Assignment 1; COL380

Priyanshu Gautam, 2017CS10362 Rachit Kumar, 2017CS10364

Abstract

In this report we present the rationale behind our codes and try to compare the different times each of the non-parallel and parallel(`MPI` with `blocking`, `non-blocking` and `collective`) programs take in order to compute the LU Decomposition of square matrix(of size $n = 8000$).

1 Serial Code

The serial code is simple $\mathcal{O}(n^2)$ (since 32 is fixed) algorithm of matrix multiplication. Simple multiplication with 3 loops as given on the course website. Since there are two loops over n, it is clear that the algorithm is $\mathcal{O}(n^2)$.

2 MPI Algorithms

2.1 Blocking

Blocking type of communication happens when until the message is successfully passed or received, the processes are *blocked* or they are non working. This inherently implies that the blocking type of communication would be slower as nothing else can happen in the background. Our algorithm can be understood from the diagram.

No additional waits are required as it stops the process altogether until the communication is completed. We follow the instructions given in the assignment statement. We use the process 0 to initiate the matrix and send the matrix to the rest of the processes using `MPI_Send` function, which is blocking by nature. Process 0 sends $\frac{1}{p}$ part of A and all of B to each process. We realise that sending B again and again might be an overkill and an overhead. And we use `MPI_Recv` function to receive all the fragments. Then each process performs its fraction of the parallel computation. Theoretically, the computation time should go down by $\frac{1}{p}$ where p is the number of processes spawned. But we must also account for communication time. After the multiplication is done, each process sends its computed part back to process 0. And the algorithm is completed.

2.2 Non Blocking

It uses the same algorithm and sequence of message passing. But since it is non blocking, it is very well possible that it may execute an instruction which depends upon the data procured from data

