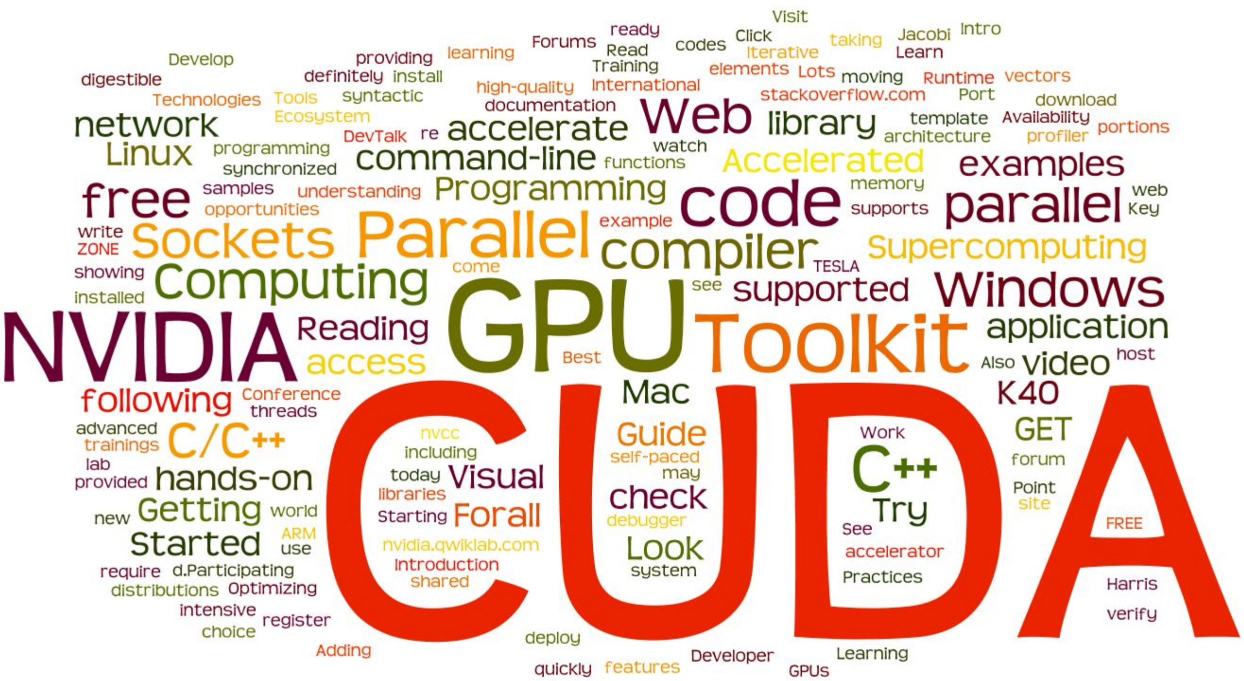
Matrix Multiplication

Introduction to Parallel Programming



Prateek Garg

Sahil Dahake

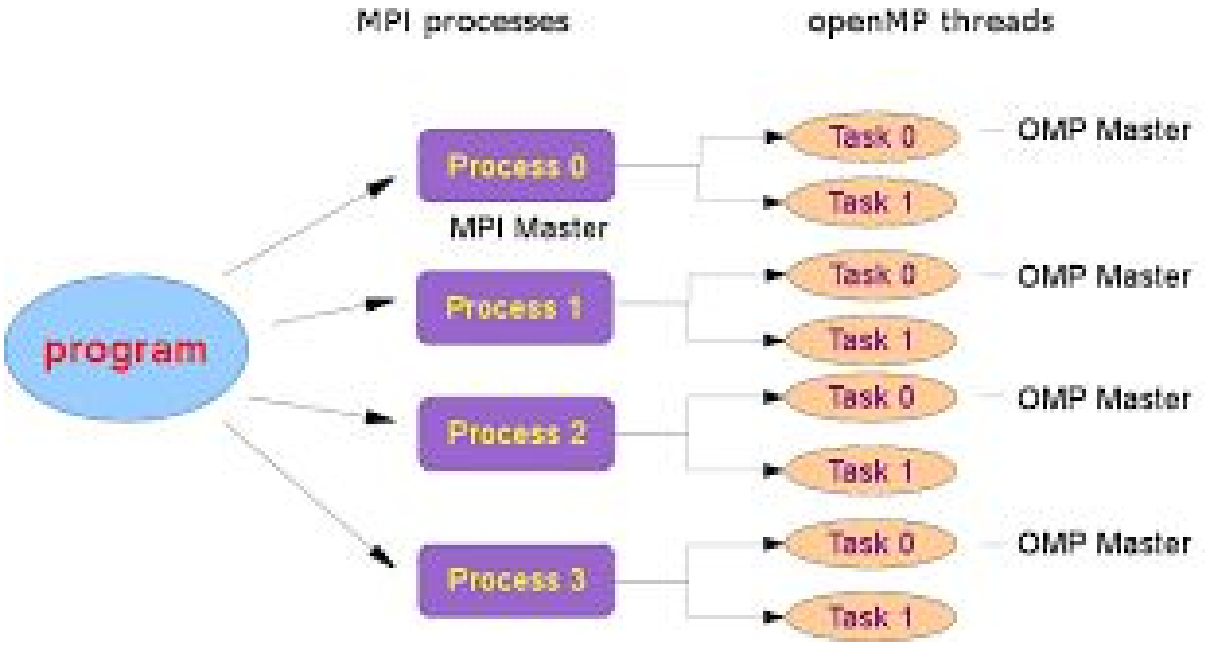
28.02.2020

2017CS10360

2017CS50488

INTRODUCTION

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.



Code Explanation :- Blocking P2P (point-to-point) communication

1. There is a restriction of N in the program, it can't be more than 10,000 since I was getting a segmentation fault if I tried to allocate more memory using malloc to initiate the matrices.

```
a.  int N = 10000;
b.          int size;
c.          double* A = (double*) malloc(32*N*sizeof(double));
d.          double* B = (double*) malloc(32*N*sizeof(double));
e.      double* answer = (double*) malloc(N*N*sizeof(double));
f.          double* C = (double*) malloc(N*N*sizeof(double));
g.      double* answer_serial = (double*) malloc(N*N*sizeof(double));
```

2. In my code the last process will get all the remaining job if N is not a factor of number of processes, the Matrix_Multiply function is:-

```
a.
b.  void Matrix_Multiply()
c.          work = m / comm_sz;
d.          if (my_rank == comm_sz-1 && (my_rank+1)*work < m)
e.                  for (i=my_rank*work; i<m; i++) {
f.                          Multiply i rows of matrices
g.                  }
h.          else
i.                  for (i=my_rank*work; i<(my_rank+1)*work; i++) {
j.                          Multiply i rows of matrices
```

3. Process with rank 0 will first get the value of N from the user by stdin then send it using P2P to each of the other processes, then it will do its part of multiplying matrix. Then rank 0 process will listen to each of other processes and get the updated matrix from each of them. After receiving it will modify the answer matrix.

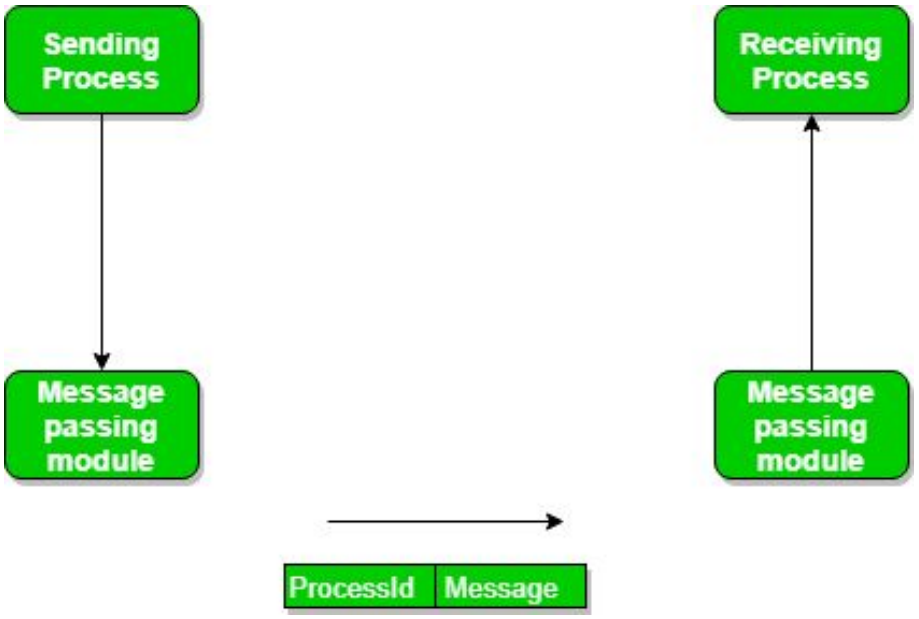
```
a.                  scanf("%d",&size);
b.                  for (int i=0; i<32*size; i++) {
c.                          Initialise A,B using random numbers between 0,1
d.                  }
e.                  for (int i=1;i<comm_sz;i++) {
f.                  MPI_Send(&size,1,MPI_INT,i,21,MPI_COMM_WORLD);
```

g. MPI_Send(A,32*size,MPI_DOUBLE,i,21,MPI_COMM_WORLD);
MPI_Send(B,32*size,MPI_DOUBLE,i,21,MPI_COMM_WORLD);
h. }
i.
j. Matrix_Multiply(A,B,answer,m,n,p,comm_sz,my_rank);
k. for (int i=1;i<comm_sz;i++) {
l. MPI_Recv(C,size*size,MPI_DOUBLE,i,21,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
m. }
n. Modify the answer matrix;

