιττά δεδομένα (συνολικά ανταλλάσσεται ίδιος αριθμός στοιχείων σε κάθε επανάληψη).

- Από τα παραπάνω διαγράμματα διαπιστώνεται ότι η μέθοδος Gauss – Seidel, ενώ για μέγεθος πίνακα 2048×2048 έχει τον μεγαλύτερο χρόνο εκτέλεσης, δε συμβαίνει το ίδιο για τις άλλες δύο διαστάσεις για 64 διεργασίες. Συγκεκριμένα ο χρόνος υπολογισμού των πινάκων φαίνεται να βελτιώνεται συγκριτικά με τις άλλες δύο μεθόδους, το οποίο μπορεί να οφείλεται στην εκμετάλλευση της τοπικότητας των δεδομένων. Ωστόσο σε όλες τις περιπτώσεις η μέθοδος Gauss - Seidel είναι αυτή με τον μεγαλύτερο χρόνο επικοινωνίας.

- Τέλος παρατηρείται ότι για τη μέθοδο Red Black, για μεγέθη πίνακα 4096×4096 και 6144×6144, η αύξηση του πλήθους των διεργασιών από 16 σε 32 και 64 δεν προκαλεί σημαντική βελτίωση στον ολικό χρόνο εκτέλεσης. Συγκεκριμένα ενώ ο χρόνος επικοινωνίας μεταξύ διεργασιών είναι μικρός, ο αντίστοιχος χρόνος υπολογισμού είναι αρκετά υψηλός, το οποίο μπορεί να οφείλεται στο σχεδιασμό του αλγορίθμου, καθώς ίσως και σε πιθανά misses στην cache (δεδομένου ότι για τον υπολογισμό μιας τιμής σε άρτια θέση απαιτούνται στοιχεία, τα οποία βρίσκονται σε περιττές θέσεις, η μεταφορά τους μπορεί να οδηγήσει σε εκτοπισμό δεδομένων προς μελλοντική χρήση).

Μετρήσεις

Παρακάτω φαίνονται όλες οι μετρήσεις αναλυτικά. Είναι όλες για 256 επαναλήψεις, χωρίς έλεγχο για σύγκλιση και ο χρόνος επικοινωνίας έχει προκύψει από την αφαίρεση των υπολογισμών από τον συνολικό χρόνο.

Jacobi 2048									
	Computation				**Total**				Commun.
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	8.425	8.421	8.420	**8.422**	8.425	8.421	8.420	**8.422**	**0.000**
2	4.220	4.213	4.217	**4.216**	4.321	4.318	4.320	**4.320**	**0.103**
4	2.108	2.110	2.109	**2.109**	2.223	2.224	2.223	**2.223**	**0.114**
8	1.050	1.048	1.050	**1.049**	1.160	1.161	1.162	**1.161**	**0.111**
16	0.686	0.685	0.685	**0.685**	0.794	0.794	0.793	**0.793**	**0.108**
32	0.301	0.341	0.300	**0.314**	0.396	0.410	0.394	**0.400**	**0.086**
64	0.056	0.057	0.054	**0.055**	0.131	0.132	0.132	**0.132**	**0.076**
GaussSeidel 2048									
	Computation				**Total**				Commun.
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	12.107	12.108	12.107	**12.107**	12.107	12.108	12.107	**12.108**	**0.000**
2	6.055	6.055	6.056	**6.056**	12.315	12.314	12.314	**12.314**	**6.258**
4	3.032	3.032	3.032	**3.032**	6.318	6.316	6.334	**6.323**	**3.290**
8	1.513	1.513	1.513	**1.513**	3.276	3.277	3.276	**3.276**	**1.763**
16	0.777	0.776	0.776	**0.777**	1.840	1.80	1.811	**1.820**	**1.043**
32	0.385	0.387	0.387	**0.386**	1.014	1.005	1.022	**1.014**	**0.627**
64	0.191	0.190	0.190	**0.190**	0.529	0.530	0.531	**0.530**	**0.339**
RedBlack 2048									
	Computation				**Total**				Commun.
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	18.665	18.678	18.664	**18.669**	18.665	18.679	18.665	**18.670**	**0.000**

