# Programming with MPI (Understanding Communication) and OpenMP – Gaussian Elimination

## Aim:

To parallelize the given sequential C program for Gaussian Elimination, using the MPI & OpenMP libraries, optimise the code, and cross compare their performances with the best pthread implementation from Assignment-1.

## Software Used:

GCC C compiler (version 11.2.1), pthread library, MPI, OpenMP.

## MPI Implementation:

### Algorithm:

In this method, thread 0 is chosen as the main thread which assigns time consuming tasks to other threads & the other threads would just share the load of the main O(n^3) time complexity for loop.

In every iteration of i, the rows that need to be updated are spitted equally among the threads, and the supporting threads would perform the necessary operations and pass the updated rows back to the main thread.

Similar to the pthreads implementation, even in this method the getPivot() function is not parallelized and is run sequentially in all iterations on the main thread, since it has very little complexity to get any significant improvement over parallelizing on top of the parallelization overhead.

The matrix rows sent to the supporting threads, are sent after splitting the matrix into blocks where each block contains a set of consecutive rows. Then overall in a single iteration of i, exactly 1 block will be assigned to the each threads.

These consecutive rows are present in consecutive memory locations, which would allow us to send the entire block in a single MPI Sent or MPI_Broadcast function. This would help a lot in reducing the number of messages passed through the MPI interface in each iteration (max = num_threads) & improve parallel programming.

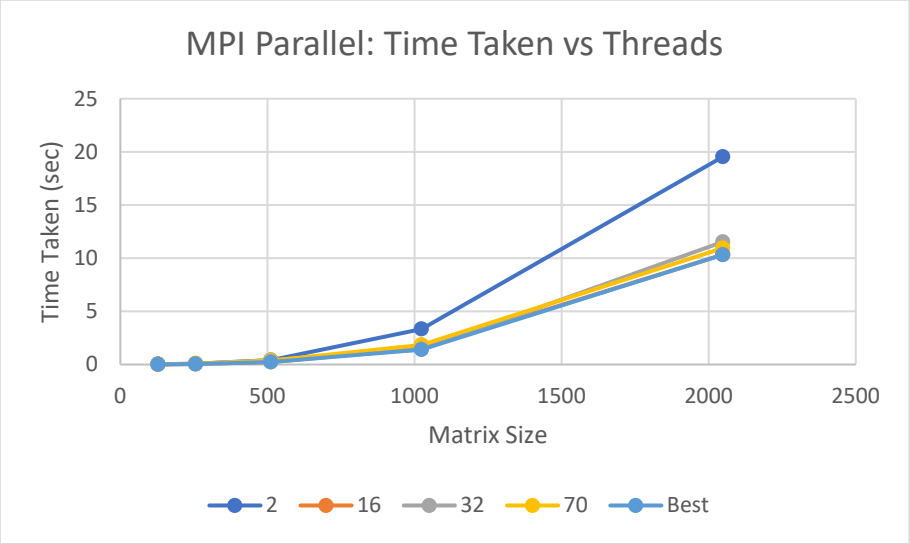
A new function called matmul() was created which the supporting threads would infinitely loop over, to support the main thread whenever required.

### Results on node2x18a (Parallel MPI Program):

The node2x18a server which contains “Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz”, the 72 core CPU was used to test the parallel performances of the Gaussian Elimination MPI versions. They are listed below (all time units are measured in seconds).