4. Process with rank != 0 will get size(N), A, B from rank 0, do their part of multiplication and send updated matrix to rank 0 process:-
    a. MPI_Recv(&size,1,MPI_INT,0,21,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    b. MPI_Recv(A,32*N,MPI_DOUBLE,0,21,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    c. MPI_Recv(B,32*N,MPI_DOUBLE,0,21,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    d. for (int i=1;i<comm_sz;i++) {
    Matrix_Multiply(A,B,C,m,n,p,comm_sz,i);
    e. MPI_Send(C,size*size,MPI_DOUBLE,0,21,MPI_COMM_WORLD);

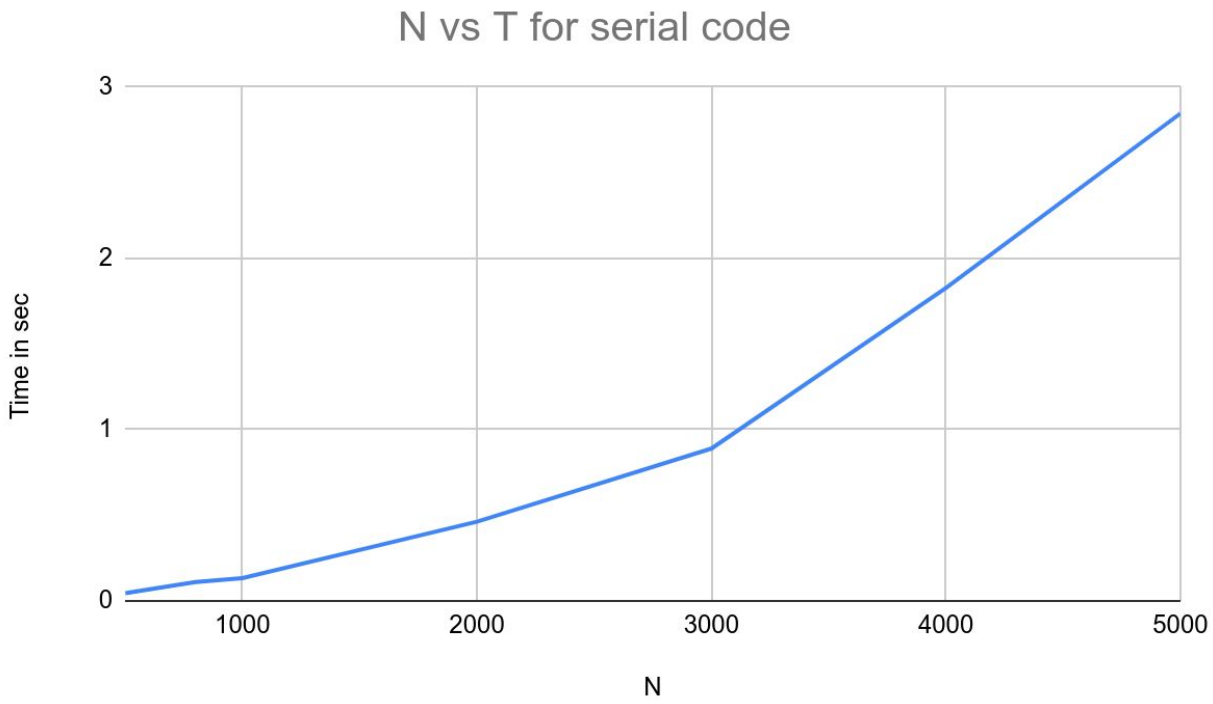
Experimental Results

The time calculated here for the parallel part is the total running time of process 0. If we don't consider any parallelisation we get serial multiplication data. The laptop used by one of us is a dual core hence maximum efficiency is expected at 2 MPI processes and other has octa core hence maximum efficiency at 8 processes. Hence only for the serial part we have shown both the graphs.



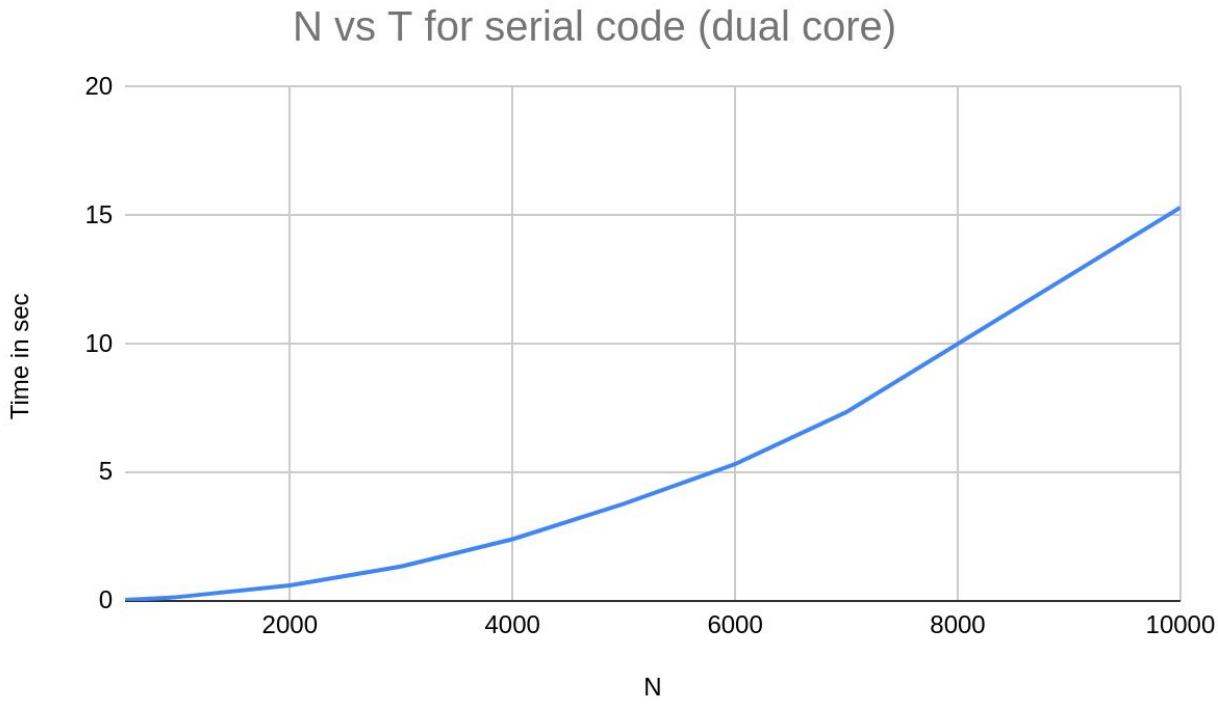
- Serial execution time without using multiple processes (octa core):-

N	T(sec)
500	0.044378
800	0.110217
1000	0.132972
2000	0.462293
3000	0.889091
4000	1.826444
5000	2.845939



- Serial execution time without using multiple processes (dual core) :-

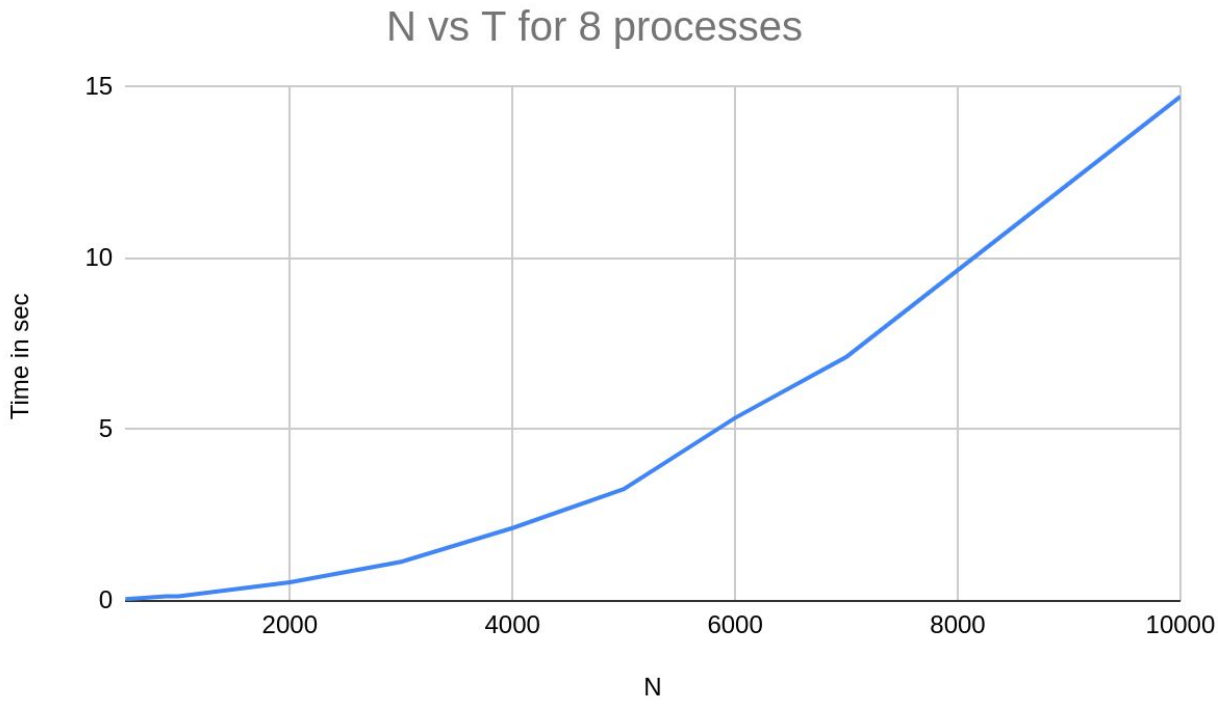
N	T (sec)
521	0.0398
890	0.1174
1000	0.15233
2000	0.5986
3000	1.34075
4000	2.4004
5000	3.7757
6000	5.323
7000	7.3495
10000	15.311



As we can clearly see, every time we double the size of the input, our time increases by 4 times, which shows that this matrix multiplication is of order O(N^2).

