COL380 - Assignment 2

Matrix Multiplication using MPI

Pratheek D'Souza Rebello - 2017CS10361
Sameer Vivek Pande - 2017CS10371

1 Algorithm Description

A summary of our algorithms is given in Figures 1 and 2. Some general finer points:

1. Division of Labour Style: We choose to give B to all processes, and divide the rows of A among them. Thus each process will compute some rows of C. This has been chosen because row wise computation is used in OMP shared memory model to prevent false sharing. Thus we can get an accurate comparison.
2. Number of Rows per Process: We divide the rows into contiguous chunks - as equally as possible. Each will get at least ARows/Number of Processes number of Rows. In addition the first ARows % Number of Processes Processes will get one additional row each. We thus have NO RESTRICTION on n.
3. Process 0 Initialises the matrices and does the distribution and collection of data to others, in addition to computing its own share of C rows.

1.1 Point to Point Communication

1.1.1 Blocking

Here we use MPI_Send() command to send and MPI_Receive() command to receive data. These are blocking calls and so any proccess that calls MPI_Send() must wait till the data is received and acked before it can move on in the code.

1.1.2 Non-Blocking

Here we use MPI_Isend() command to send and MPI_Ireceive() command to receive data. These are non-blocking calls. The process can send data and continue to run its code. When the process expects the correct values to be present e.g. when it needs to malloc space based on received data, it must call a MPI_Wait(). Code will then halt at this MPI_Wait() till acked.