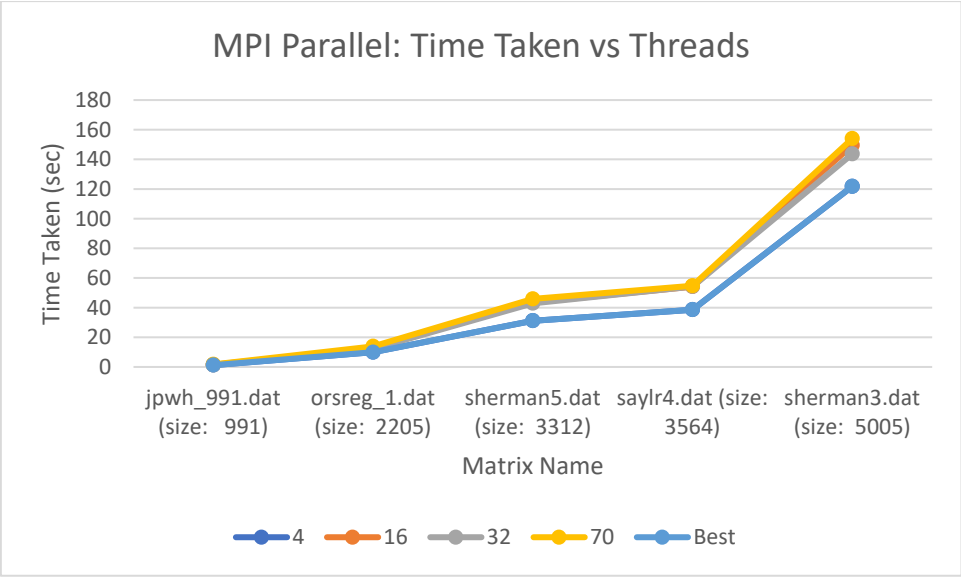
**a) Internal Input:**

| Mat Size / Num Threads | 2 | 16 | 32 | 70 |
|---|---|---|---|---|
| **128** | 0.021107 | 0.015107 | 0.014647 | 0.020563 |
| **256** | 0.087043 | 0.054309 | 0.048062 | 0.081726 |
| **512** | 0.408798 | 0.231638 | 0.405716 | 0.372433 |
| **1024** | 3.33423 | 1.513559 | 1.40785 | 1.842334 |
| **2048** | 19.53686 | 10.32634 | 11.5346 | 10.94641 |




**b) External Input:**

| File Name / Num Threads | 4 | 16 | 32 | 70 |
|---|---|---|---|---|
| **jpwh_991.dat (size: 991)** | 1.221907 | 1.501379 | 1.252859 | 1.727963 |
| **orsreg_1.dat (size: 2205)** | 9.948886 | 12.62937 | 12.27077 | 13.86897 |
| **sherman5.dat (size: 3312)** | 31.15757 | 43.50957 | 42.93624 | 45.89079 |
| **saylr4.dat (size: 3564)** | 38.58076 | 54.14468 | 54.27009 | 54.73255 |
| **sherman3.dat (size: 5005)** | 121.7746 | 149.5279 | 143.6266 | 154.0029 |