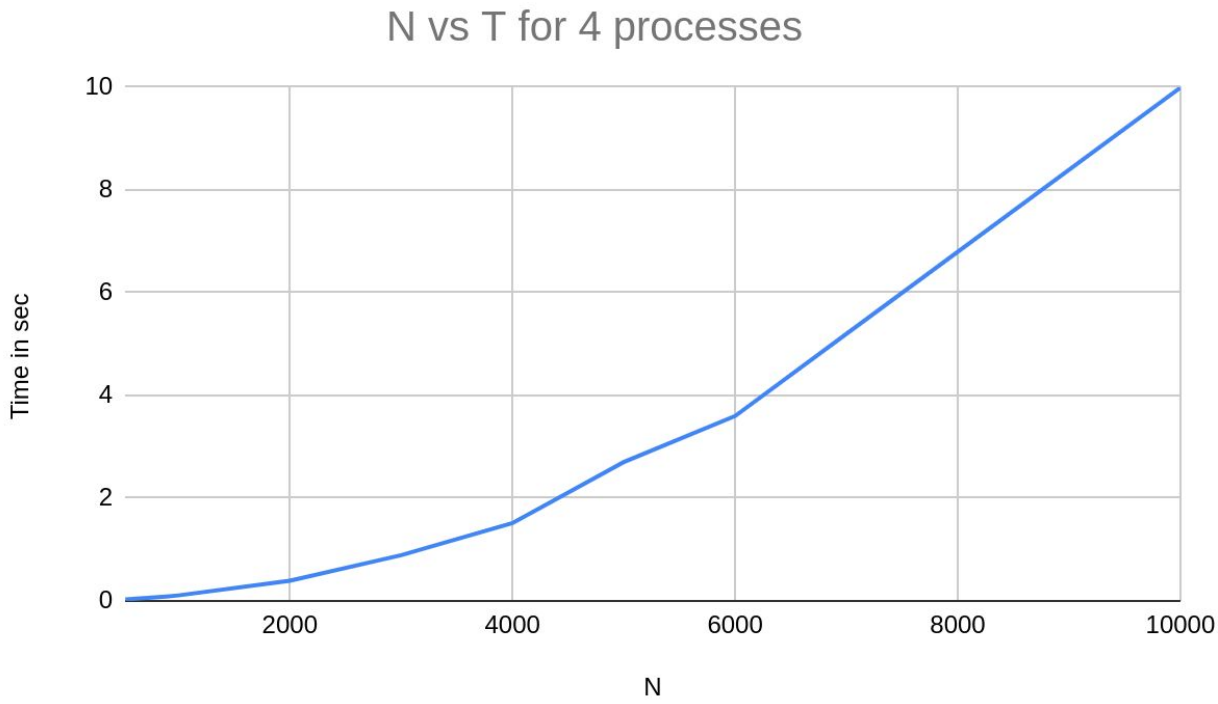
- Parallel execution time for 8 processes :-

N	T(sec)
521	0.05126
890	0.13055
1000	0.1359
2000	0.5454
3000	1.1416
4000	2.1261
5000	3.26255
6000	5.3415
7000	7.1237
10000	14.7253



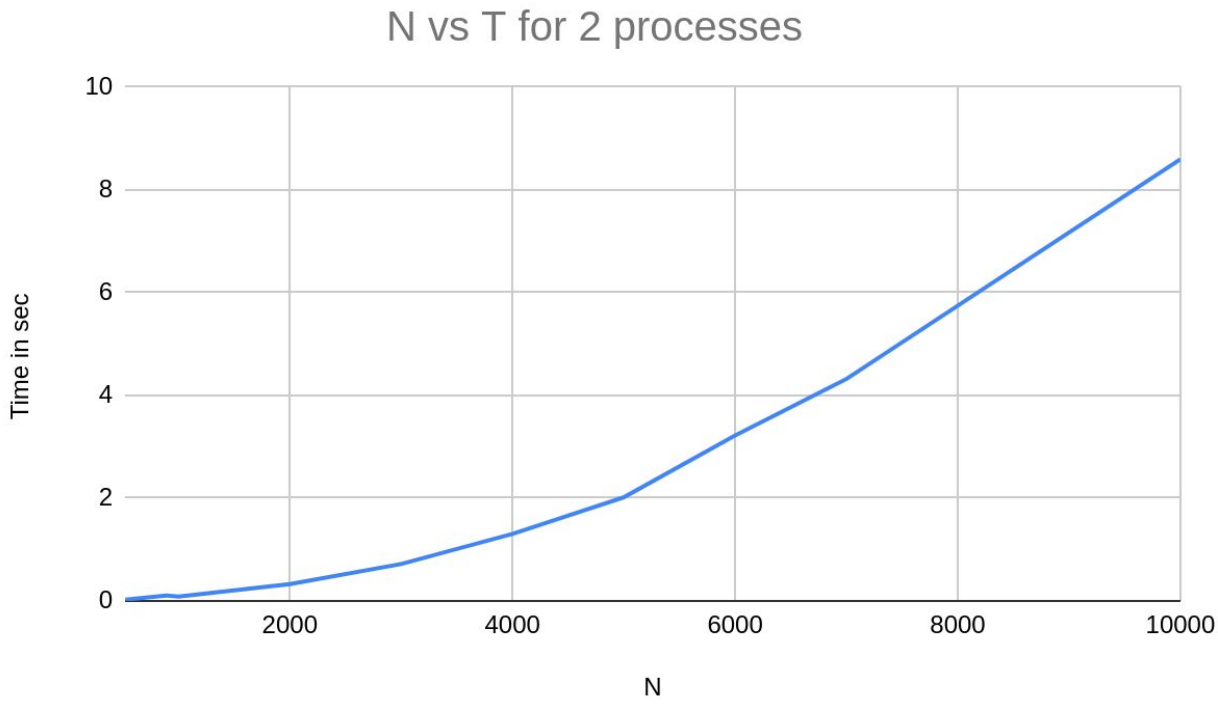
- Parallel execution time for 4 processes :-

N	T(sec)
521	0.0319
890	0.0847
1000	0.10648
2000	0.39123
3000	0.889091
4000	1.51554
5000	2.70283
6000	3.59691
7000	5.19646
10000	9.98713



- Parallel execution time for 2 processes:-

N	T(sec)
521	0.02346
890	0.10334
1000	0.0856
2000	0.32766
3000	0.71812
4000	1.30474
5000	2.0157
6000	3.21967
7000	4.31842
10000	8.59392



Analysis and Special Remarks

1. The time complexity can be interpreted very well from the graph as O(n2) which is the theoretical time complexity for matrix multiplication.
2. The sample code assumes N to be a multiple of number of processes. This is because the work should be divided almost equally between all processes.

Collective Communication

Collective Communication is a communication operation in which a group of processes works together to distribute or gather together a set of or one or more values.

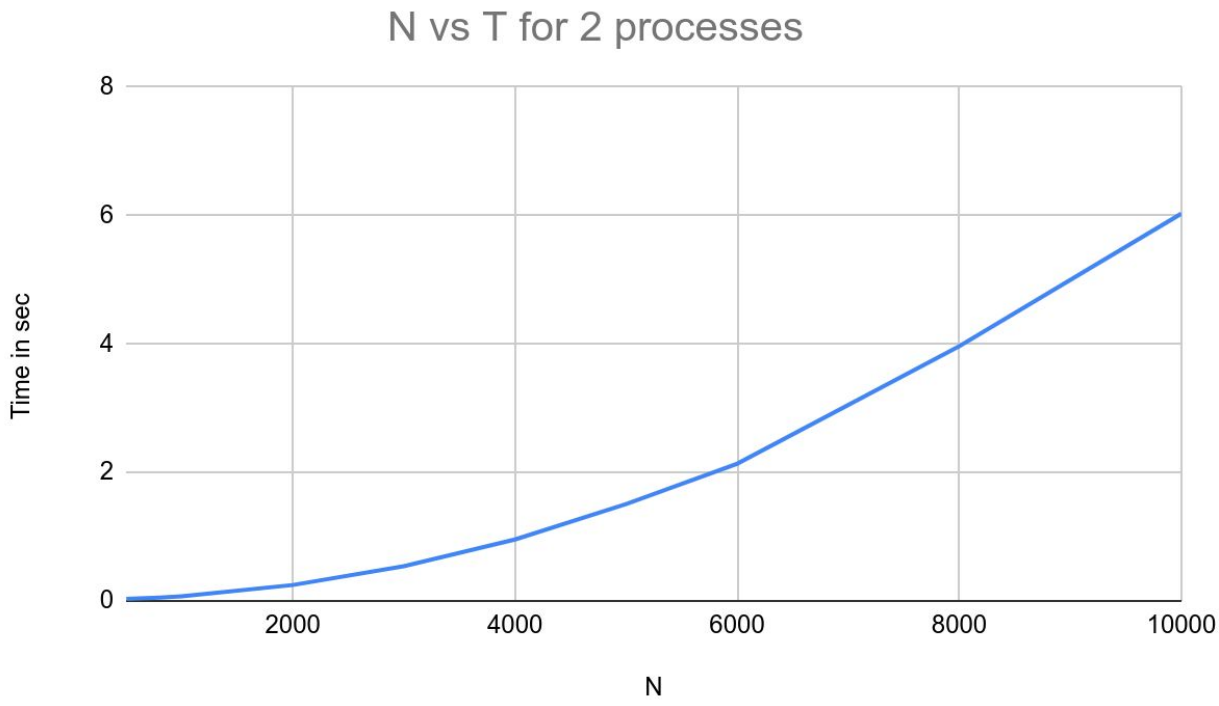
Operations Used: Scatter, Gather, Broadcast

