Serial and Parallel 9-Point Stencil

Nathan Marshall, Kyler Febbroriello
Coastal Carolina University
Conway, SC, USA
{ntmarshal, kafebbror}@coastal.edu

***Abstract*—The Serial 9-point stencil algorithm was used to evaluate the way heat transfers on a 2D plane, but the serial version of the program could not effectively run large size iterations. From this point we decided parallelizing should greatly reduce the elapsed time of the program. First we utilized OpenMP which would spawn threads and calculate the number of rows each thread would calculate. After receiving the speedup and efficiency data, we believed rewriting the parallelized code using MPI to create processes instead of threads would allow us to expand the program across multiple nodes. Multiple nodes would allow us to increase the computing power for the programs and thus reduce the elapsed time even further.**

***Index Terms*—9-Point Stencil, Parallel, Serial, Speedup, Efficiency, OpenMP, and MPI**

I. Introduction

Experimentation started with the serialization of a matrix-based 9-point stencil simulation. The algorithm created is provided the number of iterations and the initial matrix size of MxM through the command line. When the program acquires the command line parameters it then uses the 9-point stencil formula shown in Algorithm 1 to simulate a dynamic 2D heat transfer process. The algorithm runs this function until max iterations are reached. When the max iteration is reached the algorithm produces a raw .dat file to record the final stencil. As the matrices grew larger the serial implementation took significantly longer to run with a big O of O(x^2). Our solution to this was to parallelize the code using OpenMP to improve the overall run time. After completing this solution we decided the next step forward was to convert our serial code to an MPI implementation that would allow for us to run multiple processes and in theory would allow us to expand the matrix sizes further because we could utilizing multiple nodes with more computing power.

Algorithm 1 StencilFunction() to calculate xNew

$xNew[i][j]$ = x[j-1][i-1] + x[j-1][i] + x[j-1][i+1] + x[j][i+1] + x[j+1][i+1] + x[j+1][i] + x[j+1][i-1] + x[j][i-1] + x[j][i]/9.0;

978-1-7281-6861-6/20/$31.00 ©2020 IEEE

II. Background

Stenciling is used in many time sensitive, real world applications as they can produce accurate estimates of an area utilizing information from an earlier timestamp. The numbers in the matrix can represent temperatures or elevations in these applications. An example of stenciling can be observed in the images in Figure 1, which shows how each time step compounds and affects the future iterations of itself.

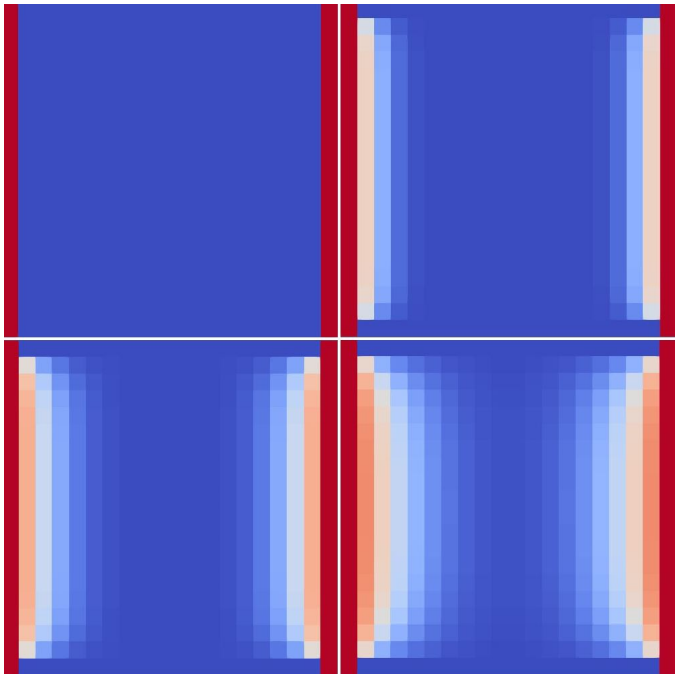

Fig. 1. The figure shows a 20 iteration 9-point stencil meant to visualize heat transfer on a 2D plane. Each sub-image is an iteration taken from the stencil to show the different time steps and the red bars on the side of the sub-image are meant to symbolize a constant heating element. In the upper left section, we can see the first time step which shows no heat transfer, but if we observe the upper right section we can observe in three iterations a change in "temperature" which was influenced by the starting stencil. If we observe the lower left we can see at iteration 10 that the "temperature" is starting to dissipate, as the stencil operations continue. The last sub-image in the lower right shows the final state to which was influenced by all previous time steps.

