A COMPARISON OF PARALLEL IMPLEMENTATIONS OF PATHFINDING ALGORITHMS

By

Charles Wilcox Johnson

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

August 2021

A COMPARISON OF PARALLEL IMPLEMENTATIONS OF PATHFINDING ALGORITHMS

By

Charles Wilcox Johnson

APPROVED:

Graduate Committee:

_____

Chair Dr. Chrisila Pettey (Computer Science)

_____

Dr. Suk Seo (Computer Science)

_____

Dr. Ferrol Aderholdt (Computer Science)

_____

Dr. Medha Sarkar, Chairperson Computer Science Department

_____

Dr. David Butler, Dean of the College of Graduate Studies

ACKNOWLEDGEMENTS

To my parents, Dr. John W. and Claire J. Johnson, without whom this thesis would not have been possible.

I would also like to thank my advisor and committee chair, Dr. Chrisila Pettey – your help with navigating through the administrative side of the thesis process and assistance throughout the writing, editing, and formatting of this thesis was invaluable. Drs. Suk Seo and Ferrol Aderholdt – your willingness to serve as members of my thesis committee and advice has been most helpful. Dr. Ralph Butler – the information and lessons learned about parallel processing in your classes served as a significant springboard in accomplishing the research for this thesis. And Dr. David Heatherly – for your continual encouragement and support.

ABSTRACT

Pathfinding algorithms are in constant use today – be it for generating driving directions or maintaining the routing tables that are used for routing traffic around the Internet. As would be expected, a given algorithm's performance varies depending upon both the type and the scale of the application. Additionally, the use of parallel computing techniques may or may not decrease the runtime for a given problem. This thesis examines three commonly used pathfinding algorithms – Dijkstra, Bellman-Ford, and Floyd-Warshall – and seeks to illustrate their strengths and weaknesses when analyzing graphs that range from small and sparse to large and dense in various parallel computing environments. As well, focus is given to the use of Graphics Processing Units (GPUs) as parallel computing devices themselves and the consequential reduction in the amount of hardware necessary for a given task.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

CPU – Central Processing Unit

E – The set of all edges in a graph

$|E|$ – The number of edges in a graph

GPU – Graphics Processing Unit

HPC – High Performance Computing

MPI – Message Passing Interface

RAM – Random Access Memory

V – The set of all vertices in a graph

$|V|$ – The number of vertices in a graph

CHAPTER I

INTRODUCTION

While we are usually not aware of their usage, pathfinding algorithms impact our lives every day. They are used to maintain the network routing tables used to route cellular traffic and Internet traffic. They are used to provide driving directions for trips around town and for trips across the country. And they are also used for things we normally don't think about – be it the fastest, shortest, or most cost-efficient routes for the transportation of cargo all over the world or medical and other scientific research, such as analyzing the spread and transmission of a virus.

Fundamentally, there are two classes of pathfinding algorithms – Single-Source Shortest-Path (SSSP) and All-Pairs Shortest-Path (APSP). SSSP algorithms compute the shortest path between a given vertex – the source vertex – and every other vertex in a graph, while APSP algorithms compute the shortest path between any two given vertices in a graph. In essence, APSP is SSSP applied to each vertex in the graph. Both types of algorithms tend to have large worst-case runtimes – $O(|V|^2)$ for SSSP and $O(|V|^3)$ for APSP – thus making lowering the runtime a key objective. This is particularly true for applications that use SSSP algorithms – for example, driving directions that factor in reported traffic delays – that strive to or must offer near-real-time updates, thus requiring the algorithm to be run extremely often.

In the macro sense, lowering the runtime on pathfinding algorithms can be accomplished by simply using faster hardware. Pathfinding algorithms are computationally intensive, which means that a faster processor with more memory and a higher bandwidth conduit to that memory should have a lower runtime. However, even if this proves to be correct when comparing two

different systems, one will eventually hit the limits of technology and be unable to continue the cycle. All of this performance increase, it should be noted, is predicated on the execution of the algorithm with a single thread. Given that these algorithms have many code sections that could be processed in parallel, it is worth exploring the application of parallel processing techniques to pathfinding algorithms.

This thesis will explore utilizing parallel processing on three algorithms – Dijkstra, Bellman-Ford, and Floyd-Warshall. When possible, more than one implementation per algorithm will be analyzed, and for all implementations, they will tested against multiple sets of graphs that range from small and sparse to large and dense so as to see how changes in the graph's properties affect the performance of the implementation. In addition to testing the implementations on CPU-based systems, testing will be done utilizing GPUs for the execution of the algorithms. GPUs offer an incredible amount of computing power, but their architecture may – or may not – lend themselves to decreasing the runtime of a given algorithm in some or all types of graphs. The results will then be compared against each other in an attempt to discern if, for a given graph set or sets, one algorithm is more favorable than another.

CHAPTER II

BACKGROUND

Overview

To understand the issues involved with implementing parallel processing for the algorithms being examined, two distinct areas of interest must be examined. The first is that of the algorithms themselves, learning about both the sequential version of the algorithm and the issues faced with parallelizing it. The second is that of the hardware that can be used for implementing parallel processing because hardware that works well for one algorithm may not perform as well when used for another algorithm.

For both ease of reading and to refrain from repetition, before examinations of the above two areas are commenced, an explanation of some key operations and data structures is in order. The first operation is that of *relaxation*. This takes place when an algorithm examines an edge (u, v) and compares the cost value for vertex u, denoted as cost(u), plus the weight value of the edge, w, to cost(v). If (cost(u) + w) is less than cost(v), then the other key operation – *update* – is performed as cost(v) is set to (cost(u) + w). For SSSP algorithms, the data structure that stores the cost or distance of a vertex from the source vertex is typically an array of size $|V|$ and named 'dist'. A second array is also frequently maintained so as to store the *predecessor* vertices and thus make it possible to map out the actual path between the source vertex and the end vertex – an operation also known as *path reconstruction*. This array is usually named 'pred'. For APSP algorithms such as Floyd-Warshall, which operate on the graph with the entire graph stored in an adjacency matrix, while the distance matrix is usually named 'dist', the matrix used to store the

necessary data for path reconstruction is normally named ‘next’, not ‘pred’, as it is storing successor, rather than predecessor, vertices.

Algorithms

Dijkstra

Overview

In 1959, Edsger Dijkstra published a paper [5] in which he described an algorithm for the solving of Single-Source Shortest Path (SSSP) problems. The algorithm has come to be very popular and is used heavily in computer network routing algorithms, such as Open Shortest Path First (OSPF) [16], as well as applications such as cargo delivery – what is the optimal route – and other applications where the path with the least cost – distance, time, money, etc – between two items is desired. It can handle both undirected and directed graphs. The three constraints for the algorithm is that the graph must be weighted, cannot contain any self-loops, and all of the weights must be positive.

Explanation of the Sequential Algorithm

Figure 1 shows the pseudo-code for the basic or traditional implementation of Dijkstra’s algorithm. The algorithm starts with a graph – *Graph* – which is usually represented in the form of an adjacency matrix, an adjacency list, or a compressed sparse row arrangement and then the number of the vertex that will be considered the source vertex. While the pseudo-code does not show it, in reality, the two data structures for a vertex’s distance or cost and its predecessor would probably be passed in by reference as arguments. Regardless, the two data structures must

be created. Finally, as noted in Line 3, a priority queue Q is created to contain all of the vertices that are considered unsettled which, at the beginning of the algorithm, is all of them. It must be dynamically resizable (i.e. a vector, a linked list, or a heap) or it could simply be a boolean array of size $|V|$, but the end result has to be a data structure that lets you 'remove' elements from it. For the implementation used in this thesis' experiments, a binary heap was chosen as it significantly increases the implementation's performance.

```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
 9      dist[source] ← 0
10
11      while Q is not empty:
12          u ← vertex in Q with min dist[u]
13
14          remove u from Q
15
16          for each neighbor v of u:
17              // Note: only v that are still in Q
18
19              alt ← dist[u] + length(u, v)
20              if alt < dist[v]:
21                  dist[v] ← alt
22                  prev[v] ← u
23
24      return dist[], prev[]
```

Figure 1. Sequential implementation of Dijkstra's algorithm [28]

Once the data structures are in order, as shown in Lines 5 through 8, each vertex has its distance value set to infinity and its predecessor value set to NULL. It is then added to Q. Finally, the distance value for the source vertex is set to zero.

On Line 11, the algorithm enters into a loop which contains the core of the algorithm with the loop continuing until all vertices in *Q* have been processed. At Line 12, the algorithm calls a function that returns the vertex in *Q* that has the smallest distance. This step is a crucial step insofar as the runtime of the algorithm is concerned and the choice of the data structure used for *Q* factors heavily into that calculation. Since Line 12 requires a search operation and Line 14 requires a delete operation, a data structure such as a minimum heap is optimal as the vertex with the smallest distance is always immediately available – no search is required – and the subsequent sort operation that takes place after the vertex is removed from *Q* ends with the next smallest vertex immediately available. Data structures such as vectors and linked lists require time-consuming searches and, implementation-dependent, potentially time-consuming delete operations.

Once a vertex, *u*, which has the shortest distance has been extracted from *Q*, Lines 16 through 22 show the guts of the algorithm. For each edge from vertex *u* to another vertex *v*, the first check is to see if *v* is in *Q*. This is because once a vertex has been analyzed, it is considered 'settled' and is not analyzed again. Assuming *v* is in *Q*, the edge (*u*, *v*) is relaxed. If this value is smaller than the distance of *v*, then *v*'s distance is updated with the smaller value. Furthermore, *v*'s predecessor is set to *u*. This process is repeated for all edges from *u* to a *v* that is in *Q* and then the entire process is repeated until *Q*, which holds the unsettled vertices, is empty, thus meaning that all of the vertices are settled and the graph's analysis is complete.

Runtime

Since the priority queue Q can be implemented in multiple ways, the algorithm's runtime is most easily given as the worst-case scenario and then the currently best known case scenario. For the worst case scenario, a basic array of size $|V|$ is used for the priority queue. Since it will have to search the array for the vertex with the smallest distance $|V|$ times – once per iteration of the algorithm – and each search will require it searching the entire array – $|V|$ elements – then that part of the runtime is $|V|^2$. Looking in the algorithm, it is also clear that every edge will be relaxed. The worst-case runtime scenario then is $O(|E| + |V|^2)$ which simplifies to $O(|V|^2)$. For the currently best known case scenario, Fibonacci heaps are used for the priority queue, thus making it a min priority queue. The runtime for that implementation simplifies down to $O(|E| + |V| \log |V|)$.

Parallel Processing Opportunities

Parallelizing Dijkstra is a rather challenging operation owing to the fact that while there are two loops in the algorithm, the distance value for a vertex is reliant on the distance value of the vertex's predecessor. Thus, while the inner loop that relaxes the vertex u's edges is trivially parallelizable, one cannot accurately calculate the distance values for other vertices until the values for the vertices directly connected to u have been calculated. Additionally, since the calculated values for the vertices directly connected to u will alter the contents of the set Q, one cannot accurately predict which vertex will be the next u until the calculations have been performed. As a result, the outer loop cannot be parallelized – neatly, that is.

While the outer loop cannot be parallelized in the nice and neat manner that the inner loop can be parallelized, that is not to say that it cannot be parallelized with a degree of brute force. If one were to process – to relax a vertex's edges and perform any necessary updates – every vertex in a graph simultaneously, then for the first iteration, where all distance values except the source vertex's are infinity, only the vertices directly connected to the source vertex would have their distance values updated. This is because of the simple fact that if two vertices are both set to infinity, even if the edge between them had a weight of zero, infinity is not less than infinity. For the second iteration, if one were to process all of the vertices simultaneously, there would be more updates as the vertices connected to the vertices updated in the first iteration update the vertices directly connected to them. It is at this point that problems can begin, for if there is an edge (u, z) and another edge (v, z), then it is possible for there to be two updates for z and that the value after the second update is not the minimum distance from z back to the source vertex. Also, particularly after multiple iterations, even if there are no problems correctly updating z, the new value could change the distance values for any other paths from z to the source vertex and since all of the vertices are being processed simultaneously, there is the question of how those changes could be affected.

It is at this point that the brute force method comes into play. Since the 'cascading results changes' described above could be computed on the next iteration, one could reason that if one iterated over the graph, processing all vertices simultaneously, then, eventually, all of the vertices will have their correct distance value. If one follows this line of reasoning, the graph's vertices could be divided into multiple parts, each of which is assigned to an individual process (it could be on the same CPU or on a different computing cluster node), and then after each

process has performed the Dijkstra algorithm on all of its vertices, a reduction function could be performed on the resulting distance and predecessor arrays, thus producing a new pair of arrays with the minimum distance values. Using the data in the new arrays, each process could repeat its calculations with the cycle continuing until none of the processes are making any updates. Such an approach works, however, as will be demonstrated in the experiments, it is not terribly time-efficient.

A much more time-efficient approach is to process all of the graph's vertices simultaneously by utilizing one or more GPUs and the hundreds and thousands of cores contained therein. As will be subsequently explained, a GPU core is not the same as a CPU core, however, in this application, processing a vertex – relaxing the edges and performing updates if so required – does not require much processing power. Since it is possible to analyze very large and dense graphs using only one GPU, the reduction step and associated delay to transfer data between the processes may very well be able to be removed, thus increasing the efficiency of this alternative parallel approach.

Bellman-Ford

Overview

Whereas Dijkstra's algorithm approaches the SSSP problem for a graph by processing each vertex one at a time and *once* – once a vertex has been deemed settled, its distance is never changed – the algorithm named after Richard Bellman [3] and Lester Ford [9] approaches the SSSP problem by taking all of the edges and repeatedly relaxing them, using the method of

increasingly correct approximation to achieve the final, correct answer. The benefit of this approach is that the algorithm is easy to implement. The negative side of this approach is that one ends up repeatedly relaxing edges that have been fully relaxed – that is, they will never be updated again – and thus much useless work is done. Even so, the Bellman-Ford algorithm can be of use. One commonly seen use is in the Routing Information Protocol (RIP) routing protocol [14] where a balance is struck between ease of programming and scale. The RIP protocol is normally only utilized in small scale networking situations – sometimes, only a single router is involved – which greatly reduces the impact of repeatedly relaxing edges that are already fully relaxed as there are not many edges. Furthermore, the algorithm is easily parallelized, thus offsetting, from a runtime point of view, some of the wasted work.

Explanation of the Sequential Algorithm

Implementing the Bellman-Ford algorithm in its traditional format is quite similar to how Dijkstra's is implemented insofar as data structures are concerned and how edges are relaxed. In Lines 3 through 6 of Figure 2, one sees an identical initialization of the distance and predecessor arrays and the source's distance value.

```
1    function BellmanFord(num_vertices, edges, source):
2
3        for each vertex v between 0 and (num_vertices - 1):
4            dist[v] ← INFINITY
5            prev[v] ← UNDEFINED
6        dist[source] ← 0
7
8        repeat |V|-1 times:
9            for each edge (u, v) with weight w in edges:
10               if (dist[u] + w) < dist[v]:
11                   dist[v] ← dist[u] + w
12                   pred[v] ← u
13
14       for each edge (u, v) with weight w in edges:
15           if (dist[u] + w) < dist[v]:
16               error "Graph contains a negative-weight cycle"
17
18       return dist[], pred[]
```

Figure 2. Basic implementation of Bellman-Ford algorithm [27]

At Line 8, one encounters the logic behind the algorithm. The algorithm is based on the premise that for a graph with $|V|$ vertices, if there is a path between any two vertices, it can have no more than $|V| - 1$ edges. Accordingly, it is reasoned that if all of the edges are relaxed $|V| - 1$ times, one will end up with a set of correct distance and predecessor values. Thus, on Line 8 there is the code of choice for a loop that will run $|V| - 1$ times. On Line 9, one sees the loop that relaxes every edge in the graph. The choice of data structure for 'edges' is entirely up to the programmer, the only constraint being that it should be one with very efficient access to the data. Lines 10 through 12 are the familiar code used for relaxing the edge and updating the dist and pred arrays as called for.

Lines 14 through 16 are optional, for if one knows that none of the edges have a negative weight, then it is impossible for there to be a negative distance value. However, since Bellman-Ford,

unlike Dijkstra, does allow for negative edge weights, then it is possible for a negative weight cycle to form, in which case, the path is invalid.

After the optional check on Lines 14 through 16, the algorithm has finished and the computed data is returned to the calling program in whatever manner the programmer wishes to do so.

In addition to the optional check on Lines 14 through 16, there is another option for this algorithm which should be carefully considered. While for simplicity's sake it is not illustrated, this option is based on the fact that it is highly likely that it will not take $|V| - 1$ repetitions to achieve a state in which all updates have been made. Accordingly, one could create a boolean variable called 'changed', for example, and set it to 'false' at the beginning of each iteration of the loop that begins at Line 8. Inside the inner loop (Line 9), where each edge is relaxed, one could add a simple test – if there is an update, set the variable to 'true'. After all edges have been relaxed, if the variable has the value of 'true', then that means that at least one vertex's dist and pred values were updated and if that happened, then that update might cause another update on a following pass. Thus, nothing is done and the loop starts again. If, however, all of the edges have been relaxed and no changes have been made – the variable 'changed' is still 'false' – then there is no need to continue relaxing the edges and one can break out of the loop begun on line 8. This does not guard against cycles, but particularly when one knows that there are no negative weights, utilizing this option can save a significant amount of time.

Runtime

The runtime for the Bellman-Ford algorithm is quite easy to calculate – O(|*E*| * |*V*|). This calculation is based on the fact there will be |*E*| calculations for each iteration of the loop and there will be (|*V*| – 1) – effectively, |*V*| – iterations of the loop. Unlike Dijkstra, there are no methods to reduce the runtime, save for the option that breaks out of the overall loop as soon as no more updates are being performed. However, as there is no guarantee that |*V*| – 1 iterations will not have to be performed, the worst-case runtime remains unchanged.

Parallel Processing Opportunities

While the outer loop of the Bellman-Ford algorithm cannot be parallelized *per se*, since it is, in truth, simply a counter for the number of times the inner loop has been executed and a limit to the number of times the inner loop can be executed, then it can be removed from consideration when parallelizing the algorithm. That leaves the inner loop, which is trivially parallelized. Care must be taken, however, to ensure data integrity for the *dist* and *pred* arrays, lest there be cases of a new, correct distance being accidentally overwritten with another new, but incorrect distance.

Floyd-Warshall

Overview

The Floyd-Warshall algorithm, proposed by Robert Floyd [8] and Stephen Warshall [25], is an algorithm for solving the All-Pairs Shortest Path problem. Its purpose is not just to find the shortest paths from *one* vertex to all of the other vertices, but to find the shortest path between

any two vertices, such that after the algorithm has processed a graph, one can name any two vertices and, should there be a path between them, the output of the algorithm will allow that path to be determined or *reconstructed*.

Explanation of the Sequential Algorithm

The fundamental theory behind the Floyd-Warshall algorithm is that for two vertices, *i* and *j*, there is either no path between them, an edge that directly connects them, or there is a vertex, *k*, such that (*i*, *k*) + (*k*, *j*) yields a path between the two vertices.

```
1  let dist be a |V| x |V| array of minimum distances initialized to ∞
2  let next be a |V| x |V| array of vertex indices initialized to null
3
4  function FloydWarshallWithPathReconstruction():
5
6      for each edge (u, v) do
7          dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
8          next[u][v] ← v
9
10     for each vertex v do
11         dist[v][v] ← 0
12         next[v][v] ← v
13
14     for k from 0 to (|V|-1) do // standard Floyd-Warshall implementation
15         for i from 0 to (|V|-1)
16             for j from 0 to (|V|-1)
17                 if dist[i][j] > dist[i][k] + dist[k][j] then
18                     dist[i][j] ← dist[i][k] + dist[k][j]
19                     next[i][j] ← next[i][k]
```

Figure 3. Basic implementation of Floyd-Warshall algorithm [29]

The algorithm begins with the creation of two matrices of size |*V*| x |*V*| – one to store the distance values and then one to store the next – or *successor* – vertex in the path. The use of matrices

makes this algorithm very memory intensive when compared to Dijkstra and Bellman-Ford, but is necessary for reasons that shall soon become apparent.

After the *dist* and *next* matrices are created and initialized to the familiar infinity and null values, albeit with the fact that there is no source vertex, on Lines 1-2 in Figure 3, the graph is read in on Lines 6-8. Line 7 has the weight for each edge (*u*, *v*) being stored in *dist*[*u*][*v*] and then *v* is stored in the corresponding location in the *next* matrix. Lines 10-12 further prep the matrices as the distance for each vertex to itself is set to 0 and the next vertex in the path from each vertex to itself is set to itself.

Line 14 is where the computational part of the algorithm begins and also contains the code that sets the value of *k*. In Lines 14 and 15, each cell of the matrix is visited. But it is at Line 16 that the algorithm sharply veers from what might be expected after having studied the Dijkstra and Bellman-Ford algorithms. Rather than to take the distance value of one vertex, add to it the weight of the edge connecting it to another vertex, and compare that sum with the distance value of the other vertex, the algorithm takes whatever distance value is at *dist*[*i*][*k*], adds to it whatever is at *dist*[*k*][*j*], and then compares it to the distance value for the current cell (*dist*[*i*][*j*]) in the *dist* matrix. The key difference that must be understood is that *dist*[*i*][*k*] is not a vertex; it is an edge – if it exists. One should recall that the *dist* matrix was initialized to infinity and then the edges of the graph were added to it, *dist*[*i*][*j*] representing not a vertex, but the weight of an edge between the vertices *i* and *j*. Thus, if no edge exists between two vertices, the corresponding cell in the *dist* matrix will remain at infinity. If there is indeed an edge between *i* and *k* and another between *k* and *j*, then if the sum of their distance values is less than that of *dist*[*i*][*j*], then

dist[*i*][*j*] is updated and a change is made in the *next* matrix (Lines 18-19). It should be noted that since there might not have been an edge at *dist*[*i*][*j*] – it may have been infinity – then after Line 18, there is now a path that connects the vertices *i* and *j* going through *k*.

Once the entire matrix has been examined with $k = 0$, the process then repeats with $k = 1, 2, 3\ldots(|V| - 1)$. Once it is complete, if there is a path between any two vertices, the *dist* matrix will provide the distance value and the *next* matrix can be utilized to construct the path between the two vertices.

Runtime

The runtime for the Floyd-Warshall algorithm, while not palatable, is easy to calculate. Given that each of the three loops has $|V|$ steps, then the runtime for the algorithm is $O(|V|^3)$.

Parallel Processing Opportunities

As is described by Weiss [26], due to how the value of *k* factors into the equation *dist*[*i*][*k*] + *dist*[*k*][*j*], one can divide the *dist* and *next* matrices into sets of rows which are then allocated to individual processes. The processes can then, working row-by-row and using MPI commands to transfer data, process the *dist* matrix. This approach keeps the implementation simple, but suffers due to the fact that before each row can be processed, it must be broadcast to all processes (which process does the broadcast is determined by the value of *k* and the row number). The amount of time necessary for these broadcasts will at least partially offset some of the performance gains from utilizing a parallel processing implementation. If one chooses to also compute the *next* matrix, the time delay will be doubled. One advantage Weiss' implementation

does have, however, is that since each process only receives part of the matrix (or matrices), then each process' memory requirements are anywhere from noticeably (two processes equals roughly half the memory needed per process) to drastically (eight nodes equals roughly one-eighth necessary per process) reduced. As a result, one has more flexibility in what hardware can be used.

While Weiss' implementation is targeted for CPU-based systems, if one has access to a GPU, then at least two additional parallel processing operations are possible. The first one [13], shall be called the naïve implementation, is a fairly basic implementation that actually, as shall be seen in the results of the experiments, works quite well. It works by accepting the fact that you cannot parallelize the outer *k* loop, but you can, however, parallelize the inner *i* and *j* loops. While the details will be spelled out in Chapter III, distilled down, it then proceeds to process the matrix row-by-row, seemingly just like the sequential algorithm, only with a slight twist – while the sequential algorithm is processed one cell at a time with one thread, the GPU version is processed one cell at a time, but there could easily be 32 threads executing at the same time – and thus 32 cells are being processed – and that is just for each of the multiple blocks in the grid.

Another approach to implementing the Floyd-Warshall algorithm utilizing a GPU is the 'blocked' (sometimes called 'tiled') approach. With this approach, the *dist* matrix is first divided into blocks based on the number of vertices. The blocks must be uniform in size and if the number of vertices does not allow for that, the *dist* matrix is padded with cells that will not have an effect on the computations. Using the aforementioned 16-vertex graph as an example, the result will be a four-by-four matrix, which shall be called *b_mtx*, each block being four vertices

high and four vertices wide. At this point, *k* refers to the blocks, not the vertices. What follows is that for the *b_mtx*, with *k* = 0, the block *b_mtx*[*k*][*k*] – or *b_mtx*[0][0] – is 'processed'. 'Processing' involves running the Floyd-Warshall algorithm on the actual *dist* matrix, but only for the cells that are in the block. This first block is called the *dependent* block. Next, the blocks on the *kth* row which are not the dependent block – these blocks are known as *semi-dependent* blocks – are processed. After that step – or before – the semi-dependent blocks in the *kth* column are processed. At this point, one can already utilize parallel processing for it does not matter in what order the semi-dependent blocks are processed, so long as they are all processed before the next step. The next step is one in which parallel processing can be heavily used, for all of the blocks which have yet to be processed, known as *independent* blocks, are processed. Once that is complete, *k* is incremented, the dependent block becomes *b_mtx*[1][1], and the process begins anew. This process can be seen in Figures 4 and 5.

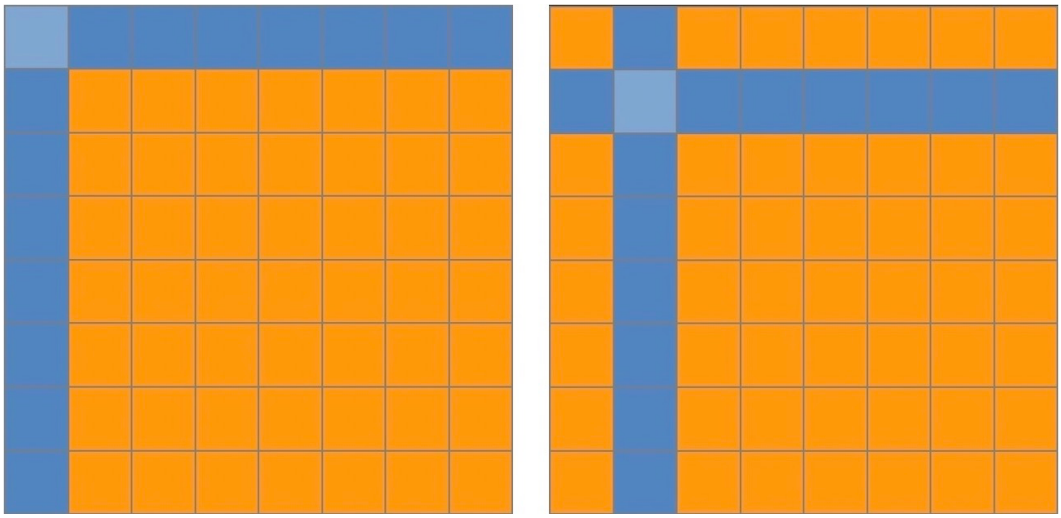

Figure 4. Blocked Floyd-Warshall matrix for *k* = 0 and *k* = 1 [11]

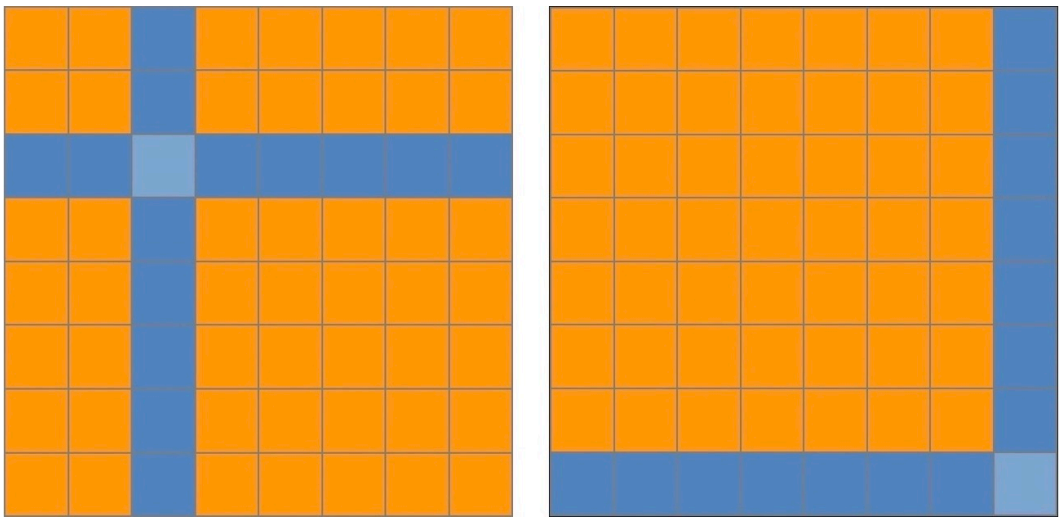

Figure 5. Blocked Floyd-Warshall matrix for $k = 2$ and $k = 7$ [11]

Implementing the blocked version of the Floyd-Warshall algorithm, while not exclusively the domain of GPUs, is best done on a GPU owing to the fact that each process will need a full and up-to-date copy of the *dist* and, if applicable, *next* matrices, thus requiring each process to need enough RAM to have complete copies of the matrices. Furthermore, after each step, the updated data for the blocks that were processed has to be broadcast to every other process and in a coherent manner. Depending upon the size of the graph, that amount of data could easily be measured in gigabytes when all of the blocks are accounted for. Keeping all of the processes on the same system, in the same memory space, and able to access the memory storing the matrices sidesteps the issue of delay due to broadcasts, but even very large many-core CPUs would have trouble outperforming – if it was possible – a GPU with hundreds and thousands of processing cores. That having been said, even tying together multiple GPUs' memory spaces with high-speed links, a many-core CPU can easily have more RAM, something which is a key factor when dealing with the memory-intensive matrices used by the Floyd-Warshall algorithm.

Hardware

Overview

Parallel processing can be implemented using only CPUs for computational processing, only GPUs for computational processing, or a mixture of both, the processors operating in a symbiotic manner with the GPU being classified as an *accelerator*. With the notation that GPUs can be added to any of the CPU-only processing options, the basic hardware options are described below. For GPUs, what will be described is how they are generally structured and function.

CPUs

Options

For systems for parallel processing that only utilize CPUs for computational processing, the options distill down to two basic choices. The first, a shared memory system, consists of all processing taking place on a single system. Be it one multi-core CPU or many multi-core CPUs and a few gigabytes of system RAM to over a terabyte of system RAM, everything stays in the same system and all processes share the same system memory space. The second option is a distributed system – a computing cluster – that consists of multiple individual systems or *nodes* that have been networked together. Each node can be a very basic system, a very advanced system, or anything in between.

Advantages/Disadvantages

The most fundamental issue when discussing the two options is that of communication and the speed thereof. For the shared memory option, where all of the processes share the same system memory space, information such as large arrays or lists of data can be shared quite easily, though if the appropriate control measures are not implemented, chaos can ensue. Looking at the distributed option, since the data directly accessible by processes is now limited to their individual memory spaces, if one process wishes to share some data with the other processes, the data must be transmitted over network links and received by the other processes. Not only does this take much longer than a simple memory access, but it adds to the amount and complexity of the code necessary for an application.

The shared memory option, while perhaps appearing superior, does come with its flaws, for while the good news is that all processes share the same system memory space, the bad news is that all processes share the same system memory space. When one is running a program on such a machine, one is having to share essentially all system resources with anything else that is running on the machine. Also, with a single system, there is no redundancy – if a component fails that renders the system inoperable, then one has become zero. Conversely, in a distributed system, not only is there redundancy by way of having multiple nodes, but one is able to manage the resource allocation far better. In a cluster environment, it is possible to specify that a given job run on certain subset of nodes and to reserve partial or entire nodes for a single user. Such control allows the cluster administrator to have multiple types of nodes – perhaps a general purpose node, a node with much more RAM than the general purpose nodes, and a node

equipped with a GPU – and serve the needs of a more diverse group of users without having to have multiple individual clusters. It also allows users to gain use of nodes where they know that no other user will be running a process that could interfere with their program, though there is still the variable of available network bandwidth between the nodes.

GPUs

History and Architecture

While usage of GPUs for non-graphics purposes was already taking place, when NVIDIA released their CUDA [17] API in 2006 along with GPUs that had such work in mind, it became much easier to perform such work. CUDA, which originally was an acronym for Compute Unified Device Architecture but now simply means CUDA, is an API that is primarily for the C/C++ and Fortran languages, though wrappers are available for others, and allows the programmer to work in a normal manner with necessary code for control of the GPU blending right in – nothing arcane to speak of. Other APIs, such as OpenCL [12] and ROCm [1], have been released, but CUDA appears to have the biggest user base and is widely seen in published research on GPUs. Insofar as hardware is concerned, NVIDIA also has the lead in published research. While AMD has secured contracts to provide both the CPUs and the discrete GPUs (as opposed to integrated GPUs on CPU chips) for two exascale supercomputers – Frontier at Oak Ridge National Laboratory [20] and El Capitan at Lawrence Livermore National Laboratory [24] – and Intel is providing both the CPUs and the discrete GPUs for Argonne National Laboratory's Argonne exascale supercomputer, Aurora [2], neither have the extensive development ecosystem

that NVIDIA offers. Consequently, this makes it much easier for researchers to utilize NVIDIA GPUs.

With regards to the architecture of GPUs, let it first be said that unless otherwise noted, this document will be referring to NVIDIA GPUs, for while the architecture on a macro scale is generally the same, when discussing details, the differences can be quite substantial. One universal difference, though, is that when one speaks of GPU 'cores', one is referring to something radically different than a CPU core, for a GPU core is little more than a basic ALU and much of what is individual for CPU cores is shared for GPU cores. This can be seen in Figure 6, which is a rough illustration of such from NVIDIA.

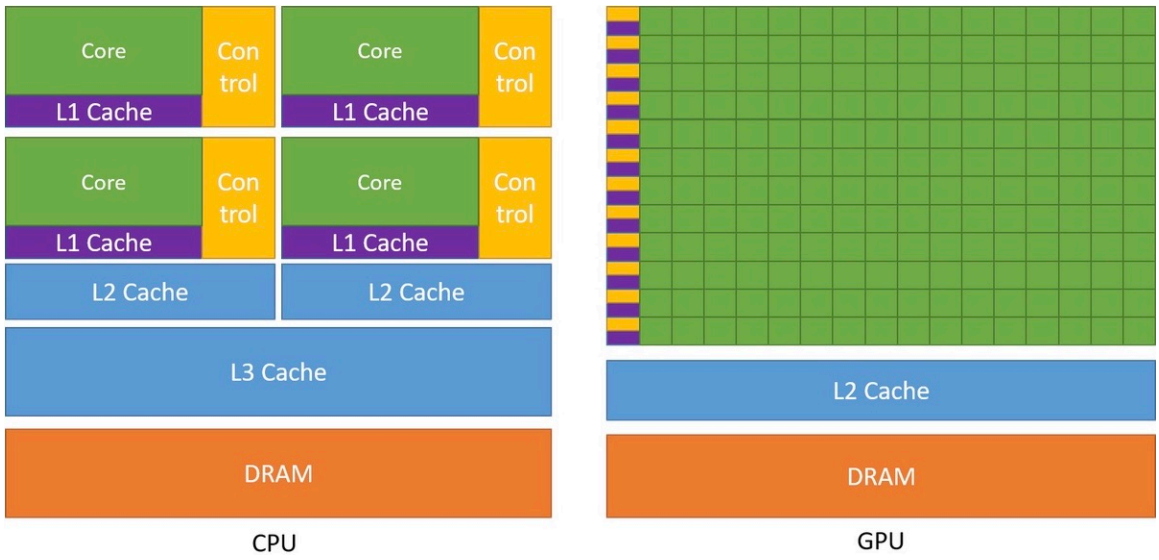

Figure 6. NVIDIA comparison of CPU and GPU architecture [19]

GPUs are built around units known as Streaming Multiprocessors (SMs). Depending upon the GPU model, there can be anywhere from a handful of SMs to literally dozens of SMs. For

example, the GA100 GPU has 84, though only 80 are available for use. The SMs share the GPU's L2 cache and are connected to the memory controllers that are used for accessing what is known as *global memory* – the GPU's version of system RAM. Inside each SM, one finds at least a number of cores – also known as 'CUDA cores' – load/store units, registers, L1 caches, shared memory, texture memory, and warp schedulers. SMs for the Volta, Turing, and Ampere microarchitectures also contain 'Tensor Cores' and/or ray-tracing cores, but they are specialty components and do not count as CUDA cores. The CUDA cores themselves are not uniform – most are designed for 32-bit floating-point math (FP32), but some are designed for 64-bit floating-point (FP64) or 32-bit integer (INT32) – and neither is the exact number of each type. The exact composition depends upon the GPU model, but for a given microarchitecture there are usually one or two models that have a much higher ratio of FP64 to FP32 cores that are targeted at HPC, data center, and scientific applications where higher precision is desired and then one or more models that are almost completely FP32 cores and are targeted at the consumer and gaming market. As of this writing, the exception is the Volta microarchitecture, which was only produced for HPC, data center, and scientific applications.

The memory model for a GPU is also variable, though most parts remain the same. In a GPU, starting from the innermost level, one has a set of registers in each SM that are shared by all of the processes or *threads* that are running on that SM. All threads also share the same L1 caches and what is known as *shared memory*, which is a small amount of memory that can only be used by threads in that SM. In some GPUs, the sizes of the caches and shared memory is fixed while for others there is a certain amount of memory that can then be allocated according to the user's

needs. *Texture memory* is a form of cache, providing fast access to data, but it is read-only to the threads in the SM. Outside of the SM, there is the shared L2 cache and then the global memory.

Programming

When writing a program that utilizes a GPU for some or all of the computational work, one must consider the matter with the manner in which a GPU operates in mind. The basic building block is just that – a block. In CUDA, for a *kernel*, which is what executes a function on a GPU, one has to supply two three-dimensional variables. The first is for the *grid*. CUDA allows you to create a one, two, or three-dimensional grid of blocks. Each block – the second variable – can also have up to three dimensions, only for blocks, one is specifying the number of processes or *threads* that each block in the grid should have. An example would be a two-dimensional block that has 16 threads on the x-axis and 16 threads on the y-axis. One could then create a two-dimensional grid with three blocks on the x-axis and three blocks on the y-axis and use the resulting structure for the analysis of a 48-by-48 pixel picture, perhaps. CUDA does place limits on how many threads a block can have and in what axis and there are also limits that affect the design of the grid. It is beyond the scope of this document to fully elaborate on the 'Why' behind grid and block design choices, but one key fact that should be kept in mind is that an SM has limits to the number of threads and blocks that it can host at any given time and in order to achieve maximum performance from a GPU, what is known as *maximum occupancy* of each SM is essential.

To understand the meaning of occupancy in the context of a GPU, one must understand how GPUs perform computational tasks. As has been noted, a GPU core is not much more than a

basic ALU which is why when writing functions for a GPU to execute, one needs to break the problem down into simple or fairly simple tasks. This may seem to create a lot more work, but that is offset by the fact that GPUs can easily perform thousands of operations per clock cycle. These operations are organized in 32-thread groups called *warps* and, normally, in each warp every thread executes the same function, though that is not to say that they execute the same operation. An example would be if the function has an if/else statement – when executing the function, the threads that test true for the 'if' condition will execute the associated instructions while the other threads in the warp do nothing, after which, the threads that test false will perform their associated instructions while the first group does nothing. Managing these warps are the *warp schedulers*. An SM can literally have dozens of warps, though how many are active depends upon the number of cores that the SM has. For example, if the SM has 128 cores, then it can have 4 active warps at any given time. As for the other warps, this is where the GPU is able to gain the increase in processing power that it can offer, for if a single thread in a warp has to make a memory access, the entire warp will stall and cannot proceed until the data requested has been returned to the requesting thread. By having multiple warps, when a warp stalls the warp scheduler can immediately replace it with another warp that is waiting to be executed, thus maximizing the usage of the SM's cores. To put it into perspective, for the Pascal-series GP104 chip, each SM has 128 FP32 cores, but can have up to 64 warps managing up to 2,048 threads spread out across up to 32 blocks, though the maximum number of threads per block is 1,024. Herein lies the explanation of occupancy, for if one created 32-thread blocks, then since an SM can have no more than 32 blocks at any given time, it would only be processing 1,024 threads at any given time when it is capable of 2,048. If each block was composed of 64 threads, then the SM is at maximum occupancy (when viewed from a thread count point of view). Also, more

warps insures that when an active warp stalls, there will be another warp ready for execution. There are other limiting factors that would preclude maximum occupancy from a thread-count point of view, such as if shared memory is used and each block uses an eighth of the available shared memory, only eight blocks can run at any given time, but the point remains – in general, to achieve the best possible performance, maximum occupancy must be kept in mind.

One other important fact that must be kept in mind is the fact that when a GPU executes a kernel, if the grid and block configuration call for 2,000 blocks with the threads per block being 1,024 – so two blocks per SM – then even if one were utilizing the latest GPU – the Ampere-series A100 – with its 108 operational SMs, one could only process 216 blocks at a given time. CUDA deals with this issue by creating a queue for the blocks, filling the SMs to capacity and when one block finishes processing, another replaces it. Consequently, if a specific order is required for processing, then multiple kernels may be necessary or the processing should be performed on the CPU.

CHAPTER III

COMPUTATIONAL METHODS

Overview

As the parallel processing implementations being tested had to be capable of running in a distributed or cluster environment, the API chosen for inter-process communication was that of MPI. Due to how MPI handles inter-process communication, the same code can be run on a cluster with one head node and seven compute nodes or a single eight-core processor without making any changes to the code. This makes development easier as one does not need access to a cluster to do at least basic testing and development.

One advantage that MPI has is that, depending upon the configuration settings, it can see a compute node with a quad-core processor as a compute node capable of one process, two, three, or four processes. Consequently, if one wishes to have up to 16 compute processes or *ranks*, a single compute node with 16 cores can provide the necessary ranks. Alternately, four quad-core nodes could be used or one could choose to use 16 nodes and only expose a single core on each one for processing. This flexibility allows one to maximize available hardware and, since MPI also allows one to pick and choose which nodes to use, does not require that the cluster's nodes be homogenous. Naturally, there are some caveats, the first of which being that placing all 16 ranks on a single node means that the latency for inter-process communication will be minimized, yet, at the same time, it also means that all ranks have to share the node's resources – something that could be a problem for memory-intensive applications. Moving to four quad-core nodes may alleviate resource issues, but now the latency for inter-process communications will

vary depending upon whether the process is on the same node or the message must go out over the network. For programs that are time-sensitive, this could also be a problem. Transitioning to 16 single-core – or at least single-*rank* – nodes fairly nulls out the timing variable and guarantees that each rank now has the entire node's available resources at its disposal, but at the expense of there now being 16 compute nodes to purchase, store, power, and maintain.

CPU Implementations

Dijkstra

Parallelizing Dijkstra's algorithm for CPU-only processing was accomplished by implementing Dijkstra's algorithm sequentially, using a binary heap as a minimum priority queue in order to reduce the runtime. The code was then altered to have a single array of structs, each of which held a distance value and a predecessor, instead of having two separate arrays for the distances and the predecessors. The reasoning for the change is because an entry for a given vertex in the *dist* and *pred* arrays could be considered a pair, for the path that ends with that vertex with that distance value has the specified predecessor vertex. If a shorter path is found, unless the predecessor vertex is updated when the distance value is updated, one will not be able to recreate the path. Given that an MPI reduction operation will need to be performed based on the distance value, by creating a struct that contains the two values and writing a custom MPI operation that compares array entries based solely on the distance value, it is possible to perform the reduction and retain the distance/predecessor pairing.

Once the actual algorithm was complete and adapted for MPI usage, the resulting program is one that reads in the graph, distributes it to all of the nodes, and then, including itself, as the master rank is also a compute rank, it parses the graph's vertices into as evenly as is possible segments and then sends those assignments to the nodes. What follows is something that is reminiscent of the Bellman-Ford algorithm's approach of increasingly correct approximation, in that each node processes the vertices assigned to it as if they were the only vertices in the graph. Once all have done so, a reduction operation on the distance/predecessor array is performed that updates the array with the minimum computed values. Thinking about how Dijkstra operates, it stands to reason that the node assigned with the vertices $0 – n/p$, where n is the number of vertices and p is the number of processes or ranks, will have the correct answers after the first iteration and then for all subsequent operations, those vertices will never be updated. As for the rest of the vertices, much like Bellman-Ford, with each iteration, more and more of the vertices will become settled until each rank, upon running Dijkstra on its vertices, never updates a single vertex. At this point, when a 'changed' variable from every rank reports 'false', the loop in which the ranks have been running will terminate, leaving the master rank with an array that holds all of the correct distances and predecessors.

The above approach works well, though if one is comparing the results from the MPI version to the results from the purely sequential version, one may find that while all distances may match, not all predecessors may match. This is due to the fact that for binary heaps, if there are two vertices with the same distance, since the heap can only pick one, it may not pick the same vertex as the purely sequential version would. That having been said, so long as the distance is correct

and the path, if traced out, is also correct – its distances add up to the stated path distance – then the algorithm has succeeded.

Bellman-Ford

The Bellman-Ford CPU-based implementation was approached and implemented in essentially the same manner as Dijkstra. The key potential difference is that for Dijkstra, each rank had to have the entire graph in addition to the entire distance/predecessor array. For Bellman-Ford, while the entire graph is passed to each of ranks in the implementation used for this thesis' experiments, one could parse up the graph and only send out a subset of edges to each node. From a performance point of view, the difference should not be noticeable, though, as will be shown in Chapter V, it can play a significant role insofar as the resources needed and/or used by each rank.

Floyd-Warshall

The CPU-based distributed implementation of the Floyd-Warshall algorithm used for the experiments tracks very closely with the implementation presented by Weiss [26]. Unlike Weiss, however, for the *dist* array, rather than using a signed integer (4 bytes), an unsigned char (1 byte) was used. The reasoning was that with the edge weights being no more than 10, it was quite unlikely that there would be a distance value greater than 255. Given that the data is stored in a square matrix, then a 400-vertex graph would have a 400-by-400 matrix that used 160,000 bytes of RAM, plus a bit more in overhead. Had an integer been used, 640,000 bytes of RAM would have been required. Given that the smallest test graph has 5,000 vertices, one can easily imagine the significance of the memory savings.

Also unlike Weiss' implementation, since, from a programming point of view, it was quite easy to add in the capability of computing the *next* matrix – Weiss only computes the *dist* matrix – that capability was added. That capability does come with a price, however, as an unsigned small int (2 bytes) will only go as high as 65,535. This means that if one wishes to allow the implementation the ability to process graphs with more vertices, then use of integers is required for the *next* matrix, for the *dist* matrix stores distances, but the *next* matrix stores vertices. Using a 5,000-vertex graph as an example, with five nodes, each is responsible for 1,000 rows or a matrix with 5,000,000 cells for the *dist* matrix. With one-byte variables, that is roughly five megabytes. Adding a *next* matrix with its four-byte variables adds 20 megabytes to the memory requirements, thus increasing the memory requirements by five times! Furthermore, when each row is broadcast, previously, with 5,000 vertices, the broadcast would have been 5,000 bytes (plus overhead). Adding the *next* matrix meant that for each row, a total of 25,000 bytes (plus overhead) had to be transmitted each time.

In addition to adding support for the *next* matrix, two other departures from Weiss are that in the revised implementation, each process/rank is explicitly told by the master rank which rows of the matrix belong to it rather than the processes/ranks determining the assignments for themselves. Also, rather than one rank reading the graph in and distributing the data which is then, post-processing, collected by another rank, the master rank, in addition to being a compute rank, does the necessary distribution and collection. In addition to simplifying matters, this has the bonus effect of only one rank needing to reside on a machine which has enough RAM to hold the entire matrices.

APSP-via-SSSP

Implementation of APSP-via-SSSP is, on the surface, rather straightforward, as all one is doing is executing the SSSP algorithm of choice on the graph once for every vertex. Looking deeper, there is the question of how it is be implemented. For a shared memory system, with an eight-core CPU, for example, one could set aside one core for system processes and use the other seven cores to run an SSSP implementation $|V|$ times where $|V|$ is the number of vertices. Alternately, one could run multiple instances of a sequential implementation, each processing the graph with a different source vertex. The same question applies for distributed systems. For this thesis, there was no effort to answer the question, for after reviewing initial runtime data for CPU-based implementations and GPU-based implementations, only GPU-based implementations were studied – the GPU-based implementations were that much faster.

GPU Implementations

Dijkstra

Implementing Dijkstra's algorithm on a GPU is, on the one hand, not that difficult – just create a one-dimensional grid with one-dimensional blocks of either 512 or 1,024 threads, assign each thread a vertex to process and a boolean variable to signify a change, and then much like the MPI CPU-based version, process each vertex until no updates are being made. On the other hand, from an efficiency viewpoint, it is not the most efficient method with the first reason being that each thread is having to do quite a bit of work – indeed, depending upon the density of the graph, a *lot* of work – and while that doesn't automatically make it inefficient, one has to remember that the thread isn't being executed on a *CPU* core and therefore the resources available to it are far

less. More to the point, as has been noted about GPUs, if/else statements can be very inefficient, for if just one thread in a warp tests ‘true’, then the other thirty-one threads have to wait until the first thread has completed its instructions, at which point, if there is an ‘else’, the roles are reversed. Unfortunately, while there has been much research on implementing Dijkstra on GPUs [10, 15, 21, 23], not much headway has been made in getting around a problem – the inefficiencies from if/else statements – that, most likely, simply cannot be solved.

