MPI Parallel Implementation of Jacobi

Edmond Jajaga
South East European University
10 PUBLICATIONS **19** CITATIONS

MPI Parallel Implementation of Jacobi

Edmond Jajaga[1] and Jolanda Kllobocishta[2]

[1]South East European University, eLearning Center
e.jajaga@seeu.edu.mk
[2]State University of Tetova, Department of Mathematics
jolanda.kllobocishta@hotmail.com

Abstract. Parallel computing has become a key technology to efficiently tackle complex scientific and engineering problems. The ability of parallelism of an algorithm provides a useful rationale to recourse to it. Forgotten in years, the Jacobi algorithm has been identified with a high parallelism affinity, which is used today as a preconditioner for multigrid methods. This paper describes the message passing implementation of Karniadakis and Kirby as tested in a 9-node cluster with distributed memory. We outline details about tasks distribution and their mapping to the processes accompanied with a thorough description of communications running in a parallel environment.

Keywords. iterative methods, systems of linear equations, C++ MPI, parallel programming

1 Introduction

There exists a lot of research regarding methods for solving systems of linear equations of the form Ax=b. Many algorithms from different math and computer science researchers are currently in place. Basically they are separated into two groups: direct and iterative methods. From the non practical application of Gaussian elimination as a direct method for solving systems Ax=b, iterative methods were widely studied from the scientists. There are lot of advantages and disadvantages between Gaussian elimination as an example of direct method and an iterative one like Jacobian [10].

The general framework of an iterative process is as simple as this: first, an initial assumption-solution is generated intuitively for the vector-solution $x^{(0)}$. Then, using this assumption the algorithm provides us with a possible solution $x^{(1)}$. Now the role of the solution $x^{(1)}$ becomes the input for the next possible solution. This process goes repeatedly, providing an array of vector-solutions $x^{(0)}$, $x^{(1)}$, $x^{(2)}$…, until we get into a satisfactory solution. Apparently there are a lot of questions which need to be answered: how does the initial assumption will be like? What kind of algorithm should be used? Do my iterative results converge to “the real” solution, and if yes, will they converge as fast as they would be better then Gaussian elimination? Much work has been done in this direction and it has been revealed that for certain algorithms and certain types of matrices of A i.e. if A is symmetric and positively determined or at least not singular, the solution actually will converge.

Some of classical iterative methods include: Jacobian, Gauss-Seidel and Richardson method. In this paper we will focus on the Jacobian method as one of the oldest iterative methods. It is worth mentioning that the most modern method for applications of numerical calculations seems to be method of Crylov subspaces [6].

Because of the high level of parallelization we have addressed the parallel implementation of Jacobi iterations. Currently, there exist some parallel implementations of Jacobi iterations like in C, C++, Fortran77 and Fortran90 [7, 9], CUDA and OpenGL [8]. We have followed the implemented Jacobi iterations given in [1] in a parallel environment through library routines of Message Passing Interface (MPI) of C++ language. MPI is designated for high performance on massive parallel machines and in cluster workstations. The application is implemented and tested in SEEUCluster of the South East European University. This cluster consists of 9 workstations PC computers 1.5Ghz/128MB/20GB operating in Linux Red Hat Enterprise 4.0 operating system, configured with OSCAR 4.2 (Open Source Cluster Application Resources) software. LAM 7.0.6/MPI 2 C++/ROMIO - Indiana University is the parallel environment for programming.

The paper is organized as follows: Section 1 gives the mathematical background of the Jacobi iterative method; the decomposition method and process mapping are treated in Section 2; a thorough description of the implemented parallel algorithm of Jacobi with MPI routines is given in Section 3.

2 Jacobi iterative method

When solving the systems of linear equations of the form:

$$\begin{cases} a_{11}x_1 + \cdots a_{1n}x_n = b_1 \\ \vdots \qquad \ddots \qquad \vdots \\ a_{n1}x_1 + \cdots a_{nn}x_n = b_n \end{cases} \quad \text{or} \quad Ax = b \qquad (*)$$

there exist two kinds of methods [1]:

Direct methods, through which we obtain the solution of the system (*) after a finite and known number of steps and the error of these kind of methods is 0, therefore are also called exact methods.