2	9.344	9.351	9.345	**9.347**	9.514	9.535	9.530	**9.526**	**0.179**
4	4.653	4.655	4.657	**4.655**	4.856	4.858	4.868	**4.861**	**0.205**
8	2.326	2.326	2.327	**2.326**	2.512	2.516	2.513	**2.514**	**0.187**
16	1.356	1.358	1.358	**1.358**	1.538	1.530	1.571	**1.546**	**0.188**
32	0.620	0.612	0.608	**0.614**	0.766	0.755	0.783	**0.768**	**0.154**
64	0.144	0.142	0.149	**0.145**	0.217	0.217	0.217	**0.217**	**0.072**
Jacobi 4096									
	Computation				**Total**				Commun.
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	33.637	33.708	33.637	**33.661**	33.637	33.708	33.638	**33.661**	**0.000**
2	16.826	16.838	16.831	**16.832**	16.973	16.978	16.986	**16.979**	**0.147**
4	8.440	8.433	8.439	**8.437**	8.613	8.593	8.593	**8.600**	**0.162**
8	4.222	4.219	4.219	**4.220**	4.367	4.365	4.369	**4.367**	**0.147**
16	3.112	3.110	3.110	**3.111**	3.255	3.251	3.250	**3.252**	**0.141**
32	2.876	2.864	2.870	**2.872**	3.012	3.012	3.012	**3.012**	**0.139**
64	2.493	2.505	2.499	**2.499**	2.658	2.657	2.657	**2.657**	**0.158**
GaussSeidel 4096									
	Computation				**Total**				**Commun.**
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	48.422	48.432	48.442	**48.432**	48.422	48.433	48.443	**48.433**	**0.000**
2	24.220	24.225	24.233	**24.226**	48.712	48.725	48.736	**48.724**	**24.498**
4	12.119	12.1231	12.124	**12.122**	24.631	24.639	24.630	**24.633**	**12.511**
8	6.064	6.062	6.064	**6.064**	12.525	12.518	12.532	**12.525**	**6.461**
16	3.245	3.242	3.241	**3.243**	6.918	6.899	6.900	**6.906**	**3.662**
32	2.725	2.593	2.732	**2.683**	5.780	5.524	5.788	**5.697**	**3.013**
64	1.222	1.218	1.222	**1.221**	2.796	2.814	2.789	**2.799**	**1.578**
RedBlack 4096									
	Computation				**Total**				**Commun.**
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	74.521	74.539	74.517	**74.526**	74.522	74.540	74.518	**74.527**	**0.000**
2	37.316	37.302	37.309	**37.309**	37.541	37.530	37.549	**37.540**	**0.231**
4	18.684	18.689	18.700	**18.691**	18.970	18.985	18.977	**18.977**	**0.286**
8	9.356	9.359	9.359	**9.358**	9.598	9.601	9.614	**9.605**	**0.246**
16	6.310	6.312	6.311	**6.311**	6.535	6.526	6.536	**6.532**	**0.221**
32	5.727	5.708	5.742	**5.726**	5.961	5.957	5.968	**5.962**	**0.236**
64	4.768	4.822	4.829	**4.806**	5.166	5.160	5.137	**5.155**	**0.348**
Jacobi 6144									
	Computation				**Total**				**Commun.**
#Proc	1st	2nd	3rd	**Average**	1st	2nd	3rd	**Average**	
1	75.763	75.800	75.798	**75.787**	75.763	75.801	75.799	**75.788**	**0.000**
2	37.916	37.901	37.905	**37.907**	38.106	38.076	38.107	**38.096**	**0.189**
4	18.977	18.996	18.966	**18.980**	19.193	19.202	19.184	**19.193**	**0.213**
8	9.498	9.487	9.498	**9.494**	9.695	9.675	9.691	**9.687**	**0.193**
16	7.000	7.003	7.009	**7.004**	7.177	7.167	7.179	**7.175**	**0.170**
32	6.590	6.579	6.618	**6.596**	6.775	6.783	6.790	**6.783**	**0.187**
64	6.147	6.136	6.128	**6.137**	6.351	6.350	6.349	**6.350**	**0.212**

GaussSeidel 6144									
	Computation				Total				Commun.
#Proc	1st	2nd	3rd	Average	1st	2nd	3rd	Average	
1	108.967	108.973	108.960	**108.967**	108.968	108.973	109.430	**109.124**	**0.156**
2	54.488	54.492	54.497	**54.492**	109.425	109.435	109.430	**109.430**	**54.937**
4	27.263	27.261	27.266	**27.263**	55.197	55.190	55.194	**55.194**	**27.930**
8	13.639	13.639	13.639	**13.639**	27.847	27.861	27.842	**27.850**	**14.210**
16	7.329	7.323	7.329	**7.327**	15.242	15.230	15.242	**15.238**	**7.911**
32	6.281	6.065	6.204	**6.183**	13.025	12.757	12.936	**12.906**	**6.722**
64	2.980	2.996	3.044	**3.007**	6.513	6.521	6.513	**6.516**	**3.508**
RedBlack 6144									
	Computation				Total				Commun.
#Proc	1st	2nd	3rd	Average	1st	2nd	3rd	Average	
1	168.476	168.475	168.463	**168.471**	168.477	168.476	168.464	**168.472**	**0.000**
2	84.264	84.220	84.306	**84.263**	84.523	84.495	84.539	**84.519**	**0.255**
4	42.070	42.087	42.088	**42.081**	42.444	42.434	42.467	**42.448**	**0.366**
8	21.088	21.054	21.061	**21.068**	21.387	21.362	21.357	**21.369**	**0.300**
16	14.232	14.245	14.238	**14.238**	14.514	14.538	14.517	**14.523**	**0.284**
32	13.221	13.279	13.259	**13.253**	13.609	13.584	13.627	**13.607**	**0.354**
64	12.161	12.111	12.178	**12.150**	12.619	12.627	12.617	**12.621**	**0.471**

