Assignment 2 - Matrix Vector Multiplication

Ayushi Agarwal
2018ANZ8503
ayushi.agarwal@cse.iitd.ac.in

Important : Please skip to Section 3 for Experimental Results

1 Introduction

The goal of this document is to provide an overview of the strategy to design a **parallel Matrix-Vector Multiplication** algorithm. The final objective of the project is to assess the performance of Matrix Vector Multiplication[1] on shared and distributed memory frameworks using OpenMP and MPI.

2 Method

Our objective is to :

- Establish a baseline by implementing a serial matrix-vector multiplication algorithm.
- Implement three techniques to parallel Matrix-Vector Algorithm :
  - Row-Striped Matrix Data Decomposition
  - Column-Striped Matrix Data Decomposition
  - Checkerboard Block Matrix Data Decomposition
- These three types of data decomposition only consider the Matrix. The Vector can also be decomposed. The Vector can either be replicated or can be block decomposed.

2.1 Sequential Algorithm

Algorithm 1 shows a sequential implementation of a Matrix-Vector multiplication. The inner loop has a complexity of $O(n)$ as there are *n* multiplications and *(n-1)* additions. The inner loop is executed *m* times, so the complexity of the algorithm is $O(mn)$ and if $m = n$ then the complexity is $O(n^2)$. The complexity of the algorithm can be improved if we explicitly make our algorithm parallel so that different parts of the algorithm can run in parallel on different computational units.

Algorithm 1: Sequential Matrix-Vector Multiplication

```
Data: A[m][n] and B[n]
Result: C[m]
for i ← 0 to m − 1 do
    C[i] ← 0;
    for j ← 0 to n − 1 do
        C[i] ← C[i] + A[i][j] ∗ B[j];
    end
end
```

2.2 Parallel Algorithm

- According to Foster task/channel model, the first step to create a parallel algorithm is to do *Data Decomposition/-Partitioning* and then schedule the tasks on the available processors.

- The data decomposition should be done in such a way that all the created tasks are roughly of the same size. This helps in load balancing when the tasks are scheduled on the processors.

- The data should be divided among the processors in such a way that the communication between the processors is minimized. Communication between processors is the biggest overhead of a parallel algorithm over sequential (which doesn't need any inter-processor communication). So, the goal should be to minimize it.

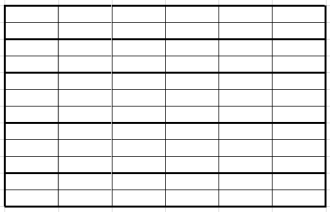
- The algorithm should be scalable. This is evaluated using the *Iso-efficiency Metric*.

2.2.1 Data Decomposition

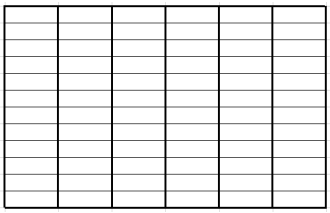
As mentioned in Section 2 and shown in Figure 1 there are three options for data decomposition for the matrix:

1. Row-wise Block Striping: In this method we divide the matrix into a group of rows which is dependent on the number of processes we want. If there are *p* processes, then each process will be responsible for a contiguous group of either $\lceil \frac{m}{p} \rceil$ or $\lfloor \frac{m}{p} \rfloor$ rows.

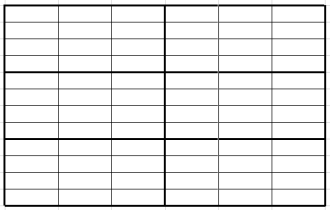
2. Column-wise Block Striping: In this method, we divide the matrix into a group of columns. If there



(a) Row-Wise



(b) Column wise



(c) Checkerboard

Figure 1: Matrix Data Decomposition

are p processes, then each process will be responsible for a contiguous group of either $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$ columns.

