Parallelizing Dijkstra's Algorithm

Cain Wock

Matthew Hill

Cody Ponder

Deirdre Brennan

# Abstract

Given a graph, our goal is to apply Dijkstra’s algorithm in parallel to find the shortest path from a root node to a target node. This will be done using the MPI API in the C language to implement this across multiple processes.  The same process will be done with openMP in order to compare the differences between each approach. What we have seen is that neither openMP or MPI are effective at parallelizing Dijkstra’s algorithm at smaller graph sizes, ranging up to around 100 vertices, while both becoming more effective as the sizes increase past this amount. After a certain point, openMP appears to run at the same times no matter how many threads are used while MPI only continues to increase in speed the large the graph becomes, and the more threads are used.

# Background

Finding the shortest point in a graph can be expanded to a variety of real-world applications including mapping for finding routes when travelling, finding the quickest route across a network for data transmission, and video game AI for moving across a map or accomplishing a set of tasks. The algorithm we plan to parallelize is Dijkstra’s algorithm. This algorithm involves picking a “base” point and scanning all connections to this point. The distance to each of these connections is noted and stored with that node. Using these results, the next “base” point is the node with the shortest distance from the original source that has not yet been used as a base. Once the base node changes, the previous base node will no longer be scanned by any node. This process continues with each base node scanning connected nodes either updating their distance to the current base distance from the source plus the distance to that node or keeping the previous value; whichever is smaller. The process terminates when either the destination node becomes the base node, or there are no more nodes to be scanned. The final value returned will be the value assigned to the destination node, the shortest path (1).

The part of this algorithm that can be parallelized is the scanning. The task of scanning nodes adjacent to the base node can be split among many processes. However, information gathered from each process must be returned before the next base node can be chosen. This means that the maximum number of processes running at a given time must be less than or equal to the number of connecting nodes to the current base. This type of parallelism will see the greatest benefit in a graph where each node has many connections. With this proposed implementation, each new base chosen will be dependent on the distances scanned from the previous base, so a lock will have to be implemented to ensure that all processes finish before the next phase of the program commences. Being able to parallelize this process is important in all of the aforementioned applications.  The parallel program we created takes a csv file describing a weighted graph and outputs the distance to each node on the graph as well as the run time for computation. The user is also able to specify the number of processes used to execute the program. For the sake of consistency, our program also exports the results into an excel file and runs each example a multitude of times.

Dijkstra's algorithm has many real-world uses today.  One real world application for this algorithm is in setting up computer networks.  In networking, the number of connections with each individual node can reach high numbers when arranged in either a star, mesh, or fully connected topology. Since each connected node would have to be pinged in order to find the distance, the process of finding each distance composes the vast majority of the run time. Using parallelism to find every distance simultaneously vastly reduces the total time required to find the shortest networking path.

Another use of the algorithm relates to map navigation.  With the inevitable implementation of automated vehicles, shortest route navigation is more important than ever before. As there are often a multitude of different routes to a given destination, finding the shortest path is critical to stay competitive and retain a happy user base. Since the shortest path may be as a unit of time rather than distance, the value may change based on weather, construction, or traffic conditions. Factoring these variables into each path makes the process of finding connection values the most time intensive part of the mapping process. By running this one in parallel, a service would be able to deliver the quickest route to the user faster. This is especially important to the navigation system, where a delay of a few seconds could mean a detour of minutes.  While the basis of all this navigation is in Dijkstra’s, the A\* algorithm has been developed as an improvement in processing time. The difference between the two methods is that A\* takes into account the coordinates of the final destination. This means that while a Dijkstra’s graph will spread out in all directions, the A\* will prioritize paths that lead close to the destination. However, this algorithm can only be used when coordinates of the final destination are available. This limits the practicality to operating on a coordinate plane.