For the experiments in this thesis, the implementation by Harish and Narayanan [10], which shall be referred to as H&N, was tested as was an implementation which is a simpler form of H&N. In H&N, the algorithm is implemented by creating a *mask* – a boolean array sized to the number of vertices – that indicates the settled status for each vertex, and then, inside a while loop that is contingent on a boolean control variable, performing the relaxation and update steps in two distinct stages. In the first stage, if the vertex’s mask indicates that it is not settled, the edges are relaxed, the update values, if any, are stored in an array designated for such data, and the vertex’s settled mask is set to settled. In the second stage, if there are any updates associated with the vertex, they are performed, the settled mask is toggled so that it is now unsettled, and the overall loop control variable is set so that when it is copied from the GPU back to the CPU, the while loop will continue. This process is repeated until no updates are performed.

In the second implementation, which shall be called the Basic implementation, rather than to have a second stage, any updates are performed immediately – there is no intermediary stage. Not having a second stage reduces the runtime inside the loop, for there is not the overhead involved with a second kernel launch and then the execution of tasks that could be done in the

first kernel. However, since the execution model of both implementations has the vertices processed repeatedly until there are no updates, any time saved by the lack of a second stage in the loop is only time saved if the implementation requires the same number of iterations or the time needed for additional iterations is less than what would be needed for the second stage.

Both implementations, insofar as the CUDA grid and block structure are concerned, operate in the manner described at the beginning of the preceding paragraph. Unfortunately, only a single-GPU implementation could be tested despite a significant amount of work being put into getting a multi-GPU implementation operational. The Gunrock [23] library has an implementation which is supposed to be multi-GPU, but time restrictions precluded that avenue of inquiry from being investigated.

Bellman-Ford

The Bellman-Ford algorithm and GPUs go hand-in-hand, for the easily parallelized inner loop keeps the kernel simple and the single if statement reduces the amount of time 'lost' when the threads that tested 'false' must do nothing. A boolean variable can be used to keep track of whether or not any more changes are being made, thus reducing the outer loop to simply setting the 'changed' variable to 'true', copying it to the GPU, executing the kernel, copying the 'changed' variable back to the CPU, and if it is true *and* the loop has run less than $|V| - 1$ times, loop again. As for the grid and block structure, like with the Dijkstra implementations, it is a simple one-dimensional grid of one-dimensional blocks.

Unlike the Dijkstra implementations, it was possible to adapt the CPU-based MPI implementation of the Bellman-Ford algorithm for use on a GPU, thus allowing experiments with up to 4 GPUs. More GPUs *could* have been used, but as will be seen in the analysis of the results, there would have been little reason to do so. It should be noted that for experiments that compared a single-GPU Bellman-Ford implementation with Dijkstra implementations, the Bellman-Ford code used was that used for the MPI experiments, only with the MPI-associated code removed so that it would not have any unnecessary functions or overhead to slow it down. As a result, one should not compare the single-GPU results to the one GPU results for the MPI version.

Floyd-Warshall

Given the $O(|V|^3)$ runtime for the Floyd-Warshall algorithm, a single GPU makes operations that would have taken hours or days to complete on a CPU possible to accomplish, returning results in a matter of minutes or perhaps a few hours. However, as with most things, there is a catch – since the Floyd-Warshall algorithm is quite memory intensive, owing to its use of a $|V| \times |V|$ matrix to store the data, then even if the distance matrix used 8-bit unsigned integers, thus allowing for distances of 0 through 255, over 9.3 GB of RAM would be required for a graph with 100,000 vertices. If one wished to have a *next* matrix so as to be able to generate the individual paths, over 37.2 GB of RAM for the *next* matrix alone would be required! Clearly, unless one has a rather large budget and can afford either the latest NVIDIA Ampere-series GPU, which has 80 GB of RAM, or a number of smaller units, each with a healthy amount of RAM, linked together so as to combine their RAM, then the size of graphs, measured by the number of vertices, capable of being processed is limited. While the higher-end consumer cards are

beginning to ship with 10-12 GB of RAM, as of this writing, the far more common consumer cards have 8 GB of RAM and, keeping in mind that they have sufficient processing power for formal research, the decision was made to restrict the graphs analyzed using the Floyd-Warshall to those which could fit both the distance matrix, the *next* matrix, and any additional data inside 8 GB.

As described in Chapter II, implementing the Floyd-Warshall algorithm on a GPU can be done rather easily as the outermost loop – the *k* loop – supplies the value of *k* for the two nested inner loops which then process each cell of the graph, working row-by-row. Programming this naïve implementation, however, isn't as simple and requires some thought. One approach would be to create a two-dimensional grid of blocks that each represent one cell in the matrix. This approach is viable, though only up to a point, for while the *x*-dimension in a CUDA grid can have $(2^{31} - 1)$ blocks, the *y*-dimension is limited to 65,535 blocks. If one is only computing the *dist* matrix and has the necessary RAM, then the grid could not be big enough for the graph in question. One could counter with the argument that blocks can have up to 1,024 threads, total, and that they can all be in the *y*-dimension, which means that a grid could support 65,535 x 1,024 or 67,107,840 cells or, in this application, vertices. In theory, that could work, however, to do so would require throwing away a key feature of a GPU – the ability to execute multiple threads simultaneously, as each block would only be processing one cell at a time, thus limiting the amount of cells that could be processed at a single time to the number of SMs times the number of blocks.

An alternative approach would be to utilize the *y*-dimension of each block, but rather than having 1,024 threads in the *y*-dimension, configure the block as a 32x32 thread block. Not only does this

make it easier to visualize the thread's place in the overall matrix, but there should also be a performance increase owing to the GPU's ability to perform what is known as *coalesced* memory access. That is, if the matrix is stored as a one-dimensional array, when the first block is executed, all 32 threads will request the value for their cell. Since all of their cells are contiguous, the memory locations being accessed will be one continuous piece of memory and, rather than to make 32 memory calls and transfers, the GPU will only have to make one call and transfer. Examining the expression *dist*[*i*][*j*] > *dist*[*i*][*k*] + dist[*k*][*j*], one can see that coalesced memory access should be possible for all three referenced cells in the matrix. It should be noted that the reason for the conditional 'should' for coalesced memory access is that such access is dependent upon the memory being accessed being contiguous – something that it may not be. That having been said, if the 32 threads are broken up into three chunks, three smaller coalesced memory calls and transfers are better than 32, which is what would happen with 1,024 thread high blocks.

For programming the 'blocked' approach, as described in Chapter II, two matrices are required for the *dist* matrix – one for the actual *dist* matrix and another, *b_mtx*, to represent the blocks, which, so as to reduce confusion, will be called 'tiles' for this explanation. In Chapter II's description of GPUs, it referred to *shared memory*, which is memory that is local only to the SM. Furthermore, any data stored in it by a thread in block A can only be seen by the other threads in block A – those in B, C, or D cannot see it. All are drawing on the same amount of memory, but not only is that memory physically local to the SM, but it is effectively local to each block.

For the tiled Floyd-Warshall approach, if each tile is 32x32, the first tile – *b_mtx*[0][0] – represents *dist*[0-31][0-31]. Recalling that the Floyd-Warshall algorithm will be run on just those cells, one option is to have each thread request data from global memory – memory which is not part of the actual GPU. For this algorithm, at least three calls and transfers per thread will be necessary and coalesced memory access may not be possible. Alternately, utilizing shared memory, each thread could request the data for its position in the matrix – one call and transfer per thread – and then store the data in the matrix that has been created in shared memory. Since all threads in the block can see everything in that matrix, when the threads go to execute the Floyd-Warshall algorithm on the matrix, all memory calls will be to memory that is part of the SM, thus yielding a very fast access time and performance. Once all processing is done, each thread then transfers the data for its cell back to global memory.

For Step Two, when the column and row tiles are computed, shared memory is used again, only for this step, a copy of the tile's data is stored in shared memory as well as a copy of the data for the dependent tile as values from both tiles are used in the computations. That fact is also why the row and column tiles can be computed in any order, as the only data each tile changes is that of the cells in its tile and the only tile that it depends upon is the dependent tile, which has already been calculated.

To call Step Three the computation of the independent tiles is not entirely accurate, for, again, shared memory is in use, but there are now three matrices in the shared memory for the data from the *dist* matrix. The independent tiles use data from the semi-dependent tile in the column they are in, the semi-dependent tile from the row they are in, and then their own data for the

computations. This is an example of shared memory being of particular use as it greatly reduces the number of calculations that are required for the threads to determine where they are in the grid and where the data from the other tiles is and reduces the number of calls to global memory.

The tiled Floyd-Warshall approach does come with some caveats, the largest one being that it is quite complex to implement and troubleshooting the code is very difficult, for while the end result is the same as the naïve GPU approach and the CPU approach, unless the algorithm has completed the last *k* loop, the values for the tiled approach will not match up with what the other two approaches would yield. Additionally, part of using shared memory is that one frequently has to stop and wait, an example of which is that when the function is executed on the GPU, it accesses its data from global memory, stores it in shared memory, and then it must stop and wait, for it cannot proceed until all of the other threads have loaded their values into the shared memory. This may be a small price to pay for the shorter access time that shared memory affords, but not only is it still there, if one doesn't remember about the need for synchronization, problems can result. Finally, while the tiled approach does easily allow for a *next* matrix, particularly if a *next* matrix is being computed, one must keep track of shared memory usage, for with each step, more shared memory is utilized.

For the experiments conducted for this thesis, all of the Floyd-Warshall experiments were conducted using only one GPU. The algorithm has been successfully implemented in a multi-GPU, multi-node environment [6], but that is beyond the scope of this thesis.

APSP-via-SSSP

Implementing APSP via SSSP was done in a straightforward manner. For each of the GPU-based SSSP implementations, code was added around the actual implementation that ran in a loop $|V|$ times where $|V|$ is the number of vertices. Beginning with the source vertex set to 0, the loop incremented the source vertex by one after each iteration until SSSP had been performed on the graph $|V|$ times, each vertex having been a source vertex for one of the iterations.

Insofar as timing the implementations, for each iteration, a timestamp was captured before the *dist/pred* array was initialized and any control values were set and copied to the GPU's memory. After the graph had been processed, another timestamp was captured, the difference computed, and then the difference was added to the variable representing the total runtime. One could argue that the timestamps should have been taken outside the loop that iterated over the vertices – the difference would have been very small, but it would exist – but one thing that was not done was to save the results. The reason behind that decision is there are at least two different options for saving the results. First, one could run SSSP on the graph with a given source vertex and then immediately write the results data to disk, thus causing the true runtime to be the amount of time to process the graph and the amount of time necessary to write the result data to disk. Alternately, one could create an array of *dist/pred* arrays before the outer loop, shift which *dist/pred* array received the results data each time, and then after the SSSP runs are complete, write the results data to disk, in which case, the runtime would be the runtime necessary for the actual SSSP runs. Due to the time disparity of the two options, the decision was made to measure

the actual runtime only and leave the issue of how and when to save the results data up to the user.

CHAPTER IV

EXPERIMENTS PERFORMED

Resources Used

Overview

Due to the cost of the equipment necessary for the experiments performed, the decision was made to utilize the resources offered by Amazon Web Services' Elastic Cloud Computing (AWS EC2) service for the experiments performed for this thesis. Taking this route allowed much flexibility in choosing amongst the variety of hardware configurations. This was particularly important when one considers that the hardware that was used for the actual experiments – and thus the hourly cost to use it – costs considerably more than the hardware that is sufficient for simple testing. Thus, for example, a virtual cluster could be 'launched' utilizing inexpensive *instances* – AWS' term for virtual machine – and since the OS and all software utilized can be or is the same (see Table 1), all of the necessary scripts can be written and properly tested such that when the virtual cluster used for the actual experiments is launched, no time is wasted.

Table 1. Operating System and Software Used

Operating System (all instances):	Ubuntu 18.04.5 LTS (GNU/Linux 5.4.0-1029-aws x86_64)
Instances (CPU):	g++ 7.5.0, gcc 7.5.0, OpenMPI 2.1.0/2.1.1, SLURM 20.02.4
Instances (GPU):	g++ 7.5.0, gcc 7.5.0, OpenMPI 4.0.5, SLURM 20.02.4 CUDA 11.0, NVIDIA Driver Version 450.80.02

CPU Resources

AWS offers a plethora of options when it comes to selecting the hardware platform on which one's virtual server or cluster is going to run. The majority are simply 'variations on a theme' – that is, the same CPU is utilized with the difference between options being the number of available threads and RAM. Other variables are local storage options and the speed of the network links. For the purposes of this thesis, two types of instances were needed for work that was going to be primarily CPU-driven.

The first was an instance that could be used to generate the graph sets used for testing. For this task, the t2.small and t2.large instances were chosen as the hourly cost is very low and the low thread count (one and two, respectively) was not an issue, as the instance was only being used for one task. Additionally, with the exception of the largest graphs, for which the t2.large's 8 GB of RAM was needed, the t2.small's 2 GB of RAM was sufficient. Table 2 offers some additional details about the t2 instances.

Table 2. AWS EC2 t2 Instance Data

Instance	Processor	Threads	RAM (GB)	Network Performance
t2.small	Intel Xeon 3.3 GHz	1	2	Low to Moderate
t2.large	Intel Xeon 3.0 GHz	2	8	Low to Moderate
Note: These are burstable instances, so CPU speed is 'up to...'				

The second type of instance necessary is that used for the virtual clusters. For the testing and preparation phase, virtual clusters were created from t2.large instances, which have more than enough computational power for the necessary tasks, so long as the graphs analyzed are not that

large. For the instance to be used for the clusters on which the experiments would be performed, the decision was not quite that simple, as AWS' options can often seem to be just the same. For the purposes of this thesis, first priority went to network link speed, followed by the fact that the desired CPU manufacturer was Intel (as opposed to AMD or AWS' custom ARM-based CPUs) and then at least 4 *cores* (so 8 threads) and 16 GB of RAM were in order. In the end, the chosen instance was the c5n.2xlarge, which is of the 'Compute Optimized' instance family and which has high-speed network links and, relatively speaking, a higher RAM to core ratio than the 'General Purpose' instances have. Also used for certain non-cluster experiments were larger versions of that instance, as is detailed in Table 3.

Table 3. AWS EC2 c5n Instance Data

Instance	Processor	# of cores	CPU speed (min/turbo)	RAM (GB)	Network Bandwidth (Gbps)
c5n.2xlarge	Intel Xeon Platinum 8124M	4	3.0/3.4 GHz	21	<= 25
c5n.9xlarge	Intel Xeon Platinum 8124M	18	3.0/3.4 GHz	96	50
c5n.18large	Intel Xeon Platinum 8124M	36	3.0/3.4 GHz	192	100
Usage:	MPI-CPU Dijkstra/B-F/F-W (cluster)		(17) c5n.2xlarge		
	MPI-CPU Dijkstra/B-F/F-W (Single-Node)		(1) c5n.9xlarge (Dijk/B-F most graphs; F-W all graphs)		
			(1) c5n.18xlarge (Dijk/B-F largest graphs)		
Notes:	Intel Xeon Platinum 8124M processors normally ran at 3.4 GHz according to /proc/cpuinfo				
	Intel Xeon Platinum 8124M has 18 cores, so the c5n.9xlarge is one entire CPU				
	and the c5n.18xlarge uses 2 CPUs (unknown if the 2 CPUs are on the same motherboard)				

GPU Resources

For instances for the GPU-driven experiments, the options were much easier to choose from. While AWS does have some GPU instances that are primarily meant for visualization purposes, for computing purposes, it has three types – p2 instances, which are older K80 (Kepler-class)

GPUs, p3 instances, which are V100s (Volta-class), and then it offers one p4 instance, which contains eight A100 (Ampere-class) GPUs. While p2 instances were briefly used for testing, all experiments were conducted using p3.2xlarge or p3.8xlarge instances (see Table 4).

Table 4. AWS EC2 p3 Instance Data

Instance	Processor	# of cores	CPU speed (min/turbo)	RAM (GB)	Network Bandwidth (Gbps)
p3.2xlarge	Intel Xeon E5-2686 v4 (Broadwell)	4	2.3/2.7 GHz	61	<= 10
	NVIDIA 1x V100			16	
p3.8xlarge	Intel Xeon E5-2686 v4 (Broadwell)	16	2.3/2.7 GHz	244	10
	NVIDIA 4x V100			64	
Usage:	MPI-GPU Bellman-Ford (cluster)		(5) p3.2xlarge		
	MPI-GPU Bellman-Ford (Single-Node)		(1) p3.8xlarge		
	All single-GPU SSSP & APSP		(1) p3.2xlarge		

Graph Sets

Approaching this project, it was decided to have 9 different sets of test graphs, varying both the size and the density of the graphs. In terms of constraints, it was decided to limit the graph sizes such that for any given algorithm, all necessary data could fit into the 8 GB of RAM that is commonly found on a consumer-grade GPU card, as such cards are typically powerful enough such that fairly serious scientific research can be performed using them. The exception to the plan was the Floyd-Warshall algorithm. Due to how memory-intensive it is, in order to remain inside the planned RAM allocation, it would only utilize the ‘small’ graph sets along with additional sets that had larger vertex counts, but did not reach the ‘medium’ graph set size.

Deciding on the sizes of the graph sets was largely influenced by the fact that processing a 5,000-vertices graph with a GPU, utilizing a typical Dijkstra implementation, means that one will only

have around 5,000 threads ‘in flight’ – a situation which will illustrate how a GPU can behave in such a situation, but is also not a realistic scenario for evaluating a GPU. It is, however, a starting point, for, keeping the RAM limitations in mind, it allowed the graph sets to be different by one order of magnitude each. The ‘small’ sets consist of 11 graphs, ranging from 5,000 vertices to 10,000 vertices, the graphs sized in 500-vertex increments. The ‘medium’ graph sets range from 50,000 to 100,000 vertices in 5,000-vertex increments, with the ‘large’ graph sets being one order of magnitude larger. So as to accommodate the APSP experiments, an additional three sets of graphs, varying in density, were created, ranging from 12,500 to 40,000 vertices with the size increments being 2,500 vertices.

Closely factored into the choice of graph set sizes was the question of a graph’s density. As the graphs are undirected graphs, then the graph’s density $D = (2 * |E|) / (|V| * (|V| - 1))$. Since the goal was to have graph sets that ranged from small to large in terms of vertices and sparse to dense in terms of density, that meant that ‘sparse’, ‘medium’, and ‘dense’ would need to be defined – something far easier said than done as the desire was for the graphs to be realistic. After much examination of many of the publicly available graphs, the densities shown in Table 5 were selected. Ideally, the densities would be the same for all three sets of graph sizes, however, as noted, memory restrictions were in play and the densities decided upon were deemed to be the best compromise.

Table 5. Graph Density Values

Graph Density Values		
# of vertices	|V| <= 100,000	|V| > 100,000
Sparse	0.002000	0.000040
Medium	0.010000	0.000200
Dense	0.050000	0.002000

Once the size and density of the graphs had been determined, random undirected graphs were generated by a utility program which had been written expressly for that purpose and were saved in the desired file format. It should be noted that when calculating the density of a graph with a given number of vertices, the degree of each vertex was varied until the desired density target had been met. Due to the program's algorithm for creating the random graphs, not all of the vertices in each graph are of the desired degree – nearly all are, but some are of a degree slightly smaller than the desired degree. From a practical point of view, that result is of no consequence for the purposes of this thesis, but from a technical point of view, it should be noted. Additional notes about the random graph generator program is that the edge weight for each edge was a randomly chosen integer between 1 and 10 and that the algorithm made sure that there were no self-loops.

SSSP Experiments

Overview

The experiments utilizing CPU-based implementations were performed utilizing the system configurations shown in Table 3.

For those performed on the ‘MPI-CPU (Cluster)’ and ‘MPI-CPU (Single-Node)’, the process was as follows: For each graph, with node configured to only be 1 rank, process it using 1 node and record the timing data. Then process it using 2 nodes and record the timing data. Repeat until the graph had been processed with 16 nodes. On the ‘MPI-CPU (Single-Node)’ system, as all computations were on a single machine, exchange ‘node’ with ‘rank’.

Dijkstra

Two sets of experiments were conducted using the MPI implementation of the CPU-based Dijkstra algorithm as described in Chapter 2. The primary purpose for the experiments performed on the MPI-CPU cluster was to analyze how the implementation performed on the 9 sets of graphs. The primary purpose for the experiments performed on the MPI-CPU Single-Node instance was to compare the implementation’s performance when run on one single machine, using CPU cores as nodes, against its performance in a cluster environment. For naming purposes, they will be referred to as ‘MPI-CPU (Cluster)’ and ‘MPI-CPU (Single-Node)’, respectively.

For both the SSSP GPU-based Dijkstra experiments, both of the two available implementations were tested on an instance with a single V100 GPU, the purpose being three-fold – to analyze their performance on the graph sets, to compare them against each other, and, eventually, compare them against the Bellman-Ford implementation.

Bellman-Ford

The CPU-based experiments performed utilizing the MPI implementation of the Bellman-Ford algorithm as described in Chapter 2 had the same goals as those run for the Dijkstra algorithm.

As a GPU-based MPI implementation of the Bellman-Ford algorithm had been successfully developed, it was possible to perform experiments in the same vein as the CPU-based MPI experiments, both as a cluster and single-node. One critical fact that should be noted is that for the single-node experiments where all four GPUs were in the same machine, while they were connected via NVLink [18], that connection was not being used – for the purposes of the experiment, they were four separate GPUs. Also tested was the Bellman-Ford GPU implementation on an instance with a single V100.

APSP Experiments

Overview

The experiments utilizing CPU-based implementations were performed utilizing the system configuration used for the SSSP experiments, as shown in Table 3.

As for the GPU-based experiments, all were performed using an instance with a single V100 GPU and the software detailed in Table 4.

Floyd-Warshall

As with the Dijkstra and Bellman-Ford SSSP experiments, the MPI-CPU Cluster and the MPI-CPU Single-Node was used for the experiments with the MPI implementation of the Floyd-Warshall experiments. Due to runtime reasons, the CPU experiments were only performed on the 5,000 (or 5K) through 10K set of sparse graphs. The reason for not performing the experiments on the medium and dense sets was that since the Floyd-Warshall algorithm will perform the same number of computations regardless of the density of the graph, there was no justifiable reason for repeating the experiments on graph sets that had the same number of vertices as the first set.

For the GPU experiments, as the greatly reduced runtime made it possible, in addition to the 5K to 10K sparse graph set, the additional graphs reaching up to 40K were tested. Also, the medium and dense versions of those graph sizes were tested out of curiosity. Two sets of experiments were performed – the first utilized the very basic GPU implementation of the Floyd-Warshall algorithm described in Chapters II and III, while the second utilized the 'blocked' approach that has also been detailed. As with the two Dijkstra GPU-based implementations, the goal was both to analyze the individual implementations' performances and to also compare the two.

APSP-via-SSSP

All three SSSP GPU implementations were used to perform APSP-via-SSSP on the 5K to 10K sparse, medium, and dense graph sets. As the greatly reduced runtimes – in comparison to the

CPU implementations – allowed for it, APSP-via-SSSP was also performed on the ‘APSP graph sets’ that ranged from 12.5K to 40K. Testing was done utilizing a single V100 GPU instance.

CHAPTER V

ANALYSIS OF RESULTS

Overview

One of the primary purposes of the experiments conducted was to analyze how the various algorithms and implementations thereof perform on graphs of different sizes (as measured in vertices) and different densities. When examining the results, two of the foremost questions were: holding the densities the same, what happens as the number of vertices increases; and holding the number of vertices the same, what happens as the graphs become more dense? Furthermore, for the implementations utilizing MPI, the speed-up as additional nodes were added was examined. Since it is often the case that using multiple nodes can actually return a *longer* runtime than just using one node, owing to MPI operations and network-related overhead, there was the question of, how many additional nodes are necessary for the runtime to return to the runtime for a single node? Also, how many additional nodes are necessary to produce a runtime that is half of that of a single node? Phrased more succinctly, is it worth using the implementation in a distributed manner? The same questions were also asked when the MPI implementation was used on a single node, the individual cores of the CPU acting as nodes, which meant that the variable of network communications was removed, though the overhead incurred by the MPI operations was still present. Runtime was, naturally, measured and compared, but pure speed was not the only attribute being examined.

SSSP Experiments

Dijkstra (CPU)

Overview

Put simply, the CPU-based implementation of the Dijkstra algorithm, adapted for a distributed environment performed quite poorly from a time perspective, in comparison with the Bellman-Ford algorithm, and yet it also performed very well, insofar as when one begins to increase the number of nodes from one node, the runtime immediately decreases – as one would expect. Also, as one would expect, there is a point at which using additional nodes yields a change in the runtime that is such that the use of additional nodes is not worth the extra resources from a practical perspective – better known as the point of diminishing returns.

Cluster

Examining the results of running the CPU-based implementation on a cluster, we find that the runtime results for the sparse 5K-10K graph set (Figure 7), when plotted on a two-dimensional chart, are what one would hope for. That is, for the 5K graph, adding a second compute node cuts the runtime in half and adding a third node cuts the runtime to a third of that of a single node. In this particular case, adding the additional nodes cut the runtime to less than a half and a third, respectively. When a fourth node is added, the runtime is cut to just under a quarter of the one-node runtime. Beginning with the addition of a fifth node, the expected finally happens – namely, the increase in the time necessary for the various ranks to communicate with one another has increased to the point of offsetting the performance gains from parallel processing. As a

result, from the fifth node to the sixteenth node, the runtime plateaus, dropping down slightly before eventually beginning to increase back to the four-node runtime. Also as expected, as the number of vertices in a graph increased, the number of additional nodes before the runtime plateaued increased with eight nodes being the maximum number of nodes before all of the graphs in the graph set plateaued. Of note is the fact that as the number of vertices in the graph increased, while it took longer for the runtime to plateau, the reduction in runtime was greatest for the first four nodes – beginning with the fifth node, if the graph had not already plateaued, the descent to the plateau was much more gentle.

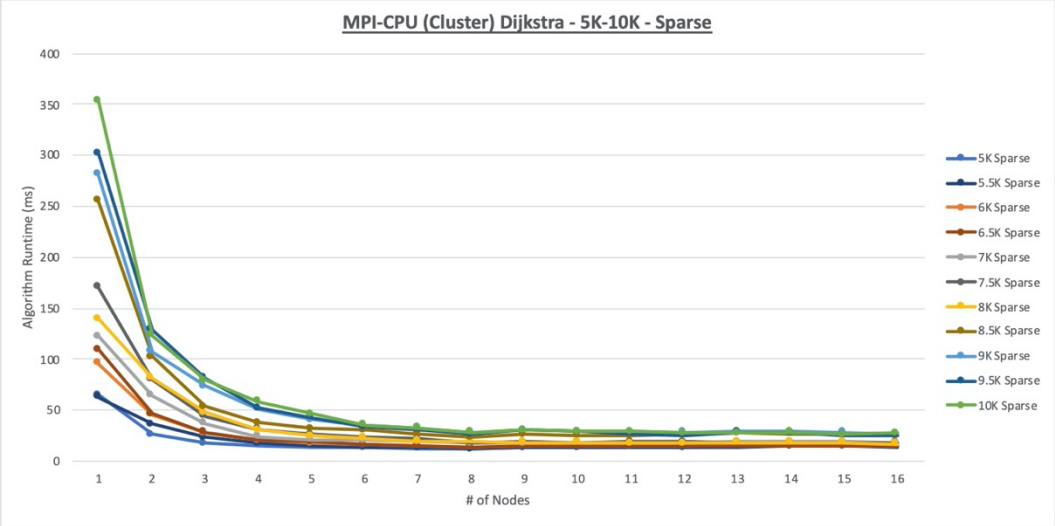

Figure 7. MPI-CPU (Cluster) Dijkstra 5K-10K Sparse Runtime Chart

Looking at the denser 5K-10K graph sets, in the medium density set (Figure 8), the initial runtimes are slightly higher, though the highest one-node runtime is actually less than its value for the sparse set, but it is what happens when the additional compute nodes are added that is significant. Rather than a very sharp drop beginning with two nodes, while there is a noticeable

drop with the addition of a second node, it is nowhere near the degree found in the sparse set. It takes six nodes before the 5K graph plateaus with the other graphs taking a much more gentle approach into the plateau that most reach beginning with the ninth node.

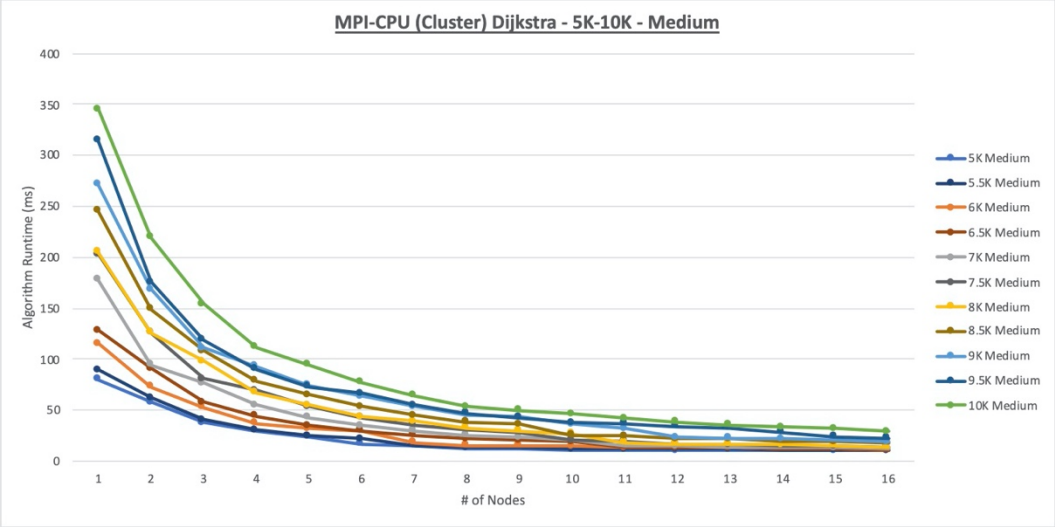

Figure 8. MPI-CPU (Cluster) Dijkstra 5K-10K Medium Runtime Chart

The dense 5K-10K graph set (Figure 9) is a radical departure from the previous two sets. Runtimes are increased by up to approximately twice that of the medium set, though, curiously, all graphs continue to either plateau, or begin to, beginning with the ninth node. The data from the experiment is a bit chaotic – something that could possibly be attributed to network congestion or, alternately, perhaps there is something unique about that particular graph set that caused such behavior – but it is possible to discern that, just like the transition from the sparse set to the medium set, while there is still a sharp drop when a second node is added, it takes longer before the graphs begin to plateau, though, again, most graphs have plateaued beginning with the ninth node.

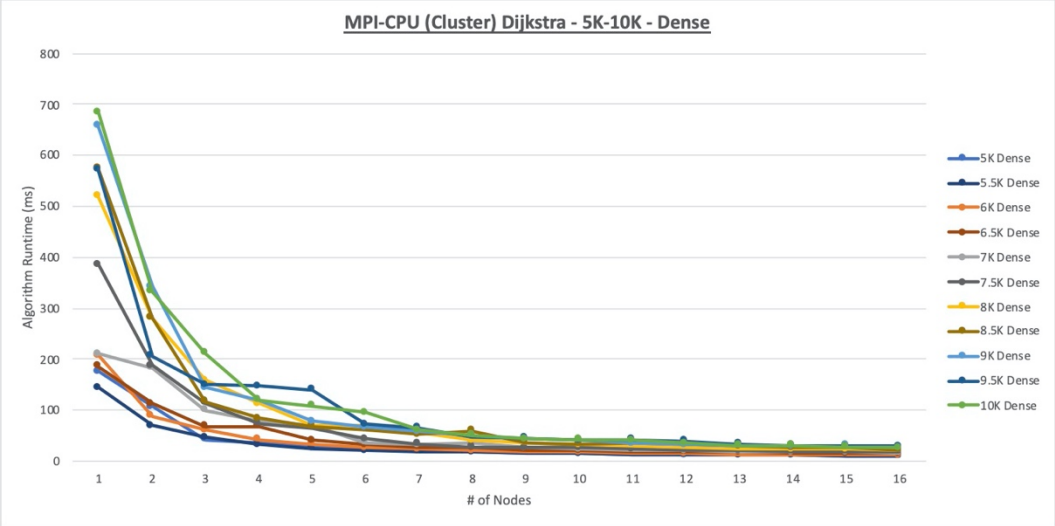

Figure 9. MPI-CPU (Cluster) Dijkstra 5K-10K Dense Runtime Chart

Summarizing, for the 5K-10K graph sets, as the density is increased, it takes an increasing number of compute nodes before MPI communications and operations begin to offset the performance gains from the additional nodes – something that is quite logical. What is not logical is that once that critical point has been reached, one would think that the runtimes would begin to increase rather than to hold steady. Since the runtimes are holding steady, the reasoned explanation is that the additional nodes are contributing to the performance of the implementation, only their positive contribution is offset essentially completely by their negative contribution. Also of note is that, at least for the 5K-10K graph sets, for some reason, eight compute nodes seems to be the point at which maximum reasonable gain is attained. There may be additional performance increases for some of the graphs, but, generalizing, anything after eight compute nodes falls in the category of ‘the point of diminishing returns’.

Examining the 50K-100K and 500K-1M graph sets, the behavior of the graphs going from sparse to dense is similar, though, understandably, it takes more compute nodes for the graphs to plateau. Such behavior is to be expected – that is to say, reasoned and logical, as the task is the same, only there is more data. What was unexpected are the results of looking at the implementation from the perspective of increasing the number of vertices. For each graph set, the number of vertices increased by an order of magnitude. However, as one moved from the 5K-10K graph set to the 500K-1M graph set, while the number of vertices were increasing by an order of magnitude, the runtimes often increased by an order of two magnitudes! (see Figures 8, 10, and 11) Strictly speaking, this increase does not hold true for all graphs, but the runtime increase is significantly greater than a single order of magnitude.

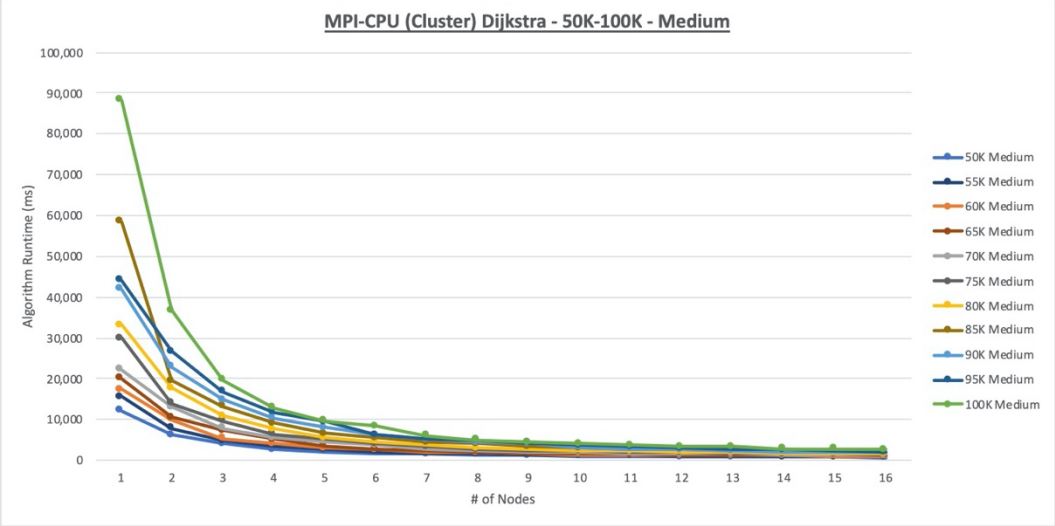

Figure 10. MPI-CPU (Cluster) Dijkstra 50K-100K Medium Runtime Chart

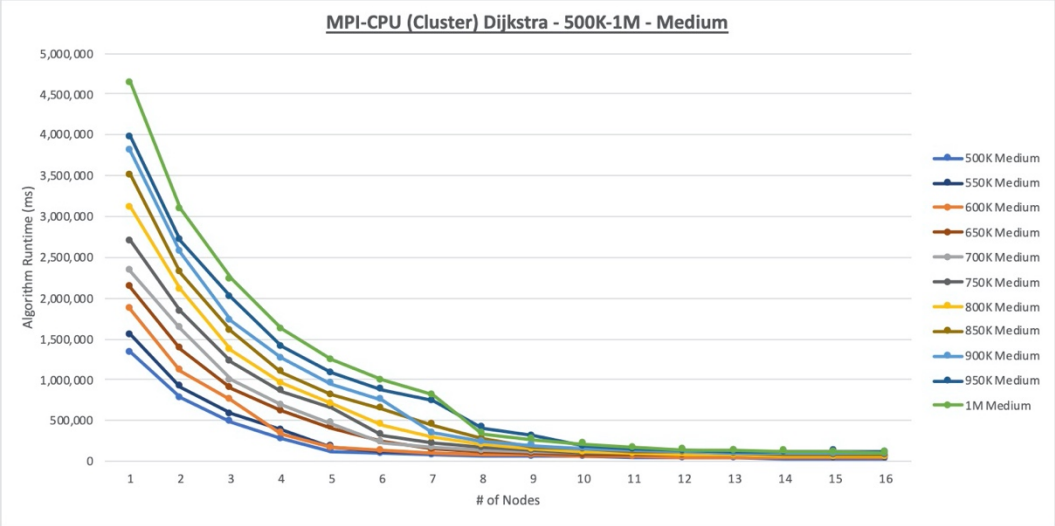

Figure 11. MPI-CPU (Cluster) Dijkstra 500K-1M Medium Runtime Chart

Single-Node

At first glance, the results of the experiment run on a single node using almost all of its 18 cores is the same as those of the experiment that was run on the cluster, as can be seen in the comparison of Figure 8 against Figure 12.

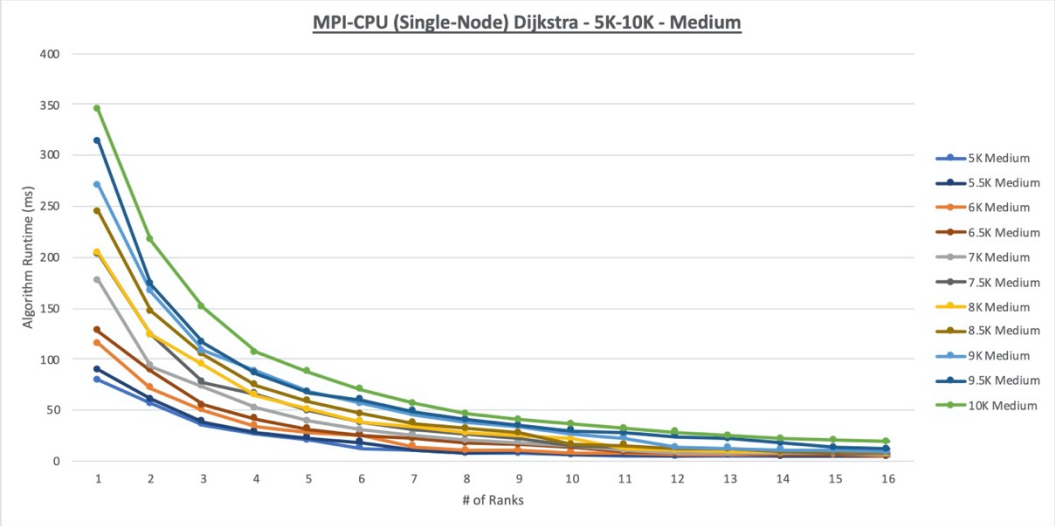

Figure 12. MPI-CPU (Single-Node) Dijkstra 5K-10K Medium Runtime Chart

A closer examination, however, shows that while the speed-up for the medium and large graph sets was approximately the same (see Appendices B and C for the speed-up charts), the speed-up results for the 5K-10K graph sets were quite different. *Speed-up* is defined as the relationship between the time taken by one node and the time taken by *n* nodes. For example, if the time taken by one node is two seconds and the time taken by two nodes is one second, then the speedup is 2/1 = 2. If the time taken for one node is six seconds and the time taken for two nodes is four seconds, then the speedup is 6/4 = 1.5. If there are *n* nodes and the speedup is *n*, then the speed-up is considered *linear*. If there are *n* nodes and the speedup is less than *n*, then the speed-up would be considered *sub-linear*. If there are *n* nodes and the speedup is greater than *n*, then the speed-up would be considered *super-linear*. In the case of the experiment performed on the cluster, for the 5K-10K graph set, the results (Figure 13) were either linear and turned sub-linear after the fourth node or, for the graphs with more vertices, the results were super-linear up until a point at which they soon became sub-linear. For the single-node experiment, the

results (Figure 14) were almost super-linear from the start, their speed-up only increasing as additional nodes were added. This effect was increasingly pronounced as the graphs gained more vertices. The graphs with the smallest number of vertices did, after the eighth node, begin to trend towards linear speed-up – indeed, the 5K and 5.5K graphs briefly became sub-linear with fifteen nodes – but, on the whole, the speed-up was super-linear for all of the graphs. For the cluster experiment, the opposite was true – rather than become super-linear, the speed-up for the graphs became sub-linear, though the graphs with the higher vertices count were the closest to a linear speed-up.

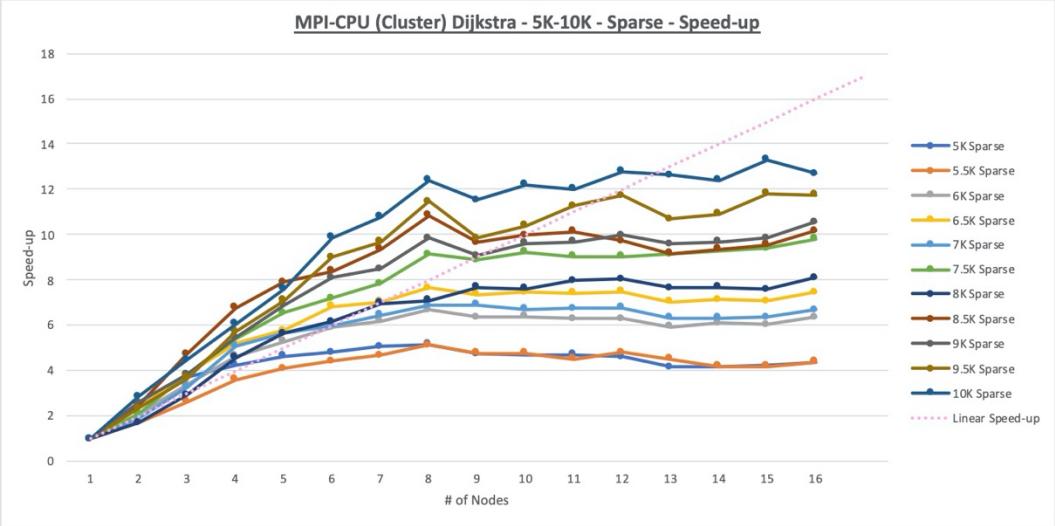

Figure 13. MPI-CPU (Cluster) Dijkstra 5K-10K Sparse Speed-up

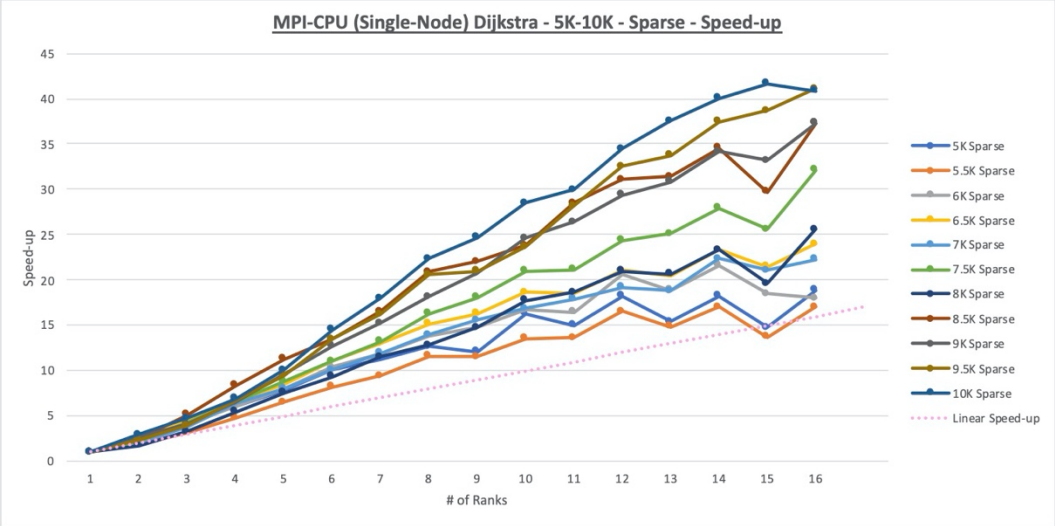

Figure 14. MPI-CPU (Single-Node) Dijkstra 5K-10K Sparse Speed-up

Moving to the medium density 5K-10K graph set, the super and sub-linear speed-up pattern holds true, though the difference between the speed-up of the highest and lowest vertex count graphs narrows. For the dense 5K-10K graph set (Figures 15 and 16), the speed-up plots for the cluster and single node experiments are very close, the cluster results finally becoming super-linear for most of the graphs. This pattern holds true for the remaining graph sets with the speed-up results for the cluster and Single-Node results being both very close and also, in the vast majority of the cases, super-linear.

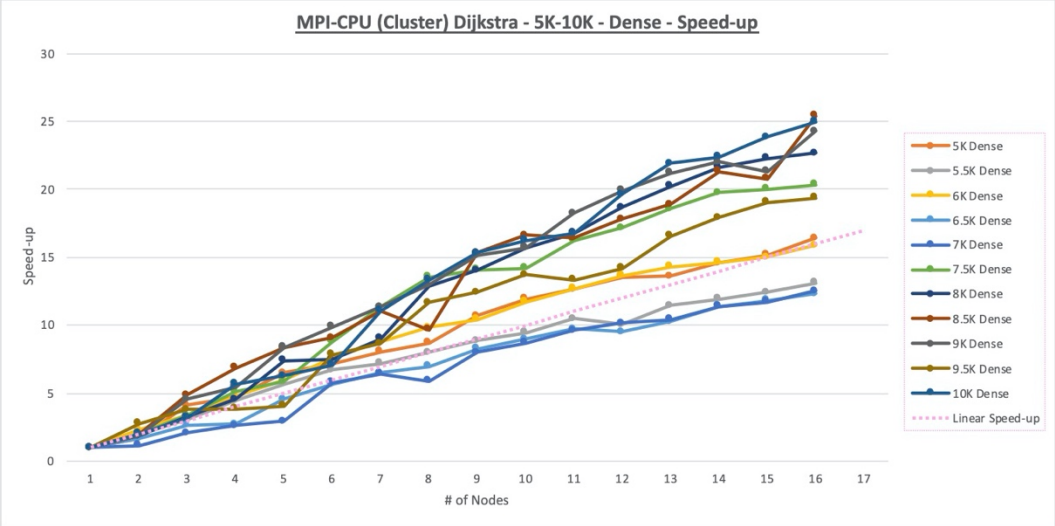

Figure 15. MPI-CPU (Cluster) Dijkstra 5K-10K Dense Speed-up

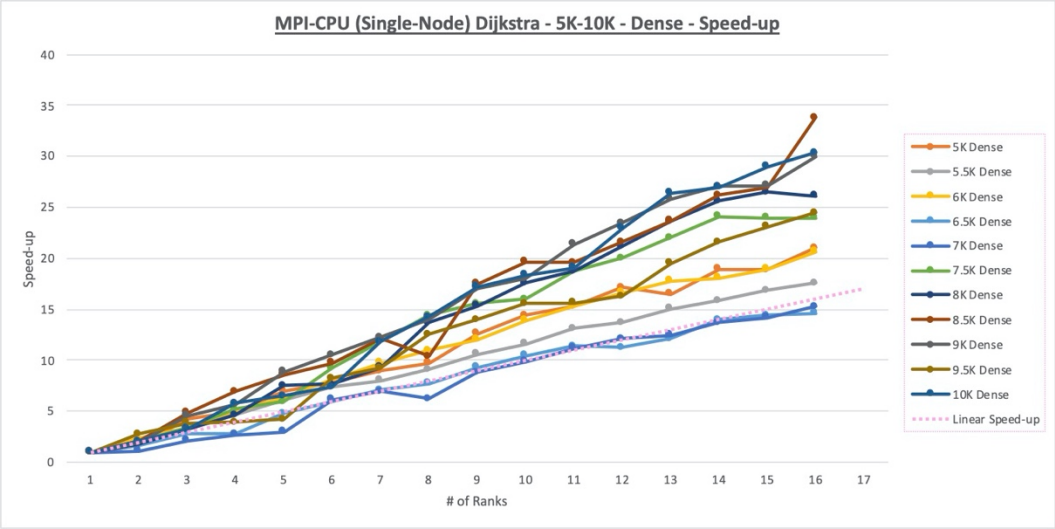

Figure 16. MPI-CPU (Single-Node) Dijkstra 5K-10K Dense Speed-up

The disparity between the sparse and medium density 5K-10K graph sets and the other graph sets can be explained by pointing to the fact that the Single-Node experiments did not need to use network links for inter-process communication. Accordingly, up until a certain point, what the results show is the consequences of having to utilize network links, for as the amount of data

being transferred between the various processes increased, so too did the speed-up values move towards being the same or very similar. Once a critical mass of sorts was reached, the two experiments were mostly the same – the Single-Node results were often a little bit faster than the cluster results. Such a difference is understandable as, again, communication over network links was not needed.