3. Checkerboard Block: In this method we divide the matrix into a grid of rows and columns similar to 2D blocks. If each block has *r* rows and *c* columns, then each process will be responsible for a matrix block having $\lceil \frac{m}{r} \rceil$ rows and $\lceil \frac{n}{c} \rceil$ columns. For example, if we have a matrix of size 10x10 and we want to divide it into blocks of 3x3 ($r = 3, c = 3$), then the partitioned blocks would be of sizes 3x3, or 4x3, or 4x4, or 3x4, that is, the maximum possible row and column size would be $\lceil \frac{10}{3} \rceil$.

The vector can also be decomposed among the processes. To store the vector input and the output, we can explore two options:

1. Replicate vector elements on all the processes. The vector only has *n* elements as opposed to the n^2 elements in the matrix when $m = n$.
   (a) If we store a row or column of the matrix and store only one element of the vector in one process, then the space complexity is $O(n)$.
   (b) If we store a row or column of the matrix and the whole vector, then we are storing $2n$ elements per process, so the complexity is still $O(n)$.

   Hence, it is acceptable to store/replicate the complete vector on all the process but we still decompose the matrix.
2. Divide the vector into blocks. Each process will be responsible for storing contiguous blocks of size $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$. We will see that this method also requires extra communication to transfer the vector elements among processes.

Since, there are 3 ways to decompose the matrix elements and 2 ways to decompose the vector elements, we have 6 possible combinations in which we can partition our problem into processes. This report is to evaluate the performance of these cases on shared memory and distributed memory systems by using OpenMP and OpenMPI.

2.2.2 Row-wise Block Striped Matrix Decomposition

As shown in Figure 1(a), we divide the matrix into a set of contiguous rows. Each primitive task *i* would be responsible for row *i*. As shown in Algorithm 1, to calculate the inner products of the result vector, we multiply a row of the matrix with the entire vector. So, ***in row-wise striping of the matrix where each task is storing a row or group of rows of the matrix, it is better to replicate the complete vector in each process or primitive task***. This way the task *i* would store the inner products for the row/rows *i* and the vector. This method saves a lot of initial communication between the processes. Because if each task stores only one element of the vector then each task would have to receive the other elements of the vector from other processes to be able to compute one inner product of the result.

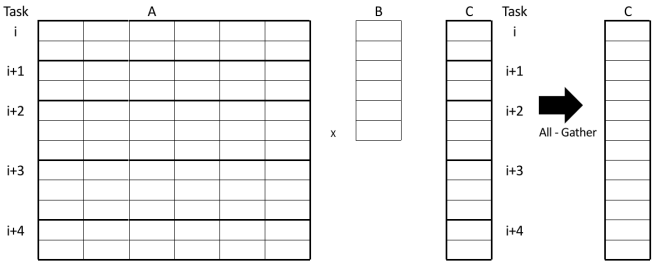


Figure 2: Row-wise striped, replicated vector, Parallel Matrix-Vector Multiplication

Figure 2 shows the algorithm for row-wise striped parallel matrix vector multiplication. Each task contains a set of contiguous rows of the Matrix A. We have assumed 5 tasks (processors) in this example. Each task has a copy of the Vector B. Each task computes its set of contiguous or block of inner products of result vector C. After all the tasks have computed their inner products (implemented as a barrier for synchronization), we perform an All-Gather for the result vector C.
At the end all the processes have the result vector C.

Complexity Analysis of Parallel Algorithm:

- **Computational Complexity:** Each task multiplies the vector B (size *n*), with a set of rows of Matrix A

(maximum number of rows = $\lceil \frac{m}{p} \rceil$ as seen in Section 2.2.1). Each processor is also responsible to add *n* partial sums with complexity $O(n)$. If $m = n$, the overall computational complexity is $O(\frac{n^2}{p})$.

- **Communication Complexity:** In an efficient All-Gather communication, each processor has to send $\lceil log\ p \rceil$ messages[1] (assuming there are *p* processors). For an all gather of the result vector C, each processor will have to send its part of $\lceil \frac{n}{p} \rceil$ (assuming $m = n$) values of the result to $p - 1$ processors. So the total number of elements passed in the network are $\frac{n(p-1)}{p}$.

  The communication complexity is therefore $O(n + log\ p)$.