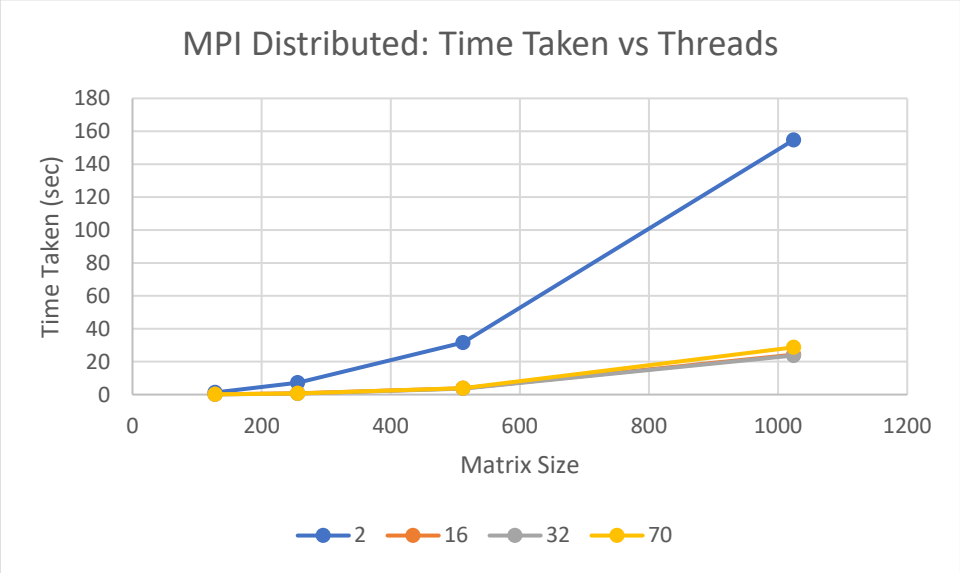


**Results on node01 to node06 (Distributed MPI Program):**

The node servers consist of 12 CPU cores each, which makes up to a total of 12*6=72 CPU cores. The same MPI program was ran on a distributed manner with network communication across these systems. The results were slow, due to the overhead of network communication during every iteration. But it has a good potential to have an improved performance when the matrix size is too high. Here are the final performances:

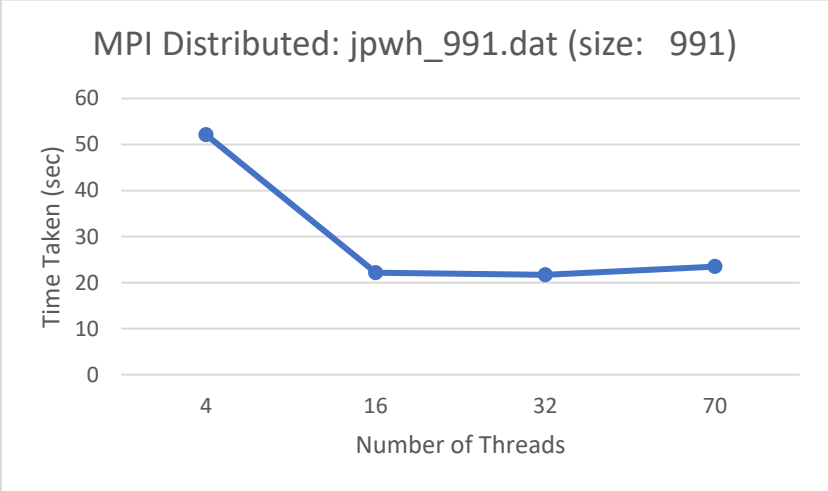
**a) Internal Input:**

| Mat Size / Num Threads | 2 | 16 | 32 | 70 |
|---|---|---|---|---|
| **128** | 1.29592 | 0.167949 | 0.127643 | 0.162209 |
| **256** | 7.218069 | 0.763665 | 0.638781 | 0.720387 |
| **512** | 31.57587 | 3.805018 | 3.65404 | 3.889204 |
| **1024** | 154.7341 | 24.26487 | 23.72897 | 28.7057 |




**b) External Input:**

| File Name / Num Threads | 4 | 16 | 32 | 70 |
|---|---|---|---|---|
| **jpwh_991.dat (size: 991)** | 52.11827 | 22.14183 | 21.70153 | 23.4873 |



## OpenMP Implementation:

### Algorithm:

The OpenMP implementation was indeed very simple and straightforward, yet very powerful. It seemed to be the best available tool used up till now to parallelize the Gaussian Elimination program.

The only addition done to the sequential program is the following line, right before the O(n^3) time complexity for loop:

```
#pragma omp parallel for private(j, k, pivotVal)
```

