NC State University

Department of Electrical and Computer Engineering

ECE/CSC 506/406: Architecture of Parallel Computers

Fall 2022

Project #1: OpenMP, MPI and Hybrid Parallel Programming

by

RAGHUL SRINIVASAN

(200483357)

1. Implementation of OpenMP, MPI & Hybrid Parallel Programming

OpenMP:

Different scheduling methods along with different Chunk sizes were tried. Among all the combinations, guided scheduling gave out the best result.

Vector Addition:

In Vector Addition, there is only one for loop in the Vector Addition Implementation which can be parallelized. OpenMP was implemented by using **#pragma omp parallel for schedule(guided)** to partition the work among the threads (where n = 1, 2, 4, 8, 16)

Matrix Addition:

In Matrix Addition, there are 2 nested loops in Matrix Addition Implementation which can be parallelized. OpenMP was implemented by using **#pragma omp parallel for schedule(guided) collapse(2)** to partition the work among the threads (where n = 1, 2, 4, 8, 16).

The keyword **Collapse** is used to partition the work of the nested loops equally among the given number of processors.

Matrix Multiplication:

In Matrix Multiplication, there are 3 nested loops in Matrix Multiplication Implementation which can be parallelized. OpenMP was implemented by using **#pragma omp parallel for schedule(guided) collapse(3)** to partition the work among the threads (where n = 1, 2, 4, 8, 16).

In addition, usage of **#pragma omp atomic** is imperative as there is a shared variable that needs to be updated with the latest value before being used by other threads.

MPI:

MPI needs explicit message sharing among the different processors to implement the desired calculation. This is implemented by using MPI_Send() and MPI_Recv(). A processor with rank 0 is considered a master processor which partitions the input data and sends them to different (slave) processors available via **MPI_Send().** Once the slave processors are done with the calculations, the results are sent back to the master processor via **MPI_Recv()**.

Vector Addition:

In Vector addition, the size of input vectors will be N * 1. So, there will be only one Column per input upon which we have to do addition. The datum in this one column is partitioned in such a way that N-1 available processors (including the processor with rank 0) will receive the same amount of data.

Whereas the final processor will receive the same amount of data that the other processors received (if the row count of the input matrix is equal to the power of 2) or less than that of other processors (if the row count of the input matrix is not in terms of the power of 2).

From Processor with rank 0, before sending the partitioned input vectors, the amount of data that is to be received by each of the slave processors is sent so that the slave processors can readily allocate memory and wait for the input vectors to perform addition.

After computation, the result is shared from slave processors to the master processor, which orders the result from different processors (including its own) in the order of processor Id.

Matrix Addition:
In Matrix addition, the size of both the input matrixes will be the same (N * M). So, to implement MPI for Matrix addition, Input matrixes are partitioned in such a way that the number of columns received by each of the processors will be equal to that of the input matrix column size. But the rows among these columns are partitioned in such a way that N-1 available processors (including the processor with rank 0) will receive the same number of rows.

Whereas the final processor will receive the same number of rows that the other processors received (if the total row count of the input matrix is equal to the power of 2) or less than that of other processors (if the total row count of the input matrix is not in terms of the power of 2).

From Processor with rank 0, before sending the desired partitioned input matrixes, the amount of data that is to be received by each of the slave processors is sent so that the slave processors can readily allocate memory and wait for the input vectors to perform addition.

After computation, the result is shared from slave processors to the master processor, which orders the result from different processors (including its own) in the order of processor Id.

Matrix Multiplication:
In Matrix multiplication, the column count of Input Matrix 1 will be equal to the row count of Input Matrix 2. But the row count of Input Matrix 1 and the Column count of Input Matrix 2 can vary. So, to implement MPI for Matrix multiplication, both of the Input matrixes cannot be partitioned in the same way.

Input Matrix 1 is partitioned in such a way that the number of columns received by each of the processors will be equal to that of the input matrix 1 column size. But the rows among these columns are partitioned in such a way that N-1 available processors (including the processor with rank 0) will receive the same number of rows. Whereas the final processor will receive the same number of rows that the other processors received (if the total row count of input matrix 1 is equal to the power of 2) or less than that of other processors (if the total row count of input matrix 1 is not in terms of the power of 2).

Unpartitioned Input Matrix 2 is sent to all of the available processors.

From Processor with rank 0, before sending the desired partitioned input matrix 1 and unpartitioned matrix 2, the below parameters are sent as well.

- Input Matrix 1 – Row Count (After Partition)
- Input Matrix 1 – Column Count / Input Matrix 2 – Row Count (Unpartitioned)
- Input Matrix 2 – Column Count (Unpartitioned)

After computation, the result is shared from slave processors to the master processor, which orders the result from different processors (including its own) in the order of processor Id.

Hybrid:

In Hybrid, the base implementation is MPI. On top of MPI, within each of the processors where ever possible, we can parallelize using OpenMP implementation.

Vector Addition:

In Vector addition, Hybrid implementation is done by using **#prama omp parallel for schedule (guided)** inside the MPI Implementation in the following sequences.

- For Partitioning Input Vectors 1 and 2 (In Master Processor) to be sent to different processors.
- For Vector Addition Calculation (In Master as well as slave processors)
- For ordering the resulting datum from different slave processors in the order of Processor ID (In Master Processor)

Matrix Addition:

In Matrix addition, Hybrid implementation is done by using **#prama omp parallel for schedule (guided) collapse(2)** inside the MPI Implementation in the following sequences.