Handling I/O : We Assume that we are reading the matrix from a file storing it in Row major order and another file storing the vector. We handle the I/O using any one of the processors only. So, one processor will read the file and communicate the data according to the decomposition scheme to other processors.

1. **Input Matrix** : One processor opens the file storing the matrix. Each row is read from the processor and sent to the processor responsible for handling the row (row *i* needs to go to task *i*).

2. **Replicated vector** : One processor will read the file storing the vector and broadcast it to all the other processors.

3. **Output Vector** : The I/O should mostly be handled by one processor to avoid scrambling of the output messages. Since, the output vector is present in all the processors at the end, any one can be used to print the output vector.

2.2.3 Column-Wise Block Striped Matrix Decomposition

As shown in Figure 1(b), we divide the matrix into a set of contiguous columns. Each primitive task *i* is responsible for column *i* or a block of columns mapped as *i*. Since, each task has a column *i* that needs to be multiplied to element *i* of the vector, hence replicating of the whole vector on all the processors is not needed. All processors save only one element or a block of elements of the vector depending on the number of columns in the partition.

Figure 3 shows a rough overview of the column-wise striped algorithm. In this example, each task contains one column of the matrix A and one element of the vector B. So each task produces a part of the inner product of the result vector i.e. a vector of partial results. So, an all-to-all exchange is done to shift the partial result *j* to task *j*. For example, Task 0 will have the first partial sum (sum0) of all the inner products of the vector C. To compute the

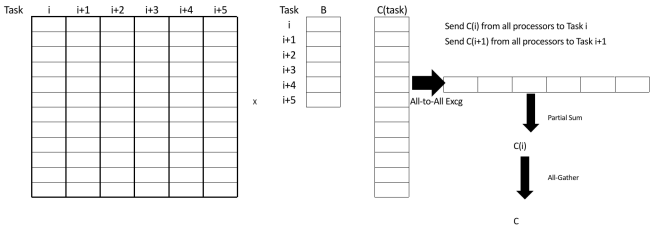


Figure 3: Column-wise striped, decomposed vector, Parallel Matrix-Vector Multiplication

first inner product of C, all the tasks need to send their first partial sum(sum0) to task 0. At the end of this, all processors will have a part of the result vector C. Then an All-Gather communication is used to collect the result vector in all the processors.
At the end, all processors will have a copy of the result vector C.

Complexity Analysis:

- **Computational Complexity:** Each task multiplies the vector element of B, with a set of columns of Matrix A (maximum number of columns per task/elements of B per task = $\lceil \frac{n}{p} \rceil$ as seen in Section 2.2.1). The computational complexity is $O(\frac{n^2}{p})$. Then each processor also adds *p* partial vectors of maximum length $\lceil \frac{n}{p} \rceil$ with complexity $O(n)$. Hence overall complexity remains same.

- **Communication Complexity:** In an efficient All-to-All exchange communication, each processor has to send and receive $n/2$ values in $\lceil log\ p \rceil$ messages[1] (assuming there are *p* processors). So the total number of elements moved are $n\lceil log\ p \rceil$.

  The communication complexity is therefore $O(nlog\ p)$.

Handling I/O : We Assume that we are reading the matrix from a file storing it in Row major order and another file storing the vector. We handle the I/O using any one of the processors only. So, one processor will read the file and communicate the data according to the decomposition scheme to other processors.

1. **Input Matrix** : One processor opens the file storing the matrix. Each row is read from the processor which scatters parts of the rows to the tasks associated with handling the respective columns. So, as in Figure 3 processor *0* will read the rows and broadcast one element each to the 6 tasks respectively.This is called Scatter operation.

2. **Replicated vector** : One processor will read the file storing the vector and send only a block of vector of size $\lceil \frac{n}{p} \rceil$ to the corresponding processor.

3. **Output Vector** : The I/O should mostly be handled by one processor to avoid scrambling of the output

messages. Since, the output vector is present in all the processors at the end, any one can be used to print the output vector.