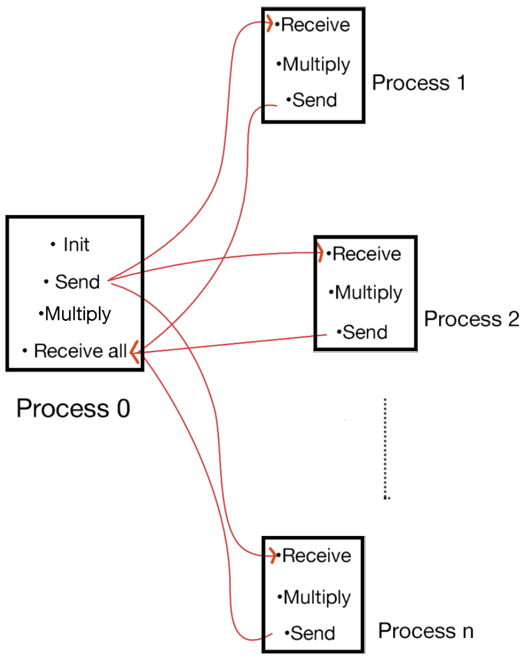


Figure 1: Blocking Algorithm

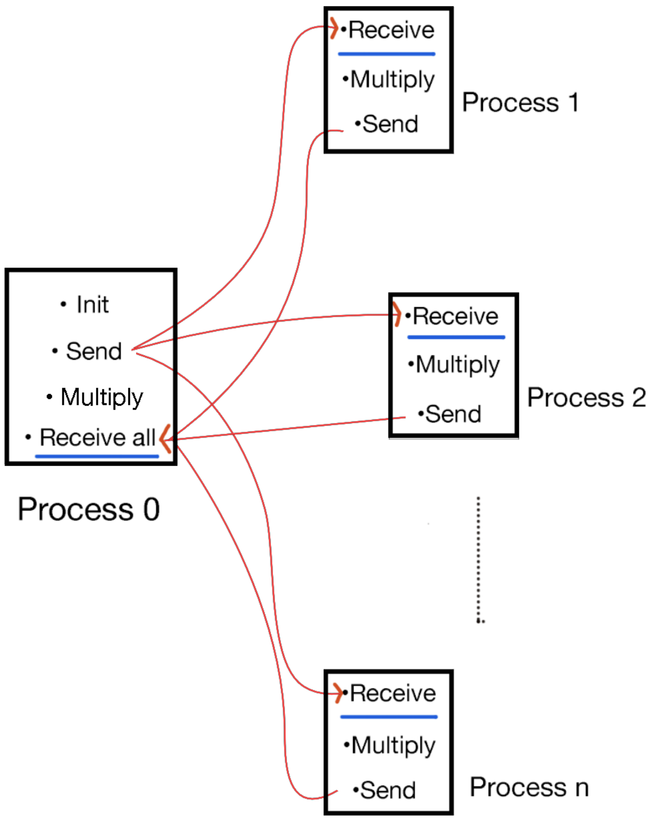
communication. Therefore, we must ensure that communication has completed. To facilitate this we make use of `MPI_Wait` procedure. In order to make the most efficient algorithm, we must use as few `MPI_Waits` as possible. The way we have used `MPI_Wait` can be seen in the figure. Blue dashes represnt wait statements.

2.3 Collective

In this, we start with broadcasting the matrix dimensions to the workers via MPI_Bcast. The second step is to broadcast the matrix data to the workers. Each worker computes its own "matrix area" with the mpi rank and performs the matrix multiplication accordingly. The third step is to collect the data via MPI_Gather and combine it into the final matrix.

3 Modifications

We have reduced the number of waits in non blocking part to maximise. Though on trying to minimise, waiting time may increase if we introduce wait at the wrong places which happened. It has to be chosen carefully otherwise program might be incorrect.



blocking.png

Figure 2: Non-Blocking Algorithm

4 Experimentation

We use three different versions of the programs: `B_P2P.cpp`, `NB_P2P.cpp` and `collective.cpp` implementing *blocking*, *non-blocking* and collective types of communication. We run each of these program using sizes of $n \times 32$ and $32 \times n$. The processor used is *2.3 GHz Dual-Core Intel Core i5* from a Macbook Pro 2017 Model. A random matrix is generated using `rand()` function of C++. We vary n and p. We also compare normal and all optimisation flags. We also make an analysis over sparse matrix. We also make a general analysis over $n \times 2^k$ and $2^k \times n$ sized matrices. The way we have divided the matrices for IPC, is based on the row major method in order to reduce false sharing and maintain cache coherence(not distributing over various cache lines).

5 Results

The results are summarised in the following subsections:

5.1 Changing n for a fixed technique

Since our machine is dual core, we shall do it for number of processors being 2 and 4.

1. Blocking

Size of matrix	Number of processes	Total Parallel Time(ms)	Communication	Serial Time(ms)
10	2	0.236	0.214	0.016
50	2	0.922	0.735	0.297
100	2	1.672	1.002	1.386
200	2	3.320	1.010	4.681
500	2	16.628	1.104	30.104
1000	2	67.821	2.055	112.71
2000	2	246.07	7.429	457.6
4000	2	982.4	25.098	1843
10000	2	6193	112.8	11919
20000	2	266162	1570.7	49986