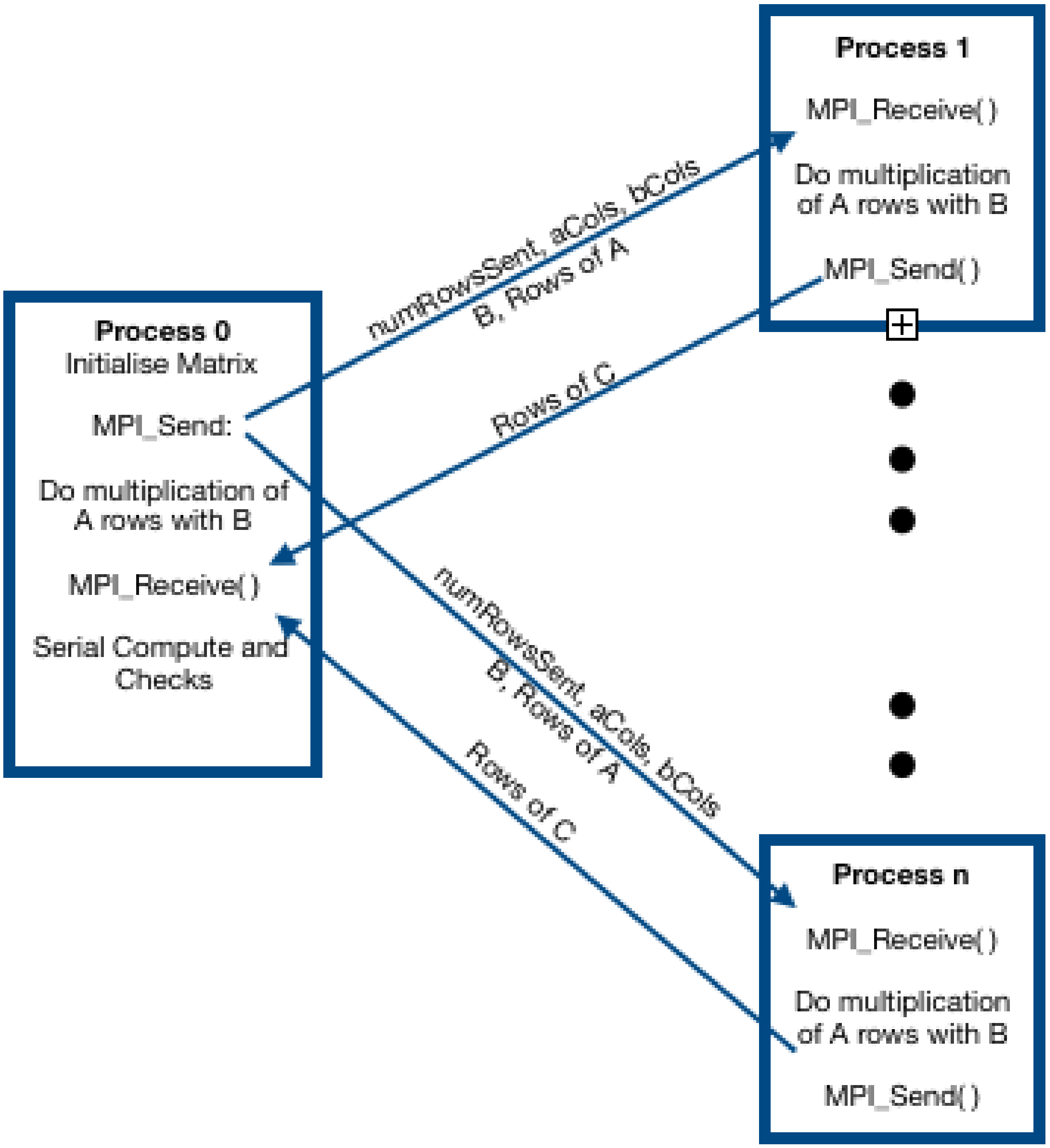

Process 0
Initialise Matrix
MPI_Send:
Do multiplication of A rows with B
MPI_Receive( )
Serial Compute and Checks
numRowsSent, aCols, bCols
B, Rows of A
Rows of C
numRowsSent, aCols, bCols
B, Rows of A
Rows of C
Process 1
MPI_Receive( )
Do multiplication of A rows with B
MPI_Send( )
Process n
MPI_Receive( )
Do multiplication of A rows with B
MPI_Send( )

Figure 1: Point to Point Algorithm

1.2 Collective Communication

1. We use broadcast to send dimensions of A and B matrices, and B matrix. MPI_Bcast() ensures all data is sent to all processes.
2. We use scatter to send chunks of rows to each process. Scatter(A, number of elements) will give each process corresponding number of elements of A. Since our

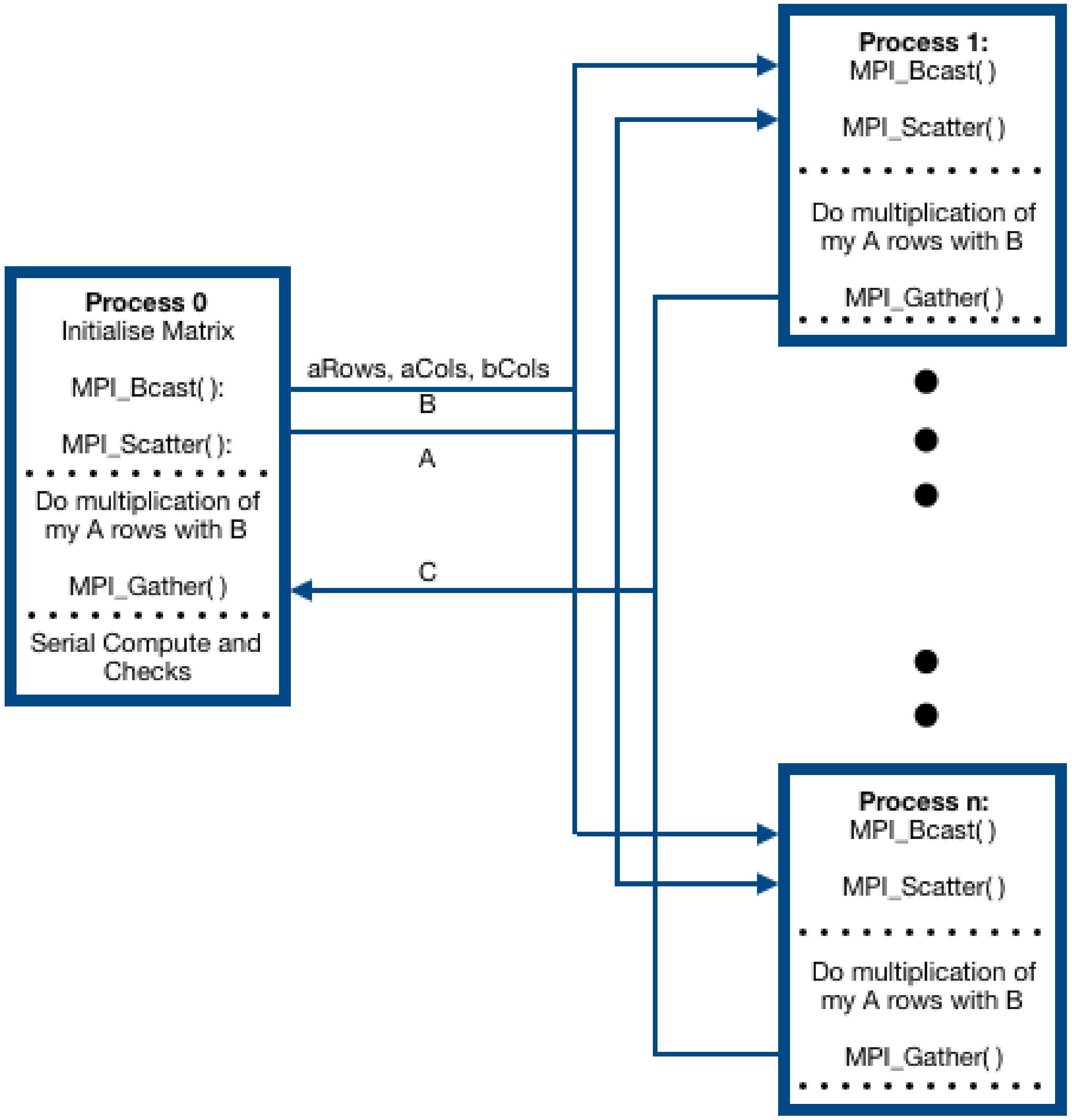


Figure 2: Collective Communication Algorithm

division of rows may not be uniform, we pad A matrix with additional rows of 0s.