Comparison

Since the cluster experiment was not repeated with slower network links – the links for the AWS instances were at least 5 Gbps as measured by iperf3 [7] – then, comparing the two experiments, it is difficult to make a definitive statement about the effect of network communications with regards to performance. However, the CPU-based parallel implementation tested relies heavily on computational horsepower – in this case, the CPU, the RAM, and the connecting bus – and inter-process communication for the MPI reduction operations. Given that the CPU, RAM, and connecting bus were the same for both experiments, then that leaves inter-process communication as the main variable. Given that the results of the two experiments were so close, one can theorize that the hardware was capable of moving data at a speed such that performance was not hindered. Had it been possible to perform the experiments on the same hardware but with the network links limited to 1 Gbps, then that hypothesis could be confirmed or refuted.

Another factor in trying to explain the very close results is the fact that for the cluster experiment, while it is not possible to determine the exact amount of L2 and L3 cache each node had access to, what is known is that each node ‘owned’ four out of 18 cores on its CPU, so possibly between 20% and 25% of the processor’s L2 and L3 caches. Each node had this amount

of cache space regardless of how many nodes were currently in use and how much data it had to process. On the single node system, while the system had access to all 18 cores and all of the L2 and L3 caches, the amount of cache space each 'node' had access to varied by the amount of 'nodes' that were in use. Thus, while the inter-process communication speed was probably increased compared to the cluster experiment, the amount of available cache could have affected the performance such that the positives and negatives for each experiment mostly balanced out.

Dijkstra (GPU)

Analysis and Comparison

As was explained in Chapter III, both of the two implementations of Dijkstra – Harish & Narayanan (H&N) and the simplified form of H&N (Basic) – are closely related with the key difference being when the update part of the algorithm is performed. For the Basic implementation, it is done immediately following an edge relaxation, while H&N's implementation stores the value for an update in an intermediate array and then performs the update in a second kernel call. After initial testing during the development of the Basic implementation, it was observed that the Basic ran faster than H&N, but since the development test graphs were primarily small, sparse graphs, it was unknown how the Basic would perform with larger and denser graphs.

Studying the runtime results (see Figure 17), one finds that for the 5K-10K graph set, the Basic returned an almost flat result for the sparse graphs – from the 5K through 10K graphs, the runtime hovers right around 0.40 milliseconds. The medium density graphs start around 0.40

milliseconds and then slowly rise, but never do they reach 0.60 milliseconds. Looking at the results for those same graph sets for H&N, one finds the sparse results to create a line that is also mostly flat, though it is between 0.70 and 0.80 milliseconds – nearly twice the Basic's runtime. The results for the medium density graphs are actually faster than the sparse results, save for the 10K graph, but the medium density results are still well above the Basic's results. The analysis thus far indicates what one might expect – the additional step in H&N is slowing it down as, otherwise, the implementations are close in performance. The dense 5K-10K runs, however, show that H&N's second step, while using up time for the sparse and medium graph sets, is actually worth something. The Basic starts at around 0.95 milliseconds for the 5K dense graph and slopes upwards to just under 1.60 milliseconds for the 10K dense graph. The H&N's runtimes, however, begin at a bit over 0.80 milliseconds and gradually increase to a bit over 1.30 milliseconds.

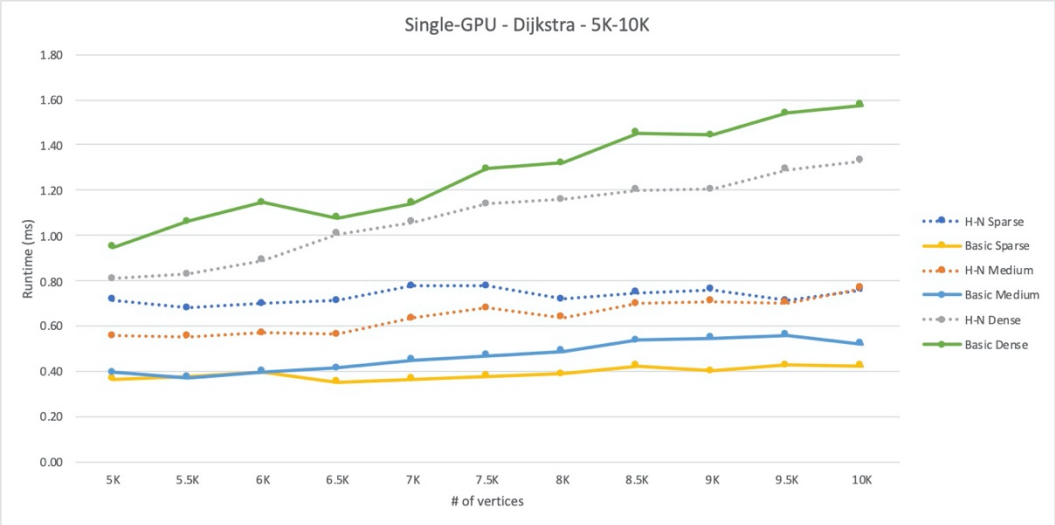

Figure 17. Single-GPU – Dijkstra – 5K-10K

One might consider the results for the 5K-10K dense graph sets to be a bit of a fluke, however, in the 50K-100K graph sets, the H&N returns faster runtimes for all three densities (see Figure 18). Of note is the fact that as the graph becomes more dense, the two implementations become farther and farther apart. The sparse graphs have runtimes that are fairly close, though the H&N is clearly the faster of the two. Moving to the medium graphs, the distance apart is much more pronounced, however, it is nothing compared to the results for the dense graph sets. The Basic started at a bit over 50 milliseconds for the 50K dense graph and ended at just over 400 milliseconds for the 100K dense graph. The H&N begin at just over 20 milliseconds for the 50K dense graph and finished at just over 140 milliseconds for the 100K dense graph.

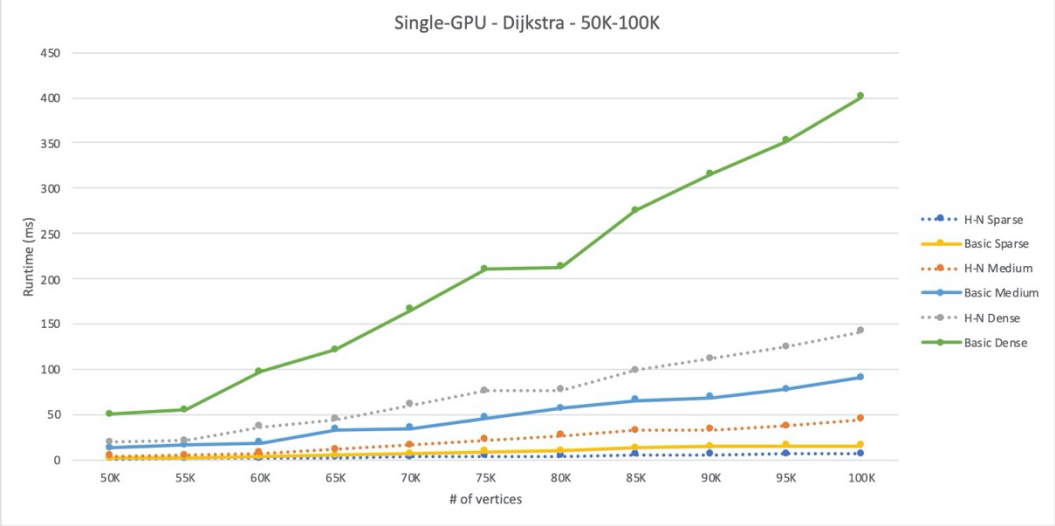

Figure 18. Single-GPU – Dijkstra – 50K-100K

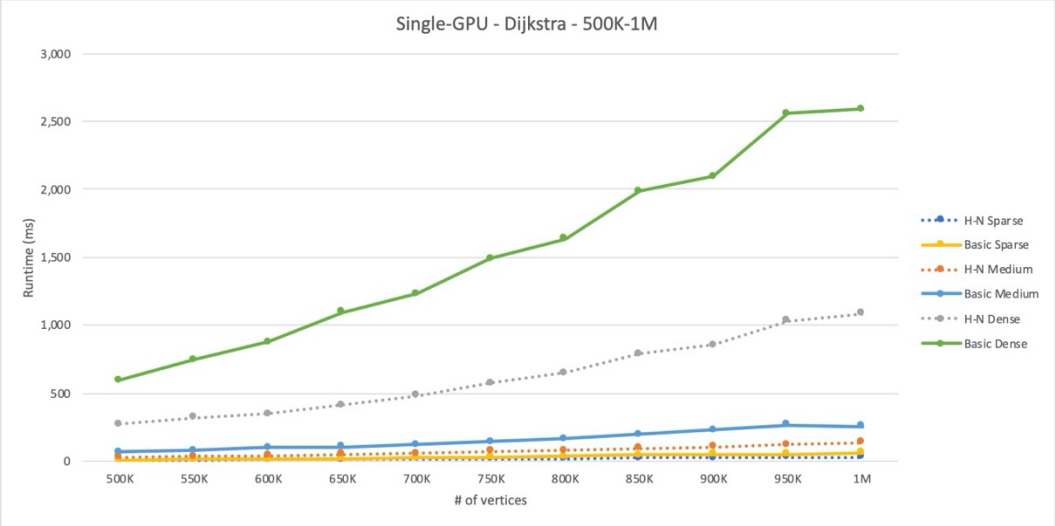

Figure 19. Single-GPU – Dijkstra – 500K-1M

As the 500K-1M graph sets reveal the same behavior (see Figure 19), it can be posited that by separating out the relaxation and update steps, H&N are able to cause the graph to 'converge' – for all of the vertices to become settled – with fewer iterations than is necessary for the Basic implementation. This theory matches with the results from the 5K-10K experiments, the runtime differences for the sparse and medium graphs being more indicative of overhead for H&N adding to the runtime, but once a certain point was reached, the Basic was having to perform many, many more iterations. Thus, for small, sparse-to-medium density graphs, the Basic implementation is faster, but only because the inefficiencies aren't enough to overcome the additional overhead in H&N. Beyond that, H&N is the implementation of choice. It should be noted that for both implementations, both the sparse and medium density graph sets all turned in runtimes that plotted out to pretty smooth lines. The dense graph results were fairly smooth, as well, so while the Basic may have performed poorly on the dense graphs, it and the H&N both

returned consistent results, though in both cases, the dense graph timings were markedly higher than the others.

Comparing the GPU runtimes to those of the MPI-CPU (Cluster) runtimes, the Basic implementation took less than 0.40 milliseconds to run Dijkstra on the 5K sparse graph. The MPI-CPU version took ~65 milliseconds with one node and slightly under 15 milliseconds with 16 nodes. Moving to the other extreme, for the 1M dense graph, the H&N implementation took ~1,084 milliseconds, while the MPI-CPU version took ~114.5 minutes with one node and 4.67 minutes with 16 nodes. Clearly, the GPU has a bit of a performance edge.

Bellman-Ford (CPU)

Overview

Unlike its Dijkstra counterpart, the CPU-based Bellman-Ford implementation performed quite differently in the Cluster and Single-Node environments. From a strict time point of view, the Bellman-Ford implementation usually performed much better than its Dijkstra counterpart. Also, unlike the Dijkstra implementation, whose Cluster and Single-Node results were essentially identical, the Single-Node Bellman-Ford results were, at least for low node count runtimes, much more in line with what one would expect when everything is run on a single system – to whit, the runtimes were significantly lower. From an efficiency point of view, however, the Bellman-Ford implementation was not terribly efficient – at least in terms of speed-up. Whereas the Dijkstra implementation usually had linear or super-linear speed-up values, it frequently took multiple nodes before the Bellman-Ford implementation achieved a speed-up of just 2.

One oddity that both the Cluster and Single-Node experiments shared is that, with a few exceptions, the runtime when the implementation transitioned from using one node to two nodes for computation rises – either by a small amount or, most of the time, an amount that could be termed a spike (see Figure 20).

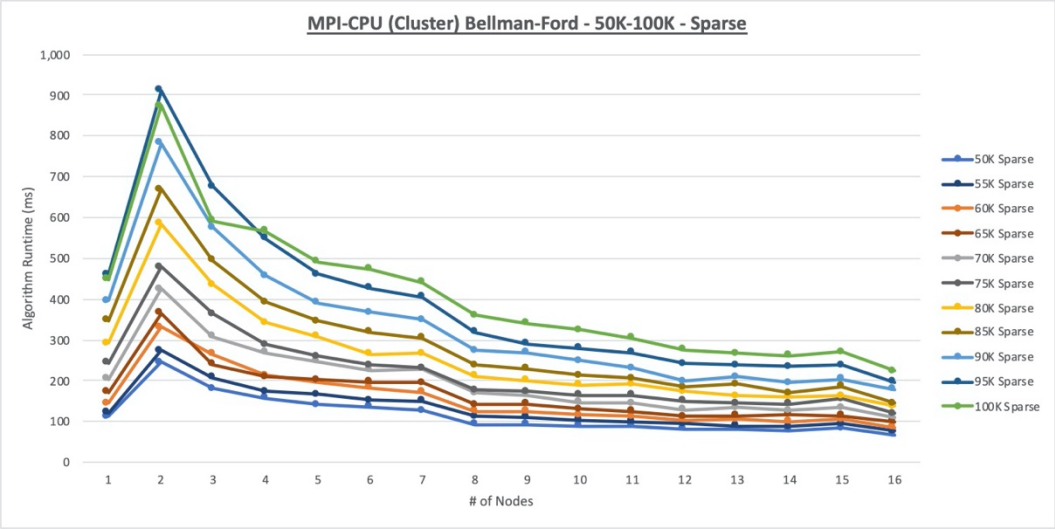

Figure 20. MPI-CPU (Cluster) Bellman-Ford 50K-100K Sparse Runtime Chart

This behavior is counterintuitive, for while one might not achieve a speed-up of two, even accounting for the time impact of the MPI communications, there should be a drop in the runtime, as seen in the Dijkstra results, rather than a spike. Accounting for this oddity is difficult to completely explain, however, it can be noted that when running on one node, the implementation is not having to engage in any network communication. While it does execute some MPI functions, just as if it were communicating with other nodes, since it is the only node, then there is no need to send and receive data across the network. Additionally, since it is the

only node, when the MPI_Allreduce function is executed, since it is the only node, then while the necessary documentation for confirmation was unavailable, it is logical to posit that the MPI function may simply bypass itself, for one needs more than one set of data to perform a comparison, let alone a reduction. Consequently, between the lack of needing to send data across the network – or, for the Single-Node option, to another process – and work with other processes, possibly waiting on their results, the runtime could be seriously reduced. Since the amount of data necessary for broadcasting the graph data is considerably more than that needed for the Dijkstra graph data, that may also be a factor.

Cluster

Reviewing the data generated from the cluster experiments revealed more about how the implementation behaves in a distributed environment than anything else. At the same time, due to some of the behavior observed, all that can be done is to simply report it, for the single-node experiments are often quite different.

Starting small, examining the 5K-10K sparse graph set, the graph lines spike upwards, moving from one node to two, and then, rather than to drop down from there, they continue to rise – the larger graphs more so than the smaller ones – before assuming a gentle slope upwards (see Figure 21). All do noticeably drop a bit from nodes 15 to 16 and the vertical spread of the graph lines does tighten a bit at node 16 – something seen in other graph sets – but, otherwise, the chart almost looks like what one would expect to see, only inverted. Logically, except for the contraction – that is, a noticeable narrowing of the vertical spread of the graph lines – at node 16, one could explain the results as being the result of trying to process graphs that are too small and

too sparse for an implementation, the result being that the communication and some of the processing overhead completely negates – and then some – any gains by processing it in parallel on a cluster.

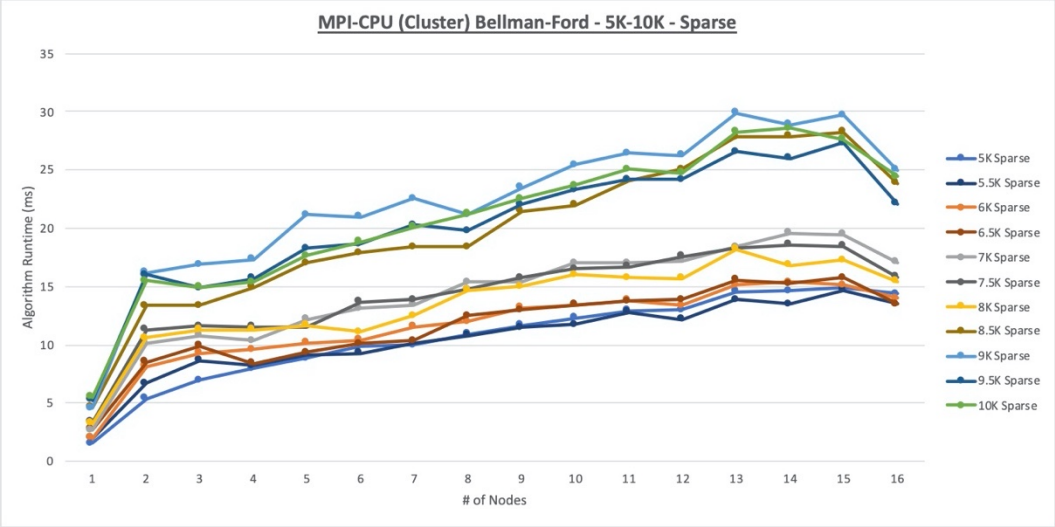

Figure 21. MPI-CPU (Cluster) Bellman-Ford 5K-10K Sparse Runtime Chart

The 50K-100K sparse graph set's runtime results are, compared to the 5K-10K sparse graph set's, quite ordinary (see Figure 22). When two nodes are employed, the runtime spikes upwards and then begins to come down as additional nodes are added. Unlike the 5K-10K set where the graph lines were bundled together in two groups, the graph lines for the 50K-100K set are distinct and fairly well spread out vertically. Examining them for trends, one will see that at five nodes, there is a subtle contraction in the vertical spread and then, from nodes five through seven, the graph lines roughly plateau – the smaller the graph, the flatter the line. At node eight, all graph lines drop down some and there is a more noticeable contraction. From node eight through fifteen, while the larger graphs are still descending a bit, it's the graph lines largely

plateau until a final slight drop and contraction at node sixteen. It is a fairly mundane graph, save for the noticeable drop and contraction at node eight and the fact that with fewer nodes, there is a bit of a shelf.

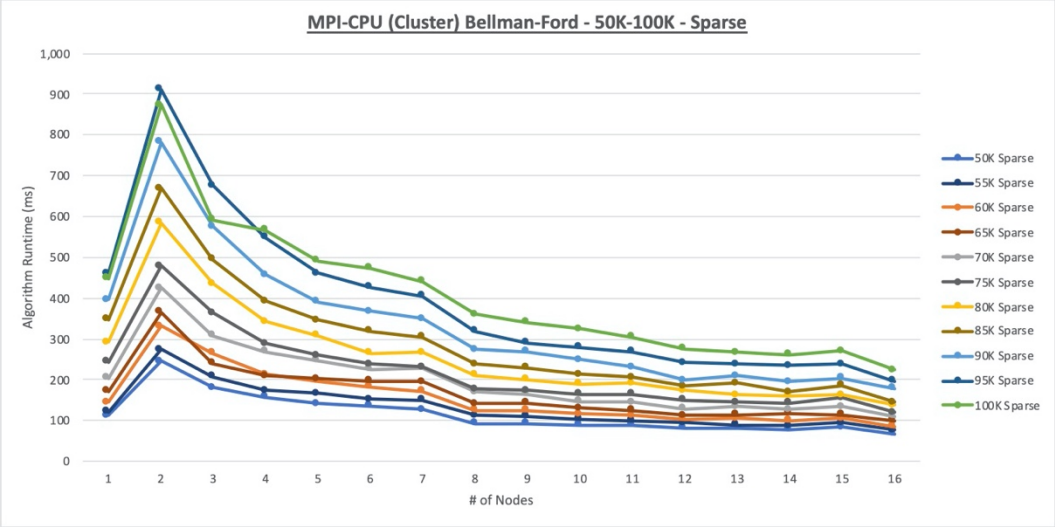

Figure 22. MPI-CPU (Cluster) Bellman-Ford 50K-100K Sparse Runtime Chart

Moving to the 500K-1M sparse graph set (Figure 23), while, as usual, the smaller graphs have smaller spikes and plateau out sooner than the larger graphs, what is striking about this graph set is how spread out, vertically, the results are. Coming down from node two, there is a subtle contraction at node five, after which there isn't a step – the graph lines just slope downwards until another contraction at node eight, after which, with the exception of the largest graphs, there is a plateau that ends at node fifteen, there being a small, but noticeable drop down and contraction at node sixteen. Aside from the performance from nodes two through five, the only real other visual difference is that the vertical spread of the lines is larger than in the 50K-100K graph set. If you look at the runtimes, however, there is a general trend of the vertex count

having increased by an order of magnitude, but the 500K-1M runtimes have increased by (2 * 10^1), so only one full order of magnitude, but moving from 300 milliseconds to 6000 milliseconds means that one should remember that '2'.

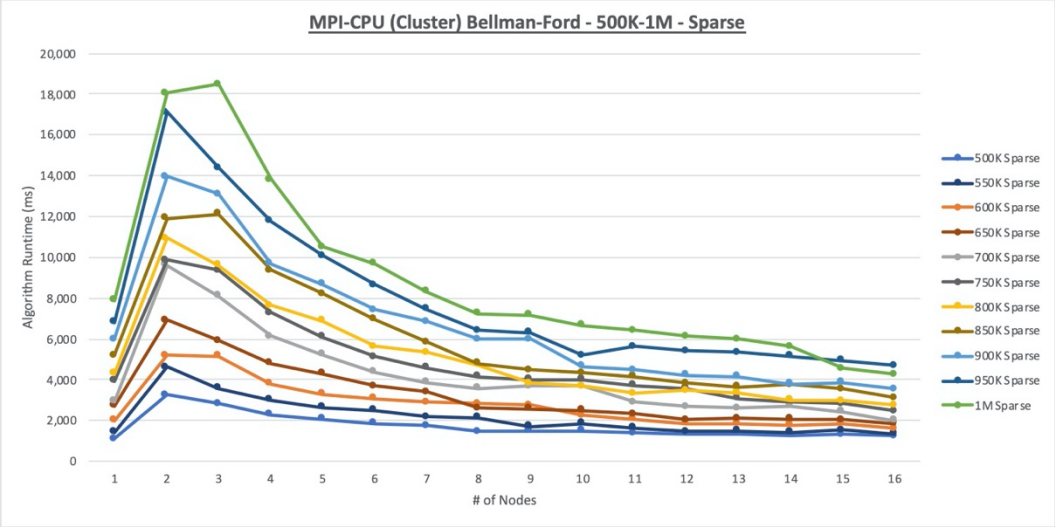

Figure 23. MPI-CPU (Cluster) Bellman-Ford 500K-1M Sparse Runtime Chart

Looking at the medium density graph sets, one finds in the 5K-10K set (Figure 24) that there is a distinct step format that is much like the 50K-100K sparse graph set, only the graph lines are much more spread out until the drop down and contraction at node eight. Like in the 5K-10K sparse graph set, the graph lines are mostly in two groups – the four largest and the seven smallest graphs – and, while there is a small drop and contraction at node sixteen, for node eight through fifteen, the lines are either plateauing or gently rising.

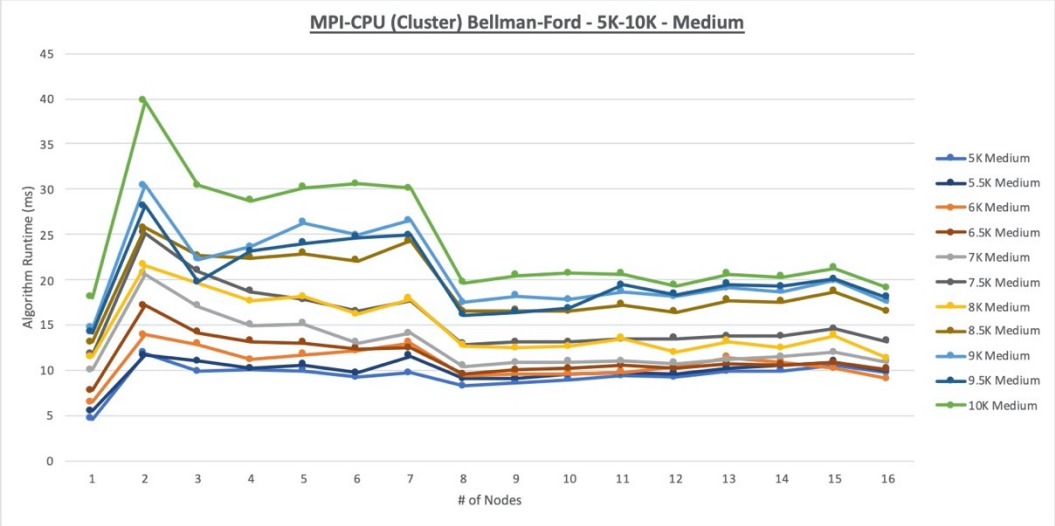

Figure 24. MPI-CPU (Cluster) Bellman-Ford 5K-10K Medium Runtime Chart

The format for the 50K-100K medium graph set (Figure 25) is much like the 5K-10K format, save for the node three to node seven section being a definite ‘step’ and then the graph lines are spread out more evenly before they drop down and contract at node eight. From there, it is the usual plateau until node sixteen, save for the largest four graphs are not presenting a very even line. Looking at the 500K-1M graph set, one might expect yet more of the same, but the chart presented is actually much like the one for the 500K-1M sparse graph set, save for the fact that the contraction is at node nine, not eight, and that the graph lines are much more evenly spread out and then there doesn’t appear to be a contraction at node sixteen. Thus, as the size of the graph increases, the more it shifts from a two-step format back to the ‘smoothly decreasing as the node count increases’ format.

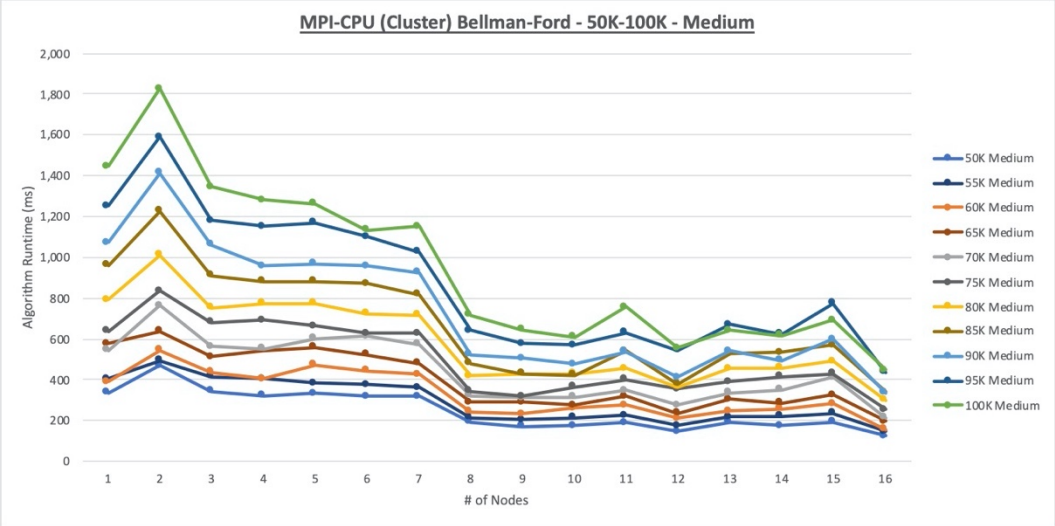

Figure 25. MPI-CPU (Cluster) Bellman-Ford 50K-100K Medium Runtime Chart

For as eclectic as the sparse and medium density graph sets have been, it is the dense graph sets that take the proverbial cake. This is because the runtime chart for the 5K-10K dense graph set (Figure 26) is much like the 50K-100K medium chart, only, from nodes three to seven, while the 5K and 5.5K graphs are mostly flat, the other graphs trend upwards and the larger the graph, the higher the value at node seven is. Indeed, for the 10K graph, the node seven runtime is higher than the node two spike runtime. The graph lines are vertically spread over a great range and yet, when the eighth node is added, all of the graph lines drop – or plummet, as the case may be – and, while the vertical spread is greater than the 50K-100K medium graph set, the graph lines then plateau, save for the four largest graphs, which trace even more jagged lines until it is time to drop and contract once more.

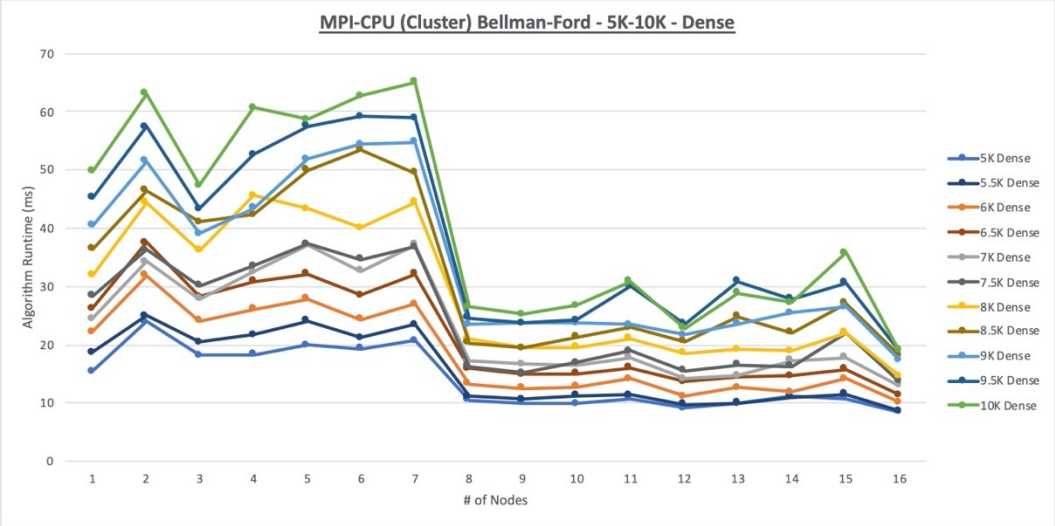

Figure 26. MPI-CPU (Cluster) Bellman-Ford 5K-10K Dense Runtime Chart

The 50K-100K dense graph set (Figure 27) is more of the same, save for the fact that it is the first graph to contain no spikes – while there are some gentle rises for some of the graphs, the rest are either flat from one node to two nodes or down. All graph lines drop until or at node three before rising and mostly plateauing, the vertical spread not as wide as the 5K-10K chart, and then dropping and contracting at node eight. At this point, while the smallest graphs do plateau, the rest have pretty jagged traces as additional nodes are added, though at node sixteen, of course, there is another drop and vertical contraction.

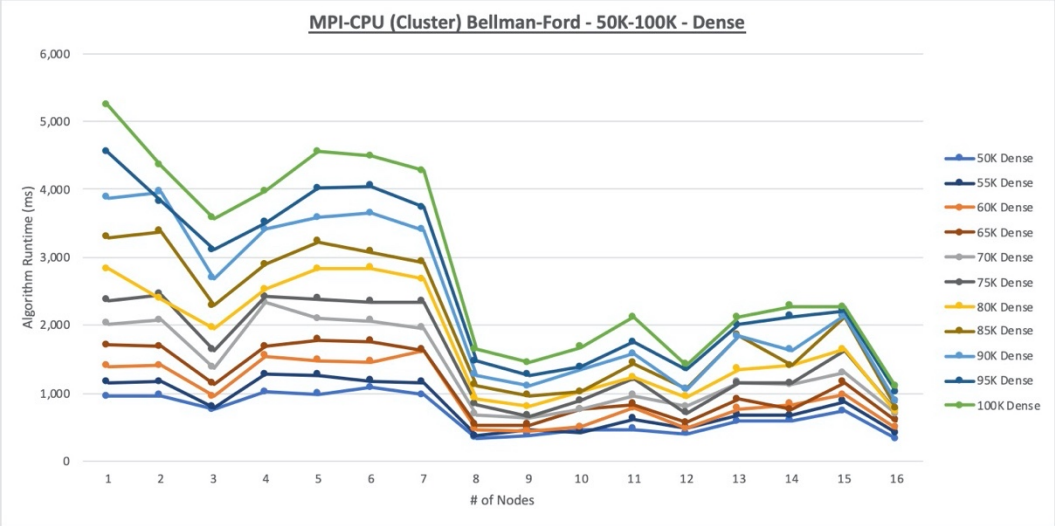

Figure 27. MPI-CPU (Cluster) Bellman-Ford 50K-100K Dense Runtime Chart

The 500K-1M dense graph set (Figure 28) is an abrupt departure from the other dense graph sets, yet it also draws from the 50K-100K dense set as, save for the 850K dense graph, it has no spikes – it is like the 50K-100K set in that regards – and then, looking at the sparse and medium 500K-1M graph sets, while the graph lines are in two separate groups – the largest four graphs and then the rest – the 500K-1M dense chart has the graph lines mostly smoothly coming down as the number of nodes increases. The curve down is steeper for the larger graphs, which mostly plateau beginning at node nine, where there is a slight vertical contraction, and have clearly defined graph lines. For the other seven, while the curve down isn't as steep, the vertical spread is tighter and then, after a drop and contraction at node nine, the graph lines are tightly graphed together.

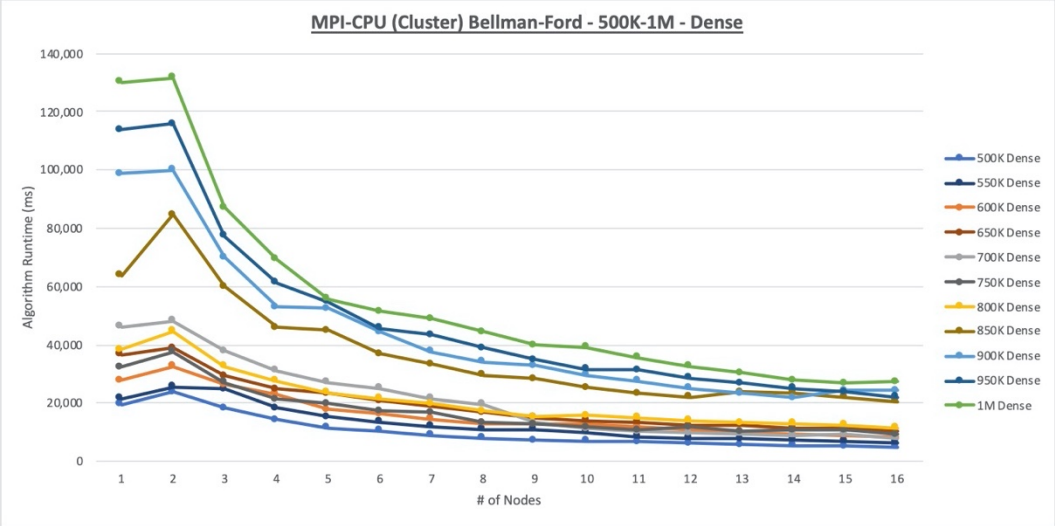

Figure 28. MPI-CPU (Cluster) Bellman-Ford 500K-1M Dense Runtime Chart

Single-Node

While there is still plenty of inter-process communication taking place, by shifting to having the entire experiment on a single system the network component is removed from the equation. As a result, communications should be much faster and, one would think, in the case of the Bellman-Ford implementation, the spike when a second node is added would either go away or be greatly reduced. As it turns out, the spike remains or, for a few graphs that didn't have it, it is gained.

Analyzing the results from the Single Node experiments is not so much a play-by-play of each graph set's results, but, rather, a comparison of the behavior as most of the graph sets have key runtimes that are the nearly the same. The runtimes identified as key are those for one node, two nodes (the spike), three nodes, and then 16 nodes.

For the Single Node experiments, the key runtimes are usually slightly faster, as one would expect due to the lack of delay inherent with network communication. However, with some exceptions that will be detailed, the effects of that lack of delay are not very profound, thus leading to the tentative conclusion that, when looking at the overall picture, the network links do not have that big of an impact on this implementation. It should be noted that that tentative conclusion is only applicable to the stated key runtimes for most of the graph sets and that it is the behavior between node three and node 16 that is the real difference.

Looking at that behavior, excluding the 5K-10K sparse graph set, one immediate observation is the lack of shelves or steps and the lack of the 'drop and vertical spread contraction' behavior that was typically seen at nodes eight and 16 (see Figure 29 for an example). Thus, the two behaviors that were not explicable are gone, however, their replacement is a good news/bad news situation. The good news is that, in general, the graphs – by which I mean the plotted results of the runtimes for each graph in the graph sets – look much like one would expect. The smallest graphs in each graph set normally have the smallest spike and then quickly flatten out, though, unlike Dijkstra, the usual behavior is for the runtime to continue to descend, as seen in the 5K graph in Figure 30, even if only by a small amount – more good news, for it means that parallel processing is still providing a benefit. Also in the good news category is that the larger graphs are also behaving as one would expect – after the spike, the graph line then descends. The bad news is that the larger graph lines can still be descending – and at a rate worth noticing – all the way until node 16 (see Figure 31). Consequently, while there are plateaus or plateau-ish areas, any generalizations will largely come at the expense of the largest graphs – the very graphs which are benefitting the most from the parallel processing!

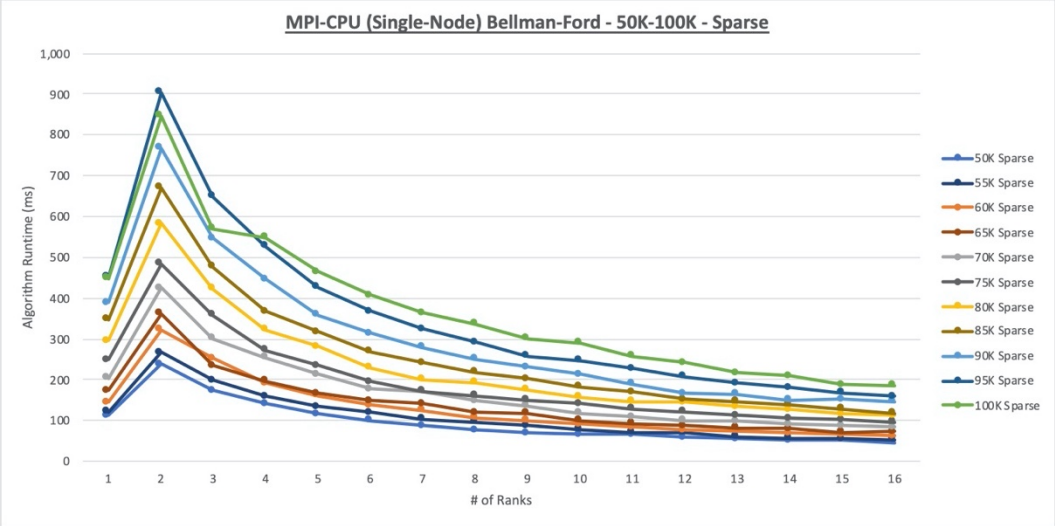

Figure 29. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Sparse Runtime Chart

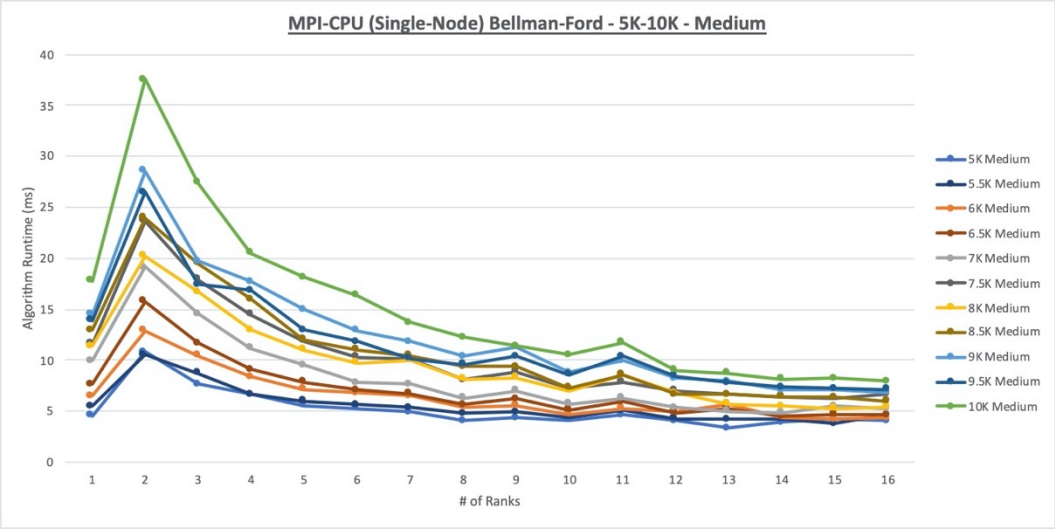

Figure 30. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Medium Runtime Chart

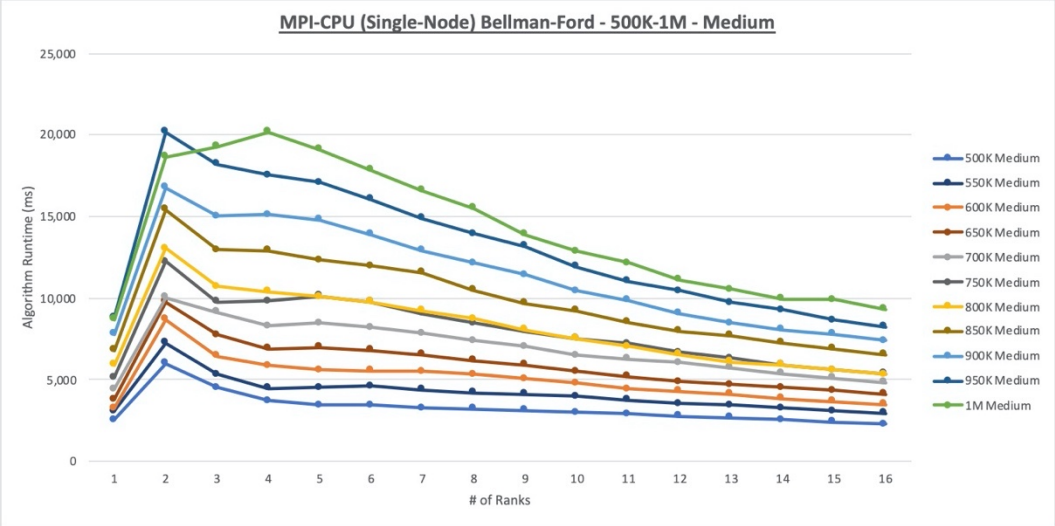

Figure 31. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Medium Runtime Chart

To provide some context and examples for the aforementioned good news/bad news, one can look at the 50K-100K sparse graph set results from the cluster and Single Node experiments (see Figures 22 and 29). A look at the one node times show no difference – quite logical. Looking at the times for the spike, one also finds essentially the same, though the Single Node times appear to be a bit faster. Three node times are a bit faster for the Single Node experiments, and then, extending out to sixteen nodes, the Single Node times are noticeably faster, though not dramatically so. The behavior in between those nodes, however, is significantly different. For the Single Node graph, there is no evidence of what looks to be the beginning of a shelf from nodes five through seven, nor is there a drop/contraction when moving to node eight. Looking to the right, as one increases the node count on the Single Node experiments, the runtime continues to decrease and while the vertical spread of the graph lines does decrease, that is simply a function of the runtimes continuing to decrease, yet because the decrease is so small for the smallest graphs, one could say that the ceiling is coming down, yet the floor isn't moving. Thus, rather

than most of the runtimes plateauing from node eight on the cluster results, the Single Node results continue to decrease and are still faster than the cluster runtimes after the cluster's graph lines drop and contract at node 16.

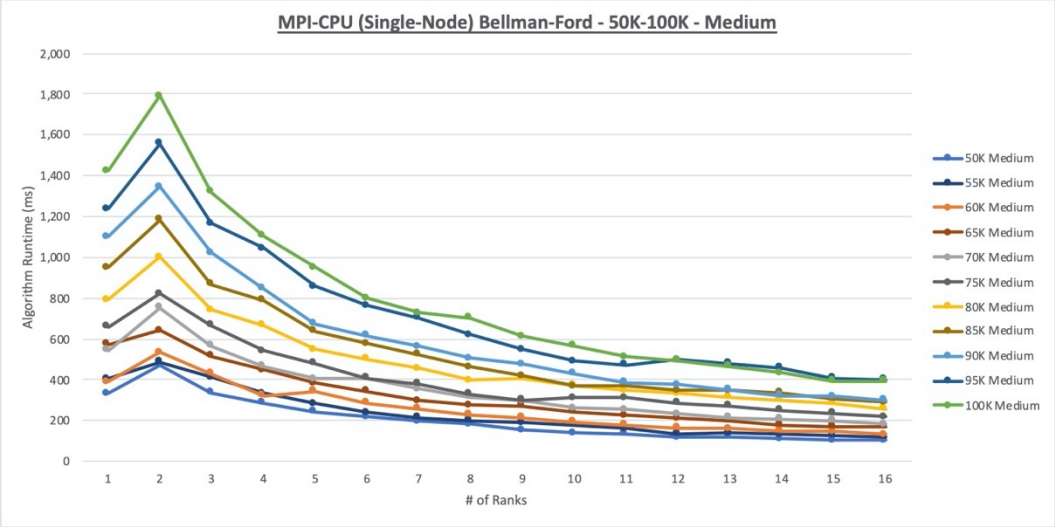

Figure 32. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Medium Runtime Chart

Looking at the 50K-100K medium graph set (Figures 25 and 32), the same behavior is seen, only more pronounced insofar as the shelf is concerned. Something of note is that at node eight, where the cluster results drop and contract, both the cluster and the Single Node experiments have about the same runtimes. However, whereas the cluster plateaus, the Single Node continues to decrease. Like with the sparse set, it is the drop and contraction at node 16 that causes the cluster's node 16 runtimes to be approaching the Single Node runtimes.

The 50K-100K dense graph set's results are like its less dense counterparts insofar as the runtimes for nodes one through three being the same or very close, comparing the Single Node to

the cluster, and the node 16 runtimes being fairly close. Unlike the medium graph set, the Single Node runtimes at node eight were faster than those of the cluster. However, in examining the graphs, looking for that data, it was discerned that for the 50K-100K graph sets, when moving from sparse to medium to dense, from node eight and higher, the runtimes roughly double.

While the 50K-100K graph sets matched up quite well for the node one, two, three, and 16 values with the node 16 deviations being expected, the 5K-10K graph sets were another story. For the 5K-10K medium graph set, the runtime values for node one were the same as the cluster, as expected, and the values for the spike at node two were a little less than the cluster, with the node three values being a little bit more than a little less than the cluster. After that, the desired happened – no step or shelf or drop and contraction – and the expected happened – with everything on one system, thus reducing the inter-process-induced communications delay, the Single Node experiments' runtimes were dramatically reduced (see Figures 24 and 30). Also, a performance plateau was present, it beginning on node eight, ten, or twelve, depending upon the size of the graph. As for the node 16 runtimes, whereas the cluster range from ~9 to ~19 milliseconds, the Single Node's range from ~4.5 to ~8 milliseconds.

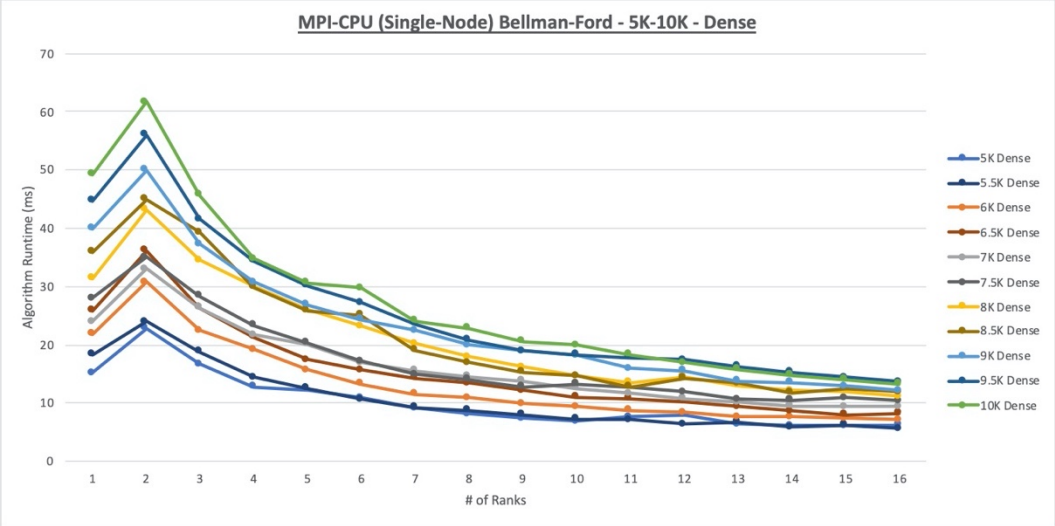

Figure 33. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Dense Runtime Chart

The 5K-10K dense graph set (Figure 33) had a similar start, but had the familiar behavior of the largest graphs never really plateauing while the smallest graphs plateaued from node nine and larger. As for the runtimes at node 16, the cluster's were a bit slower, but not by much – there was much overlap.

