Parallel Programming - Matrix Multiplication

Keerthana C J

June 14, 2021

1 Introduction

The main objective of the exercise is to parallelize matrix multiplication and to understand the various aspects to consider during parallelization. A MPI version and a Hybrid MPI-OpenMP version are implemented for performing square matrix - matrix multiplication. We try to understand the scaling of the code across different processors and nodes. In order to achieve a better calculation efficiency of multiplying the matrices held within each processor, we use the LAPACK & BLAS libraries. We try to understand how a reduction in computation time using an efficient computational library can lead to change in the scaling of the code. As in the exercise of Jacobi method, where an order of magnitude in the size of the matrix changes the scaling, we try to determine which sizes of the matrix produce a change in the scaling. We try to define small and big sizes of the matrix, where the small size is expected not to scale and the big size is expected to scale almost linearly.

2 Matrix multiplication

The matrix - matrix multiplication is a commonly used problem for bench-marking. The problem involves the calculation of a resultant matrix from the multiplication of two input matrices. A naive algorithm for such multiplication has an asymptotic complexity of $O(m.n.p)$, and if the matrices are square we have $O(n^3)$ complexity. It is a compute bound problem. A naive serial algorithm for square matrix-matrix multiplication of size n involves the code:

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    for(k=0; k<n; k++)
      C[i][j] += A[i][k]*B[k][j]
```

3 Parallelization

To do a matrix multiplication in parallel, where the A and B matrices are distributed among the processors, each row of A needs to be multiplied by each column of B. Every processor has entire rows of A and B, but doesn't have an entire column of B. The columns of B

are spread among the processors which have to be collected and then multiplied with the rows of A. We can gather the columns of B one by one, or we can gather a few columns of B everytime and perform the multiplication with rows of A in each processor. A simple pseudocode for working out the multiplication can be given as follows:

```
for(i=0; i<np; i++)
  MPI_Allgather(From: B_loc, Store in: B_temp);
  Multiply(A_loc, B_temp, C_loc);
```

