Parallel Programming - Jacobi Method

Keerthana C J

June 14, 2021

1 Introduction

The main objective of this exercise is to parallelize a serial code which solves the 2D Laplace equation using Jacobi Method. After the code is parallelized, we make a performance analysis of the code tested for two different sizes of the domain i.e. the matrix dimension $N = 1200$ and $N = 12000$. For parallelization, we use Open MPI. Two versions of the code are created one with a blocking send and receive communication pattern between the processors and the other one with a non-blocking communication. We present the communication and calculation times of both versions of the code and for both sizes also. The speed up is also plotted for each test case. The different test cases are:

- $N = 1200$ and blocking
- $N = 1200$ and non-blocking
- $N = 12000$ and blocking
- $N = 12000$ and non-blocking

2 Laplace Equation using Jacobi Method

The Laplace equation is a second order partial differential equation which in 2D takes the form:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{1}$$

, where f is a double differentiable function. To solve PDEs we resort to numerical methods. We use a Finite Difference approach to discretize the PDE on a square domain with equal size grids. This converts the PDE to a set of simultaneous algebraic equations taking the form $Ax = b$, where A is 2D square matrix of size N, which is the number of points used to discretize the domain along one direction, x is the unknown vector containing the solution to the function f at each discretized point in the domain, while b is the right hand side vector consituting the boundary conditions provided for the domain. To solve for x, different technquies are available of which the most fundamental iterative technique is the Jacobi method. Here, we already have serial code implementing the Jacobi method for solving the Laplace equation with given boundary conditions. We understand the main aspects of the code and begin to parallelize the code.

3 Serial Code

The serial code has the following parts:

- Allocate a 2D array of given dimension along with the boundaries. The boundaries do not belong to the domain, hence, if the given dimension is N, the 2D array is allocated for $(N+2) \times (N+2)$
- Set up the boundary conditions for the left BC and bottom BC, as an decremental value from 100 to 0 starting from the origin.
- Set up the initial values inside the domain
- Begin and continue for a fixed number of cycles the iterative process. At each iteration, the value of each inner matrix element needs to be recomputed from elements of the current iteration. The equation for calculation is:

$$V_{i,j}^{new} = 0.25(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}) \tag{2}$$

  where the subscripts i and j denote the position of the element in the 2D discretized domain.
- After updating, copy the new matrix into the old's memory and continue iterations until completion.

4 Parallelize

The parallelize involves splitting the domain across the processors. Hence, from the serial code, performing the parallelization invloves the following:

- Each processor allocated a 2D array of size lesser than the entire domain. A domain decomposition in 1 dimension leads to each processor containing a 2D matrix size of $N \times N/np$, where np is the number of local processors. As seen in the previous exercise, the domain decomposition needs a 2 buffer layers in the local matrix for boundary exchange. Hence considering boundary exchange and boundary conditions, each processor allocates a 2D array (local matrix) of size $(N+2) \times (N/np+2)$.
- Each processor sets the boundary conditions for the boundaries included within the each processor.
- Each processor sets the initial values for the interior of its local matrix.
- As the iteration loop begins, to calculate the value of each new element in the matrix, we need the current value of the elements to the right, left, top and bottom of this element. The interior elements inside each processor already contains the data for calculation. When we reach the borders of the local matrix, we need the information from the other processors. This leads to the boundary exchange as discussed in the previous exercise. The boundary exchange leads to communication between the processors and the borders can be evolved once the communication is performed. We split the iteration loop into the following tasks.

