Parallel Performance Studies for an Elliptic Test problem on the Cluster maya 2013; Using 1-D and 2-D domain subdivisions

Kourosh M. Kalayeh, kourosh2@umbc.edu,

Department of Mechanical Engineering, University of Maryland, Baltimore County

Technical Report HPCF–2014–25, `www.umbc.edu/hpcf` > Publications

Abstract

One of the most important aspects of parallel computing is the communication between processes since it has tremendous impact on overall performance of this method of computing. Consequently, it is important to implement the parallel code in a way that communications between processes are taking place in a most efficient way. In this study we want to investigate the effect of domain subdivision, 1-D or 2-D, on performance of parallel computing. In this regard, the Poisson equation is solved as a test problem using finite difference method with both 1-D and 2-D domain subdivisions. Both aforementioned methods show good speedup. Although in most cases the grid-structured communication show slightly better performance, the overall performance of 2-D domain subdivision does not indicate the superiority of this method.

Key words. Parallel Computing, MPI, Finite Difference, Poisson Equation, 2-D Domain Subdivision, Grid Topology

1 Introduction

1.1 HPCF

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Started in 2008 by more than 20 researchers from ten academic departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. The facility is open to UMBC researchers at no charge. Researchers can contribute funding for long-term priority access. System administration is provided by the UMBC Division of Information Technology, and users have access to consulting support provided by dedicated full-time graduate assistants. See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

Released in Summer 2014, the current machine in HPCF is the 240-node distributed-memory cluster maya. The newest components of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two state-of-the-art NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Phi 5110P accelerators. These new nodes are connected along with the 84 nodes with two quad-core 2.6 GHz Intel Nehalem X5550 CPUs and 24 GB memory by a high-speed quad-data rate (QDR) InfiniBand network for research on parallel algorithms. The remaining 84 nodes with two quad-core 2.8 GHz Intel Nehalem X5560 CPUs and 24 GB memory are designed for fastest number crunching and connected by a dual-data rate (DDR) InfiniBand network. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

The studies in this report use default Intel C compiler version 14.0 (compiler options `-std=c99 -Wall -O3`) with Intel MPI version 4.1. All results in this report use dedicated nodes with remaining cores idling using the `--exclusive` option in the `SLURM` submission script. The default is to use `--shared`, which allocates all processes to cores on one CPU, while `--exclusive` allocates tasks to cores on both CPUs. There is no significant advantage to `--shared` for production runs, that is, performance studies are the only time that this option should be used.

This report is an update to the technical report [1], which considers the same problem on the cluster maya using one-dimensional domain subdivision with blocking communications. In this study, we expand

aforementioned report by studying the parallel performance of the same problem 1) using one-dimensional domain subdivision with nonblocking communication and 2) using two-dimensional domain subdivision.

Consistent with technical report [1], Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two spatial dimensions is solved as a test problem. The problem is solved using finite difference method [2]. Discretizing the spatial derivatives by finite difference yields a system of linear equations with a large, sparse, highly structured, symmetric, positive definite system matrix. As mentioned in [1], this linear system is a classical test problem for iterative solvers. The parallel, matrix-free implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both collectively among all parallel processes and between pairs of processes in every iteration. Therefore, this method provides an excellent test problem for the overall, real-life performance of a parallel computer [1].

The outline of the remaining parts of this report can be categorized as follows; in Sec. 2 the test problem along with numerical method used for solving it, will be explained briefly. In the same section, the correctness of the numerical method will be checked using Matlab by plotting *numerical* solution along with numerical error of the method. This section will be followed by Sec. 3, in which the Matlab code will be transformed to C-code and the convergence of method will be investigated. Sec. 4, will be started by implementing the *parallel* code of *serial* C-code which was developed in former chapter, using one-dimensional domain subdivision with blocking and nonblocking communications, this chapter will be continued with two dimensional domain subdivision with *grid topology* and row/column communicators. In the next chapter, we will present and discuss the results for all three methods of communications. And finally, the conclusion section summarizes and compares the results.

In this report, wherever any code is presented we implicitly assumed that the used variables has been declared accordingly.

2 The Elliptic Test Problem; Poisson Equation

In this report, the Poisson equation with Dirichlet boundary condition as stated in Eq. (2.1) is considered for test problem;

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{2.1}$$

In which $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$ is the unit square domain, $\partial\Omega$ is the boundary of the domain Ω and Laplace operator defined as $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. The right hand side of Eq. (2.1), f, is chosen to be a function in the form of Eq. (2.2). This choice enables us to solve the partial differential equation (PDE) stated in (2.1) analytically and have the true solution as Eq. (2.3), this is important since it enables us to check our numerical solution.

$$f(x,y) = -2\pi^2 \cos(2\pi x)\sin^2(\pi y) - 2\pi^2 \sin^2(\pi x)\cos(2\pi y) \tag{2.2}$$

$$u(x,y) = \sin^2(\pi x)\sin^2(\pi y) \tag{2.3}$$

The numerical method which has been chosen to be used for solving the above PDE is finite difference method [2, 3].

2.1 Finite Difference Discretization

A grid of $N+2$ mesh points in each spatial dimension, $\Omega_h = \{(x_i, y_j) = (ih, jh), i, j = 0, \ldots, N+1\}$ with uniform mesh width $h = 1/(N+1)$ is defined on the domain Ω. By applying second-order finite difference

approximation to the x and y derivative one can obtain

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) \approx \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} \tag{2.4}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} \tag{2.5}$$

By applying (2.4) and (2.5) to (2.1) we can get

$$\begin{aligned} &- \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} \\ &- \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} \approx f(x_i, y_j) \end{aligned} \tag{2.6}$$

for convenience lets introduce u_{ij} as an approximation to $u(x_i, y_j)$ i.e. $u_{ij} \approx u(x_i, y_j)$, thus we will have

$$-\frac{u_{i-1j} - 2u_{ij} + u_{i+1j} + u_{ij-1} - 2u_{ij} + u_{ij+1}}{h^2} = f \tag{2.7}$$

and by further simplification we will have

$$-u_{i-1j} - u_{i+1j} + 4u_{ij} - u_{ij-1} - u_{ij+1} = fh^2 \tag{2.8}$$

Eq. (2.8) is system of N^2 linear equations with N^2 unknown for interior points of the domain and it can be written in standard form of $Au = b$ with $A \in \mathbb{R}^{N^2 \times N^2}$, and $u, b \in \mathbb{R}^{N^2}$. The system matrix A can be defined recursively as tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. As a result, we will have

$$A = \begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix} \tag{2.9}$$

with matrix S =tridiag $(-1, 4, -1) \in \mathbb{R}^{N \times N}$ and matrix $T = -I \in \mathbb{R}^{N \times N}$. For the meshes with large N, iterative methods such as the conjugate gradient (CG) method are appropriate for solving this linear system.The system matrix A is known to be symmetric positive definite and thus method is guaranteed to converge for this problem. In each iteration, the CG method needs exactly two inner products between vectors, three vector updates, and one matrix-vector product involving the system matrix A. An important observation that can be made here is, this matrix-vector product is only place that matrix A plays a role in the algorithm, and as a result matrix-free implementation of the CG method is possible which results in avoiding setting up any matrix. In order to be able to make use of this observation, one has to write a function to carry out this matrix-vector product explicitly.