Dijkstra’s algorithm can also be used in programming video game AI. In older games that took place on a square grid like chess, or even a hexagonal grid, the shortest path was easily computable due to the small number of connections between each node. However, with the modern-day capability to move in almost any direction in many games and to encounter both irregular or temporary obstacles, finding the shortest path is a far more computationally taxing endeavor. Being able to find multiple paths simultaneously would allow a quicker route computation given the large number of possible connections. Speed is an important factor in a medium with user interaction and a constantly changing environment. This is another example where the A\* algorithm has taken over for more efficient calculations.

The last example in which Dijkstra’s algorithm can be of use is the tracking of infectious diseases.  Given an epidemic, researchers can analyze a graph of contacts of those who are sick and make predictions about the spread of the disease.  Based on factors like proximity and likelihood of interaction, this method has the potential to create more accurate quarantine zones and ensure the correct quantity of treatment is delivered to the right locations.  Overall, as problems in this format become more complex over time, the need to parallelize the process becomes more important.

Parallelizing Dijkstra’s algorithm has a few intellectual challenges.   One part of the program that offers a challenge is dealing with the race conditions that arise when finding different routes to the same nodes. When looking at multiple routes, the end destination can occur from multiple paths, and if one path is shorter, but has not been calculated yet, the program may assume a longer route is the shortest. This forced us to come up with a method for determining new routes while not missing ones yet to be calculated. Maintaining communication between processes was going to be important and would require locks to be in place at critical points such as rebasing. The majority of this program helped parallelizes a serial process. Since Dijkstra’s algorithm is at its core serial, we had to modify it in a way that allows the most efficient use of each process and minimizes the time spent performing serial operations. Each step of the algorithm had to be communicated to each process in order to prevent redundant routes from occurring. With constant communication, a means to optimize the process of relaying each processor’s work must be determined. The most challenging aspect of this is that Dijkstra’s algorithm is presented serially in most examples, and using MPI, global variables are not updated across multiple process without specification. To prevent making the program too serial, a method for communication was decided that best uses MPI to communicate without forcing multiple processes to wait for the results from another.

# Approach

In order to create our program, we used parts of some base serial implementations as a code base. Since this is a widely used algorithm, a few examples were available online (1).  Stack Overflow was also a valuable resource throughout this project. The forums page often provides insight on how to use specific sections of code to their best ability. We also used the MPI documentation quite often, as there were many MPI calls required for this program.  Each process is looking for the shortest route, broadcasting to the other processes if the shortest route is found and listening to hear if any of the other processes have found the shortest route. For the graphs we used for calculations, we used multiple kinds to show the varying effect of parallelizing on the speed of calculation. This began with small quantities of nodes (i.e. 10) with small quantities of connections (i.e. 2 per node) and scaled up to larger quantities of nodes with large quantities of connections.  Rather than creating a separate program for serial implementation, our serial program took the form of running the parallelized program with only a single thread. Our implementation took the form of two programs: A parallelization using MPI, and a parallelization using openMP.

The MPI implementation went through several interactions. The basic logic of the implementation was to separate the calculations distance between the different nodes as much as possible. This allowed for best parallelization of what work there is to do for the algorithm. This initially was difficult due to the fact that, especially will small graphs and graphs with small connections, dependencies on the total distance between the source node and each other node. This dependency required previous calculations to be done, forcing a certain degree of linearity in the program. In addition, since separate processes were doing calculations that had dependencies elsewhere, it required constant communication between the processes.

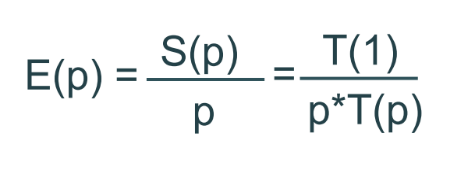
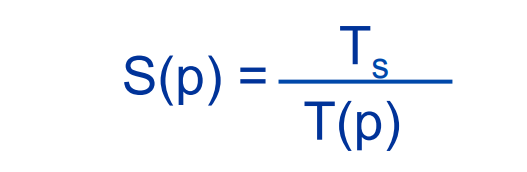
An initial, and extremely significant, issue that was present with each iteration of the MPI application was memory allocation. Each size graph that was used in the testing was hard coded in separate files that needed to be read into the program. Each process had a section of the graph designated to it in order to calculate the different distances. There were several different ways that this could be done, but to limit the amount of memory allocated, rather than allocate the entire graph to each process, only a specific portion was allocated. This problem was significant enough that the hard-coded graphs used in testing openMP couldn’t be used for MPI. Instead, randomly generated graphs were used.

