**Assignment 2: Part 1**

Medha Kant, Namrata Priyadarshini

In this assignment we were supposed to calculate the product of two matrices A (of size N\*32) and B (of size 32\*N), the result of which should be an N\*N matrix.

The serial code for matrix multiplication of matrices A and B, and storing the storing the result in another matrix C would look like this:

for (i = 0; i < m; i++){

        for (j = 0; j < p; j++){

            C[i\*p + j] = 0.0;

            for (k = 0; k < n; k++)

                C[i\*p + j] += A[i\*n + k] \* B[k\*p + j];

        }

    }

As the value of p is 32 and m=n=N in our case, therefore the theoretical time complexity for this should be O(N^2).

**Implementation of the Parallel mechanism**

1. **Blocking P2P Communication**
2. MPI\_Init is invoked to initiate MPI and synchronize all participating processes.
3. After initiating, process 0 works as a master process, that sends data to other processes and then all processes execute the identical program for multiplying two matrices.
4. The number of processes is determined by the user runtime command:  
   $ mpirun -np proc object\_file  
   where proc is the number of processes.
5. Blocking mpi calls, MPI\_Send and MPI\_Recv is called to send to receive messages from one process to another.

*Strategies used*

* 1. Master process divides rows of matrix A for (proc) processes.
  2. Data sent by Master process to all other processes using MPI\_Send:
* Number of rows of matrix A
* Offset of the 1st row that is being given to that process
* Rows of matrix A that the process will work on all rows of matrix B
  1. Data is received by other processes using MPI\_Recv.
  2. Master process as well as all other processes work on the matrix multiplication on a given number of rows.
  3. Slave processes sends calculated matrix and offsets and number of rows to the Master process.
  4. Master process receives data from all other process in new Matrix C.

1. **Non-Blocking P2P Communication**
2. MPI\_Init is invoked to initiate MPI and synchronize all participating processes.
3. After initiating, process 0 works as a master process, that sends data to other processes and then all processes execute the identical program for multiplying two matrices.
4. The number of processes is determined by the user runtime command:  
   $ mpirun -np proc object\_file  
   where proc is the number of processes.
5. Non-Blocking mpi calls, MPI\_Isend and MPI\_Irecv is called to send to receive messages from one process to another.

*Strategies used*

* 1. Master process divides rows of matrix A for (proc-1) processes.
  2. Data sent by Master process to all other processes using MPI\_Isend:
* Number of rows of matrix A
* Offset of the 1st row that is being given to that process
* Rows of matrix A that the process will work on all rows of matrix B
  1. Data is received by other processes using MPI\_Irecv.
  2. Slave processes work on the matrix multiplication on a given number of rows and then send back the calculated matrix to the master process.
  3. Master process receives data from all other process in new Matrix C.

*Non-Blocking send and receive calls after which MPI\_Wait() is invoked:*

1. After master sends number of rows and offset to other processes MPI\_Wait is invoked because the processes then allocate the space to the matrices A,B and C using malloc functions and similarly slave processes calls MPI\_Wait() after receiving number of rows and offset so that they know how much memory to allocate to each of the matrices.
2. After master process has sent data to all the slave processes and after slave processes have received the data.
3. After slave processes have sent back number of rows and offset to the process and similarly after master process has received them.
4. After slave processes have sent the calculated matrix C and master process has received the data C.
5. **Collective Communication**
6. MPI\_Init is invoked to initiate MPI and synchronize all participating processes.
7. After initiating, process 0 works as a master process, that sends data to other processes and then all processes execute the identical program for multiplying two matrices.
8. The number of processes is determined by the user runtime command:  
   $ mpirun -np proc object\_file  
   where proc is the number of processes.
9. Blocking mpi calls, MPI\_Send and MPI\_Recv is called to send to receive messages from one process to another. Also, MPI\_Bcast is used to send data at once to all other processes.