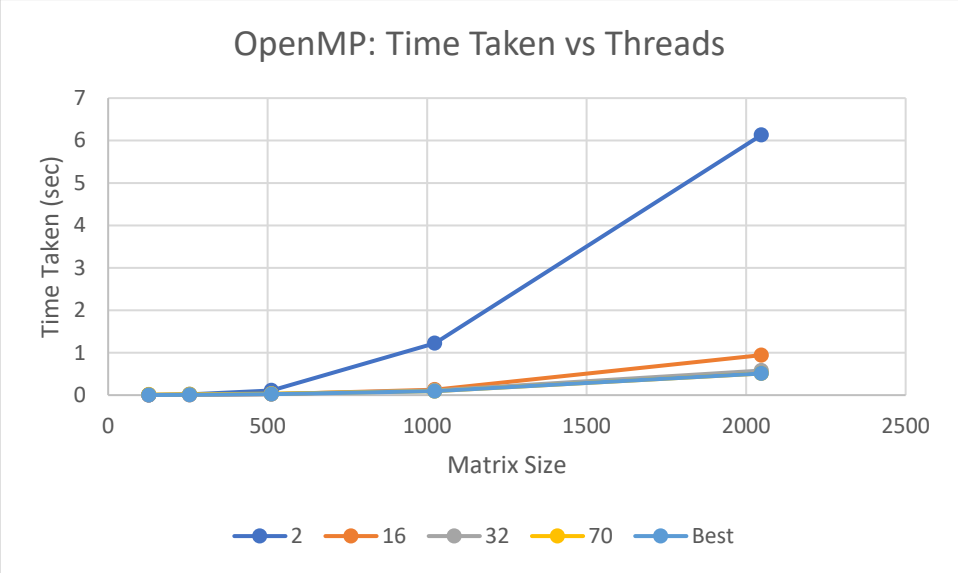
This line optimised and parallelised the code in the for loop by itself, right during compilation. Variables j, k & pivotVal were defined as private since it is necessary to create local copies of them for each thread, to ensure the logic remains consistent.

And this 1 line addition was found to be so powerful that, the optimisation done automatically by it, was much faster than all of the previous pthread and MPI implementation, which shows the efficiency of the optimisation ability of OpenMP during compilation of the code.

### Results on node2x18a (Parallel OpenMP Program):

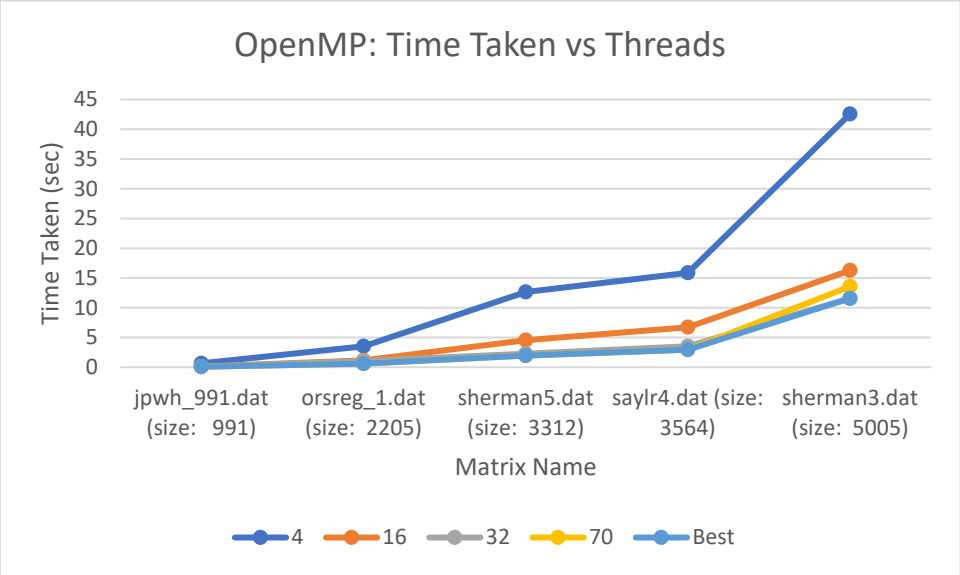
a) **Internal Input:**

| Mat Size / Num Threads | 2 | 16 | 32 | 70 |
|---|---|---|---|---|
| **128** | 0.003023 | 0.002476 | 0.005458 | 0.011107 |
| **256** | 0.017611 | 0.005742 | 0.010848 | 0.018525 |
| **512** | 0.110627 | 0.023711 | 0.025479 | 0.032788 |
| **1024** | 1.222651 | 0.129809 | 0.110165 | 0.093477 |
| **2048** | 6.131502 | 0.942796 | 0.578848 | 0.509428 |