The first iteration of the MPI program started quite complicated. A source code was found available online that we used to start with and make changes to fit our needs. This proved to be problematic for two main reasons. Issues with memory allocation were the first, and the second was that due to the complexity of the way the program was written, modifications and debugging quickly became impossible. The next iteration struggled with proper application of MPI calls used in combination with Dijkstra’s algorithm. Along with memory allocation issues, the large amount of debugging and improper use of MPI calls to manipulate the graph caused the need for a new iteration. The final iteration that was used also struggled with memory allocation, this issue was, at least to some degree, solved. However, the solution was based on source code uploaded to Scribd, written by Dionysios Zelios, a computational physicist. The reading of the hard-coded graphs still struggled with memory issues and caused continuous segmentation faults. However, it had a means of randomly generating a graph, this allowed us to process data of a given size, even though it was not the same graphs. Giving us a means to interpret the application of MPI Dijkstra’s algorithm. Although the randomly generated graphs were not necessarily accurate in their exact structure. It allowed us to run the program and do calculations as though it were a proper graph. This gave us run times and a general idea of how MPI as a whole would impact Dijkstra’s algorithm, as well as what improvements could be done.

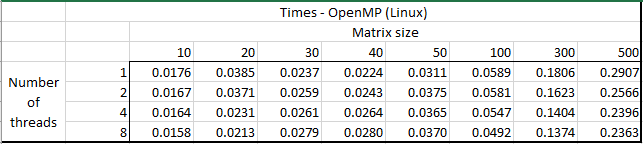
The program using openMP proved easier to implement than using MPI.  The syntax of openMP allows it to be utilized in only a few lines of code before each parallelized section.  The first part of the program that we parallelized was allocating the memory for the matrix size. This uses a given number of threads to make the arrays in a for loop.  After the distance arrays have been created, we can also use an openMP pragma to parallelize setting the initial distance of each not to infinity and setting the visited Boolean to false.  While the overall shortest path algorithm cannot be parallelized, sections of it can be. Our program uses an openMP pragma to update the distance to each adjacent node from the base simultaneously.  The section that takes this information to make a rebasing decision must be run in serial due to the dependency of whatever node is the currently smallest distance node that has yet to be visited. After the program is finished, we also use parallelization to free each matrix in the same format that we allocated the memory in the beginning.

# Results

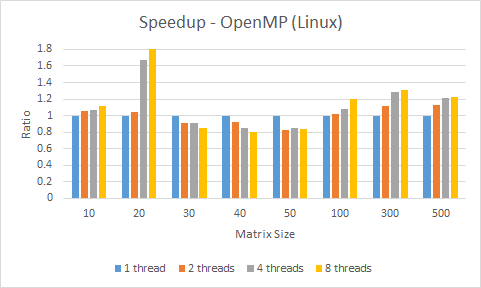
For our results, we took a variety of different-sized graphs. We did size 10, 20, 30, 40, 50, 100, 300, 500 node graphs to vary the input size that would be run within each of the two programs. Then, each of the different graphs was run using 1 process for the linear result. After that, 2, 4, 8, 12, and 16 processes were used. This gave us 48 runtimes, 8 linear and 40 parallel. The speedup and efficiency for each was then calculated. Speedup was calculated doing linear runtime divided by parallel runtime. Efficiency was then calculated doing linear runtime divided by parallel runtime, then divided by the number of processes.