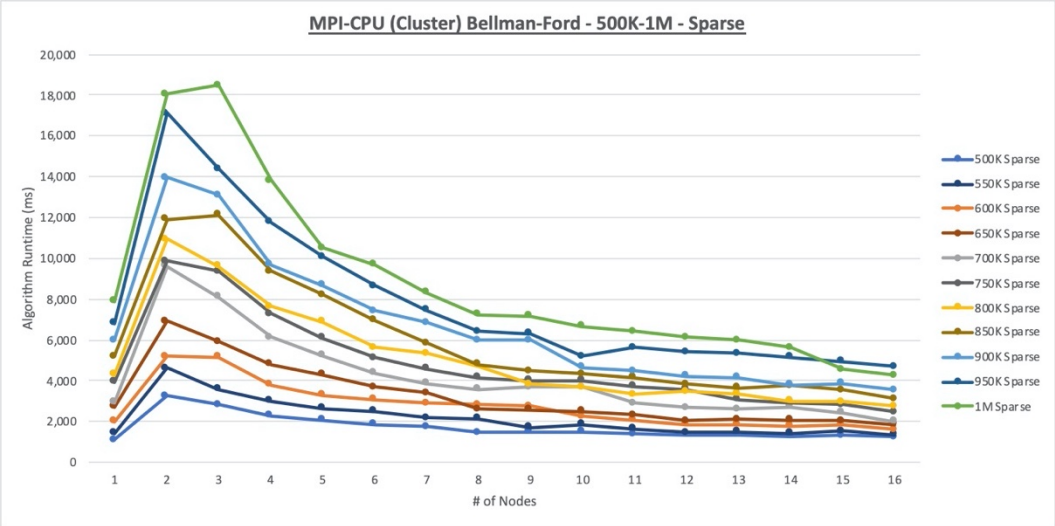

Figure 34. MPI-CPU (Cluster) Bellman-Ford 500K-1M Sparse Runtime Chart

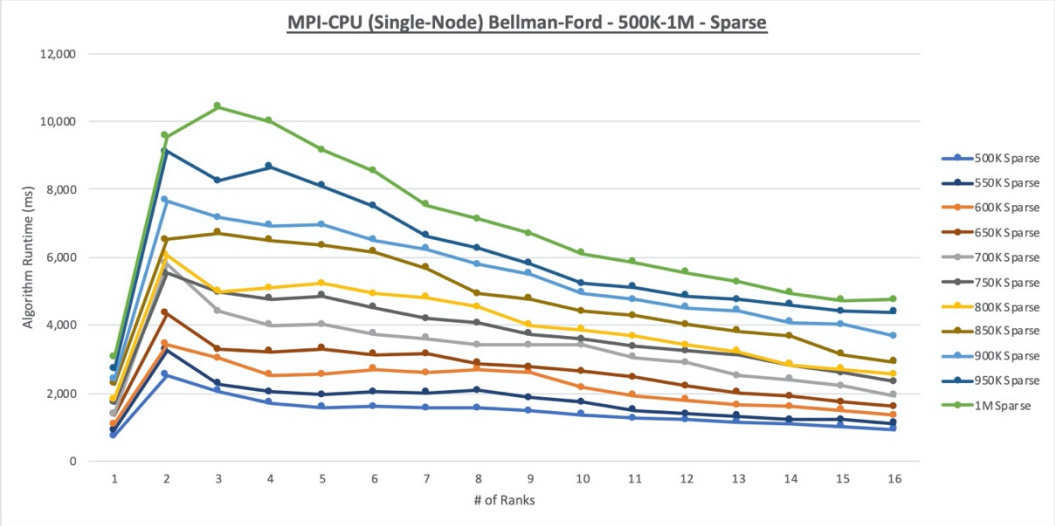

Figure 35. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Sparse Runtime Chart

Studying the 500K-1M graph sets allows one to see the difference network communications – or the lack thereof – makes. It also gives insight as to when a system is becoming 'saturated' – that is, the memory calls and transfers are so numerous that network-induced delays are negated. This can be seen in the 500K-1M sparse graph set. As can be seen in Figures 34 and 35, the two sets of results match up (or very closely) at node eight and from there and higher, they essentially track one another. Before that, however, while the one node runtime is the same, the two node runtime balloons up to over 18,000 for the one-million vertex graph with the other graphs' runtime rising as well. From there, on the cluster, the runtimes trend downwards. For the Single Node experiments, it's not so much a matter of a peak being lopped off, but rather smushed down due to the fact that the processes on the Single Node system had direct access to the data – a simple memory call – whereas the cluster processes had to wait significantly (relatively speaking) longer for all of their data to be transmitted about the network.

For the 500K-1M medium graph set, which also has the ‘smushed spike’, the cluster and Single Node runtimes do not match up until node 10. This can be explained by the graphs in the graph set being denser, which means that while the processes for the two options – cluster and Single Node – are having to do the same amount of work, the network is having to handle even more data.

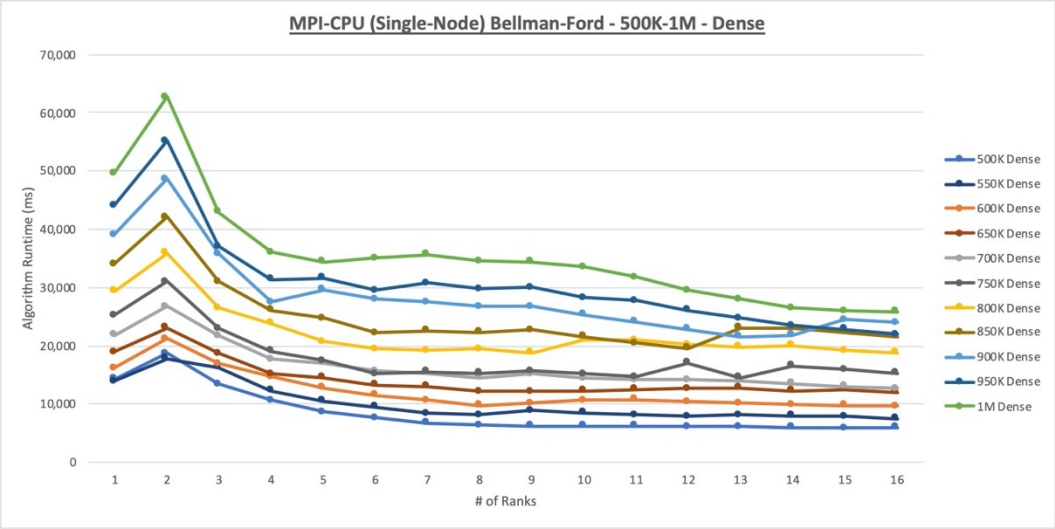

Figure 36. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Dense Runtime Chart

The last of the 500K-1M graph sets – the dense set – is, like the 5K-10K sparse graph set, quite unique. Unlike the cluster results (Figure 28), which have an incredibly high one and two node values for the four largest graphs and the graph set, itself, runs slower, the Single-Node 500K-1M dense set (Figure 36) has a more ‘conventional’ look to it – the graphs spike and then go down. However, compared to other runtime charts for the Single Node experiments, it quickly drops, flattening out at node four, and by node six, some of the graphs have plateaued. From node six, some continue trending downwards until they level off and then some bottom out and

then rise back up. In terms of grouping, the six smallest graphs, from node eight on, are pretty much flatlined and are fairly close together – the largest of the pack – 750K – does separate itself towards the end, while the other five are widely apart after node four, them gradually coming back together at node 14. As a result, if one is looking to find the best runtime/node performance, unless one wishes to extend it out to nodes 13 through 16 – how important the 1M graph's runtime is is the main determining factor – then this is a graph set where one must really consider the application. For all other graph sets, the divider lines on the charts represent no more than 5,000 milliseconds and that's only the 500K-1M medium graph set chart. For the 500K-1M dense graph set chart, the divider lines represent 10,000 milliseconds. Consequently, that 'Well, it doesn't drop down too much more' could easily be two or three seconds. Conversely, if those two or three seconds require three to four additional nodes, then the question is, How important are those two or three seconds?

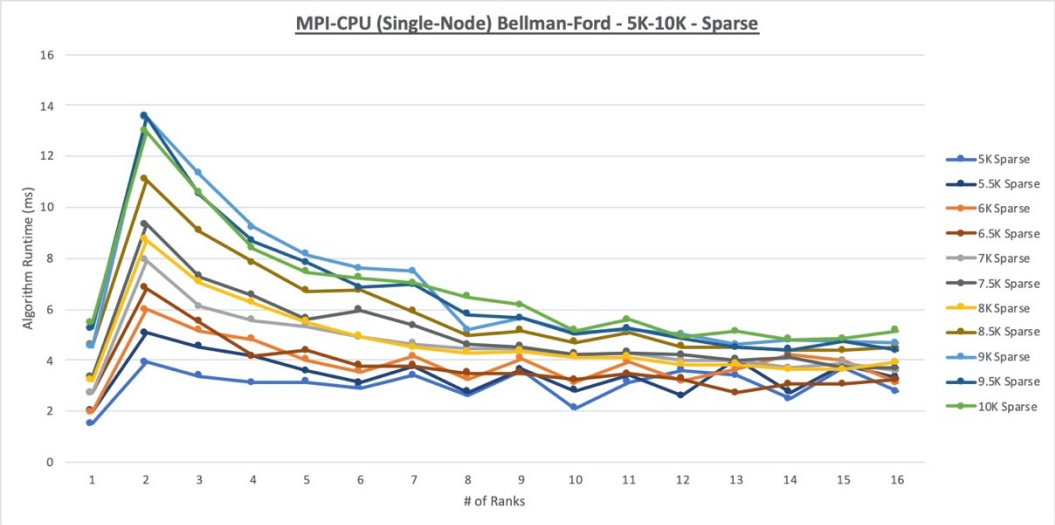

Figure 37. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Sparse Runtime Chart

Regarding the one graph that has yet to be discussed – the 5K-10K sparse graph (Figure 37) – it is also unique, in that while the one node runtimes are the same, Single Node vs cluster, the comparison ends there. The two node spikes are lower but then, after a fairly smooth descent to node five, the graph lines for the smaller graphs drop down at node six, go back up at node seven, and then almost all graph lines drop down a bit – no contraction – just a collective shift – at node eight. They go back up at node nine, down at ten – for most of the smaller graphs, that pattern continues until node 16. Most of the midsize and larger graphs, from node 12 on, flatten out. One is thus left with a graph that looks quite chaotic in comparison with its Single Node counterparts and little in the way of an explanation. Given the small, sparse size of the graphs and the fact that it was the smaller ones that were fluctuating the most, one could reason that since the fluctuations didn't begin in earnest until node eight, what is being seen is the results of processes having to wait on others to finish processing their parts of the graph and/or MPI reduction operations and broadcasts. That explanation does not explain why the graph lines were dipping down on even nodes.

Comparison

While quite a few comparisons between the two hardware configurations have already been made, a subject that has been mentioned, yet never explored, is that of speed-up. Nice, smooth graphs – or at least graphs that can be explained – are nice, but if it takes multiple nodes to break even, let alone get just a speed-up of 2, then the system, the implementation, or the algorithm is not very efficient.

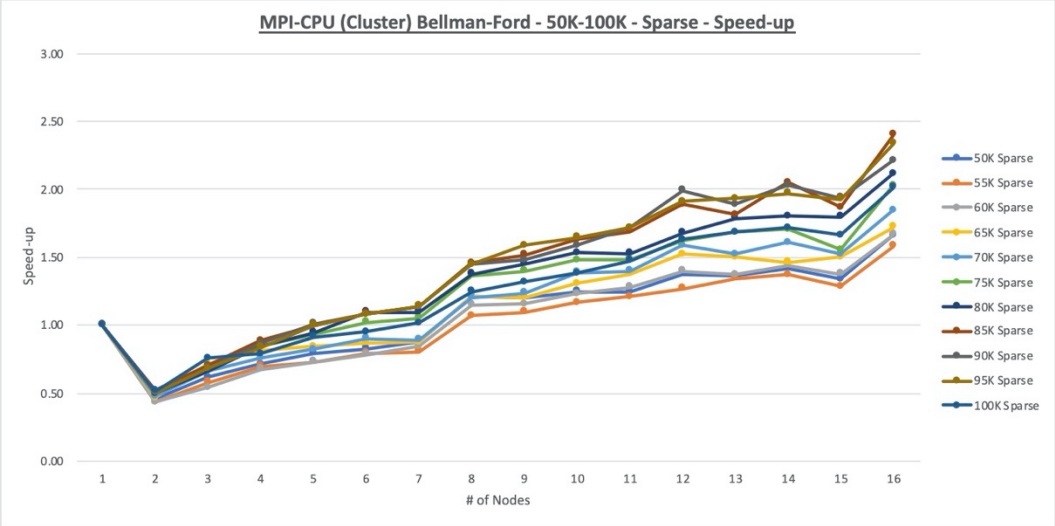

Figure 38. MPI-CPU (Cluster) Bellman-Ford 50K-100K Sparse Speed-up Chart

It has been detailed in the previous section how the Single Node configuration is, at least for the larger graphs in each set, faster than the cluster configuration, despite the fact that the two configurations often end with the values for 16 nodes being close to the same. Examining the 50K-100K graph set, using the cluster, for the sparse set (Figure 38), after the natural plunge due to the spike at two nodes, it's not until five nodes are being used that even one of the graphs achieves a speed-up of 1, and eight nodes are required to cause all of the graphs to have a speed-up of at least 1. 12 nodes were required for one of the graphs to just graze the bottom of the '2x Speed-up' line and only the six largest graphs crossed the '2x' line – the others were somewhere between 1.5 and 2 – and that feat took 16 nodes. Looking to the 50K-100K medium graph set (Figure 39, after a dip for two nodes, the graph lines move upwards at node three and most stay on the high side of the '1x' line through node seven. The drop at node eight takes at least one graph above the '2x' line with the rest between 1.5 and 2. What follows is an up and down section of the chart, the graph lines remaining between 1.5 and 2.5 with a few exceptions before,

after a final dip, they shoot upwards to where a few graphs are above 3 and the rest are between 2.5 and 3. Again, 16 nodes is required to accomplish this. For the dense set (Figure 40), up through seven nodes, while the good news is that the graphs remain just below to somewhere between 1 and 1.5 speed-up and the drop at node eight takes them to between 2.7 and 3.25, the speed-up lines then go up and down – as low as below 1.5 at node 15 – before shooting upwards to 2.75 to 4.75.

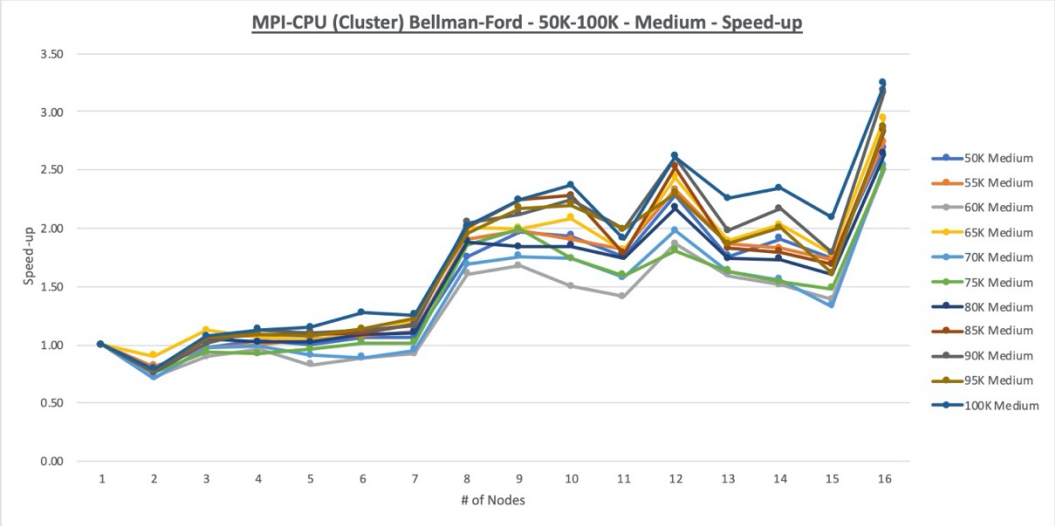

Figure 39. MPI-CPU (Cluster) Bellman-Ford 50K-100K Medium Speed-up Chart

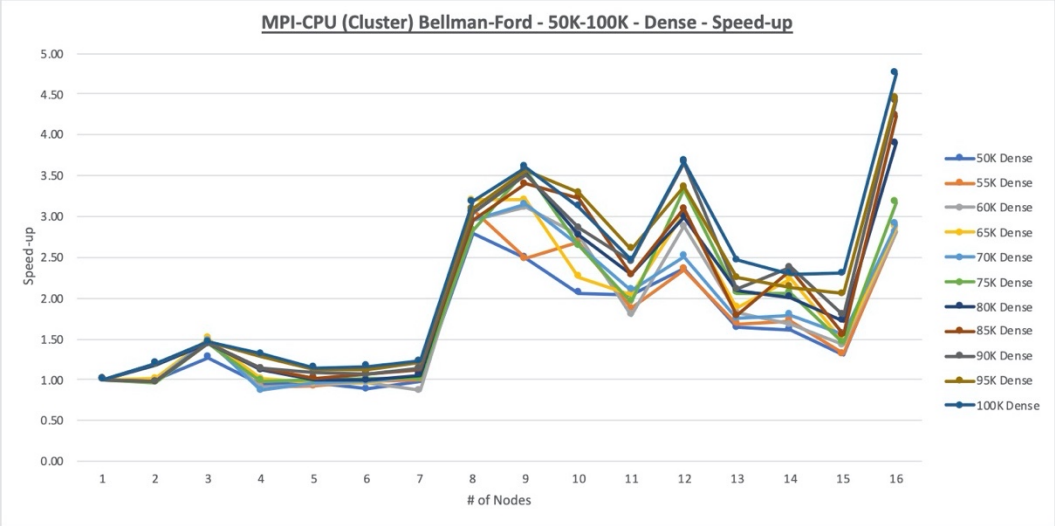

Figure 40. MPI-CPU (Cluster) Bellman-Ford 50K-100K Dense Speed-up Chart

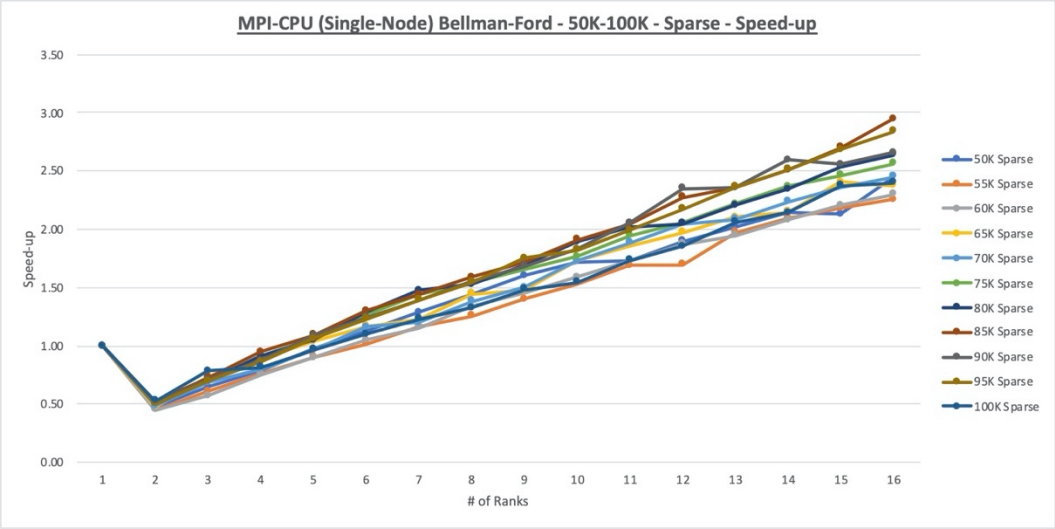

Figure 41. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Sparse Speed-up Chart

Looking at the speed-up plots for the Single Node configuration, the 50K-100K sparse graph set (Figure 41) requires four nodes to get one graph near the '1x' line and six nodes to get all of the graphs above the '1x' line. It took 11 nodes to get at least one graph above the '2x' line and 13 to almost get all across. With 16 nodes, the graphs are spread out between ~2.3 and just shy of 3.

For the medium graph set, three nodes has about half of the graphs over the ‘1x’ line, four nodes has them all, and then seven nodes has one of the graphs brushing the bottom of the ‘2x’ line, though it takes 10 nodes to get all above ‘2x’. 12 nodes has the bottom of the ‘3x’ line being brushed and then 16 nodes has all but one or two above the ‘3x’ line and those below it are just below it. And then, for the dense set (Figure 42), essentially all are at ‘1x’ at node two and all but two graphs are above ‘3x’ with eight nodes. Due to those two graphs, it’s not until 11 nodes are in use that all graphs are above ‘3x’, with all being above ‘4x’ at 16 nodes, some being at 5.5 or slightly above.

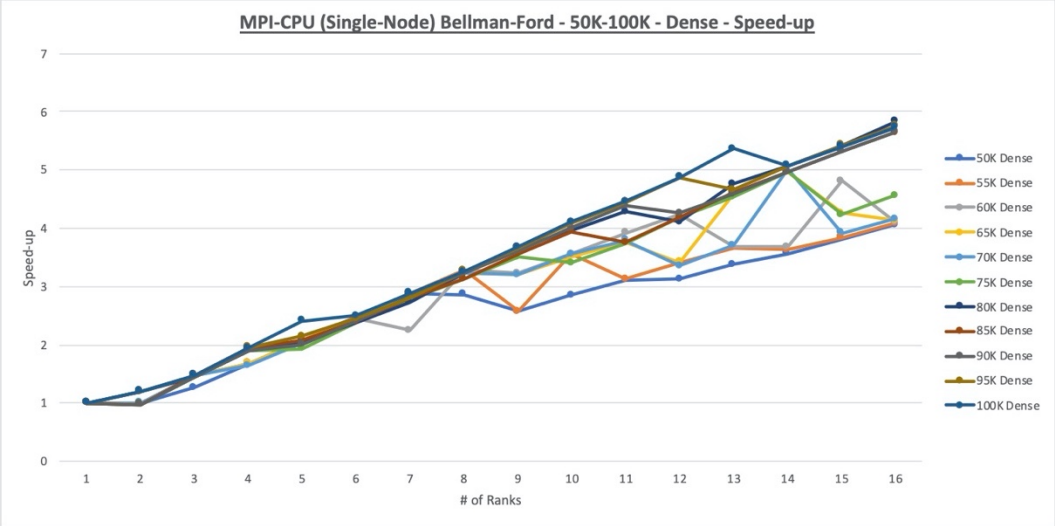

Figure 42. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Dense Speed-up Chart

Examining the speed-up charts for the 500K-1M graph sets, one finds that the cluster has at least some, if not a lot more, speed-up than the Single Node does. Looking to the charts for the 5K-10K graph sets, one finds that the Single Node does better. The seeming ‘discrepancy’ for the 500K-1M graph sets (see Figures 43 and 44 for medium graph comparison), the cluster

outperforming the Single Node and on the largest graph set, can be fairly easily explained by pointing to the fact that to have speed-up, there must be a change in runtimes. For the Single Node experiments, given how the runtimes were 'flattened out', as the number of nodes increased, the runtime difference didn't decrease as it much as it did for the cluster experiments. Perversely, this caused the Single Node results to look worse – when viewed from the speed-up perspective – when the runtimes, up until a point, were actually faster.

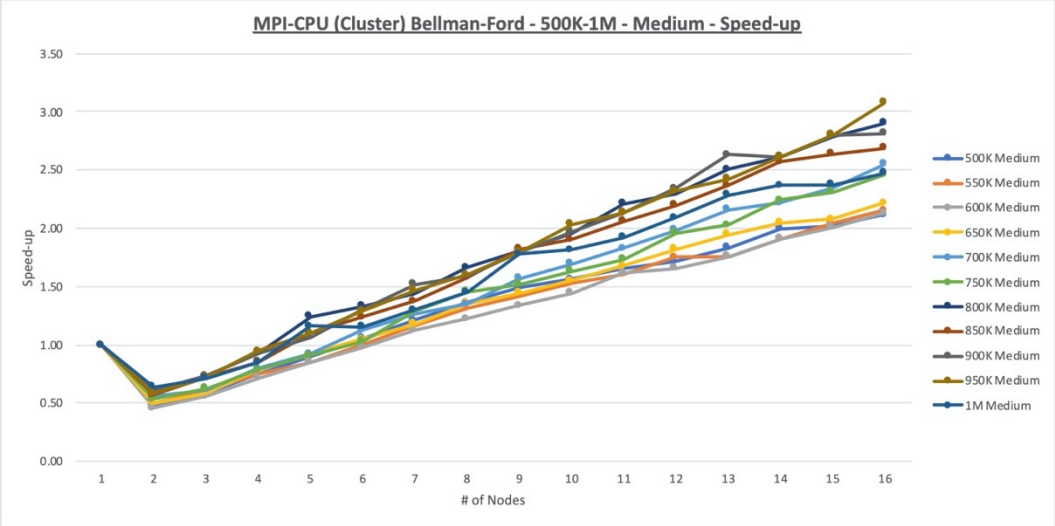

Figure 43. MPI-CPU (Cluster) Bellman-Ford 500K-1M Medium Speed-up Chart

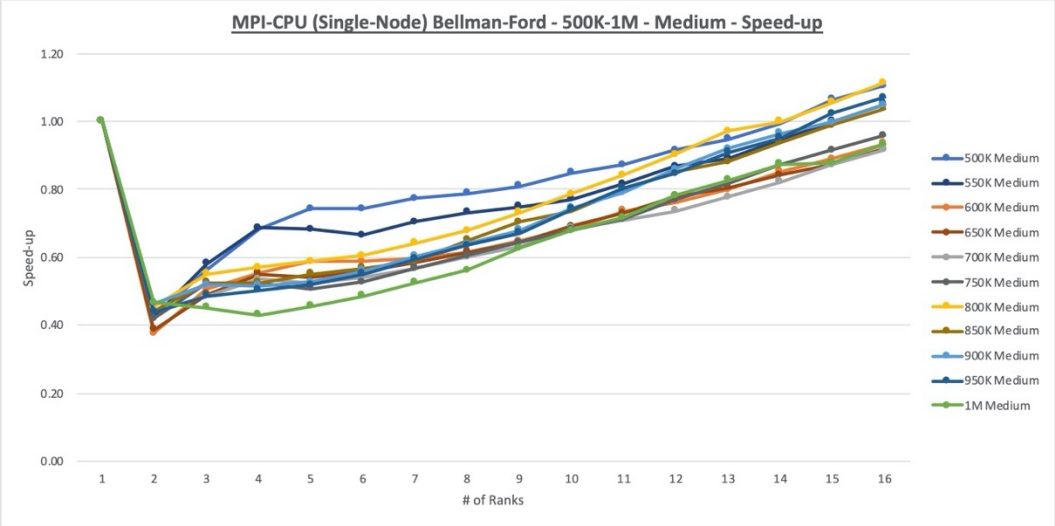

Figure 44. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Medium Speed-up Chart

Directly comparing the two hardware options in general, one can see that at least with this CPU-based Bellman-Ford implementation, in terms of runtime, the Single Node option is the better of the two, though not by much. In terms of speed-up, set in context, again, the Single Node option wins out. From a cost point of view – a machine with a 16-core processor is a lot cheaper than 16 machines – the Single Node option is going to win out. However, as has been noted, if one is running everything on one machine, then all resources are shared and the amount available varies based on how many processes are running. If one has multiple machines, then the resources available per process are not going to change. Also, as was experienced in performing these experiments, running 16 MPI processes on an 18-core CPU was not a problem. Running 16 MPI processes with only 96 GB of RAM normally wasn't a problem, but when the largest and densest graphs were to be processed, the tests had to be moved to an instance with more RAM. With a cluster, each node will almost certainly have more than 6 GB of RAM that it can allocate to a process, thus allowing for more data to be processed. Given that Bellman-Ford is an edge-driven

algorithm and it would be difficult to distribute the graph data in a CSR format, then while the Single Node option gets the nod if there were no constraints, if there are, depending upon what they are, of course, the cluster option might be the best option.

Bellman-Ford (GPU)

The initial intent when drawing up the list of experiments for this thesis was take the Bellman-Ford algorithm, implement it for usage on a GPU, and then extend that to using multiple GPUs to perform SSSP on a graph. While it was not strictly 'plug and play', the CPU-based Bellman-Ford implementation that utilizes MPI for inter-process communication was quickly adapted for the task.

One of the known issues with the planned experiments is that of timing. More specifically, for the distributed Bellman-Ford implementation, each node processes its edges, updating its copy of the *distPred* array (the array of structs containing the distance values and predecessors for the vertices) as it does so, and then all ranks participate in a collective reduction operation which yields them with a *distPred* array that contains the smallest distance values, along with the associated predecessor vertices. Applied to the GPU experiments, that meant that each GPU would have to copy the graph data into its global memory, initialize its *distPred* array, perform the computations, copy the *distPred* array to the host system and then wait while the hosts performed the reduction operation. Once that was complete, the updated *distPred* array would be copied back onto the GPU, the GPU would perform the calculations, and the cycle would continue until the preprogrammed logic said that it – the algorithm – was done. It was known

that the weak – or, rather, longest – link in the distributed implementation would be the reduction operation and the associated network communication. The key question was, How long would the link be? Would it be possible to spread the workload over two GPUs, for example, and have a runtime that was at least even with the one GPU runtime? The answer was a resounding 'No…but'. The results for the 5K-10K medium graph set, run as a cluster, can be seen in Figure 45. The results for the same graph set, but running on a system that contained all four GPUs (Single Node), can be seen in Figure 46. Having all GPUs on the same system clearly helped and the 'but' in the answer is that Peer2Peer [22] memory sharing was not in use, nor were the high-speed NVLink connections used – all GPU-to-GPU communication had to go through the PCIe bus. Had these features been used, all four GPUs' global memory could have been combined into one large memory space. While exploring and exploiting that is beyond the focus of this thesis, it is clear that, at least for this application, in order to harness the power of multiple GPUs, one must bring them together – on the same system – and then work to minimize the time not spent in calculations.

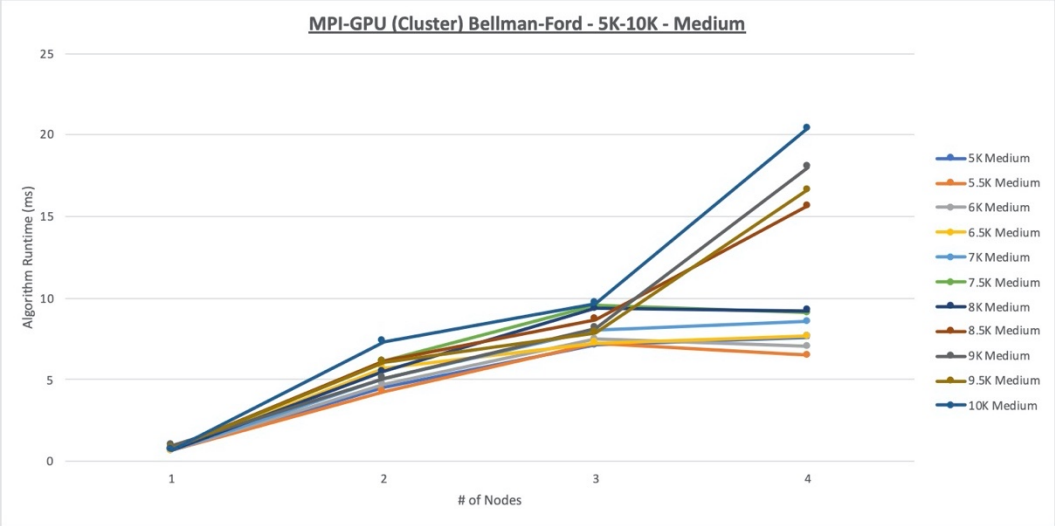

Figure 45. MPI-GPU (Cluster) Bellman-Ford 5K-10K Medium Runtimes

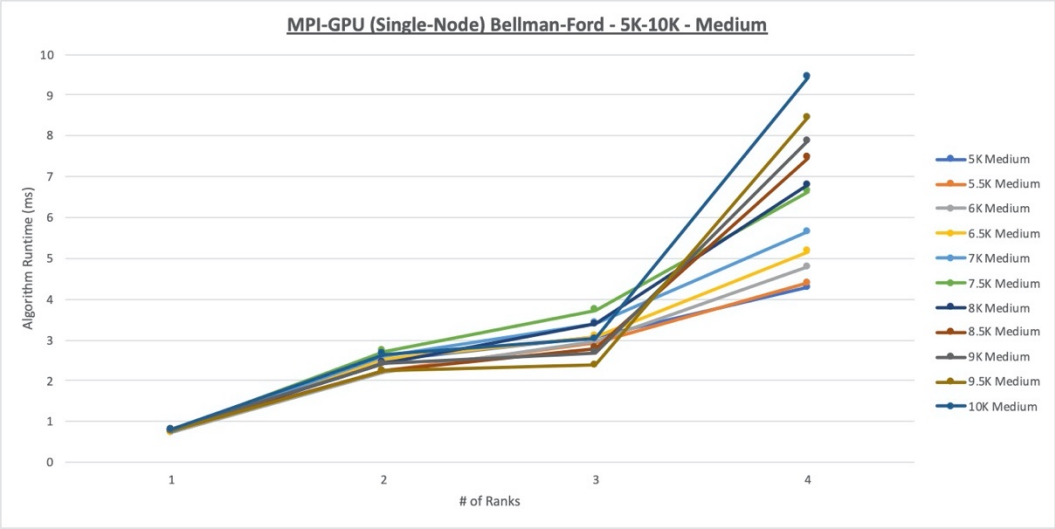

Figure 46. MPI-GPU (Single-Node) Bellman-Ford 5K-10K Medium Runtimes

As a result of the failure to decrease runtimes, while the performance charts are included in Appendices H and I and the Excel file with the runtime data is included with the rest of the runtime data, a detailed examination of those experiments will not be undertaken. Instead, the

next section will contain a comparison of the two Dijkstra GPU implementations and the Bellman-Ford GPU implementation.

Single GPU Comparisons

While GPU threads can be used to do quite a bit of computation, such as is seen in the blocked Floyd-Warshall kernels, the usual practice is to keep the kernels fairly short and simple. With the Bellman-Ford algorithm, that practice can be easily followed – compute which edge the thread is to relax based on its position in the grid, retrieve the data, relax the edge, update if necessary, and then exit. Since the grid will be one-dimensional, there is also the bonus effect of the memory call, requesting the edge data, most likely being part of a coalesced memory access, for the threads in each warp will be requesting data that, in theory, has been stored sequentially in the global memory.

The result, when compared against the two Dijkstra implementations, is, for the 5K-10K graph sets, runtimes for the sparse and medium sets that frequently overlap one another and are higher than those of the Basic implementation, but almost always lower than the H&N implementation, the plot lines staying mainly just below the 0.60 millisecond line on the chart. For the dense set, however, the Bellman-Ford implementation starts out with a lower runtime than the H&N and, the plotline having a lower slope, ends with a runtime of just over one millisecond to the H&N's slightly-above-1.3 milliseconds. The 5K-10K runtimes, while showing that the Bellman-Ford does better than the Dijkstra implementations for denser graphs, are, admittedly, not very spectacular. The 50K-100K results, shown in Figure 47, however, show how it can really take the lead. The plot line for the sparse set is nearly flat and the medium set isn't impacted by the

denser graphs all that much. As can be seen, the Bellman-Ford implementation began the medium set at about 1.5 milliseconds for the 50K graph and finished the 100K graph in a bit under 4 milliseconds. The H&N started the sparse set at about 1.75 milliseconds and finished the sparse 100K graph in ~7.5 milliseconds. As for the dense set, while the Bellman-Ford began on the dense set slightly slower than the Basic did on the sparse set, by the 100K graph, the dense set was completed and done so well under the Basic's sparse time.

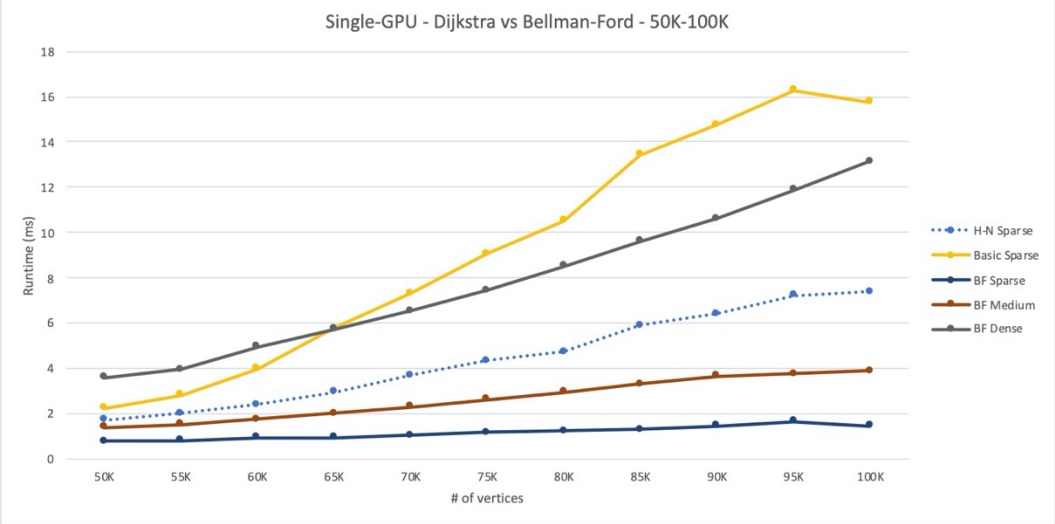

Figure 47. Dijkstra and Bellman-Ford Single-GPU Comparison (50K-100K graph set)

While the Bellman-Ford implementation didn't do as spectacularly on the 500K-1M graphs – it completed both the sparse and medium sets under or, for the last few medium graphs, right around the plot for the H&N sparse results, but then went above the H&N medium plotline for the last three dense graphs – it still quite vividly proved a point. That having been said, since the CPU-based experiments showed that the Bellman-Ford implementation ran considerably faster than the Dijkstra implementation, one cannot automatically credit the programming for the gains.

APSP Experiments

Floyd-Warshall (CPU)

Like the CPU-based Dijkstra and Bellman-Ford implementations, the Floyd-Warshall CPU-based implementation was tested with both a cluster and a single-node configuration, the hardware – more precisely, the instances – being the same. As for the results, the two sets of Floyd-Warshall experiments turned in runtimes that were all but the same – literally. A few runtimes differed by up to two or three-tenths of a second, but, otherwise, the differences were measured in hundredths of a second or less.

Due to runtime restrictions – recall that Floyd-Warshall has a runtime of $O(|V|^3)$ – only the 5K-10K sparse graph was tested. As it was, the one-node runtime for the 10K graph was 46 minutes – with 16 nodes, it was brought down to 2.9 minutes – so, realistically, going much higher wasn't feasible. That having been said, since the Floyd-Warshall algorithm operates on a matrix and will perform the same number of calculations for a graph with a given number of vertices no matter how dense or sparse the graph is, while the test graphs were from the sparse graph set, for measurement purposes, effectively, all three graphs in the 5K-10K family were tested.

The runtime graphs were not all that remarkable in comparison with the other runtime graphs, as is exhibited in Figure 48. The speed-up graphs, on the other hand, were remarkable, for they display near totally linear speed-up. For the cluster results (see Figure 49), beginning at node 12, the speed-up begins to fall from linear such that it is clearly noticeable at node 16, but the

reduction from linear speed-up is still a very small amount and the reduction decreases as the graphs grow larger. That is to say, the 10K graph has barely fallen from the line denoting linear speed-up, while the 5K graph line is the farthest away. Not surprisingly, for the single-node results, the drop-off, which begins at node 14, is much less.

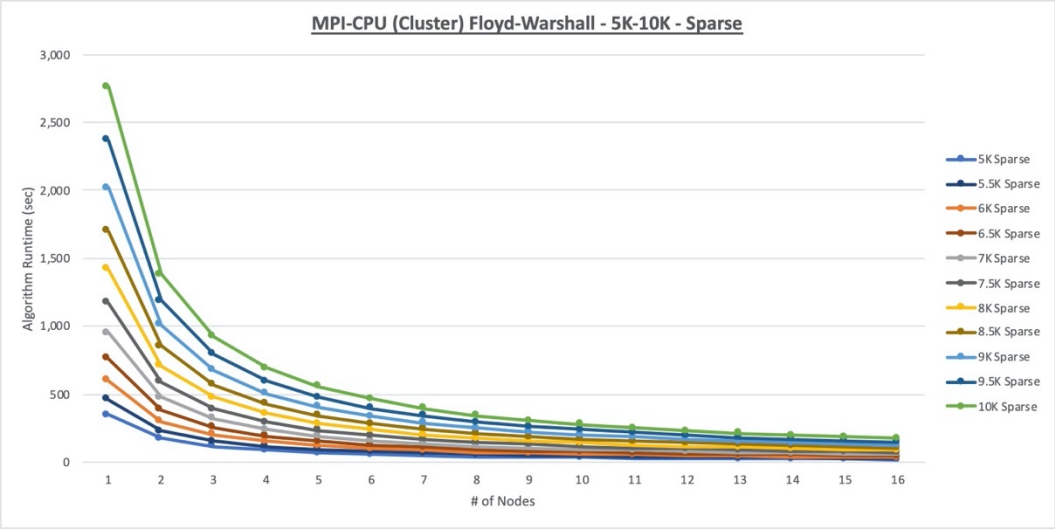

Figure 48. MPI-CPU (Cluster) Floyd-Warshall 5K-10K Sparse Runtime Chart

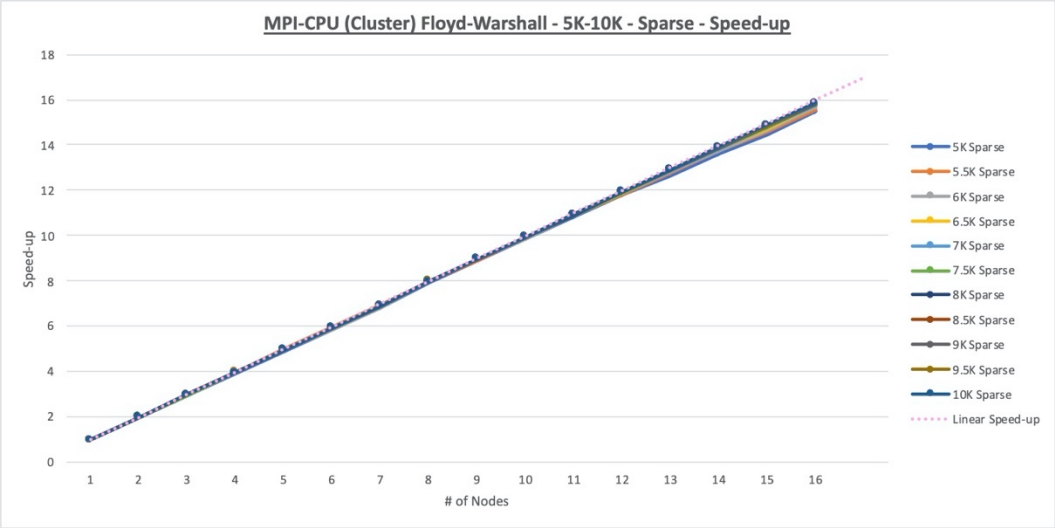

Figure 49. MPI-CPU (Cluster) Floyd-Warshall 5K-10K Sparse Speed-up Chart

Given that the CPU-based Floyd-Warshall implementation processes the graph one row at a time, while there was not an expectation *per se* when beginning the experiments, receiving essentially identical results, Cluster and Single-Node, was quite surprising. Given the amount of inter-process communication, there was the expectation that the Cluster experiments would have longer runtimes. Since they did not, that indicates that the network links were not a bottleneck. It should be noted that while there is a lot of inter-process communication taking place, the amount of data being sent out, while not trivial, is still quite small and, unlike the SSSP implementation, there are no collective reduction operations.

Floyd-Warshall (GPU)

As can be seen in Figure 50, the performance gain by utilizing a GPU for the processing is rather large, to put it mildly, and that is with the CPU runtimes being those resulting from using 16 nodes for computation. Figure 51 shows the 5K-10K CPU results in the context of the results from the experiments that were performed using GPU-based implementations. While some visualizing is necessary, recalling that the nominal runtime for Floyd-Warshall is $O(|V|^3)$, one can roughly extrapolate the plot of the CPU-based implementation and quickly understand why GPU testing of up to and including 40K graphs was possible while CPU testing beyond 10K wasn't reasonable.

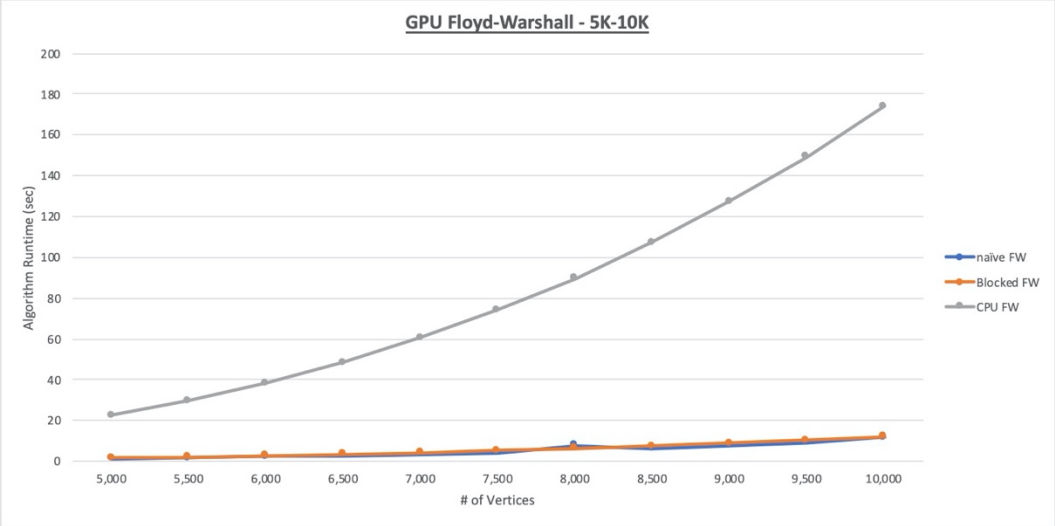

Figure 50. Chart comparing runtimes for 5K-10K graphs for CPU and GPU Floyd-Warshall implementations

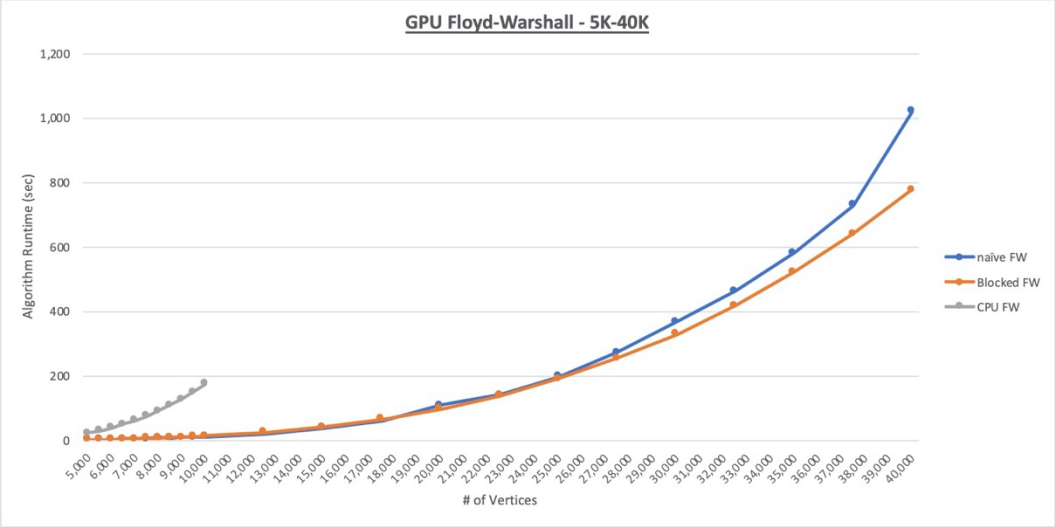

Figure 51. Runtimes for 5K-40K graphs for GPU Floyd-Warshall implementations and runtimes for 5K-10K graphs for CPU implementation

Examining the GPU implementations themselves and comparing their performance, it can be seen that up until about the 22.5K graph, the two are mostly pacing one another and that when they do diverge, it is a gradual separation (excluding the naïve implementation's runtime jump

for the 40K graph). The two implementations pacing each other as far as they do is somewhat puzzling as the naïve implementation is called naïve for a reason – namely, with the exception of determining the optimal block size during development, no attempts at optimization have been made and, even for a GPU, Floyd-Warshall is a computation-intensive algorithm. That having been said, there are two factors that should be taken into consideration. The first is that the V100 GPU is a big GPU – with 80 operational SMs and 64 cores per SM, it has 5,120 cores available, which means that it can be processing that many cells in the *dist* matrix at any given time. While the naïve implementation has not been optimized, by virtue of the amount of processing power available to it, it can maintain a higher level of performance than might be expected. The second factor to consider is that while the blocked Floyd-Warshall kernels may have a level of optimization in them through the use of shared memory, for CUDA kernels, they are rather complex. Furthermore, the repeated requirement to synchronize all of the threads in the block, while necessary, does induce a time delay. As a result, while the blocked implementation performs better overall, it is plausible that what is seen in Figure 51 is a case of the additional work involved in the blocked implementation slowing down its runtime such that it and the naïve implementation, assisted by the amount of available computational processing power, are pacing one another. Eventually, though, between the optimization techniques utilized in the blocked implementation and the graph size passing a point at which, mostly likely, sheer computational processing power is not enough to keep runtimes down, the two implementations' runtime plots diverge.

APSP-via-SSSP

Due to how the Floyd-Warshall algorithm works, using it to perform APSP on the 5K-10K sparse graph set will yield the same runtimes as the medium and dense graph sets would. Since the SSSP implementations used for this thesis do have different runtimes for graphs of differing densities, then that meant that to perform APSP-via-SSSP on the 5K-10K graph set, all three graph sets would have to be processed. While that may sound like a negative at first, it actually is a positive, for the Floyd-Warshall algorithm is effectively running the worst-case scenario – a complete graph – each time, yet in the real world, the densities will fluctuate. Thus, being able to run APSP on graphs with the same number of vertices, yet varying densities, is advantageous.