Stenciling takes a center point in the new time iteration to calculate. It then utilizes information from the previous time stamp and averages the chosen point and the 8 surrounding points to obtain the new time stamps data. The serial implementation of this is shown in Algorithm 2. As each

time stamp loops through the rows and columns in order to calculate the value of xNew using the StencilFunction located in Algorithm 1. At the end of each time iteration the algorithm swaps the pointers to each array for the next time iteration.

Algorithm 2 9-Point Serial Stencil

```
Read array x from input file
Copy x contents to array xNew
for n ← numIterations do
  for i ← 1 to rows − 1 do
    for j ← 1 to cols − 1 do
      StencilFunction()
    end for
  end for
  tmp ← x
  x ← xNew
  xNew ← tmp
end for
```

To improve the serialized version, it was decided that parallelization was necessary to reduce run times. The first parallelized implementation of the stencil code was designed to use OpenMP. OpenMP allowed for multiple threads to be spawned, which allowed the arithmetic to be split into multiple pieces. Its downside is more overhead, although this is deemed necessary to run larger size iteration counts. After successfully implementing OpenMP our team was determined to increase speed-ups and reduce overhead where possible, thus MPI was the next step to implement. MPI works differently then OpenMP as it works by spawning multiple processes of the code and passing message parse between each new process. This allows for the program to be run across multiple machines/nodes to allow for exponentially more speed-up on larger matrices and iteration counts.

III. Parallel Background

Parallelism in a computing sense is the process of taking a given problem and splitting said problem into smaller pieces to be worked on in tandem with the goal being to shorten the time it takes to solve the problem. The way computers achieve this can be seen in Figure 2, but it does such by breaking up the problem into independent pieces to give to different cores, processes, or threads so they can complete their own allocated subsections. The amount of effort put into a program by the developer to parallelize a section of code is directly related to the potential amount of speedup. In our first solution we utilized OpenMP which allows for exponential amounts of speed-ups, but it does taper off after a certain thread count and problem size. So to remedy this MPI was implemented. MPI is a more advanced functionality to implement then OpenMP, but allows for a higher degree of speed-up then OpenMP.

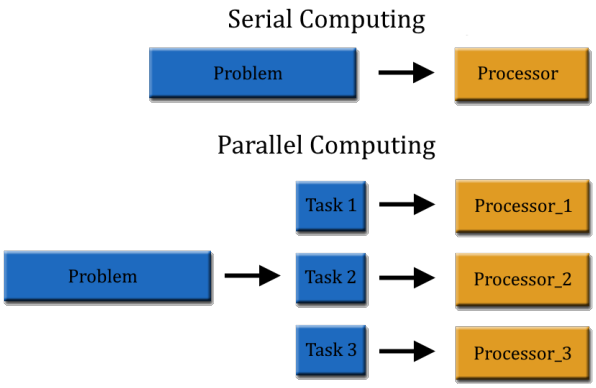

Fig. 2. The figure shows how a problem is split up by a computer into multiple subsections to be worked on. It can also be observed that the computer creates multiple "processors" or normally otherwise called threads, processes, or core.

IV. OpenMP Parallelization Implementation

OpenMP was the first attempt at parallelizing the code. Since the stencil utilizes the previous time stamp to make the calculation, we could break up the loop structures to have the different number of threads complete the work independently and store the answer before the next time iteration. Using OpenMP within Algorithm 3 allowed us to implement this idea and decrease the elapsed time of the program.

Algorithm 3 9-Point Parallel Stencil

```
Read array x from input file
Copy x contents to array xNew
for n ← numIterations do
  #pragma omp parallel for
  for i ← 1 to rows − 1 do
    for j ← 1 to cols − 1 do
      StencilFunction()
    end for
  end for
  tmp ← x
  x ← xNew
  xNew ← tmp
end for
```

V. MPI Parallelization Implementation

After OpenMP testing reached optimal levels experiments were done to attempt a higher speed-up and time execution rate using MPI. MPI is a message passing interface that is a professional standard in the field of parallel programming. It works by making multiple processes instead of traditional threads and then uses these processes to each independently run the compiled programs. As such MPI does not use shared memory like OpenMP and opts to instead use message parsing for its communications. MPI is more advanced in practice, but can often equate to better results. The reason this is so is that programmers must avoid deadlocks and take special care to send to each correct process, but MPI allows operators

to avoid hardware slowdowns/limitations and allows for more computing power aka more nodes.