The mulitplication of *A_loc* and *B_temp* can be done with a naive implementation of rectangular matrix multiplication or we can use the BLAS libraries to perform an efficient matrix multiplication. We implement both versions and see how the reduction in computation time affects the scaling.

4 Theoretical performance & Efficiency

The experiments are carried out in the super computer cluster, Ulysses at SISSA. The nodes used for the computation have Intel Xenon CPU E5-2680 v2 processors. Each node contains 20 processors. Hence the theoretical performance is calculated as:

$$Theoretical\,peak\,performance = No.\,of\,procs \times Frequency \times FLOPS/cycle \tag{1}$$

The Xenon E5-2680 has *Sandy Bridge* architecture, hence it does $8FLOPS/cycle$. The frequency of the processors is $2.8GHz$. Hence subsituting the values each node has a theoretical peak performance of $448GFLOPS/sec$. The actual performance is calculated as:

$$Actual\,performance = \frac{Actual\,FLOPS\,doneby\,per\,proc \times procs}{Time\,for\,calculation} \tag{2}$$

Actual FLOPS done by per processor is the size of matrix multiplication that is done by the processor which is the product o f number of rows in *A_loc*, number of columns in B_temp, number of columns in *A_loc* and the number of the times the multiplication is performed. The efficiency is given as:

$$Efficiency = \frac{Actual\,performance}{Theoretical\,performance} \tag{3}$$

5 Performance Analysis

The following versions of the code are used to do the performance analysis:

- MPI with Naive Multiplication
- MPI with BLAS libraries for Multiplication
- Hybrid MPI-OpenMP with Naive Multiplication
- Hybrid MPI-OpenMP with BLAS libraries for Multiplication

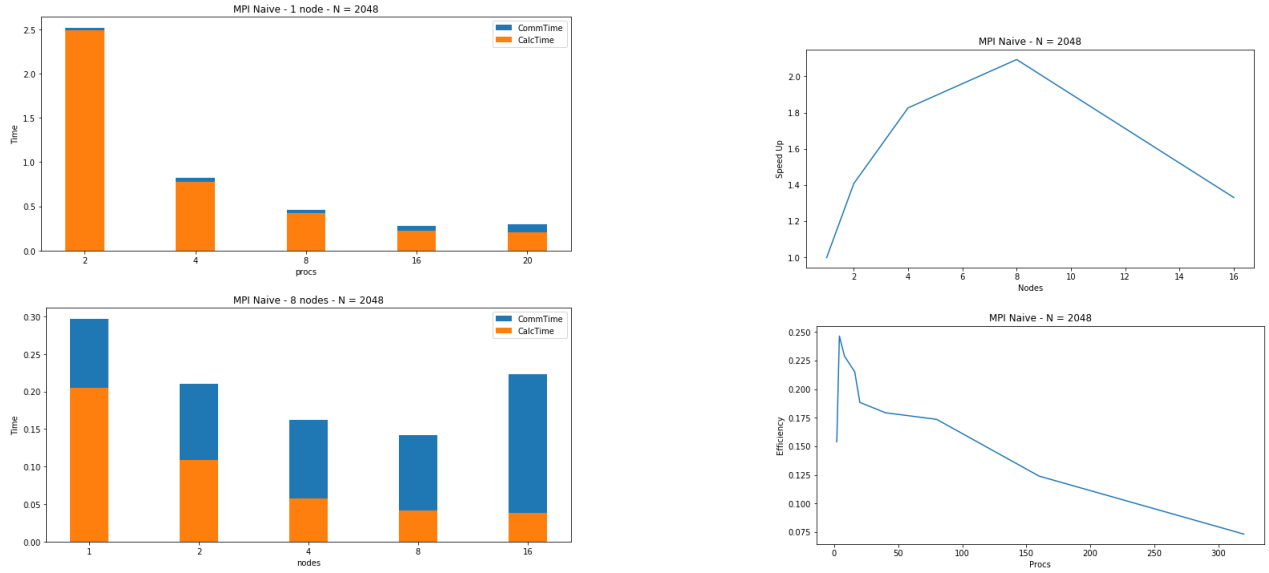

Figure 1: MPI Naive Kernel for a matrix size 2048. The top left panel shows the computation & communication times in 1 node, the bottom left shows the same across the nodes. The top right shows the Speed Up across the nodes and the bottom right shows the efficiency across the nodes

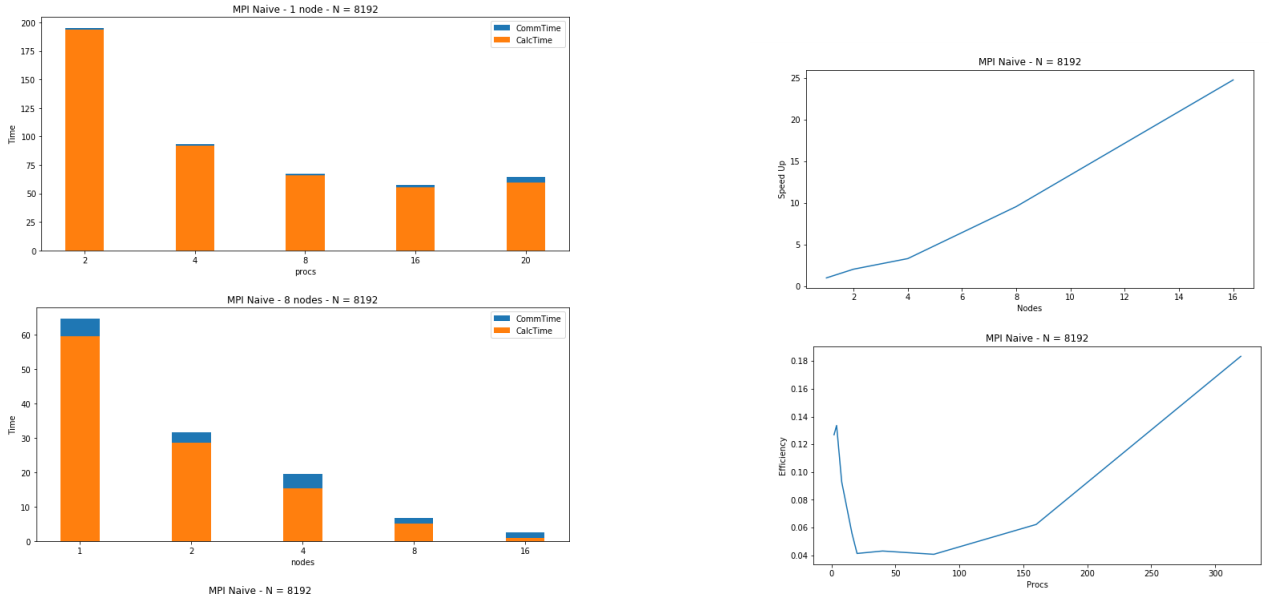

Figure 2: MPI Naive Kernel for a matrix size 8192. The top left panel shows the computation & communication times in 1 node, the bottom left shows the same across the nodes. The top right shows the Speed Up across the nodes and the bottom right shows the efficiency across the nodes

We try to understand the communication & computation time for each case and analyze the scaling behaviour and the efficiency. We all also discuss how the size of the matrix affects when the kernels for multiplication become efficient.