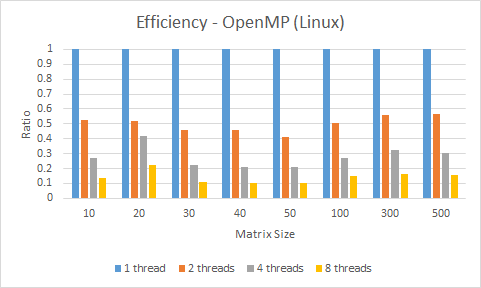
(3)

In order to maintain consistent results, all programs, with the different-sized graphs and different process counts, were ran at the same time and from the same machine. This allowed for each result to be as close to similar as possible. Another not is that our times to not include the parsing of data.  We found that this was difficult to make a parallel implementation of, and when included in the time took a significant portion of the processing. Because such a large part is spent on parsing data, it would be difficult to see the correlation between thread number and speedup if we had included it in our tests.  When testing on the Linux machine, our baseline was defined by the time needed to run the program with only one process. The following figure depicts this information for openMP. Our initial times were obtained by running the openMP code 100 times using a loop on the terminal. This led to strange times when running in parallel as none of the graphs ran any faster with multiple threads. When this changed to a for loop within the code to run the timing and Dijkstra’s algorithm then finally averaging out the times it provided much better results that provided better running time when parallelized.

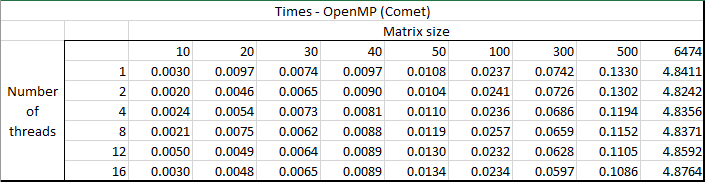


Using these times, we also calculated the speedup and efficiency of each matrix given the different thread sizes.

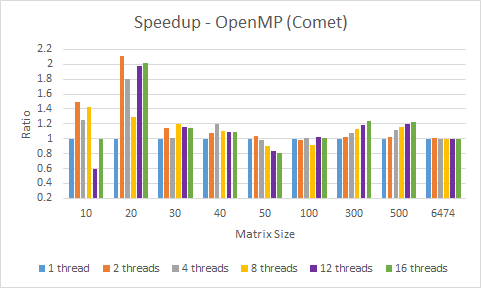


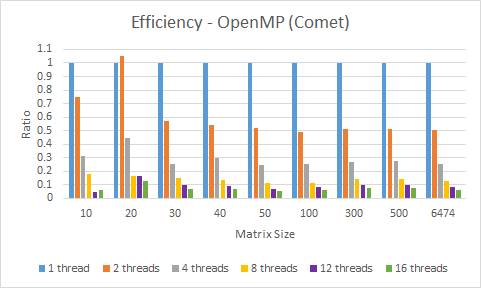


Our results show that given a large number of trials, the openMP approach, when ran on the school Linux system, this demonstrates some effectiveness of parallelization.  However, if someone is determining the effectiveness of the parallelization using openMP then the efficiency is not that great as even with 500 vertices at 2 threads, it is 0.52% as efficient as the serial version. The optimal matrix size in our results is twenty vertices. Additionally, as the matrix size trends toward very large values, the speedup increases after a 100 vertices count graph. We think that some inconsistencies can be attributed to the variable performance of the school’s server. Because we are running a program with such a low runtime, a delay of even fractions of a second in our medium can throw off the results completely, although this only appears to be a problem with smaller sized graphs.  In order to try and create a more reliable data set, we ran the same parameters using the Comet system. The following figures show these results.



As it can be seen, with a smaller data size it becomes very erratic in terms of effectiveness. It doesn’t start to get better until around the 100 vertices graph size. With this we can see the speedup & efficiency of openMP using the Comet cluster.