*Strategies used*

* 1. Master Process divides rows of matrix A for (proc) processes.
  2. Data sent by Master process to all other processes using MPI\_Send:
* Rows of matrix A that the process will work on all rows of matrix B
  1. Data broadcasted by Master process to all other processes using MPI\_Bcast:
* The entire matrix B
* The value of N, total number of rows in matrix A.
* The number of rows of A that each process has to multiply.
  1. Data is received by other processes using MPI\_Recv.
  2. Master process as well as all other processes work on the matrix multiplication on a given number of rows. If the number of rows is not exactly divisible by proc, then the extra left rows are calculated by the master process.
  3. Slave processes sends calculated matrix and offsets and number of rows to the Master process.
  4. Master process receives data from all other process in Matrix C.

**Compiling and running the program**

To compile the program, simply run the makefile. The makefile would produce an executable **\*\*matmul\*\***. To compile the program with O3 optimisation, uncomment the O3 in **\*\*CFLAGS\*\*** in the makefile.

To run the program use:

```bash

mpirun -np [Nubmer of processor to use] matmul [matrix size] [parallel mode]

```

The *\*Number of processor to use\** should be less than or equal to the number of processors your system has. Also, there are no restriction on matrix size. It doesn't necessarily have to be a perfectly divisible by the number of processor specified. The *\*parallel mode\** can be 0,1,2.

- 0 > Blocking P2P Communication

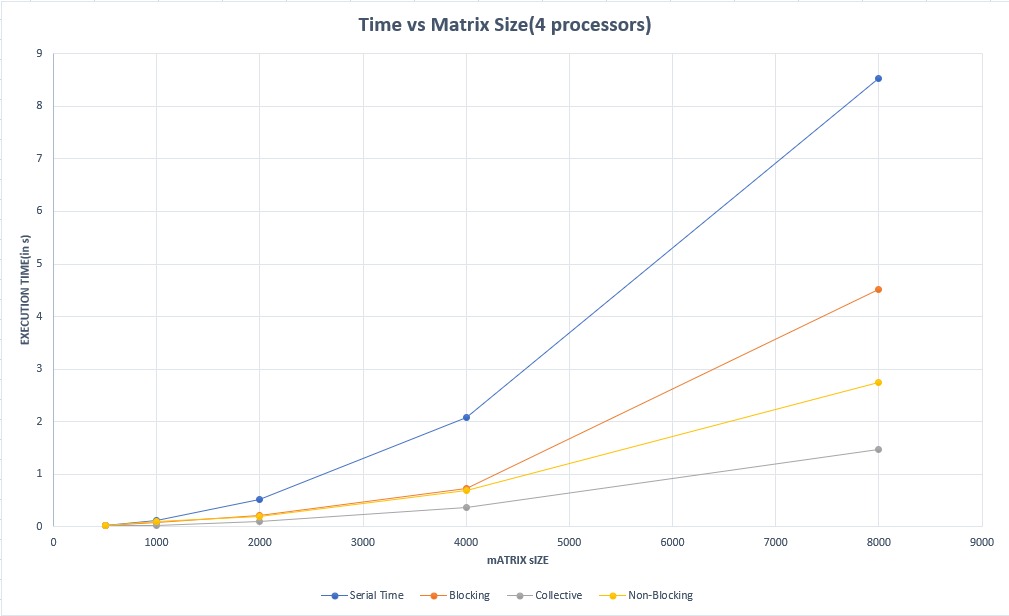
- 1 > Non-Blocking P2P Communication

- 2 > Collective Communication

**Plots and Results**

The execution time for serial and parallel matrix multiplication on 4 processors -