3. We use gather to collect the computed rows of C in Process0. Gather(C, number of elements) will collect corresponding number of elements of C from each process.

2 Experimentation

1. In collective communication we tried two approaches - one using scatter and gather to send the rows of A and collect rows of C respectively. Other approach used MPI_Send and MPI_Recv to send the rows of A and receive the rows of C. We found using scatter and gather was faster
2. In case of blocking and non blocking, we tried to send the index number of starting row for each processor and sending it back to process 0. In other approach we tried to keep the track of these numbers in the parent process itself. The latter approach turned useful because of communication took more time.
3. We have tried to minimize the number of waits in non-blocking communication by reordering our code to send independent values together and using the dependant code at once.
4. We tried to use O3 flag which reduced the run time of our code by 2-10x in most cases. But since the compiler may perform unknown optimizations we cannot use that to compare various mechanisms for our analysis.

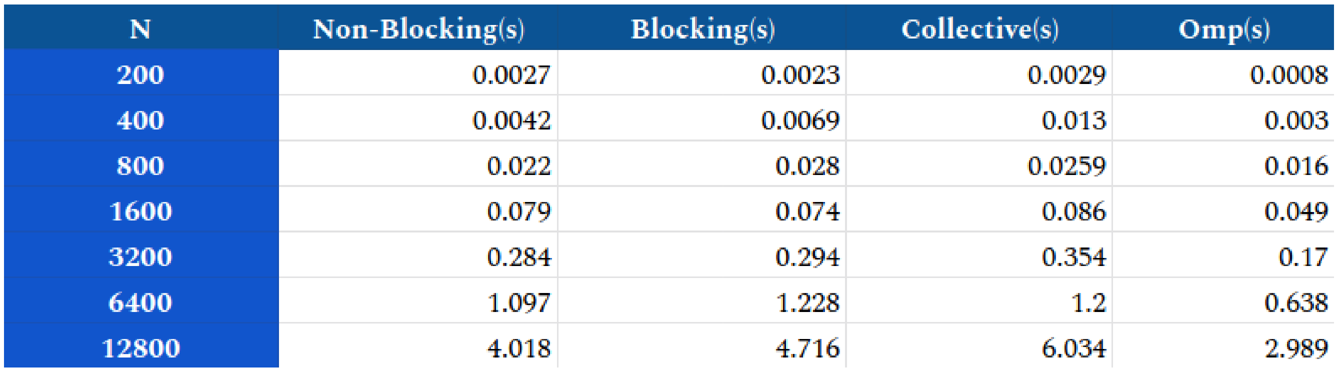
3 Observations

We have run our code on an Intel Core i7 9th Gen Processor with 12 Cores.

3.1 Time v/s Matrix Size

Explanations:

1. Matrix multiplication is O(Arows x Acols x Bcols), but Acols = 32 for all our cases, while Arows = Bcols = n. Thus we have an O(n^2) algorithm. All time v/s n graphs are parabolic, and time v/s n^2 graphs are approximately straight lines. This confirms the theoretical complexity of the algorithm.
2. 1 Process: All methods work approximately in the same time, providing us with a sanity check that OMP and MPI on a single processor, both do the same thing.



N	Non-Blocking(s)	Blocking(s)	Collective(s)	Omp(s)
200	0.0027	0.0023	0.0029	0.0008
400	0.0042	0.0069	0.013	0.003
800	0.022	0.028	0.0259	0.016
1600	0.079	0.074	0.086	0.049
3200	0.284	0.294	0.354	0.17
6400	1.097	1.228	1.2	0.638
12800	4.018	4.716	6.034	2.989