In light of the above and Eq. (2.8), the `Ax.c` function has been written to calculate `v=Au` in which `v` and `u` are vectors with $N^2 \times 1$ dimension and `A` is a matrix with $N^2 \times N^2$ dimension. The C-interface of the used algorithm to calculate the vector `v` is

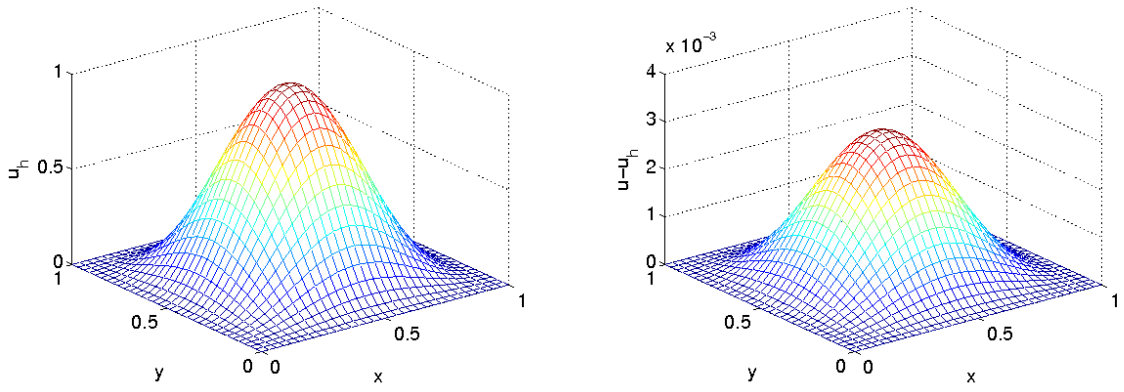
```
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
                      temp = 4.0*u[i+N*j];
    if (j > 0)        temp -= u[i+N*(j-1)];
    if (i > 0)        temp -= u[i-1+N*j];
    if (i < N-1)      temp -= u[i+1+N*j];
    if (j < N-1)      temp -= u[i+N*(j+1)];

    v[i+N*j]=temp;
  }
}
```

It is noteworthy to mention that although C has row major ordering, by using `v[i+N*j]` we are using 1-D column major ordering to store vectors `u` and `v` as a 2-D $N \times N$ array. This holds true for all vectors and matrices presented in this study.

2.2 MATLAB Code

As earlier stated, we start solving the problem using Matlab version of the code in order to make sure that algorithm is working. Fig. 2.1 shows numerical solution $u_h(x_1, x_2)$ and numerical error of the method with 33×33 mesh points i.e. N=32 and $h = 1/32$, using this code. Clearly, the shape of the solution is consistent with true solution of the problem. Also the maximum error occurred at the center of the domain and it is in the order of 10^{-3} which is again consistent with theoretical predicted error $h^2 \approx 10^{-3}$ for finite difference method [2].



(a) (b)

Figure 2.1: mesh plots of (a) the numerical solution u_h, of PDE stated in (2.1) vs. (x_1, x_2) (b)the numerical error $\|u - u_h\|_\infty$ vs. (x_1, x_2) for mesh resolution $N = 32$

3 C-Code; Convergence study for the model problem

Based on Matlab code, we write the serial C-code. As already mentioned, we make use of an important observation from numerical method used to solve this problem, specifically, matrix-free implementation of the CG method. As a result we don't build any matrix, instead we write `Ax.c` function to carry out the matrix-vector product needed in this method. We study the convergence of finite difference method in solving Eq. (2.1) using serial C-code. The CG method was started with zero vector as initial guess and the solution is accepted as converged when the Euclidean vector norm of the residual is reduced to the fraction 10^{-6} of the initial residual. The summary of convergence study of finite difference method on this problem is presented in Table 3.1 for sequence of progressively finer meshes. The true solution of the problem, u, as stated in Eq. (2.3) is used for estimation the error of the numerical method.

The total dimension of the linear system (DOF), the norm of finite difference error $\|u - u_h\|_\infty$, the ratio of consecutive errors $\|u - u_{2h}\|/\|u - u_h\|$, the number of CG iterations `#iter`, and wall clock time in `HH:MM:SS` and seconds has been tabulated for each mesh resolution, N, in Table 3.1. As shown in the mentioned table, the finite difference method converge as expected for this problem. The ratio of consecutive errors is always around 4 which is consistent with theoretical predicted value.

Table 3.1: Convergence study of finite difference method for solving (2.1) using serial C-code.

N	DOF	$\|u-u_h\|_\infty$	Ratio	`#iter`	Wall clock time	
					Seconds	HH:MM:SS
32	1024	3.01E-03	N/A	48	<0.01	<00:00:01
64	4096	7.78E-04	3.87	96	<0.01	<00:00:01
128	16384	1.98E-04	3.94	192	0.01	00:00:01
256	65536	4.98E-05	3.97	387	0.01	00:00:01
512	262144	1.25E-05	3.99	783	0.9	00:00:01
1024	1048576	3.13E-06	4.00	1581	10.01	00:00:10
2048	4194304	7.80E-07	4.01	3192	97.96	00:01:38
4096	16777216	1.94E-07	4.03	6452	782.66	00:13:03
8192	67108864	4.74E-08	4.09	13033	6301.73	01:45:02
16384	268435456	1.16E-08	4.10	26316	51120.88	14:12:01

Table 4.1: Summary of information needed to be received by process `id` from adjacent processes in order to be able calculate `l_v` on process `id` in blocking and nonblocking method of communication

`i`	`l_j`	Needed information	Process to be sent from
`0≤i<N`	`0<l_j<l_N-1`	-	-
`0≤i<N`	`l_j = 0`	`l_u[i+N*(l_N-1)]`	`idleft`
`0≤i<N`	`l_j = l_N-1`	`l_u[i+N*(0)]`	`right`

4 Performance Studies on maya 2013

4.1 One-Dimensional Domain Subdivision

Clearly first step for studying the performance of parallel computing in solving Poisson equation (2.1) is to parallelize the serial C-code developed in former section. In doing so, we implicitly assume that the mesh resolution, N, is devisable by total number of processes, np. In one-dimensional parallel implementation of CG method the domain and consequently each vector is split into as many blocks as parallel processes are available and one block distributed to each process. That is, each parallel process possesses its own block of each vector, and normally no vector is ever assembled in full on any process. In light of the above, `Ax.c` function is the major issue in parallelizing the code.

We start parallelizing the serial code by some simple steps like defining `l_N = N/np` as a local mesh resolution on each process, and changing vectors `u` and `v` to `l_u` and `l_v`, respectively with size of `l_n = n/np` with `n = N*N`. Consequently, the total size of problem on each process also would be changed to `l_n`. Since we are splitting domain in 1-D, the subscripts of the domain on each process would be `0≤i<N` and `0≤l_j<l_N`.

For illustrative purposes, Fig. 4.1 is showing the block of `l_u`'s stored in `np = 3` processes, both in the domain of the problem and in matrix format. By looking at Eq. (2.8) and Fig. 4.1 one can conclude that, the update for l_v_{i0} depends on l_u_{i-10}, l_u_{i+10}, l_u_{i1} on process "id" and on $l\_u_{il\_N-1}$ on process "id-1". Thus, in serial "`Ax.c`" we can calculate `l_v[i+N*l_j]`for all `0≤i<N` and `1≤l_j<l_N-1` locally. But for `l_v[i+N*l_j]` for 0≤i<N and `l_j =0` we need the `l_u[i+N*(l_N-1)]` to be sent from process "idleft" i.e. "id-1" (we call this vector `gl`). Analogously, `l_v[i+N*l_j]`for 0≤i<N and `l_j=l_N -1`, needs `l_u[i+N*0]`from process "idright" i.e. "id+1" (we call this vector `gr`). Table 4.1 summarizes the information needed to be sent from processes `idright` and `idleft` in order to calculate `l_v` on process `id`.

In light of the above, the parallel `Ax.c` function consists of three blocks of code; block A, B, D. Communications take place in block A, while calculations take place in blocks B and D. More specifically, in block B local calculations are carried out, and in block C calculations which need information from other processes are taken care of.

