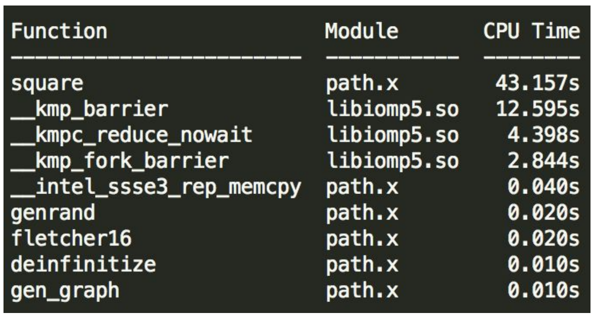
# CS 5220 Project3 Group 4 Final

Yuan Huang(yh638) Robert Carson(rac428) Ian Vermeulen(iyv2)

For the first stage of the  project, we have done the profiling, the MPI parallelization and some tuning on the code.

## 1. Profiling

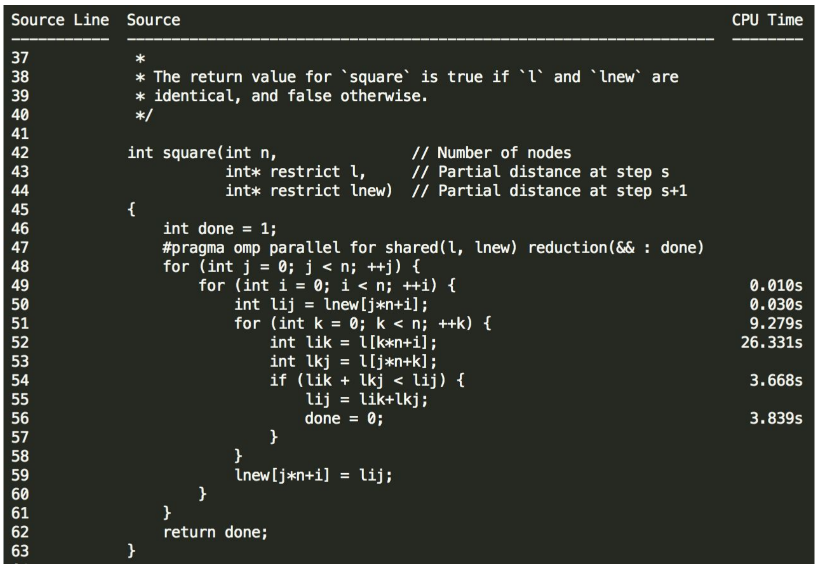
The first step before attempting to improve performance was to profile the code and identify the hotspots where performance could use the most improvement. To do this, we used VTune Amplifier on the compute nodes. The overall performance measurements are shown below in Figure 1.a.



```
Function                  Module       CPU Time
------------------------  -----------  --------
square                    path.x        43.157s
__kmp_barrier             libiomp5.so   12.595s
__kmpc_reduce_nowait      libiomp5.so    4.398s
__kmp_fork_barrier        libiomp5.so    2.844s
__intel_ssse3_rep_memcpy  path.x         0.040s
genrand                   path.x         0.020s
fletcher16                path.x         0.020s
deinfinitize              path.x         0.010s
gen_graph                 path.x         0.010s
```

Figure 1.a Overall Profiling of the Code

Unsurprisingly, most of the execution time is spent in the square method. This report also showed us that a significant amount of time was being spent in openMP modules, indicating room for improvement. We drilled down into the square  method and the results are shown below in Figure 1.b.



```
Source Line  Source                                                             CPU Time
-----------  -----------------------------------------------------------------  --------
37            *
38            * The return value for `square` is true if `l` and `lnew` are
39            * identical, and false otherwise.
40            */
41
42           int square(int n,               // Number of nodes
43                      int* restrict l,     // Partial distance at step s
44                      int* restrict lnew)  // Partial distance at step s+1
45           {
46               int done = 1;
47               #pragma omp parallel for shared(l, lnew) reduction(&& : done)
48               for (int j = 0; j < n; ++j) {
49                   for (int i = 0; i < n; ++i) {                                 0.010s
50                       int lij = lnew[j*n+i];                                    0.030s
51                       for (int k = 0; k < n; ++k) {                             9.279s
52                           int lik = l[k*n+i];                                  26.331s
53                           int lkj = l[j*n+k];
54                           if (lik + lkj < lij) {                                3.668s
55                               lij = lik+lkj;
56                               done = 0;                                         3.839s
57                           }
58                       }
59                       lnew[j*n+i] = lij;
60                   }
61               }
62               return done;
63           }
```