5.1 Multiplication with MPI

For MPI code, the matrix size of 2048 & 8192 are analyzed. Initially a dry run of different matrix sizes is done, from which we choose the two aforementioned sizes.

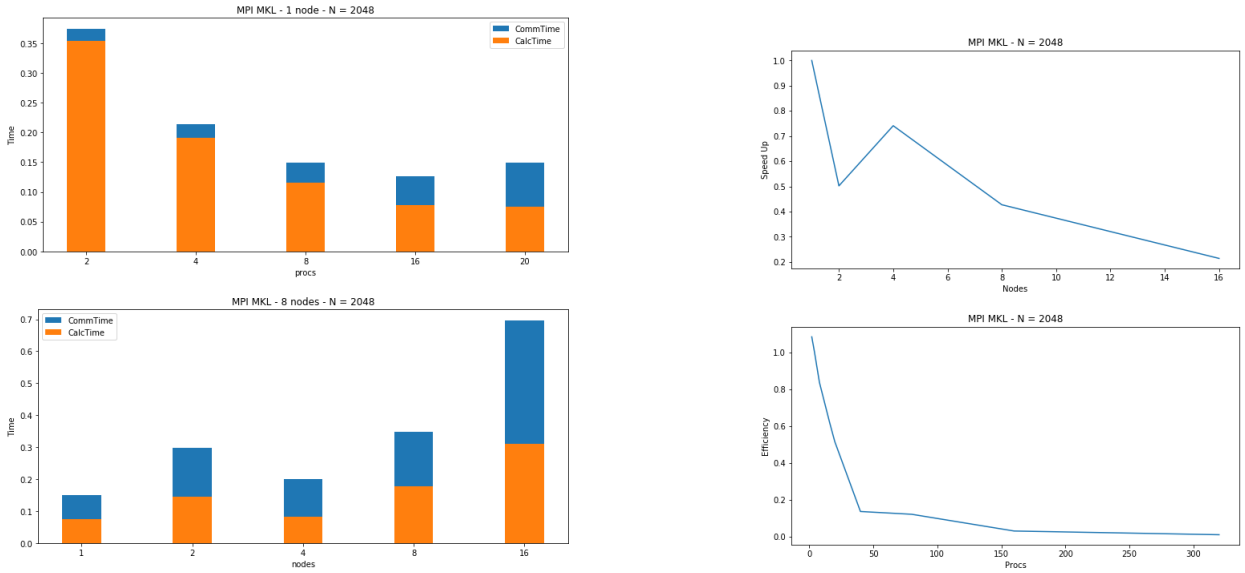

Figure 3: MPI MKL Kernel for a matrix size 2048. The top left panel shows the computation & communication times in 1 node, the bottom left shows the same across the nodes. The top right shows the Speed Up across the nodes and the bottom right shows the efficiency across the nodes

5.1.1 Naive Kernel

From the figure 1, we can see that inside a node, the communication time is negligible compared to the computation time, while the converse occurs as we increase the number of the nodes. The computation time becomes very less when compared to the communication time. This is shown in the speed up plot, where we see the speed up falls beyond 8 nodes. The efficiency of the code is 25% of the peak performance initially and falls to 7.5% with 320 processors. The efficiency drop can be Figure 2 shows the results for the matrix size of 8192. The communication time is negligible compared to the calculation times both inside a node and across the nodes. The code scales well, almost linearly until 16 nodes. The efficiency initially drops and then increases, though as by magnitude remains very small. This is mainly attributed to the naive algorithm used for the matrix multiplication which is totally not efficient.