Figure 52 shows the results of the running APSP-via-SSSP on the 5K-10K dense graph set as well as the 12.5K-40K dense graph set. For comparison, the runtimes for the naïve and blocked Floyd-Warshall implementations are overlaid. It is quite clear that, excluding the Basic Dijkstra implementation which has already established itself at not performing well for dense graphs, effecting APSP with SSSP algorithms is significantly faster. One fact, though, that should be noted is that while the graphs tested are more representative of ‘dense’ graphs, the Floyd-Warshall runtimes apply to up to and including the densest of graphs, while the SSSP-based runtimes will change if the graph is made denser.

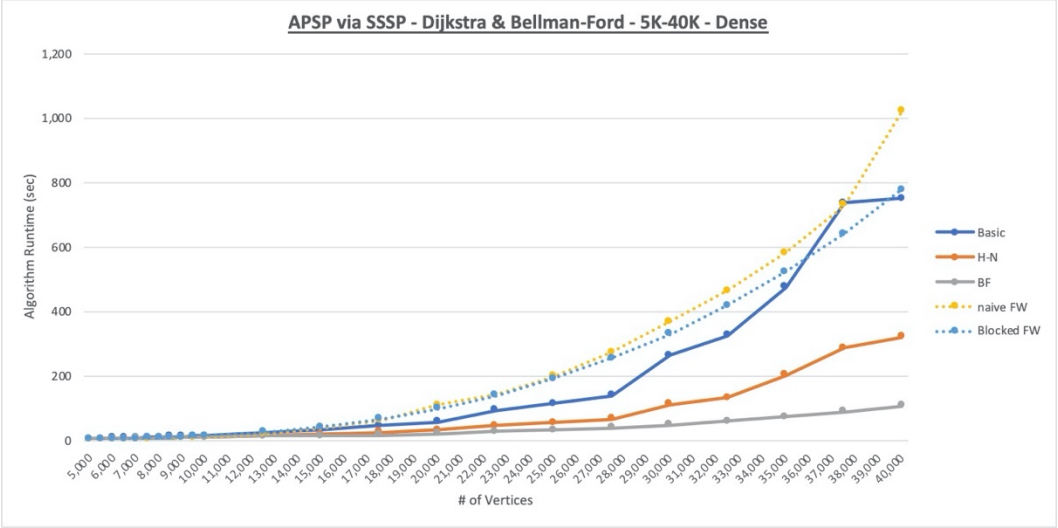

Figure 52. APSP-via-SSSP Runtime Chart 5K-40K Dense with Floyd-Warshall data overlaid

While the above caution about changing densities is true, at the same time, Figure 53 shows the results of APSP-via-SSSP on the sparse 5K-10K and 12.5K-40K graph sets.

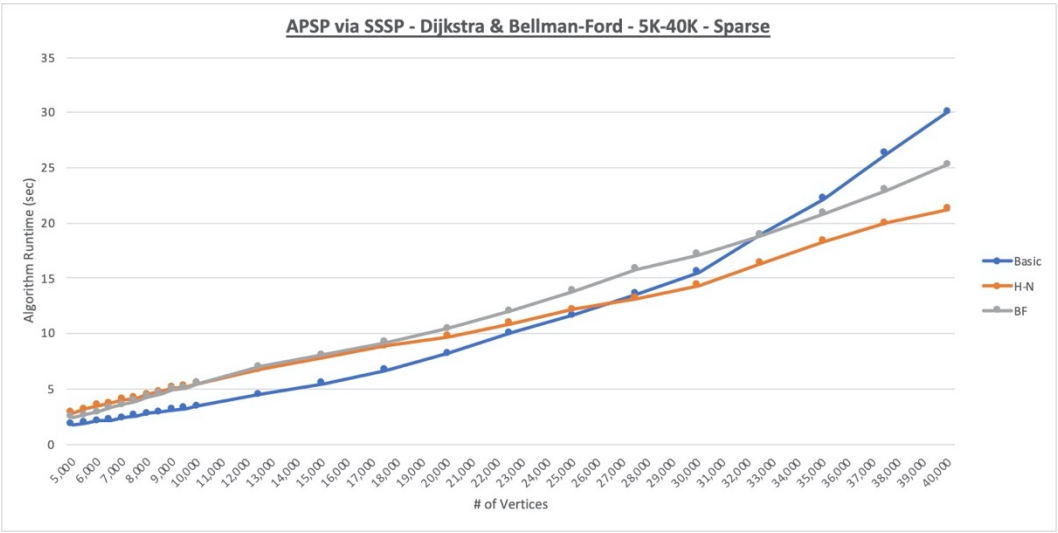

Figure 53. APSP-via-SSSP Runtime Chart 5K-40K Sparse

As can be seen by a quick comparison of the runtimes, performing APSP-via-SSSP on sparse graphs is much, much more efficient than using the traditional Floyd-Warshall algorithm.

CHAPTER VI

CONCLUSIONS

As planned, multiple pathfinding algorithms, implemented using parallel processing techniques, were analyzed utilizing a plethora of test graphs whose characteristics ranged from low densities with small vertex counts to high densities with large vertex counts. Upon analyzing the resulting performance data, however, the desired goal of determining an algorithm implementation's strengths and weaknesses vis-à-vis its performance on the different test graphs was unobtainable for the CPU-based algorithm implementations. With regards to the GPU-based implementations, some conclusions were possible, however, they were focused primarily on pure speed, for once the graphs had more than 10,000 vertices, the 'fastest, second fastest, etc' order of the implementations did not change, regardless of the graph's density.

The fact that the planned goals and intentions of this thesis were not fully met should not, however, be mistaken for a lack of learning taking place and information being gathered. Rather, the experiments that produced that learning and information were simply not on the original task list. For example, in performing the CPU-based SSSP tests, which consisted of Dijkstra and Bellman-Ford implementations being run on a cluster with 16 compute nodes and then run on an 18-core CPU, the MPI framework treating each of the cores as a node, there were multiple unexpected observations. The implementation of Dijkstra, which is difficult to parallelize, performed poorly, time-wise, but returned linear and super-linear speed-up results, scaling very well across both systems. The easily parallelized Bellman-Ford algorithm performed very well, time-wise, but as soon as a second node was added, the runtime almost always spiked and multiple additional nodes were usually required just to return the runtime to the one-node

runtime – if that was ever achieved. For the Bellman-Ford implementation, achieving a speed-up of 2 was noteworthy. Additionally, while the Bellman-Ford results only partially matched, if one were to compare the Dijkstra runtime charts of the distributed system to those of the single CPU, with a few exceptions, one would think they were identical. Indeed, for the CPU-based Floyd-Warshall implementation, which ran on the same systems, they essentially were – the differences were usually measured in hundredths of a second.

The GPU-based Dijkstra experiments yielded no surprises to speak of as preliminary testing had indicated that the more simplified implementation would run more quickly than the more complex implementation, but only up until a point. Formal testing confirmed this, though with the exception of the dense graph sets, the more simplified implementation did not fare too badly. What did yield a surprise was the performance of the Bellman-Ford implementation, as it handily outperformed both Dijkstra implementations on all but the small, sparse graph sets. Unfortunately, a shortcoming was revealed by a random check of the GPU during the experiments with an NVIDIA monitoring utility – namely, the Bellman-Ford implementation used more memory than expected and, as a result, grossly exceeded the 8 GB target limit when processing the largest and most dense graphs. This discovery was made by chance and neither of the Dijkstra implementations were examined, so it cannot be officially ruled out as an issue with the Dijkstra implementation. However, given the data structures used to store the graph for the two algorithms, it is most likely not an issue for the Dijkstra implementations as their data structures are more compact in nature.

While the GPU-based Floyd-Warshall experiments generated results that were much better than CPU-based Floyd-Warshall experiments, it was the 'add-on' APSP-via-SSSP experiments that proved to be perhaps the most noteworthy of the GPU experiments. One could argue that the fact that the Floyd-Warshall algorithm is density-blind – that is, no matter how sparse or dense the graph, it will perform the same number of calculations for a graph with a given number of vertices – is a benefit, for if one has a table mapping runtimes to the number of vertices in a graph, one could, all other things held equal, fairly accurately predict the runtime for a graph with a given number of vertices. The problem with that line of thinking is that the Floyd-Warshall runtimes represent, effectively, the runtime for the most dense graph possible with that number of vertices. In real life, though, the density of the graph will usually be less, if not a lot less, and the SSSP algorithms have shown that their runtime is directly related to the density of the graph. Accordingly, if the graph is a sparse graph, then performing APSP by way of repeated SSSP runs should yield a considerable performance gain. This theory was tested and mostly proven correct – for the graphs up to 10K, the Floyd-Warshall, Dijkstra, and Bellman-Ford implementations were all within five seconds of one another, the Floyd-Warshall implementations often being the fastest. For the 12.5K to 40K graphs, however, with the exception of the least efficient Dijkstra implementation when processing the dense graphs, the APSP-via-SSSP method proved to be a much more time efficient method for performing APSP.

Reflecting on the experiments performed, both planned and unplanned, two future avenues of inquiry are opened. The first is that for the distributed processing experiments, be it multiple nodes or multiple cores, the MPI interface was always used. While such an interface is required when the processes are in different memory spaces, if all of the processes are in the same

memory space, then a shared memory approach using OpenMP could be employed, thereby reducing the overall amount of memory required and eliminating some redundant data structures (for example, all processes would share one array containing distance/predecessor values rather than each having their own). Given how well the Bellman-Ford implementation performed on the GPU, which is, in a global sense, a shared memory environment, it would be worth investigating its performance on a CPU when utilizing OpenMP rather than MPI for parallel processing. The second avenue of inquiry revolves around the issue of GPU-to-GPU communication. As has been noted, when performing the experiments on an AWS EC2 instance that had multiple GPUs, while the GPUs were equipped with the NVLink interface and could have shared their memory, the decision was made to treat each GPU as a discrete unit – an action that mirrored how each core was being treated for the MPI-CPU (Single-Node) experiments. One of the effects of this decision was that all data traffic between the GPUs travelled over the PCIe data bus. Connecting the GPUs together with the much faster NVLink interface and carefully using the Peer2Peer memory sharing might allow for faster multi-GPU runtimes rather than the slower runtimes measured in the experiments.

BIBLIOGRAPHY

[1] AMD. Welcome to AMD ROCm Platform. Retrieved from https://rocmdocs.amd.com/en/latest/.

[2] Argonne Leadership Computing Facility. Aurora. Retrieved from https://www.alcf.anl.gov/aurora.

[3] Bellman, Richard. 1958. On a routing problem *Quart. Appl. Math.* **16** (1958), 87-90. DOI: https://doi.org/10.1090/qam/102435

[4] Davidson, Andrew, Baxter, Sean, Garland, Michael, and Owens, John D. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (*IPDPS '14*). IEEE Computer Society, USA, 349–359. DOI: https://doi.org/10.1109/IPDPS.2014.45

[5] Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1,** 269–271 (1959). https://doi.org/10.1007/BF01386390

[6] Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., and Lavenier, D. "Efficient Multi-GPU Computation of All-Pairs Shortest Paths," 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 2014, pp. 360-369, DOI: 10.1109/IPDPS.2014.46.

[7] ESnet/Lawrence Berkeley National Laboratory. iPerf3. Retrieved April 16, 2021 from https://iperf.fr/.

[8] Floyd, Robert W. 1962. Algorithm 97: Shortest path. Commun. ACM 5, 6 (June 1962), 345. DOI:https://doi.org/10.1145/367766.368168

[9] Ford, Lester R. Jr. 1956. *Network Flow Theory*. Paper P-923. Santa Monica, California: RAND Corporation.

[10] Harish, Pawan, and Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th international conference on High performance computing* (*HiPC'07*). Springer-Verlag, Berlin, Heidelberg, 197–208.

[11] Katz, Gary J. and Kider, Joseph T. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (*GH '08*). Eurographics Association, Goslar, DEU, 47–55.

[12] Khronos Group. OpenCL Overview – The Khronos Group. Retrieved from https://www.khronos.org/opencl/.

[13] Mahmud, Saaduddin. 2020. Parallel Programming With CUDA Tutorial (Part-4: The Floyd-Warshall Algorithm). (July 2020). Retrieved April 13, 2021 from https://saadmahmud14.medium.com/parallel-programming-with-cuda-tutorial-part-4-the-floyd-warshall-algorithm-5e1281c46bf6.

[14] Malkin, G. 1998. RIP Version 2. (November 1998). Retrieved April 13, 2021 from https://www.rfc-editor.org/info/rfc2453.

[15] Martín, Pedro J., Torres, Roberto, and Gavilanes, Antonio. 2009. CUDA Solutions for the SSSP Problem. In *Proceedings of the 9th International Conference on Computational Science: Part I* (*ICCS '09*). Springer-Verlag, Berlin, Heidelberg, 904–913. DOI:https://doi.org/10.1007/978-3-642-01970-8_91

[16] Moy, J. 1998. OSPF Version 2. (April 1998). Retrieved April 13, 2021 from https://www.rfc-editor.org/info/rfc2328.

[17] NVIDIA Corporation. CUDA Zone | NVIDIA Developer. Retrieved from https://developer.nvidia.com/cuda-zone.

[18] NVIDIA Corporation. NVLink and NVSwitch. Retrieved from https://www.nvidia.com/en-us/data-center/nvlink/.

[19] NVIDIA Corportation. Programming Guide :: CUDA Toolkit. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[20] Oak Ridge National Laboratory (Leadership Computing Facility). Frontier. Retrieved from https://www.olcf.ornl.gov/frontier/.

[21] Ortega-Arranz, H., Torres, Y., Llanos, D. R., and Gonzalez-Escribano, A. "A new GPU-based approach to the Shortest Path problem," 2013 International Conference on High Performance Computing & Simulation (HPCS), Helsinki, Finland, 2013, pp. 505-511, doi: 10.1109/HPCSim.2013.6641461.

[22] Schroeder, Tim C. Peer-to-Peer & Unified Virtual Addressing. Retrieved April 16, 2021 from https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[23] The Regents of the University of California. Gunrock. Retrieved from https://gunrock.github.io/docs/#/.

[24] Thomas, Jeremy. 2020. LLNL and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer. (March 2020). Retrieved April 13, 2021 from https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer.

[25] Warshall, Stephen. 1962. A Theorem on Boolean Matrices. J. ACM 9, 1 (Jan. 1962), 11–12. DOI:https://doi.org/10.1145/321105.321107

[26] Weiss, Stewart. 2019. Chapter 5 Floyd's Algorithm. (October 2019). Retrieved from http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter05.pdf.

[27] Wikipedia. 2021. Bellman-Ford algorithm – Wikipedia. Retrieved April 16, 2021 from https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[28] Wikipedia. 2021. Dijkstra's algorithm – Wikipedia. Retrieved April 16, 2021 from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[29] Wikipedia. 2021. Floyd-Warshall algorithm – Wikipedia. Retrieved April 16, 2021 from https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

APPENDICES

APPENDIX A

SOURCE CODE

Source code for the experiments performed, utility programs such as the random graph generator used to produce the test graphs, and Excel files containing the results of the experiments can be found at https://github.com/kf4ayt/MTSU_Thesis. Alternately, I can be contacted via email at cwj2z@mtmail.mtsu.edu.

APPENDIX B

MPI-CPU (CLUSTER) DIJKSTRA DATA CHARTS

The runtime and speed-up charts for the MPI-CPU (Cluster) Dijkstra experiments are presented as follows: Starting with the 5K-10K sparse graph set, the runtime charts for two graphs will be displayed on one page and then associated speed-up charts will be on the following page. For the ninth graph set (500K-1M dense), the runtime and speed-up charts will be on the same page.