Figure 3: Time Observations for 8 Processes

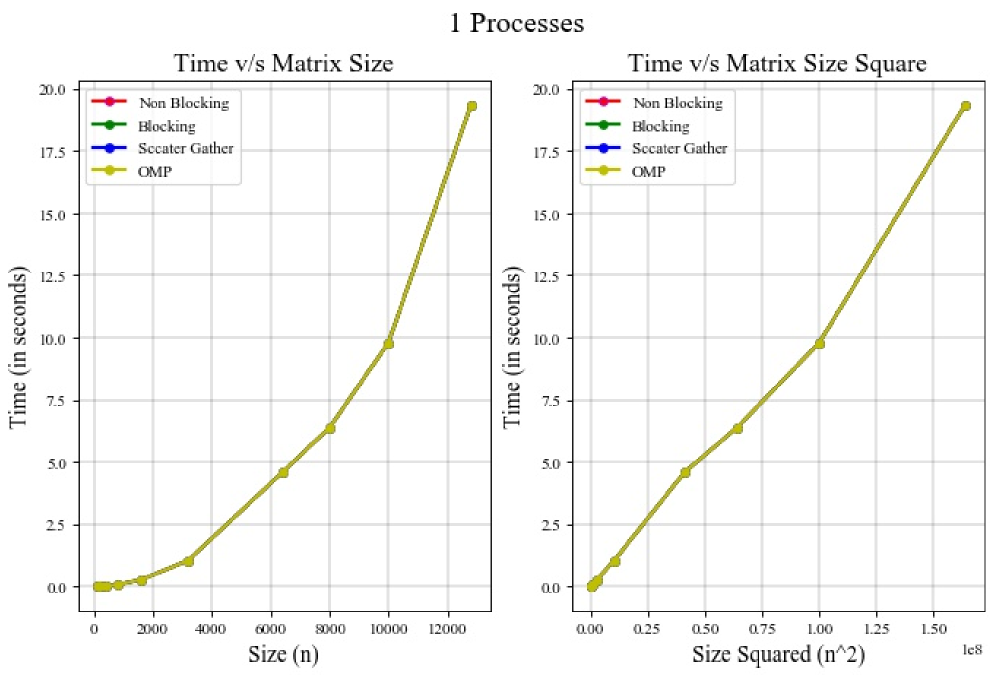


Figure 4

3. Multiple Processes: OMP universally does better than MPI, confirming that shared memory model works faster than distributed memory model.

4. Among the different MPI conventions, we see that Non Blocking is the fastest. It does not require us to halt code execution after sending some data and allows the process to do some independent computation while the data is being transferred.

5. Collective and Blocking show competing trends. On fewer processes blocking is faster than collective, whereas on 12 and 16 processes, collective is faster. This makes sense since as we increase the number of processes, the number of blocking calls goes on increasing, and communication overheads slow down the code.