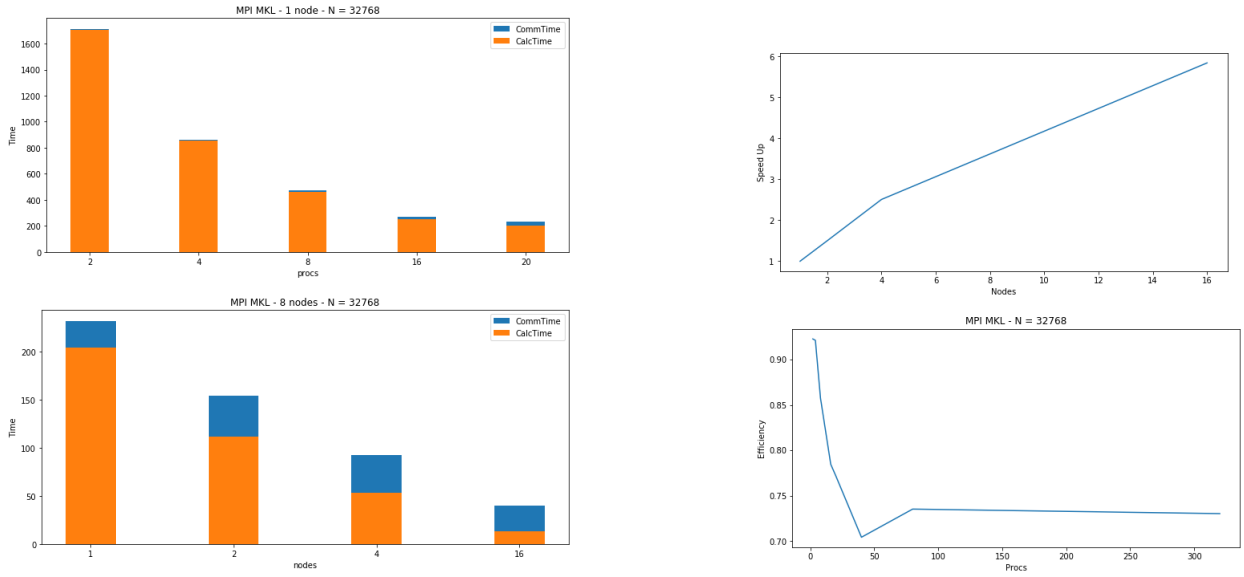

Figure 4: MPI MKL Kernel for a matrix size 32768. The top left panel shows the computation & communication times in 1 node, the bottom left shows the same across the nodes. The top right shows the Speed Up across the nodes and the bottom right shows the efficiency across the nodes

5.1.2 MKL Kernel

The MKL Kernel performs efficient computation due to which the total time reduces drastically from 2.5s to 0.35s for the size 2048 with 2 processors. From figure 3, the communication time starts to become significant due to which we do not observe any scaling across the nodes. There is some scaling within a node, as we go beyond the node, the communication time drastically increases and becomes much significant than the computation time. The efficiency which is at 100% for 1 node drastically drops beyond a node and falls almost to 0.

Due to an efficient kernel for multiplication, a linear scaling is not observed for the size 8192. Hence we move towards higher sizes. For the size 32768, a linear scaling is achieved. The communication time is negligible within a node and across the nodes, the communication time stays almost constant. the efficiency drops from 90% to 70% as we increase the number of processors.