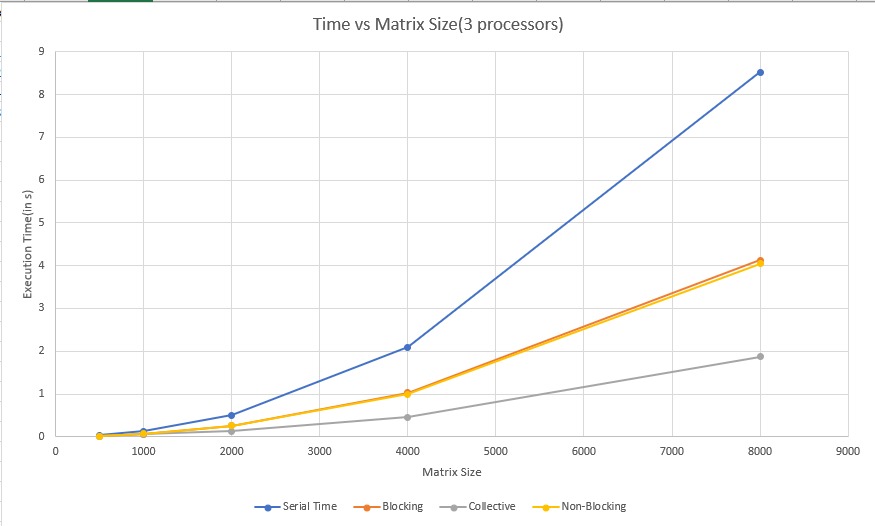
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **Serial Time** | **Blocking P2P** | **Non-Blocking P2P** | **Collective** |
| **500** | 0.035 | 0.027 | 0.018 | 0.021 |
| **1000** | 0.126 | 0.077 | 0.096 | 0.033 |
| **2000** | 0.513 | 0.218 | 0.192 | 0.112 |
| **4000** | 2.09 | 0.73 | 0.7 | 0.37 |
| **8000** | 8.52 | 2.87 | 2.74 | 1.47 |



**As is clear from the table and from the graph, all the programs for matrix multiplication, parallel and serial, show an N2 dependence. Also, the best performer is the collective communication.**

The execution time for serial and parallel matrix multiplication on 3 processors -

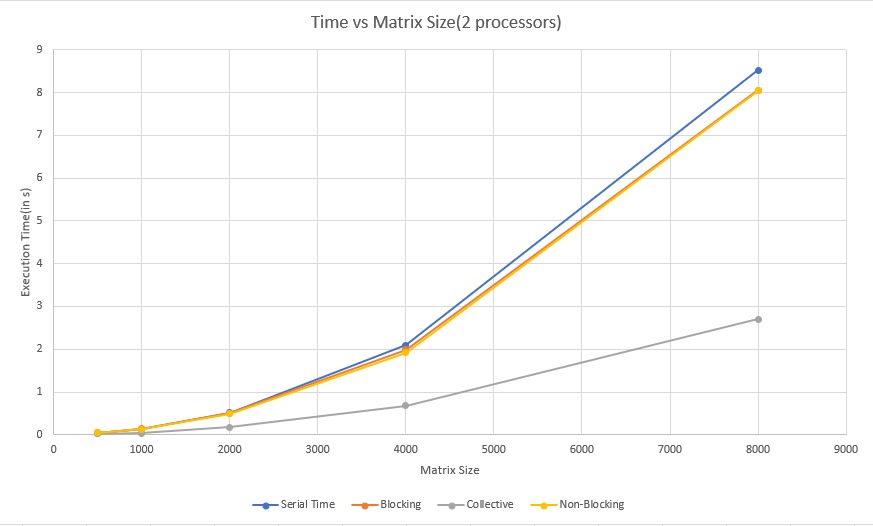
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **Serial Time** | **Blocking P2P** | **Non-Blocking P2P** | **Collective** |
| **500** | 0.033 | 0.016 | 0.018 | 0.014 |
| **1000** | 0.126 | 0.065 | 0.076 | 0.06 |
| **2000** | 0.504 | 0.257 | 0.26 | 0.124 |
| **4000** | 2.09 | 1.03 | 0.99 | 0.46 |
| **8000** | 8.52 | 4.13 | 4.04 | 1.87 |

****

**Again, we see clearly from the table and from the graph, all the programs for matrix multiplication, parallel and serial, show an N2 dependence. Again, the best performer is the collective communication. The non-blocking and blocking code performance are almost equal.**