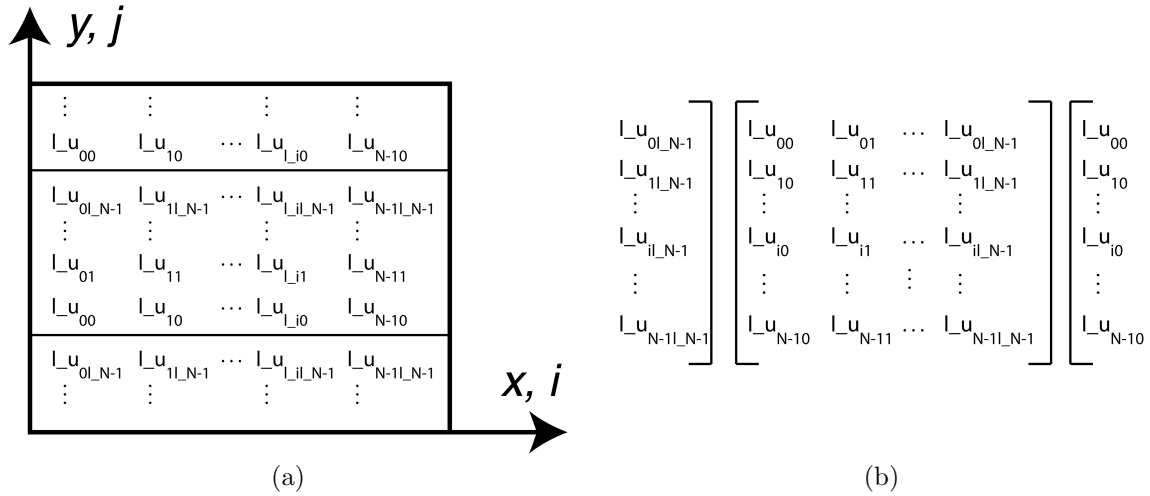


Figure 4.1: (a) Schematic of the unit square domain (b) matrix format of problem (2.1) split one dimensionally into `np=3` processes results in `N*l_N` elements on each process

For implementing the code using blocking communications, in order to avoid "deadlock" which is typical error in parallel algorithms using these kind of communications, the following efficient way for communicating data between processes is used; processes with *even* ranks receive first and then send information, and processes with *odd* ranks send first and then receive the information. This has been accomplished by an "`if-else`" statement. Below is the actual code used for communicating `gl` and `gr` in `Ax.c` function using blocking communication.

```
  /*Block A*/
if ((id%2) == 0){
  MPI_Recv(gl, N, MPI_DOUBLE, idleft, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(gr, N, MPI_DOUBLE, idright, 2, MPI_COMM_WORLD, &status);
  MPI_Send(&l_u[0+N*(l_N-1)], N, MPI_DOUBLE, idright, 1, MPI_COMM_WORLD);
  MPI_Send(&l_u[0], N, MPI_DOUBLE, idleft, 2, MPI_COMM_WORLD);
}else{
  MPI_Send(&l_u[0+N*(l_N-1)], N, MPI_DOUBLE, idright, 1, MPI_COMM_WORLD);
  MPI_Send(&l_u[0], N, MPI_DOUBLE, idleft, 2, MPI_COMM_WORLD);
  MPI_Recv(gl, N, MPI_DOUBLE, idleft, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(gr, N, MPI_DOUBLE, idright, 2, MPI_COMM_WORLD, &status);
}
```

It is noteworthy to mention that in calculation blocks in `Ax.c` function, special cautious should be taken to avoid hitting boundaries of the domain. This can be accomplished by using several simple `if` statements.

Table 5.1 shows the wall clock time of solving test problem (2.1) with 3 different mesh resolutions; 4096×4096, 8192×8192, and 16384×16384 using one dimensional domain subdivision with blocking communications up to 32 nodes with 1, 2, 4, 8, and 16 processes per node.

Also Table 5.4 is showing the parallel performance of solving the problem using 1-D domain subdivision with blocking communications with total processes p =1, 2, 4, 8, 16, 32, 64, 128, and 256 with 8 processes per node (whenever possible), while Table 5.5 is showing the performance study with the same total number of processes but with 16 processes per node (whenever possible). More specifically 8 ppn case uses 1, 2, and 4 processes per node for $p = 1$, $p = 2$, and $p = 4$ respectively and 8 processes per node in all other situations and 16 ppn case uses 1, 2, 4, 8, processes per node for $p = 1$, $p = 2$, $p = 4$, and $p = 8$ respectively and 16 processes per node for all other situations.

Parallel code for one dimensional domain subdivision with nonblocking communications is implemented just by replacing the "MPI_Send" and "MPI_Receive", with the nonblocking send and receive commands i.e. "MPI_Isend" and "MPI_Ireceive" respectively and modifying some of the arguments of those two functions like "Requests" and "Statuses" [4].

Table 5.2 is showing the wall clock time on maya for solving (2.1) using 1-D domain split with nonblocking communications. Consistent with above explanation, Tables 5.6 and 5.7 are also showing the performance of parallel computing using 1-D domain split with nonblocking communications with 8 processes per node and 16 processes per node whenever possible respectively.

4.2 Two-Dimensional Domain Subdivision; Grid Topology

In MPI, a topology is mechanism for associating different addressing schemes with the processes belonging to a group [4]. In this study, we want to identify the processes in MPI_COMM_WORLD with the coordinates of a square grid and each row and each column of grid needs to form its own communicator.

The first assumption that need to be made in this section is that the total number of processes is complete square i.e. $p = q^2$. Then 2-D processes grid by can be set up using MPI_Cart_create command with following syntax

```
MPI_Cart_create(MPI_COMM_WORLD, ndim, dim_size, wrap_around, reorder,
&grid_comm);
```

in which MPI_COMM_WORLD is the communicator that we wish to make our grid communicator from, ndim is the number of dimensions in the grid, dim_size is size of each dimension, wrap_around is the periodicity of each dimension, and reorder enables the system to optimize the mapping of the grid of processes to the underlying physical processors by possibly reordering the processes in the group underlying MPI_COMM_WORLD. In our case, the appropriate values of above arguments are

```
ndim = 2;
dim_size[0] = dimsize[1] = q;
wrap_around[0] = wrap_around[1] = 1;
reorder = 1;
```

After making grid-structured communicator, row and column communicators, row_comm and col_comm, are created for each row and column in the grid using MPI_Cart_sub with following syntax

```
MPI_Cart_sub(grid_comm, row_free_coords, &row_comm);
MPI_Cart_sub(grid_comm, col_free_coords, &col_comm);
```

In the above syntax for MPI_Cart_sub, the row_free_coords and col_free_coords are arrays of boolean and they specify wether each dimension belongs to new communicator or not. Consequently, for our case they are

```
row_free_coords[0] = 0;
row_free_coords[1] = 1;
col_free_coords[0] = 1;
col_free_coords[1] = 0;
```

Fig. 4.2 is showing an schematic of 2-D square grid for $np = 4^2 = 16$, with two sample row and column communicators, the numbers in parentheses are representing two-dimensional Cartesian coordinates of the each process.

Fig. 4.3 is also showing the unit square domain of the test problem (2.1) with grid structure along with l_u's stored in process with coordinates (i, j). Fig . 4.3 (b) is showing the stored l_u's in interior process (i, j) in matrix format with its four adjacent processes.

Figs. 4.2 and 4.3 give very useful insights about the structure of this kind of communication. These figures are also used to determine the data that needed to be sent and received from processes in the grid_comm in order to calculate l_v.