Κώδικας

jacobi_mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>
#include "mpi.h"
#include "utils.h"

inline void Jacobi(double ** u_previous, double ** u_current, int
   X_min, int X_max, int Y_min, int Y_max) {
    int i, j;
    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;
}

int main(int argc, char ** argv) {
    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;
#ifdef TEST_CONV
int global_converged=0, converged=0; //flags for convergence,
   global and per process
struct timeval cts, ctf;   //Timer: converge-cts,ctf
#endif
MPI_Datatype dummy;     //dummy datatype used to align user-
   defined datatypes in memory

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

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

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\n");
    exit(-1);
}
else {
    global[0]=atoi(argv[1]);
    global[1]=atoi(argv[2]);
    grid[0]=atoi(argv[3]);
    grid[1]=atoi(argv[4]);
}

//----Create 2D-cartesian communicator----//
//----Usage of the cartesian communicator is optional----//

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

//----Compute local 2D-subdomain dimensions----//
//----Test if the 2D-domain can be equally distributed to all
   processes----//
//----If not, pad 2D-domain----//

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];
    }
}

//----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----//

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

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

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

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

double *addr=malloc(1);

if (rank==0) {
    free(addr);
    addr = &(U[0][0]);
}

//----Rank 0 scatters the global matrix----//
MPI_Scatterv(addr, scattercounts, scatteroffset, global_block,
               &u_current[1][1], 1, local_block, 0,
                  MPI_COMM_WORLD);

/*Make sure u_current and u_previous are
    both initialized*/
memcpy(u_previous[0], u_current[0], (local[0]+2)*(local[1]+2)*
   sizeof(double));

if (rank==0)
    free2d(U,global_padded[0],global_padded[1]);

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

```
//----Find the 4 neighbors with which a process exchanges
   messages----//
int north, south, east, west;
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);

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

//---Define the iteration ranges per process-----//
int i_min, i_max, j_min, j_max;

i_min = (north < 0) ? 2 : 1;
j_min = (west < 0) ? 2 : 1;

if ((rank_grid[0] + 1) * local[0] >= global[0]) {
    i = global[0] - rank_grid[0] * local[0];
    i_max = i > 0 ? i : 1;
}
else {
    i_max = local[0] + 1;
}

if ((rank_grid[1] + 1) * local[1] >= global[1]) {
    i = global[1] - rank_grid[1] * local[1];
    j_max = i > 0 ? i : 1;
}
else {
    j_max = local[1] + 1;
}

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

//----Computational core----//
MPI_Request s_request[4], r_request[4];
MPI_Status s_status[4], r_status[4];
int req;

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

```
#undef T
#define T 256
for (t=0;t<T;t++) {
#endif
    swap=u_previous;
    u_previous=u_current;
    u_current=swap;

    //----Receive boundaries to and from neighbors----//
    req=0;
    if (north >= 0) {
        MPI_Irecv(&u_previous[0][1], j_max, MPI_DOUBLE, north, t
            , MPI_COMM_WORLD, &r_request[req]);
        req++;
    }
    if (west >= 0) {
        MPI_Irecv(&u_previous[1][0], 1, vertical_border, west, t
            , MPI_COMM_WORLD, &r_request[req]);
        req++;
    }
    if (south >= 0) {
        MPI_Irecv(&u_previous[ local[0]+1 ][1], j_max,
            MPI_DOUBLE, south, t, MPI_COMM_WORLD, &r_request[req
            ]);
        req++;
    }
    if (east >= 0) {
        MPI_Irecv(&u_previous[1][ local[1]+1 ], 1,
            vertical_border, east, t, MPI_COMM_WORLD, &r_request[
            req]);
        req++;
    }

    //----Send boundaries to neighbors----//
    req = 0;
    if (north >= 0) {
        MPI_Isend(&u_previous[1][1], j_max, MPI_DOUBLE, north, t
            , MPI_COMM_WORLD, &s_request[req]);
        req++;
    }
    if (west >= 0) {
        MPI_Isend(&u_previous[1][1], 1, vertical_border, west, t
            , MPI_COMM_WORLD, &s_request[req]);
        req++;
    }
    if (south >= 0) {
        MPI_Isend(&u_previous[ local[0] ][1], j_max, MPI_DOUBLE,
             south, t, MPI_COMM_WORLD, &s_request[req]);
        req++;
```