The execution time for serial and parallel matrix multiplication on 2 processors –

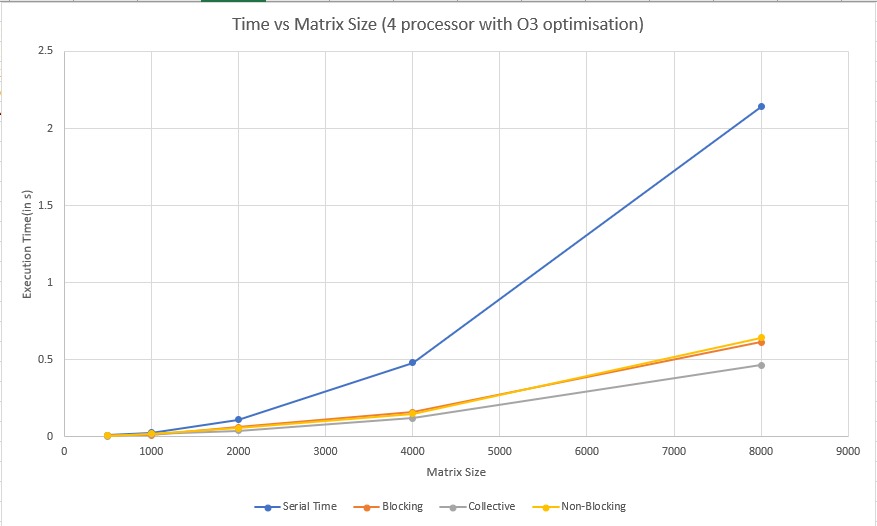
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **Serial Time** | **Blocking P2P** | **Non-Blocking P2P** | **Collective** |
| **500** | 0.035 | 0.051 | 0.048 | 0.025 |
| **1000** | 0.126 | 0.14 | 0.13 | 0.04 |
| **2000** | 0.513 | 0.5 | 0.49 | 0.18 |
| **4000** | 2.09 | 1.99 | 1.919 | 0.68 |
| **8000** | 8.52 | 8.05 | 8.04 | 2.7 |

****

**Again, we see clearly from the table and from the graph, all the programs for matrix multiplication, parallel and serial, show an N2 dependence. Again, the best performer is the collective communication. The non-blocking and blocking code performance are almost equal and equal to the serial code. This is because, no computation is being done on rank 0, therefore the effective number of cores doing the multiplication is 1.**

The execution time for serial and parallel matrix multiplication on 4 processors compiled with O3 optimisation flag –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **Serial Time** | **Blocking** | **Collective** | **Non-Blocking** |
| **500** | 0.011 | 0.01 | 0.002 | 0.007 |
| **1000** | 0.026 | 0.011 | 0.015 | 0.021 |
| **2000** | 0.11 | 0.062 | 0.038 | 0.057 |
| **4000** | 0.48 | 0.16 | 0.12 | 0.15 |
| **8000** | 2.14 | 0.614 | 0.465 | 0.643 |

****

**This time we see a massive boost in performance. The serial code is performing as good as the best-case parallel code without O3 optimisation. This is because the O3 optimisation flag does parallelise the internal loops in the programs as and where possible. Also, for this case, the performance of all three parallel codes is almost similar with collective code still being better than the others.**

**Conclusion**

1. The N2 theoretical time complexity was also seen practically in all the cases.
2. Collective is better than blocking and non-blocking code, this maybe because the other collective program sends all the data at once to all the processes, this saves some time and also in the collective case the master is free to do computation as well.
3. The O3 optimisation code does optimise the serial code, but it also optimises the parallel code, thereby increasing the performance of an already good performing code.

**Future Work**

1. We are storing both the matrix in the rows major form as specified in the assignment statement, but the performance can be increased if we store the second matrix in column major format thus fast access of the columns.
2. In blocking and non-blocking code, we can try to perform computation in master processor as well.