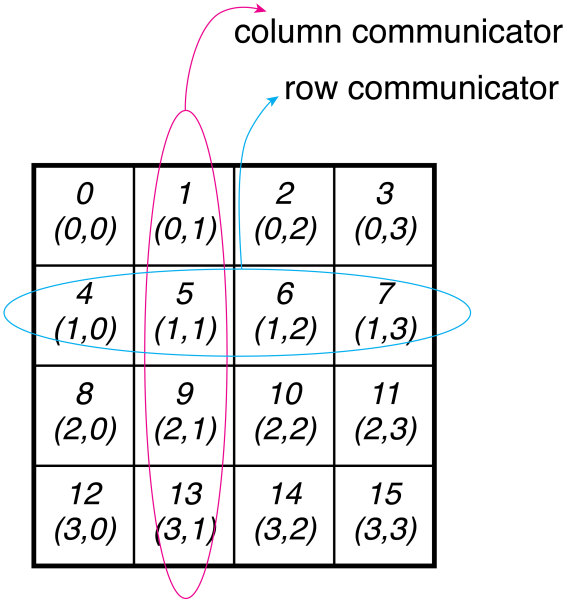
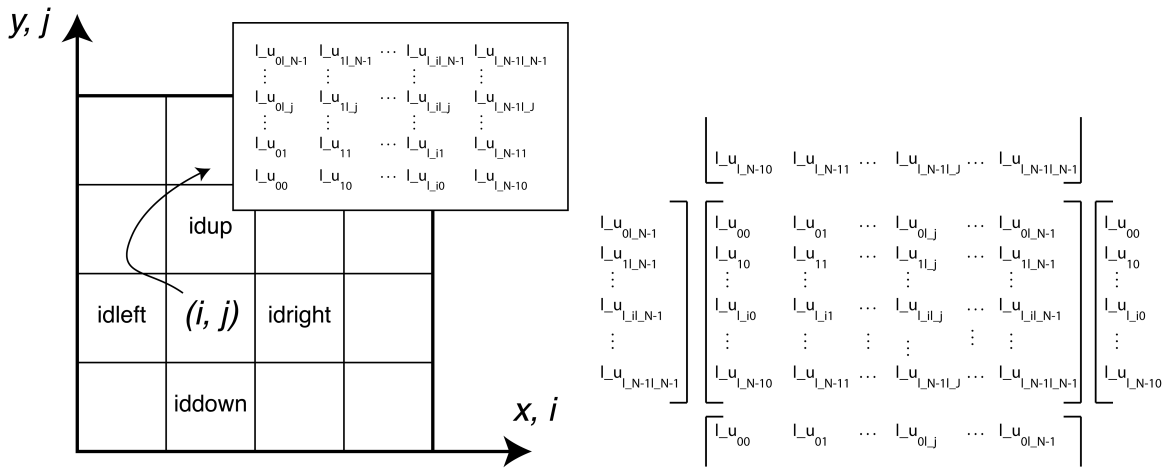


Figure 4.2: Schematic of *grid-structured* processes with their rank and coordinates for a case $16 = 4^2$. There are 4 column and 4 row communicators in this particular case and one of each has been shown for illustrative purposes.



(a) (b)

Figure 4.3: (a)Schematic of unit square domain of test problem (2.1) split two dimensionally into $4^2 = 16$ processes along with `l_u`'s stored in interior process with coordinates (i, j) (b) `l_u`'s stored in process (i, j) in matrix format with its four adjacent processes

As mentioned earlier, in this section unlike Sec. 4.1 the vectors `u` and `v` are split into two dimensions, i and j. Consequently we have `l_i` and `l_j` rather than `i` and `l_j` that we had in one dimensional split. Thus, based on Fig. 4.3, in the parallel `Ax.c` function, we can compute the products for `l_v[l_i, l_j]`, or in 2-D column-major ordering, `l_v[l_i+l_N*l_j]` for all `0<l_i<l_N-1` locally i.e. those only need information from `l_u` on the same process. But for `l_v[l_i + l_N*l_j]` for `0<l_i<l_N-1` and `l_j = 0` we need `l_u[l_i+l_N*(l_N-1)]` to be sent from process `idleft` (see Fig. 4.3). Similarly one can identify `l_u`'s that need to be communicated and processes that need to carry out these communications in order or calculate

Table 4.2: Summary of information needed to be received by process (i, j) from four adjacent processes in order to be able calculate `l_v` in grid-structures communicator

l_i	l_j	Needed information	Process to be sent from
0<l_i<l_N-1	l_j = 0	l_u[l_i+l_N*(l_N-1)]	idleft
0<l_i<l_N-1	l_j = l_N-1	l_u[l_i+l_N*(0)]	idright
l_i = 0	0<l_j<l_N-1	l_u[l_N-1+l_N*(l_j)]	idup
l_i = l_N-1	0<l_j<l_N-1	l_u[0+l_N*(l_j)]	iddown
l_i = 0	l_j = 0	l_u[0+l_N*(l_N-1)]	idright
		l_u[l_N-1+l_N*(0)]	idnorth
l_i = 0	l_j = l_N-1	l_u[0+l_N*(0)]	idright
		l_u[l_N-1+l_N*(l_N-1)]	idnorth
l_i = l_N-1	l_j = 0	l_u[l_N-1+l_N*(l_N-1)]	idleft
		l_u[0+l_N*(0)]	iddown
l_i = l_N-1	l_j = l_N-1	l_u[l_N-1+l_N*(0)]	idright
		l_u[0+l_N*(l_N-1)]	iddown

`l_v[l_i+l_N*l_j]` on process (i, j). Table 4.2 summarizes these information.

The information which has to be sent from `idleft` and/or `idright` will be sent using row communicator, `row_comm`, and the one that needs to be sent from `idup` and `iddown` will be sent using column communicator, `col_comm`.

Careful examination of Table. 4.2 reveals that the information to be sent from `idleft` and `idright` are contiguous in the memory however the one that need to be sent from `idup` and `iddown` are not contiguous in the memory and as a result we can not use typical form of `MPI_Send` and `MPI_Recv`. The efficient solution to overcome this problem is using MPI derived datatypes, and then `MPI_Send` and `MPI_Recv` with `new_mpi_t` as datatype argument, thus we will have

```
MPI_Type_vector(l_N, 1, l_N, MPI_DOUBLE, &new_mpi_t);
```

In light of the above, the `Ax.c` function consists of 5 blocks of codes; block A, block B, block D, block E, and block F. Block A is for communicating needed information from 4 adjacent processes in order to construct `l_v` on process (i, j), we call these arrays `gl, gr, gu` and `gd` consistent with Fig. 4.4. Similar to our blocking code the if-statements are used in order to specify the order of send and receive in row and column communicators. More specifically, the processes with even j-coordinate will receive first and processes with odd j-coordinate will send first. So the code will be

```
/*Block A*/
/* Communicating gl and gr in row_comm */
  if ((j_col%2) == 0){
    MPI_Recv(gl, l_N, MPI_DOUBLE, idleft, 1, row_comm, &status);
    MPI_Recv(gr, l_N, MPI_DOUBLE, idright, 2, row_comm, &status);
    MPI_Send(&l_u[0+l_N*(l_N-1)], l_N, MPI_DOUBLE, idright, 1, row_comm);
    MPI_Send(&l_u[0], l_N, MPI_DOUBLE, idleft, 2, row_comm);
  }else{
    MPI_Send(&l_u[0+l_N*(l_N-1)], l_N, MPI_DOUBLE, idright, 1, row_comm);
    MPI_Send(&l_u[0], l_N, MPI_DOUBLE, idleft, 2, row_comm);
    MPI_Recv(gl, l_N, MPI_DOUBLE, idleft, 1, row_comm, &status);
    MPI_Recv(gr, l_N, MPI_DOUBLE, idright, 2, row_comm, &status);
  }
```

Similar explanation hold true for communicating `gu` and `gd` in `col_comm` i.e. processes with even i-coordinate will receive first and processes with odd i-coordinate will receive first. So we will have

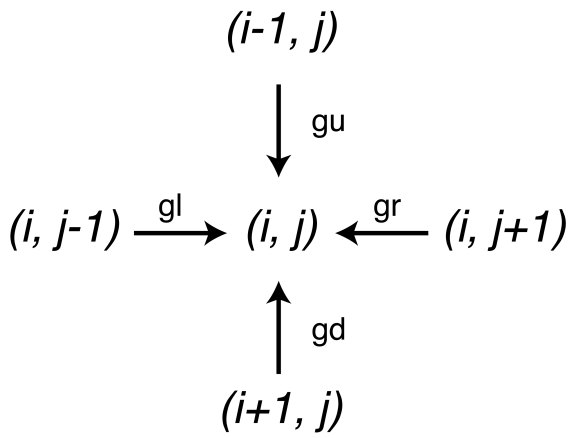


Figure 4.4: Schematic of the needed communication for interior process (i, j). Arrays `gl` and `gr` are used to store the information that has been sent from processes $(i, j-1)$ and $(i, j+1)$ respectively. And `gu` and `gd` are used to store the information that has been sent from processes $(i-1, j)$ and $(i+1, j)$ respectively

```
/* Communicating gu and gd in col_comm */
  if ((i_row%2) == 0){
    MPI_Recv(gu, l_N, MPI_DOUBLE, idup, 3, col_comm, &status);
    MPI_Recv(gd, l_N, MPI_DOUBLE, iddown, 4, col_comm, &status);
    MPI_Send(&l_u[l_N-1+0*l_N], 1, new_mpi_t, iddown, 3, col_comm);
    MPI_Send(&l_u[0], 1, new_mpi_t, idup, 4, col_comm);
  }else{
    MPI_Send(&l_u[l_N-1+0*l_N], 1, new_mpi_t, iddown, 3, col_comm);
    MPI_Send(&l_u[0], 1, new_mpi_t, idup, 4, col_comm);
    MPI_Recv(gu, l_N, MPI_DOUBLE, idup, 3, col_comm, &status);
    MPI_Recv(gd, l_N, MPI_DOUBLE, iddown, 4, col_comm, &status);
  }
```