```
        }
        if (east >= 0) {
            MPI_Isend(&u_previous[1][ local[1] ], 1, vertical_border
                , east, t, MPI_COMM_WORLD, &s_request[req]);
            req++;
        }

        MPI_Waitall(req, r_request, r_status);
        MPI_Waitall(req, s_request, s_status);

        gettimeofday(&tcs,NULL);
        Jacobi(u_previous, u_current, i_min, i_max, j_min, j_max);
        gettimeofday(&tcf,NULL);
        tcomp = tcomp + (tcf.tv_sec-tcs.tv_sec) + (tcf.tv_usec-tcs.
            tv_usec)*0.000001;

        #ifdef TEST_CONV
        gettimeofday(&cts, NULL);
        if (t%C==0) {
            /*Test convergence*/
            converged = converge(u_previous, u_current, i_max, j_max
                );
            MPI_Allreduce(&converged, &global_converged, 1, MPI_INT,
                MPI_LAND, MPI_COMM_WORLD);
        }
        gettimeofday(&ctf, NULL);
        tconv = tconv + (ctf.tv_sec-cts.tv_sec) + (ctf.tv_usec-cts.
            tv_usec)*0.000001;
        #endif
    }
    MPI_Barrier(MPI_COMM_WORLD);
    gettimeofday(&ttf,NULL);

    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)
        *0.000001;

    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
    MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);

    //----Rank 0 gathers local matrices back to the global matrix
        ----//
    if (rank==0) {
            U=allocate2d(global_padded[0],global_padded[1]);
    }
```

```
    if (rank==0) {
        addr = &(U[0][0]);
    }

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

    //----Printing results----//
    if (rank==0) {
        printf("Jacobi␣X␣%d␣Y␣%d␣Px␣%d␣Py␣%d␣Iter␣%d␣ComputationTime
           ␣%lf␣ConvergeTime␣%lf␣TotalTime␣%lf␣midpoint␣%lf\n",
           global[0],
            global[1], grid[0], grid[1], t-1, comp_time, conv_time,
               total_time, U[global[0]/2][global[1]/2]);

        #ifdef PRINT_RESULTS
        char * s=malloc(50*sizeof(char));
        sprintf(s,"resJacobiMPI_%dx%d_%dx%d",global[0],global[1],
           grid[0],grid[1]);
        fprint2d(s,U,global[0],global[1]);
        free(s);
        #endif

    }
    MPI_Finalize();
    return 0;
}
```

gaussSeidel_mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>
#include "mpi.h"
#include "utils.h"

int main(int argc, char ** argv) {
    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;
    #ifdef TEST_CONV
    int global_converged=0, converged=0; //flags for convergence,
       global and per process
```

```
    struct timeval cts, ctf;   //Timer: converge-cts,ctf
    #endif
    MPI_Datatype dummy;     //dummy datatype used to align user-
       defined datatypes in memory
    double omega;           //relaxation factor

    struct timeval tts, ttf, tcs, tcf;   //Timers: total-tts,ttf,
       computation-tcs,tcf, converge-cts,ctf
    double ttotal=0, tcomp=0, tconv=0, total_time, comp_time,
       conv_time;

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

    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\n");
        exit(-1);
    }
    else {
        global[0]=atoi(argv[1]);
        global[1]=atoi(argv[2]);
        grid[0]=atoi(argv[3]);
        grid[1]=atoi(argv[4]);
    }

    //----Create 2D-cartesian communicator----//
    //----Usage of the cartesian communicator is optional----//

    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

    //----Compute local 2D-subdomain dimensions----//
```

```
//----Test if the 2D-domain can be equally distributed to all
   processes----//
//----If not, pad 2D-domain----//

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];
    }
}

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

//----Datatype definition for the 2D-subdomain on the local
   matrix----//
MPI_Datatype local_block;
```

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

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

double *addr=malloc(1);

if (rank==0) {
    free(addr);
    addr = &(U[0][0]);
}

//----Rank 0 scatters the global matrix----//
MPI_Scatterv(addr, scattercounts, scatteroffset, global_block,
                &u_current[1][1], 1, local_block, 0,
                   MPI_COMM_WORLD);

/*Make sure u_current and u_previous are
    both initialized*/
memcpy(u_previous[0], u_current[0], (local[0]+2)*(local[1]+2)*
   sizeof(double));

if (rank==0)
    free2d(U,global_padded[0],global_padded[1]);