Within experiments, the MPI code was changed to accommodate the difference in communication. The MPI modified code operated off the same calculations but did so a bit differently. The differences are visualized in Figure 3.

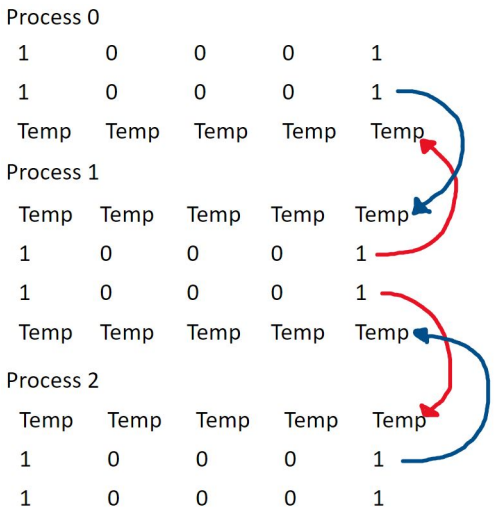

Fig. 3. The figure shows the implementation of halo rows and how each process sends its data to processes above and below. This is necessary so that each process can calculate the appropriate heat value in the matrix. As each process stores the received rows within the extra temporary rows and sends its top-most and bottom-most rows to the process that is ranked above and below as visualized. The only exception to this is the top-most and bottom-most ranked processes only have to send and recv one row since they either only have one process above or below them.

Each process spawned false temporary rows in the matrix to hold the necessary extra rows needed for the heat transfer formula. The processes then used the functions "Irecv" and "Isend" provided by the MPI library to send the actual values to replace these new temp rows. This allowed each process to calculate its subblock values accurately.

VI. Experimentation

All testing and activity was done on the Expanse cluster a dedicated eXtreme Science and Engineering Discovery Environment cluster designed by Dell and SDSC delivering 5.16 peak petaflops, and will offer Composable Systems and Cloud Bursting. The facility is shown in Figure 4

Fig. 4. The figure above shows a cluster from the Expanse facility and is what was used in the process of parallelizing the 9-point stencil algorithm and was subsequently used for testing of the OpenMP program.

To increase the productivity and effectiveness of large iterations using the stencil program, OpenMP and MPI were implemented to parallelize the stencil program. When testing the OpenMP implementation our team used one node on the Expanse cluster, which tested from 1 to 32 threads. The MPI testing followed suit and used one active node on Expanse and opted to use 1 to 32 processes for testing. For both OpenMP and MPI testing each tested matrices sized from (1000*1000), (2000*2000), (3000*3000), and (4000*4000). This allowed for a large sample size. All matrix sizes ran 1000 iterations to better compare the differences when matrix size is increased. For each thread we calculated the elapsed time and used it to calculate the Speedup which we defined in Equation 1 and the Efficiency which we defined in Equation 2.

VII. OpenMP Results

A. Elapsed Time

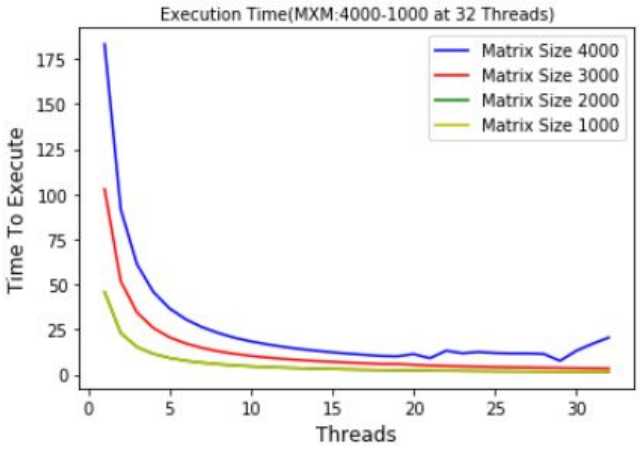

Fig. 5. The figure shows the Elapsed time of the process of matrix sizes between 1000 and 4000. For each matrix size we tested with the number of threads ranging from 1 to 32. In all cases the elapsed time started at a maximum and quickly decreased as more threads were added. This trend continued except in the case of the 4000x4000 matrix where at thread counts above 20 the benefit seemed to stop.