5.2 Hybrid MPI-OpenMP

The hybrid code is run with a configuration of 2 MPI processes per socket spwaning 5 threads each. Both the Naive & MKL versions are run with the same config.

5.2.1 Naive Kernel

From figure 5, we can observe that the communication time becomes almost equal to computation time for 160 processors i.e. 8 nodes. Hence we see no speed up beyond 8 nodes. When compared to the MPI version, the difference in the magnitude of total time is negligible, but the communication time starts to explode at 16 nodes, which doesn't happen

for the hybrid version. This is mainly due to less number of MPI processes communicate with each other when compared to the pure MPI version. Exploiting the shared memory within a node provides better scaling. The figure 6 shows the results for the matrix size of

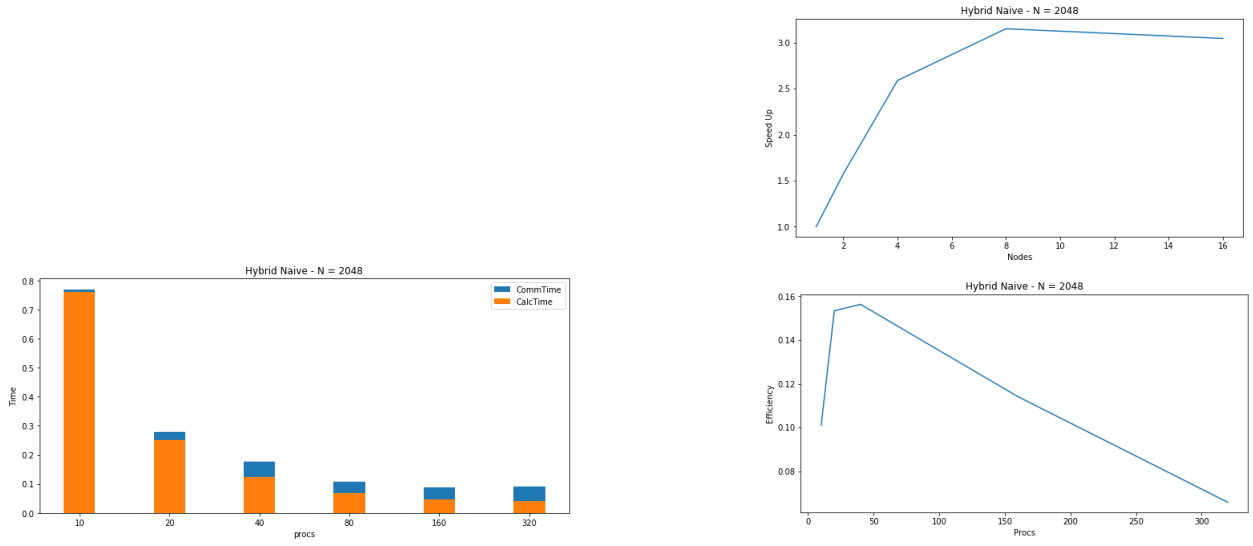

Figure 5: Hybrid MPI-OpenMP with Naive Kernel for a matrix size 2048. The bottom left shows the communication & computation times across the processors. The top right shows the Speed Up across the processors and the bottom right shows the efficiency across the processors

8192. Due to the huge computation time and the communication time being negligible, we achieve an almost linear scaling. The efficiency increases as we increase the number of processors, though its magnitude remains small due to the naive kernel used for multiplication. We achieve a similar scaling in the MPI Naive, with almost similar times for computation & communication, with the hybrid version taking a little less time. We can infer that as the matrix size becomes bigger when the communication time becomes very insignificant, an almost linear scaling can be achieved by both MPI and Hybrid MPI-OpenMP, with the Hybrid version costing a little less in total time taken.

5.2.2 MKL Kernel

With an efficient kernel, the communication time becomes significant for the size 2048 for the hybrid version.As we can see from figure 7, no scaling is achieved as we increase the processors, as similar to the MPI version but the total time goes down by half of that of the MPI version. While for the size 32768 from fig 8, the communication time is negligible, hence it produces a linear speed up. The total time taken is similar to the MPI version. The efficiency drops from 72% to 64%.