- For Partitioning Input Matrixes 1 and 2 (In Master Processor) to be sent to different processors.
- For Matrix Addition Calculation (In Master as well as slave processors)
- For ordering the resulting datum from different slave processors in the order of Processor ID (In Master Processor)

Matrix Multiplication:

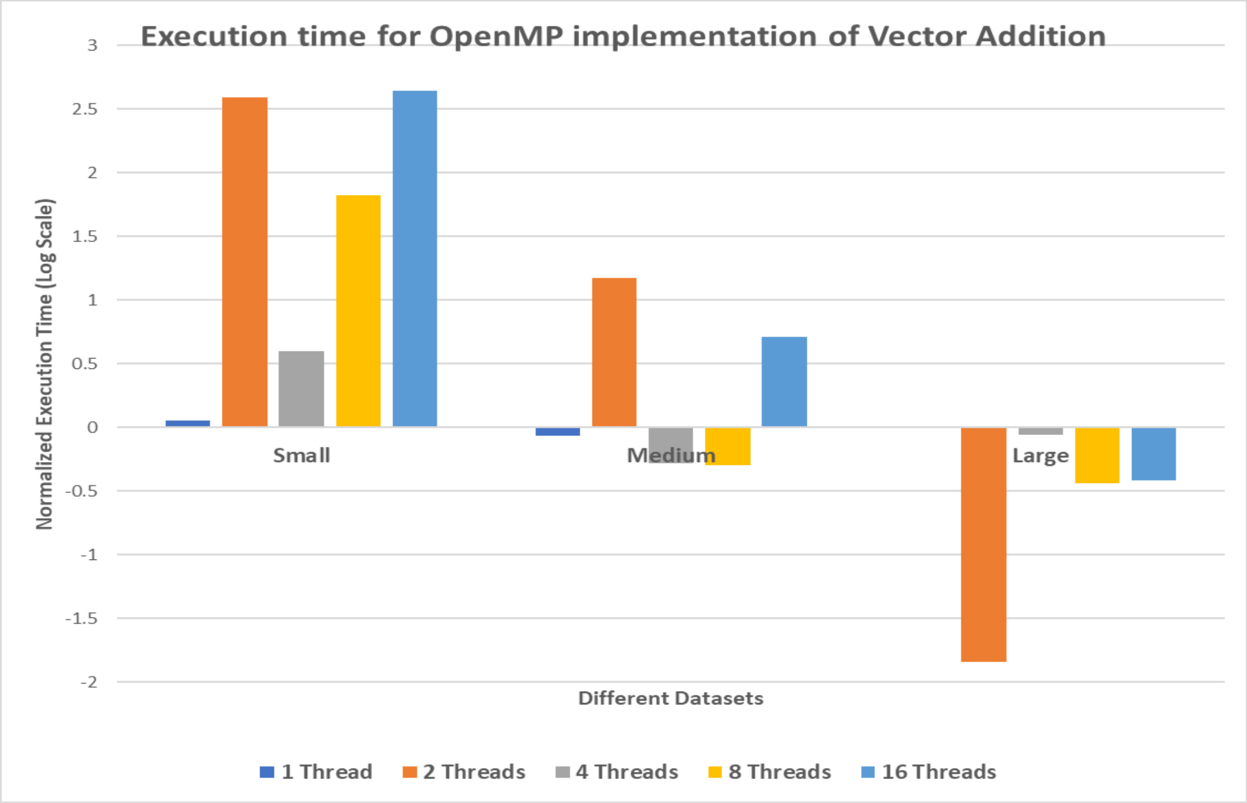
In Matrix multiplication, Hybrid implementation is done by using **#prama omp parallel for schedule (guided) collapse(2)** inside the MPI Implementation in the following sequences.

- For Partitioning Input matrix 1 (In Master Processor) to be sent to different processors.
- For Matrix Multiplication Calculation (In Master as well as slave processors)
- For ordering the result data from different slave processors in the order of Processor ID (In Master Processor)
- **#pragma omp atomic** is also used in Matrix multiplication computation as there is a shared variable among the loops (In the Matrix Multiplication formula itself).