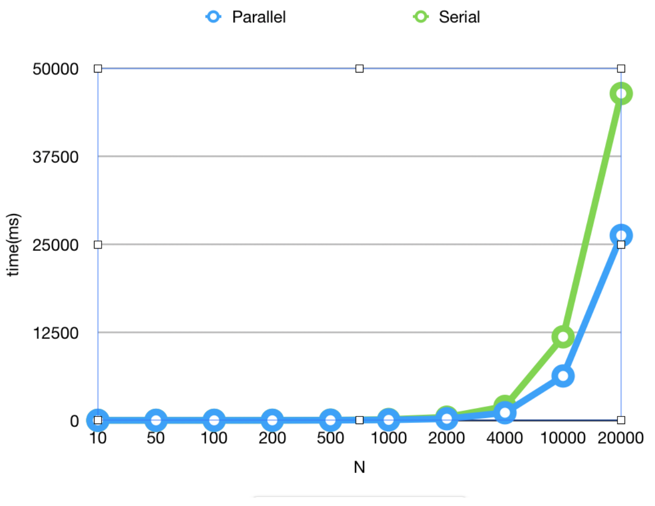


Figure 3: Blocking Algorithm:p=2

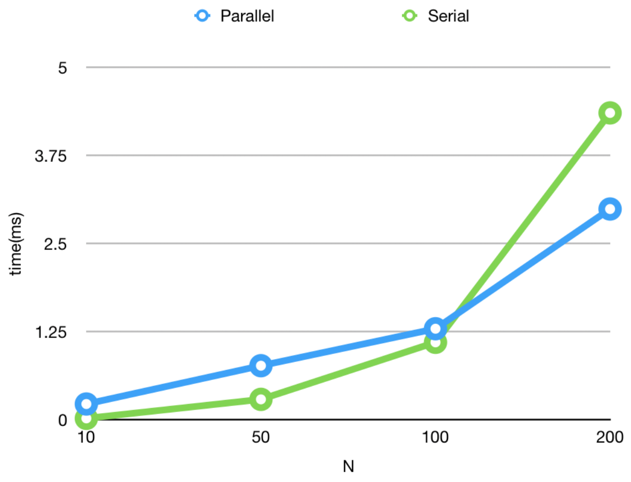


Figure 4: Blocking Algorithm:p=2

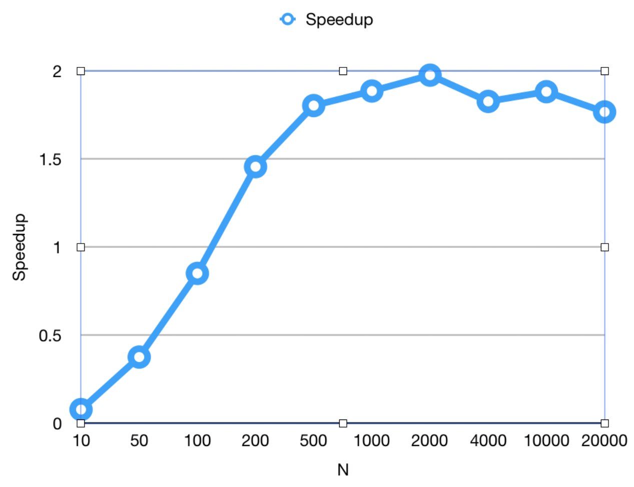


Figure 5: Blocking Algorithm:p=2

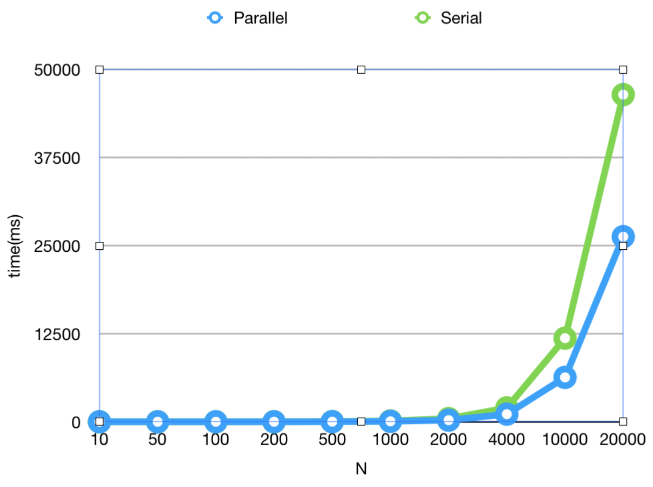


Figure 6: Blocking Algorithm:p=4

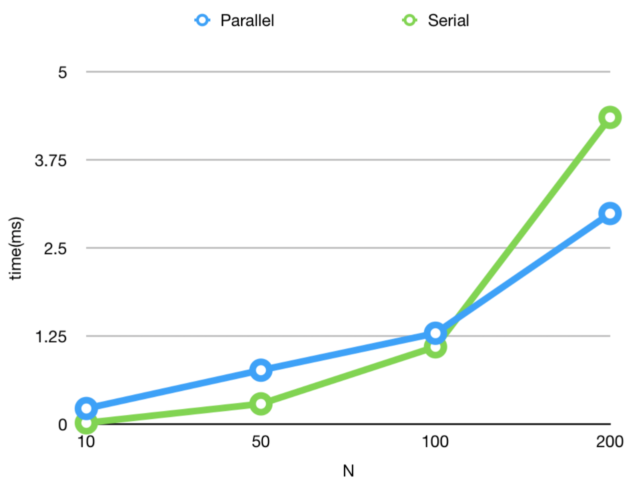


Figure 7: Blocking Algorithm:p=4

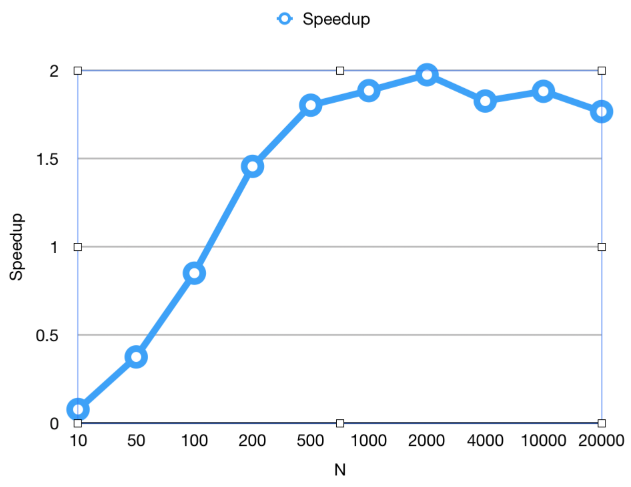


Figure 8: Blocking Algorithm:p=4

2. NonBlocking

Size of matrix	Number of processes	Total Parallel Time(ms)	Communication	Serial Time(ms)
10	2	0.188	0.161	0.012
50	2	0.592	0.436	0.281
100	2	1.497	0.934	1.082
200	2	4.923	2.723	4.359
500	2	28.131	14.434	27.312
1000	2	111.028	56.007	109.599
2000	2	456.18	233.08	439.27
4000	2	1804.854	907.066	1766.3
10000	2	11295.48	5644.4	11051
20000	2	45894	23086	46770