2.2.4 Checkerboard Block Decomposition

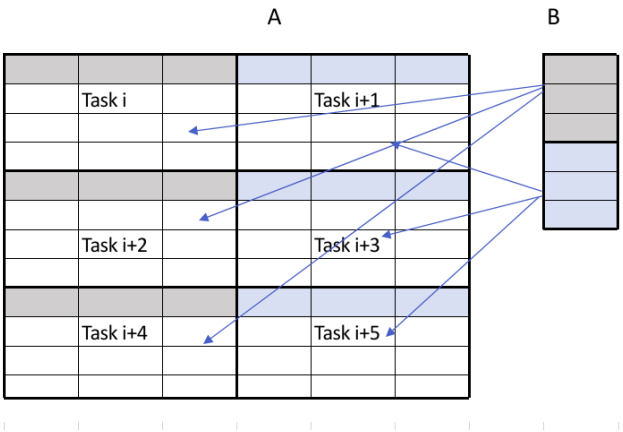


Figure 4: Checkerboard Decomposition for Parallel Matrix-Vector Multiplication

As shown in Figure 1(c), we divide the matrix into blocks of rows and columns. Vectors are also distributed by blocks among all the tasks. So each process now works on a 2D sub-grid of matrix. In Figure 4, we have shown how the broadcast of the Vector B would take place across the tasks. We have assumed 6 tasks (6 processors). Each task would multiply the block $A_{i,j}$ with B_j. We redistribute vector B in the following two ways, assuming that the grid of *p* tasks is of size *k*x*l*(In the figure, $k = 3, l = 2$):

- If $k = l$, then the distribution of B is fairly easier. In the first step, the vector B is divided among the k tasks. The tasks at grid position $(i, 0)$ sends its portion of B to the task at grid position $(0, i)$. After this first step, each process in the first row of the task grid has their equivalent portions of B. In the second step, the processes in the first row of the task grid broadcast their portion of B to the other tasks in the same column of the grid.

- If $k \neq l$ (as is in our example), then the distribution involves first gathering the complete Vector B in task at grid position $(0, 0)$. Then we scatter the Vector B among the tasks in the first row of the task grid (hence dividing B among *l* tasks. Then the processes in the first row of the grid will broadcast the blocks of vector B to the other tasks in the same column of the grid.

After the computation, each task would store partial sums of the result vector. So, all the tasks in the same row of the task grid will perform a sum-reduction on their portions of result vector C and store it in the first column of the task grid.

Complexity Analysis: We will assume that $m = n$ and that $k = l = \sqrt{p}$ so we have a square grid of processes. Since p is a square, the maximum matrix block size would be $\lceil \frac{n}{\sqrt{p}} \rceil$ x $\lceil \frac{n}{\sqrt{p}} \rceil$.

- **Computational Complexity:** The computational complexity would be $O(\frac{n^2}{p})$.

- **Communication Complexity:** When p is a square, we first distribute B from the first column to the first row. So we basically move $\lceil \frac{n}{\sqrt{p}} \rceil$ in this step. Hence complexity for this step is $O(\frac{n}{\sqrt{p}})$. In the next step each process in the first row broadcasts the whole portion of B to tasks in its column. So $\frac{n}{\sqrt{p}}$ values are moved among $\sqrt{p}$ tasks in $log(\sqrt{p})$ steps. So the complexity is $O(log(\sqrt{p})\frac{n}{\sqrt{p}})$. Similarly, during reduction, $\frac{n}{\sqrt{p}}$ values are moved among $\sqrt{p}$ tasks in $log(\sqrt{p})$ steps. So complexity for reduction is $O(log(\sqrt{p})\frac{n}{\sqrt{p}})$. The overall communication complexity is $O((log\ p)\frac{n}{\sqrt{p}})$.

3 Experimental Setup and Results

We have implemented all the three variations of matrix-vector multiplication in both OpenMP and OpenMPI using C++[I].

- The results are averaged over 100 execution times.

- We have run the code on different machines and explored the speedup obtained by the parallel version of the multiplication with the serial implementation.

- We have also compared the execution time improvement of the parallel implementation with increase in the number of processors.