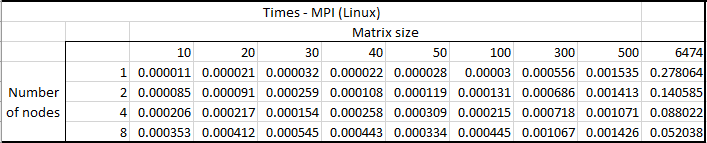




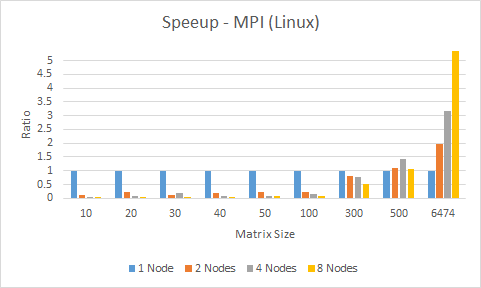
Using Comet, it was easier to observe the trend in the effectiveness of openMP on matrices of varying sizes.  The optimal improvement was found in a matrix with 20 vertices. One we reached very large graphs, the speedup did improve again but not as effectively as with 20 vertices. We are not sure why 20 vertices were the best point for both the Linux and Comet servers other than because it is such a small sample that it is not very reliable to determine effectiveness of the parallelization. At lower sizes, it is generally worse to parallelize Dijkstra's algorithm as it appears to take more time allocating resources for the threads and the sharing of data between them than running in serial takes.

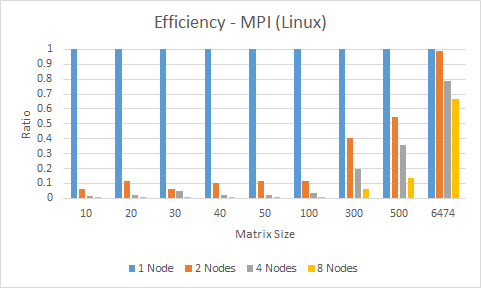
As the graph sizes go up, we do notice that the graphs are more complex with more connections per node. The more interconnectivity you have between nodes, the more effective parallelization is. Once the graph sizes reach around 100 vertices, there is an increase trend in speed which only gets better as more threads are used to run the code. This trend appears to be true until a real-world example we obtained of 6474 vertices. At this point there is no effective difference in speed between any number of threads used. This may be due to the overhead that openMP has that simply slows down the gathering of data between all the threads that causes it to be ran at times similar to in serial while the efficiency quickly declines as more threads are used.

Unfortunately, when running the MPI program on the Comet, several errors occurred, which prevented us from getting run times from them. However, we did multiple runs on the Linux machine in order to find a steady runtime for 1, 2, 4, and 8 nodes. Each being ran on graphs of size 10, 20, 40, 50, 100, 300, 500, and 6474. This allowed us to give direct comparison between MPI and openMP, as well as determine how effective MPI is at improving Dijkstra’s algorithm. The runtimes are presented in the following graph, with the time listed in seconds. The time was calculated by using the MPI\_Wtime() method, starting when Dijkstra’s algorithm begins computations and ending once the final computations are done and compiled in the core node.



Based on these times, it was clear that MPI was very poorly suited for small data sizes. As can be seen in the following speedup and efficiency graphs, Dijkstra’s algorithm was best running linear in any graph with less than 500 nodes.





With the efficiency and speedup, it became very clear that the large amount of communication that occurs with MPI causes significant slowdown when working with small amounts of data. This is due to the significant number of calls such as MPI\_Scatter and MPI\_Gather, that requires sending and receiving data with all nodes. All graphs with the exception of 500 and 6474 had increase in runtime as the number of nodes used were increased. This is clear since the amount of communication increases. Once we reached the 500 graph size though, we saw a reasonable speedup with 2 and 4 nodes, but minimal at 8 nodes. This was likely the cap on how efficient a graph of size 500 could be made through parallelization. When a substantially larger graph was used, of size 6474, we saw much clearer speedup and efficiency. In fact, we saw an efficiency of nearly 1 with 2 nodes, which showed that when using MPI on larger graphs, it can become much more efficient. This is mainly what we expected to see. The inclusion of a larger graph was done specifically when we saw that the 500 graph started to show slight improvements on runtime, this was especially useful in giving room to see where MPI improves better than openMP.

# Conclusion

Going into this project we believed that Dijkstra’s algorithm could be parallelized effectively using MPI and openMP.  After using both methods we determined that each method has a distinct advantage over the other. MPI is a more portable environment that is available on a larger variety of machines.  While it is difficult to implement, it allows the user to have a lot of control over each aspect of the parallelization process. In contrast openMP holds the advantage of being remarkably simple to implement.  In only a few lines of code the user can parallelize their program. The chief drawback of this is overhead cost and availability.