- Exchange boundaries between the processors, the top boundary is sent to processor $i+1$ while the bottom boundary is received from the processor $i-1$.
  $\rightarrow$ Exchange Boundary
- The calculation is performed in the bulk (domain excluding the boundaries).
  $\rightarrow$ Evolve Bulk
- The iteration is done in the borders once the boundary exchange is performed.
  $\rightarrow$ Evolve Border

5 Blocking and Non-blocking communication

For exchanging the data between the processors, two types of MPI communication can be performed. One is the blocking send-receive, while the other is the non-blocking send-receive. During the blocking communication, no other function can be performed by the processor. The processor waits for the send-receive operation to be completed to being the next task. While, the non-blocking communication is one where the communication and other operations can be overlapped. While the data bus is used for sending/receiving information, the processor can perform the calculations which are independent of the data that is exchanged between the processors. This leads to a reduction in the entire time of the operation. One of the goals of the exercise is to implement two versions of the code one involving blocking communication while the other involving non-blocking communication. We would like to investigate how far the overlapping of communication and calculation times can increase the speed up of the code.

6 Performance Analysis

The code is analyzed for the following four tests cases within one node and across nodes:

- $N = 1200$ and blocking
- $N = 1200$ and non-blocking
- $N = 12000$ and blocking
- $N = 12000$ and non-blocking

We perform 10 iterations in each case and measure the communication and calculation times. The total time is the sum of the communication and calculation time. The speed up is calculated using the total time using the formula:

$$Speed\,Up\,for\,n\,processors = \frac{Total\,time\,for\,10\,iterations\,on\,1\,processor}{Total\,time\,for\,10\,iterations\,on\,n\,processors} \tag{3}$$

If we consider nodes, then the Speed up is calculated as:

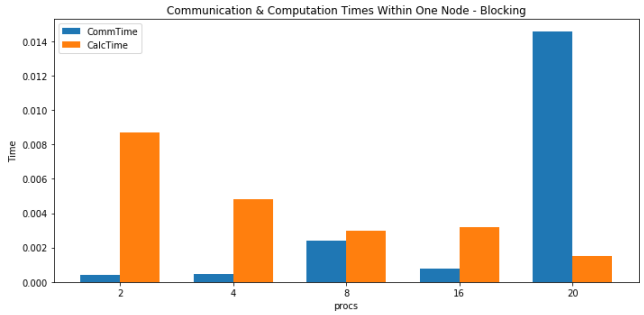
$$Speed\,Up\,for\,n\,nodes = \frac{Total\,time\,for\,10\,iterations\,on\,1\,node\,using\,all\,processors}{Total\,time\,for\,10\,iterations\,on\,n\,nodes\,using\,all\,processor} \tag{4}$$

The experiments were conducted using the supercomputer Ulysses at SISSA. Each node of Ulysses used over here has 20 processors. The code is run for the matrix sizes $N = 1200$ and $N = 12000$ for both blocking and non-blocking versions of the code.

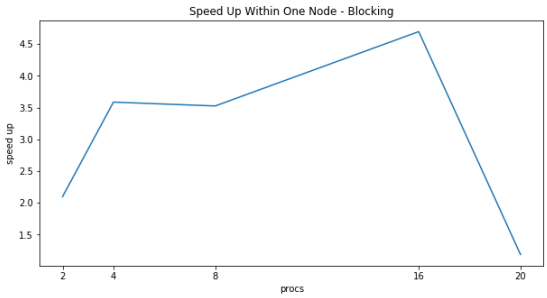
6.1 Blocking Communication

6.1.1 Within a single node

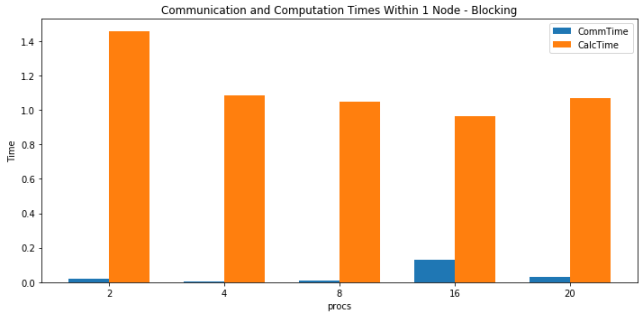
Figure 1 contains the plots for the Blocking communication on a single node where the right panel provide the speed up and the left panel provide the communication and computation times. The top panel is for the size $N = 1200$ and the bottom panel is for $N = 12000$. From the plots, we can see a considerable increase in the communication time for the size