3.1 Machine - Intel i7-7700

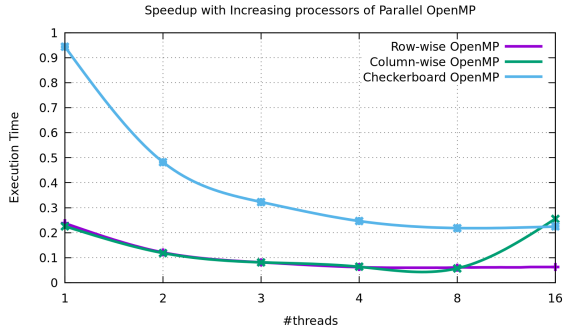
This system has 4 physical cores and 2 threads per core. So it has 8 logical cores. It is a one socket machine.

As shown in Figure 5(b) we see that the OpenMP Implementation for the row-wise striped matrix is better than the column wise striped because in column wise we have to update the final result matrix elements in lock-step since all the individual threads only execute partial multiplication of the rows with columns. However, in row-wise all the threads individually have one final result element so there is no lock-step between the threads.

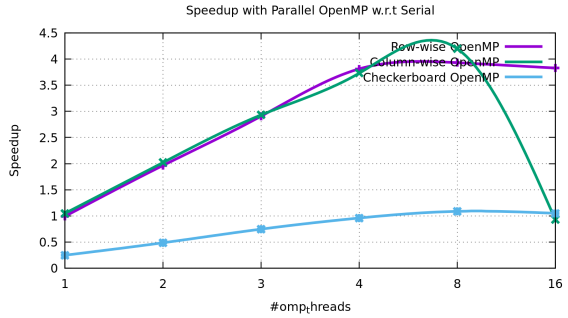
The checkerboard implementation needs extra work to create blocks of matrix even when the matrix is shared among all threads. This is because, even though the matrix is seen by all threads, each thread has to know what block of data to work on. So checkerboard implementation also turns out to be worse than row-wise here as compared to serial implementation.

[I]We will release the code-base here: `https://github.com/agarwal-ayushi/Parallel-Matrix-Vector-Multiplication`

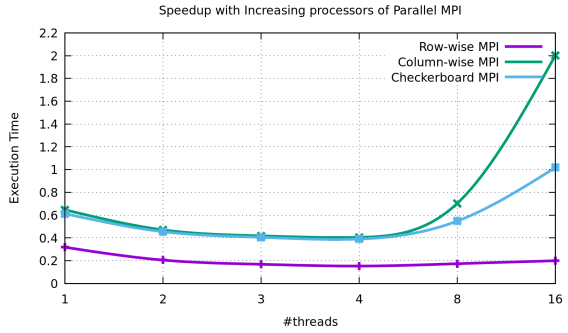
- However, Figure 5(a) shows that the checkerboard implementation is the most scalable implementation amongst the three. Since its execution time keep decreasing as the number of processes increase. We see an increase in the execution time for Column-Wise method because of the lock-step. So as the number of Processes increase the time to complete the result calculation in lock-step increases.



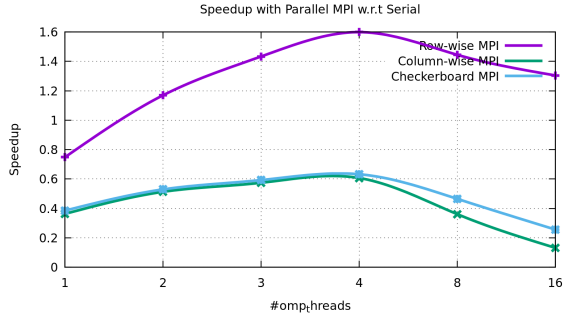
(a) Execution Time with Increasing Number of processes - OPENMP



(b) Speedup of OpenMP w.r.t Serial Implementation



(c) Execution Time with Increasing Number of processes - MPI



(d) Speedup of MPI w.r.t Serial Implementation

Figure 5: Results for Matrix 10000x10000 and Vector 10000x1 on Machine Intel i7-7700 with 8 logical cores

As shown in Figure 5(d), we only see a speedup in the parallel MPI implementation upto 4 processes. This is because MPI implementation, every process runs on a different physical core. So, as we exceed 4, we see a reduced efficiency of these algorithms.