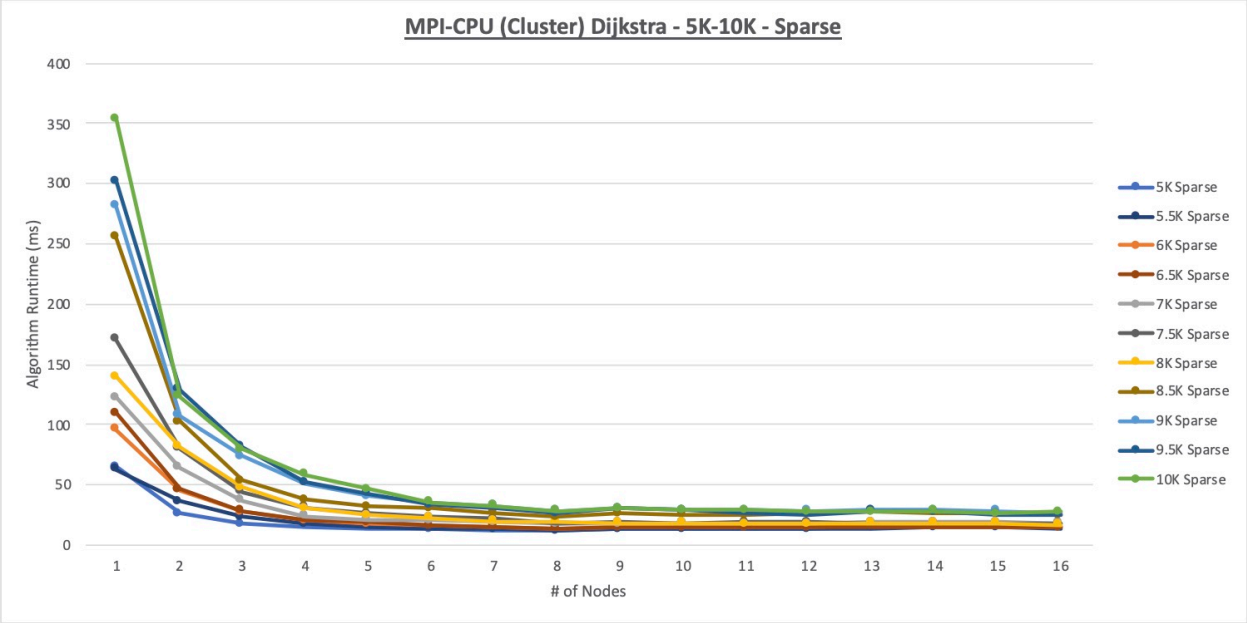

Figure B1. MPI-CPU (Cluster) Dijkstra 5K-10K Sparse Runtime Chart

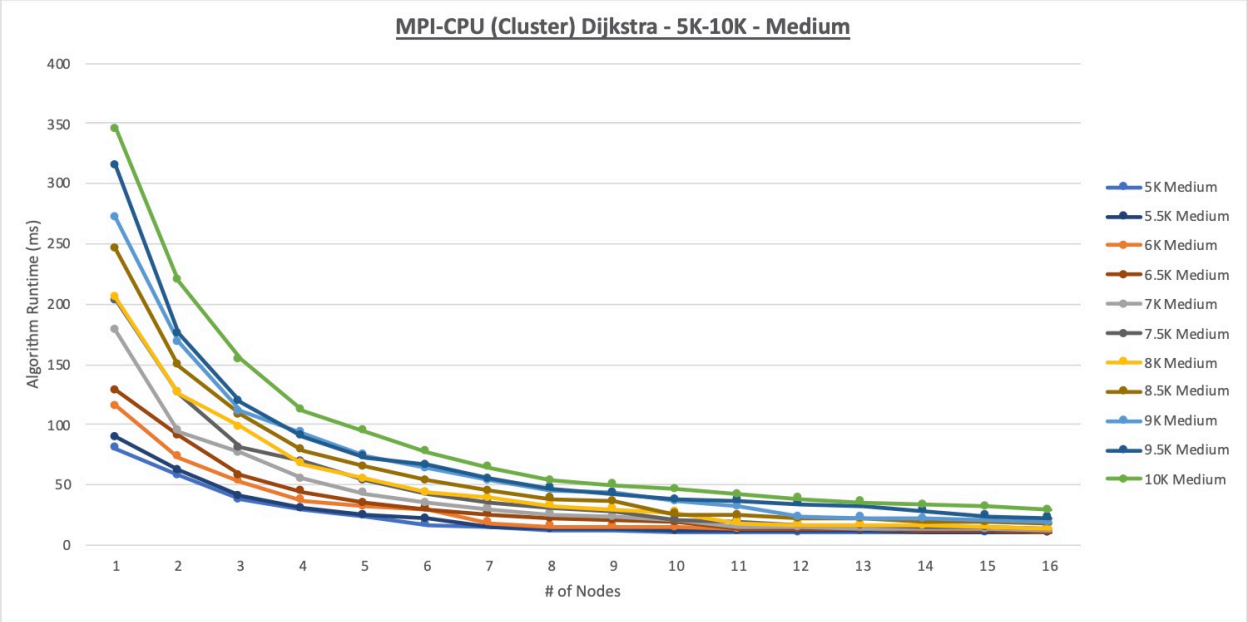

Figure B2. MPI-CPU (Cluster) Dijkstra 5K-10K Medium Runtime Chart

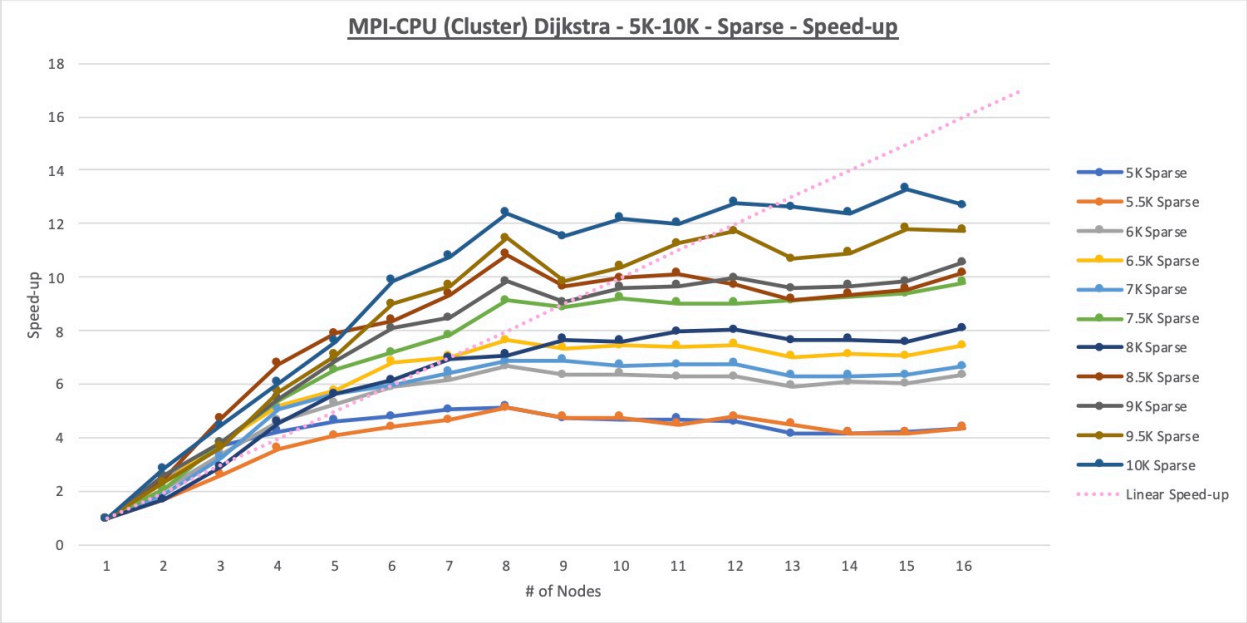

Figure B3. MPI-CPU (Cluster) Dijkstra 5K-10K Sparse Speed-up Chart

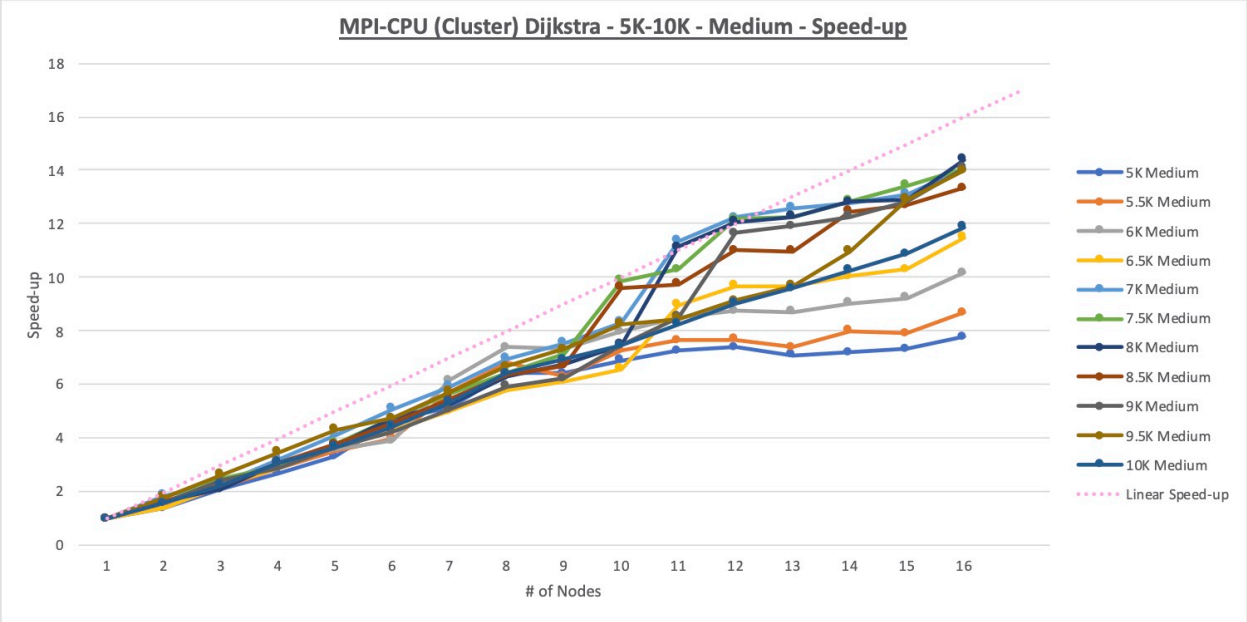

Figure B4. MPI-CPU (Cluster) Dijkstra 5K-10K Medium Speed-up Chart

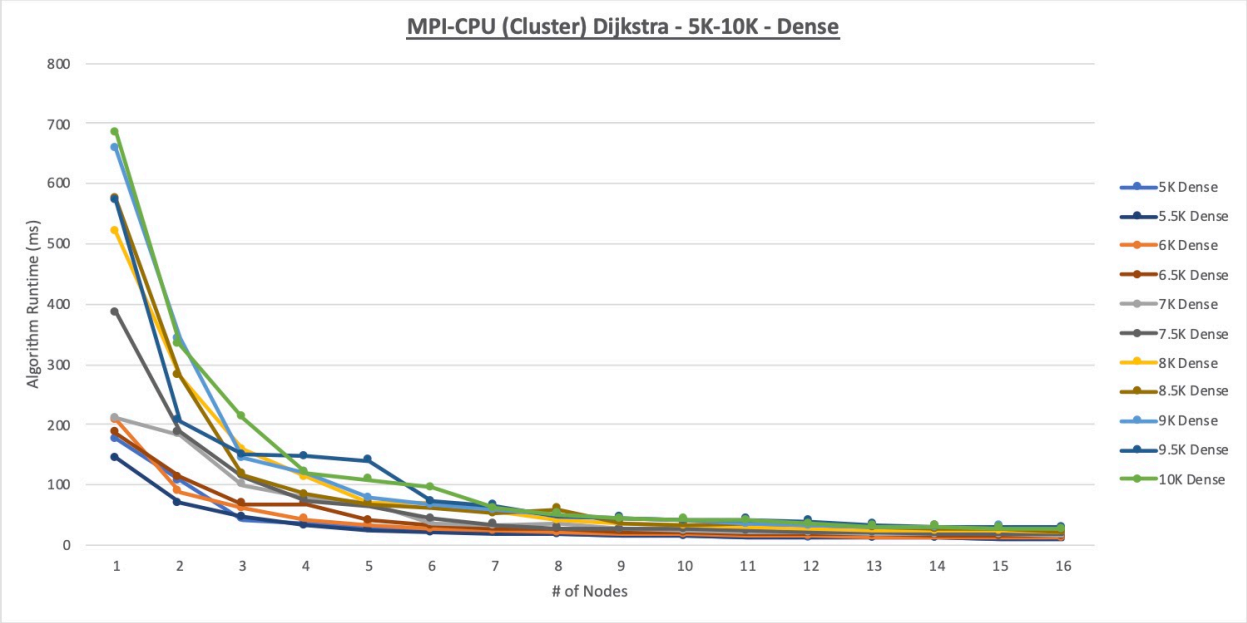

Figure B5. MPI-CPU (Cluster) Dijkstra 5K-10K Dense Runtime Chart

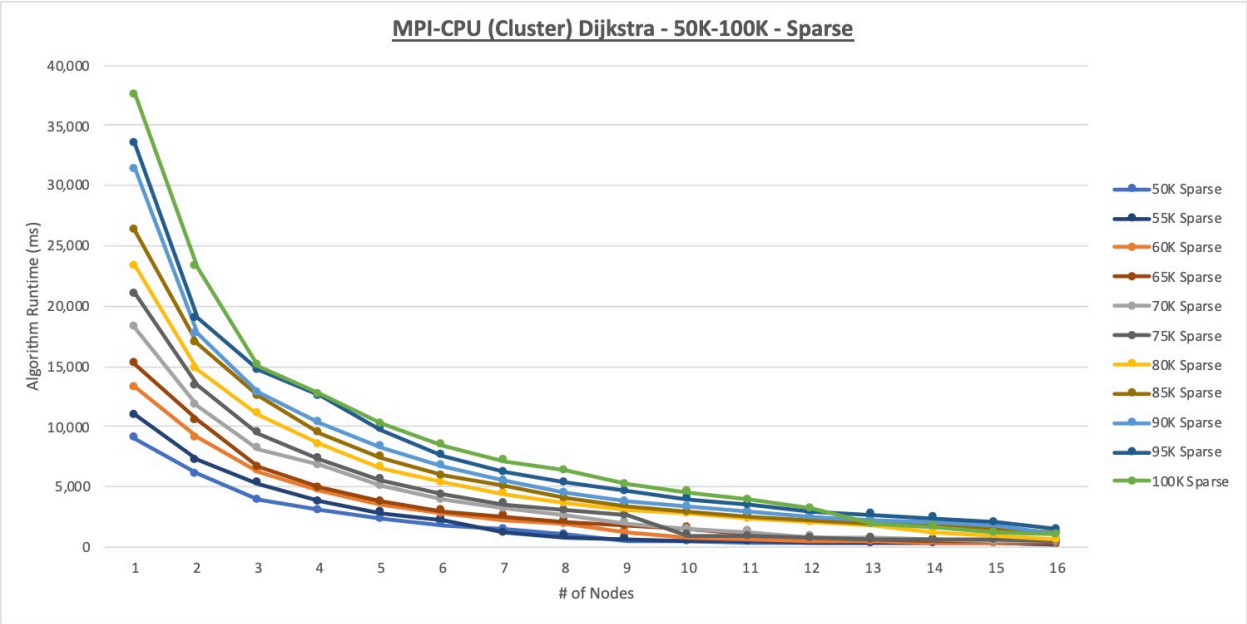

Figure B6. MPI-CPU (Cluster) Dijkstra 50K-100K Sparse Runtime Chart

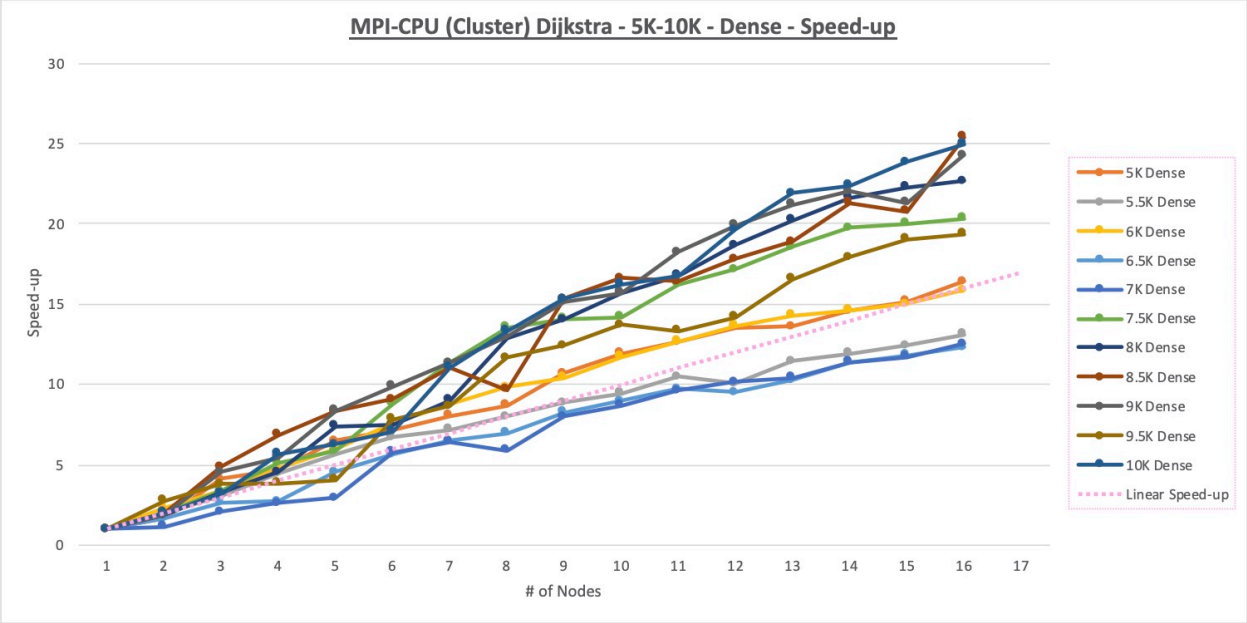

Figure B7. MPI-CPU (Cluster) Dijkstra 5K-10K Dense Speed-up Chart

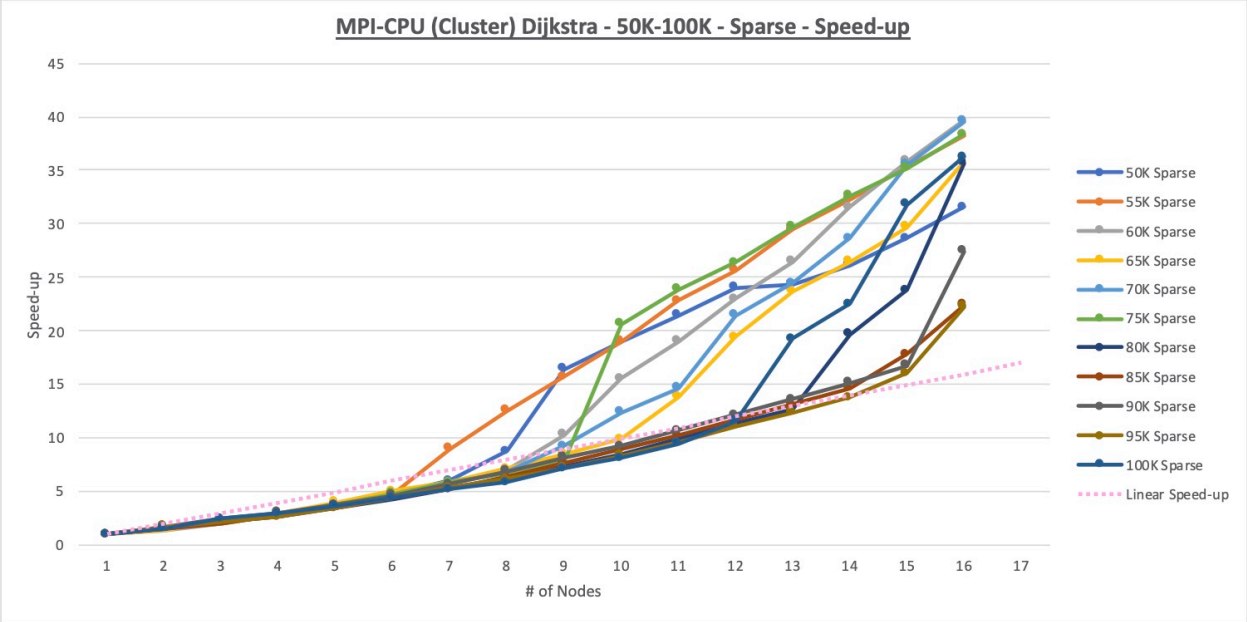

Figure B8. MPI-CPU (Cluster) Dijkstra 50K-100K Sparse Speed-up Chart

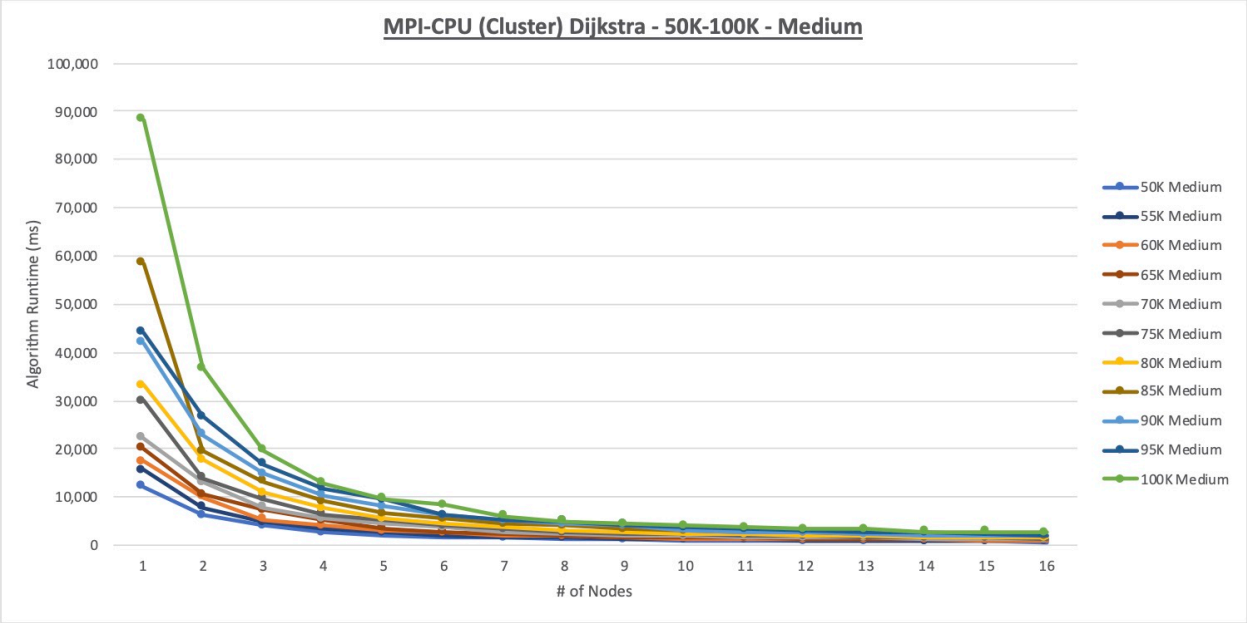

Figure B9. MPI-CPU (Cluster) Dijkstra 50K-100K Medium Runtime Chart

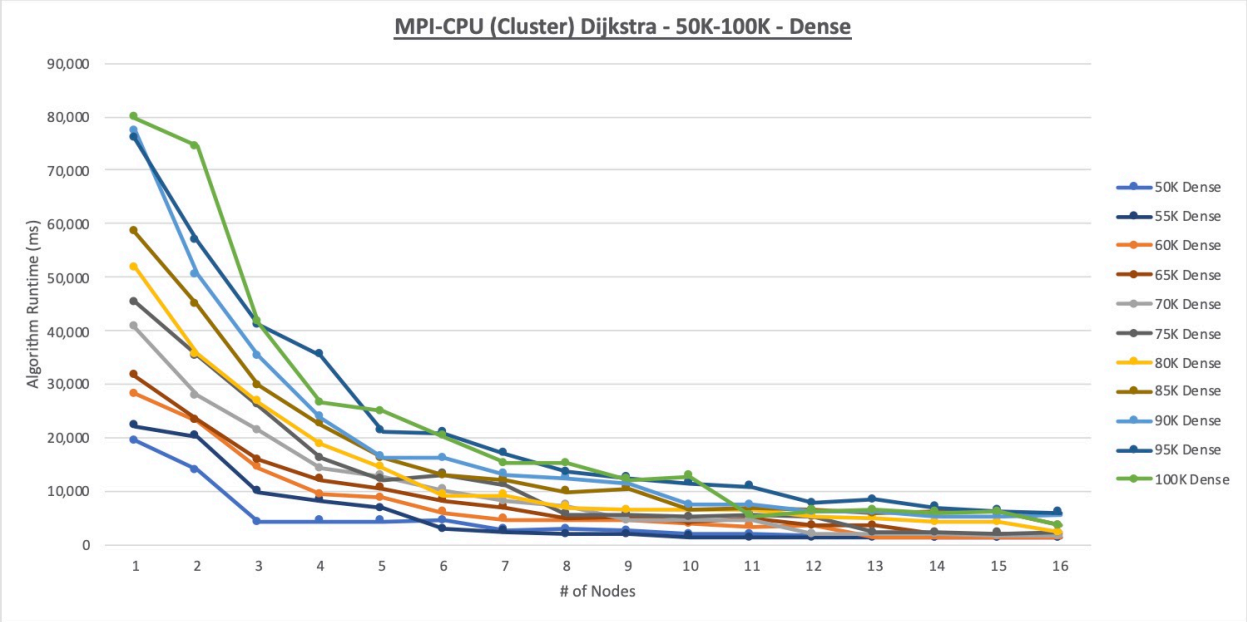

Figure B10. MPI-CPU (Cluster) Dijkstra 50K-100K Dense Runtime Chart

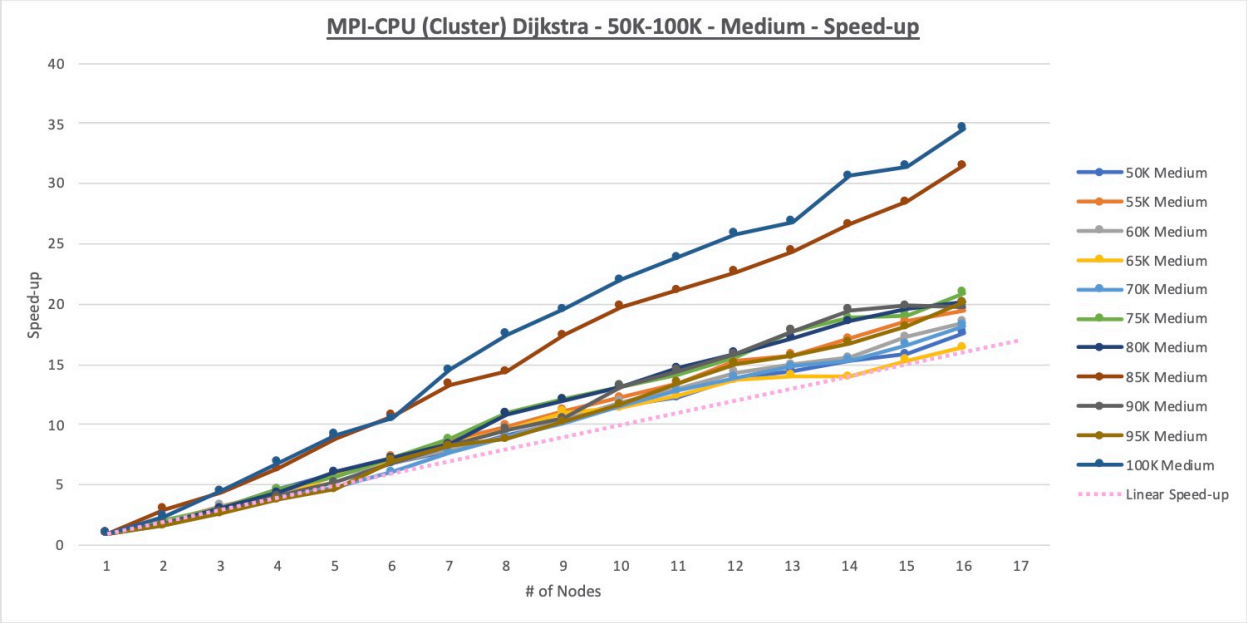

Figure B11. MPI-CPU (Cluster) Dijkstra 50K-100K Medium Speed-up Chart

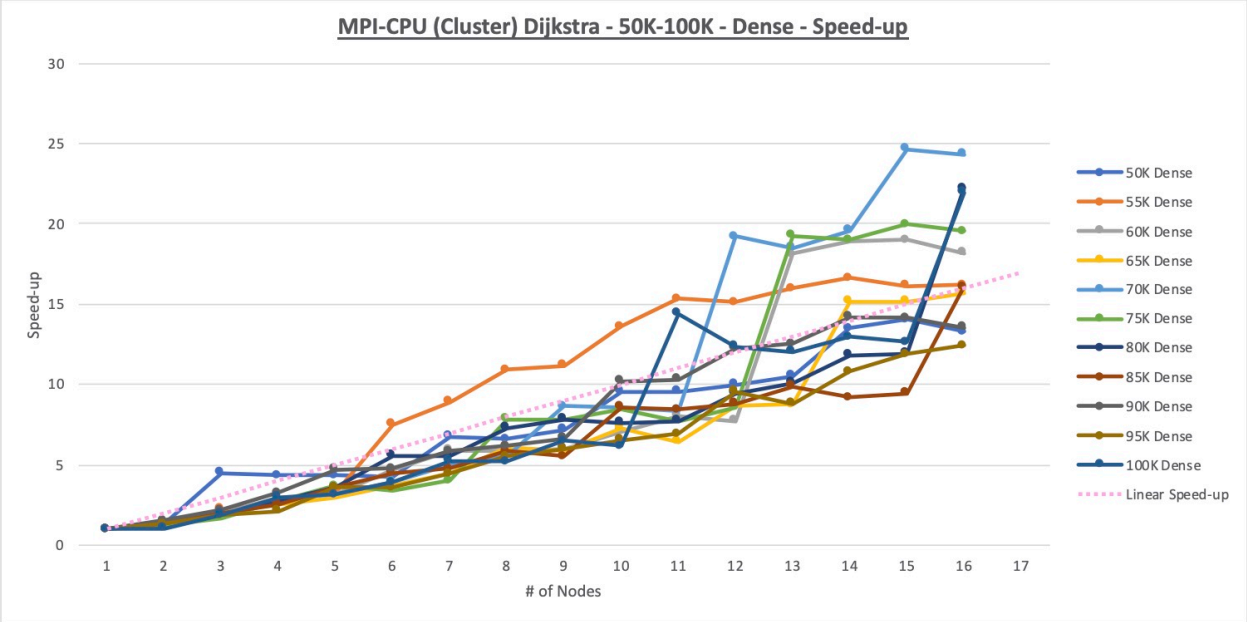

Figure B12. MPI-CPU (Cluster) Dijkstra 50K-100K Dense Speed-up Chart

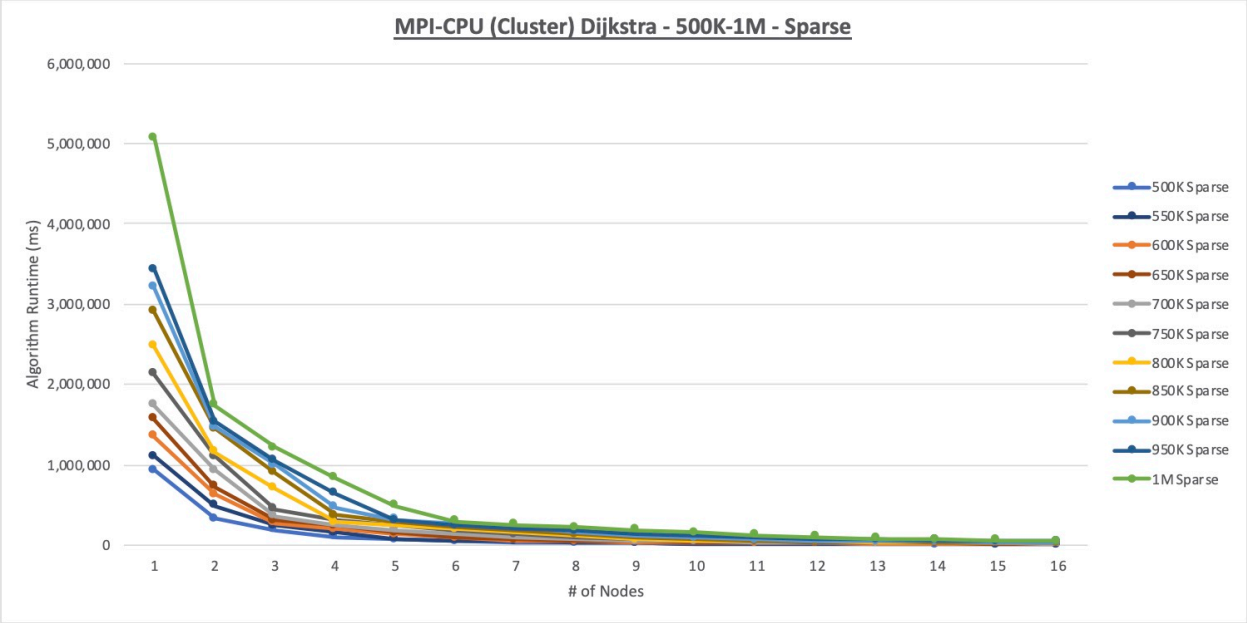

Figure B13. MPI-CPU (Cluster) Dijkstra 500K-1M Sparse Runtime Chart

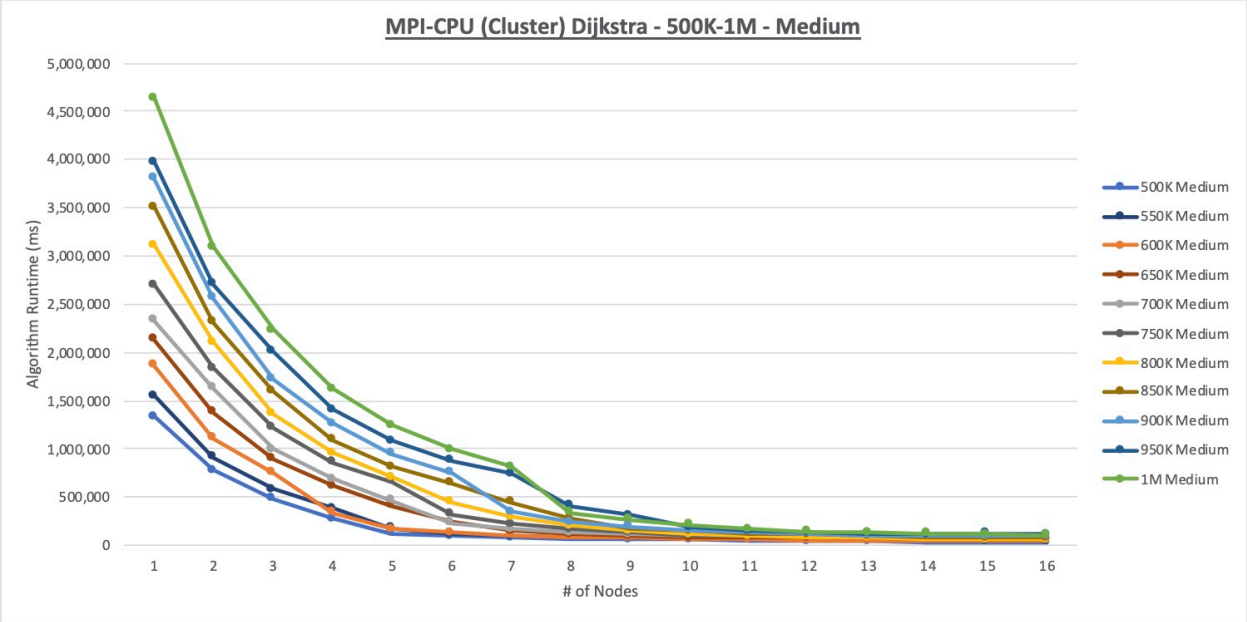

Figure B14. MPI-CPU (Cluster) Dijkstra 500K-1M Medium Runtime Chart

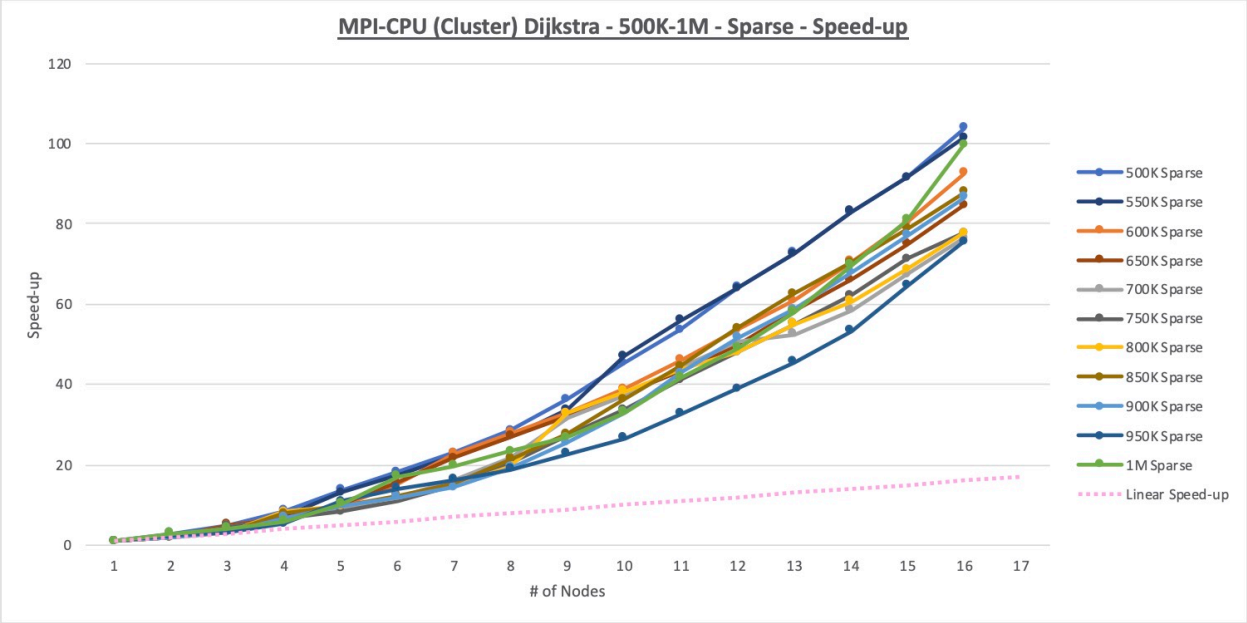

Figure B15. MPI-CPU (Cluster) Dijkstra 500K-1M Sparse Speed-up Chart

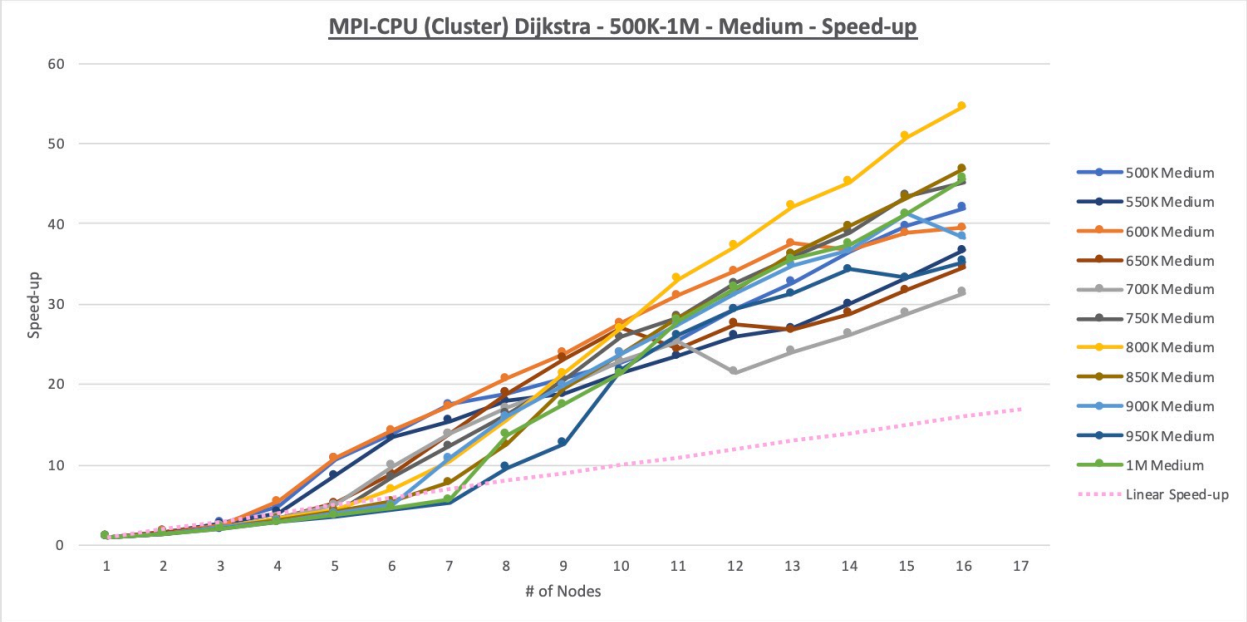

Figure B16. MPI-CPU (Cluster) Dijkstra 500K-1M Medium Speed-up Chart

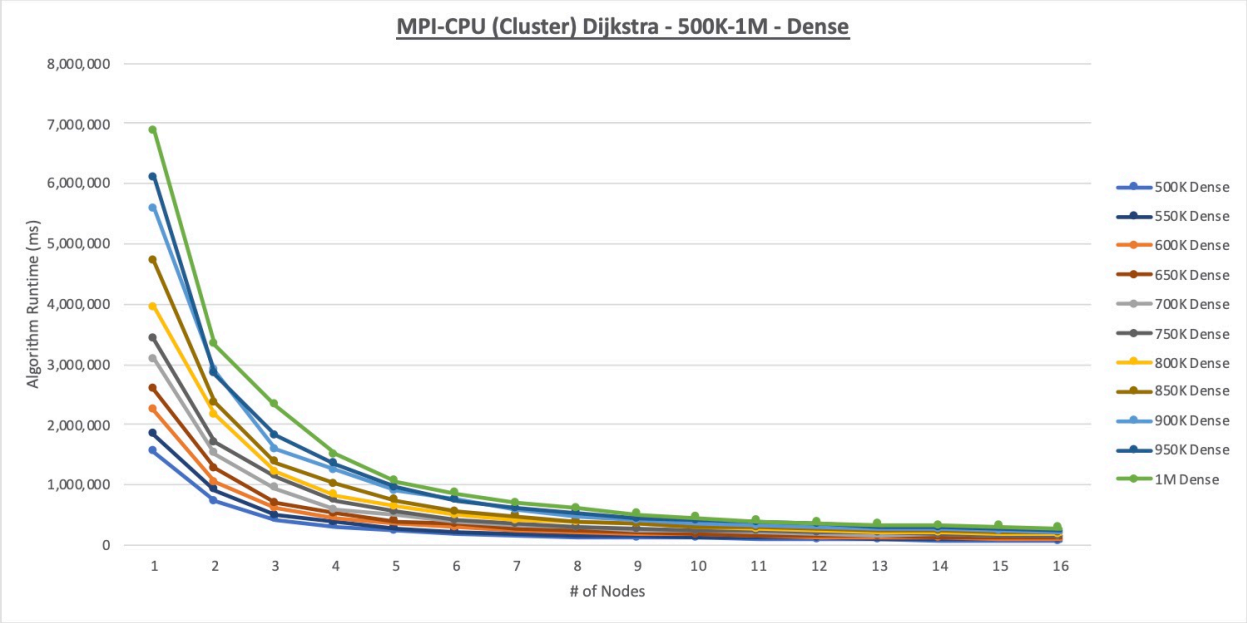

Figure B17. MPI-CPU (Cluster) Dijkstra 500K-1M Dense Runtime Chart

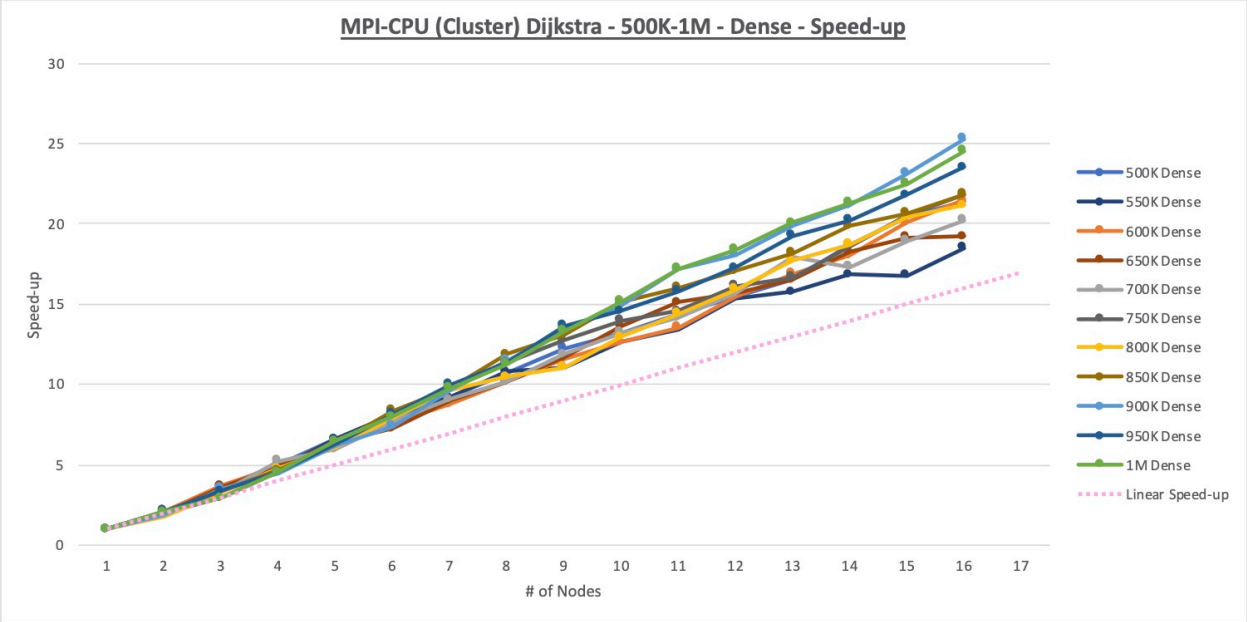

Figure B18. MPI-CPU (Cluster) Dijkstra 500K-1M Dense Speed-up Chart

APPENDIX C

MPI-CPU (SINGLE-NODE) DIJKSTRA DATA CHARTS

The runtime and speed-up charts for the MPI-CPU (Single-Node) Dijkstra experiments are presented as follows: Starting with the 5K-10K sparse graph set, the runtime charts for two graphs will be displayed on one page and then associated speed-up charts will be on the following page. For the ninth graph set (500K-1M dense), the runtime and speed-up charts will be on the same page.