**b) External Input:**

| File Name / Num Threads | 4 | 16 | 32 | 70 |
|---|---|---|---|---|
| **jpwh_991.dat (size: 991)** | 0.661793 | 0.118334 | 0.084672 | 0.100493 |
| **orsreg_1.dat (size: 2205)** | 3.530288 | 1.164427 | 1.035961 | 0.600996 |
| **sherman5.dat (size: 3312)** | 12.65117 | 4.543816 | 2.26962 | 1.957547 |
| **saylr4.dat (size: 3564)** | 15.87535 | 6.72201 | 3.521165 | 2.956282 |
| **sherman3.dat (size: 5005)** | 42.57163 | 16.27973 | 11.58422 | 13.58726 |



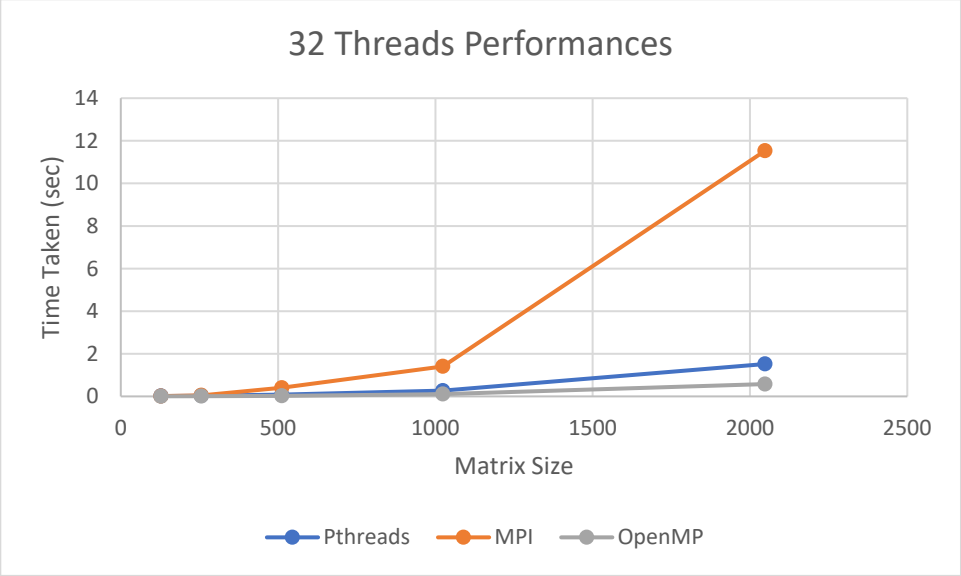

## Cross Comparison:

Here is the comparison of the performance of Gauss Elimination with pthreads, MPI & OpenMP on the same server, i.e. the node2x18a. In order to visualize it in a 2D space, the comparison with be done 1st with 32 threads, and then with the best performances across threads.