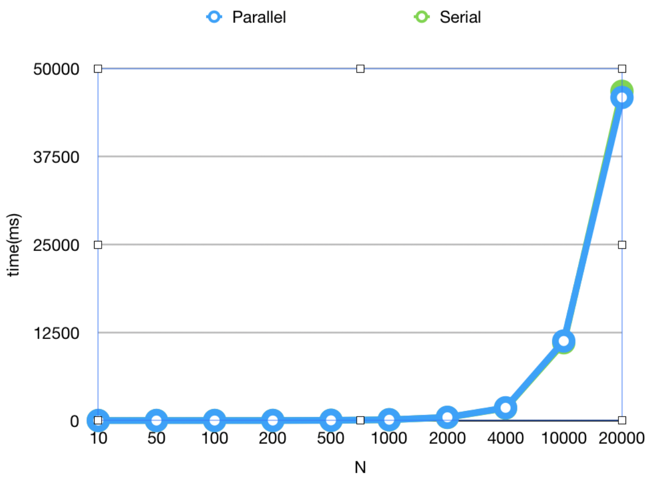


Figure 9: Non-Blocking Algorithm:p=2

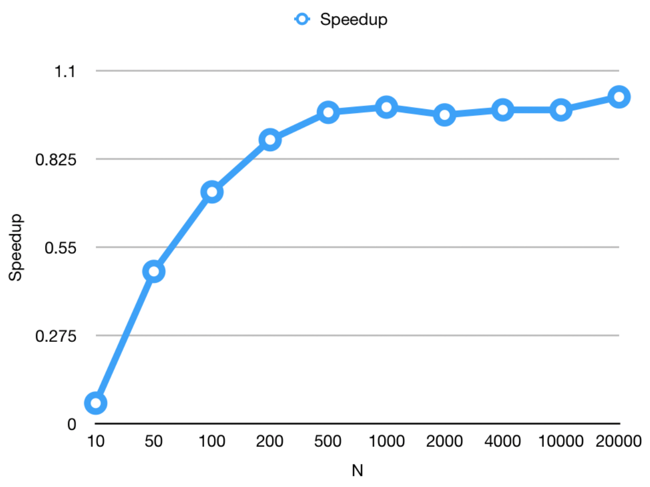


Figure 10: Non-Blocking Algorithm:p=2

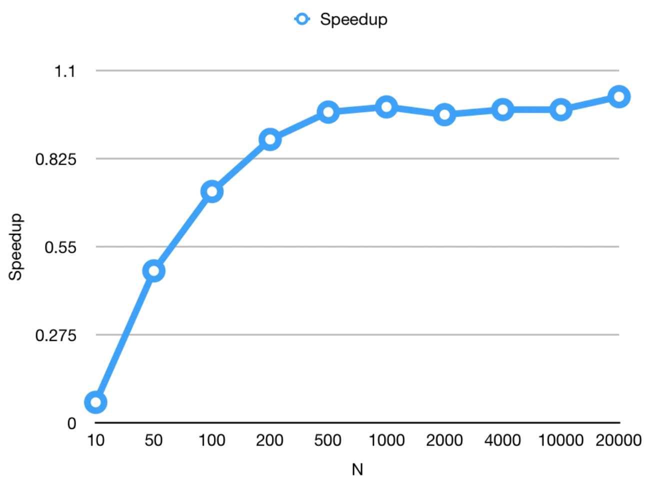


Figure 11: Non-Blocking Algorithm:p=2

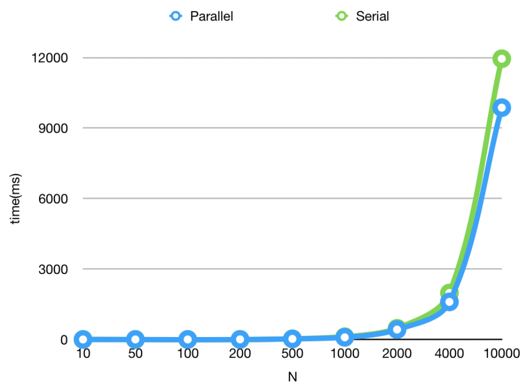


Figure 12: Non-Blocking Algorithm:p=4

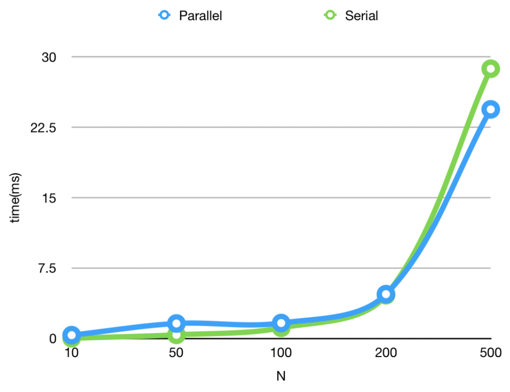


Figure 13: Non-Blocking Algorithm:p=4

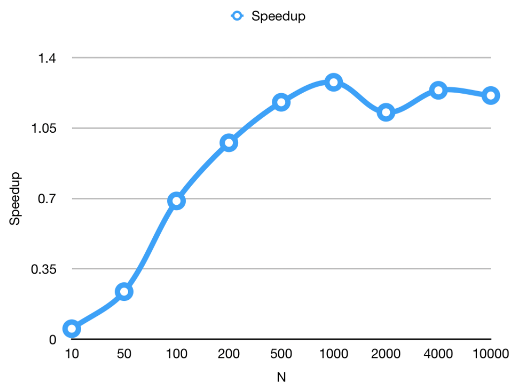


Figure 14: Non-Blocking Algorithm:p=4

3. Collective

Size of matrix	Number of processes	Total Parallel Time(ms)	Communication	Serial Time(ms)
10	2	0.295	0.269	0.018
50	2	0.642	0.51	0.286
100	2	1.307	0.848	1.11
200	2	2.753	0.959	4.39
500	2	13.225	1.607	28.215
1000	2	51.21	4.286	115.681
2000	2	223.4	15.746	466.718
4000	2	1038.9	86.713	1824.673
10000	2	5005.8	337.38	11320.477
20000	2	22927	3272	47526.882