2. Graph Analysis (All Graphs are in LOG SCALE)

a. OpenMP Implementation

- **Vector Addition**



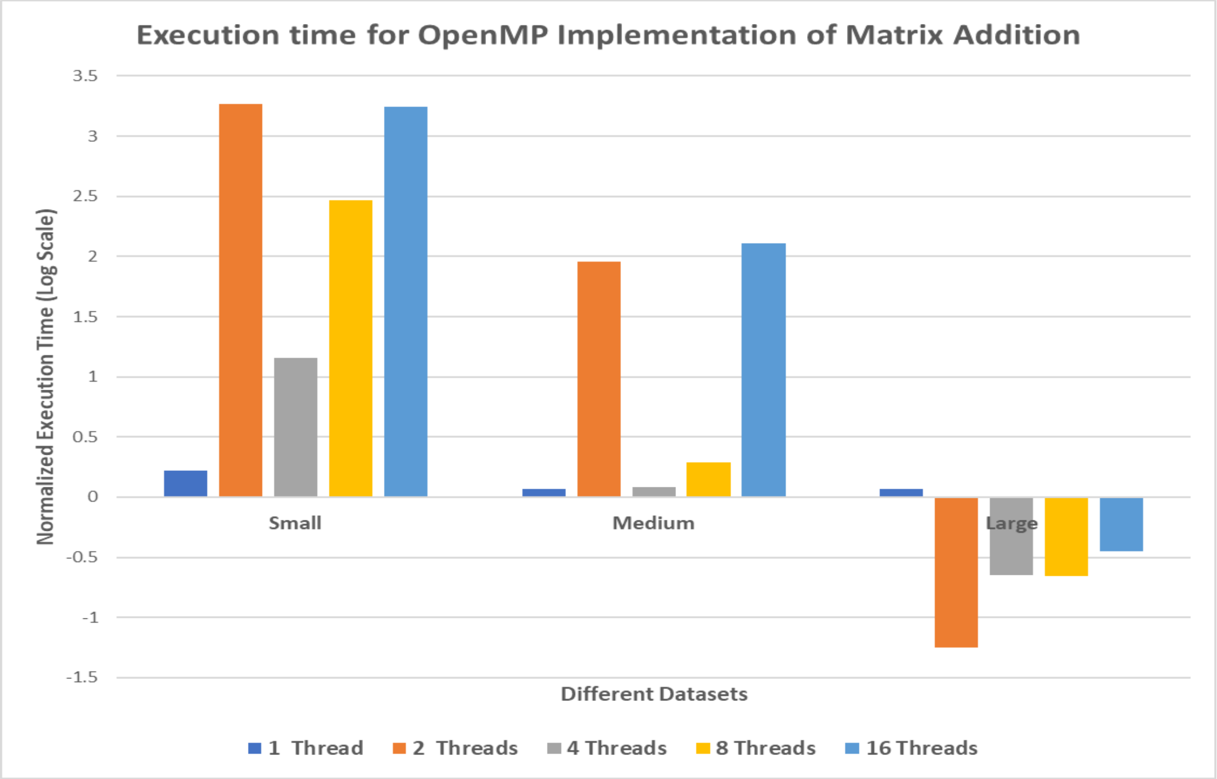
In the above graph, it is inferred that OpenMP is effective in reducing the execution time for vector addition only if the input data sets are comparatively larger. This is because when the input data set is large, the parallelizable computation is large compared to other data sets.

For small data sets and increasing thread count, we can see that the execution time is relatively high which is because the amount of input data is less compared to the resources which are ready/available to do the computation.

We can also infer that for different data sets, with thread count 1, there is no significant difference in execution time as thread count 1 signifies only one thread.

The best execution time is observed for a large set and thread count 2. Even for larger data sets, we can see the execution time is on the increasing trend as thread size increases (but the execution time is less than Serial). This is because the number of parallelizable computations has attained saturation for large data sets at thread count 2.

- **Matrix Addition**



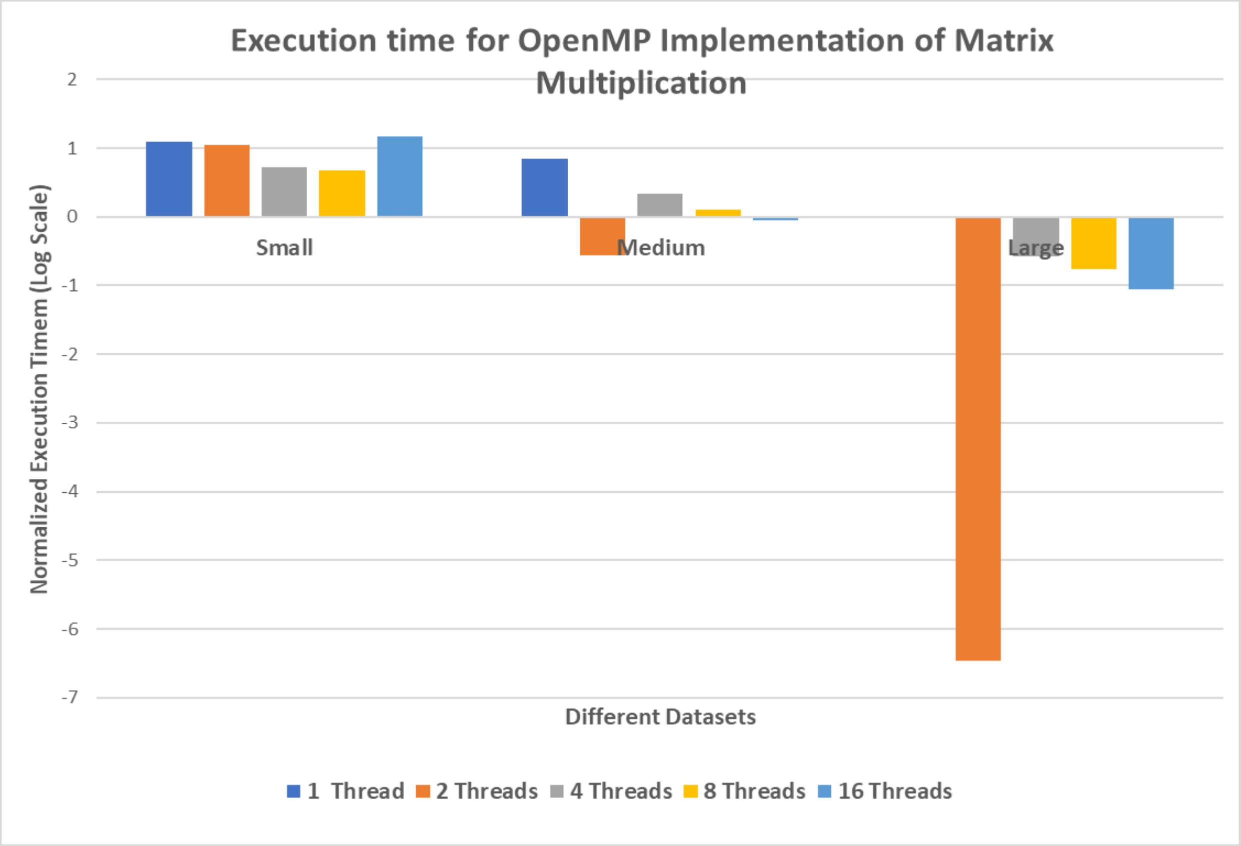
In the graph, it is inferred that OpenMP is effective in reducing the computation time for matrix addition only if the input data sets are larger. This is because when the input data set is large and there is a major chunk of computation that can be parallelized compared to other data sets.