Figure 1.b Profiling of square()

## 2. Performance and Speedup

### 2a. MPI implementation

In an attempt to increase the performance of the code, the parallelization model was changed from one that uses a shared memory model (OpenMP) over to a distributed memory model (MPI). In order to use MPI, a 1D domain decomposition was performed along the row indices of the Lnew array. Next since pointers are not valid across processors, the actual contents of the  L array need to be passed between processors. Although, when L is being initialized it is desired to only have the master processor initialize the L array. Then the initial values of L can be passed to all the other processors using the MPI_Bcast call. This means that the contents of Lnew have to be passed to every processors copy of L during the for loop of the shortest path function using the MPI_Allgatherv command. Overall, the problem with this is that several barriers are placed in the MPI function calls every loop count which will lead to slow downs. Later versions of this code could get around this by using less safe methods of MPI calls that don’t rely on the intrisic use of a barrier. These methods would need to ensure that the data is not touched while it is getting transfered between processors.

### 2b. Performance

The running time of MPI and OpenMP code are shown in the following figure. We can see that the MPI version of code is faster at small size of graph and OpenMP is faster at large size. That’s because OpenMP can use shared memory while MPI have to exchange the datas among the processors. When the size of graph gets large, the number of data needed in communication becomes large. So the MPI code spend a lot of time in exchanging the datas among processes. That’s why the MPI code works better at small size but performs badly at large size.

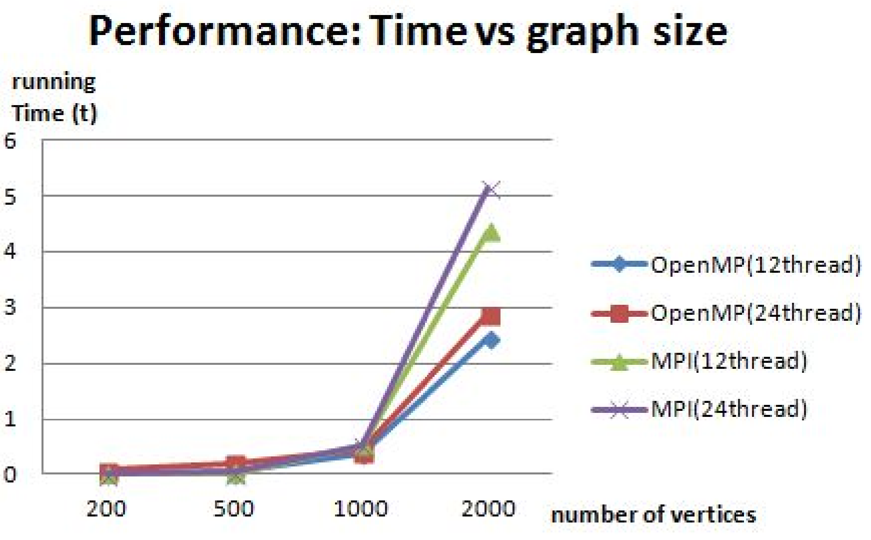


Figure 2.b running time vs number of vertices for different code

## 3.Performance models

### 3a. Weak and Strong Scaling for OpenMP

The original OpenMP version of the code can be analyzed for its strong and weak scaling capabilities. The strong scaling will be tested with a matrix that is 2000x2000. The strong scaling will be defined by Equation 1, and then strong scaling efficiency will be defined by Equation 2. The results can be seen in Table 1.