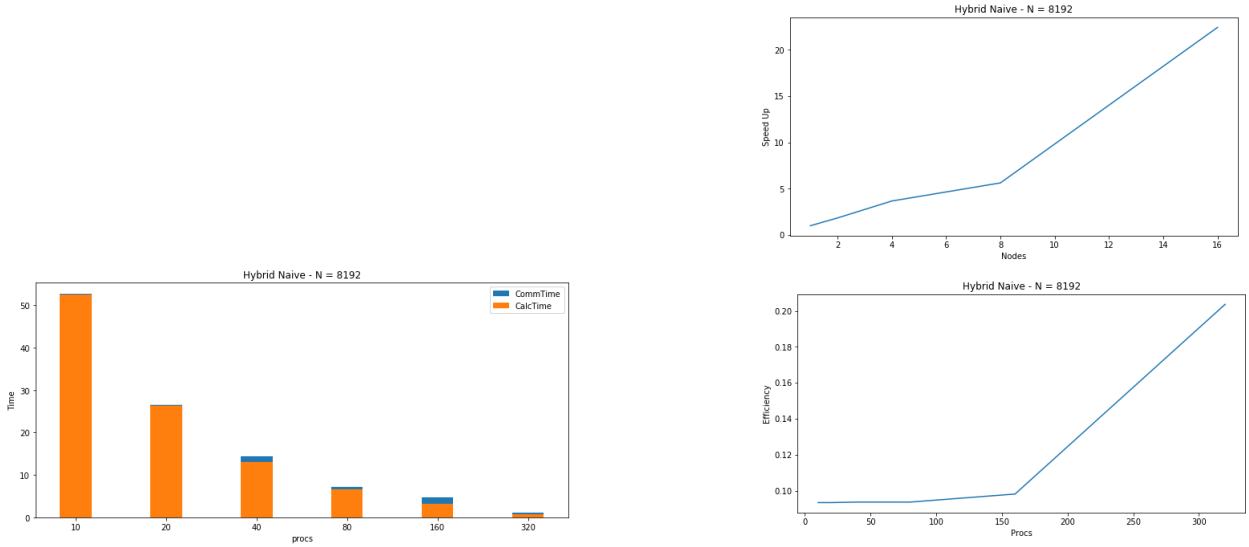

Figure 6: Hybrid MPI-OpenMP with Naive Kernel for a matrix size 8192. The bottom left shows the communication & computation times across the processors. The top right shows the Speed Up across the processors and the bottom right shows the efficiency across the processors

6 Conclusion

From the exercise, we are able to understand that the making a hybrid version of MPI-OpenMP brings a scaling similar to MPI, with the hybrid version costing a bit lesser in terms of total time taken. The Hybrid code reduces some communication time by using less MPI processes and takes advantage of the shared memory concept. Using a very efficient kernel for computation shows that total time reduces drastically and when the matrix size becomes bigger the reduction can be over an order of magnitude. This causes the communication time to be significant and affects the scaling behaviour. Due to this we need bigger sizes to achieve a linear scaling. The trade off between the computation and communication times impacts the scaling with both MPI & Hybrid MPI-OpenMP.