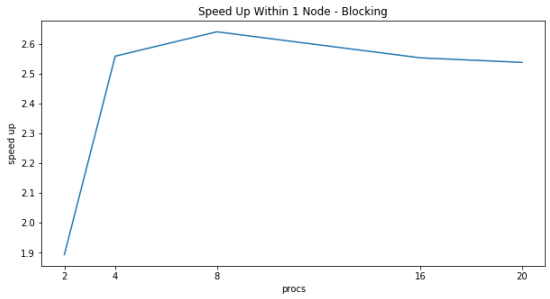
(a) Size $N = 1200$



(b) Size $N = 1200$



(c) Size $N = 12000$



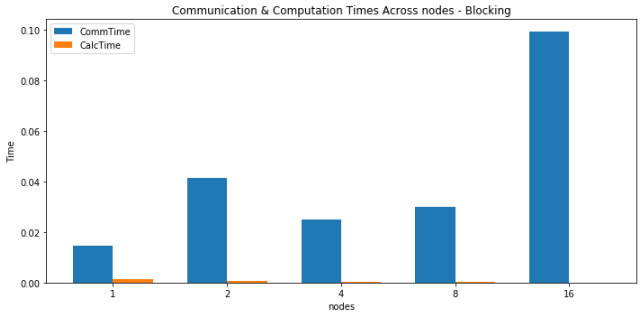
(d) Size $N = 12000$

Figure 1: Within One Node. Left Panel: Communication and computation times on 1 node for non-blocking communication. The yellow bars indicate the calculation time and the blue bars indicate the communication time. Right Panel: Speed up on 1 node for Blocking communication.

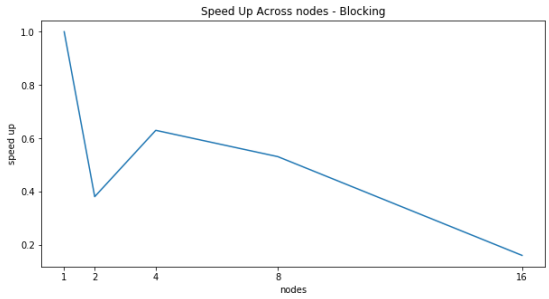
$N = 1200$, as the number of processors increases. Due to this we observe a slight speed up until 16 processors beyond which the speed up falls drastically. While for the size $N = 12000$, the communication time is almost negligible compared to the calculation time. Hence we observe a speed up, though magnitude wise lesser than $N = 1200$ has an increasing trend.

6.1.2 Across nodes

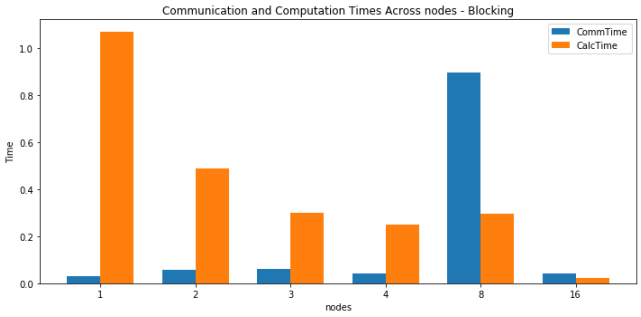
Now, we move beyond a node to observe the scaling behaviour. From fig 2, we can observe that for the size $N = 1200$, there is absolutely no scaling, while for the size $N = 12000$, we see an almost linear scaling, except for a dip at 8 nodes due to excessive communication time. This can be due to the issue where the multithreading option was not turned off while performing the experiment, thereby increasing the waiting time in some processors.