The elapsed time figure shows that as more threads were added the execution time had a dramatic decrease. Most iterations saw a decline in execution time except for the 4000x4000 matrix with 20 or more threads in which the execution time did not go down as expected. This may be due to the number of iterations selected. Overall the results followed the expected trend of decreasing slower as matrix size increased.

B. Speedup

$$Speedup = \frac{SerialTime}{ParallelTime} \tag{1}$$

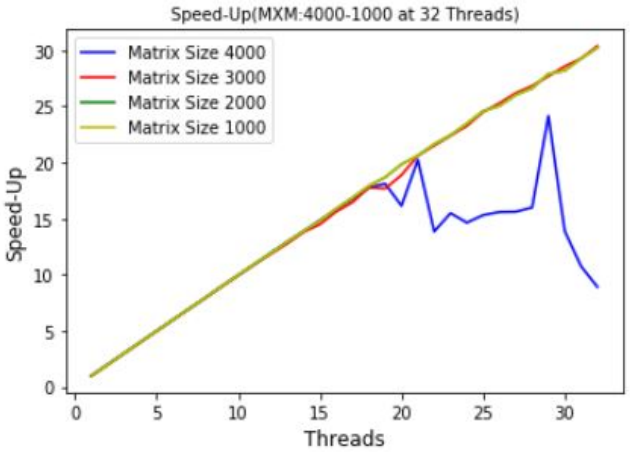

Fig. 6. The figure shows the Speedup calculated using Equation 1. Each line represents a different matrix size ranging from 1000x1000 to 4000x4000. In general the trend of the line is very close to linear as in most cases the number of threads corresponds to the amount of speedup obtained. However, after thread 20 in the 4000x4000 matrix, the speedup significantly dropped which can be attributed to the elapsed time not dropping as expected. This could be due to the size of the matrix or other factions, more experimentation is required to determine the cause.

After vigorous testing, the speed-up was calculated for each thread using Equation 1. This equation assists in evaluating the rough performance gain between the serial program and the newly developed parallelized program. In a perfect scenario the speedup would be equal to the thread count as this would mean running with 100% efficiency. Our program saw this trend in the lower thread counts. In most cases the speedup was equal or very close to the number of threads used. This means that the parallelization was very effective in speeding up the program as the theoretical maximum speedup was obtained in most of the test. The 4000x4000 matrix again had deviated from this norm. This may be due to the number of iterations on the large size matrix causing memory issues thus causing this significant drop. With more testing on different number of iterations and averaging the results together it is believed the the deviation would level out.

C. Efficiency

$$Efficiency = \frac{Speedup}{NumberofThreads} = \frac{\frac{SerialTime}{ParallelTime}}{NumberofThreads} \tag{2}$$

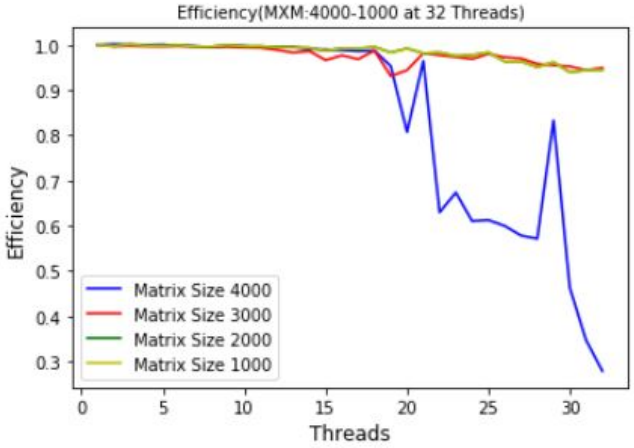

Fig. 7. The figure represents the efficiency of the program when parallized as defined by Equation 2. Each line represents a different sized matrix ranging from 1000x1000 to 4000x4000. In general the efficiency was better than expect as it remained above 90% in most cases. This means that the speedup was nearly optimal for each thread count on the matrix size. The 4000x4000 matrix however saw a dramatic drop in efficiency starting at around the 20 thread mark. This may be due to the large amount of iterations we chose causing a memory problem or an anomaly in our specific test. With more experimentation on the matrix size with a range of iterations, we believe we can determine the problem if there is one.