For small data sets and increasing thread count, we can see that the execution time is relatively high which is because the amount of input data is less compared to the resources which are ready to do the computation.

We can also infer that for different data sets, with thread count 1, there is no significant difference in execution time as thread count 1 signifies only one thread.

The best execution time is observed for a large set and thread count 2. Even for larger data sets, we can see the execution time is on the increasing trend as thread size increases (but the execution time is less than Serial). This is because the count of parallelizable computation has attained saturation for large data sets at thread count 2.

- **Matrix Multiplication**



In the graph above, it is inferred that OpenMP is effective in reducing the computation time for matrix multiplication only if the input data sets are medium/larger. This is because when the input data set is medium/large and there is scope for computation that can be parallelized compared to other data sets.

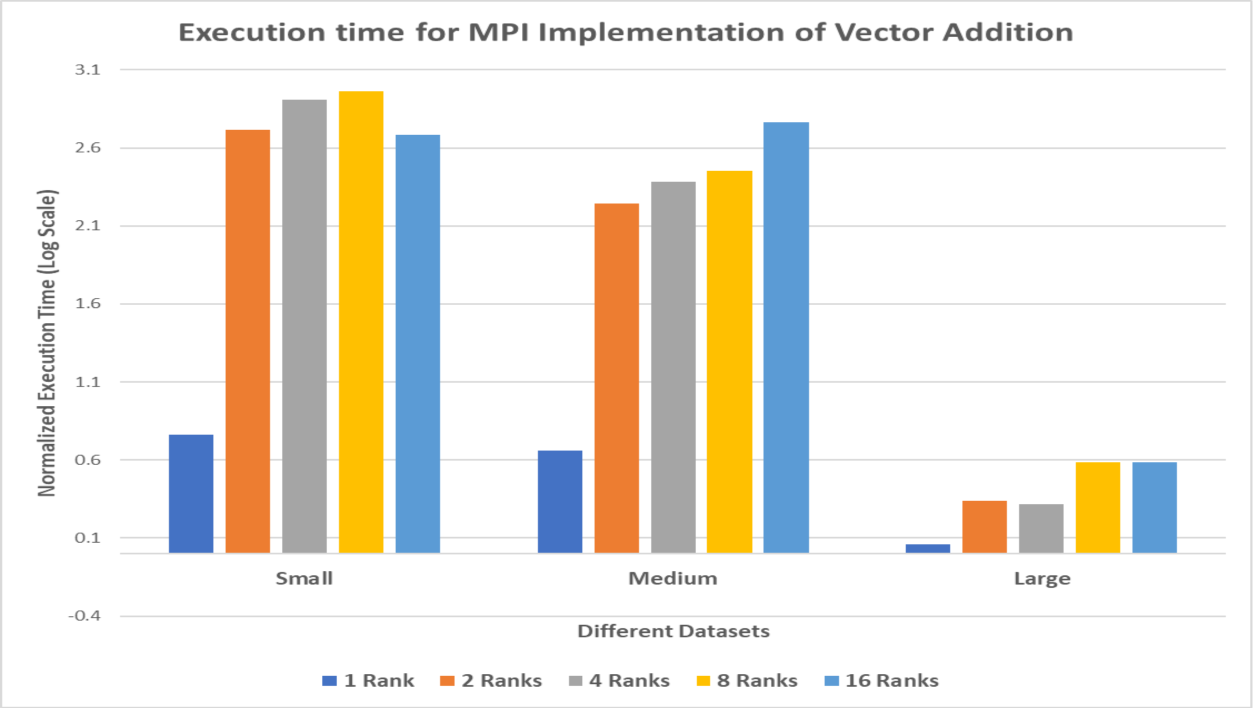
For small data sets and increasing thread count, we can see that the execution time is relatively high compared to serial which is because the amount of input data is less compared to the resources which are ready to do the computation.

We can also infer that for medium data sets, with the thread count increasing, the execution time is approaching the serial time and for thread count 16, the execution time is marginally lower than the serial.