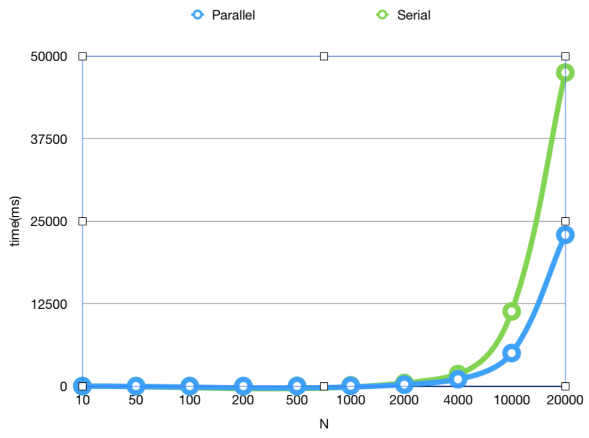


Figure 15: Collective Algorithm:p=2

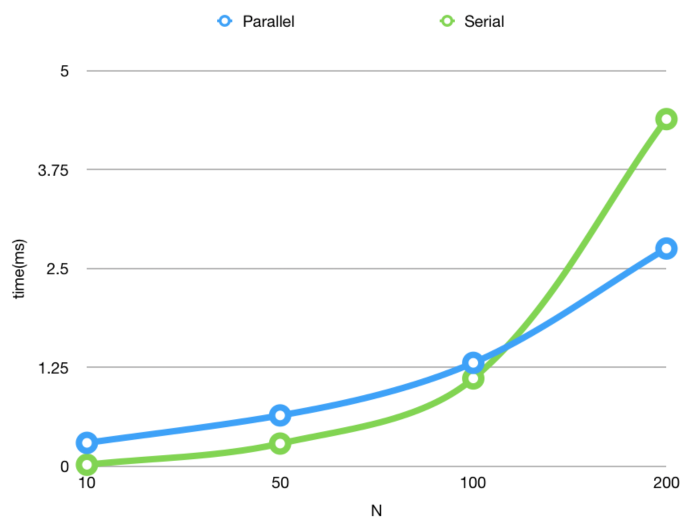


Figure 16: Collective Algorithm:p=2

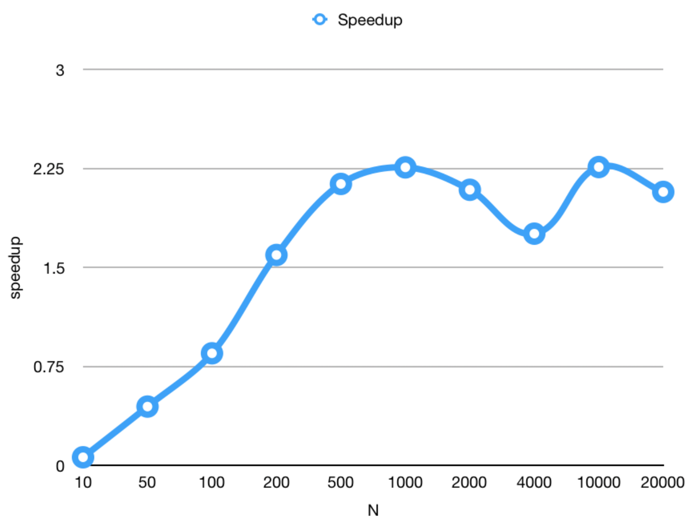


Figure 17: Collective Algorithm:p=2

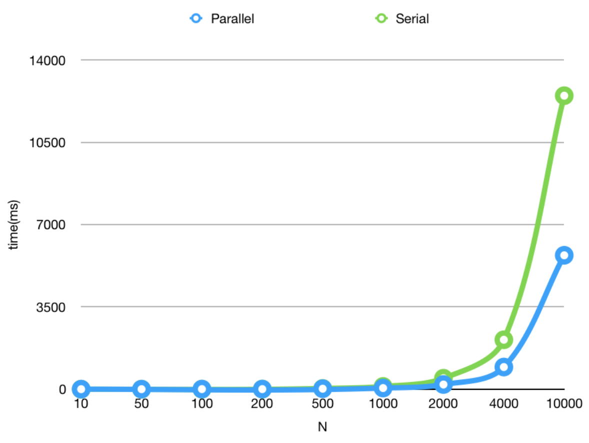


Figure 18: Collective Algorithm:p=4