**a) Internal Input:**

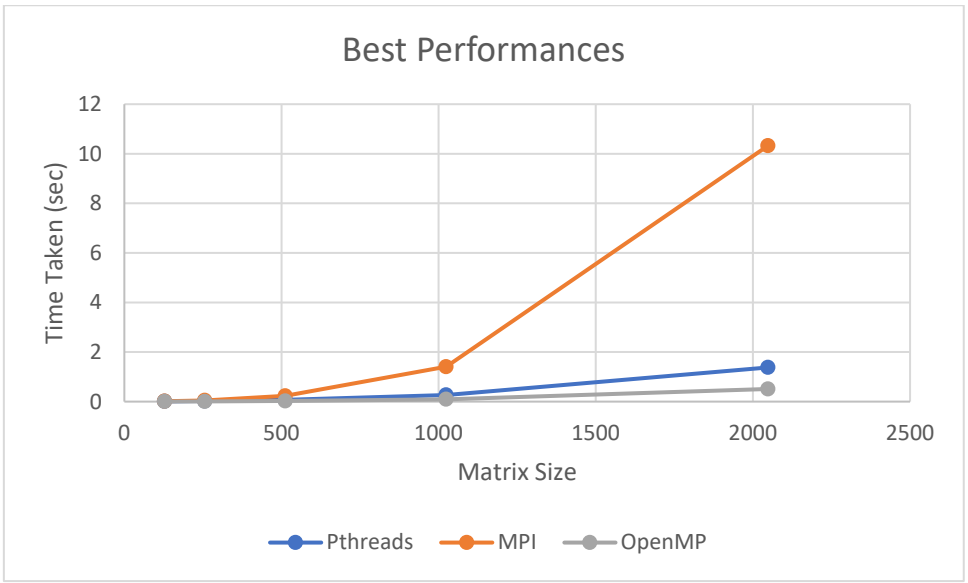
**(i) 32 Thread Cross Comparison:**

| Mat Size / Method | Pthreads | MPI | OpenMP |
|---|---|---|---|
| **128** | 0.016532 | 0.014647 | 0.005458 |
| **256** | 0.034221 | 0.048062 | 0.010848 |
| **512** | 0.080886 | 0.405716 | 0.025479 |
| **1024** | 0.268919 | 1.40785 | 0.110165 |
| **2048** | 1.519789 | 11.5346 | 0.578848 |




**(ii) Best Performances Cross Comparison:**

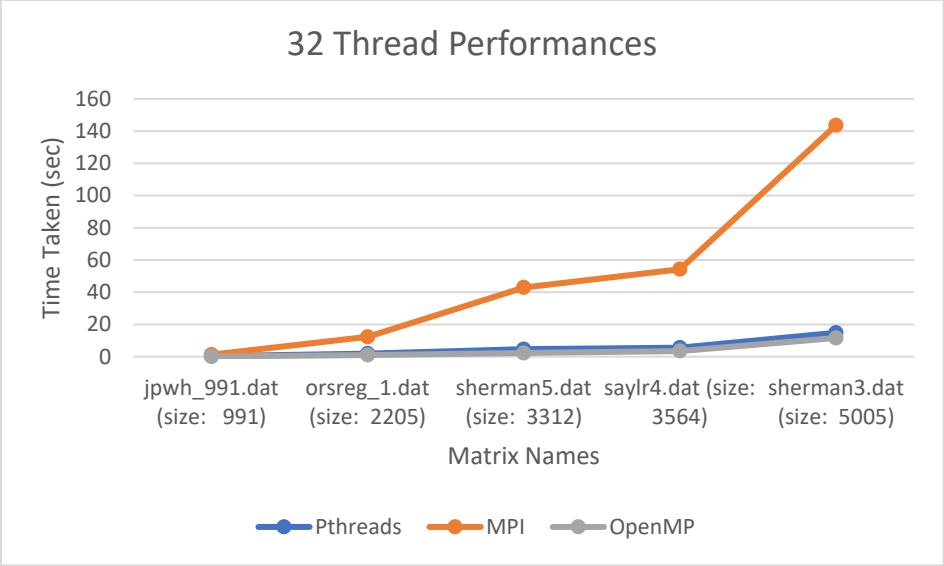
| Mat Size / Method | Pthreads | MPI | OpenMP |
|---|---|---|---|
| **128** | 0.007941 | 0.014647 | 0.002476 |
| **256** | 0.02406 | 0.048062 | 0.005742 |
| **512** | 0.07144 | 0.231638 | 0.023711 |
| **1024** | 0.266466 | 1.40785 | 0.093477 |
| **2048** | 1.377353 | 10.32634 | 0.509428 |