The best execution time is observed for a large set and thread count 2. Even for larger data sets, as thread size increases, we can see the execution time is less than that of thread count 2 (but the execution time is less than Serial). This is because the count of parallelizable computation has attained saturation for large data sets at thread count 2.

b. MPI Implementation

- **Vector Addition**



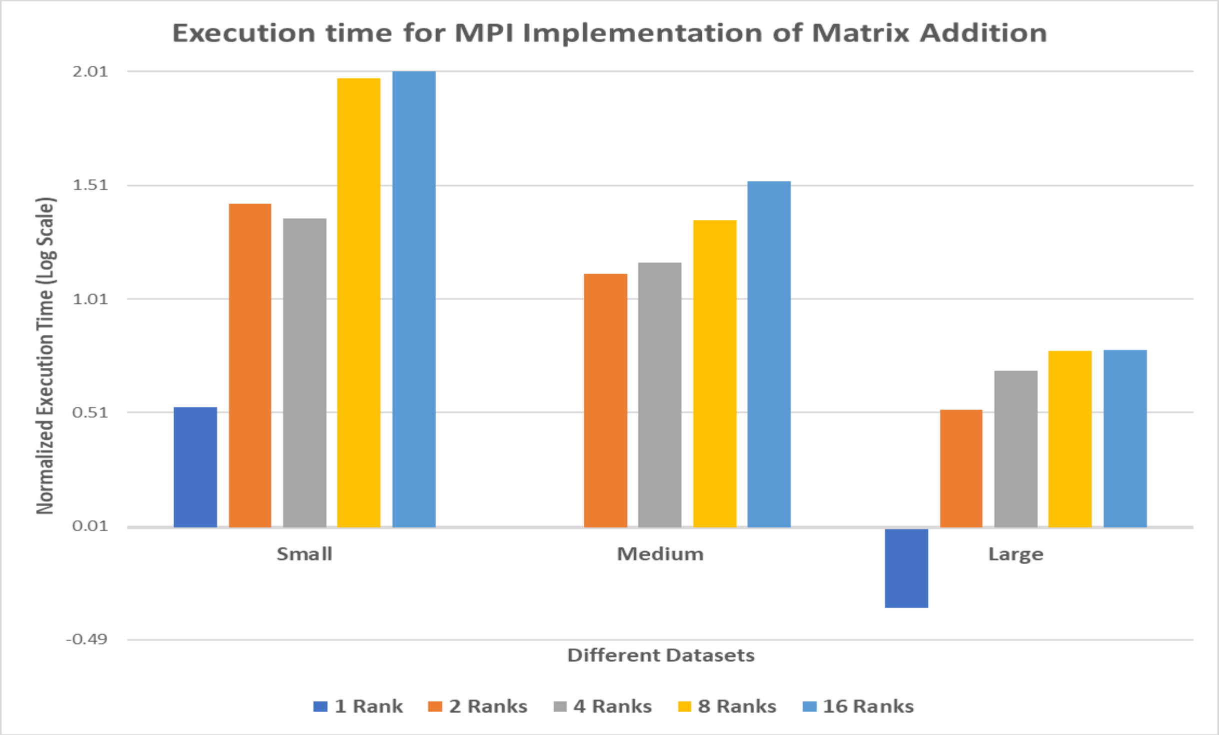
In the graph above, it is inferred that MPI is not effective in reducing the computation time for vector addition as the execution time is not less than that of serial for none of the above scenarios. This is because, as there is an explicit message sharing in MPI, in addition to vector addition computation, the execution time is on the higher side.

We can also observe that for the same data set and with increasing rank (increase in Processor count), the execution time also increases. This is because, as the processor count increases, the slave processor count increases. As a result, the number of explicit message transfers from Master to Slave Processors and vice versa increases, but the number of vector addition computations that can be parallelized is fixed and less, thereby increasing the time.

For small/medium data sets and increasing thread count, we can see that the execution time is relatively high compared to serial which is because the amount of vector addition computation is less compared to the explicit message transfers.

Even for larger data sets, as thread size increases, we can see the execution time increases. This implies that for vector addition, Serial is a much better approach than MPI implementation.

- **Matrix Addition**



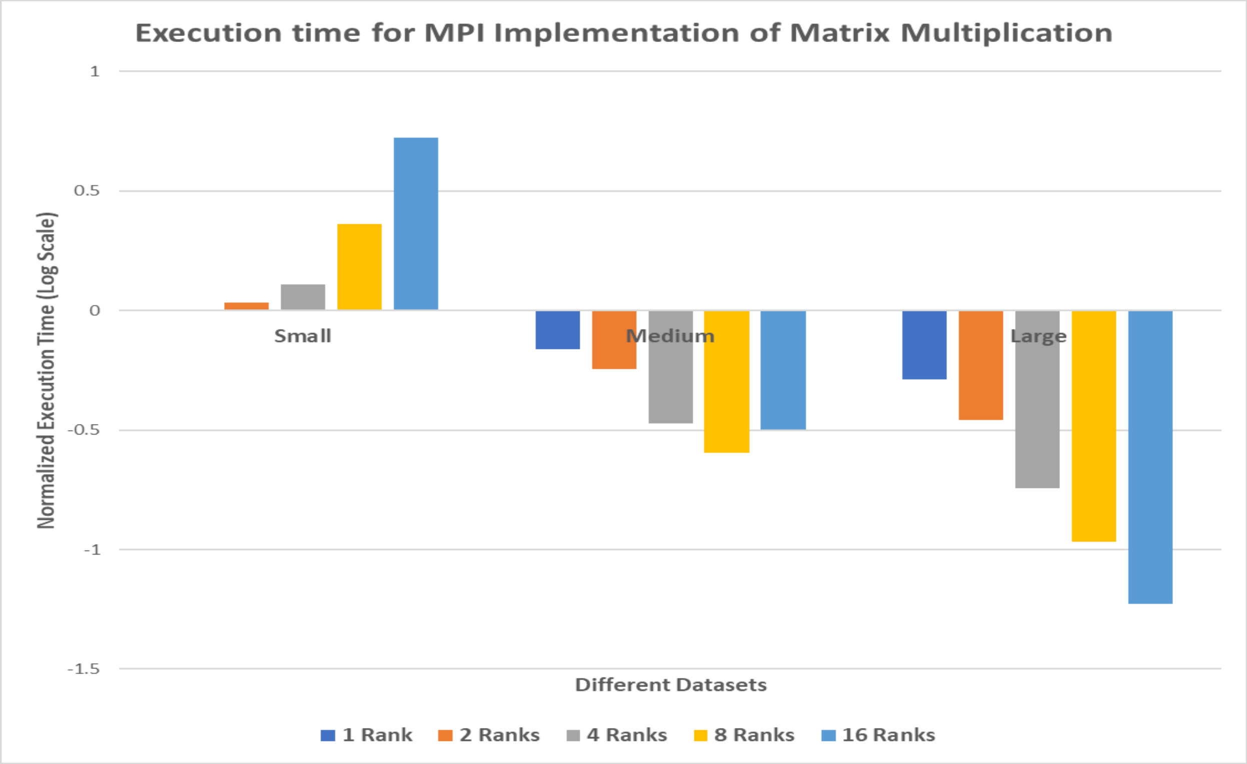
In the above graph, it is inferred that for most of the cases, MPI is not effective in reducing the computation time for matrix addition as the execution time is not less than that of serial except for one scenario (large Data set, Rank -1). This is because, as there is an explicit message sharing in MPI, in addition to matrix addition computation, the execution time is on the higher side.