$$Strong\ Scaling\ =\ \frac{t_{serial}}{t_{parallel}} \tag{1}$$

$$Strong\ Scaling\ Efficiency\ =\ \frac{t_{serial}}{t_{parallel}}\frac{1}{p} \tag{2}$$

where $t_{serial}$ and $t_{parallel}$ are the times taken when running in serial or parallel, respectively, and $p$ is the number of the threads. The strong scaling efficiency is essentially what percentage of perfect linear scaling is actually achieved. This is plotted on Figure 3.a.1.

| Threads | Total Time (seconds) | Strong Scaling | Strong Scaling Efficiency |
|---|---|---|---|
| 1 | 2.95E+01 | 1.00E+00 | 100% |
| 2 | 1.28E+01 | 2.29E+00 | 115% |
| 4 | 6.30E+00 | 4.68E+00 | 117% |
| 8 | 3.55E+00 | 8.30E+00 | 104% |
| 12 | 2.73E+00 | 1.08E+01 | 90% |
| 16 | 2.39E+00 | 1.23E+01 | 77% |
| 24 | 2.89E+00 | 1.02E+01 | 43% |

Table 1: Strong Scaling for path simulation with 2000 x 2000 matrix size.

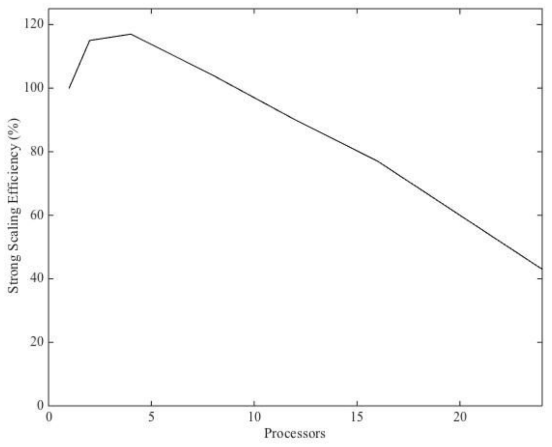


Figure 3.a.1: Plot of strong scale efficiency representing data from Table 1.

Based on Figure 3a, it can be seen that after 8 threads that the amount of performance gain obtained by adding more threads goes down. In fact based on Table 1, it can be seen that between 16 and 24 threads that it actually hurts the performance of the code to add more threads.

For weak scaling, five different simulations were performed scaling the number of threads equally with the number of elements per side of the path. The results can be seen in Table 2. Weak scaling will be defined by Equation 3. In these tests, an increase of the number of elements by a factor of n led to an increase of threads by a factor of $n^2$. A plot of the weak scaling can be seen in Figure 3.a.2.

$$Weak\ scaling\ =\ \frac{t_{serial}(n(p))}{t_{parallel}(n(p))} \tag{3}$$

| Threads | Elements per side | Total time (seconds) | Weak scaling |
|---|---|---|---|
| 1 | 200 | 1.02E-02 | 100.0% |
| 2 | 283 | 1.95E-02 | 52.5% |
| 4 | 400 | 2.67E-02 | 38.3% |
| 8 | 566 | 5.70E-02 | 17.9% |
| 16 | 800 | 1.53E-01 | 6.7% |

Table 2: Weak scaling for path simulation with varying elements and threads

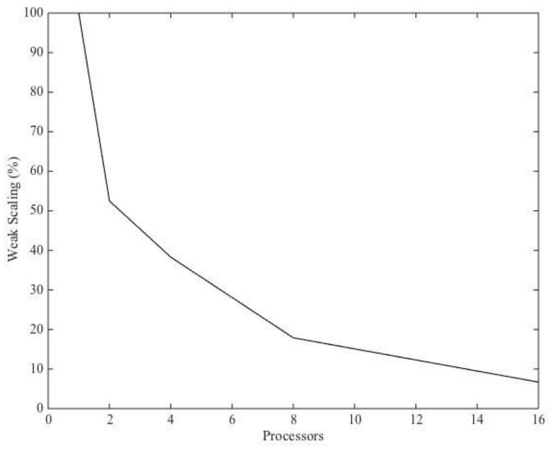


Figure 3.a.2: Plot of weak scaling representing data from Table 2.