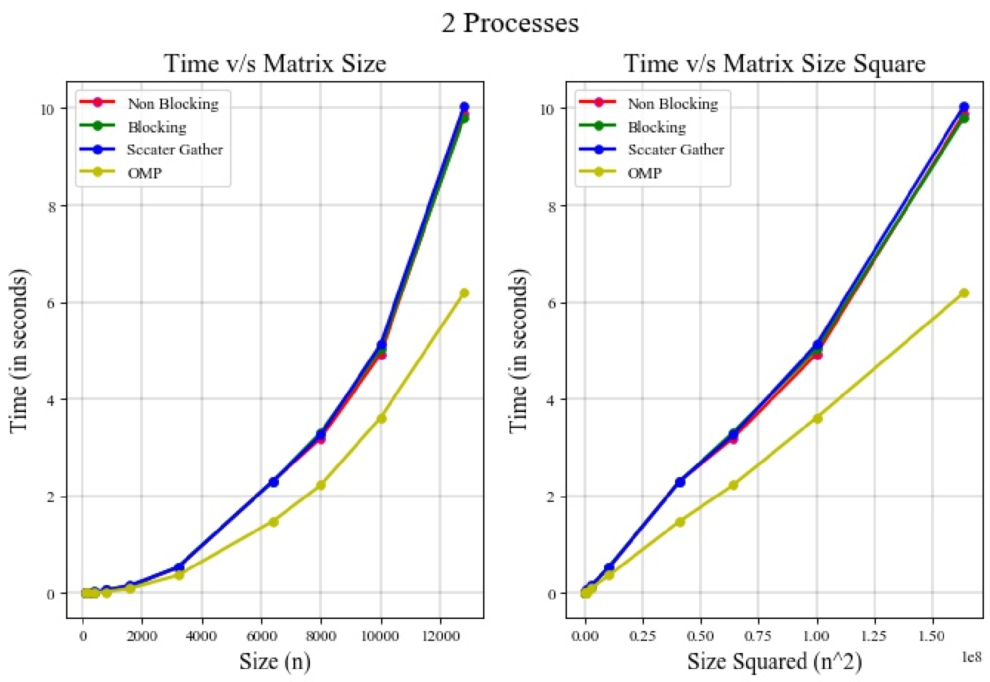


Figure 5

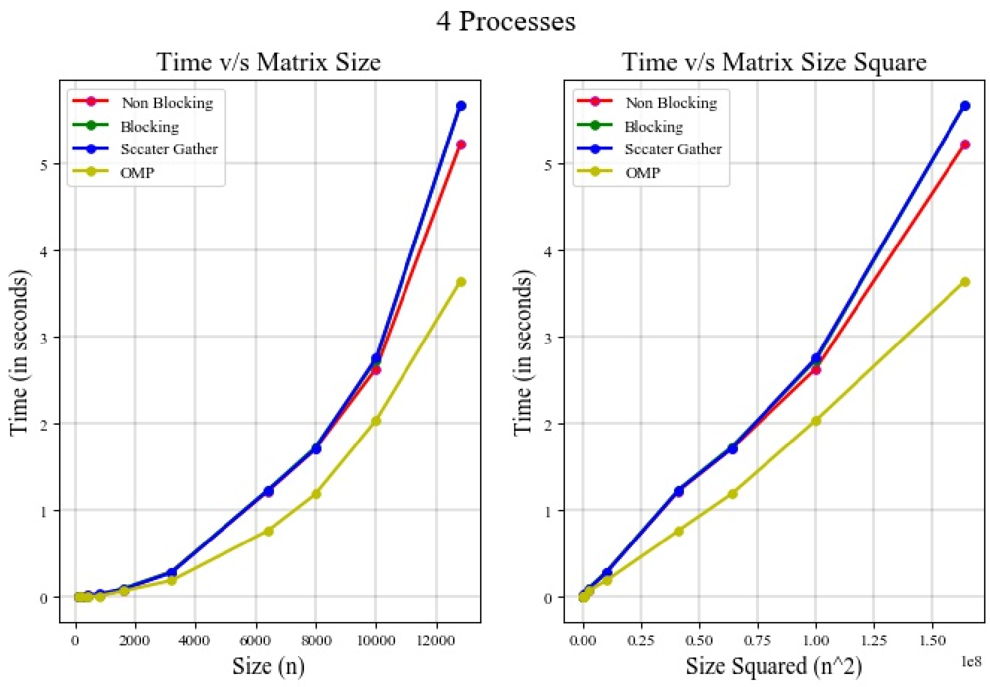


Figure 6

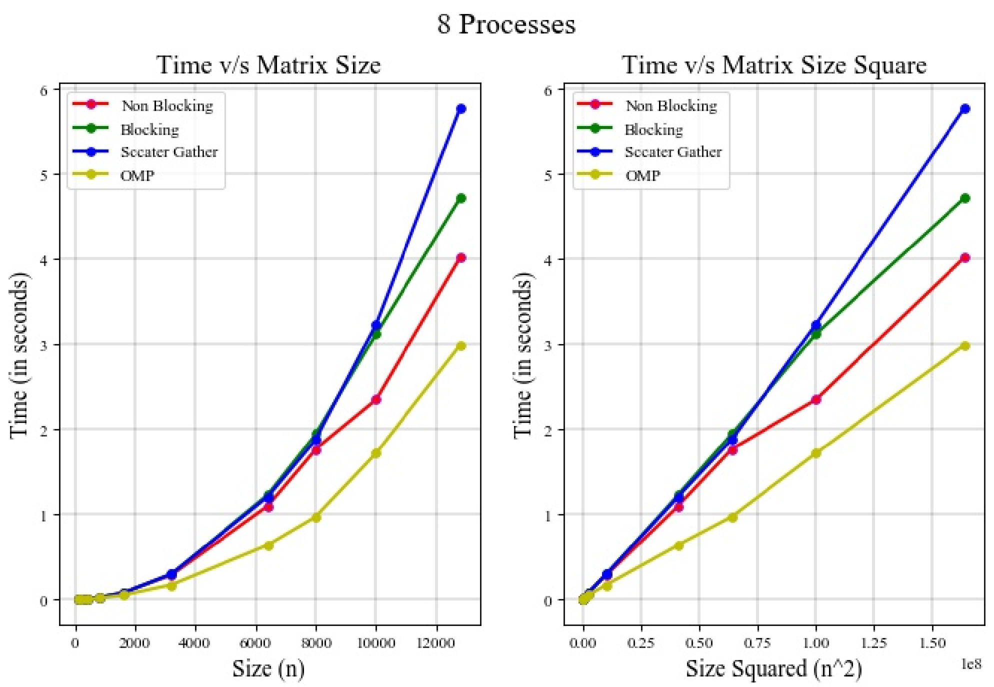


Figure 7

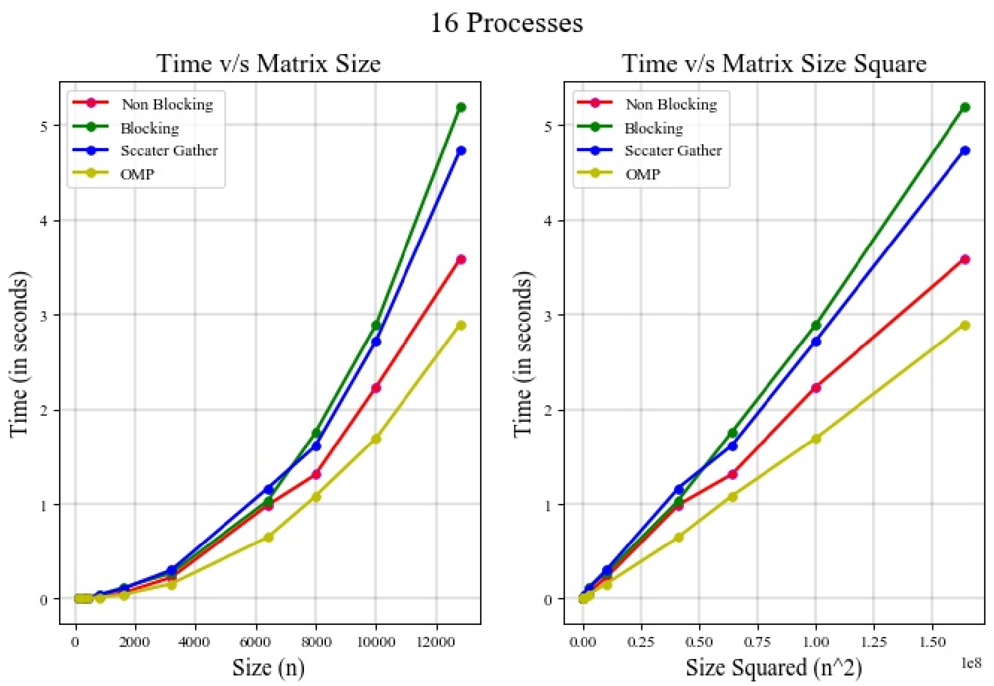


Figure 8

Scatter Gather (Collective) is not so sensitive to number of processes.

6. Our code is independent of the type of matrix - sparse or dense.

3.2 Time v/s Processors

Processes	Non-Blocking (s)	Blocking (s)	Collective (s)	Omp (s)
2	3.19	3.3	3.45	2.22
4	1.71	1.73	1.81	1.19
6	1.24	1.27	1.35	0.83
8	1.76	1.94	1.96	0.97
12	1.27	1.32	1.42	1.03
16	1.31	1.76	1.88	1.08

Figure 9: Time Observations for n = 8000 Matrix Size

Explanations:

1. Small Matrix size: We see an approximate speedup trend as expected, but the graphs are highly noisy due to small number of computation, compared to communication.