**b) External Input:**

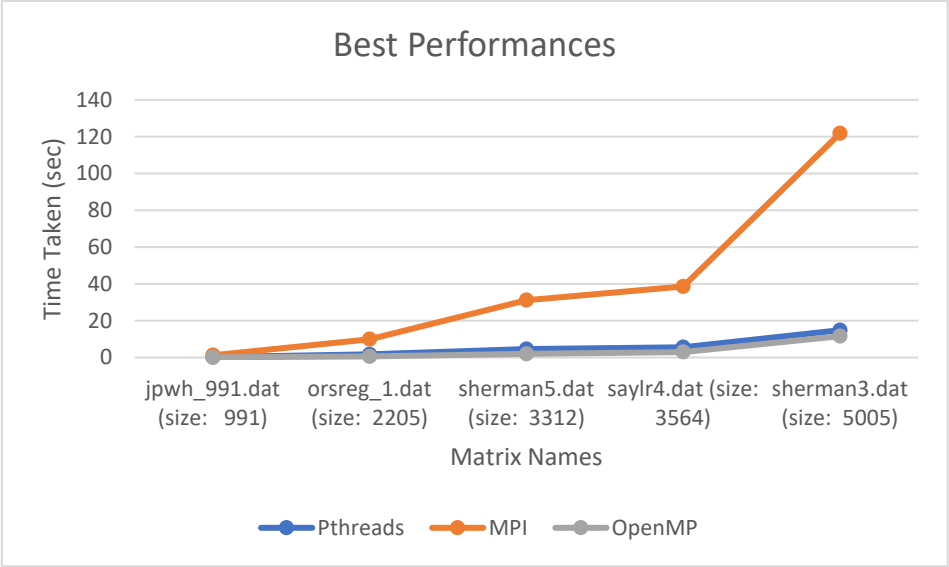
**(i) 32 Thread Cross Comparison:**

| File Name / Num Threads | Pthreads | MPI | OpenMP |
|---|---|---|---|
| **jpwh_991.dat (size: 991)** | 0.293424 | 1.252859 | 0.084672 |
| **orsreg_1.dat (size: 2205)** | 1.929066 | 12.27077 | 1.035961 |
| **sherman5.dat (size: 3312)** | 4.640102 | 42.93624 | 2.26962 |
| **saylr4.dat (size: 3564)** | 5.584708 | 54.27009 | 3.521165 |
| **sherman3.dat (size: 5005)** | 14.846271 | 143.6266 | 11.584218 |




**(ii) Best Performances Cross Comparison:**

| File Name / Num Threads | Pthreads | MPI | OpenMP |
|---|---|---|---|
| **jpwh_991.dat (size: 991)** | 0.233257 | 1.221907 | 0.084672 |
| **orsreg_1.dat (size: 2205)** | 1.696003 | 9.948886 | 0.600996 |
| **sherman5.dat (size: 3312)** | 4.581856 | 31.15757 | 1.957547 |
| **saylr4.dat (size: 3564)** | 5.584708 | 38.58076 | 2.956282 |
| **sherman3.dat (size: 5005)** | 14.846271 | 121.7746 | 11.584218 |



## Inferences:

The following inferences have been observed after all the tests:

(i) Out of all the 3 methods, OpenMP gives the best parallel performance with Gaussian Elimination, and it the easiest to implement as well.

(ii) MPI seemed to be the slowest method for implementing Gaussian Elimination, since message passing is needed in every iteration of the main loop, which adds up a major portion of the complexity in comparison to the remaining operations performed in the loop.

(iii) But however, MPI has a bigger potential for application that do not require frequent message passing, and especially they can be run in more than 1 computers at once connected via a network.

(iv) Pthreads offers more flexibility to configure, the parallel programming algorithm in the method that we prefer, but however in case of Gaussian Elimination, the most effective pthread implementation was implementable in OpenMP just with 1 line of code, which was impressive.

## Result:

Hence the C program for Gaussian Elimination is successfully implemented with MPI & OpenMP libraries. In addition, the MPI code have been run in a distributed manner from node01 to node06 as well.

And the final results are successfully cross compared with Pthreads, MPI, OpenMP implementations and the final inferences have been made.