Other blocks of the `Ax.c` function are calculating blocks; in block B, `l_v` is calculated using local information, in block D, `l_v` is calculated for cases 1 and 2 in Table. 4.2. Block E is for cases 3 and 4 in aforementioned table. And finally, block F is calculating `l_v` for cases 5, 6, 7, and 8 in Table. 4.2. Also in blocks *D, E,* and F we have to be careful about the boundary, we accomplish this with simple if-statements.

5 Results and Discussion

5.1 One Dimensional Domain Subdivision with Blocking Communication

As already mentioned, Table 5.1 collects the results of the performance study for solving Poisson's equation (2.1) in parallel using one dimensional domain subdivision with blocking communications, for mesh resolutions of $N = 4096$, $N = 8192$, and $N = 16384$. More specifically, the wall-clock time of each mesh resolution is reported for every single combination of 2^i nodes and 2^j processes on each node with $i = 0, 1, 2, 3, 4, 5$ and $j = 0, 1, 2, 3, 4$ to ultimately use the maximum cores on each node. The reported time is in the format of `HH:MM:SS`.

As explained earlier Tables 5.4 and 5.5 are reporting the performance of parallel computing using 1-D domain split with blocking communications for 8 and 16 processes per node respectively. More specifically in these tables, (a) wall clock time in seconds, (b) wall clock time in `HH:MM:SS` format, (c) observed speedup defined as $S_p = T_1(N)/T_p(N)$, and (d) observed efficiency defined as $E_p = S_p/p$ are demonstrated.Also Figs. 5.1 and 5.2 are showing observed speedup and efficiency of the method for 8 and 16 processes per node

respectively. As stated in the tables and shown in the aforementioned figures, the performance of the one with 8 processes per node is significantly better than the one with 16 processes per node.

5.2 One Dimensional Domain Subdivision with nonblocking Communication

Table 5.2 collects the results of the performance study for solving Poisson's equation (2.1) in parallel using one dimensional domain subdivision with nonblocking communications, for mesh resolutions of $N = 4096$, $N = 8192$, and $N = 16384$. More specifically, the wall-clock time of each mesh resolution is reported for every single combination of 2^i nodes and 2^j processes on each node with $i = 0, 1, 2, 3, 4, 5$ and $j = 0, 1, 2, 3, 4$ to ultimately use the maximum cores on each node. The reported time is in the format of `HH:MM:SS`.

Analogously, Tables 5.6 and 5.7 are tabulated with wall clock time in (a) seconds, (b) `HH:MM::SS` format (c) observed speed up S_p, and (d) observed efficiency E_p for 1-D split with nonblocking communications using 8 and 16 processes per node in order to investigate the performance of this method of communication. Figs. 5.3 and 5.4 are showing observed speedup and efficiency of the method for 8 and 16 processes per node respectively. Performance of parallel computing with nonblocking communications using 8 processes per node is much higher, which is in line with the data obtained from blocking communications.

Also comparing the results from blocking and nonblocking communications with 8 processes per node presented in Tables 5.4 and 5.6, Figs. 5.1 and 5.3 reveals that blocking and nonblocking communications result in identical performance. The efficient way of communication in blocking implementation of the code is the reason for this behavior. In fact this method of communication enhances the performance of blocking code to such an extent that the superiority of nonblocking communications diminishes.

As a result of this observation, we carry out the performance study of solving the problem using two-dimensional domain subdivision with blocking communications only.

5.3 Two-Dimensional Grid-structured Communicator

Table 5.3 is showing the wall-clock time for solving the problem split into two dimensions using grid-structured communications. As mentioned earlier, for two dimensional split total number of processes should be a complete square and consequently only some of the combinations of nodes and processes per node can be used in performance study. For instance, with one node, only 1, 4, and 16 processes per node and similarly for 2 nodes, only 2, and 8 processes per node can be used. With this logic we tabulate Table 5.3. The *NA* in the aforementioned table indicates the combinations that yield to a total number of processes which are not complete square.

As in Sec. 5.2, the performance study for two-dimensional domain subdivision is carried out with up to 256 total number of processes with 8 and 16 processes per node (whenever possible). The results are shown in Tables 5.8 and 5.9 respectively. Also Figs. 5.5 and 5.6 are showing speedup and efficiency of parallel computing for two-dimensional split with 8 and 16 processes per node respectively. As indicated in aforementioned tables and figures, consistent with results from one dimensional domain subdivision, the performance of parallel computing with 16 processes per node decreases dramatically in comparison to 8 processes per node. Also, results from two-dimensional split are suggesting slight improvement with respect to one dimensional split, although we expect to see much better performance. The reason for this behavior can be the fact that this cluster, and its network card specifically, is so powerful that brings down the superiority of two dimensional split.

Table 5.1: Wall clock time in HH:MM:SS for solving Eq. (2.1) by finite difference method with parallel computing on cluster maya 2013 using the Intel compiler with Intel MPI. The results are obtained using *one-dimensional* subdomain devision with *blocking* communication

(a) Mesh resolution $N \times N = 4096 \times 4096$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	00:12:23	00:06:22	00:03:14	00:01:43	00:00:42	00:00:17
ppn= 2	00:06:17	00:03:15	00:01:37	00:00:42	00:00:16	00:00:09
ppn= 4	00:03:30	00:01:49	00:00:58	00:00:26	00:00:10	00:00:06
ppn= 8	00:02:40	00:01:22	00:00:43	00:00:19	00:00:06	00:00:04
ppn= 16	00:02:43	00:01:23	00:00:42	00:00:17	00:00:07	00:00:06
(b) Mesh resolution $N \times N = 8192 \times 8192$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	01:41:21	00:50:47	00:25:32	00:12:56	00:06:38	00:03:21
ppn= 2	00:51:06	00:25:49	00:13:05	00:05:39	00:03:21	00:01:30
ppn= 4	00:28:43	00:14:31	00:07:20	00:03:48	00:01:58	00:00:54
ppn= 8	00:21:32	00:10:57	00:05:37	00:02:53	00:01:31	00:00:42
ppn= 16	00:21:03	00:10:36	00:05:23	00:02:51	00:01:31	00:00:46
(c) Mesh resolution $N \times N = 16384 \times 16384$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	13:54:17	06:57:50	03:29:55	01:45:21	00:53:26	00:27:19
ppn= 2	00:06:17	03:31:58	01:47:03	00:53:59	00:27:29	00:14:14
ppn= 4	03:57:15	02:04:20	01:02:24	00:30:36	00:15:35	00:08:11
ppn= 8	02:57:12	01:29:13	00:44:40	00:22:34	00:11:41	00:06:16
ppn= 16	02:50:22	01:25:24	00:43:11	00:21:49	00:11:29	00:06:19

Table 5.2: Wall clock time in HH:MM:SS for solving Eq. (2.1) by finite difference method with parallel computing on cluster maya 2013 using the Intel compiler with Intel MPI. The results are obtained using *one-dimensional* subdomain devision with *nonblocking* communication