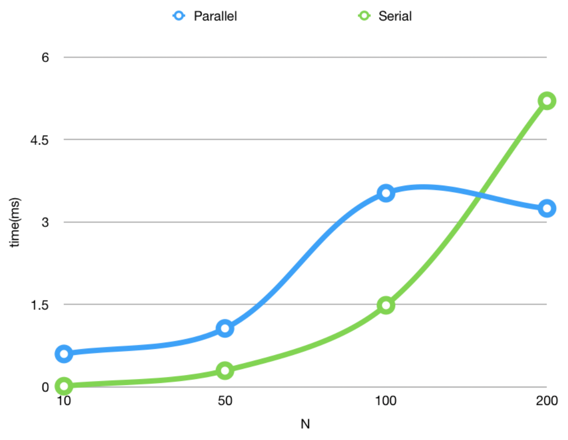


Figure 19: Collective Algorithm:p=4

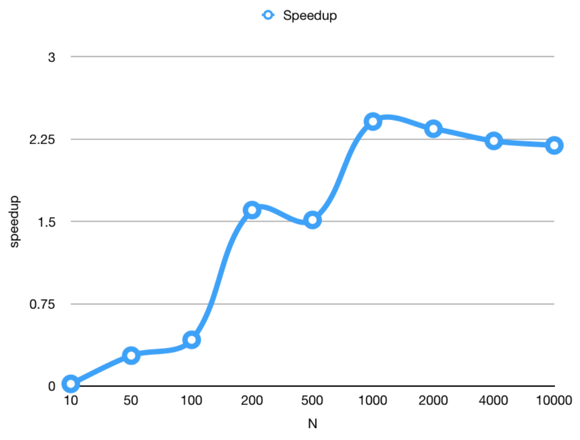


Figure 20: Collective Algorithm:p=4

5.2 Comparison of various techniques

Number of processes	Blocking Speedup	Non-Blocking Speedup	Collective Speedup
2	1.8863	0.9871	2.2589
4	2.2642	1.2767	2.4095
8	2.1206	1.6471	2.2466