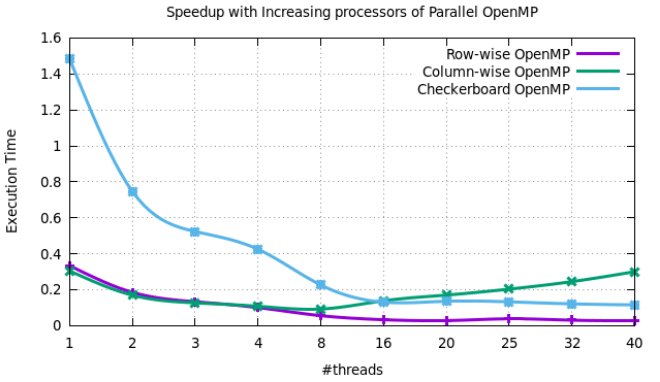
Figure 5(c) shows that since this machine is only a single socket, we don't see much scalability in these algorithms.

We have examined our results for various matrix and vector sizes and we have observed similar behaviour on this machine.

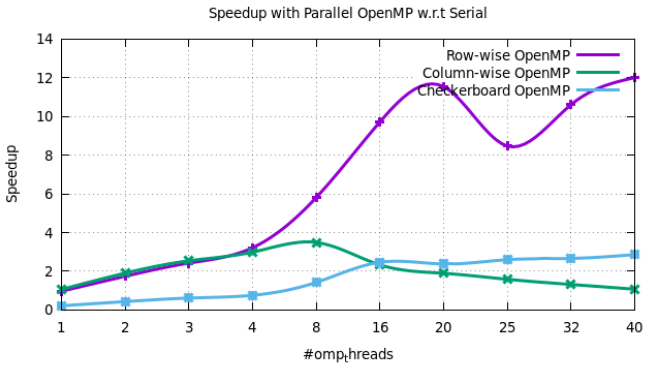
3.2 Machine - Intel(R) Xeon(R) CPU E5-2630 v4

This machine has 2 sockets and 10 physical cores per socket and 2 threads per core. So, there are 40 logical cores.

We see similar results for OpenMP implementations. We have used matrix size of 10000. Figure 6 (a) and (b) show that the speedup w.r.t serial implementation is best for row-wise striped matrix. However, the checkerboard implementation achieves a 2x reduction in execution time with 40 processes. So it the most scalable implementation.



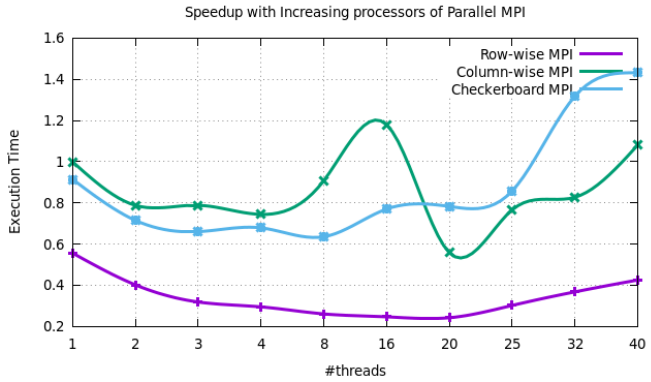
(a) Execution Time with Increasing Number of processes - OPENMP



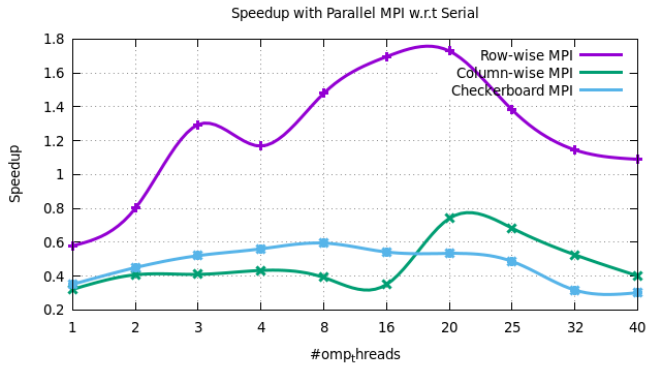
(b) Speedup of OpenMP w.r.t Serial Implementation

Figure 6: Results for Matrix 10000x10000 and Vector 10000x1 on Machine Intel(R) Xeon(R) CPU E5-2630 v4 with 40 logical cores

Figure 6 (c) and (d) show the MPI implementation of MVM on this machine. The speedup is maximum for row-wise implementation algorithm. The speedup reduces as soon as we cross the number of physical cores (which is 20) in the machine.