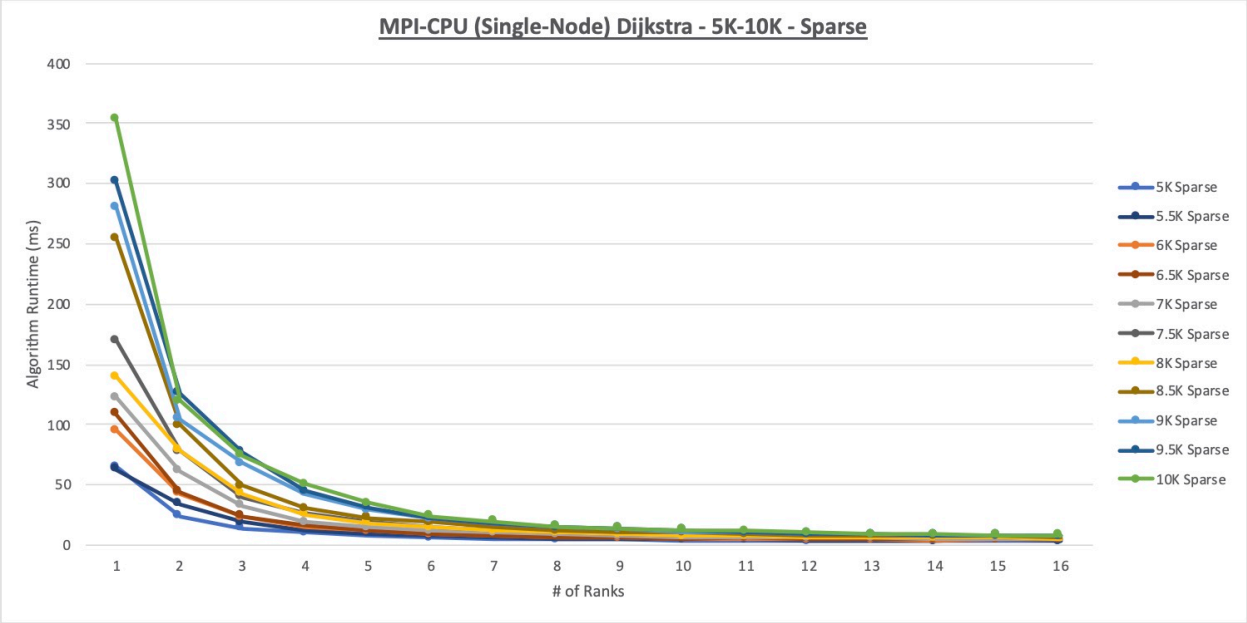

Figure C1. MPI-CPU (Single-Node) Dijkstra 5K-10K Sparse Runtime Chart

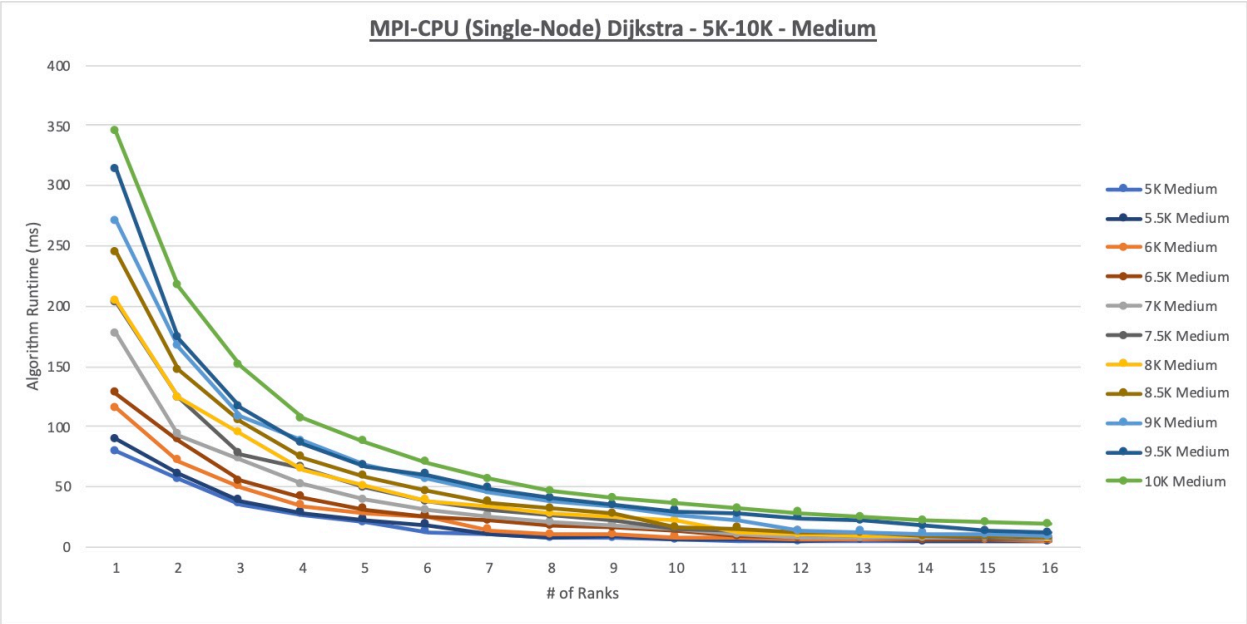

Figure C2. MPI-CPU (Single-Node) Dijkstra 5K-10K Medium Runtime Chart

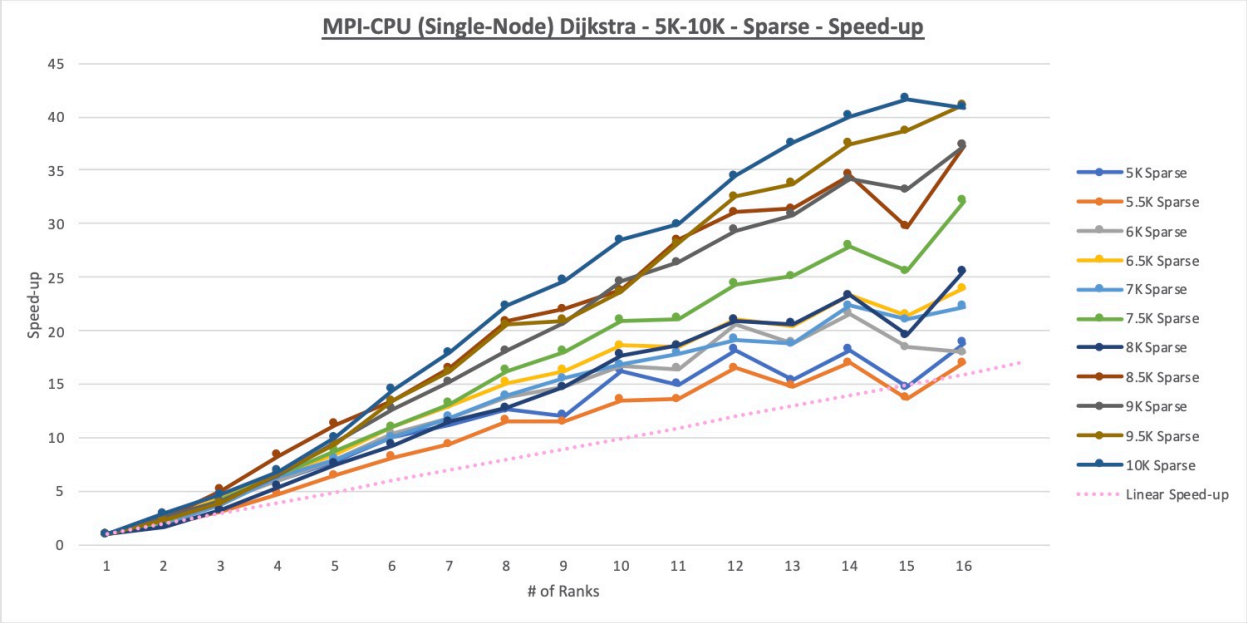

Figure C3. MPI-CPU (Single-Node) Dijkstra 5K-10K Sparse Speed-up Chart

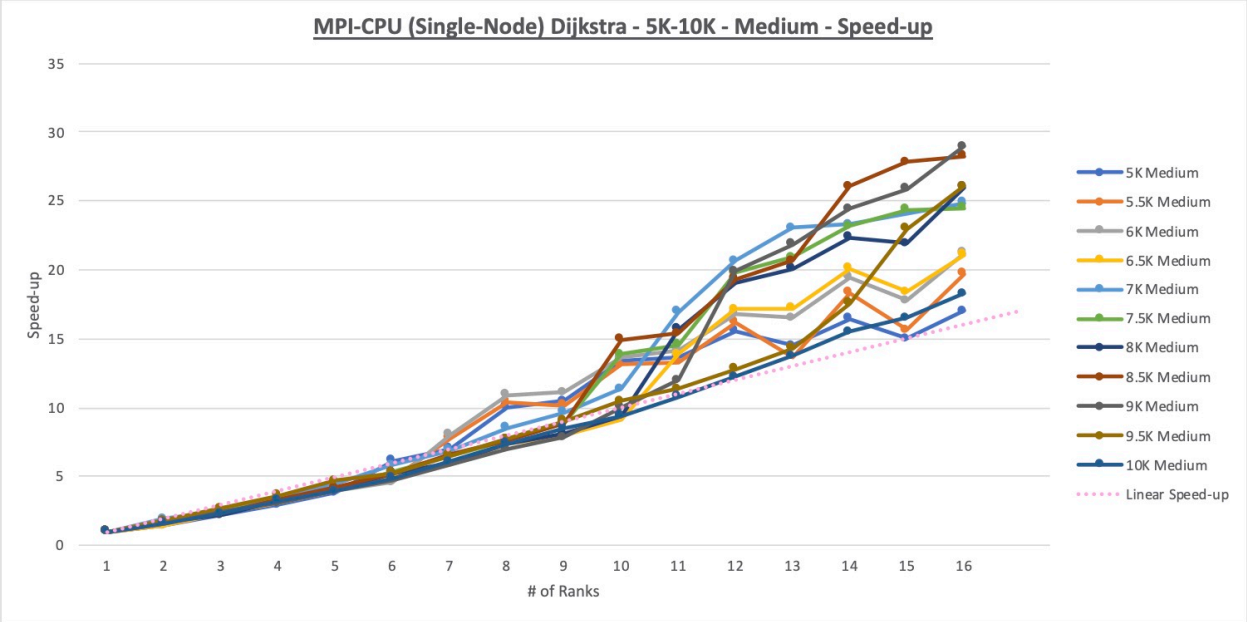

Figure C4. MPI-CPU (Single-Node) Dijkstra 5K-10K Medium Speed-up Chart

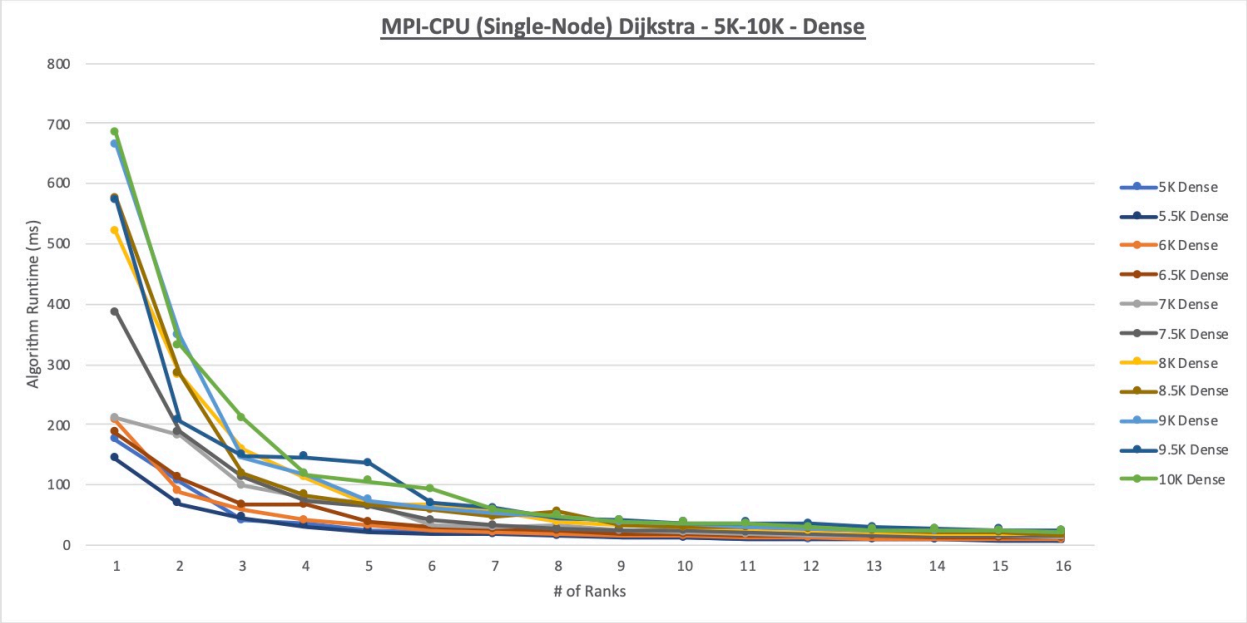

Figure C5. MPI-CPU (Single-Node) Dijkstra 5K-10K Dense Runtime Chart

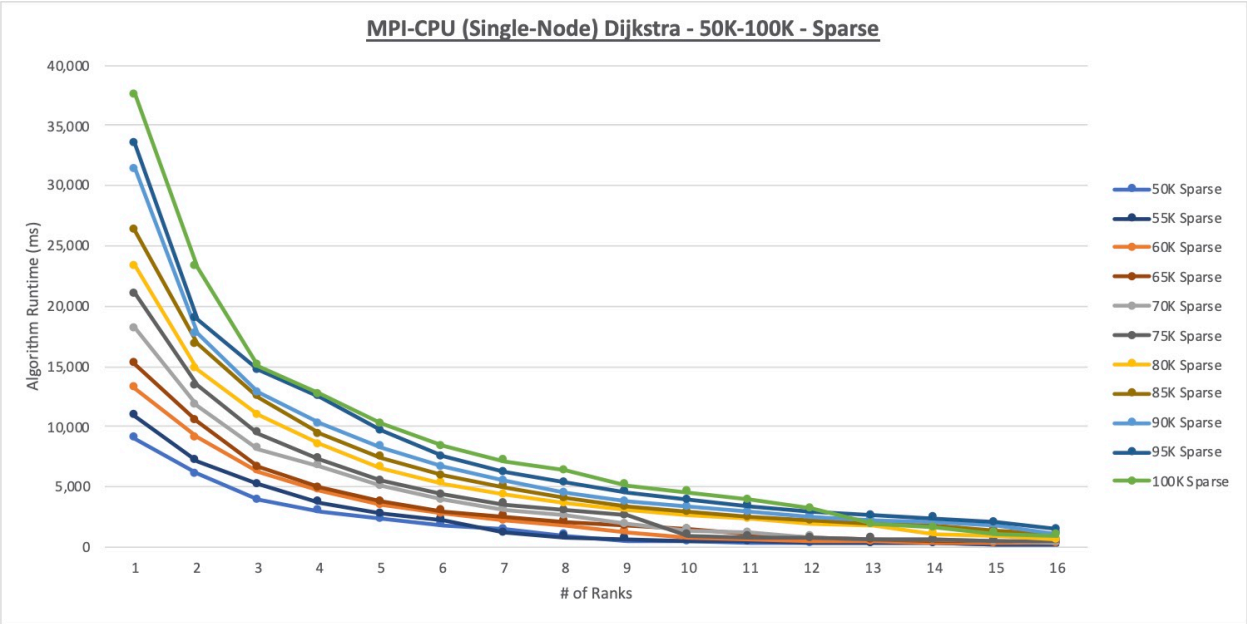

Figure C6. MPI-CPU (Single-Node) Dijkstra 50K-100K Sparse Runtime Chart

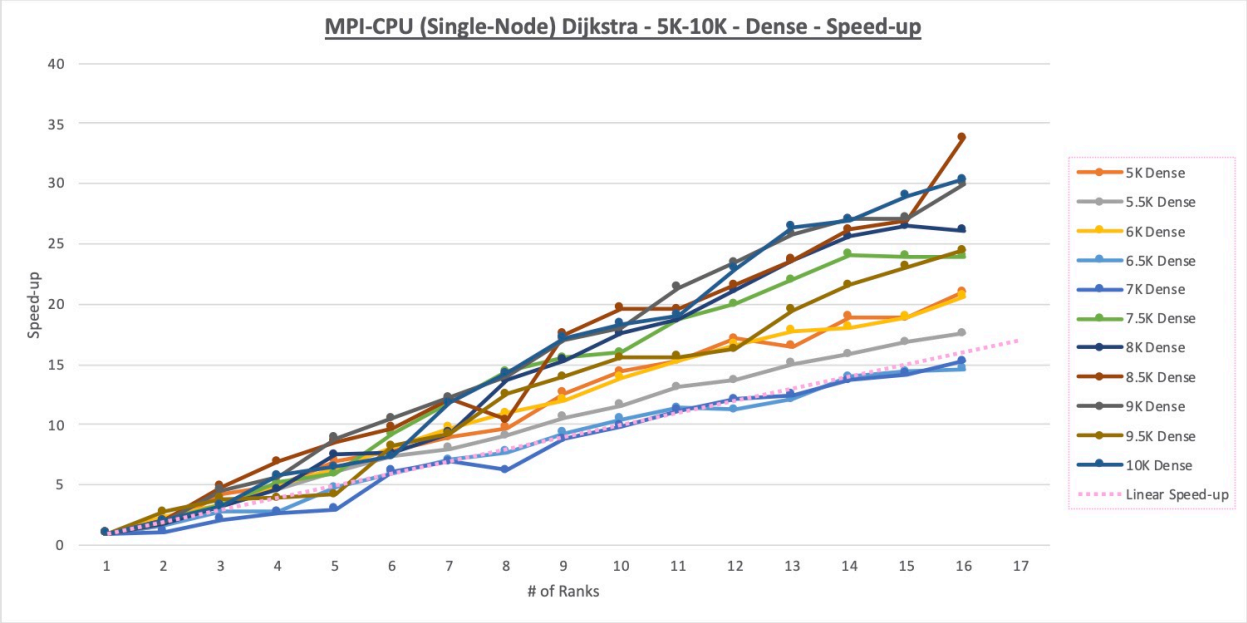

Figure C7. MPI-CPU (Single-Node) Dijkstra 5K-10K Dense Speed-up Chart

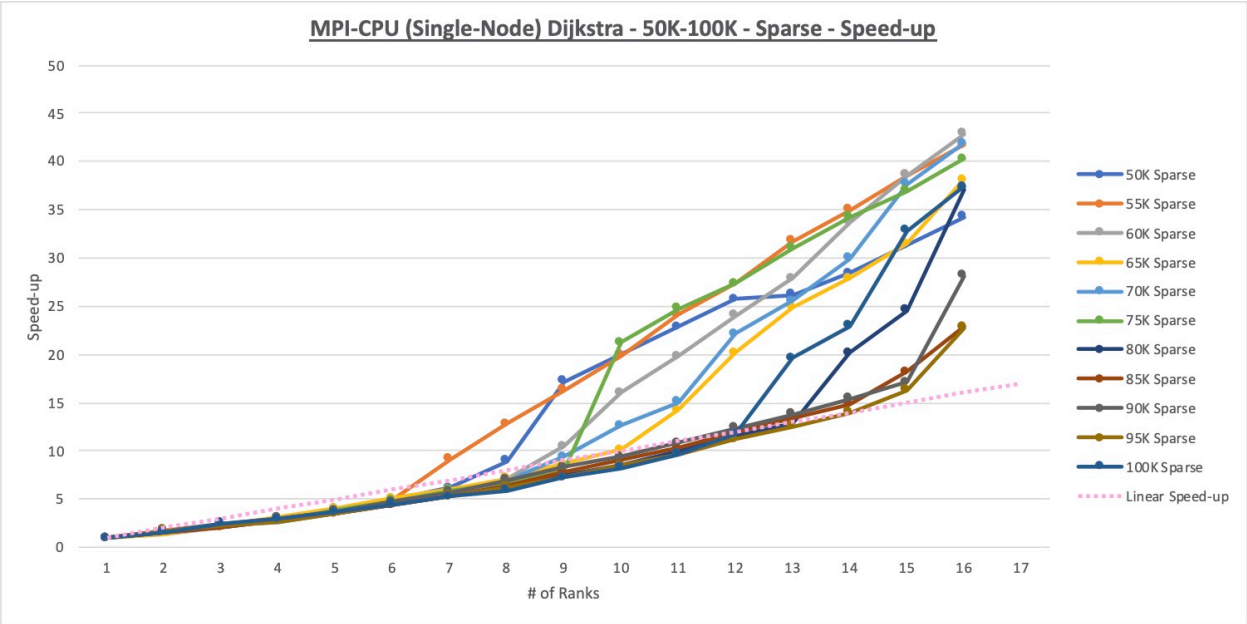

Figure C8. MPI-CPU (Single-Node) Dijkstra 50K-100K Sparse Speed-up Chart

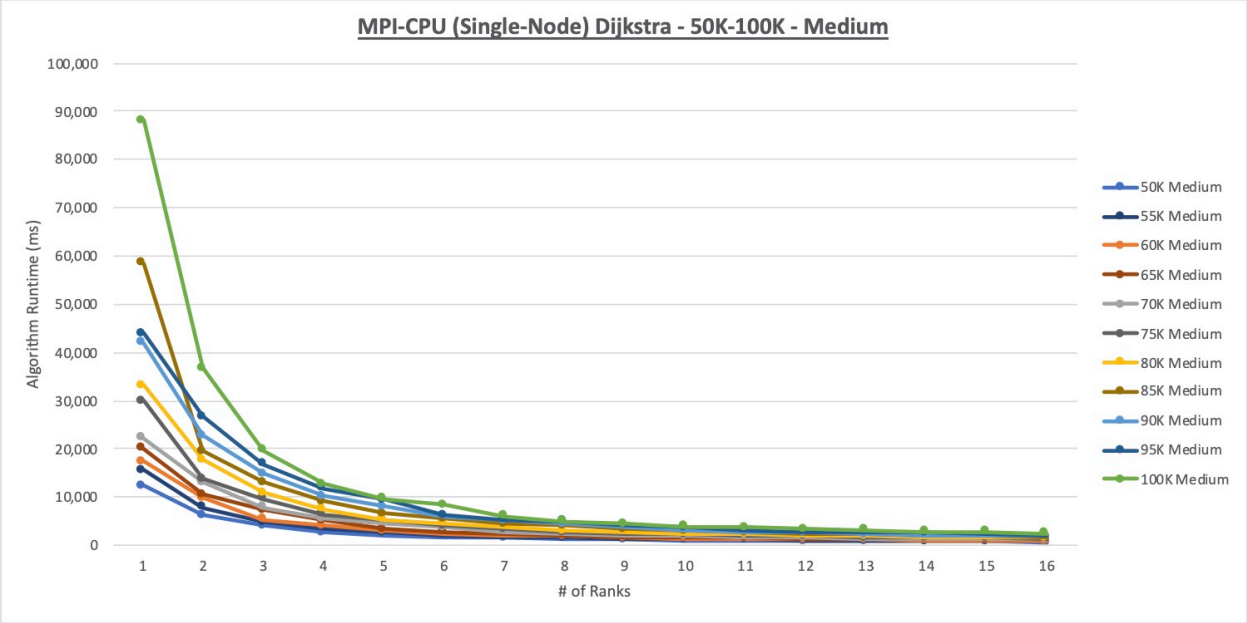

Figure C9. MPI-CPU (Single-Node) Dijkstra 50K-100K Medium Runtime Chart

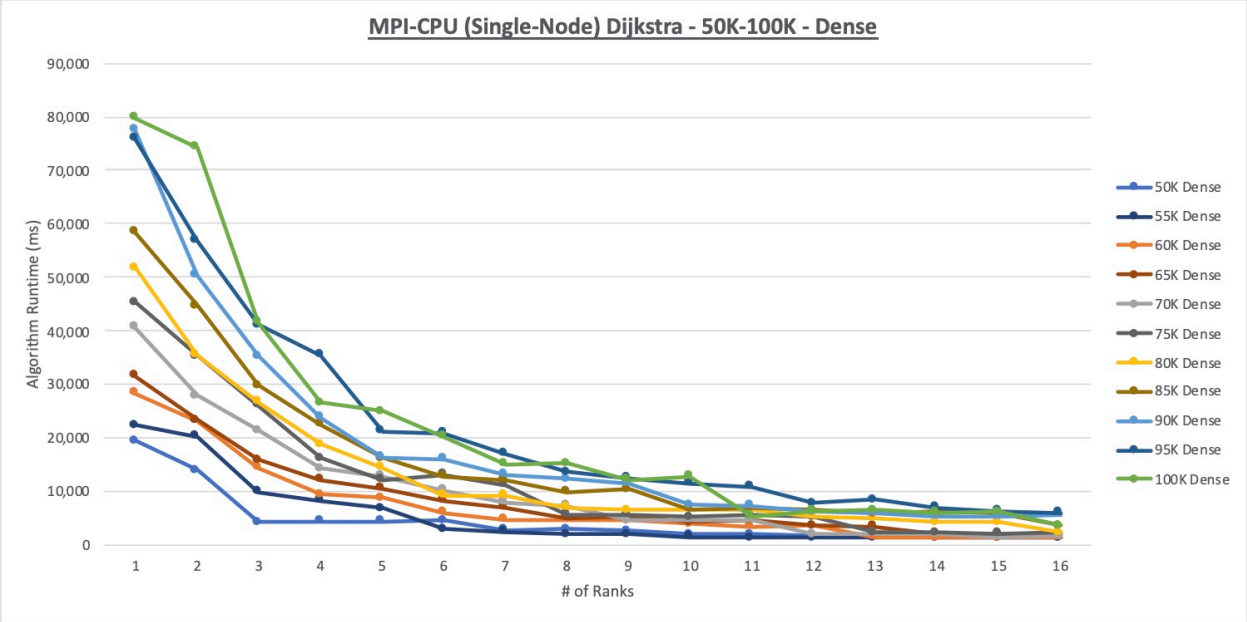

Figure C10. MPI-CPU (Single-Node) Dijkstra 50K-100K Dense Runtime Chart

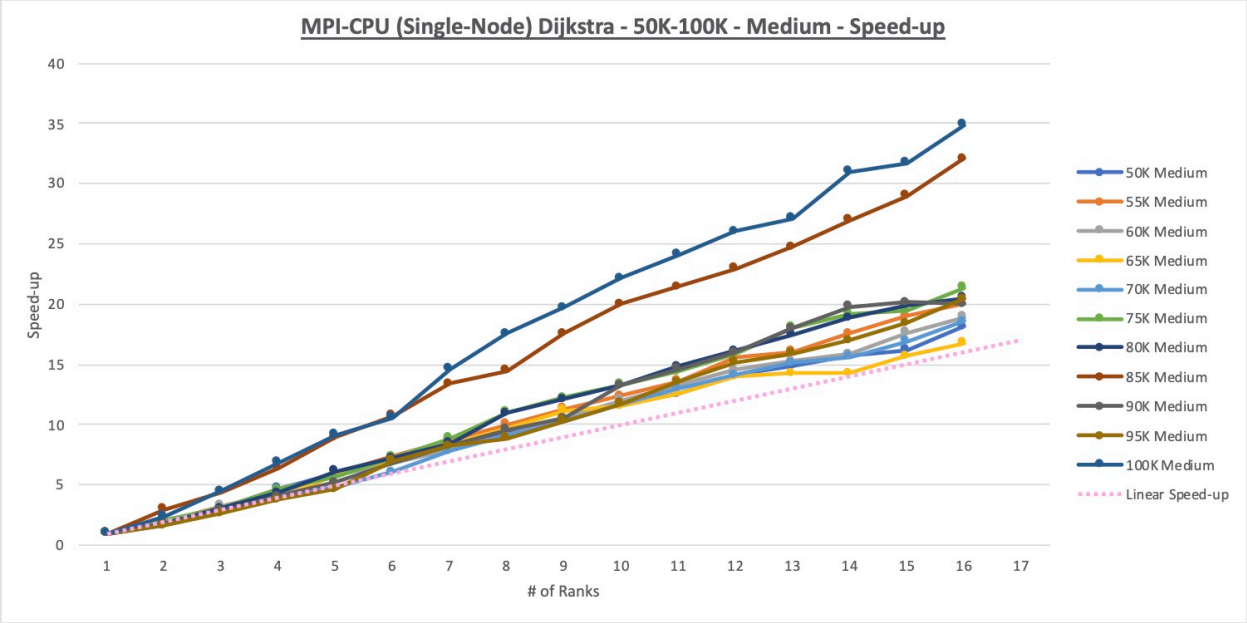

Figure C11. MPI-CPU (Single-Node) Dijkstra 50K-100K Medium Speed-up Chart

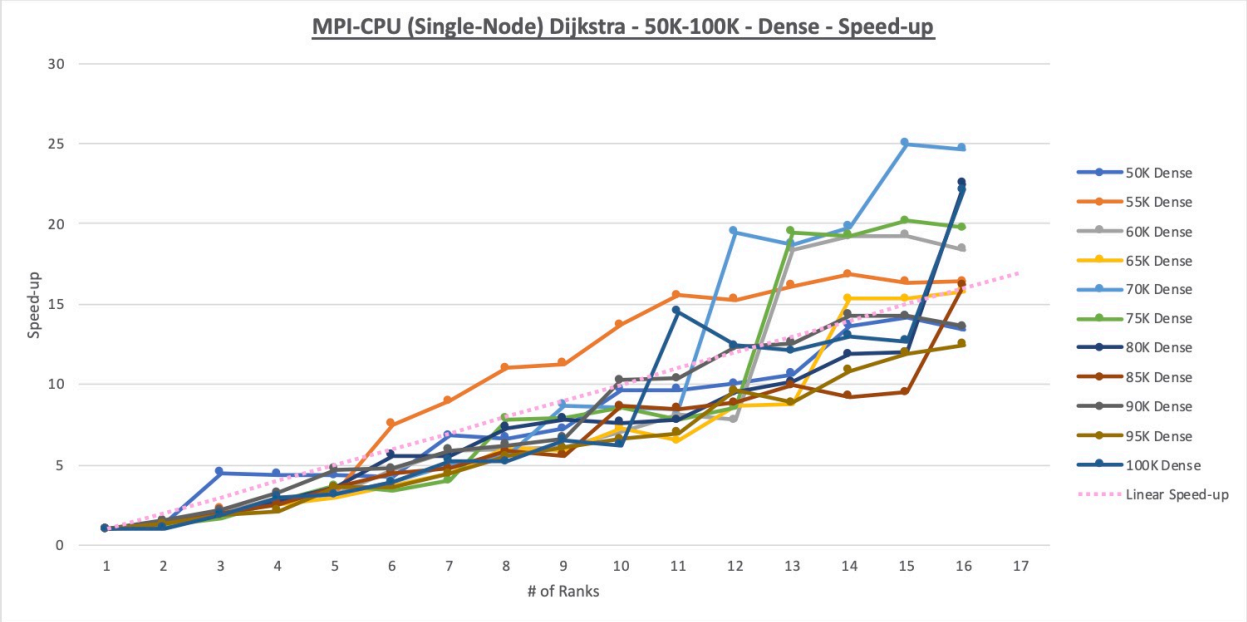

Figure C12. MPI-CPU (Single-Node) Dijkstra 50K-100K Dense Speed-up Chart

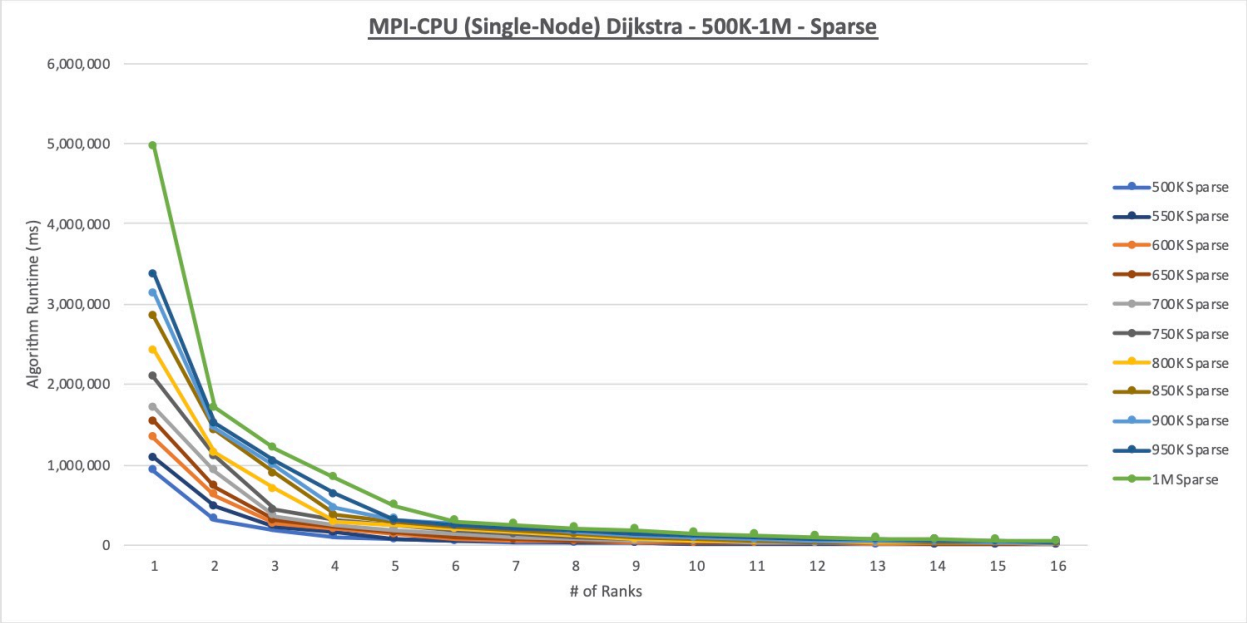

Figure C13. MPI-CPU (Single-Node) Dijkstra 500K-1M Sparse Runtime Chart

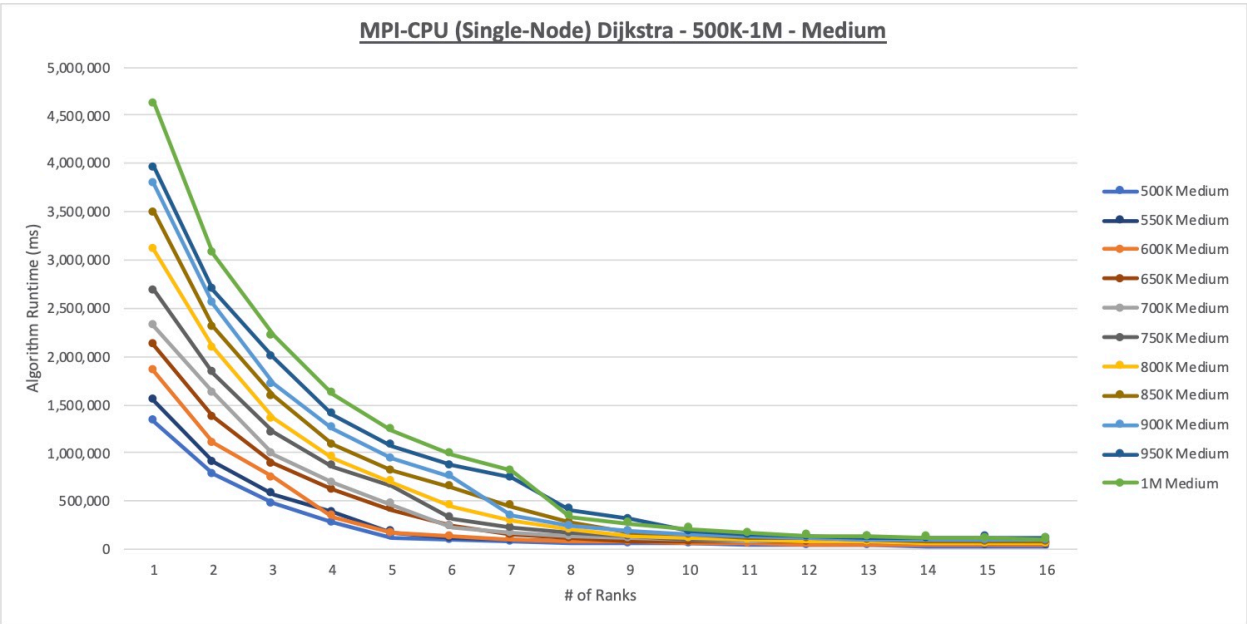

Figure C14. MPI-CPU (Single-Node) Dijkstra 500K-1M Medium Runtime Chart

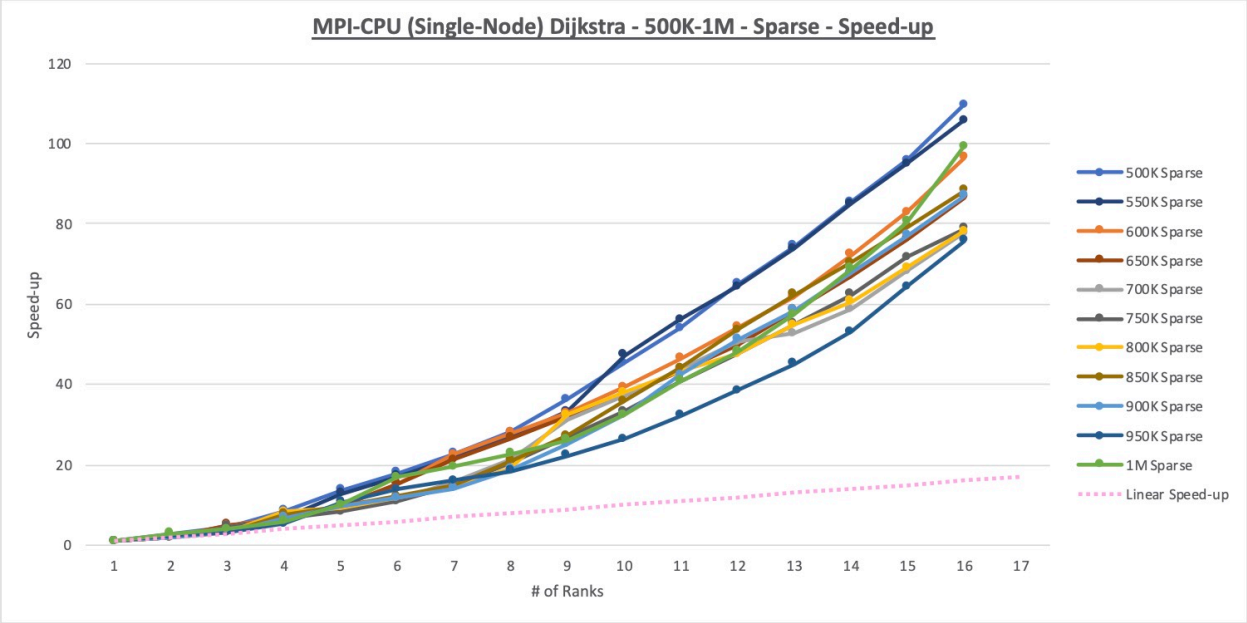

Figure C15. MPI-CPU (Single-Node) Dijkstra 500K-1M Sparse Speed-up Chart

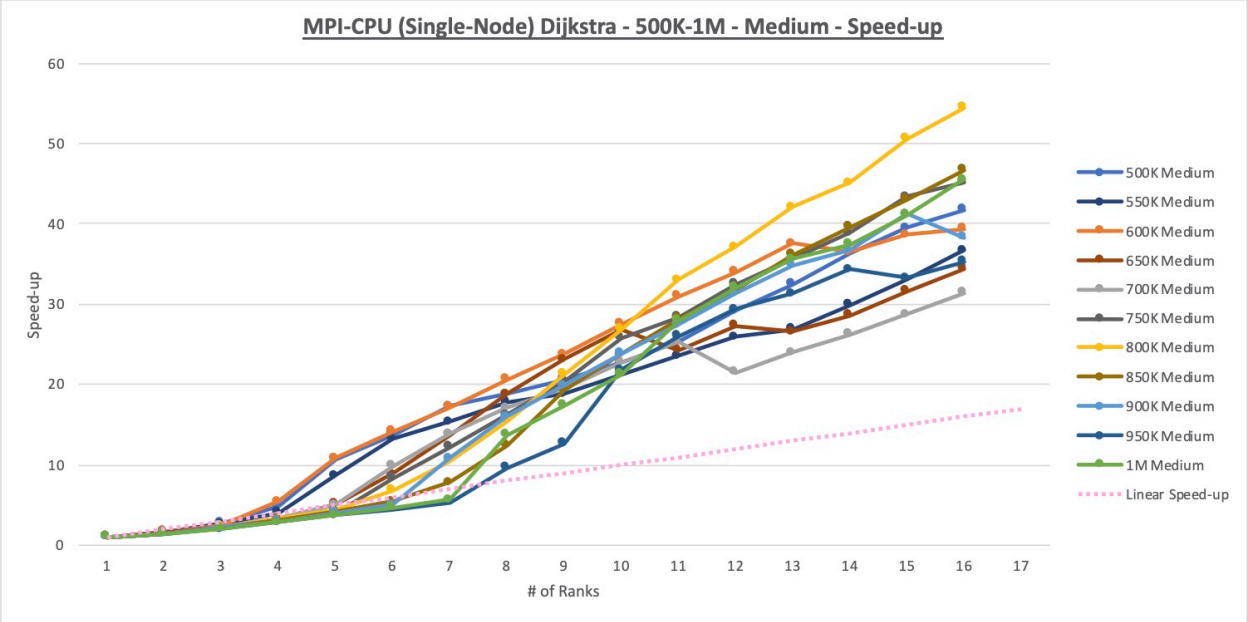

Figure C16. MPI-CPU (Single-Node) Dijkstra 500K-1M Medium Speed-up Chart

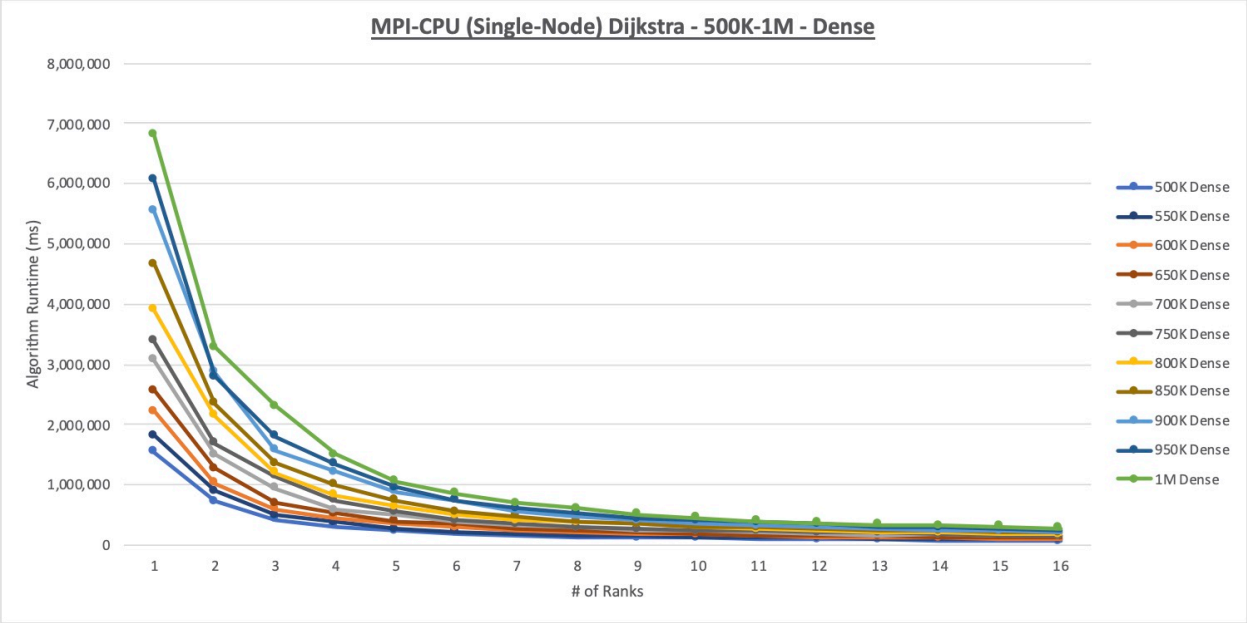

Figure C17. MPI-CPU (Single-Node) Dijkstra 500K-1M Dense Runtime Chart

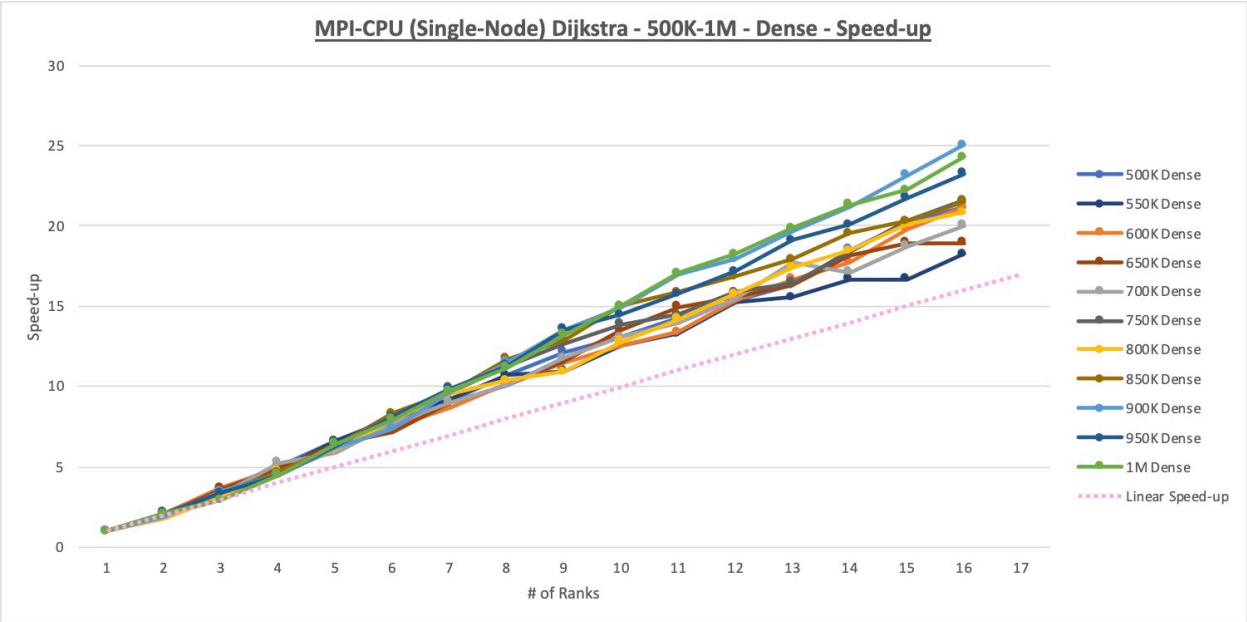

Figure C18. MPI-CPU (Single-Node) Dijkstra 500K-1M Dense Speed-up Chart

APPENDIX D

MPI-CPU (CLUSTER) BELLMAN-FORD DATA CHARTS

The runtime and speed-up charts for the MPI-CPU (Cluster) Bellman-Ford experiments are presented as follows: Starting with the 5K-10K sparse graph set, the runtime charts for two graphs will be displayed on one page and then associated speed-up charts will be on the following page. For the ninth graph set (500K-1M dense), the runtime and speed-up charts will be on the same page.