The efficiency data was as expected or better. In most cases the efficiency was in the 90 percent range. While the 4000x4000 matrix saw a dramatic drop in the higher number of threads, most other instances saw the expected output.

VIII. MPI Results

A. Elapsed Time

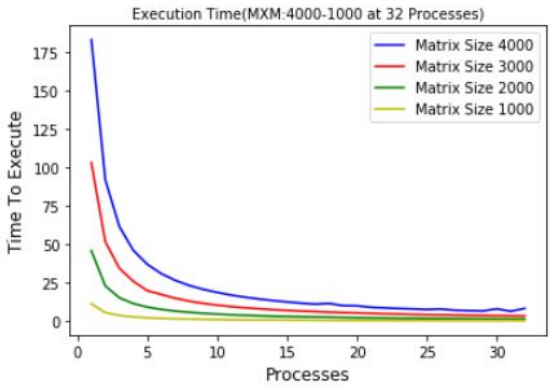

Fig. 8. The figure shows the Elapsed time of the process of matrix sizes between 1000 and 4000. For each matrix size we tested with the number of processes ranging from 1 to 32. In all cases the elapsed time started at a maximum and quickly decreased as more processes were spawned. This trend continued for every additional process added, but appeared to taper off and exponentially slow down when execution time approached zero.

The figure shows the Elapsed time of the process of matrix sizes between 1000 and 4000. For each matrix size we tested with the number of processes ranging from 1 to 32. In all cases the elapsed time started at a maximum and quickly decreased as more processes were spawned. This trend continued for every additional process added, but appeared to taper off and exponentially slow down when execution time approached zero. The data recorded supports the expected drop off when implementing MPI.

B. Speed-Up

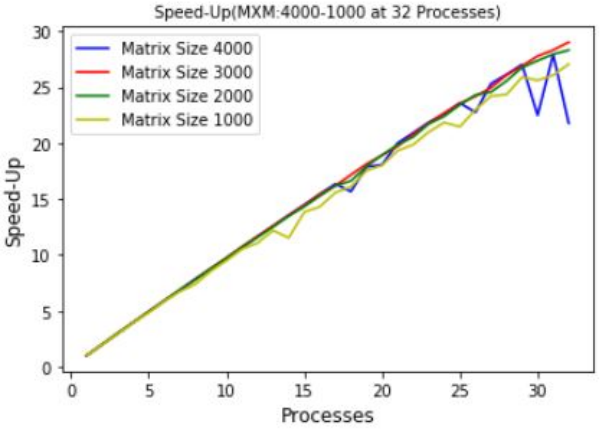

Fig. 9. The figure shows the Speedup calculated using Equation 1. Each line represents a different matrix size ranging from 1000x1000 to 4000x4000. In general the trend of the line is nearly linear with a few outliers, although this could be due to memory issues on Expanse itself.

After vigorous testing, the speed-up was calculated for each thread using Equation 1. This equation assists in evaluating the rough performance gain between the serial program and the newly developed parallelized program. In a perfect scenario the speedup would be equal to the process count as this would mean running with 100% efficiency. Our program saw this trend throughout almost all process counts. In most cases the speedup was equal or very close to the number of processes used. This means that the parallelization was very effective in speeding up the program as the theoretical maximum speedup was obtained in most of the test. The few outliers can be chalked up to memory issues with Expanse.