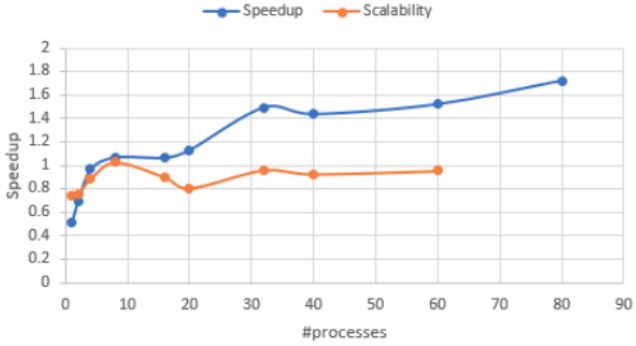


(c) Execution Time with Increasing Number of processes - MPI

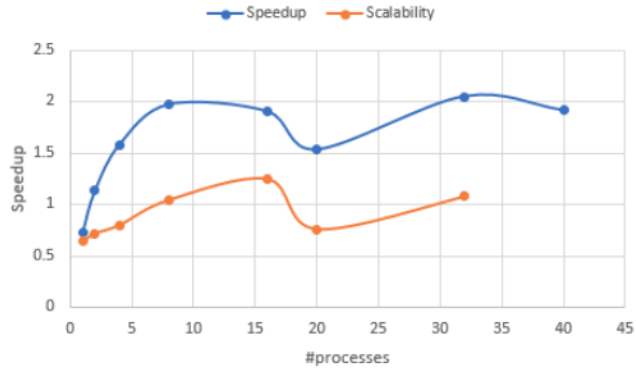


(d) Speedup of MPI w.r.t Serial Implementation

Figure 6: Results for Matrix 10000x10000 and Vector 10000x1 on Machine Intel(R) Xeon(R) CPU E5-2630 v4 with 40 logical cores (contd.)



(a) Results obtained on 8 Nodes, 10 MPI tasks/node and 10 CPUs/node



(b) Results obtained on 4 Nodes, 10 MPI tasks/node and 10 CPUs/node

Figure 7: Results for Matrix 10000x10000 and Vector 10000x1 for Row-wise Striped MVM implementation on HPC clusters

3.3 HPC Clusters

We haven't been able to complete our experiments on the HPC clusters. The idea is to use different number of nodes in the HPC clusters which actually have no shared memory to study the MPI implementation of MVM algorithm. Each node can run a subset of total Mpitasks.

So we need to schedule task in the form of $< number\ of\ nodes >:< number\ of\ cpus/node >:< number\ of\ mpitasks/node >$. So if the task needs to use number of processes equal to 40, we can used 4 nodes and 10 mpitasks per node. The performance also depends on the underlying MPI architecture which includes the kind of interconnect between nodes.

Figure 7 shows the implementation of row-wise striped MPI matrix-Vector multiplication on HPC clusters. The setup is that we have used 4 and 8 number of nodes and 10 cpu/nodes. We also have used 10 mpitasks per node so the total number of processes in these two cases would be 40 and 80 respectively. We notice that the speedup obtained on a 4-node steup is higher as compared to a 8-node setup. We predict that this is because of higher number of communications. However this is not completely true. Because if we consider 20 processes, MPI would first schedule 10 mpitasks on one node and then move to a second node. So in both cases, effectively only 2 nodes should have been used.

4 Conclusions

We see that the parallel implementation of matrix-vector multiplication is efficient as well as scalable for increasing number of processors. Extension : HPC cluster results

5 References

1. Parallel Programming in C with MPI and OpenMP