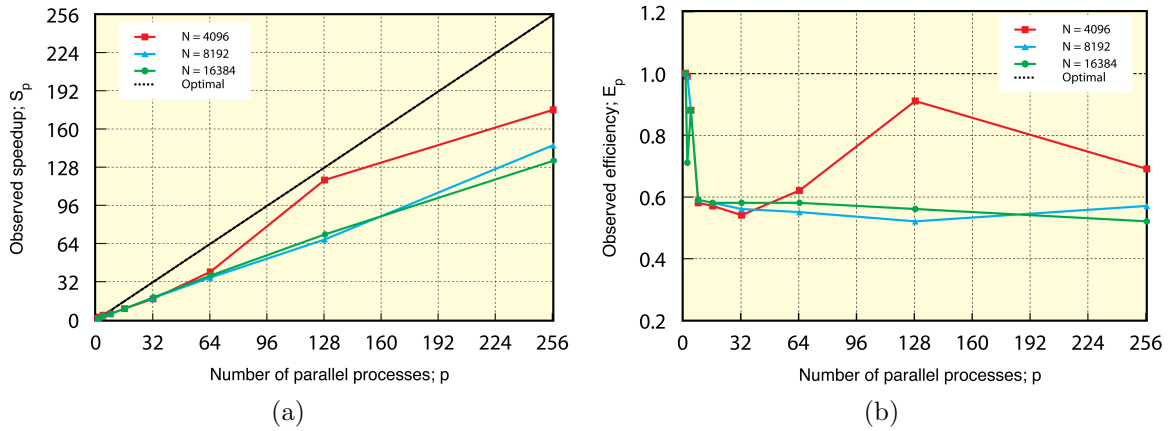
(a) Mesh resolution $N \times N = 4096 \times 4096$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	00:12:29	00:06:18	00:03:10	00:01:35	00:00:42	00:00:00
ppn= 2	00:06:18	00:03:12	00:01:37	00:00:42	00:00:16	00:00:00
ppn= 4	00:03:31	00:01:49	00:00:56	00:00:24	00:00:09	00:00:00
ppn= 8	00:02:41	00:01:22	00:00:43	00:00:18	00:00:06	00:00:00
ppn= 16	00:02:42	00:01:22	00:00:42	00:00:17	00:00:06	00:00:00
(b) Mesh resolution $N \times N = 8192 \times 8192$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	01:41:20	00:50:44	00:25:30	00:12:56	00:06:33	00:03:18
ppn= 2	00:51:18	00:25:45	00:13:02	00:05:39	00:03:21	00:01:28
ppn= 4	00:28:27	00:14:17	00:07:22	00:03:48	00:01:57	00:00:53
ppn= 8	00:21:30	00:10:51	00:05:29	00:02:53	00:01:30	00:00:42
ppn= 16	00:21:03	00:10:38	00:05:23	00:02:51	00:01:30	00:00:45
(c) Mesh resolution $N \times N = 16384 \times 16384$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	13:54:02	06:57:22	03:29:17	01:45:03	00:53:33	00:27:06
ppn= 2	09:44:11	03:31:54	01:46:14	00:53:45	00:27:29	00:14:07
ppn= 4	03:52:55	01:56:16	00:59:40	00:30:22	00:15:36	00:08:09
ppn= 8	02:56:16	01:28:40	00:44:42	00:22:33	00:11:24	00:06:13
ppn= 16	02:50:11	01:25:23	00:43:20	00:21:57	00:11:28	00:06:29

Table 5.3: Wall clock time in HH:MM:SS for solving Eq. (2.1) by finite difference method with parallel computing on cluster maya 2013 using the Intel compiler with Intel MPI. The results are obtained using *two-dimensional* subdomain devision with blocking communication. As indicated in the table only combinations of nodes and processes per node which yield to a complete square total number of processes can be used in *grid-structured* communication.

(a) Mesh resolution $N \times N = 4096 \times 4096$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	00:12:23	NA	00:03:09	NA	00:00:40	NA
ppn= 2	NA	00:03:13	NA	00:00:40	NA	00:00:08
ppn= 4	00:03:38	NA	00:00:55	NA	00:00:09	NA
ppn= 8	NA	00:01:23	NA	00:00:19	NA	00:00:04
ppn= 16	00:02:41	NA	00:00:43	NA	00:00:06	NA
(b) Mesh resolution $N \times N = 8192 \times 8192$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	01:41:20	NA	00:25:03	NA	00:06:28	NA
ppn= 2	NA	00:25:35	NA	00:06:35	NA	00:01:24
ppn= 4	00:27:47	NA	00:07:24	NA	00:01:56	NA
ppn= 8	NA	00:10:58	NA	00:02:52	NA	00:00:39
ppn= 16	00:21:18	NA	00:05:27	NA	00:01:34	NA
(c) Mesh resolution $N \times N = 16384 \times 16384$						
	1 node	2 node	4 node	8 node	16 node	32 node
ppn= 1	13:54:03	NA	03:22:52	NA	00:50:36	NA
ppn= 2	NA	03:25:07	NA	00:51:26	NA	00:13:40
ppn= 4	03:47:40	NA	00:58:43	NA	00:15:14	NA
ppn= 8	NA	01:27:29	NA	00:22:29	NA	00:05:55
ppn= 16	02:49:28	NA	00:43:22	NA	00:11:15	NA

Table 5.4: Performance study of solving (2.1) on cluster maya 2013 using *one-dimensional* subdomain devision with *blocking* communication. In obtaining these results 8 processes per node are used whenever possible.

(a) Wall clock time in seconds									
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	743.27	377.13	210.23	160.26	82.11	43.15	18.65	6.36	4.23
8192	6081.34	3066.05	1723.06	1292.38	657.42	337.23	173.41	90.81	41.64
16384	50056.87	35051.14	14235.05	10631.79	5353.07	2679.84	1353.6	700.58	375.83
(b) Wall clock time in HH:MM:SS									
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	00:12:23	00:06:17	00:03:30	00:02:40	00:01:22	00:00:43	00:00:19	00:00:06	00:00:04
8192	01:41:21	00:51:06	00:28:43	00:21:32	00:10:57	00:05:37	00:02:53	00:01:31	00:00:42
16384	13:54:17	09:44:11	03:57:15	02:57:12	01:29:13	00:44:40	00:22:34	00:11:41	00:06:16
(c) Observed speedup S_p									
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	1.97	3.54	4.64	9.05	17.23	39.85	116.87	175.71
8192	1.00	1.98	3.53	4.71	9.25	18.03	35.07	66.97	146.05
16384	1.00	1.43	3.52	4.71	9.35	18.68	36.98	71.45	133.19
(d) Observed efficiency E_p									
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	0.99	0.88	0.58	0.57	0.54	0.62	0.91	0.69
8192	1.00	0.99	0.88	0.59	0.58	0.56	0.55	0.52	0.57
16384	1.00	0.71	0.88	0.59	0.58	0.58	0.58	0.56	0.52



(a) (b)

Figure 5.1: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) using *1-D* sub domain split with *blocking* communications. In obtaining these results 8 processes per node were used whenever possible.

Table 5.5: Performance study of solving (2.1) on cluster maya 2013 using *one-dimensional* subdomain devision with *blocking* communication. In obtaining these results 16 processes per node are used whenever possible.