We can also observe that for the same data set and with increasing rank (increase in Processor count), the execution time also increases. This is because, as the processor count increases, the slave processor count increases. As a result, the number of explicit message transfers from Master to Slave Processors and vice versa increases, but the number of matrix addition computations that can be parallelized is fixed and less, thereby increasing the time.

For small/medium data sets and increasing thread count, we can see that the execution time is relatively high compared to serial which is because the amount of matrix addition computation is less compared to the explicit message transfers.

For larger data sets, as thread size increases, we can see the execution time increases. But for Rank 1, we can see that though the execution time is less than 1, it is comparatively closer to Serial time (Normalized time = 0.53). The cost involved in configuring extra processors is higher compared to this minimal speed-up. This implies that for matrix addition, the Serial method is a much better approach than MPI implementation considering explicit message transfer.

- **Matrix Multiplication**



In the graph above, it is inferred that for large data sets with increasing ranks, MPI implementation is effective in reducing the execution time by nearly 2x compared to serial.

We can also observe that for the small data set and with increasing rank (increase in Processor count), the execution time also increases. This is because, as the processor count increases, the slave processor count increases. As a result, the number of explicit message transfers from Master to Slave Processors and vice versa increases, but the number of matrix multiplication computations that can be parallelized is fixed and less, thereby increasing the time.

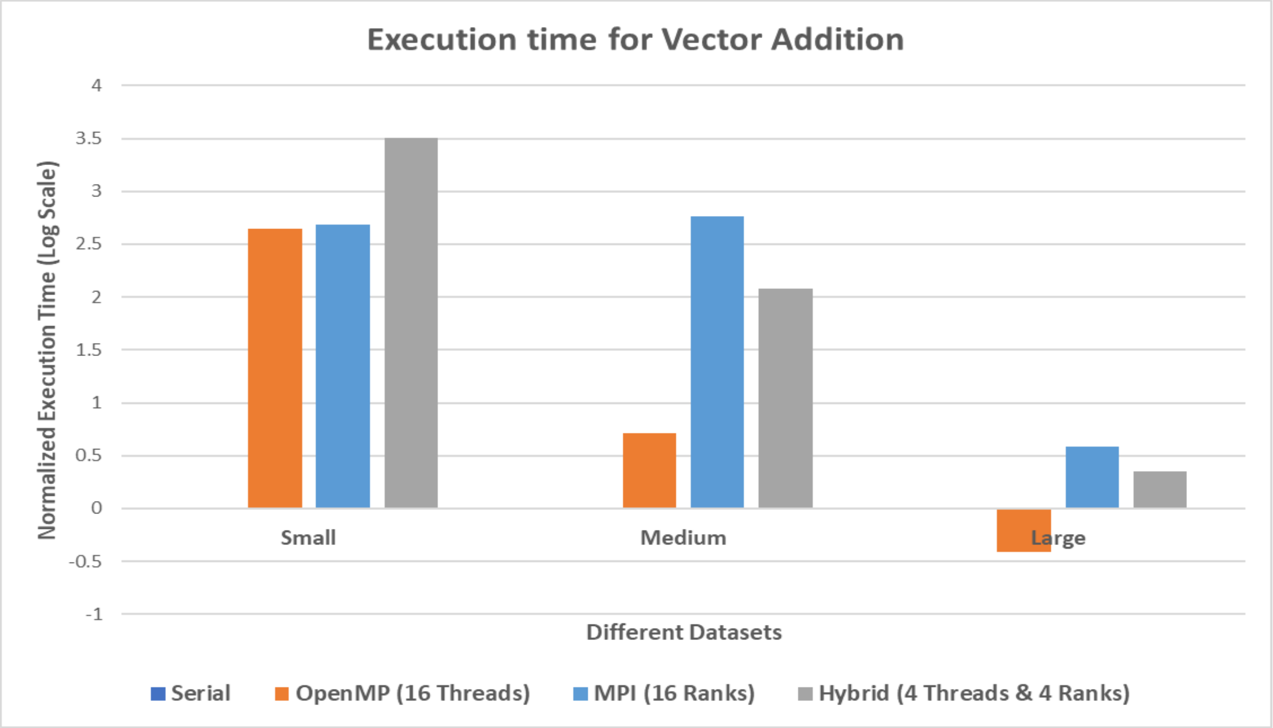
For medium/large data sets and increasing thread count, we can see that the execution time is significantly getting lowered. This is because the number of computations involved in matrix multiplications in these sets is higher and can be parallelized to a greater extent. And this computation outweighs the explicit message transfer among the processors, thereby significantly reducing the execution time.