The OpenMP does not currently have terrific weak scaling attributes, and this is due to several different reasons. One of them is that each time the square function is called a new team of threads needs to be created. This is an expensive operation, and the code could be set up such that this only occurs once. Next, a domain decomposition as the ones discussed in class for the water assignment could be of use here as well to better the weak scaling.

## 3b. Weak and Strong Scaling for initial naive MPI

The same methodology used for the strong and weak scaling for the OpenMP is used for the MPI codebase. However, it should be noted that since only 12 processors are available on each node a one to one comparison between the OpenMP and the MPI codes. The strong scaling results are given in Table 3. The results of the strong scale efficiency are given in Figure 3.b.1.

| Processes | Total Time (seconds) | Strong Scaling | Strong Scaling Efficiency |
|---|---|---|---|
| 1 | 2.70E+01 | 1.00E+00 | 100% |
| 2 | 1.26E+01 | 2.14E+00 | 107% |
| 4 | 9.02E+00 | 2.99E+00 | 75% |
| 8 | 5.26E+00 | 5.13E+00 | 64% |
| 10 | 6.80E+00 | 3.97E+00 | 40% |
| 12 | 4.51E+00 | 5.98E+00 | 50% |

Table 3: Strong Scaling for path simulation with 2000 x 2000 matrix size.

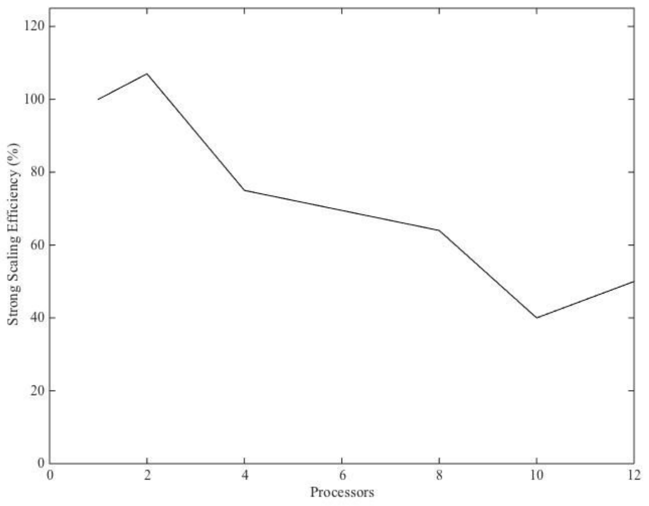


Figure 3.b.1: Plot of strong scale efficiency representing data from Table 3.

It can be seen when comparing Table 1 and Table 3 that the OpenMP currently has better strong scaling. It can also be seen from Figure 3.b.1 that the strong scale efficiancy starts to improve towards the end. One possible reason behind this could be that the MPI framework is better able to communicate across the entire node instead of just portions of it.

The weak scaling part of the codebase still had 5 simulations. Although, it was not able to quite have a one to one comparison with the OpenMP weak scaling. The resuts are given in Table 4, and then those results are plotted in Figure 3.b.2.

| Processes | Elements per side | Total time (seconds) | Weak scaling |
|---|---|---|---|
| 1 | 200 | 8.65E-03 | 100.0% |
| 2 | 283 | 1.91E-02 | 45.2% |
| 4 | 400 | 2.58E-02 | 33.5% |
| 8 | 563 | 5.83E-02 | 14.8% |
| 12 | 693 | 1.11E-01 | 7.8% |