Compared to each other, openMP and MPI are not effective at smaller sized graphs. However, openMP appears to become relevant faster than MPI but dips in effectiveness at very high sized graphs which MPI picks up to be only better as more nodes are added to compute the algorithm. MPI efficiency becomes better as the sizes go up compared to openMP which appear to stay consistent no matter the size and number of threads used (some outliers excluded).

In terms of effectiveness, we believe that with a moderately sized graph (100-1000 vertices) openMP would be the better choice to use. If the graph is complex in that each node has a high chance of having multiple connections based on the size of the graph, then more threads would be better to use, otherwise, less as then some threads wouldn’t be working otherwise. At larger sized graphs (1000+ vertices) MPI appears to be the better choice since the speedup ratio increased vastly at a very large sized graph. As with openMP the number of nodes to use would be determined by how complex the graph itself is.

In terms of effectiveness, it appears that Dijkstra’s algorithm can be parallelized with varying degrees of efficiency. As with most things, lower sized data would normally make the time required to parallelize the algorithm actually take longer. So as long as there is a large enough data, Dijkstra’s algorithm can be effectively parallelized with either openMP or MPI.

Looking back on it, there are several things which we could have done differently to help us determine how effective it is to parallelize Dijkstra’s algorithm. One thing that we could have done (which we couldn’t figure out how to do) is get MPI working correctly so we could compare the difference between it and openMP on both the Linux terminal and the Comet cluster. This would have allowed us to have a more precise comparison between the two, rather than the rough estimate we did obtain. However, it likely would not have changed the overall results that showed MPI was inefficient on small graphs, and more efficient on larger ones. Another thing we could have done was create a program that parallelized Dijkstra’s algorithm by using Pthreads and another one using a hybrid of MPI and openMP. This would have given us more data to determine which is best at different data sizes and number of threads/processes being ran. One more thing we could have done is create graphs of the same size with varying degrees of complexity to see if it is effective to parallelize the algorithm no matter the size or only if the complexity reflects the size of the graph. Hybridization could have likely been best on the smaller graphs, and allowed the creation of a program that was more consistent in efficiency when moving from smaller graphs to larger ones. This would have been potentially possible due to the ability to limit communication between processes using a hybrid of openMP and MPI.

# References

1. https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm 2.
2. http://demonstrations.wolfram.com/WeightedRandomGraph/
3. IT388\_Lect05\_performance.pdf, presented by Dr. Follman
4. <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>
5. https://www.scribd.com/document/263886667/Parallelization-of-Dijkstra-s-algorithm
6. <https://snap.stanford.edu/data/>
7. Mpitutorial.com

# Appendix

A\* algorithm : Widely used process of finding a path between multiple points using heuristics.

Dijkstra’s algorithm : Algorithm for finding shortest paths between nodes producing a shortest-path tree.

Efficiency : The ratio of useful work as compared to resources.

MPI : Message Passing Interface project library that can be used for parallel processing often by supercomputers.

MPI\_Gather : MPI command takes elements from many processes and gathers them into one process.

MPI\_Scatter : MPI command that sends chunks of an array to different processes.

OpenMP : An API that implements multithreading, a method of parallelizing threads concurrently allocating threads to different processors

Pthreads :POSIX Threads; a parallel execution model that allows a program to control multiple different flows of work that overlap in time.

Speedup: a measure of the relative performance of two systems processing the same problem. Seen here as a different number of threads processing the same problem.

### Source Code

dijMPIv2.c : This is the MPI program that was run on given Matrices

DijOpenMP.c : This is the OpenMP program that was run on given Matrices

### Other Files

We have also included all of the matrices files we tested the program on as well as the .sb versions for comet testing.

Matrix10

Matrix20

Matrix30

Matrix40

Matrix50

Matrix100

Matrix300

Matrix500

Matrix6474

OpenMP.sb