Iterative methods, which solving the system (*) produce an array of approximate values (vector-solution), which converges in some defined circumstances to the exact solution of the system.

The Jacobi method is one of the oldest iterative methods which are engaged in solving the system (*). In order to describe the method procedure firstly we set the system in more proper form like the following:

$$\begin{cases} x_1 = -\dfrac{\sum_{j=2}^{n} a_{1j}x_j}{a_{11}} + \dfrac{b_1}{a_{11}} \\ \vdots \qquad \ddots \qquad \vdots \\ x_i = -\dfrac{\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j}{a_{ii}} + \dfrac{b_i}{a_{ii}} \\ \vdots \qquad \ddots \qquad \vdots \\ x_n = -\dfrac{\sum_{\substack{j=1 \\ j\neq i}}^{n-1} a_{nj}x_j}{a_{nn}} + \dfrac{b_n}{a_{nn}} \end{cases} \quad \text{or} \quad x^{(k+1)} = Tx^{(k)} + c \qquad (1)$$

Hence, the formula in terms of its elements would look like the following:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}\, x_j^{(k)}\right), \quad i = 1, 2, \dots, n. \qquad (**)$$

Formula (**) meanwhile represents the Jacobian iterations. This method assumes we have all the input values of x in the previous iteration (k). But, usually there are not given all the x values. What we can do is to make an initial prepossession for x and to generate another group of solutions for x from the equation (**), which in fact will represent the input values for the next iteration (k+1). After we have found the group of x values for the previous iteration we continue generating new groups again and again until we arrive at an acceptable solution. These iterations produce an array of approximate values for the "real" solution of the system (*).

With the word "acceptable solution" we will intend that group of solutions, respectively to the approximations, of x with the required accuracy.

If the vector-solution values are getting closer to the expected solution with the growth of iterations, then it is said that it converges to the solution of the system (*). In the majority cases the approximation array results with a good estimation for the values of x i.e. converges. Note that this algorithm is nonfunctional for all matrixes. One of the wide exceptions is any matrix with any 0 in diagonal.

We stress out the following requirements of the Jacobi method, in order for it to be functional:

1. All the members of the main diagonal of the matrix A must be nonzero ($a_{ii}\neq 0$, $\forall i = 1,2,..n$).
2. This method requires a duplicate storage for the vector-solution x. This, because none of the members (elements) of x can be overwritten while all the x-elements are calculated for that iteration i.e. it is needed an array of current solution to manipulate with and another array, we will note it with *xold*, holding values of the previous iteration.
3. Components (elements) of the new iteration vector solution are not dependent of each other, hence can be computed at the same time. This identifies the high potential of the parallelization of Jacobi method.

4. The solution provided by the Jacobi method not always converges. It is ensured only within some specified circumstances, which will be discussed in the next subsection.

3 Parallelism of Jacobi iterations

Referring to the equation (**) it can be observed the high potential of the parallelism of Jacobi. In contrast to Gaus-Seidel method, in which are used x-values of the previous and current iteration when finding the x-values of the next iteration, Jacobi puts borders between iterations; values of the vector-solution x are calculated only from the vector-solution of the previous iteration (noted with xold). Because of this, when it comes to parallelization Gauss-Seidel has certain disadvantages, even that the convergence rate of the Jacobi-type iteration is no better than the convergence rate of the corresponding Gauss-Seidel iteration for any nonnegative matrix A [2].