//----Define datatypes or allocate buffers for message passing
   ----//
MPI_Datatype vertical_border;
MPI_Type_vector(local[0],1,local[1]+2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&vertical_border)
   ;
MPI_Type_commit(&vertical_border);

//----Find the 4 neighbors with which a process exchanges
   messages----//
int north, south, east, west;
```

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

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

//---Define the iteration ranges per process-----//
int i_min, i_max, j_min, j_max;

i_min = (north < 0) ? 2 : 1;
j_min = (west < 0) ? 2 : 1;

if ((rank_grid[0] + 1) * local[0] >= global[0]) {
    i = global[0] - rank_grid[0] * local[0];
    i_max = i > 0 ? i : 1;
}
else {
    i_max = local[0] + 1;
}

if ((rank_grid[1] + 1) * local[1] >= global[1]) {
    i = global[1] - rank_grid[1] * local[1];
    j_max = i > 0 ? i : 1;
}
else {
    j_max = local[1] + 1;
}

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

//----Computational core----//
MPI_Request s_request[4], r_request[4];
MPI_Status s_status[4], r_status[4];
int req, req2;

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

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

        //----Receive boundaries from west and north neighbors----//
        req=0;
        if (north >= 0) {
            MPI_Irecv(&u_current[0][1], j_max, MPI_DOUBLE, north, t,
                MPI_COMM_WORLD, &r_request[req]);
            req++;
        }
        if (west >= 0) {
            MPI_Irecv(&u_current[1][0], 1, vertical_border, west, t,
                MPI_COMM_WORLD, &r_request[req]);
            req++;
        }

        MPI_Waitall(req, r_request, r_status);

        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]+(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;
            }
        gettimeofday(&tcf,NULL);
        tcomp = tcomp + (tcf.tv_sec-tcs.tv_sec) + (tcf.tv_usec-tcs.
           tv_usec)*0.000001;

        //----Receive boundaries from south and east neighbors----//
        req2=0;
        if (south >= 0) {
            MPI_Irecv(&u_current[ local[0]+1 ][1], j_max, MPI_DOUBLE
                , south, t, MPI_COMM_WORLD, &r_request[req2]);
            req2++;
        }
        if (east >= 0) {
            MPI_Irecv(&u_current[1][ local[1]+1 ], 1,
                vertical_border, east, t, MPI_COMM_WORLD, &r_request[
                req2]);
            req2++;
        }

        //----Send boundaries to neighbors----//
        req=0;
        if (north >= 0) {
```

```
            MPI_Isend(&u_current[1][1], j_max, MPI_DOUBLE, north, t,
                MPI_COMM_WORLD, &s_request[req]);
            req++;
        }
        if (west >= 0) {
            MPI_Isend(&u_current[1][1], 1, vertical_border, west, t,
                MPI_COMM_WORLD, &s_request[req]);
            req++;
        }

        if (south >= 0) {
            MPI_Isend(&u_current[ local[0] ][1], j_max, MPI_DOUBLE,
               south, t, MPI_COMM_WORLD, &s_request[req]);
            req++;
        }
        if (east >= 0) {
            MPI_Isend(&u_current[1][ local[1] ], 1, vertical_border,
                east, t, MPI_COMM_WORLD, &s_request[req]);
            req++;
        }

        MPI_Waitall(req2, r_request, r_status);
        MPI_Waitall(req, s_request, s_status);

        #ifdef TEST_CONV
        gettimeofday(&cts, NULL);
        if (t%C==0) {
            /*Test convergence*/
            converged = converge(u_previous, u_current, i_max, j_max
               );
            MPI_Allreduce(&converged, &global_converged, 1, MPI_INT,
                MPI_LAND, MPI_COMM_WORLD);
        }
        gettimeofday(&ctf, NULL);
        tconv = tconv + (ctf.tv_sec-cts.tv_sec) + (ctf.tv_usec-cts.
           tv_usec)*0.000001;
        #endif
    }
    MPI_Barrier(MPI_COMM_WORLD);
    gettimeofday(&ttf,NULL);

    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)
        *0.000001;

    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
```

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

    //----Rank 0 gathers local matrices back to the global matrix
       ----//
    if (rank==0) {
            U=allocate2d(global_padded[0],global_padded[1]);
    }

    if (rank==0) {
        addr = &(U[0][0]);
    }

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

    //----Printing results----//
    if (rank==0) {
        printf("GaussSeidel X %d Y %d Px %d Py %d Iter %d
           ComputationTime %lf ConvergeTime %lf TotalTime %lf
           midpoint %lf\n", global[0],
            global[1], grid[0], grid[1], t-1, comp_time, conv_time,
               total_time, U[global[0]/2][global[1]/2]);
        #ifdef PRINT_RESULTS
        char * s=malloc(50*sizeof(char));
        sprintf(s,"resGaussSeidelMPI_%dx%d_%dx%d",global[0],global
           [1],grid[0],grid[1]);
        fprint2d(s,U,global[0],global[1]);
        free(s);
        #endif