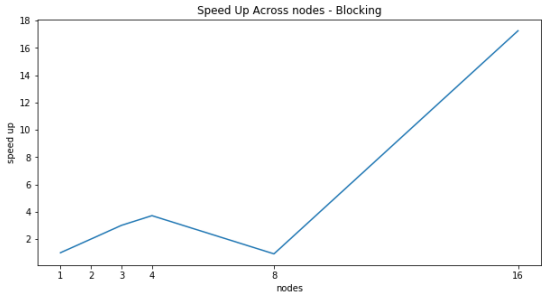
(a) Size $N = 1200$



(b) Size $N = 1200$



(c) Size $N = 12000$



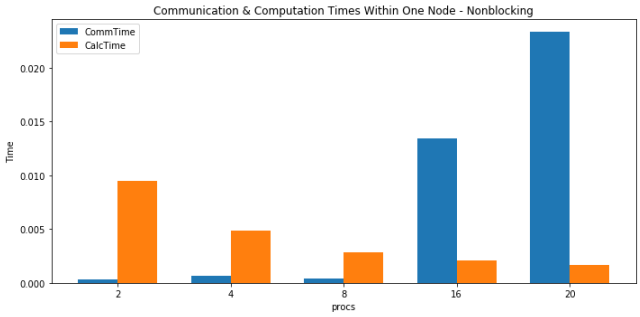
(d) Size $N = 12000$

Figure 2: Across Nodes. Left Panel: Communication and computation times on 1 node for non-blocking communication. The yellow bars indicate the calculation time and the blue bars indicate the communication time. Right Panel: Speed up on 1 node for Blocking communication.

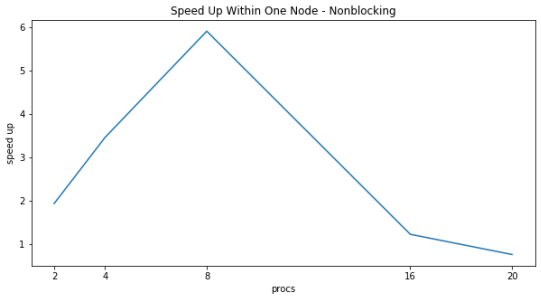
6.2 Non-blocking Communication

6.2.1 Within a single node

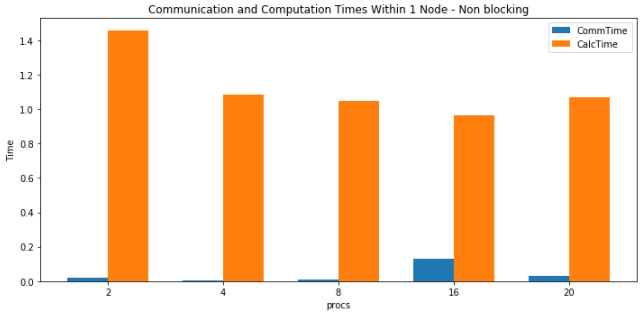
Figure 3 contains the plots for the non-blocking communication where the right panel provide the speed up and the left panel provide the communication and computation times. The top panel is for the size $N = 1200$ and the bottom panel is for $N = 12000$. We see that for the size $N = 1200$, the communication time drastically increases beyond 8 processors while the calculation time keeps reducing leading to a total reduction in the speedup beyond 8 processors. While for the size $N = 12000$, the communication time is negligible compared to the computation time. Due to this, the speed up increases as we increase the number of processors and saturates beyond an extent for a single node. The speed up and the



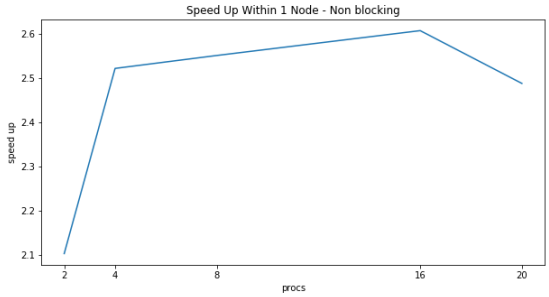
(a) Size $N = 1200$



(b) Size $N = 1200$



(c) Size $N = 12000$



(d) Size $N = 12000$

Figure 3: Within One Node. Left Panel: Communication and computation times on 1 node for non-blocking communication. The yellow bars indicate the calculation time and the blue bars indicate the communication time. Right Panel: Speed up on 1 node for non-blocking communication.

calculation time increases for the size $N = 12000$ and 20 processors as all the processors of a node are utilized due to which the maximum processing speed of each processor is reduced.