(a) Wall clock time in seconds

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	743.27	377.13	210.23	160.26	162.59	82.5	41.56	17.08	6.77
8192	6081.34	3066.05	1723.06	1292.38	1263.15	636	323.01	171.45	91.49
16384	50056.87	35051.14	14235.05	10631.79	10221.6	5123.9	2590.83	1309.33	688.61

(b) Wall clock time in HH:MM:SS

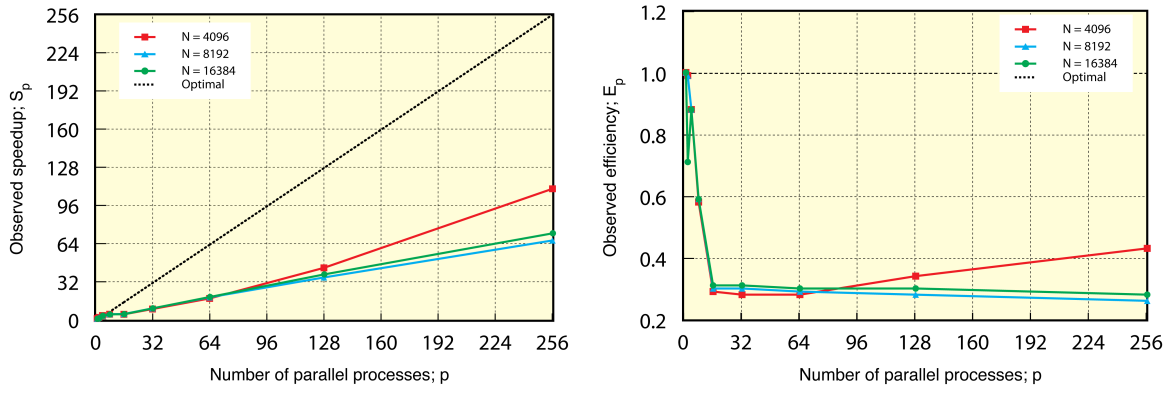
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	00:12:23	00:06:17	00:03:30	00:02:40	00:02:43	00:01:23	00:00:42	00:00:17	00:00:07
8192	01:41:21	00:51:06	00:28:43	00:21:32	00:21:03	00:10:36	00:05:23	00:02:51	00:01:31
16384	13:54:17	09:44:11	03:57:15	02:57:12	02:50:22	01:25:24	00:43:11	00:21:49	00:11:29

(c) Observed speedup S_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	1.97	3.54	4.64	4.57	9.01	17.88	43.52	109.79
8192	1.00	1.98	3.53	4.71	4.81	9.56	18.83	35.47	66.47
16384	1.00	1.43	3.52	4.71	4.90	9.77	19.32	38.23	72.69

(d) Observed efficiency E_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	0.99	0.88	0.58	0.29	0.28	0.28	0.34	0.43
8192	1.00	0.99	0.88	0.59	0.30	0.30	0.29	0.28	0.26
16384	1.00	0.71	0.88	0.59	0.31	0.31	0.30	0.30	0.28



(a) (b)

Figure 5.2: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) using *1-D* subdomain split with *blocking* communications. In obtaining these results 16 processes per node are used whenever possible.

Table 5.6: Performance study of solving (2.1) on cluster maya 2013 using *one-dimensional* subdomain division with *nonblocking* communication. In obtaining these results 8 processes per node are used whenever possible.

(a) Wall clock time in seconds

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	748.98	378.37	210.62	161	82.49	42.53	18.11	6.27	4.33
8192	6079.61	3078.07	1706.81	1290.05	650.79	329.48	172.06	90.23	42.31
16384	50042.28	35051.14	13974.89	10576.04	5320.06	2682.23	1353.15	684.46	372.95

(b) Wall clock time in HH:MM:SS

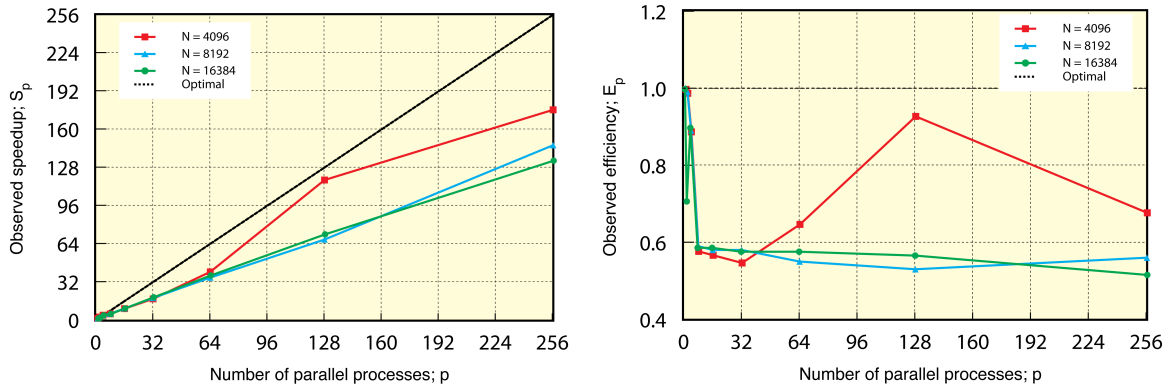
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	00:12:29	00:06:18	00:03:31	00:02:41	00:01:22	00:00:43	00:00:18	00:00:06	00:00:04
8192	01:41:20	00:51:18	00:28:27	00:21:30	00:10:51	00:05:29	00:02:52	00:01:30	00:00:42
16384	13:54:02	09:44:11	03:52:55	02:56:16	01:28:40	00:44:42	00:22:33	00:11:24	00:06:13

(c) Observed speedup S_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	1.98	3.56	4.65	9.08	17.61	41.36	119.45	172.97
8192	1.00	1.98	3.56	4.71	9.34	18.45	35.33	67.38	143.69
16384	1.00	1.43	3.58	4.73	9.41	18.66	36.98	73.11	134.18

(d) Observed efficiency E_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	0.99	0.89	0.58	0.57	0.55	0.65	0.93	0.68
8192	1.00	0.99	0.89	0.59	0.58	0.58	0.55	0.53	0.56
16384	1.00	0.71	0.90	0.59	0.59	0.58	0.58	0.57	0.52



(a) (b)

Figure 5.3: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) on cluster maya 2013 using *1-D* subdomain split with *nonblocking* communications. In obtaining these results 8 processes per node are used whenever possible.

Table 5.7: Performance study of solving (2.1) on cluster maya 2013 using *one-dimensional* subdomain devisions with *nonblocking* communications. In obtaining these results 16 processes per node are used whenever possible.

(a) Wall clock time in seconds

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	748.98	378.37	210.62	161	161.85	81.98	41.58	17.35	5.99
8192	6079.61	3078.07	1706.81	1290.05	1263.44	638.33	322.7	167.91	90.32
16384	50042.28	35051.14	13974.89	10576.04	10211.37	5122.68	2600.31	1316.56	687.69

(b) Wall clock time in HH:MM:SS

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	00:12:29	00:06:18	00:03:31	00:02:41	00:02:42	00:01:22	00:00:42	00:00:17	00:00:06
8192	01:41:20	00:51:18	00:28:27	00:21:30	00:21:03	00:10:38	00:05:23	00:02:48	00:01:30
16384	13:54:02	09:44:11	03:52:55	02:56:16	02:50:11	01:25:23	00:43:20	00:21:57	00:11:28

(c) Observed speedup S_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	1.98	3.56	4.65	4.63	9.14	18.01	43.17	125.04
8192	1.00	1.98	3.56	4.71	4.81	9.52	18.84	36.21	67.31
16384	1.00	1.43	3.58	4.73	4.90	9.77	19.24	38.01	72.77

(d) Observed efficiency E_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
4096	1.00	0.99	0.89	0.58	0.29	0.29	0.28	0.34	0.49
8192	1.00	0.99	0.89	0.59	0.30	0.30	0.29	0.28	0.26
16384	1.00	0.71	0.90	0.59	0.31	0.31	0.30	0.30	0.28