    }
    MPI_Finalize();
    return 0;
}
```

redBlack_mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>
#include "mpi.h"
#include "utils.h"

inline void RedSOR(double ** u_previous, double ** u_current, int
   X_min, int X_max, int Y_min, int Y_max, double omega, int fix) {
    int i, j;
```

```
    for (i=X_min; i<X_max; i++)
        for (j=Y_min; j<Y_max; j++)
            if ((i+j)%2 == fix)
                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]);
}

inline void BlackSOR(double ** u_previous, double ** u_current, int
   X_min, int X_max, int Y_min, int Y_max, double omega, int fix) {
    int i, j;
    for (i=X_min; i<X_max; i++)
        for (j=Y_min; j<Y_max; j++)
            if ((i+j)%2 == fix)
                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]);
}

int main(int argc, char ** argv) {
    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;
    #ifdef TEST_CONV
    int global_converged=0, converged=0; //flags for convergence,
       global and per process
    struct timeval cts, ctf;   //Timer: converge-cts,ctf
    #endif
    MPI_Datatype dummy;     //dummy datatype used to align user-
       defined datatypes in memory
    double omega;           //relaxation factor

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

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

    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\n");
        exit(-1);
    }
    else {
        global[0]=atoi(argv[1]);
        global[1]=atoi(argv[2]);
        grid[0]=atoi(argv[3]);
        grid[1]=atoi(argv[4]);
    }

    //----Create 2D-cartesian communicator----//
    //----Usage of the cartesian communicator is optional----//

    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

    //----Compute local 2D-subdomain dimensions----//
    //----Test if the 2D-domain can be equally distributed to all
       processes----//
    //----If not, pad 2D-domain----//

    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];
        }
    }

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

//----Datatype definition for the 2D-subdomain on the local
   matrix----//
MPI_Datatype local_block;
MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
MPI_Type_commit(&local_block);

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

```
double *addr=malloc(1);

if (rank==0) {
    free(addr);
    addr = &(U[0][0]);
}

//----Rank 0 scatters the global matrix----//
MPI_Scatterv(addr, scattercounts, scatteroffset, global_block,
                &u_current[1][1], 1, local_block, 0,
                   MPI_COMM_WORLD);

/*Make sure u_current and u_previous are
    both initialized*/
memcpy(u_previous[0], u_current[0], (local[0]+2)*(local[1]+2)*
   sizeof(double));

if (rank==0)
    free2d(U,global_padded[0],global_padded[1]);

//----Define datatypes or allocate buffers for message passing
   ----//
MPI_Datatype vertical_border;
i = (local[0]%2) ? local[0]/2 + 1 : local[0]/2;
MPI_Type_vector(i,1,2*local[1]+4,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&vertical_border)
   ;
MPI_Type_commit(&vertical_border);

MPI_Datatype horizontal_border;
i = (local[1]%2) ? local[1]/2 + 1 : local[1]/2;
MPI_Type_vector(i,1,2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&
   horizontal_border);
MPI_Type_commit(&horizontal_border);

//----Find the 4 neighbors with which a process exchanges
   messages----//
int north, south, east, west;
MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);

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

//---Define the iteration ranges per process-----//
int i_min,i_max,j_min,j_max;
i_min = (north < 0) ? 2 : 1;
```

```
    j_min = (west < 0) ? 2 : 1;

    if ((rank_grid[0] + 1) * local[0] >= global[0]) {
        i = global[0] - rank_grid[0] * local[0];
        i_max = i > 0 ? i : 1;
    }
    else {
        i_max = local[0] + 1;
    }

    if ((rank_grid[1] + 1) * local[1] >= global[1]) {
        i = global[1] - rank_grid[1] * local[1];
        j_max = i > 0 ? i : 1;
    }
    else {
        j_max = local[1] + 1;
    }

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

    //----Define fix for redblack computation and where data will be
        sent to/received from in red phase----//
    /*These computations are necessary if we want the program to
       give correct results
        when local[0] or local[1] are odd numbers
    */
    int fix_red, fix_black;
    int rec_nw, send_nw, rec_s, send_s, rec_e, send_e;

    fix_red = (rank_grid[0]*local[0] + rank_grid[1]*local[1])%2;
    fix_black = (fix_red+1)%2;
    rec_nw = fix_red+1;
    send_nw = fix_black+1;
    /*Lower left cell will be red(even) if local[0]%2==fix_black,
       black(odd) otherwise*/
    rec_s = (local[0]%2 == fix_black) ? 1 : 2;
    send_s = rec_s%2 + 1;
    /*Upper right cell will be red(even) if local[1]%2==fix_black,
       black(odd) otherwise*/
    rec_e = (local[1]%2 == fix_black) ? 1 : 2;
    send_e = rec_e%2 + 1;