3.1 Data decomposition

Based on equation (**) we partition the problem into the following sub problems:

D1: sum1 = 0
D2: sum2 = $\sum_{j=2}^{n} a_{ij} x_j$
D3: x1 = (-sum1 - sum2 + b1)/A11
D4: sum1 = a11x1
D5: sum2 = $\sum_{j=3}^{n} a_{ij} x_j$
D6: x1 = (-sum1 - sum2 + b2)/A22
D7: sum1 = a11x1+ a12x2
D8: sum2 = $\sum_{j=4}^{n} a_{ij} x_j$
D9: x1 = (-sum1 - sum2 + b3)/A33
D10: sum1 = $\sum_{j=1}^{3} a_{ij} x_j$
D11: sum2 = $\sum_{j=5}^{n} a_{ij} x_j$
D12: x1 = (-sum1 - sum2 + b4)/A44
$\vdots \qquad \vdots$

If we observe the problem decomposition we will find out that there are independent sub problems like: 1, 2, 4, 5, 7, 8, 10, 11, etc. and those dependent: 3 dependent from 1 and 2, 6 from 4 and 5 etc. This way the process of parallelization looks more feasible and every process can do many of these tasks depending of the mapping topology. The dependency graph of the problem decomposition would look like in Fig. 1.

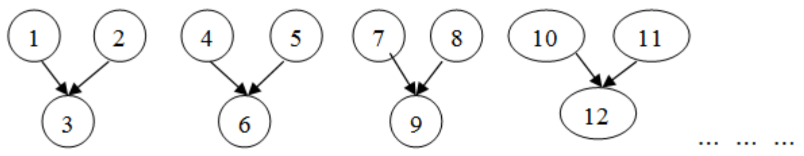


Fig. 1. Dependency graph of the decomposed sub problems

For a more balanced workload, the matrix dimension is divided with the total number of processes which will be initialized in the MPI environment; the obtained result will be the number of rows to be computed by a process, except the last process which will in turn get also the remaining last rows of the residue.

4 MPI implementation

MPI as a message passing library accommodates a natural and easy partitioning of the problem, it provides portability and efficiency, and it has received wide acceptance by academia and industry [1].

Function ReadMatrix serves as input for capturing coefficients of matrix A of system (*), while ReadVector serves for initialization of values for right side vector of (*), namely b. These functions are called from the main function main() and will be executed only by process 0, while at this stage pointers of matrix A, and vectors b and x from all other processes are null.

Within the main function of the application of Jacobi parallel implementation i.e. main() function, the following variables are declared:

- `nbproc` will hold the number of nodes which will do calculation in the parallel environment
- `myrank` indicates the corresponding number of the node (process) of the base communicator COMM_WORLD
- `n` holds the dimension of the square matrix A_{nxn} and also the number indicating the amount of variables of vector solution `x[n]`
- `abstol` provides the permitted error of the required result. We have instantiated it with $\varepsilon = 10^{-2}$ as a random value.

In order to give the user free choice of matrices values there are used dynamic allocation arrays. This rationale does not create close blocks, which means the operating system is more comfortable when deciding where to place things in memory. All the arrays used in our application use this methodology. After the instantiation of the input values the MPI routines for initialization and setting the parallel environment are executed.

The value of n received from the user will be distributed through the MPI routine Bcast to all the initialized processes by the communicator. In order to ensure all the processes received what was dedicated to them it is used Barrier() function.

After the input instantiation the function Jacobi takes the control of running the application. This function implements the Jacobi iterations based on system (**). It receives the number of processes initialized (mynode), the total number of nodes (numnodes), matrices dimension (N), matrix A as double** array, a vector x where will be placed the solution, a vector b holding the right vector of the system (*) and the permitted error (abstol). The main node is considered process 0 which delegates the tasks to other nodes, returns the vector solution and the number of iterations maxit.

As mentioned earlier on Section II. 2., the rows per node mapping are done through `rows_local` variable. It records the number of rows each node is responsible for calculation. For example, if `N=10` and number of nodes is `numnodes=3` than the first two nodes will make calculation within an iteration for 3 rows (since `floor(10/3)=3)`, whereas the last node will do calculations for `rows_local = 10-3*(3-1) = 4` rows.

After specifying the number of rows per node, next comes the distribution of matrices A and b from the equation $Ax = b$.