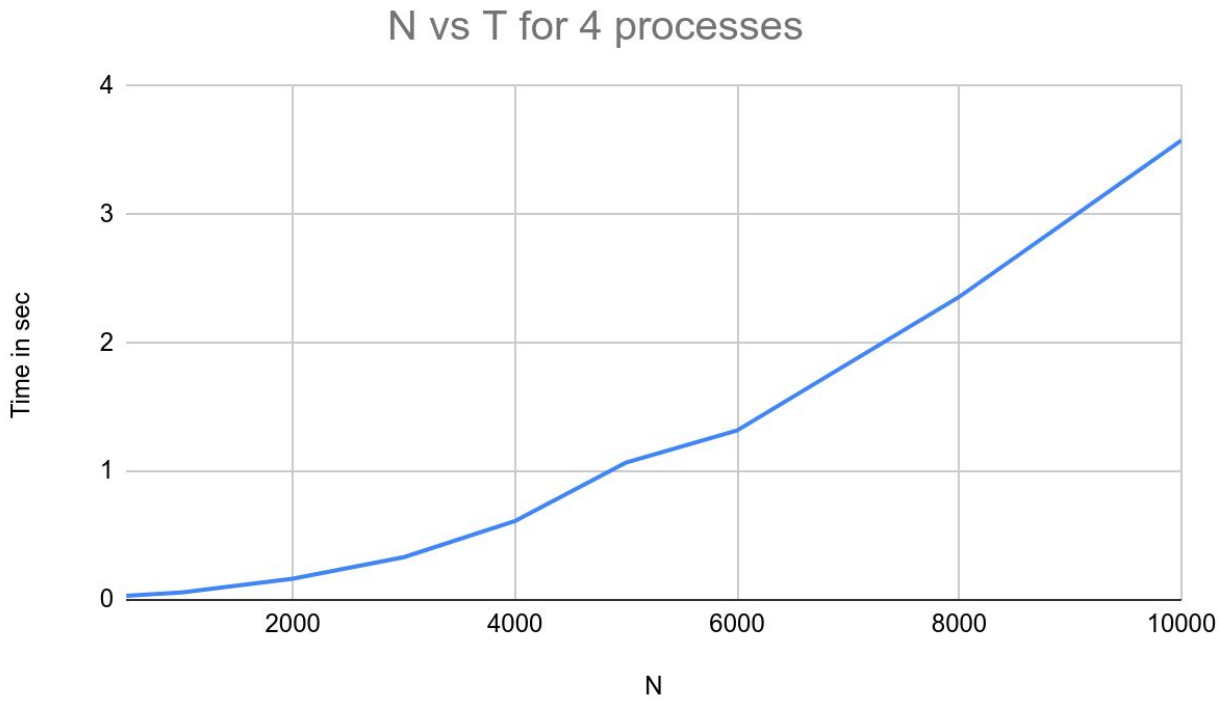
Main Code:

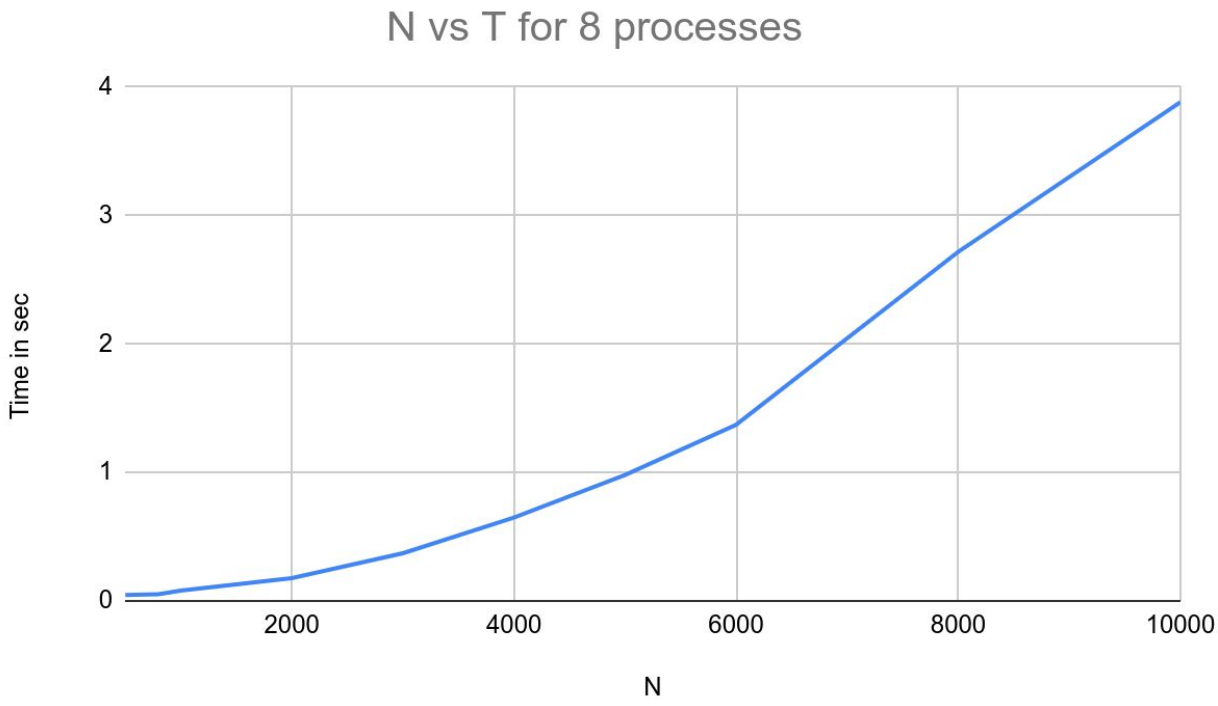
```
float *localCalcA = (float *)malloc((N*32/np) * sizeof(float *));
float *localCRow = (float *)malloc((N*N/np) * sizeof(float *));
MPI_Bcast(b, 32*N, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Scatter(a, N*32/np, MPI_FLOAT, localCalcA, N*32/np, MPI_FLOAT, 0, MPI_COMM_WORLD);
Matrix_Multiply(localCalcA, b, localCRow, N/np, 32, N);
MPI_Gather(localCRow, N*N/np, MPI_FLOAT, c, N*N/np, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

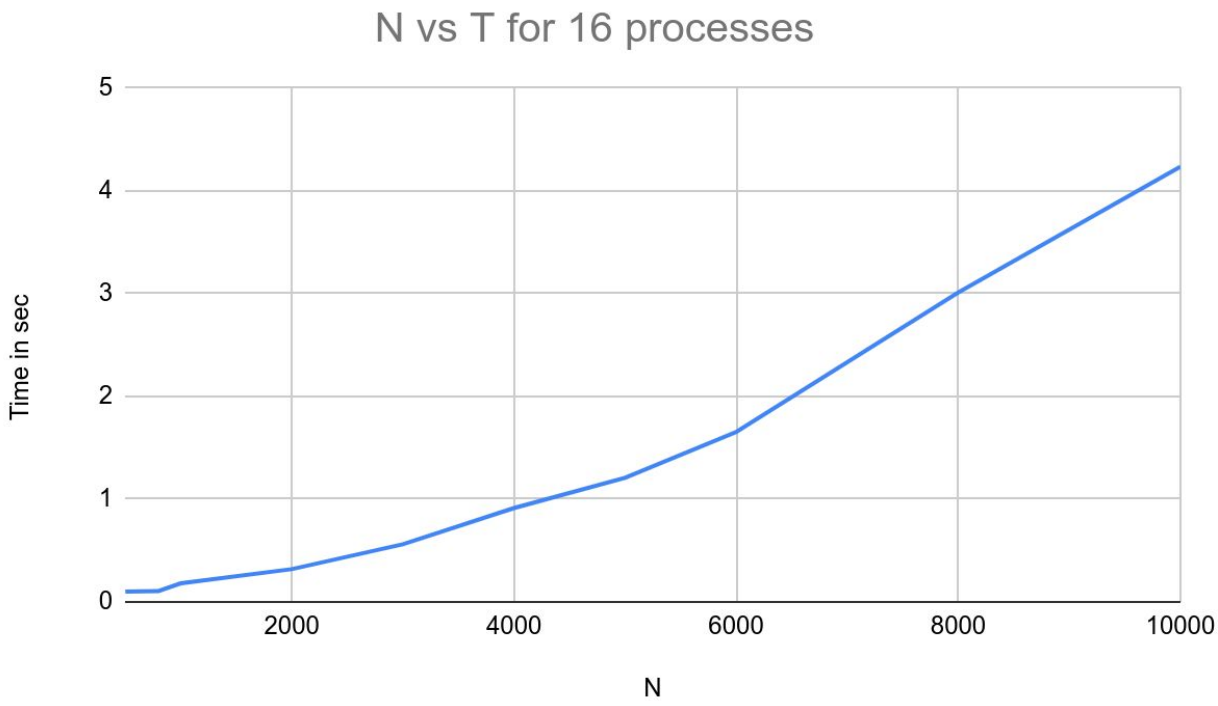
Algorithm: First we divide the N rows of A matrix in N/np blocks. We pass these blocks to all processes (scatter) and also pass the B matrix (broadcast). We then multiply the small blocks in different processes and obtain the parts of C. After all the parts of C are obtained, we gather all of them to complete it.

Experimental Analysis: (Without O3 optimization)

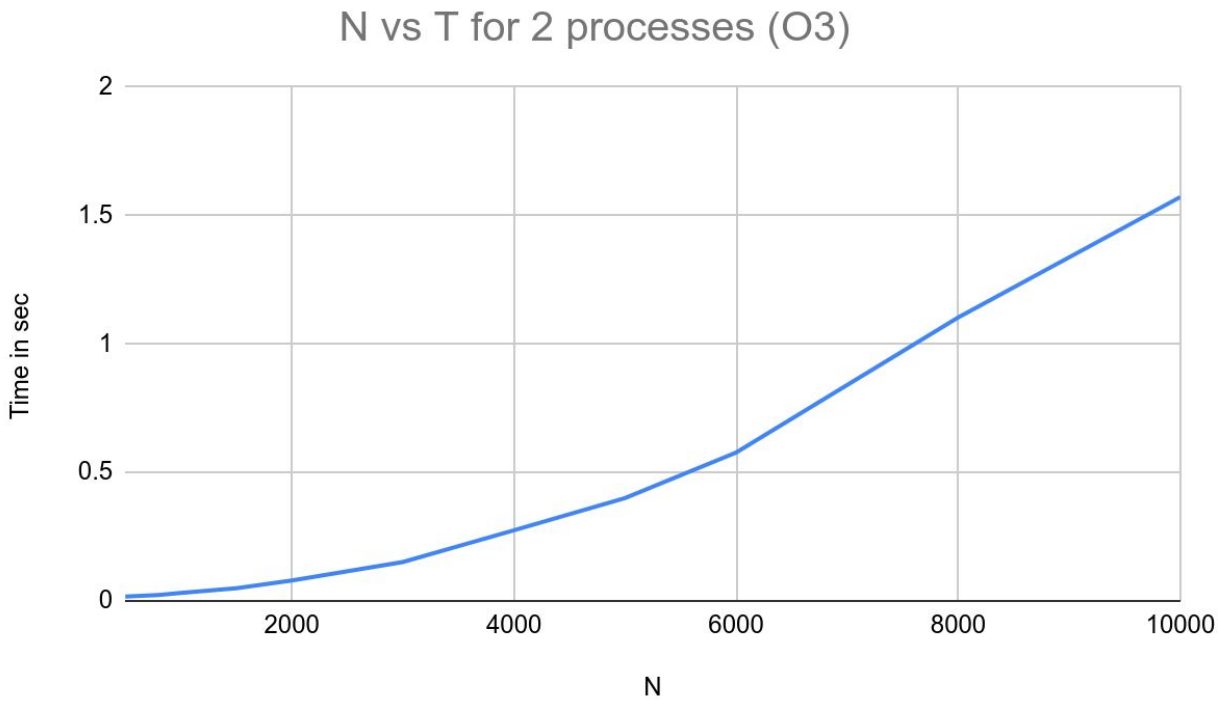


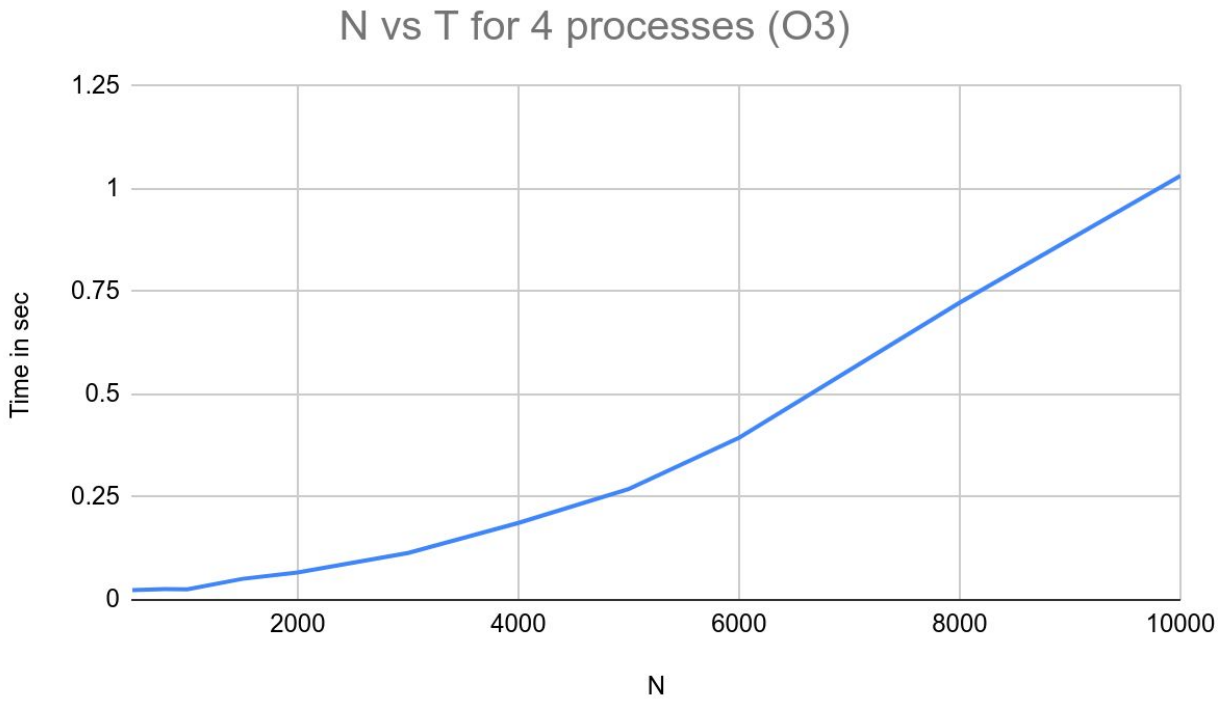


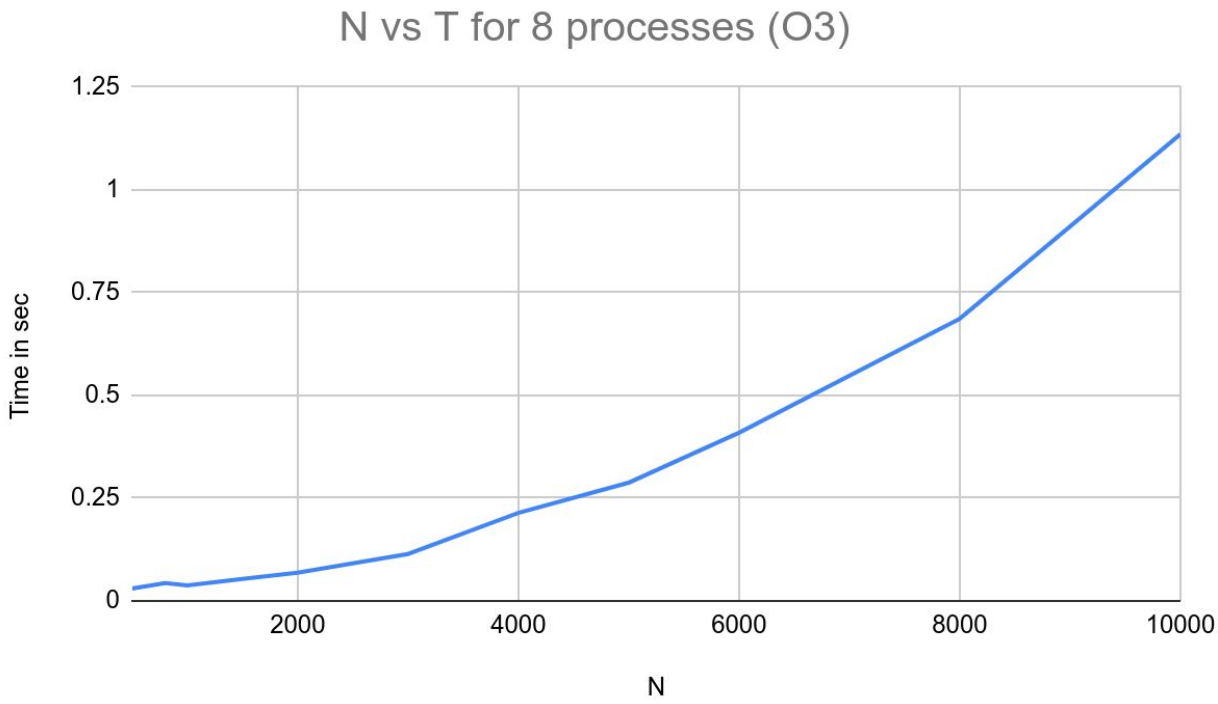

N vs T for 8 processes