Table 4: Weak scaling for path simulation with varying elements and processes

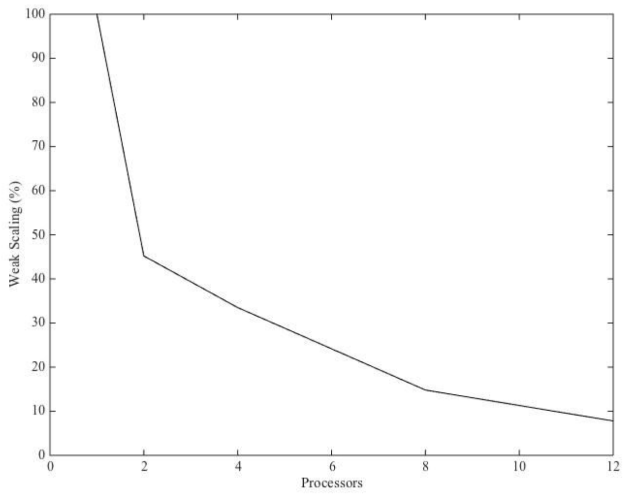


Figure 3.b.2: Plot of weak scaling representing data from Table 4.

Based on this figure, the weak scaling performs worse than the OpenMP codebase. One reason behind this poor weak scaling could be due to the current number of barriers that the MPI code makes when spreading the update of the L array to all the processors. Then it also has to make a reduce call to determine if all the processors have finished. Improvements could be made by implementing Canon's algorithim into the square function instead of the simple domain decomposition used currently.

## 4. Progress since stage one

We have done the following tunings since stage 1:

1) Making the communication in the MPI code non-blocked. We are currently using blocked communications in the code, making the communication unblocked can let us do the calculation work while waiting for the communications to finish.

2) Applying the blocking and copying tunings using the methods in matrix multiplication problem.

3) Use better domain decomposition method instead of the simple stripe domain decomposition.

## 5. Applying serial tunings

We tried the methods we used in matrix multiplication problem.

### 5a. Transpose on the right matrix while copying

We use transpose of  the matrix to get better locality. For Matrix multipilication C=A*B, we can visit both matrix on the right rowwise order after transpose. This can give better locality thus improve the performance. We do the transpose while copying, which can minimize the cost of doing the transpose.

## 6. Tuning in communication

We make the communication between processes in the MPI code.

### 6a. master and slave model

We take one process to just do the communication, copying and reduction of the result. We make this decision because if we want to let the master also do the calculation, other process have to wait till it finishes. The waiting will take a lot of time.

### 6b. Domain decomposition

If we want to do the non-blocking communication, we should cut the domain into smaller pieces. That's because we should have something to work on when we are doing the communication on other things.

The following graph 6b.1 shows how we did that:

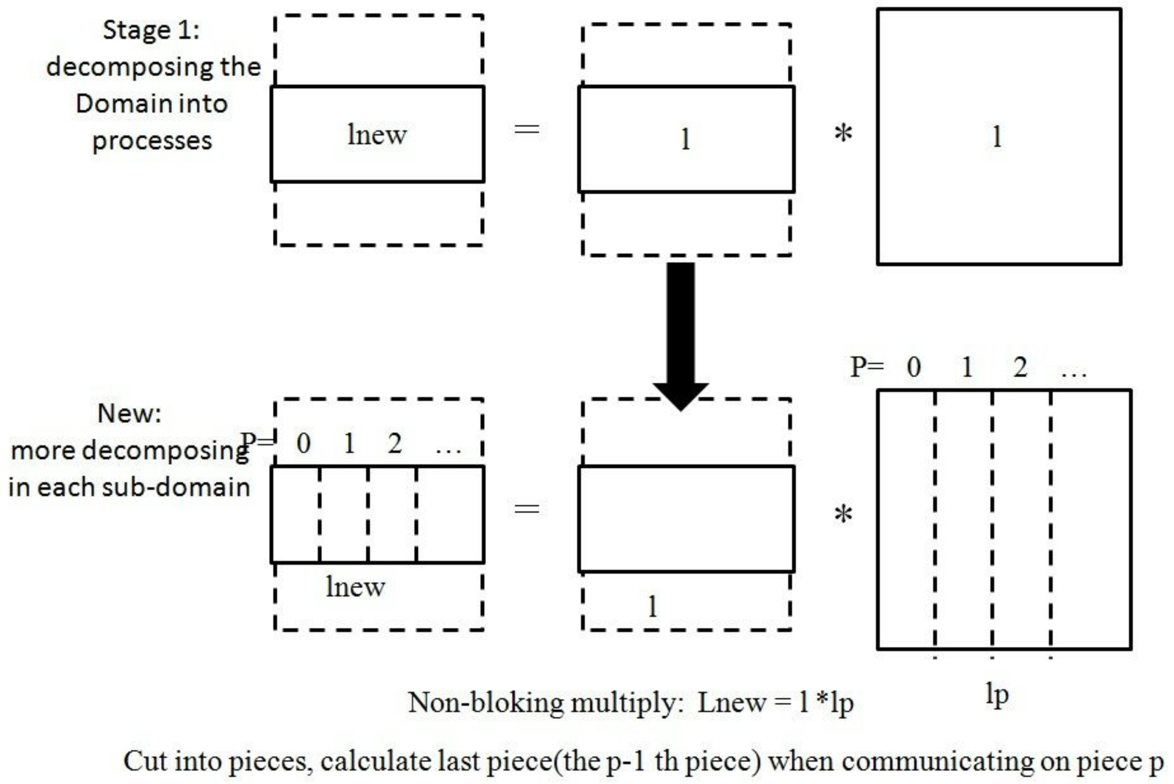

Stage 1: decomposing the Domain into processes
lnew
=
l
*
l
New: more decomposing in each sub-domain
P= 0 1 2 …
lnew
l
lp
Non-bloking multiply: Lnew = l *lp
Cut into pieces, calculate last piece(the p-1 th piece) when communicating on piece p

figure 6b.1

We first send the piece p=0 of the  to the process. And then for each time, we do the shortest path on the matrix l and the part p-1 th on matrix lp to get the p-1 th part on lnew. At the same time we do the communication of the pth part of lp. So we can do the communication and the calculation at the same time.

## 7. Peformance Model of Tuned Code:

### 7.a Strong and Weak Scaling of Tuned Code:

The same methodology used for the strong and weak scaling in Section 3 is used for the tuned MPI codebase. However, it should be noted that since only 12 processors are available on each node a one to one comparison between the OpenMP and the MPI codes. The strong scaling results are given in Table 5. The results of the strong scale efficiency are given in Figure 5.a.1.

| Processes | Total Time (seconds) | Strong Scaling | Strong Scaling Efficiency |
|---|---|---|---|
| 1 | 2.70E+01 | 1.00E+00 | 100% |
| 2 | 5.77E+00 | 4.67E+00 | 234% |
| 4 | 1.41E+00 | 1.92E+01 | 479% |
| 8 | 6.78E-01 | 3.98E+01 | 497% |
| 10 | 5.65E-01 | 4.77E+01 | 477% |
| 12 | 4.50E-01 | 5.99E+01 | 499% |

Table 5: Strong Scaling for path simulation with 2000 x 2000 matrix size.