Distribution of matrices rows are done from node 0, because it holds the whole matrices. It sends the tasks to other processes based on their corresponding rank (node number) through the MPI routine for point-to-point communication, Send, specifically for our example:

- for i=1 a message will be send to *process 1* for j = 0, 1, 2 $\rightarrow$ 3 rows i.e. `A[1*3+(0,1,2)]` or `A[3], A[4]` and `A[5]`
- for i=2 a message will be send to *process 2* for j = 0, 1, 2, 3 $\rightarrow$ 4 rows i.e. `A[2*3+(0,1,2)]` or `A[6], A[7], A[8]` and `A[9]`

Rows `A[0], A[1]` and `A[2]` will be processed from process 0 because they are already on process 0 and there is no need of distribution. Note that the rows distribution of process 2 is done through the last rows distribution code. Likewise to matrix A distribution, the vector b distribution is done.

Normally, before task reception of processes (except process 0), because we are dealing with dynamic allocation arrays, it is required to create their structures in these nodes and then through Recv routine to be received messages dedicated for the corresponding nodes.

Vectors x and b will be of length `rows_local` i.e. every node will process the rows taken over. Specifically for our example, process 1 will process rows 3, 4 and 5; therefore it will return the values for variables x3, x4 and x5, in each iteration.

After rows distribution has been performed, the Jacobi iterations are ready to begin. But yet to do it, locations are reserved in corresponding processes for values that will be hold for vectors `xold`, `count` and `displacements`. Vector xold will serve for saving the vector solution from previous iteration of the current one. Understandably that this vector is of length N i.e. the number of variables of system (*) and as initial values for its components we will take 1. The other two vectors length is of the same as the communicator size, namely the value of variable numnodes. In our example these vectors would be of length 3. Before entering the final stage of the Jacobi iterations the following variables needs to be described:

- `i_global` holds the indices of rows processed by the corresponding nodes i.e. on process 2 `i_global` takes values 6, 7, 8 and 9, while on process 0 takes values 0, 1 and 2 and on process 1 takes 3, 4 and 5.
- `displacements` is an integer type vector of size of the communicator. Input *i* specifies the dislocation (depending on the recvbuf) where will be placed data incoming from process *i.*
- `local_offset` the local swerve or the index of the first row of matrices A and b of each process. For example, process 1 ka `local_offset=3` i.e. the first row that will be processed in this node is the third row.
- `count` saves the number of rows which are received from the previous vector solution (`xold`).

As an example, for testing the Jacobi implementation so far explained a system of 10 linear equations is used running on 3 nodes. The Jacobi iterations are implemented with the following lines of code of the Jacobi function:

```
for(k=0; k<maxit; k++){
  error_sum_local = 0.0; sum1 = 0.0; sum2 = 0.0;
  for(j=0; j < i_global; j++)
    sum1 = sum1 + A[i][j]*xold[j];
  for(j=i_global+1; j < N; j++)
    sum2 = sum2 + A[i][j]*xold[j];
  x[i] = (-sum1 - sum2 + b[i])/A[i][i_global];
  error_sum_local += (x[i]-xold[i_global])*(x[i]-
xold[i_global]);
}
COMM_WORLD.Allreduce(&error_sum_local,&error_sum_global,1
,DOUBLE,SUM);
COMM_WORLD.Allgatherv(x,rows_local,DOUBLE,xold,count,disp
lacements,DOUBLE);
  ...
```

These lines of code do the following:

- Find the values of vector solution `x[]` based on formula (**)
- Calculate the sums of the local errors referring to the new values of vector solution `x[]` and those of the previous iteration `xold[]` based on the Euclidean norm (***)

The function *Allgatherv* enables us to specify how much information we are expecting from every node. Recall that standard *Allgather* assumes that each node sends the same amount of data [3]. Specifically, nodes will send their corresponding results of rows, respectively of the corresponding components of vector solution `x`. Based on Table 1 we can indicate the following:

Process 0: sends the first three components of the vector solution `x(0,1,2)`
Process 1: sends the next coming three components of the vector solution `x(3,4,5)`
Process 2: sends the last four components of the vector solution `x(6,7,8,9)`

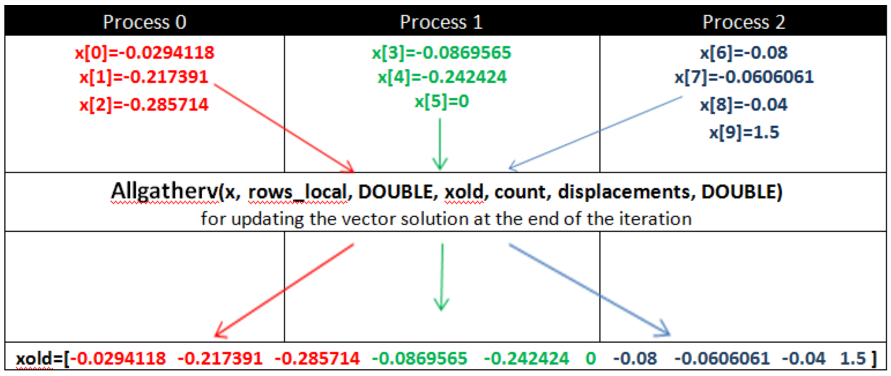

Process 0
Process 1
Process 2
x[0]=-0.0294118
x[1]=-0.217391
x[2]=-0.285714
x[3]=-0.0869565
x[4]=-0.242424
x[5]=0
x[6]=-0.08
x[7]=-0.0606061
x[8]=-0.04
x[9]=1.5
Allgatherv(x, rows_local, DOUBLE, xold, count, displacements, DOUBLE)
for updating the vector solution at the end of the iteration
xold=[-0.0294118 -0.217391 -0.285714 -0.0869565 -0.242424 0 -0.08 -0.0606061 -0.04 1.5]

Table 1. Updating the vector solution `xold` through *Allgatherv*

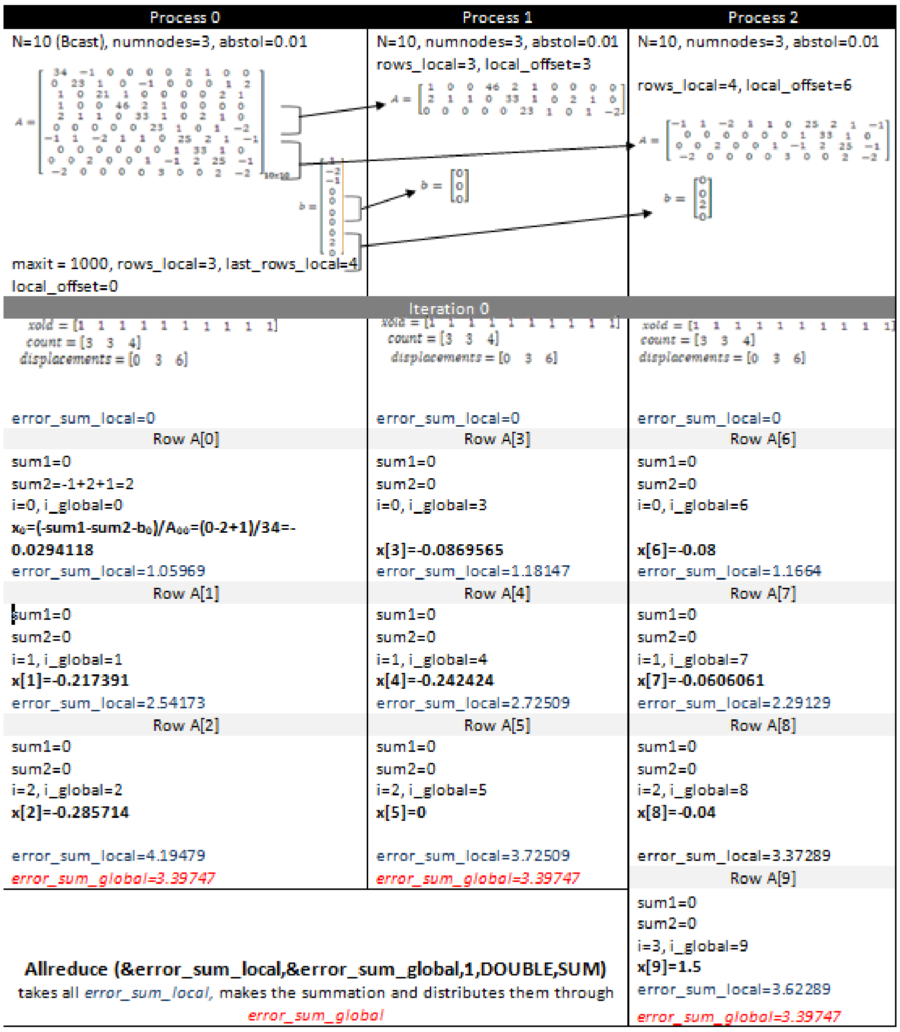
For each iteration, MPI routine *Allreduce* will help us to sum up the whole local errors in one variable `error_sum_global`, which will be compared with the permitted error `abstol` (Table 2). If the result holds the required accuracy then the iterations will stop, the memory will be cleared out of arrays `A`, `b`, `x`, `xold`, `count` and `displacements` and the function will return the value of the last iteration `k`. Otherwise, the iterations will continue until an accepted solution is obtained. In order iterations to not last forever, except the Euclidian norm, there is placed another stopping criteria – the maximum number of iterations `maxit` which is instantiated in our example with the value 1000. Finally, the execution returns to the main() function, which gives the final iteration and closes the parallel environment.