The best execution time is observed for a large set and thread count of 16. Even for larger data sets, as thread size increases, we can see the execution time is reducing for increasing thread count.

Hence, the cost involved in configuring extra processors is significantly lower compared to the speed-up that we get for matrix multiplication. This implies that for matrix multiplication, MPI implementation is a better approach when considering cost vs performance.

c. Execution Time Analysis

- **Vector Addition**



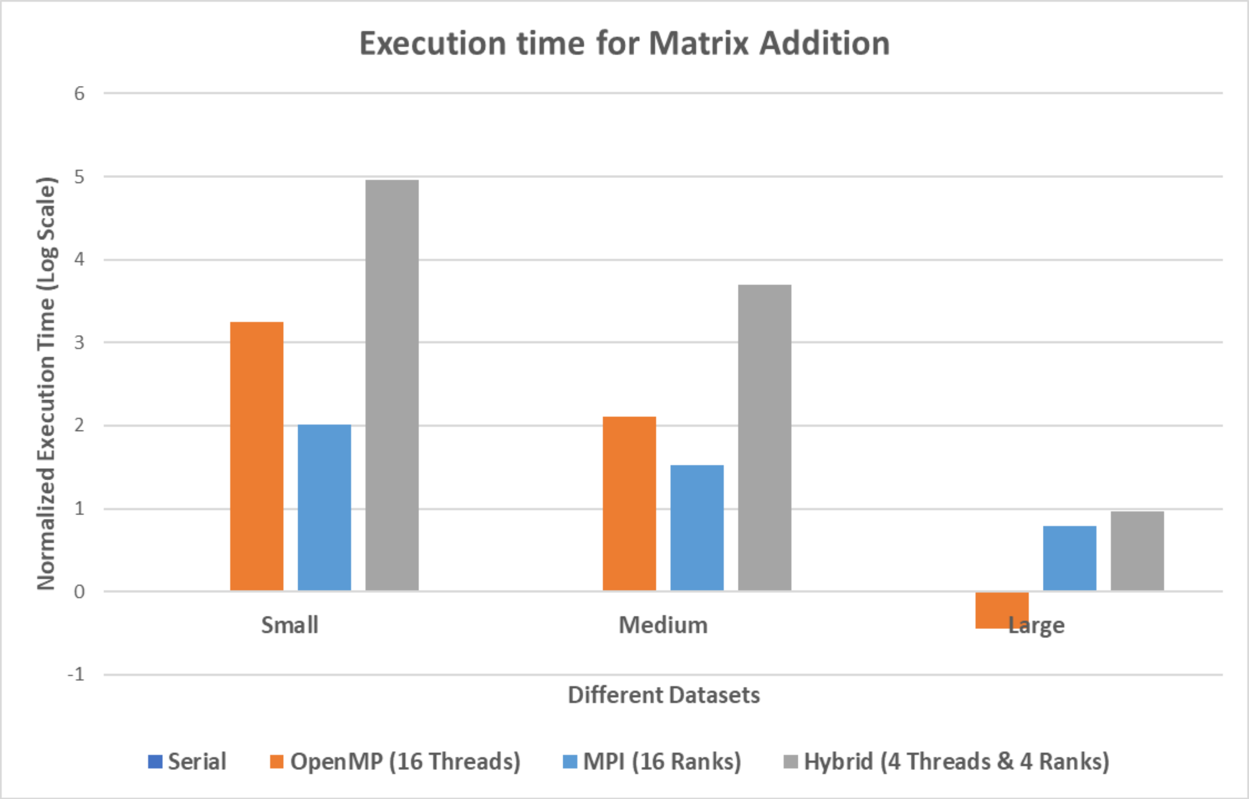
In the graph above, it can be inferred that for vector addition, parallel programming methods of OpenMP alone are effective (that too only for large data sets).

When we consider MPI and Hybrid, they are not effective for any of the three data sets as the execution time is higher than that of serial. This is because the parallelizable Computation involved in Vector Addition calculation is less in comparison to the explicit message transfer involved among the processors. But the difference in time in comparison to serial keeps dropping as size increases.

We can also observe that for MPI and Hybrid, the execution time keeps reducing in comparison to serial as the data set size increases. This implies, as the data set size increases, the number of overheads for explicit data transfer reduces in comparison to the computation involved in Vector Addition.

Also, when we compare MPI and Hybrid, the execution time for Hybrid is less compared to MPI for medium and large data sets though the number of resources allocated to MPI is higher than Hybrid. This is because of some OpenMP parallelization implemented inside each of the processors.