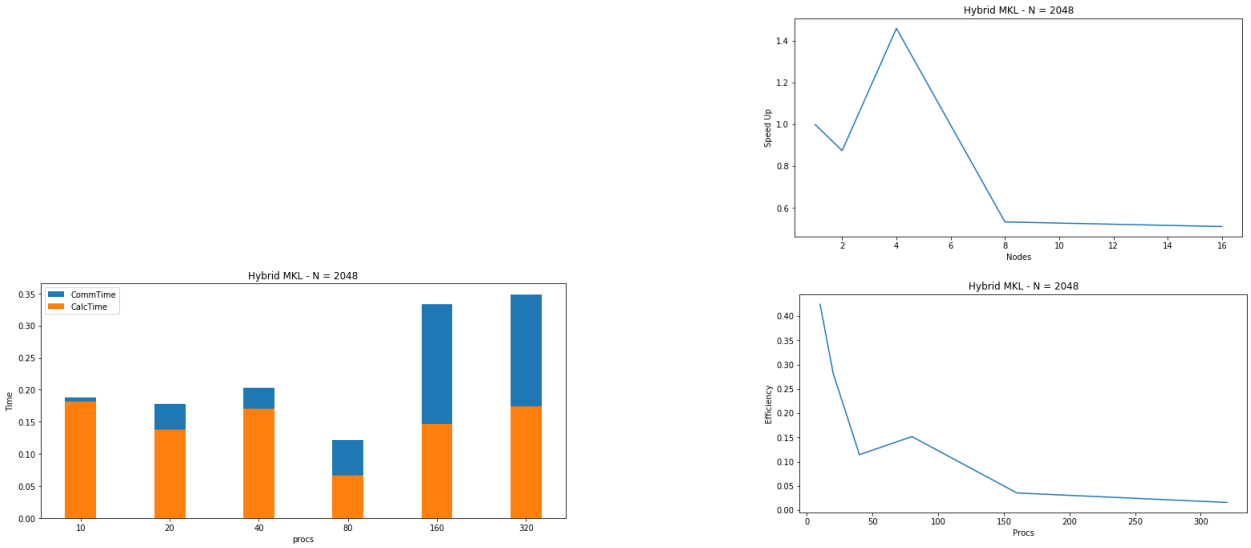

Figure 7: Hybrid MPI-OpenMP with MKL Kernel for a matrix size 2048. The bottom left shows the communication & computation times across the processors. The top right shows the Speed Up across the processors and the bottom right shows the efficiency across the processors

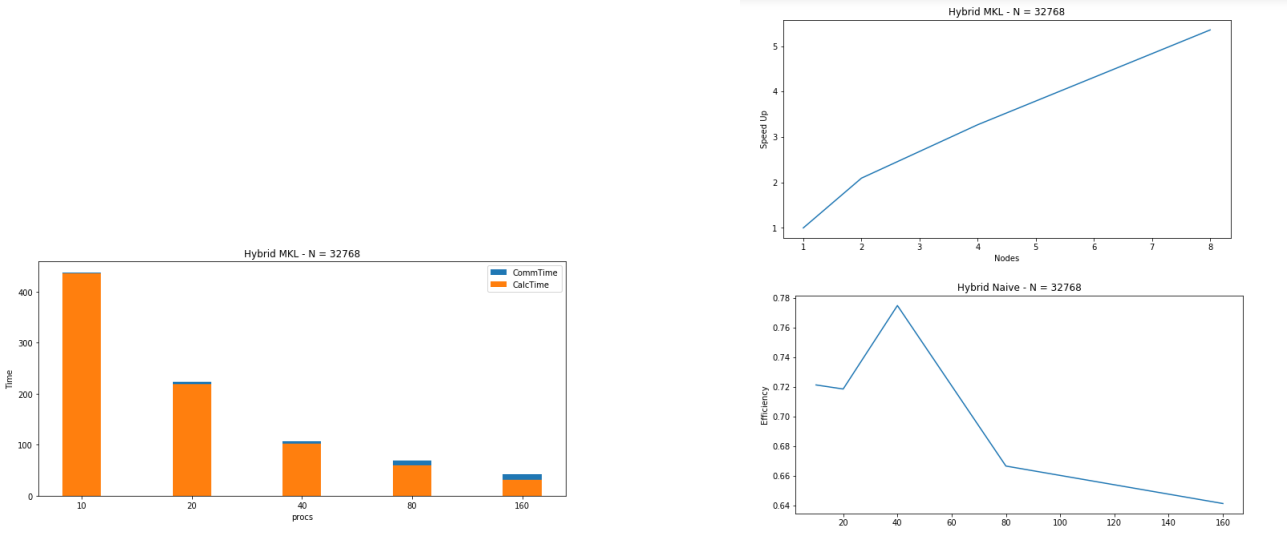

Figure 8: Hybrid MPI-OpenMP with MKL Kernel for a matrix size 32768. The bottom left shows the communication & computation times across the processors. The top right shows the Speed Up across the processors and the bottom right shows the efficiency across the processors