Time in sec
0
1
2
3
4
2000
4000
6000
8000
10000
N


N vs T for 16 processes
Time in sec
0
1
2
3
4
5
2000
4000
6000
8000
10000
N

Experimental Analysis: (With O3 optimization)






N vs T for 8 processes (O3)
1.25
1
0.75
0.5
0.25
0
Time in sec
2000
4000
6000
8000
10000
N

Non Blocking Communication

A nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

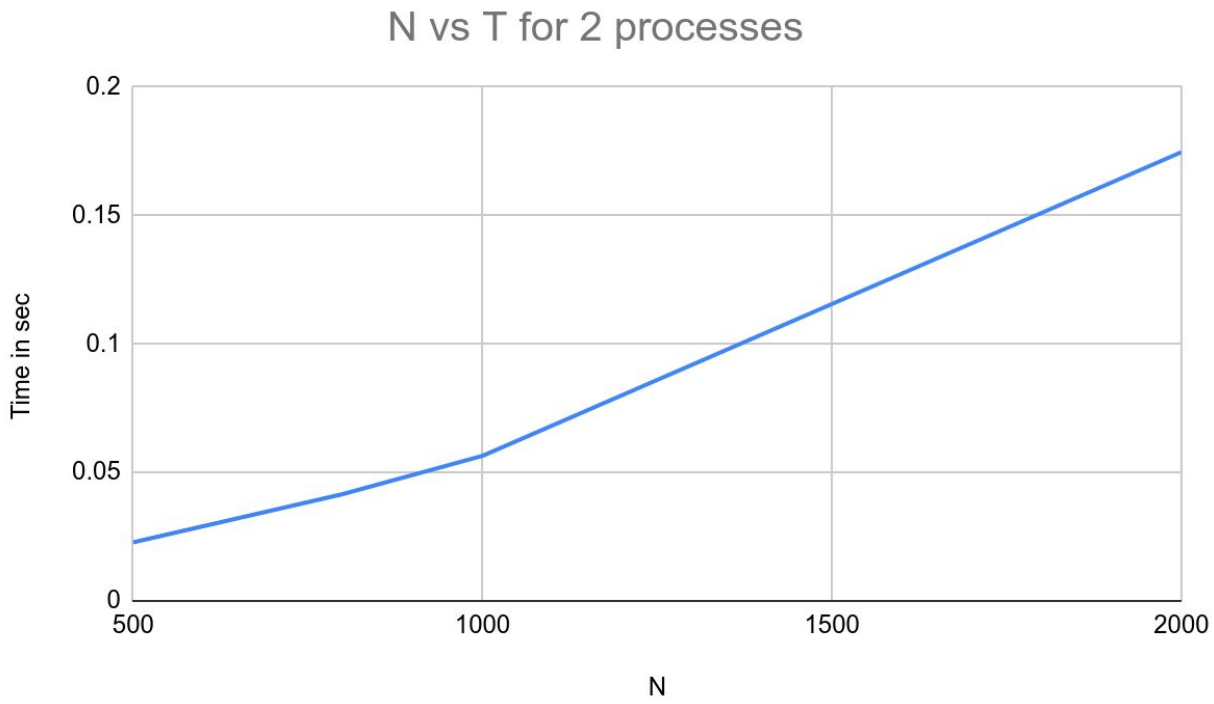
Operations Used: Scatter, Gather, Isend, IRecv, Wait

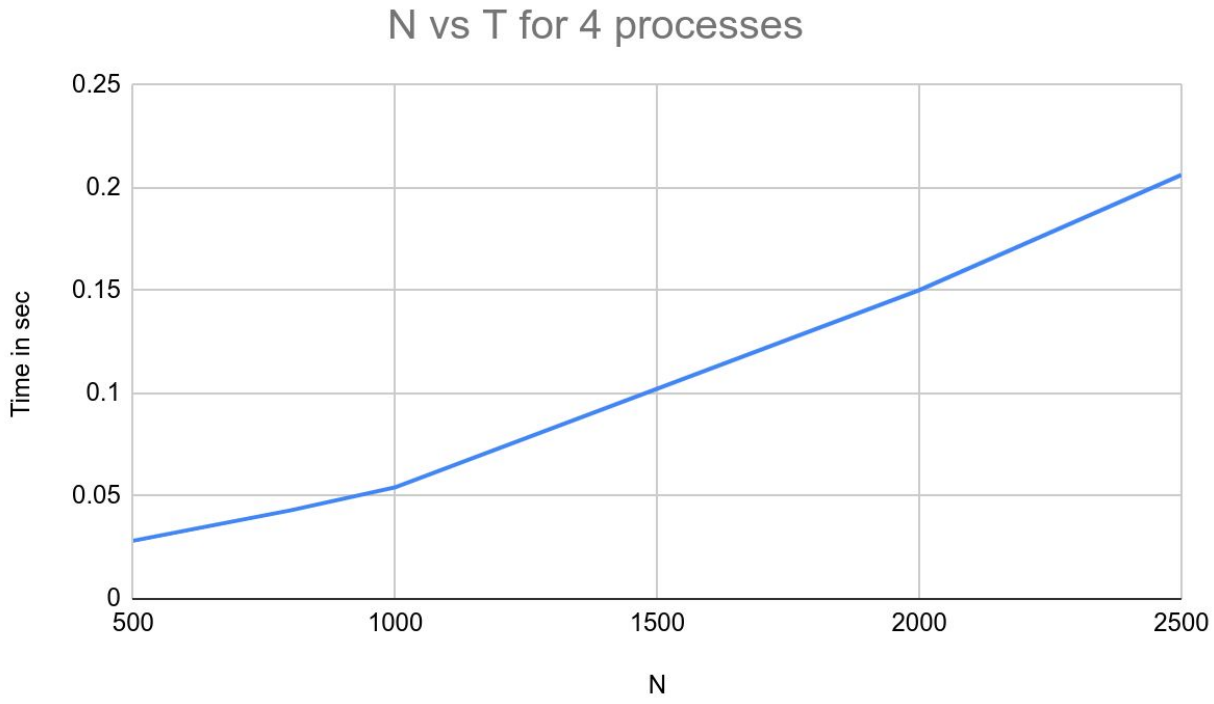
Algorithm: First we divide the N*32 values of A matrix in N*32/np blocks. We pass these blocks to all processes (scatter) and also pass the B matrix in a similar fashion. We then multiply the small blocks in different processes and obtain various parts of C. Then we slide the value of A by 1 and run this iteration np times.

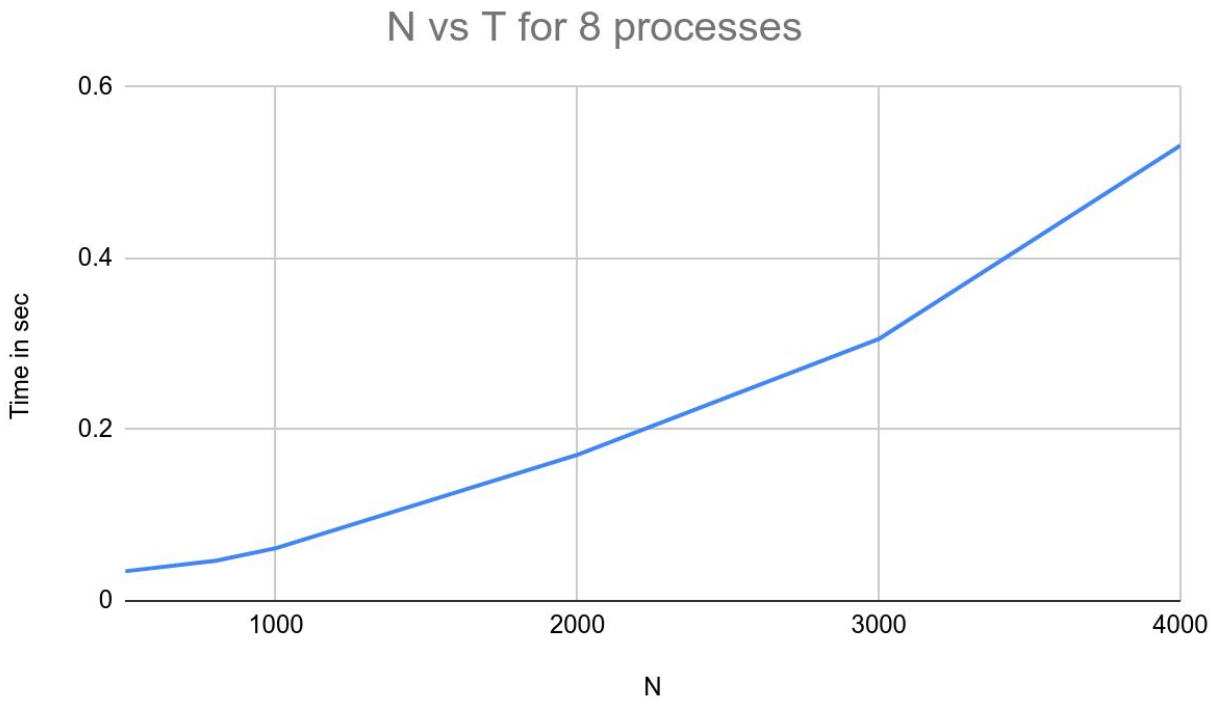
Main Code:

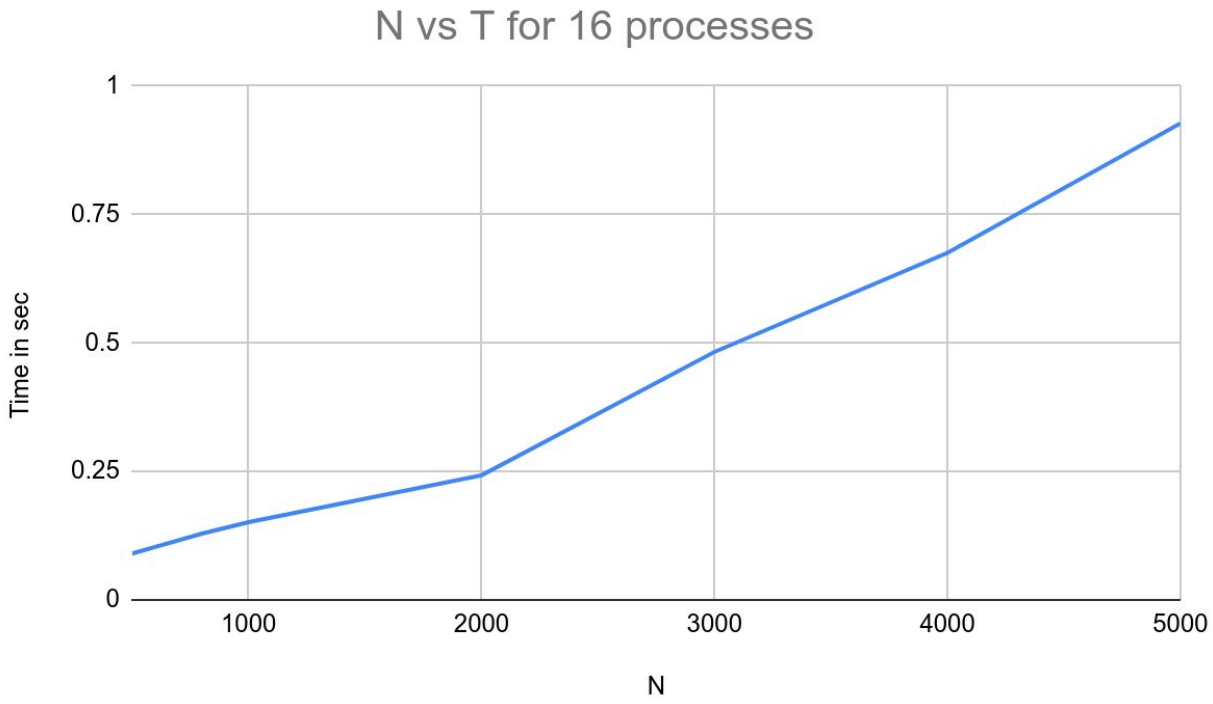
```
float localCalcA[N*32/np];
float localCalcB[32*N/np];
MPI_Scatter(a, N*32/np, MPI_FLOAT, localCalcA, N*32/np, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Scatter(b_column, 32*N/np, MPI_FLOAT, localCalcB, 32*N/np, MPI_FLOAT, 0, MPI_COMM_WORLD);
j = mype;
int k = 0;
float sum = 0;
float localCalcC[N*N/np];
for(int iter = 0; iter<np; iter++){
        for(i=0; i<N/np;i++){
                for(k=0;k<N/np;k++){
                        sum = 0;
                        for(int f=0;f<32;f++){
                                sum += localCalcA[f + i*32]*localCalcB[f+k*32];
                                }
                                localCalcC[k*N+i+j*N/np] = sum;
                        }
                }
                if(j == np-1){
                        j = 0;
                }
                else{
                        j++;
                }
                MPI_Isend(localCalcA, N*32/np, MPI_FLOAT, (mype==0 ? np-1:mype-1), 0, MPI_COMM_WORLD,
&request);
                MPI_Irecv(localCalcA, N*32/np, MPI_FLOAT, (mype==np-1 ? 0:mype+1), 0, MPI_COMM_WORLD,
&request);
                MPI_Wait(&request, &status);
        }