- **Matrix Addition**



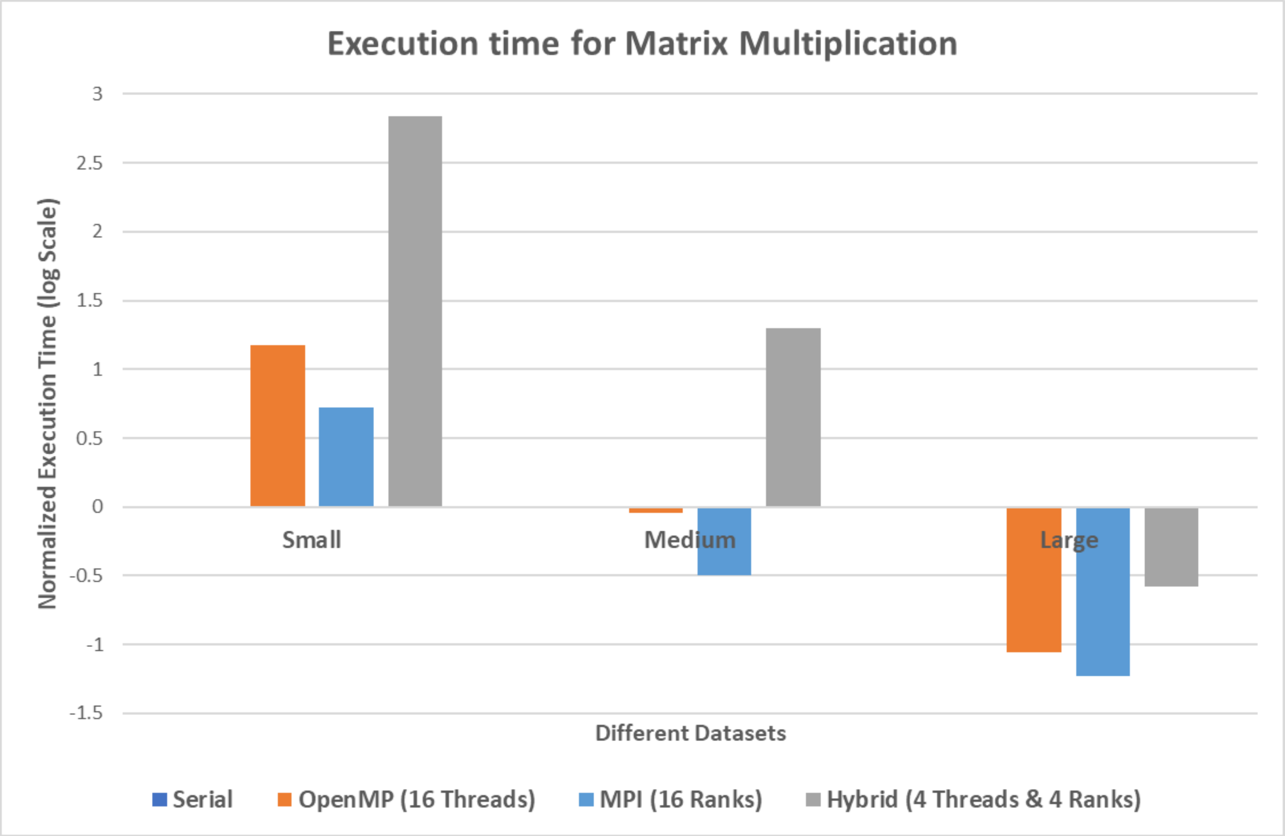
In the above graph, it can be inferred that for matrix addition, parallel programming methods of only OpenMP are effective (that too only for large data sets).

When we consider MPI and Hybrid, they are not effective for any of the three data sets as the execution time is higher than that of serial. This is because the parallelizable Computation involved in Matrix Addition calculation is less in comparison to the explicit message transfer involved among the processors. But the difference in time in comparison to serial keeps dropping as size increases.

We can also observe that for MPI and Hybrid, the execution time keeps reducing in comparison to serial as the data set size increases. This implies, as the data set size increases, the number of overheads for explicit data transfer reduces in comparison to the computation involved in Matrix Addition.

Also, when we compare MPI and Hybrid, the execution time for Hybrid is not less compared to MPI for any of the data sets. But the difference in execution times keeps reducing as the data set size increases. When we approach a large data set, MPI and Hybrid have almost the same execution time. This is because of some OpenMP parallelization schemes implemented in Hybrid although there is some difference in resources allotted to MPI and Hybrid.

- **Matrix Multiplication**



In the above graph, it can be inferred that for matrix multiplication, parallel programming methods of OpenMP, MPI, and Hybrid are effective (for medium and large data sets).

When we consider all three parallel programming methods, they are not effective for small data sets as the execution time is higher than that of serial. This is because the parallelizable Computation involved in Matrix Multiplication calculation is less in comparison to the explicit message transfer involved among the processors.

We can also observe that for MPI, the execution time keeps reducing in comparison to serial as the data set size increases. This implies, as the data set size increases, the number of overheads for explicit data transfer reduces in comparison to the computation involved in Matrix Multiplication. For Large data sets, among the three, we can say that MPI is the most effective one.

Also, when we compare MPI and Hybrid, the execution time for Hybrid is not less compared to MPI for any of the data sets. But the difference in execution times keeps reducing as the data set size increases. When we approach large data set, both MPI and Hybrid have better execution times than serial, but MPI is more effective than Hybrid. This is because of the configuration of Ranks in MPI and Ranks/Threads in Hybrid.

CONCLUSION

From the above results, we can say that implementing parallelization is

- Application/Computation specific
- Too much of parallelization resources lead to a larger execution time.