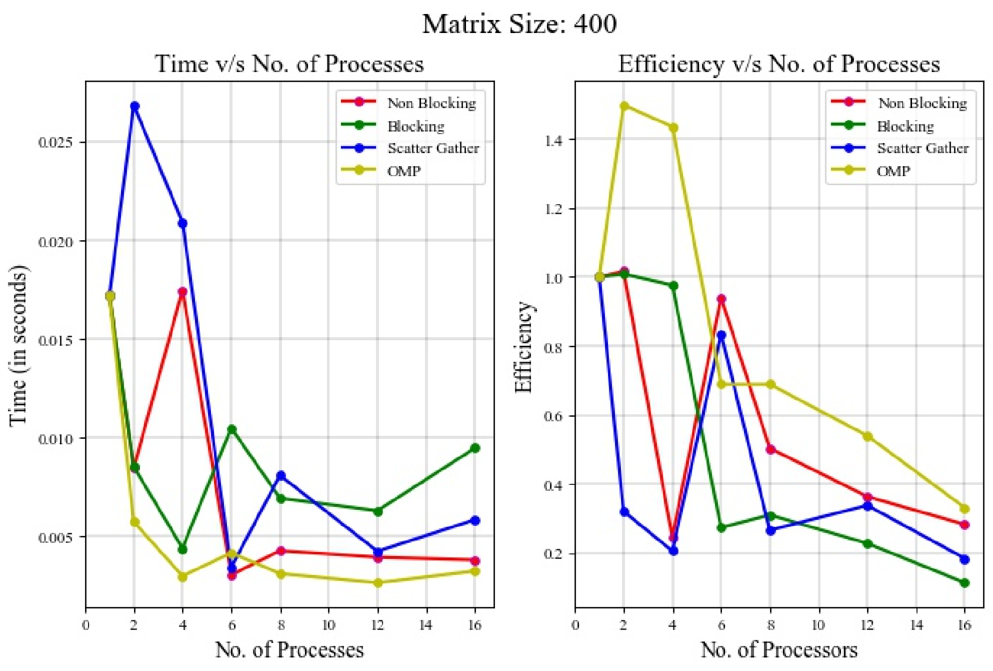


Figure 10

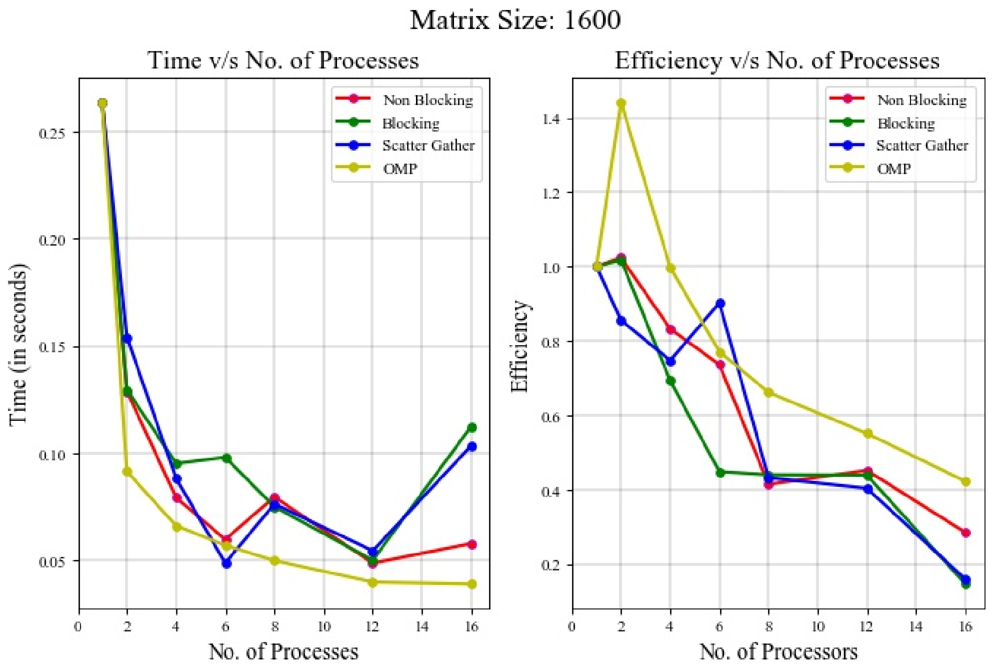


Figure 11

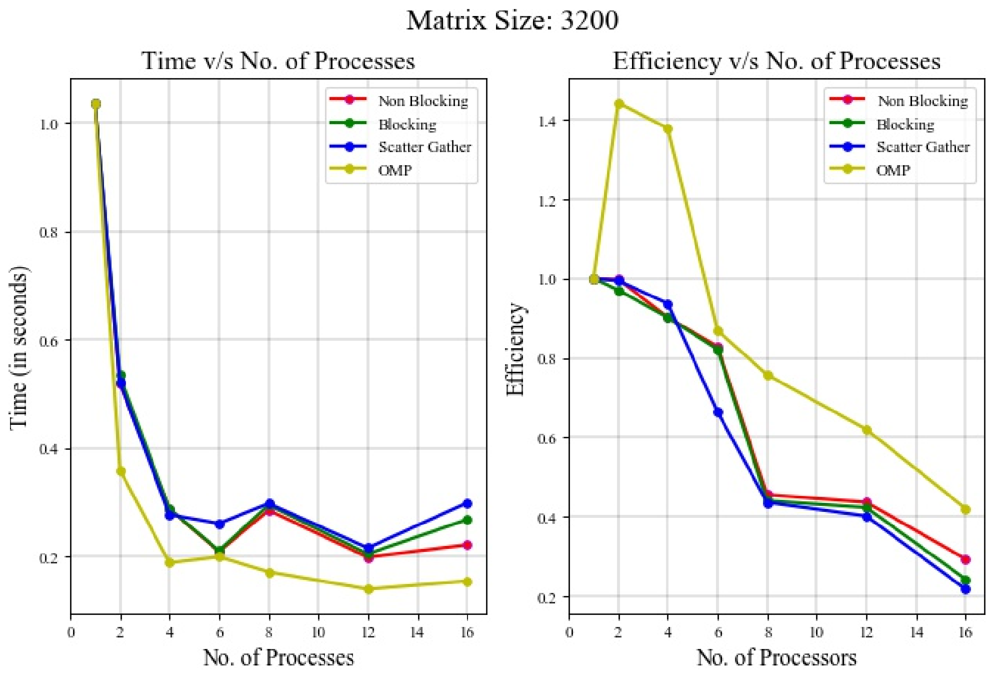


Figure 12

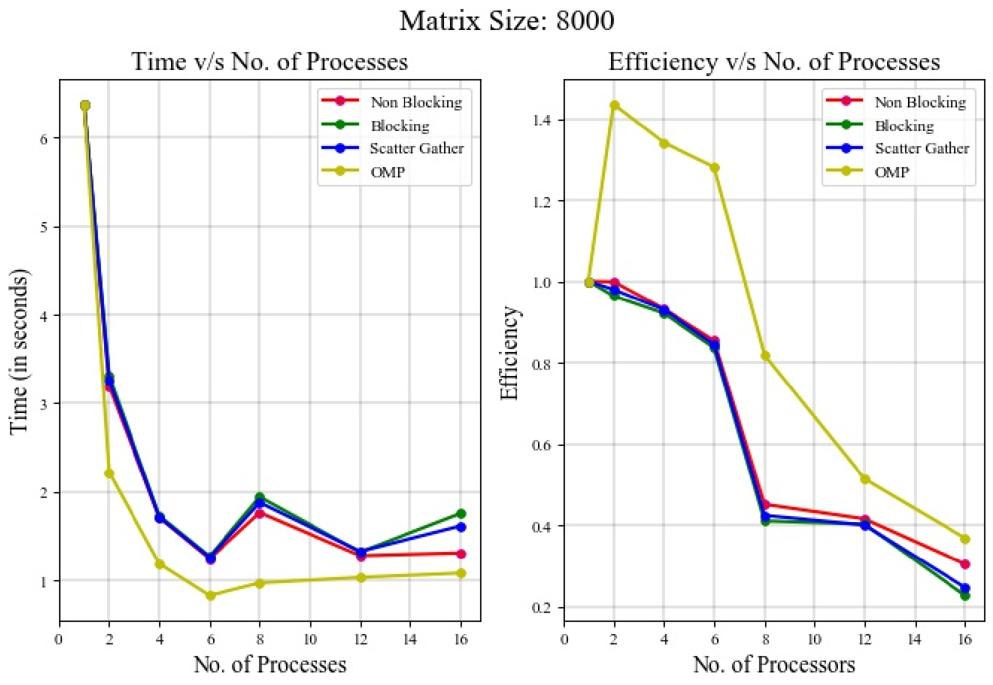


Figure 13

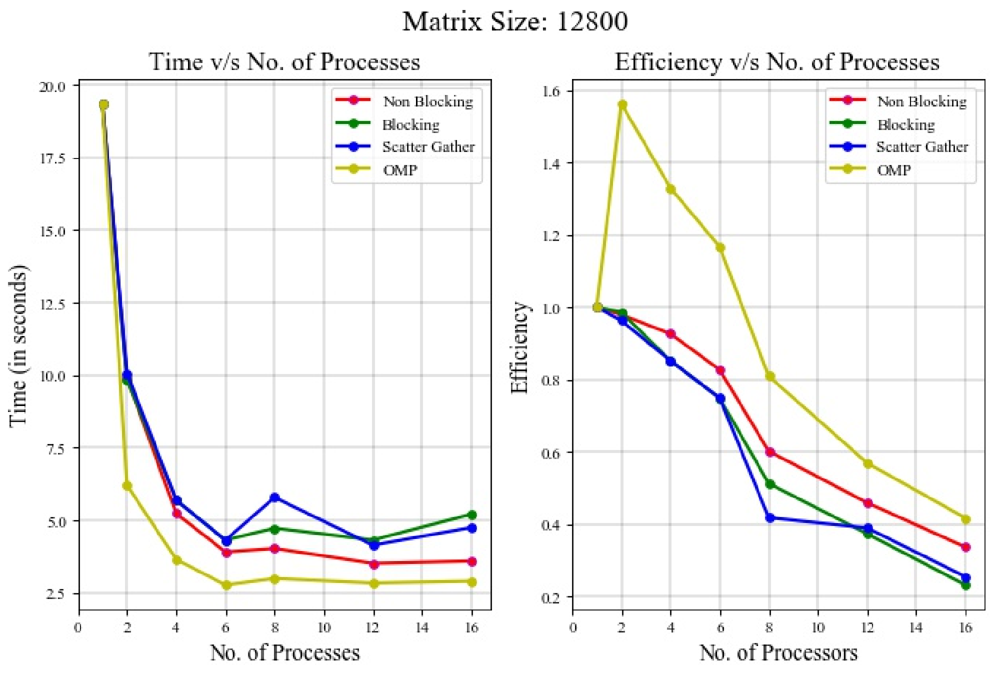


Figure 14

2. Smoother graphs are observed from size   1600.
3. OMP is universally the best, followed by non-blocking for the same reasons as discussed above.
4. We do not see any firm trend between blocking and collective communication, and conclude that we cannot make any assumptions about their comparative performance.

4 Discussion and Conclusion

Overall we verify by experiment our intuition that shared memory model (OMP) is faster than distributed memory model (MPI). And within distributed memory model, non-blocking P2P communication works fastest.