        MPI_Gather(localCalcC, N*N/np, MPI_FLOAT, c, N*N/np, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Experimental Analysis: (Without O3 optimization)

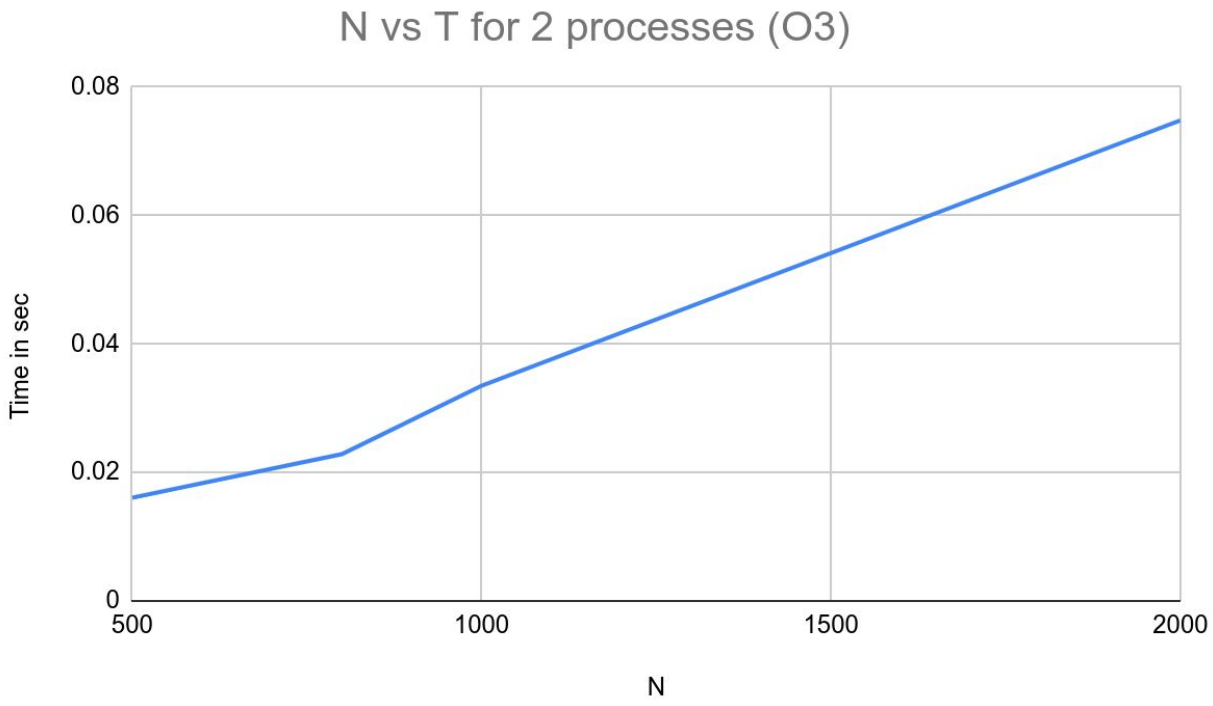


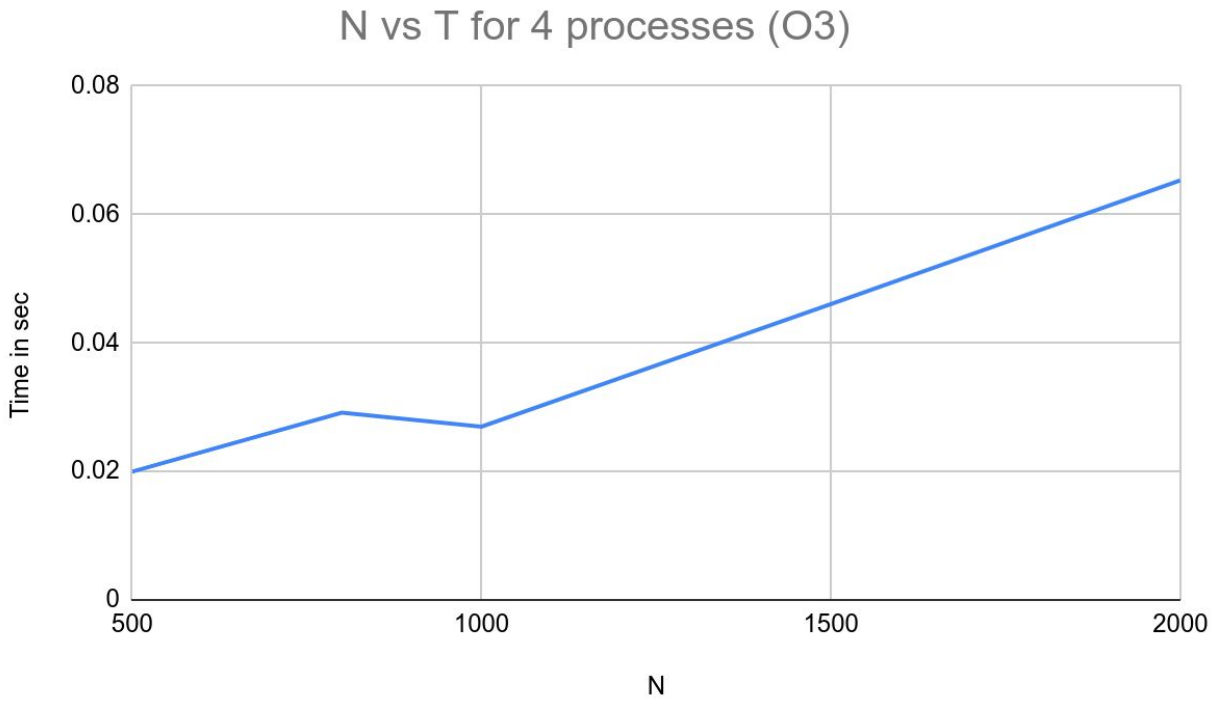


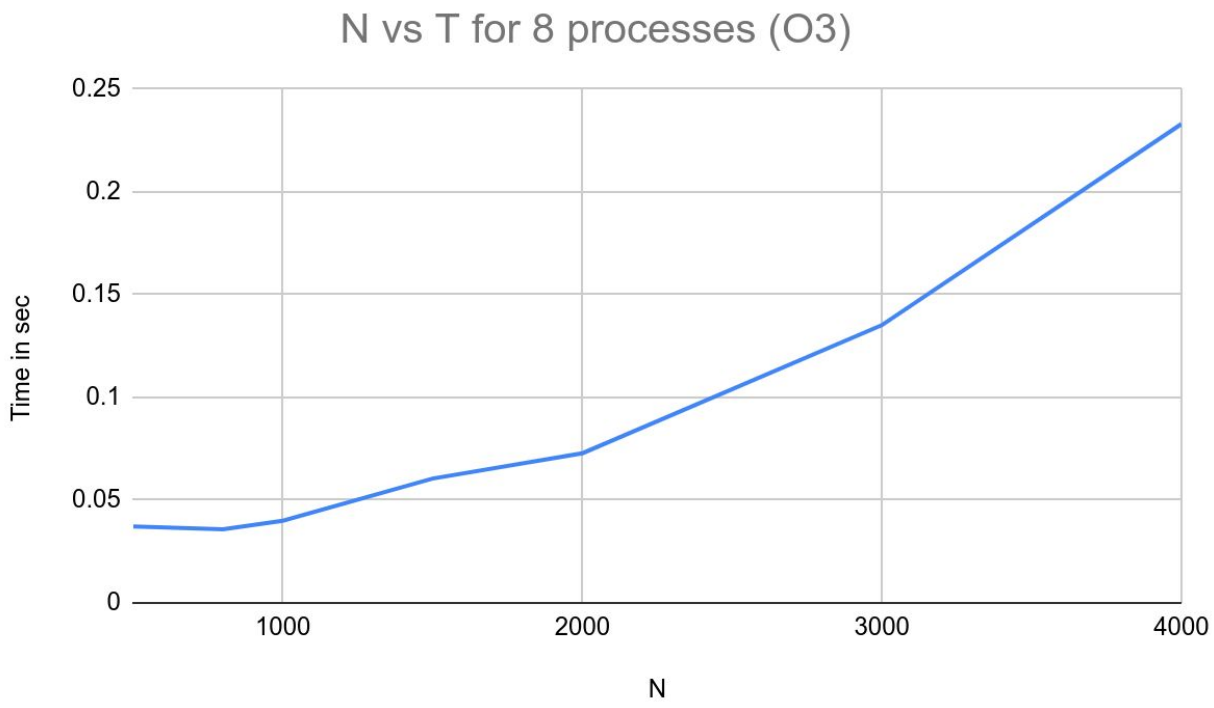

N vs T for 8 processes
0.6
0.4
0.2
0
Time in sec
1000
2000
3000
4000
N

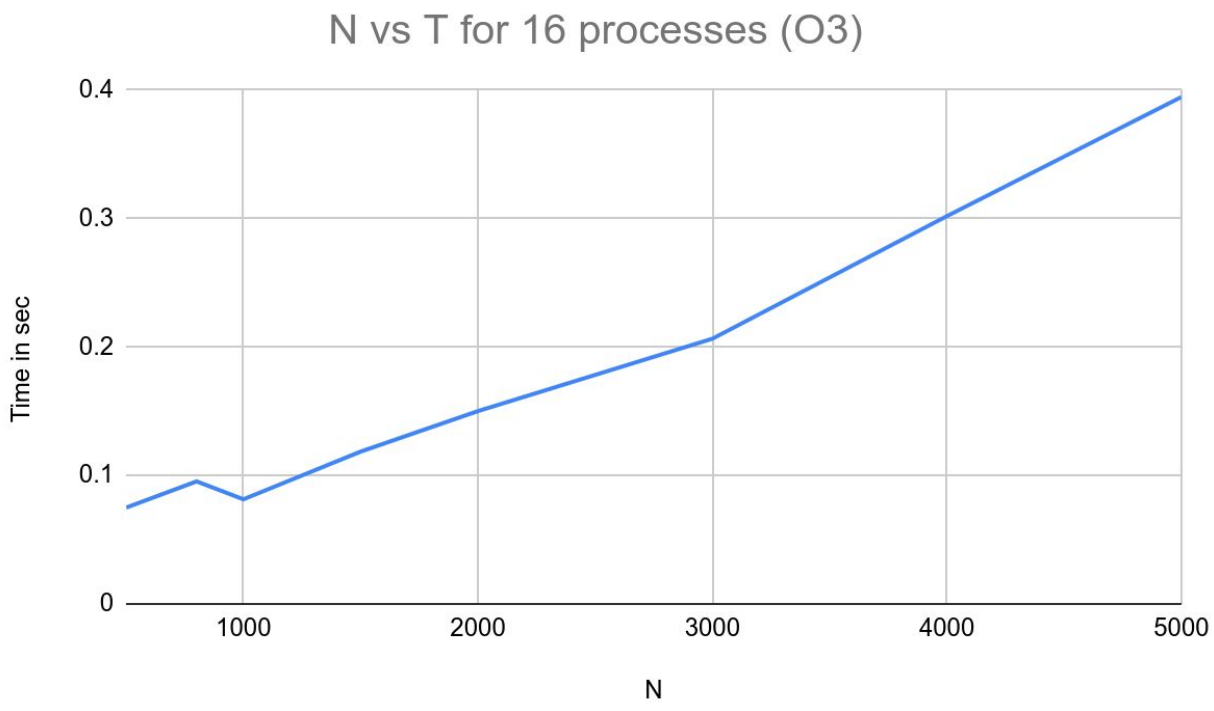

N vs T for 16 processes
1
0.75
0.5
0.25
0
Time in sec
1000
2000
3000
4000
5000
N

Without O3 optimization:





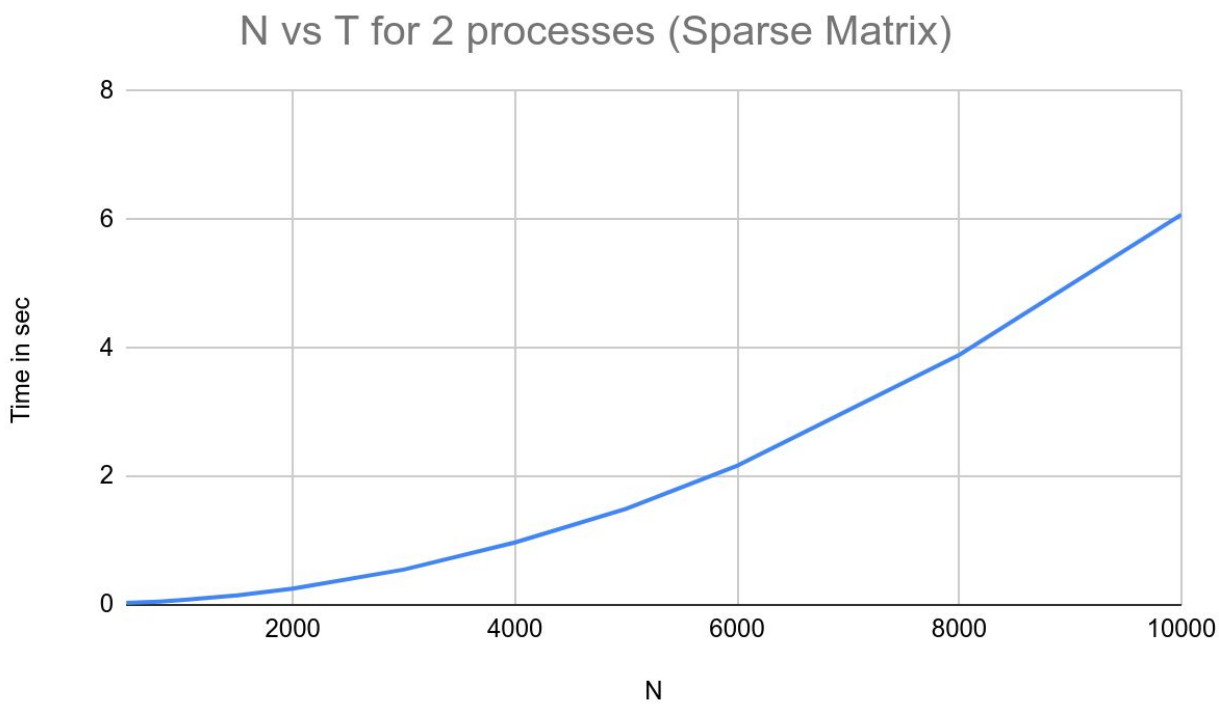

N vs T for 8 processes (O3)
Time in sec
0.25
0.2
0.15
0.1
0.05
0
1000
2000
3000
4000
N


N vs T for 16 processes (O3)
Time in sec
0.4
0.3
0.2
0.1
0
1000
2000
3000
4000
5000
N

Machine Specifications:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 142
Model name: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Stepping: 10
CPU MHz: 1764.604
CPU max MHz: 4000.0000
CPU min MHz: 400.0000
BogoMIPS: 3984.00
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 8192K
NUMA node0 CPU(s): 0-7

Sparse vs Dense Matrix Comparison:



This is for Collective Communication Programme. This is quite similar to the one we observed for dense matrix. This is because, 0*0 takes approximately the same computation time as any other small float computation.

Conclusion:

- In all the cases, the graph we obtain, is quadratic in N, if we keep the number of cores constant.
- We observe that the best performance is observed when we keep the number of processes to be equal to 4 which is synchronous to the finding in LU Decomposition problem. This is totally a system property.

Restrictions on N:

We have put a restriction on N, that it should be a multiple of number of processes in Collective Communication and Non Blocking Communication Codes. This is because we are using the scatter and gather calls which distribute the data equally among different processes.

There is also one more restriction on N. In the non-blocking code, after N = 2500, the code gives segmentation fault. This is an MPI error and is caused due to insufficient RAM in the system. The problem is solved by increasing the number of cores. But even for 8 cores, N=6000 is the max N till which the code works. This is totally a system error and can be debugged using either collective communication approach or the blocking call approach.

REFERENCES

1. https://www.geeksforgeeks.org/inter-process-communication-ipc/
2. Parallel Programming with C, MPI and OpenMP by Michael J. Quinn