During the testing of our example on SEEUCluster, an acceptable result is obtained at the seventh iteration when error_sum_global takes a value 0.00867182 with the vector solution x=[0.0267993, -0.0943336, -0.060582, -0.000628417, 0.000524287, 0.00245644, -0.00043372, -0.00262186, 0.0878125, 0.0629833].

5 Conclusion

As we outlined on this paper, the initial distribution of the main matrix A and right one b, the Jacobi method is very parallelizable. Only two MPI routines are needed in every iteration – Allreduce for calculating the error and Allgatherv for distributing the vector solution x updated at the end of each iteration. The algorithm gathers the calculated components of vector solution x from every node to formulate the whole vector solution. This method implies that matrix vector multiplications in different nodes are done independently.

As opposed to the high level of parallelism, the Jacobi algorithm suffers from numerous communications issues and do not scale easy for a big number of nodes. However, this paper demonstrates the ability of Jacobi parallelism based on message passing technology through MPI routines. It can serve as a “floor” for further studies, measurements of parallel programming performance, and comparison with other iterative methods.



Process 0	Process 1	Process 2

N=10 (Bcast), numnodes=3, abstol=0.01
$A = \begin{bmatrix} 34 & -1 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 0 \\ 0 & 23 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 2 \\ 1 & 0 & 21 & 1 & 0 & 0 & 0 & 0 & 2 & 1 \\ 1 & 0 & 0 & 46 & 2 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 33 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 23 & 1 & 0 & 1 & -2 \\ -1 & 1 & -2 & 1 & 1 & 0 & 25 & 2 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 33 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & -1 & 2 & 25 & -1 \\ -2 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 2 & -2 \end{bmatrix}_{10x10}$
$b = \begin{bmatrix} 1 \\ -2 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$
maxit = 1000, rows_local=3, last_rows_local=4
local_offset=0
	N=10, numnodes=3, abstol=0.01
	rows_local=3, local_offset=3
	$A = \begin{bmatrix} 1 & 0 & 0 & 46 & 2 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 33 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 23 & 1 & 0 & 1 & -2 \end{bmatrix}$
	$b = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
		N=10, numnodes=3, abstol=0.01
		rows_local=4, local_offset=6
		$A = \begin{bmatrix} -1 & 1 & -2 & 1 & 1 & 0 & 25 & 2 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 33 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & -1 & 2 & 25 & -1 \\ -2 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 2 & -2 \end{bmatrix}$
		$b = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$