    //----Computational core----//
    MPI_Request s_request[4], r_request[4];
    MPI_Status s_status[4], r_status[4];
```

```
    int req;

    MPI_Barrier(MPI_COMM_WORLD);
    gettimeofday(&tts,NULL);
    #ifdef TEST_CONV
    for (t=0;t<T && !global_converged;t++) {
    #endif
    #ifndef TEST_CONV
    #undef T
    #define T 256
    for (t=0;t<T;t++) {
    #endif
        swap=u_previous;
        u_previous=u_current;
        u_current=swap;

        //----Receive boundaries from neighbors----//
        req=0;
        if (north >= 0) {
            MPI_Irecv(&u_previous[0][rec_nw], 1, horizontal_border,
               north, t, MPI_COMM_WORLD, &r_request[req]);
            req++;
        }
        if (west >= 0) {
            MPI_Irecv(&u_previous[rec_nw][0], 1, vertical_border,
               west, t, MPI_COMM_WORLD, &r_request[req]);
            req++;
        }
        if (south >= 0) {
            MPI_Irecv(&u_previous[ local[0]+1 ][rec_s], 1,
               horizontal_border, south, t, MPI_COMM_WORLD, &
               r_request[req]);
            req++;
        }
        if (east >= 0) {
            MPI_Irecv(&u_previous[rec_e][ local[1]+1 ], 1,
               vertical_border, east, t, MPI_COMM_WORLD, &r_request[
               req]);
            req++;
        }

        //----Send boundaries to neighbors----//
        req=0;
        if (north >= 0) {
            MPI_Isend(&u_previous[1][send_nw], 1, horizontal_border,
                north, t, MPI_COMM_WORLD, &s_request[req]);
            req++;
        }
        if (west >= 0) {
```

```
            MPI_Isend(&u_previous[send_nw][1], 1, vertical_border,
                west, t, MPI_COMM_WORLD, &s_request[req]);
            req++;
        }
        if (south >= 0) {
            MPI_Isend(&u_previous[ local[0] ][send_s], 1,
                horizontal_border, south, t, MPI_COMM_WORLD, &
                s_request[req]);
            req++;
        }
        if (east >= 0) {
            MPI_Isend(&u_previous[send_e][ local[1] ], 1,
                vertical_border, east, t, MPI_COMM_WORLD, &s_request[
                req]);
            req++;
        }

        MPI_Waitall(req, s_request, s_status);
        MPI_Waitall(req, r_request, r_status);

        gettimeofday(&tcs,NULL);
        RedSOR(u_previous, u_current, i_min, i_max, j_min, j_max,
            omega, fix_red);
        gettimeofday(&tcf,NULL);
        tcomp = tcomp + (tcf.tv_sec-tcs.tv_sec) + (tcf.tv_usec-tcs.
            tv_usec)*0.000001;

        /*Send and receive boundaries are swapped in this phase*/
        //----Receive boundaries from neighbors----//
        req=0;
        if (north >= 0) {
            MPI_Irecv(&u_current[0][send_nw], 1, horizontal_border,
                north, t, MPI_COMM_WORLD, &r_request[req]);
            req++;
        }
        if (west >= 0) {
            MPI_Irecv(&u_current[send_nw][0], 1, vertical_border,
                west, t, MPI_COMM_WORLD, &r_request[req]);
            req++;
        }
        if (south >= 0) {
            MPI_Irecv(&u_current[ local[0]+1 ][send_s], 1,
                horizontal_border, south, t, MPI_COMM_WORLD, &
                r_request[req]);
            req++;
        }
        if (east >= 0) {
            MPI_Irecv(&u_current[send_e][ local[1]+1 ], 1,
                vertical_border, east, t, MPI_COMM_WORLD, &r_request[
```

```
            req]);
        req++;
    }

    //----Send boundaries to neighbors----//
    req=0;
    if (north >= 0) {
        MPI_Isend(&u_current[1][rec_nw], 1, horizontal_border,
            north, t, MPI_COMM_WORLD, &s_request[req]);
        req++;
    }
    if (west >= 0) {
        MPI_Isend(&u_current[rec_nw][1], 1, vertical_border,
            west, t, MPI_COMM_WORLD, &s_request[req]);
        req++;
    }
    if (south >= 0) {
        MPI_Isend(&u_current[ local[0] ][rec_s], 1,
            horizontal_border, south, t, MPI_COMM_WORLD, &
            s_request[req]);
        req++;
    }
    if (east >= 0) {
        MPI_Isend(&u_current[rec_e][ local[1] ], 1,
            vertical_border, east, t, MPI_COMM_WORLD, &s_request[
            req]);
        req++;
    }