C. Efficiency

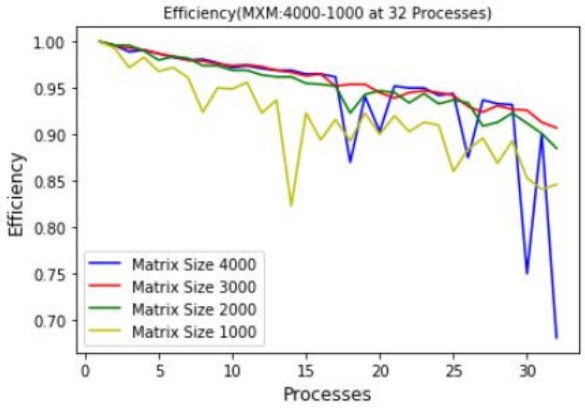

Fig. 10. The figure represents the efficiency of the program when parallelized as defined by Equation 2. Each line represents a different sized matrix ranging from 1000x1000 to 4000x4000. For larger sized matrices they stayed just about at or above 90 percent the exception to this is the latter half of the 1000x1000, which had poor efficiency which could be due to the smaller size which is expected. Another exception is the 4000x4000 matrix tapers off severely at the end which could caused by the smaller time execution values. Although there are a few spots of efficiency being poor the majority of the efficiency is very good and better than expected.

Each line represents a different sized matrix ranging from 1000x1000 to 4000x4000. For larger sized matrices they stayed just about at or above 90 percent the exception to this is the latter half of the 1000x1000, which had poor efficiency which could be due to the smaller size which is expected. Another exception is the 4000x4000 matrix tapers off severely at the end which could caused by the smaller time execution values. Although there are a few spots of efficiency being poor the majority of the efficiency is very good and better than expected.

IX. Comparison

A. Expected Scaling

As can be seen in Figure 11 when time is put forth eventually MPI will outweigh and perform better than an OpenMP program. This is noteworthy to keep in mind, as the results in Figure 12, Figure 13, and Figure 14 give a good insight. This is because when implementing both methods they gave near similar results, although the OpenMP performed better in most cases this was caused by the fact of both programs being run on one node. If extra compatiability were added for the MPI iteration then results should follow suit on the predicted path.

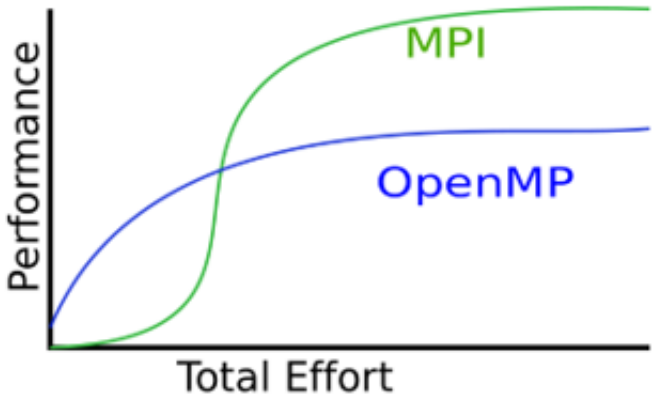

Fig. 11. The figure shows the expected results in the scalability differences between MPI and OpenMP via Manning Publications.

B. Time Execution

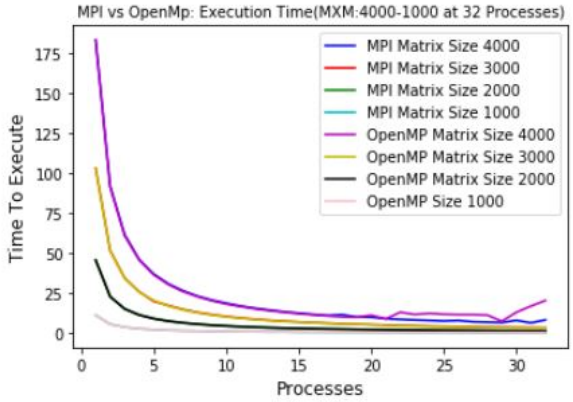

Fig. 12. The figure shows the time to execute of both OpenMP and MPI. We can observe that both methods are very close and almost on top of each other. The exceptions that show are that the OpenMP is better except for higher thread/process counts on the 4000*4000 matrix. This is expected since the MPI code is only running off of 1 node.

The time to execute of both OpenMP and MPI. We can observe that both methods are very close and almost on top of each other. The exceptions that show are that the OpenMP is better except for higher thread/process counts on the 4000*4000 matrix. This is expected since the MPI code is only running off of 1 node.