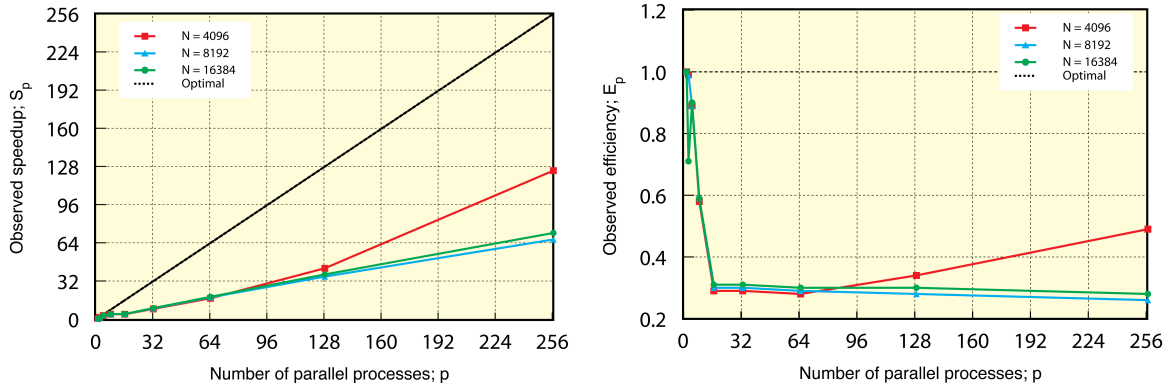
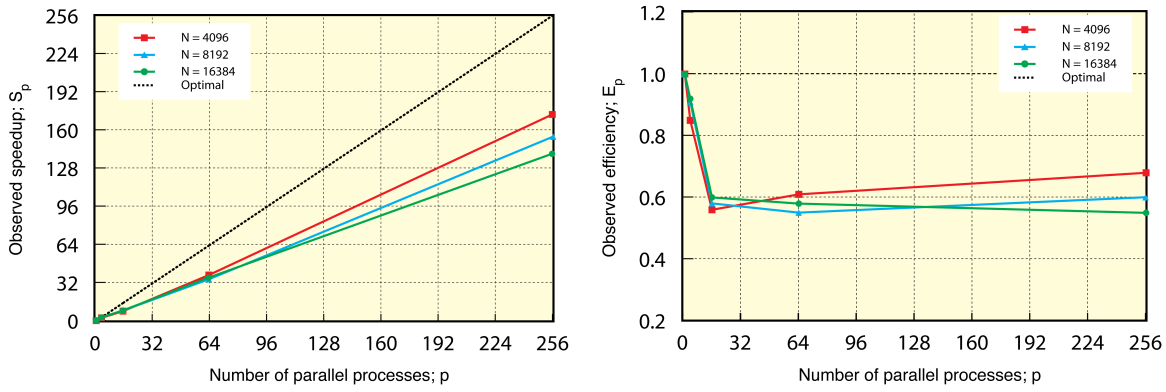


Figure 5.4: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) on cluster maya 2013 using *1-D* split with *nonblocking* communications. In obtaining these results 16 processes per node are used whenever possible.

Table 5.8: Performance study of solving (2.1) on cluster maya 2013 using *two-dimensional* subdomain devision with blocking communication. In obtaining these results 8 processes per node are used whenever possible

(a) Wall clock time in seconds					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	742.78	218.35	82.56	18.91	4.27
8192	6080.04	1666.84	658.06	171.66	39.32
16384	50042.98	13660.22	5248.86	1349.19	354.97
(b) Wall clock time in HH:MM:SS					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	00:12:23	00:03:38	00:01:23	00:00:19	00:00:04
8192	01:41:20	00:27:47	00:10:58	00:02:52	00:00:39
16384	13:54:03	03:47:40	01:27:29	00:22:29	00:05:55
(c) Observed speedup S_p					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	1.00	3.40	9.00	39.28	173.95
8192	1.00	3.65	9.24	35.42	154.63
16384	1.00	3.66	9.53	37.09	140.98
(d) Observed efficiency E_p					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	1.00	0.85	0.56	0.61	0.68
8192	1.00	0.91	0.58	0.55	0.60
16384	1.00	0.92	0.60	0.58	0.55

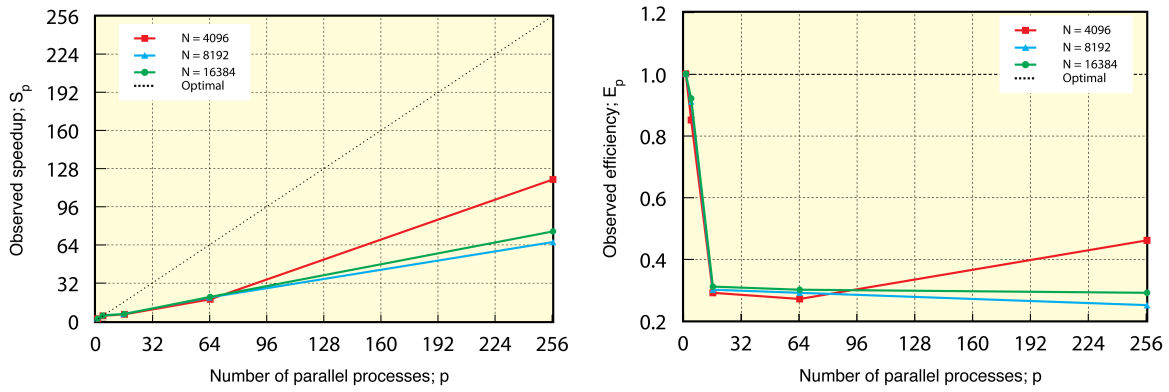


(a) (b)

Figure 5.5: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) on cluster maya 2013 using *2-D* subdomain split with blocking communications. In obtaining these results 8 processes per node are used whenever possible.

Table 5.9: Performance study of solving (2.1) on cluster maya 2013 using *two-dimensional* subdomain devision with blocking communications. In obtaining these results 16 processes per node are used whenever possible.

(a) Wall clock time in seconds					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	742.78	218.35	160.82	43.14	6.31
8192	6080.04	1666.84	1278.04	327.14	93.72
16384	50042.98	13660.22	10168.43	2602.15	675.01
(b) Wall clock time in HH:MM:SS					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	00:12:23	00:03:38	00:02:41	00:00:43	00:00:06
8192	01:41:20	00:27:47	00:21:18	00:05:27	00:01:34
16384	13:54:03	03:47:40	02:49:28	00:43:22	00:11:15
(c) Observed speedup S_p					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	1.00	3.40	4.62	17.22	117.71
8192	1.00	3.65	4.76	18.59	64.87
16384	1.00	3.66	4.92	19.23	74.14
(d) Observed efficiency E_p					
N	$p=1$	$p=4$	$p=16$	$p=64$	$p=256$
4096	1.00	0.85	0.29	0.27	0.46
8192	1.00	0.91	0.30	0.29	0.25
16384	1.00	0.92	0.31	0.30	0.29



(a) (b)

Figure 5.6: (a) Observed speedup (b)observed efficiency in solving Eq. (2.1) on cluster maya 2013 using t*2-D* subdomain split with blocking communications. In obtaining these results 16 processes per node are used whenever possible.

6 Conclusions

In this report, the Poisson equation was solved as a test problem to study the performance of parallel computing on cluster maya 2013 using one dimensional domain subdivision with blocking, and nonblocking communications, and two dimensional domain subdivision with grid-structured communications.

All of the results from one dimensional split with blocking and nonblocking communications as well as two dimensional split showed satisfactory speedup. Moreover, one dimensional split with both methods of communications suggested identical results. As discussed in details, this behavior stems from the efficient implementation of send/receive commands used in blocking method of communications. Another important observation was the huge improvement of performance with using 8 processes per node whenever possible instead of using 16 processes per node.

The results from two dimensional split showed slight improvement, which was not what we expected. The reason for this behavior can be the power of cluster maya 2013 and its network card in particular which diminishes the superiority of two dimensional split. In consistent with one dimensional domain subdivision, the results using two dimensional split with 8 processes per node showed much higher performance in comparison to 16 processes per node.

7 Acknowledgments

This technical report started as a final project for Math 627, *Introduction to Parallel Computing*, instructed by Professor Matthias K. Gobbert during Fall 2014 at UMBC.

I would like to express my sincere gratitude to my advisor, Professor Panos G. Charalambides for his support. I also would like to thank Professor Gobbert for all of his help and guidance .

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

References

[1] S. Khuvis and M. K. Gobbert, "Parallel performance studies for an elliptic test problem on the cluster maya," Technical Report HPCF–2014–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[2] K. Atkinson, *An Introduction to Numerical Analysis.* Wiley, 2nd ed., 1989.

[3] D. Braess, *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics.* Cambridge ; New York: Cambridge University Press, 3rd ed., 2007.

[4] P. Pacheco, *Parallel Programming with MPI.* San Francisco, Calif: Morgan Kaufmann, 1997.