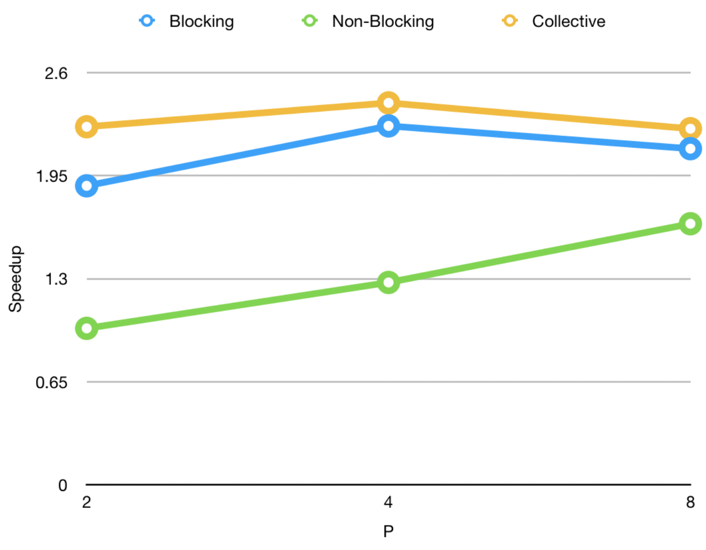


Figure 21: Speedup

5.3 Using $n \times 2^k$ and $2^k \times n$ matrices

Blocking

n	m	Number of processes	Speedup
1000	32	2	1.87
1000	64	2	1.93
1000	128	2	1.95
1000	256	2	1.89
2000	32	2	1.93
2000	64	2	1.97
2000	128	2	1.98

Collective

n	m	Number of processes	Speedup
1000	32	2	2.27
1000	64	2	2.18
1000	128	2	2.14
1000	256	2	2.14
2000	32	2	2.30
2000	64	2	2.26
2000	128	2	2.18

5.4 Analysis over sparse matrices

The results remain the same

6 Inferences

We will try to summarise the major takeaways and the tables.

6.1 Comparison over n

Blocking We see that for $n \leq 50$, `serial multiplication` is faster by magnitudes of ≈ 4 *to* 10. While as we get beyond 500, we see the efficiency getting saturated to ≈ 2. This can be attributed to higher cost of communication for small values of n. This communication time is the bottleneck for small n, while as we see as n is increased, speedup reaches 2.
Collective Collective performs the best in terms of speedup, as it neither requires waits, and it also uses less data over communicating channels. The efficiency of collective can be seen in the less fraction of communication times. **Non blocking** Non blocking is not faster, is actually slower than serial multiplication on our machine. This is **not expected** and **anomalous** behaviour. This can be attributed to different IPC protocols on mac, triggering wait after a very long time.

6.2 Comparison over number of processes

As our machine is a dual core, maxima happens at 2 processes and it is downhill from there. There is no improvement in parallel time, keeping speed up almost same but decreasing efficiency. As the machine is dual core, for $p > 2$, it uses its two processes turn by turn. Therefore the lost time is clearly visible in the communication time, as seen in our tables.

6.3 Sparse matrices

There was no change when dealing with sparse matrices

6.4 Comparison of various techniques

We observe that the fastest technique observed is `collective`. This is because we do not overdo any transmission like in `blocking`. `non-blocking` is the slowest even though it should have been the fastest. This can be attributed to different IPC protocols on our machines. We see that across number of processes, the best speed up occurs at $p = 4$, while efficiency is highest at $p = 2$.

6.5 Using $n \times 2^k$ and $2^k \times n$ matrices

We observe that moving along higher k, speedup of blocking increases while collective decreases. Higher speed ups can be attributed to a better trade-off between serial and parallel. While, for collective, since we are broadcasting therefore as m increases the trade-off of m increases.

6.6 Comparison of communication times

We see that for $p = 2$, the communication times are not excessively increasing. For smaller values of n, communication time forms a major portion of parallel algorithm. As we can see in the table. For larger values, they are negligible. Collective takes the least amount of communication time compared to non-blocking taking the most. For $p = 4$, and so on, communication times form major portion as cores are selected on round robin.

6.7 Result with optimisation flags

With `O1` flag, the speedup becomes double. This is surprising as it means for $p = 2$ speed up becomes ≈ 4.5 implying a reduction in actual time taken. This can be attributed to `O1` flag not being able to optimise serial code as efficiently as the parallel one. This can be seen from our data that with `O1` flag, the code is faster for matrices of smaller sizes(Almost twice), which leads to code getting much faster for parallel implementation. With rest of the flags, speed up actually decrease from non optimised code.