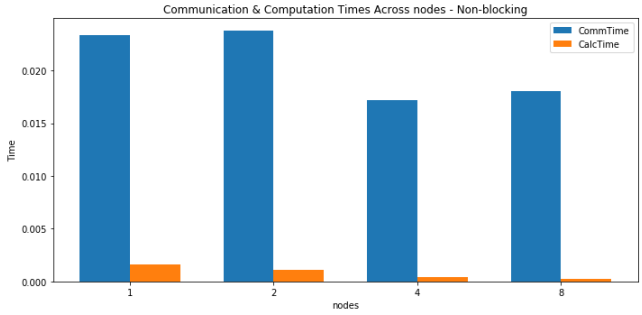
6.2.2 Across nodes

We try to move beyond one node and we go until 16 nodes. From fig 4, we can see that the communication time for the size $N = 1200$ increases or remains almost constant as we increase the number of nodes, while for the size $N = 12000$, the communication time increases very little compared to the the computation time. The computation decreases constantly as we increase the number of processors/nodes, leading to an almost linear speed up.

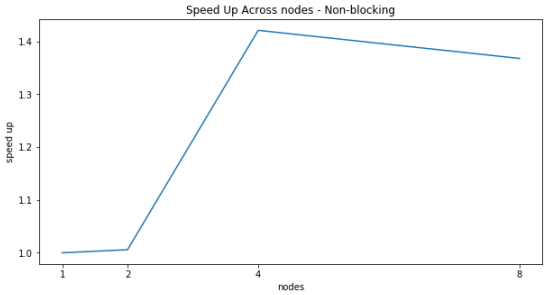
We see that the non-blocking communication for both sizes have a magnitude wise 1.5 times higher speed up. This shows how effective is hiding a part of the communication time can be in reducing the total time taken for the code.

7 Conclusion

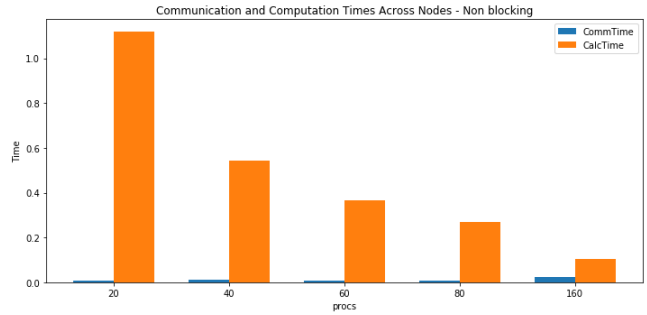
From the exercise, we are able to understand, how the domain size can affect the scaling due to low computation time compared to the communication time. We also understood the



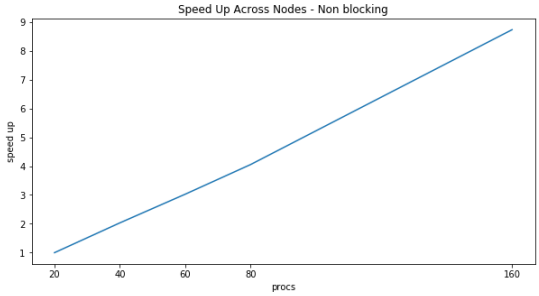
(a) Size $N = 1200$



(b) Size $N = 1200$



(c) Size $N = 12000$



(d) Size $N = 12000$

Figure 4: Across Nodes. Left Panel: Communication and computation times on 1 node for non-blocking communication. The yellow bars indicate the calculation time and the blue bars indicate the communication time. Right Panel: Speed up on 1 node for non-blocking communication.

behaviour MPI blocking and non-blocking communication and how it affects the scaling of the code.