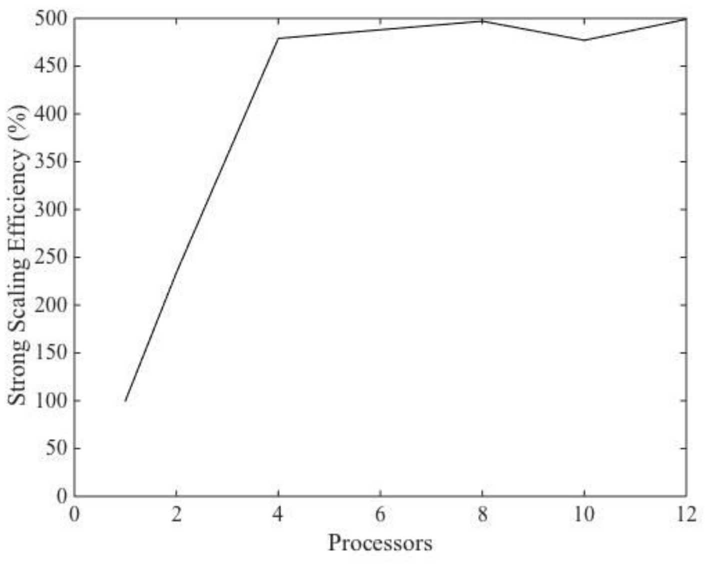


Figure 7a.1: Plot of strong scale efficiency representing data from Table 5

It can be noted from Figure 7a.1 that a large increase in the strong scaling efficiancy can be noted in comparison to the original OpenMP and naive MPI codes. One of the biggest reasons behind this is that processors are able to do additional work while the communications are occurring. Thus, the large communication costs are now hidden up to a degree. Also, it can be seen from Figure 6b.1 that the domain decomposition is similar to a rectangular blocking scheme that might be used in a matrix - matrix multiplication code to reduce the amount of cache misses. When the number of processors are increased the amount of time spent during the calculations between communications calls are decreased, so the limiting factor becomes again the communications costs. Therefore, it can be seen that the strong scaling efficiency starts to reach a steady state.

The weak scaling part of the codebase still had 5 simulations. Although, it was not able to quite have a one to one comparison with the OpenMP weak scaling. The resuts are given in Table 6, and then those results are plotted in Figure 3.b.2.

| Processes | Elements per side | Total time (seconds) | Weak scaling |
|---|---|---|---|
| 1 | 200 | 8.65E-03 | 100.0% |
| 2 | 283 | 1.56E-02 | 55.6% |
| 4 | 400 | 1.29E-02 | 66.9% |
| 8 | 563 | 2.94E-02 | 29.4% |
| 12 | 693 | 2.92E-02 | 29.6% |

Table 6: Weak scaling for path simulation with varying elements and processes

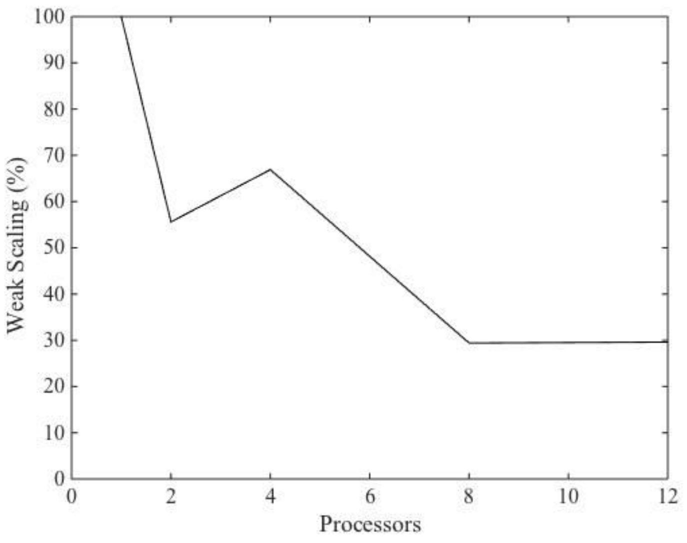


Figure 7a.2: Plot of weak scaling representing data from Table 6.

Based on the above figure, it can be seen that the weak scaling for the tuned code is better than both the OpenMP and naive MPI. It can also be seen that a steady state is starting to be reached for the tuned code relatively early at 8 processors, while the other two versions saw a continued decline in the weak scaling.

## 8. Conclusion

Our goal for this project was to implement the Floyd-Warshall algorithm using MPI in a more efficient way than the initial openMP implementation. Our first attempt worked, but was not any faster than the original implementation. The high overhead of communication between processes and the blocking nature of these communications prevented any speedup. In our second implementation, however, we focused on serial and parallel tunings rather than simply trying to implement the code using MPI. By performing copy optimizations to improve the serial code and implementing more advanced domain decomposition to minimize blocking in each process, a significant speedup was acheived.