Iteration 0		
$xold = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$ $count = [3\ 3\ 4]$ $displacements = [0\ 3\ 6]$	$xold = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$ $count = [3\ 3\ 4]$ $displacements = [0\ 3\ 6]$	$xold = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$ $count = [3\ 3\ 4]$ $displacements = [0\ 3\ 6]$
error_sum_local=0	error_sum_local=0	error_sum_local=0
Row A[0]	Row A[3]	Row A[6]

sum1=0
sum2=-1+2+1=2
i=0, i_global=0
x_0=(-sum1-sum2-b_0)/A_{00}=(0-2+1)/34=-0.0294118
error_sum_local=1.05969
	sum1=0
	sum2=0
	i=0, i_global=3
	x[3]=-0.0869565
	error_sum_local=1.18147
		sum1=0
		sum2=0
		i=0, i_global=6
		x[6]=-0.08
		error_sum_local=1.1664

Row A[1]	Row A[4]	Row A[7]

sum1=0
sum2=0
i=1, i_global=1
x[1]=-0.217391
error_sum_local=2.54173
	sum1=0
	sum2=0
	i=1, i_global=4
	x[4]=-0.242424
	error_sum_local=2.72509
		sum1=0
		sum2=0
		i=1, i_global=7
		x[7]=-0.0606061
		error_sum_local=2.29129

Row A[2]	Row A[5]	Row A[8]

sum1=0
sum2=0
i=2, i_global=2
x[2]=-0.285714
error_sum_local=4.19479
error_sum_global=3.39747
	sum1=0
	sum2=0
	i=2, i_global=5
	x[5]=0
	error_sum_local=3.72509
	error_sum_global=3.39747
		sum1=0
		sum2=0
		i=2, i_global=8
		x[8]=-0.04
		error_sum_local=3.37289

Allreduce (&error_sum_local,&error_sum_global,1,DOUBLE,SUM)
takes all *error_sum_local*, makes the summation and distributes them through *error_sum_global*
		Row A[9]
		sum1=0
		sum2=0
		i=3, i_global=9
		x[9]=1.5
		error_sum_local=3.62289
		error_sum_global=3.39747

Table 2. The first iteration of our running example illustrated in Fig. 4 calculated through Jacobi method in a parallel environment with 3 nodes

References

1. G. E. Karniadakis dhe R. M. Kirby II, Parallel Scientific Computing in C++ and MPI, Cambridge University Press, 2003
2. J. N. Tsitsiklis, *A Comparison of Jacobi and Gauss-Seidel Parallel Iterations*, Massachusetts Institute of Technology, Appl. Math. Left. Vol. 2, No. 2, pp. 167-170, 1989
3. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, dhe J. Dongarra, MPI: The Complete Reference, Massachusetts Institute of Technology, 1996
4. F. Hoxha, “Ushtrime të analizës numerike”, SH. P. “Libri universitar”, 1997
5. A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to parallel computing, 2nd Edition, January 16, 2003
6. H. A. van der Vorst, *Krylov subspace iteration*, Computing in Science & Engineering, January/February 2000
7. JACOBI A Program or the Jacobi Iteration, Available at http://people.sc.fsu.edu/~jburkardt/vt2/fsu_open_mp_2008/jacobi/jacobi.html
8. R. Amorim, G. Haase, M. Liebmann, dhe R. W. dos Santos, *Comparing CUDA and OpenGL implementations for a Jacobi iteration*, 19 December 2008
9. Parallel Jacobi Iterative Scheme, Available at http://scv.bu.edu/~kadin/alliance/apply/solvers/jacobi_parallel.html
10. W. R. Fraser, *Gaussian Elimination vs. Jacobi Iteration*, Project report, 21 November 2008
11. S. Youssef, Iterative methods for sparse linear systems, second edition, January 2000