C. Speed-Up

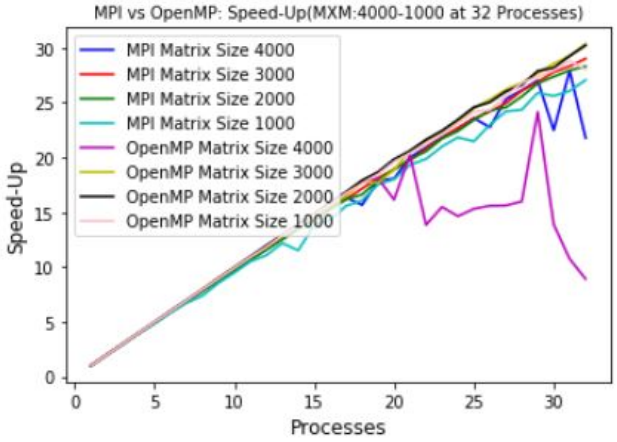

Fig. 13. The figure shows the speed-up of both the OpenMP and MPI results. It can be observed that both methods have almost identical linear lines, but the OpenMP implementation is a bit better with speed-up except with higher thread/process counts on the 4000*4000 matrix. Again this can be expected due to the fact that MPI is only scaling on 1 node on Expanse. The 4000*4000 exception could be do to memory issues on the OpenMP implementation.

The speed-up of both the OpenMP and MPI results. It can be observed that both methods have almost identical linear lines, but the OpenMP implementation is a bit better with speed-up except with higher thread/process counts on the 4000*4000 matrix. Again this can be expected due to the fact that MPI is only scaling on 1 node on Expanse. The 4000*4000 exception could be do to memory issues on the OpenMP implementation.

D. Efficiency

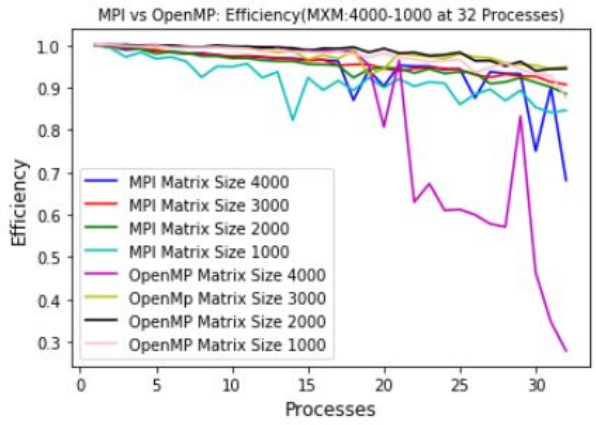

Fig. 14. The figure shows the efficiency of both the OpenMP and MPI results. It can be shown that efficiency is again near linear and the same for both implementations although OpenMP has a little better efficiency in a majority of the data points with the exception of the 1000*1000 matrix which is quite a bit lower. This could be due to memory issues on Expanse. The same can be said for the reason MPI is substanally better than the OpenMP 4000*4000 matrix high thread/process count.

The efficiency of both the OpenMP and MPI results. It can be shown that efficiency is again near linear and the same

for both implementations although OpenMP has a little better efficiency in a majority of the data points with the exception of the 1000*1000 matrix which is quite a bit lower. This could be due to memory issues on Expanse. The same can be said for the reason MPI is substanally better than the OpenMP 4000*4000 matrix high thread/process count.

X. Conclusions

The Serial 9-point stencil algorithm was used to evaluate the way heat transfers on a 2D plane, but the serial version of the program could not effectively run large iterations without taking massive amounts of time. With this in mind parallelization was implemented with OpenMP to increase the performance of the 9-point stencil algorithm. The parallelized code was tested vigorously by using high iteration counts and varying different sizes, which provided execution times and other measurable data points. These data points showed that there was an exponential increase in speed-up when there was an increase in thread counts, which highly supports the idea that parallelizing the 9-point stencil algorithm increased the performance overall. Although it can be noted that after a certain size threshold the efficiency of the parallelized code became less than desired. As such after implementing OpenMP it was decided to implement MPI to increase the scalability of the stencil program. MPI was tested with the same parameters as OpenMP was tested with and yielded the same exponential results as the OpenMP except it allows for higher scalability since it can be run on multiple nodes unlike OpenMP.

References

[1] Robey, R., amp; Zamora, Y. (2021). Parallel and high performance computing. Manning.

[2] Xsede User Portal: San diego supercomputer center expanse user guide. XSEDE User Portal — San Diego Supercomputer Center Expanse User Guide. (n.d.). Retrieved April 12, 2022, from https://portal.xsede.org/sdsc-expanse