    MPI_Waitall(req, s_request, s_status);
    MPI_Waitall(req, r_request, r_status);

    gettimeofday(&tcs,NULL);
    BlackSOR(u_previous, u_current, i_min, i_max, j_min, j_max,
        omega, fix_black);
    gettimeofday(&tcf,NULL);
    tcomp = tcomp + (tcf.tv_sec-tcs.tv_sec) + (tcf.tv_usec-tcs.
        tv_usec)*0.000001;

    #ifdef TEST_CONV
    gettimeofday(&cts, NULL);
    if (t%C==0) {
        /*Test convergence*/
        converged = converge(u_previous, u_current, i_max, j_max
            );
        MPI_Allreduce(&converged, &global_converged, 1, MPI_INT,
             MPI_LAND, MPI_COMM_WORLD);
    }
    gettimeofday(&ctf, NULL);
```

```
        tconv = tconv + (ctf.tv_sec-cts.tv_sec) + (ctf.tv_usec-cts.
            tv_usec)*0.000001;
        #endif
    }
    MPI_Barrier(MPI_COMM_WORLD);
    gettimeofday(&ttf,NULL);

    ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)
        *0.000001;

    MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
    MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);
    MPI_Reduce(&tconv,&conv_time,1,MPI_DOUBLE,MPI_MAX,0,
        MPI_COMM_WORLD);

    //----Rank 0 gathers local matrices back to the global matrix
        ----//
    if (rank==0) {
            U=allocate2d(global_padded[0],global_padded[1]);
    }

    if (rank==0) {
        addr = &(U[0][0]);
    }

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

    //----Printing results----//
    if (rank==0) {
        printf("RedBlack␣X␣%d␣Y␣%d␣Px␣%d␣Py␣%d␣Iter␣%d␣
            ComputationTime␣%lf␣ConvergeTime␣%lf␣TotalTime␣%lf␣
            midpoint␣%lf\n", global[0],
             global[1], grid[0], grid[1], t-1, comp_time, conv_time,
                total_time, U[global[0]/2][global[1]/2]);
        #ifdef PRINT_RESULTS
        char * s=malloc(50*sizeof(char));
        sprintf(s,"resRedBlackMPI_%dx%d_%dx%d",global[0],global[1],
            grid[0],grid[1]);
        fprint2d(s,U,global[0],global[1]);
        free(s);
        #endif

    }
    MPI_Finalize();
    return 0;
```

```
}
```

utils.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "utils.h"

double max(double a, double b) {
    return a>b?a:b;
}

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

double ** allocate2d ( int dimX, int dimY ) {
    double ** array, * tmp;
    int i;
    tmp = ( double * )calloc( dimX * dimY, sizeof( double ) );
    array = ( double ** )calloc( dimX, sizeof( double * ) );
    for ( i = 0 ; i < dimX ; i++ )
        array[i] = tmp + i * dimY;
    if ( array == NULL || tmp == NULL) {
        fprintf( stderr,"Error in allocation\n" );
        exit( -1 );
    }
    return array;
}

void free2d( double ** array, int dimX, int dimY) {
    if (array==NULL) {
        fprintf(stderr,"Error in freeing matrix\n");
        exit(-1);
    }
    if (array[0])
        free(array[0]);
    if (array)
        free(array);
}
void init2d ( double ** array, int dimX, int dimY ) {
    int i,j;
    for ( i = 0 ; i < dimX ; i++ )
        for ( j = 0; j < dimY ; j++)
```

```
            array[i][j]=(i==0 || i==dimX-1 || j==0 || j==dimY-1)
               ?0.01*(i+1)+0.001*(j+1):0.0;
}

void zero2d ( double ** array, int dimX, int dimY ) {
    int i,j;
    for ( i = 0 ; i < dimX ; i++ )
        for ( j = 0; j < dimY ; j++)
            array[i][j] = 0.0;
}

void print2d(double ** array, int dimX, int dimY) {
    int i,j;
    for (i=0;i<dimX;i++) {
        for (j=0;j<dimY;j++)
            printf("%lf␣",array[i][j]);
        printf("\n");
    }
}

void fprint2d(char * s, double ** array, int dimX, int dimY) {
    int i,j;
    FILE * f=fopen(s,"w");
    for (i=0;i<dimX;i++) {
        for (j=0;j<dimY;j++)
            fprintf(f,"%lf␣",array[i][j]);
        fprintf(f,"\n");
    }
    fclose(f);
}
```