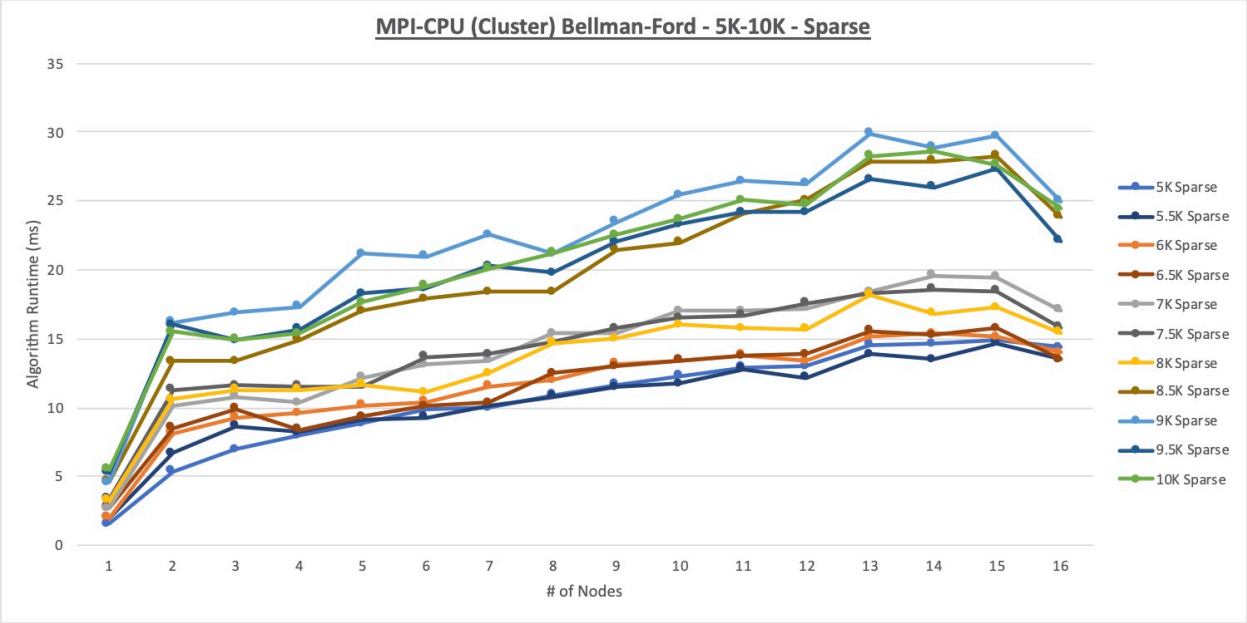

Figure D1. MPI-CPU (Cluster) Bellman-Ford 5K-10K Sparse Runtime Chart

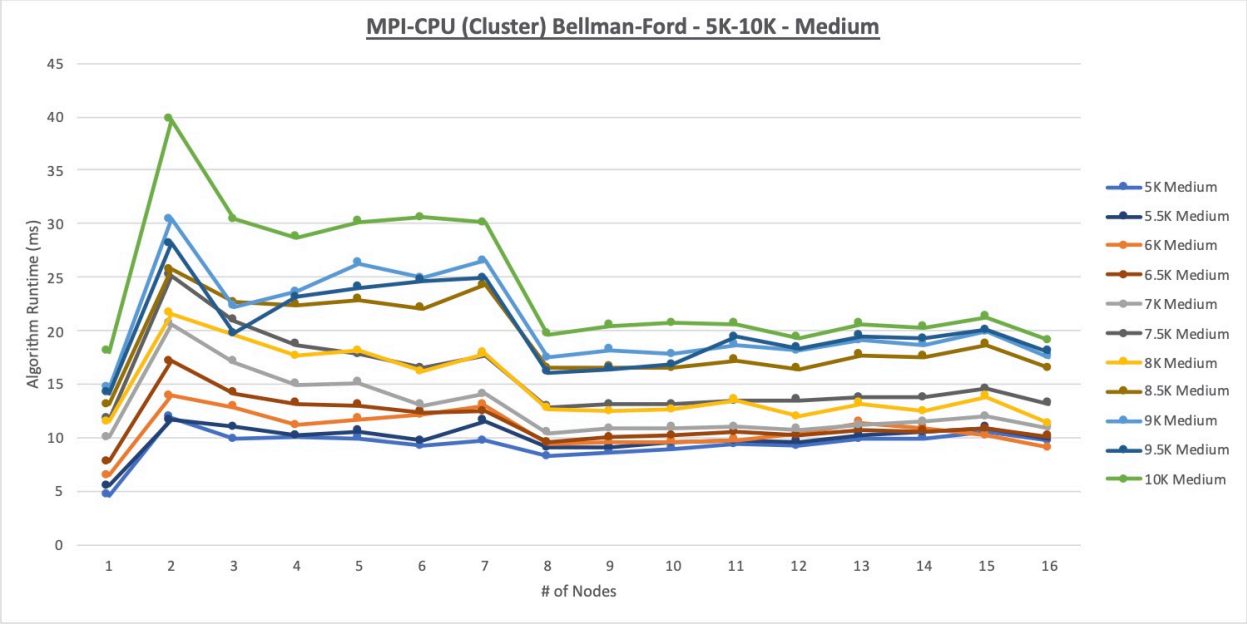

Figure D2. MPI-CPU (Cluster) Bellman-Ford 5K-10K Medium Runtime Chart

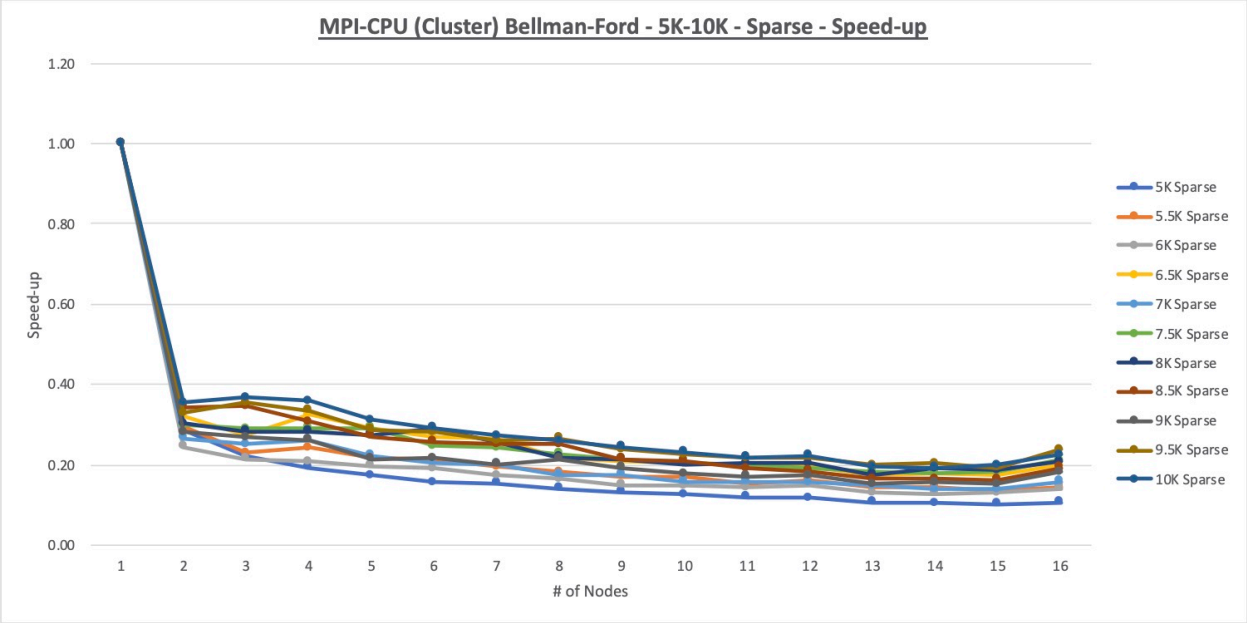

Figure D3. MPI-CPU (Cluster) Bellman-Ford 5K-10K Sparse Speed-up Chart

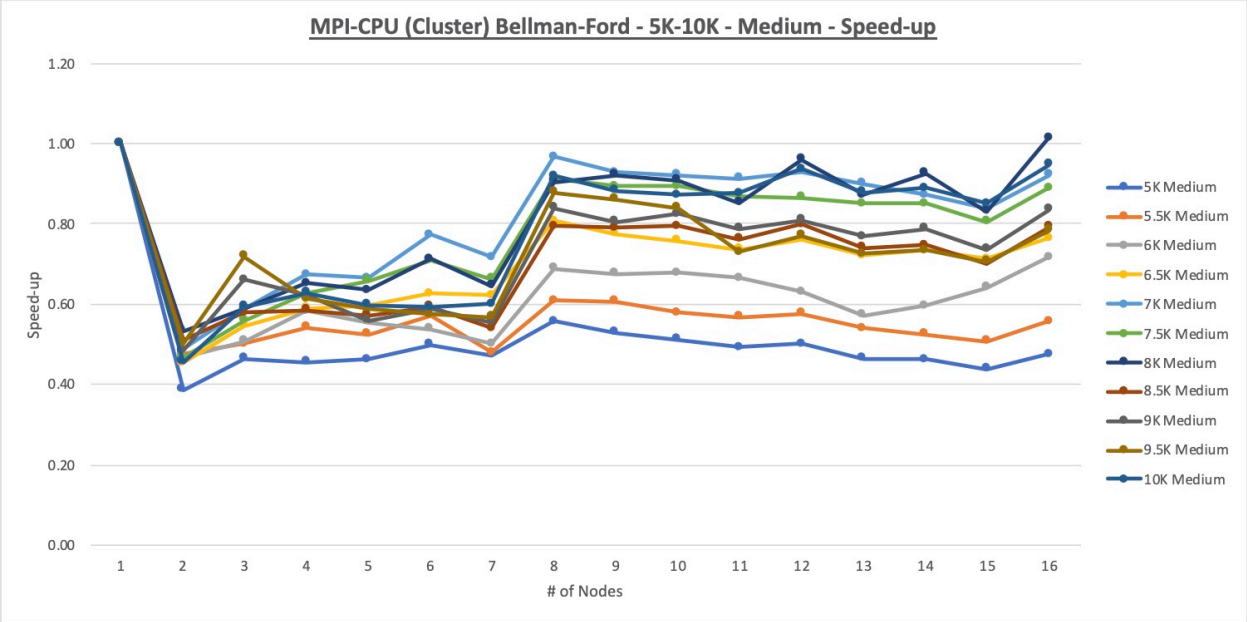

Figure D4. MPI-CPU (Cluster) Bellman-Ford 5K-10K Medium Speed-up Chart

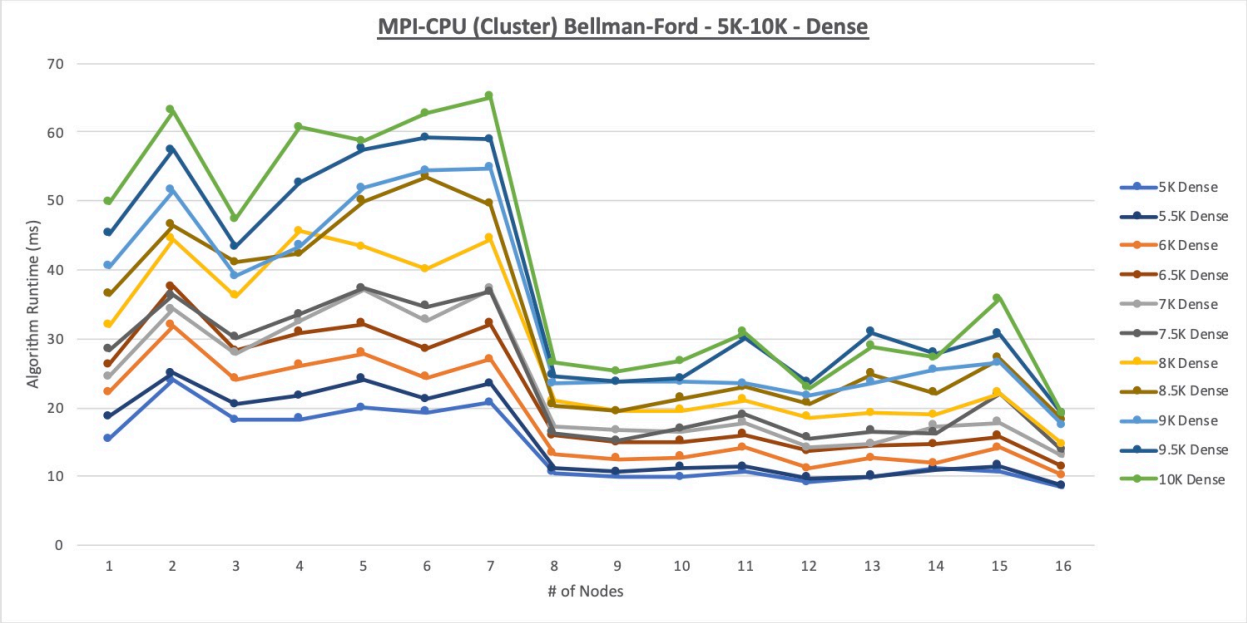

Figure D5. MPI-CPU (Cluster) Bellman-Ford 5K-10K Dense Runtime Chart

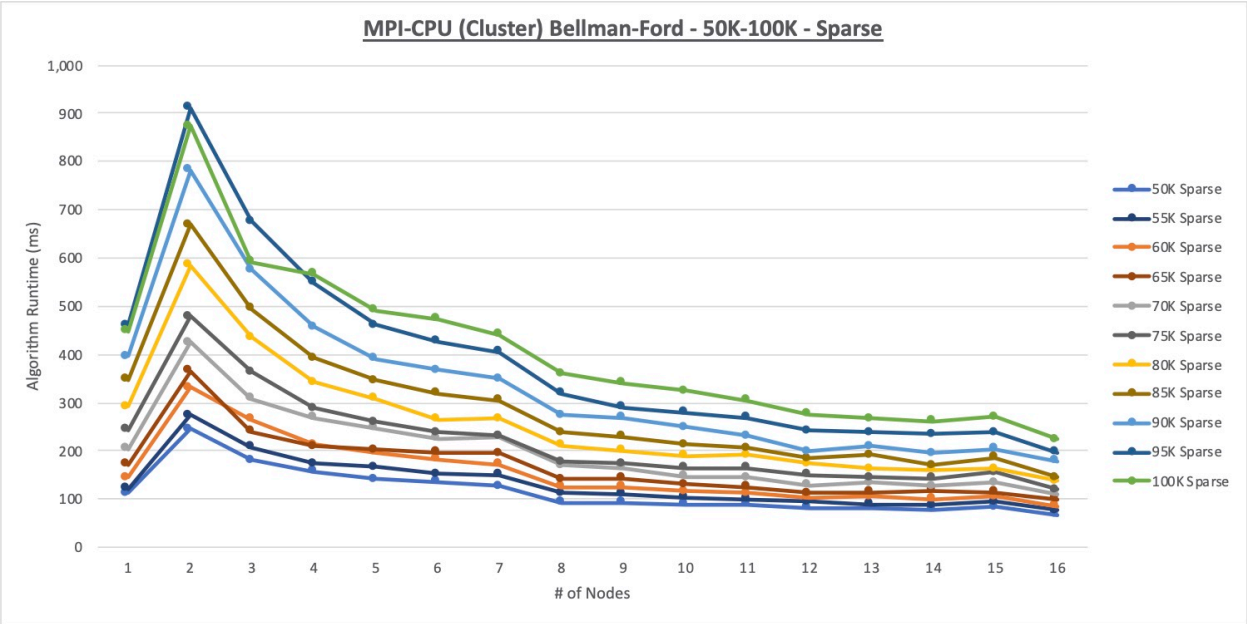

Figure D6. MPI-CPU (Cluster) Bellman-Ford 50K-100K Sparse Runtime Chart

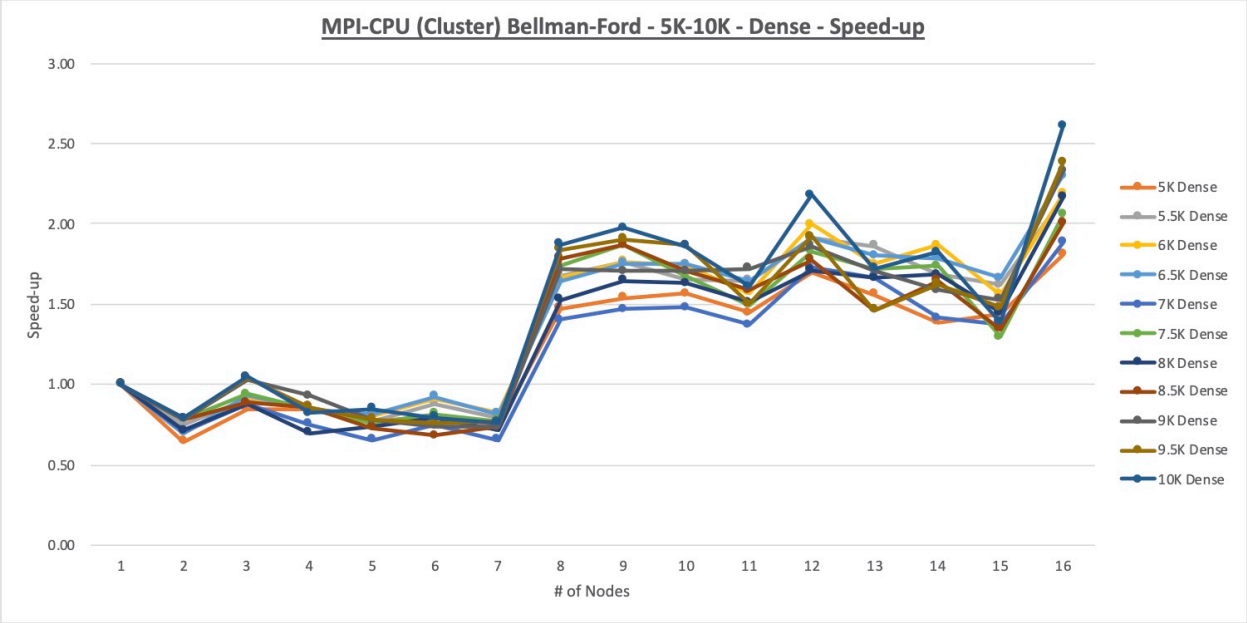

Figure D7. MPI-CPU (Cluster) Bellman-Ford 5K-10K Dense Speed-up Chart

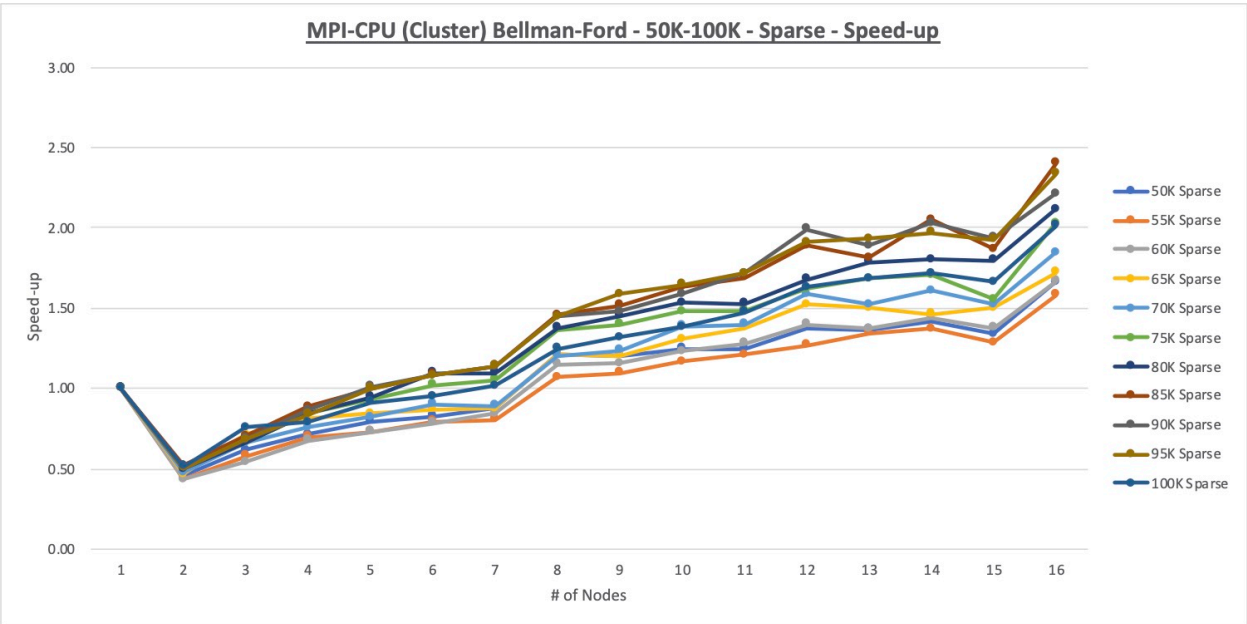

Figure D8. MPI-CPU (Cluster) Bellman-Ford 50K-100K Sparse Speed-up Chart

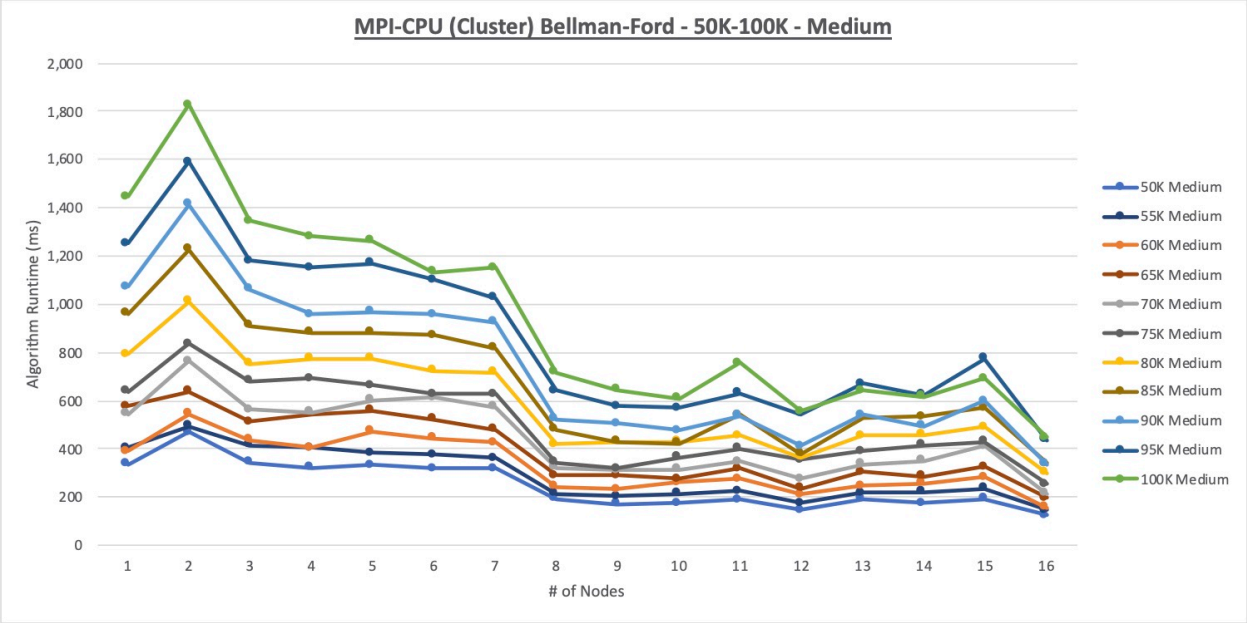

Figure D9. MPI-CPU (Cluster) Bellman-Ford 50K-100K Medium Runtime Chart

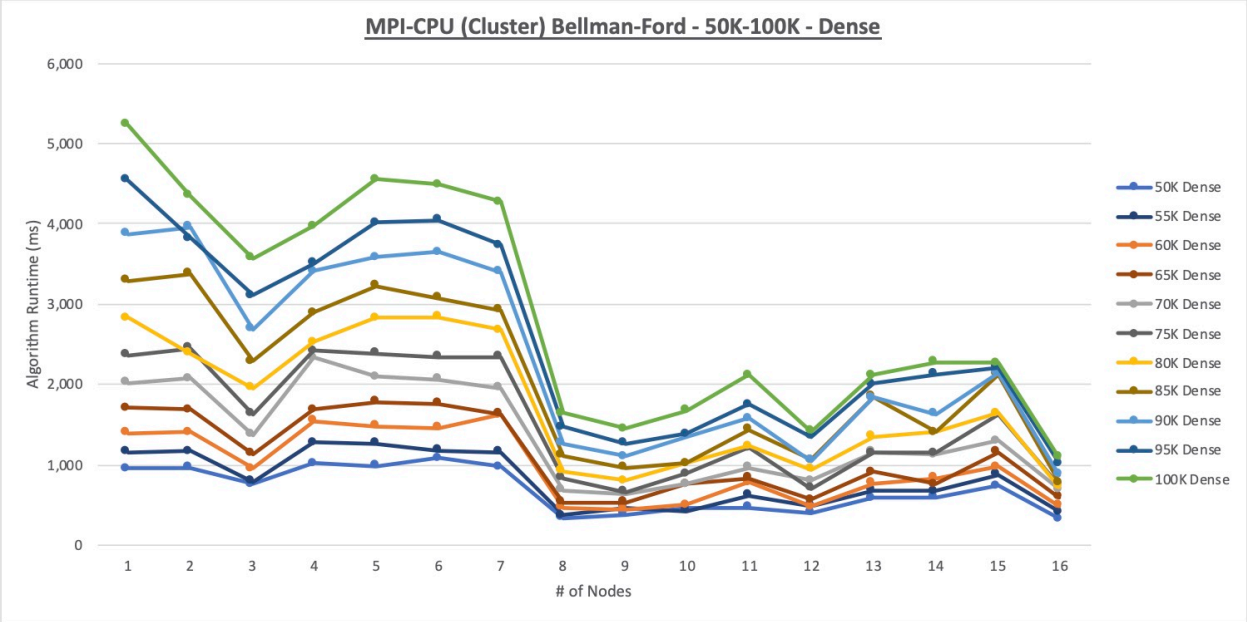

Figure D10. MPI-CPU (Cluster) Bellman-Ford 50K-100K Dense Runtime Chart

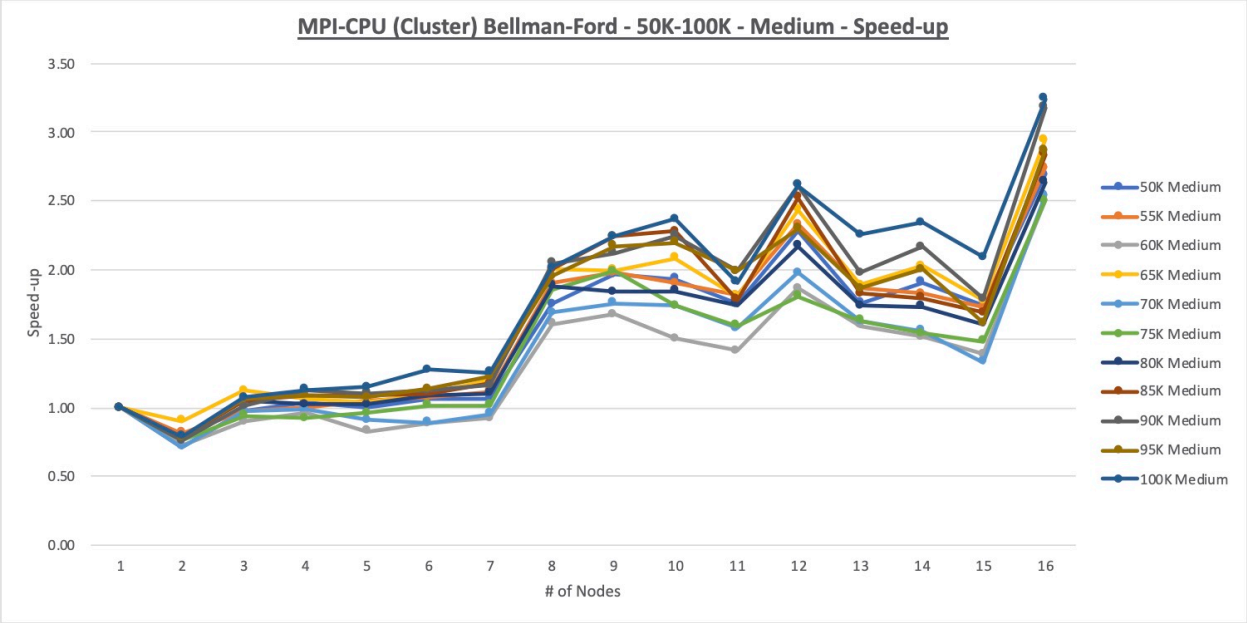

Figure D11. MPI-CPU (Cluster) Bellman-Ford 50K-100K Medium Speed-up Chart

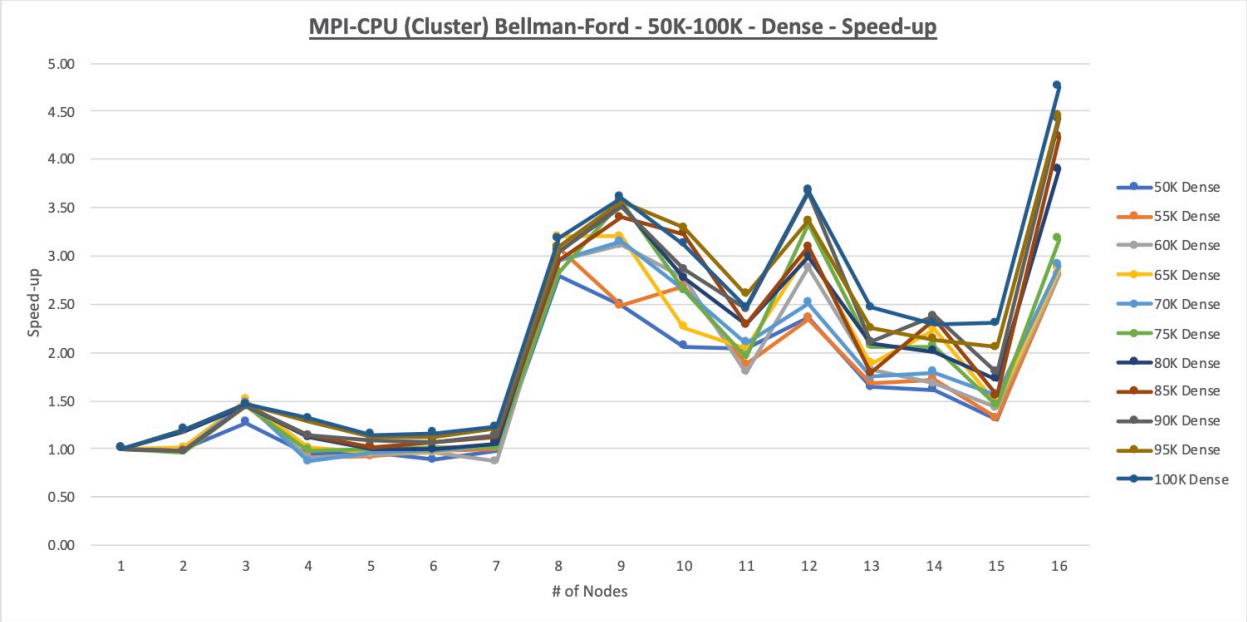

Figure D12. MPI-CPU (Cluster) Bellman-Ford 50K-100K Dense Speed-up Chart

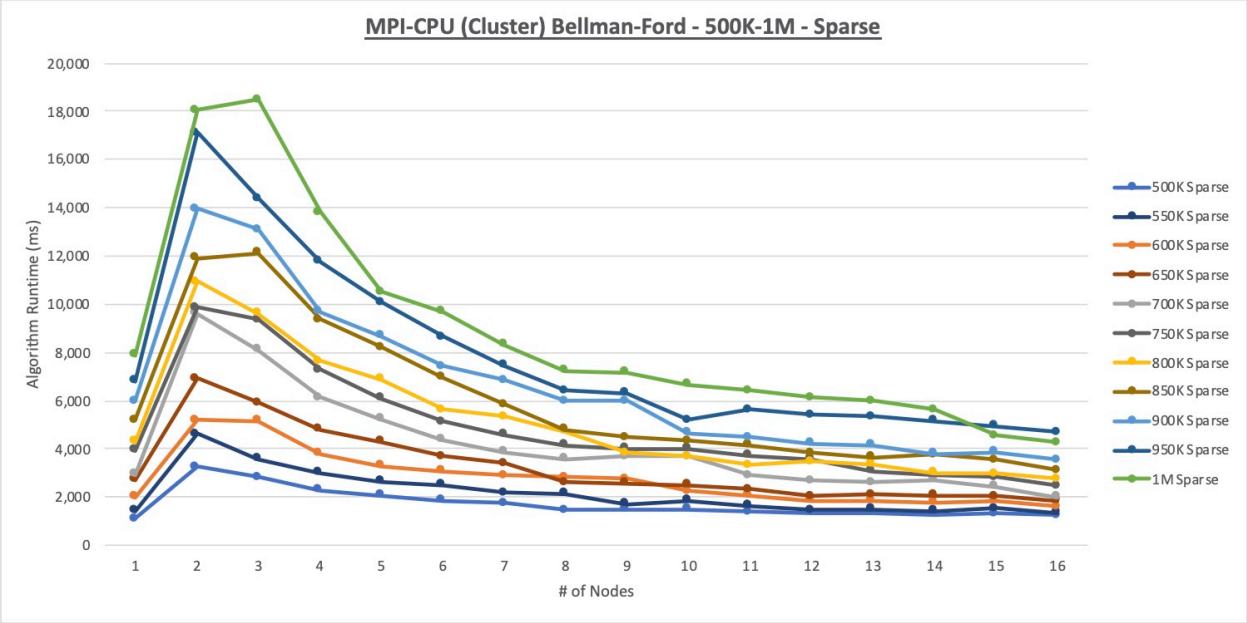

Figure D13. MPI-CPU (Cluster) Bellman-Ford 500K-1M Sparse Runtime Chart

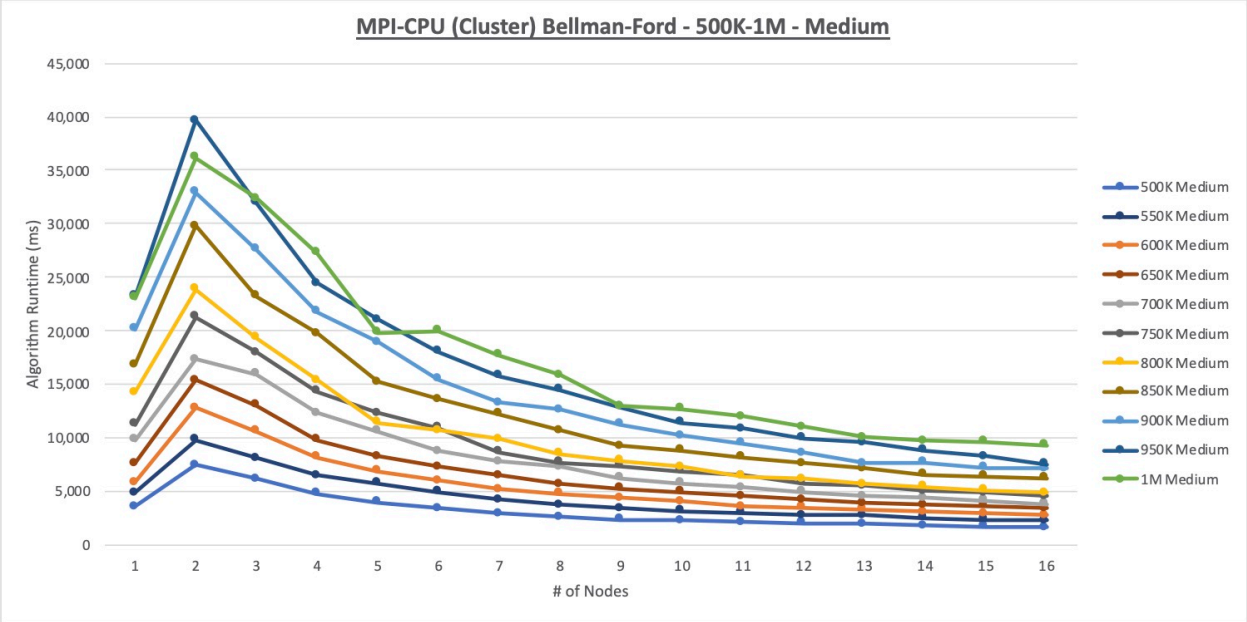

Figure D14. MPI-CPU (Cluster) Bellman-Ford 500K-1M Medium Runtime Chart

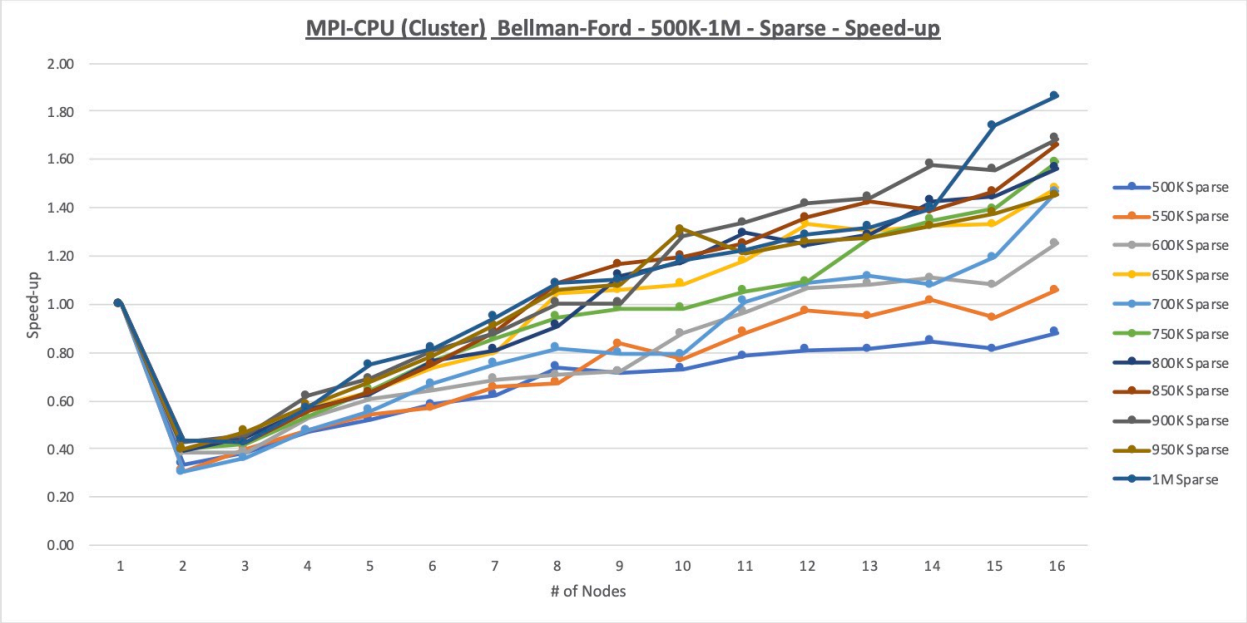

Figure D15. MPI-CPU (Cluster) Bellman-Ford 500K-1M Sparse Speed-up Chart

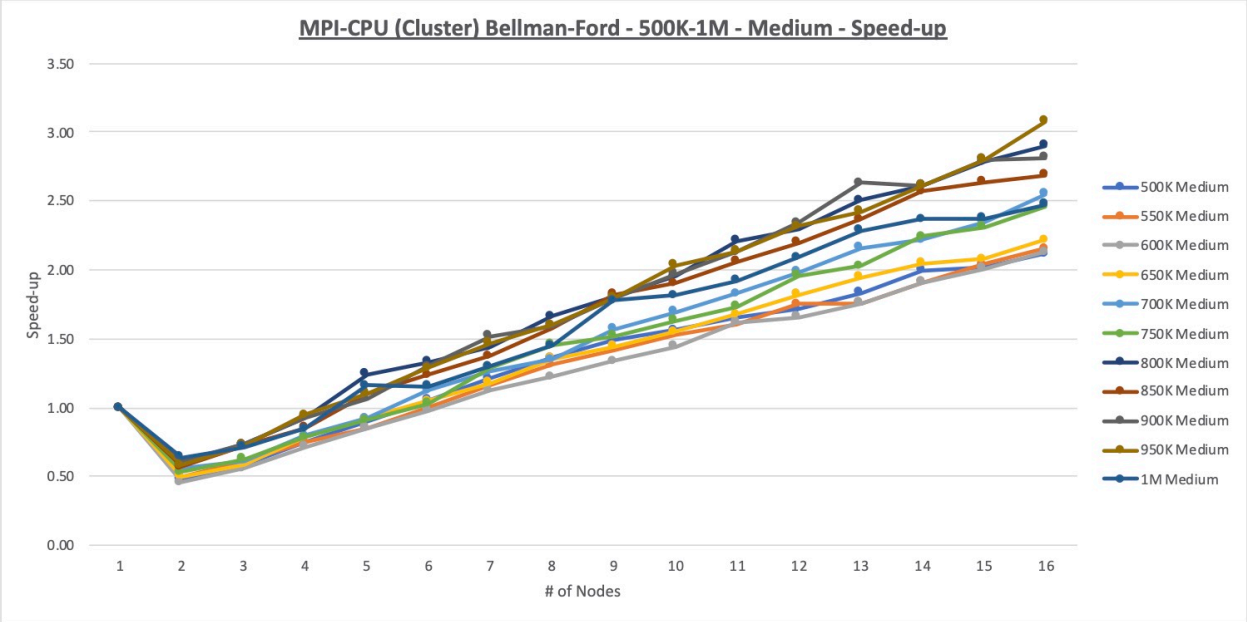

Figure D16. MPI-CPU (Cluster) Bellman-Ford 500K-1M Medium Speed-up Chart

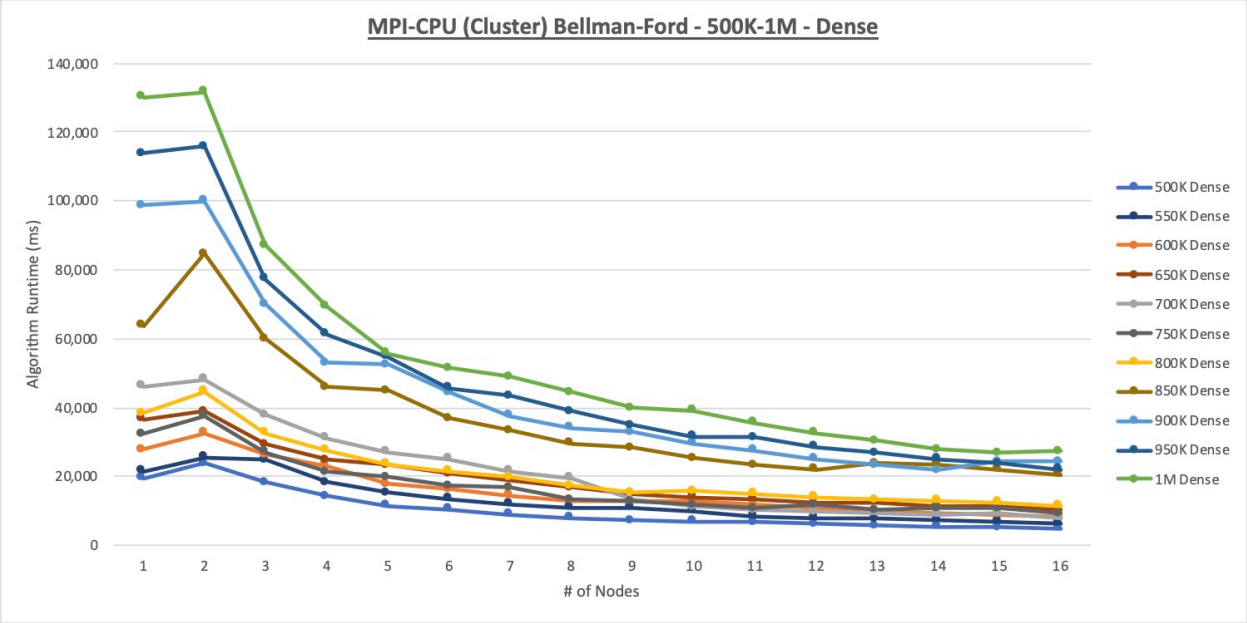

Figure D17. MPI-CPU (Cluster) Bellman-Ford 500K-1M Dense Runtime Chart

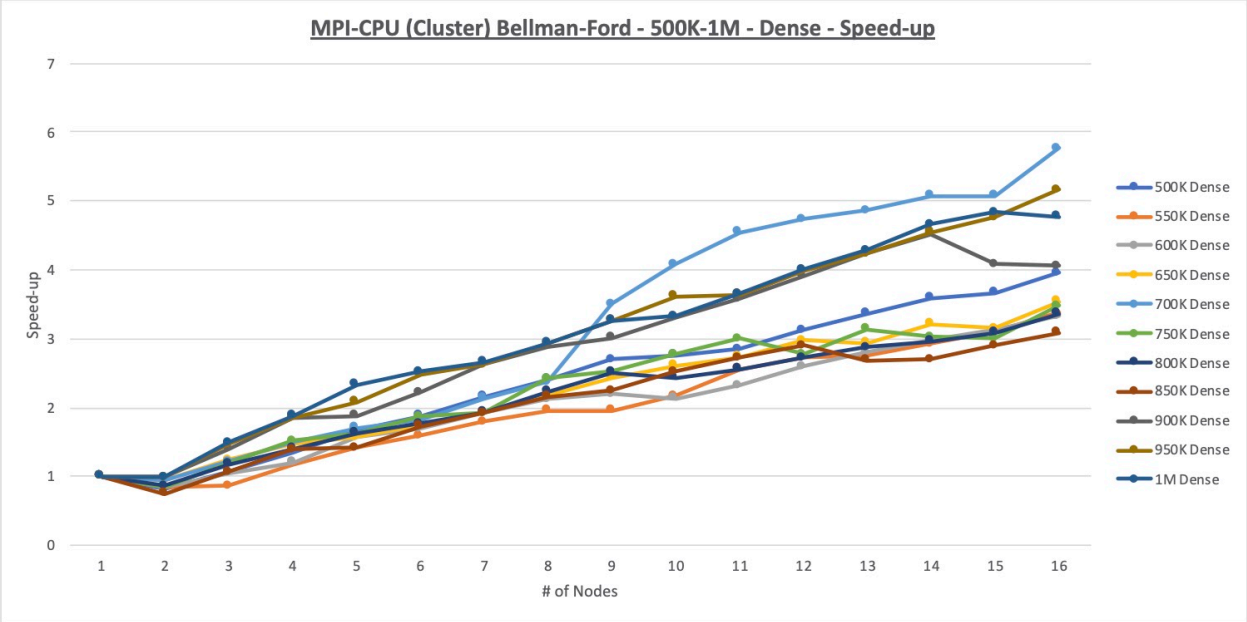

Figure D18. MPI-CPU (Cluster) Bellman-Ford 500K-1M Dense Speed-up Chart

APPENDIX E

MPI-CPU (SINGLE-NODE) BELLMAN-FORD DATA CHARTS

The runtime and speed-up charts for the MPI-CPU (Single-Node) Bellman-Ford experiments are presented as follows: Starting with the 5K-10K sparse graph set, the runtime charts for two graphs will be displayed on one page and then associated speed-up charts will be on the following page. For the ninth graph set (500K-1M dense), the runtime and speed-up charts will be on the same page.

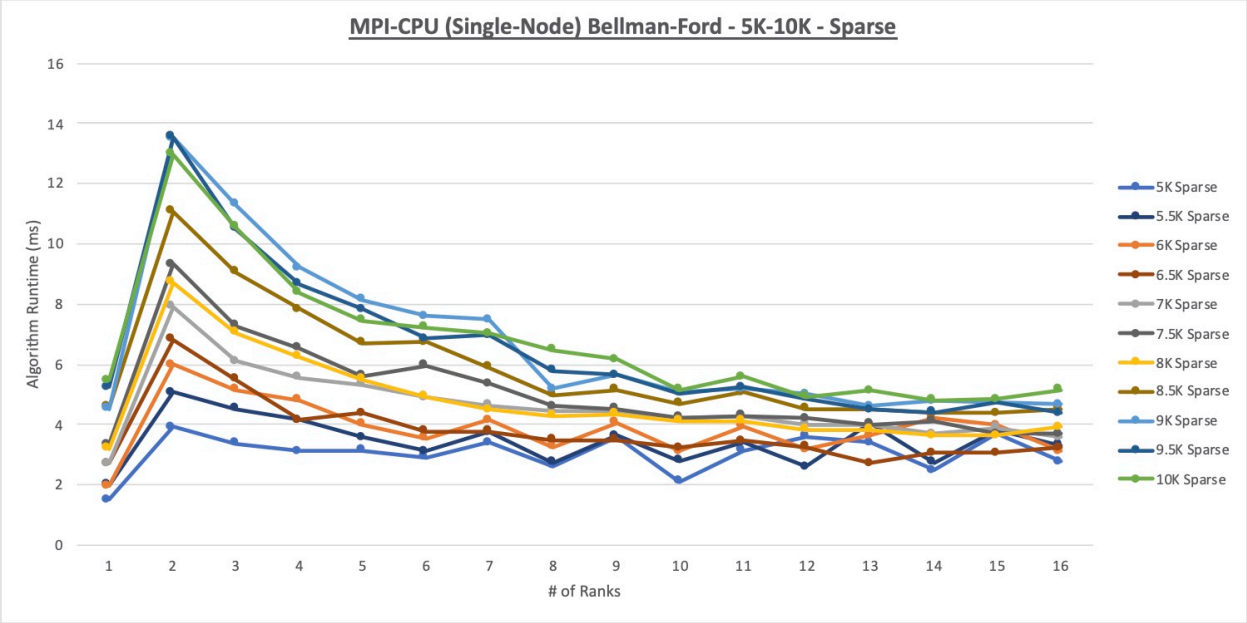

Figure E1. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Sparse Runtime Chart

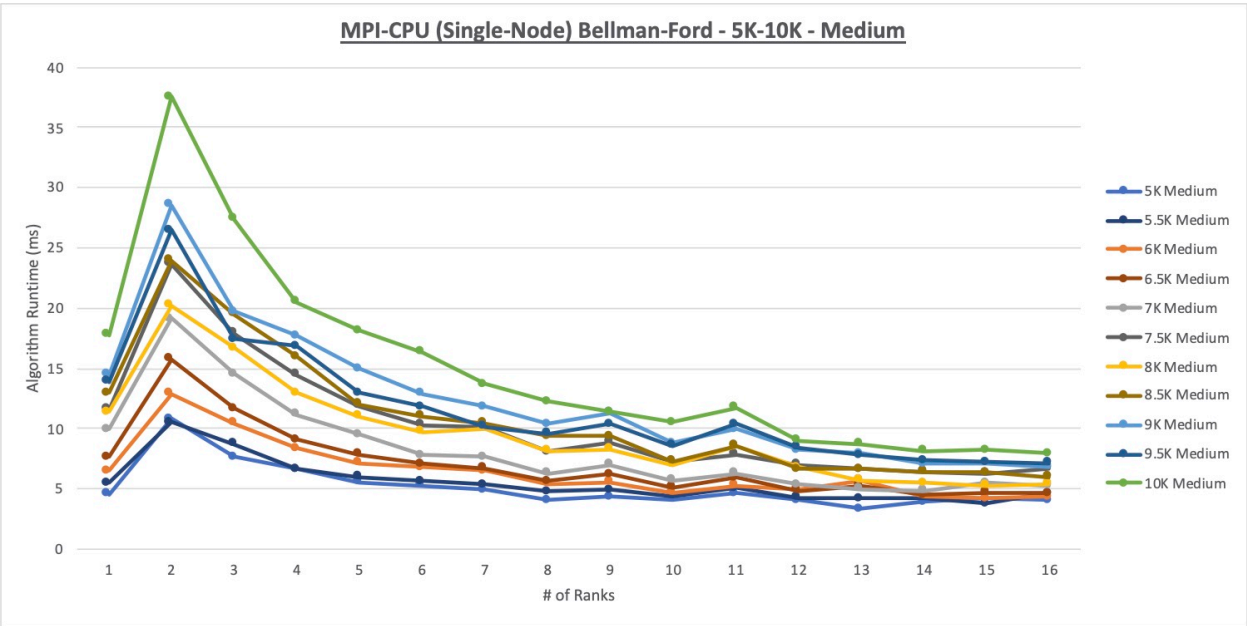

Figure E2. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Medium Runtime Chart

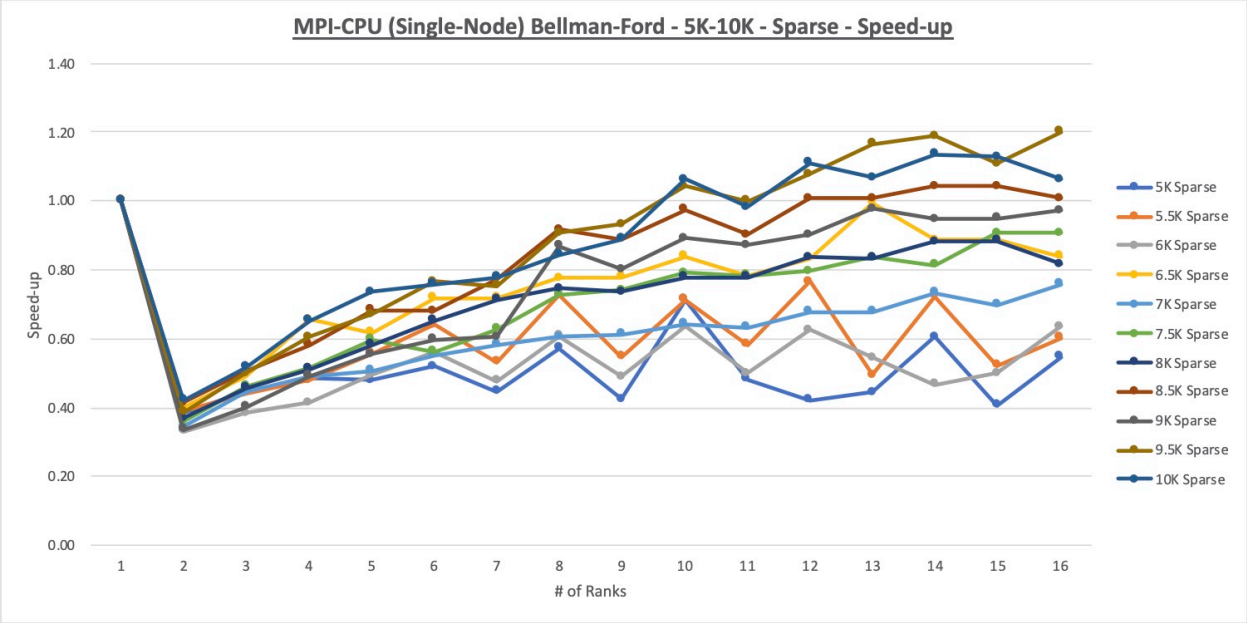

Figure E3. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Sparse Speed-up Chart

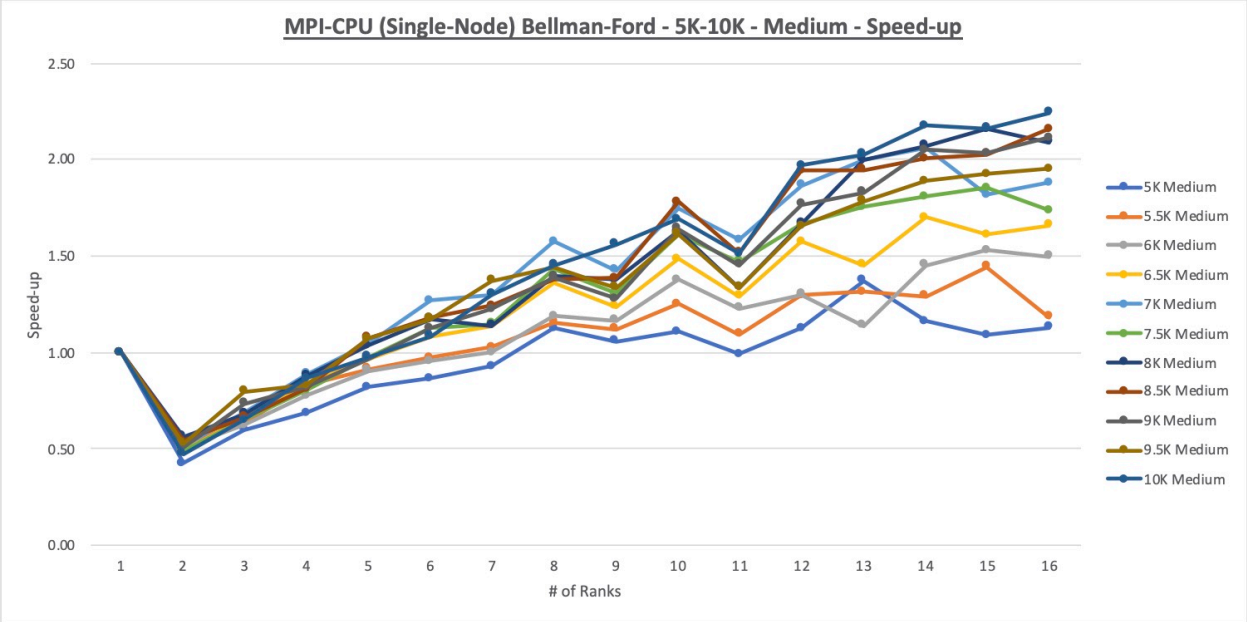

Figure E4. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Medium Speed-up Chart

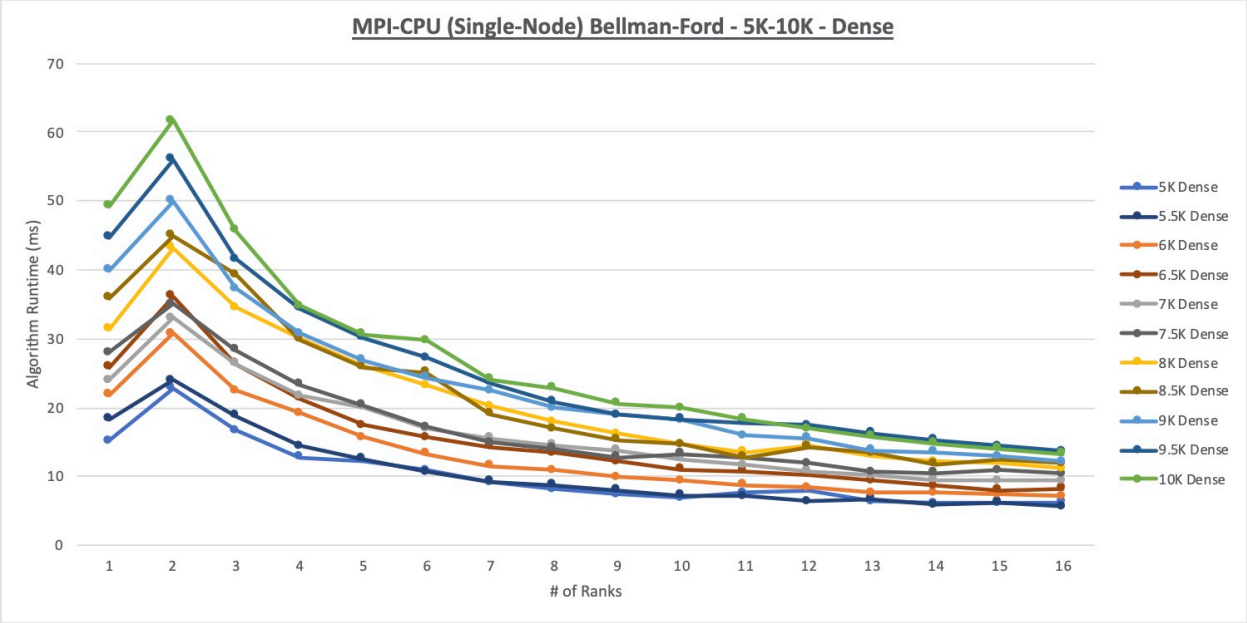

Figure E5. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Dense Runtime Chart

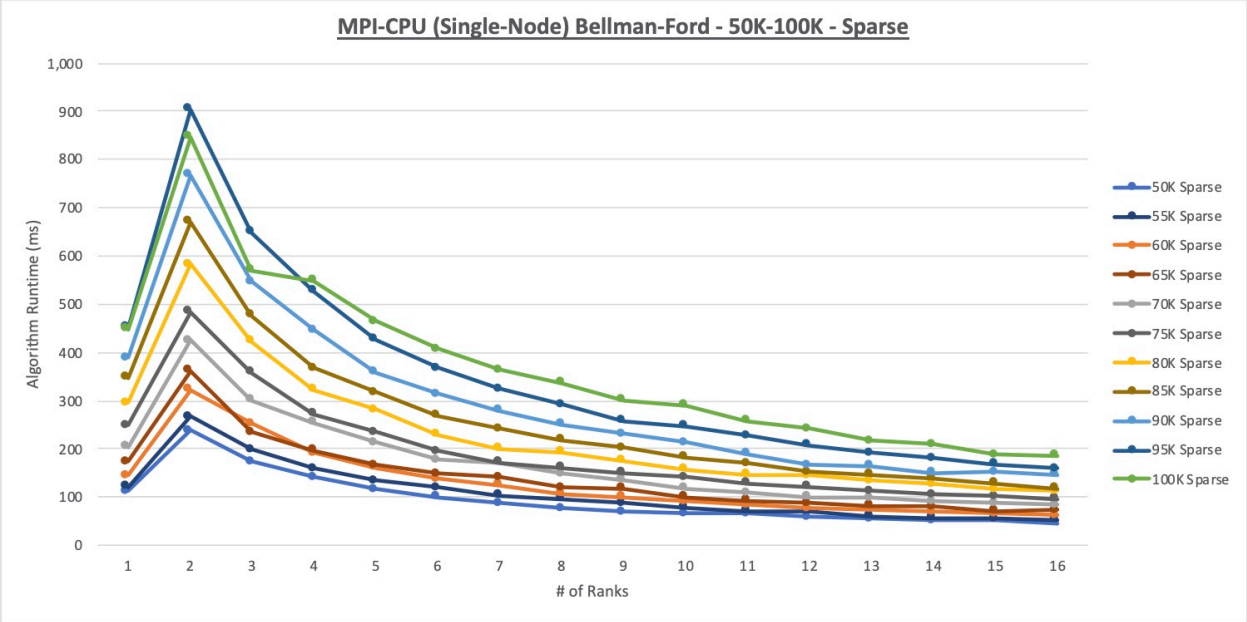

Figure E6. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Sparse Runtime Chart

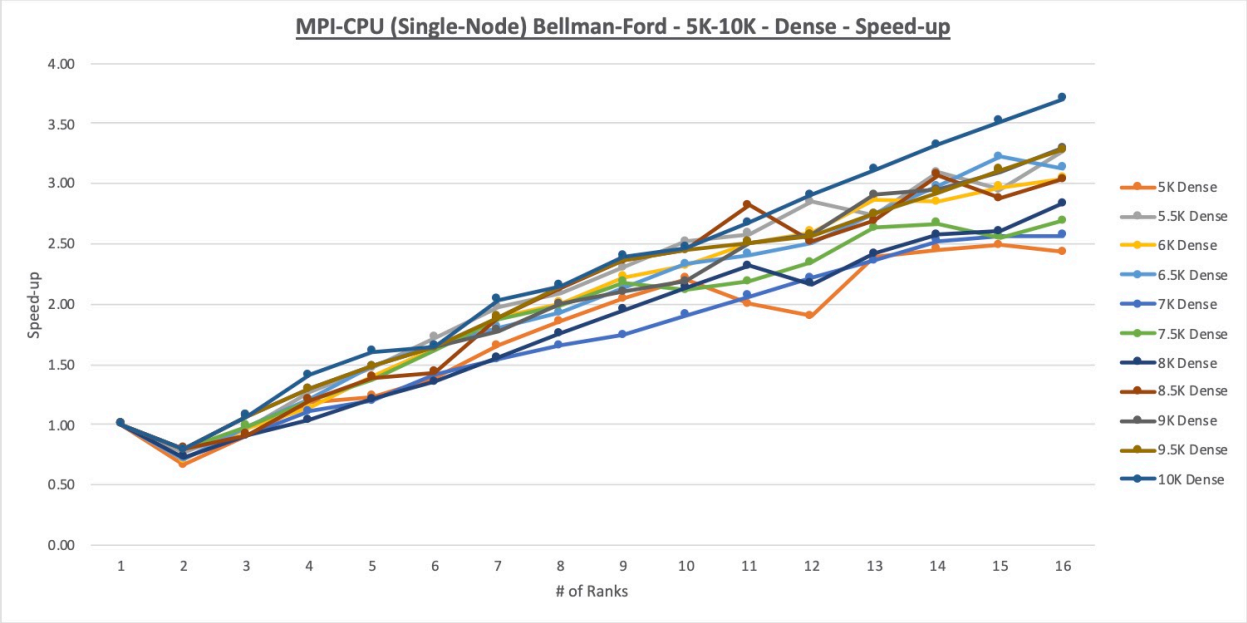

Figure E7. MPI-CPU (Single-Node) Bellman-Ford 5K-10K Dense Speed-up Chart

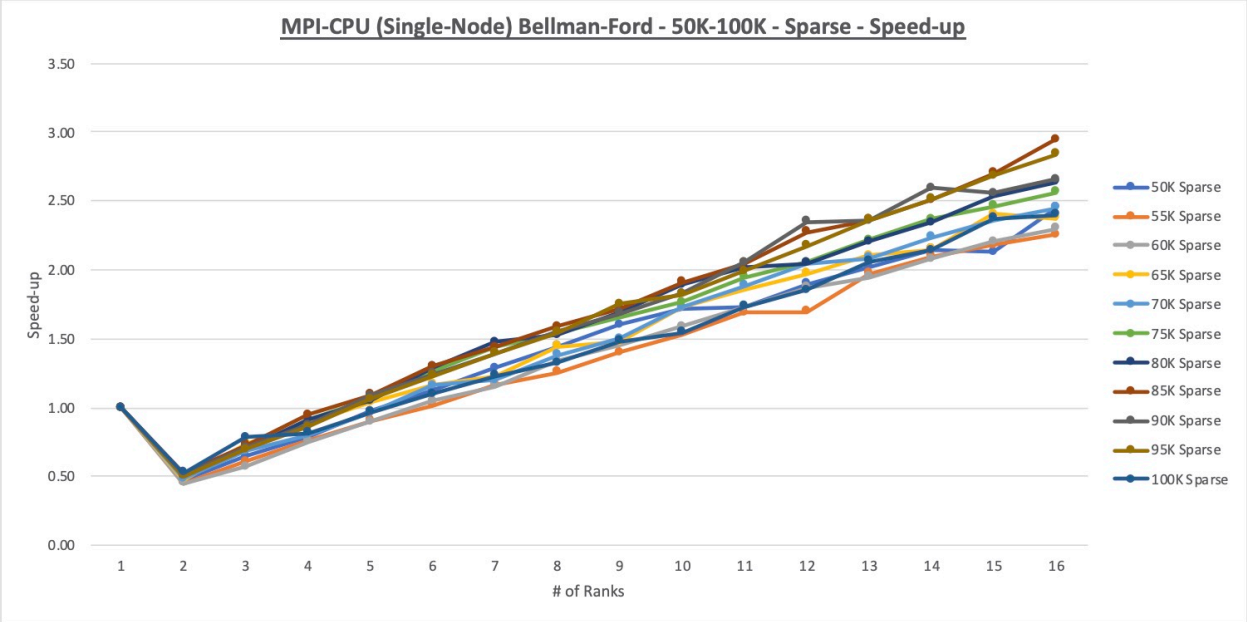

Figure E8. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Sparse Speed-up Chart

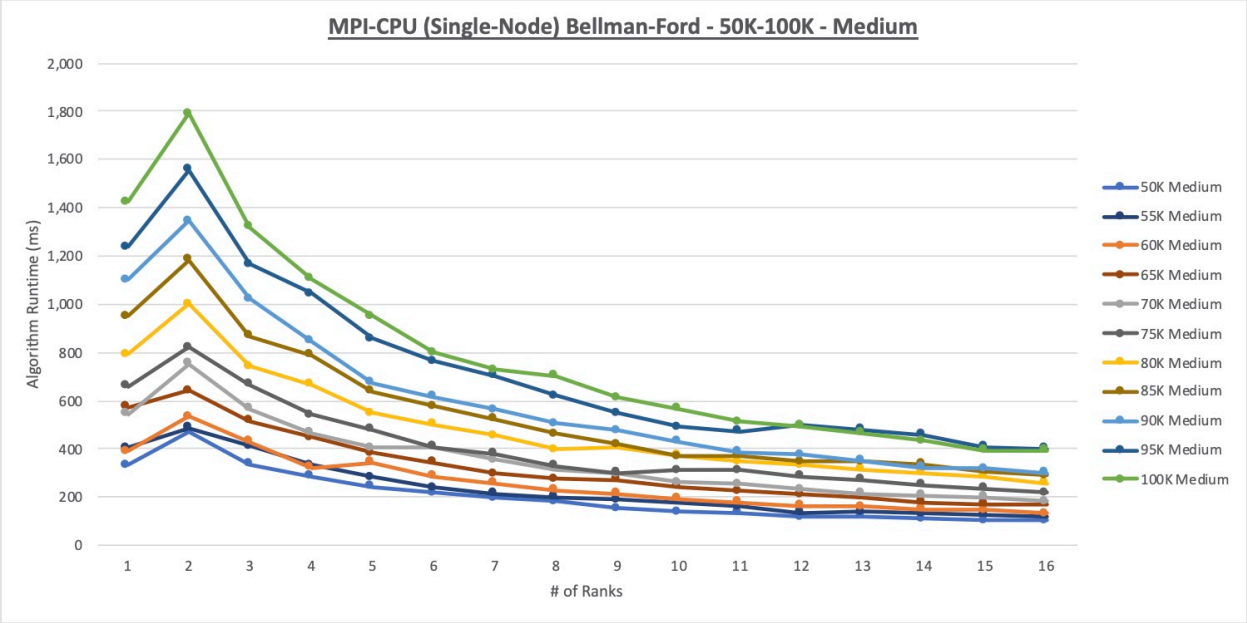

Figure E9. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Medium Runtime Chart

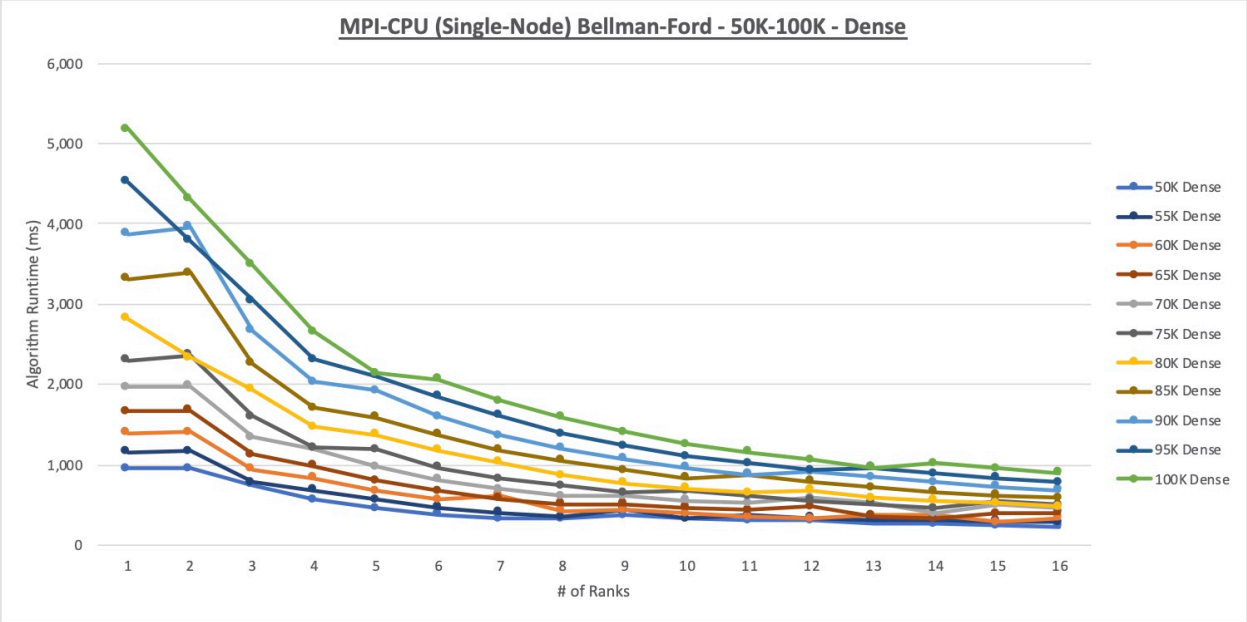

Figure E10. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Dense Runtime Chart

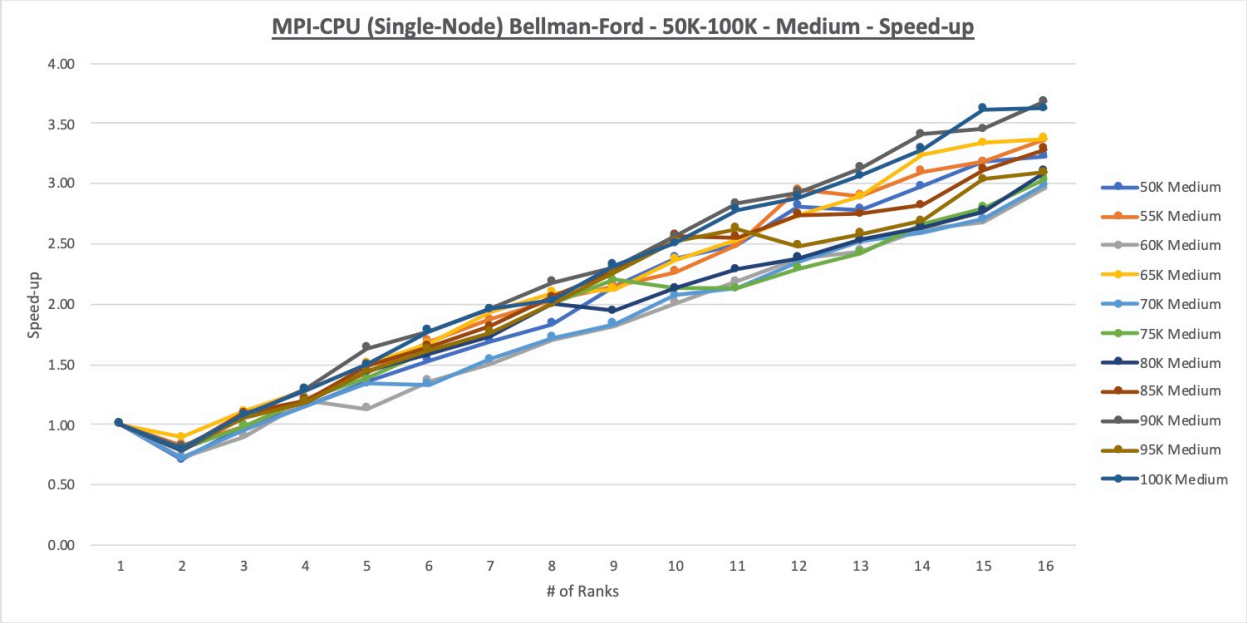

Figure E11. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Medium Speed-up Chart

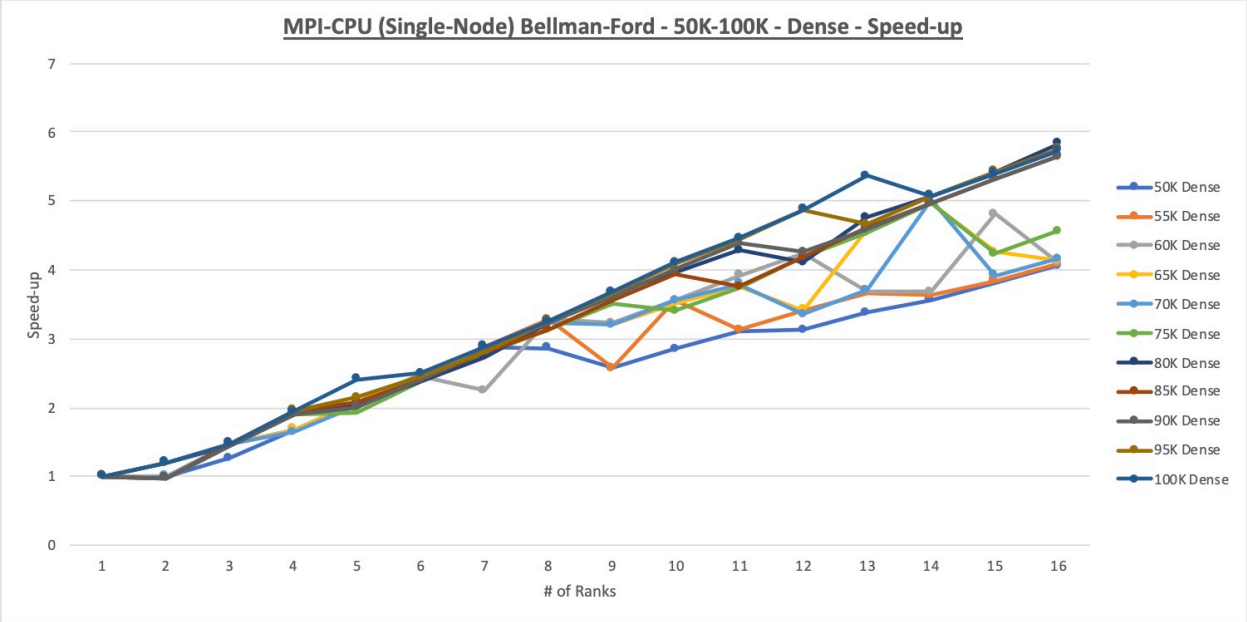

Figure E12. MPI-CPU (Single-Node) Bellman-Ford 50K-100K Dense Speed-up Chart

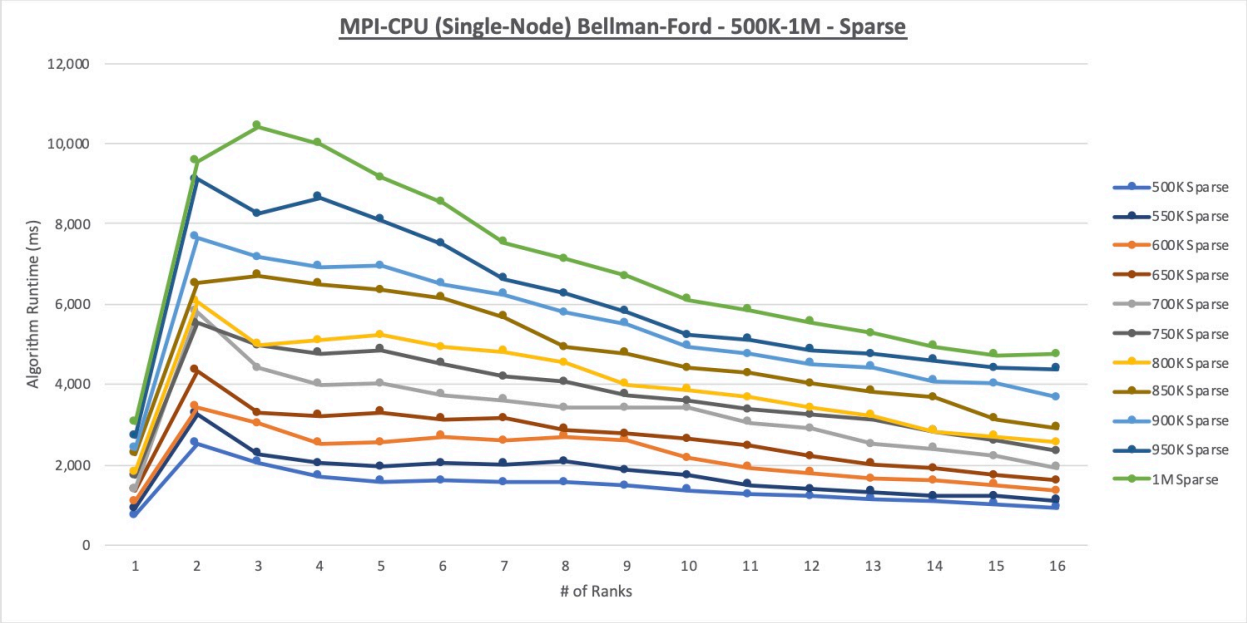

Figure E13. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Sparse Runtime Chart

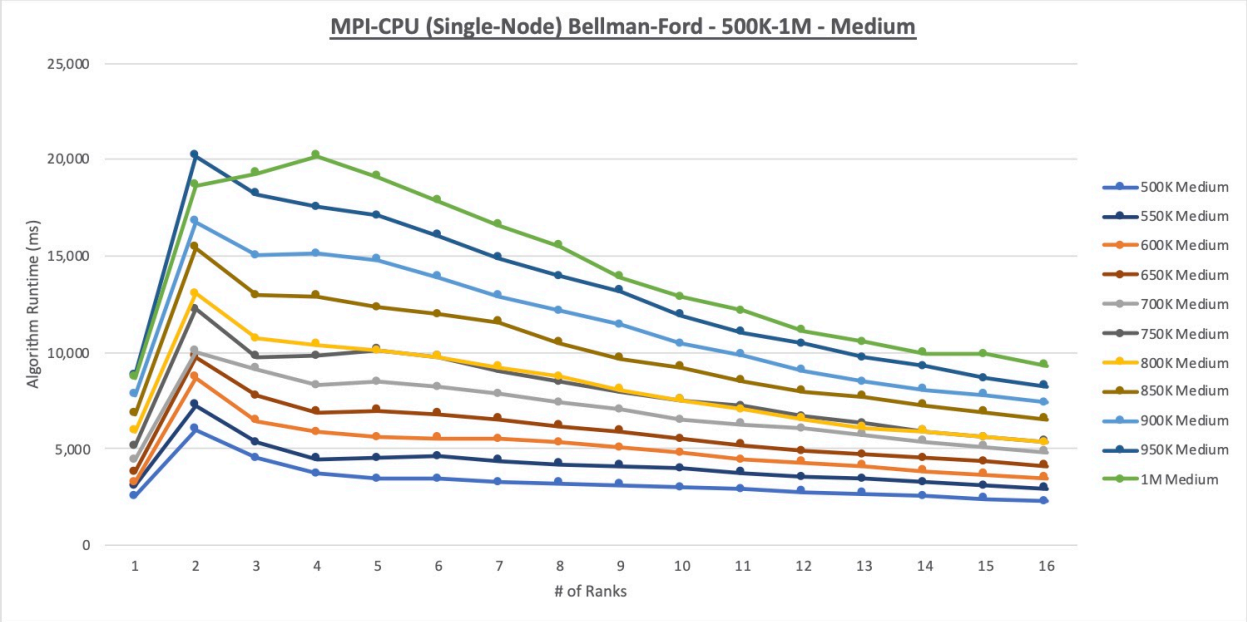

Figure E14. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Medium Runtime Chart

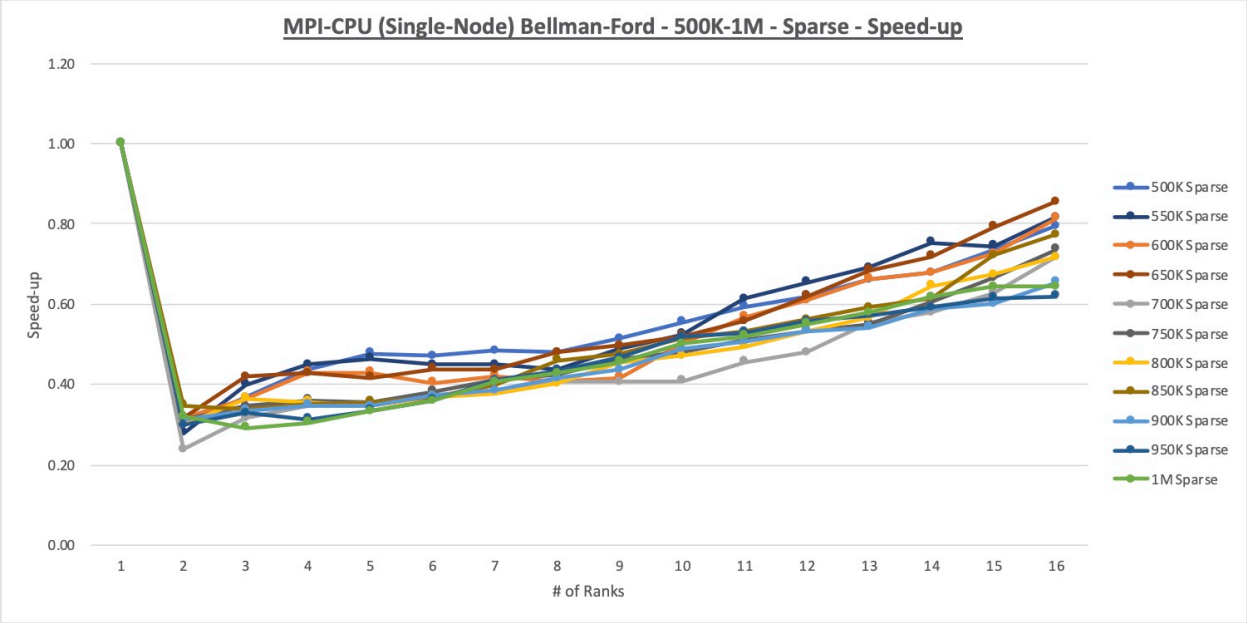

Figure E15. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Sparse Speed-up Chart

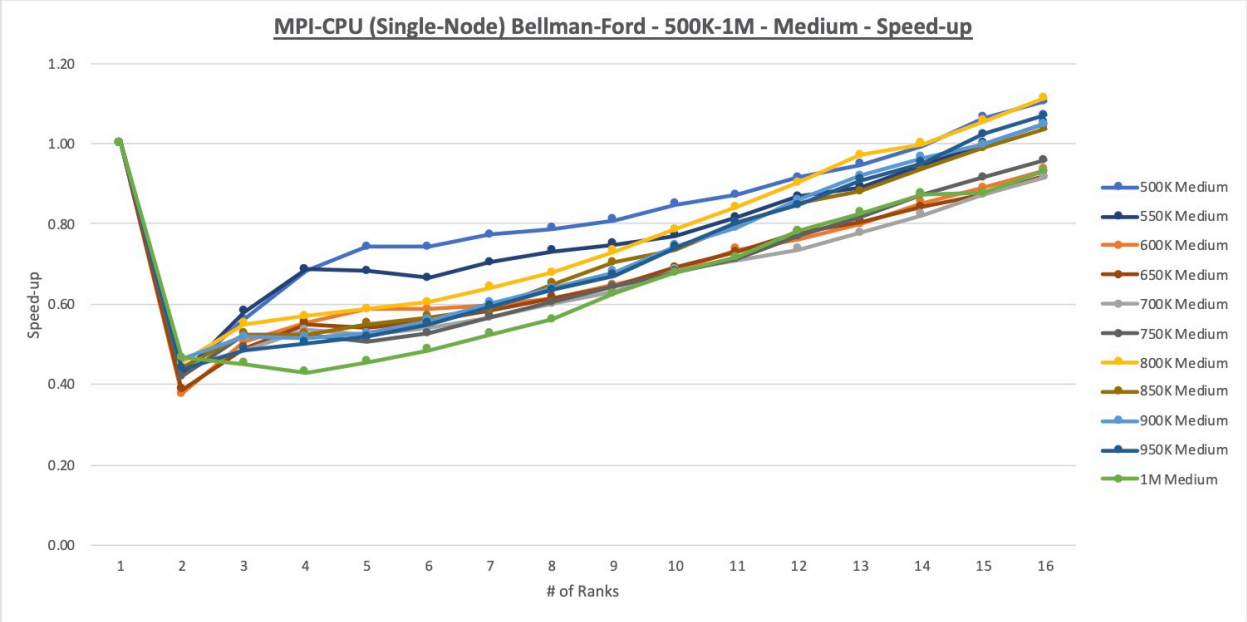

Figure E16. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Medium Speed-up Chart

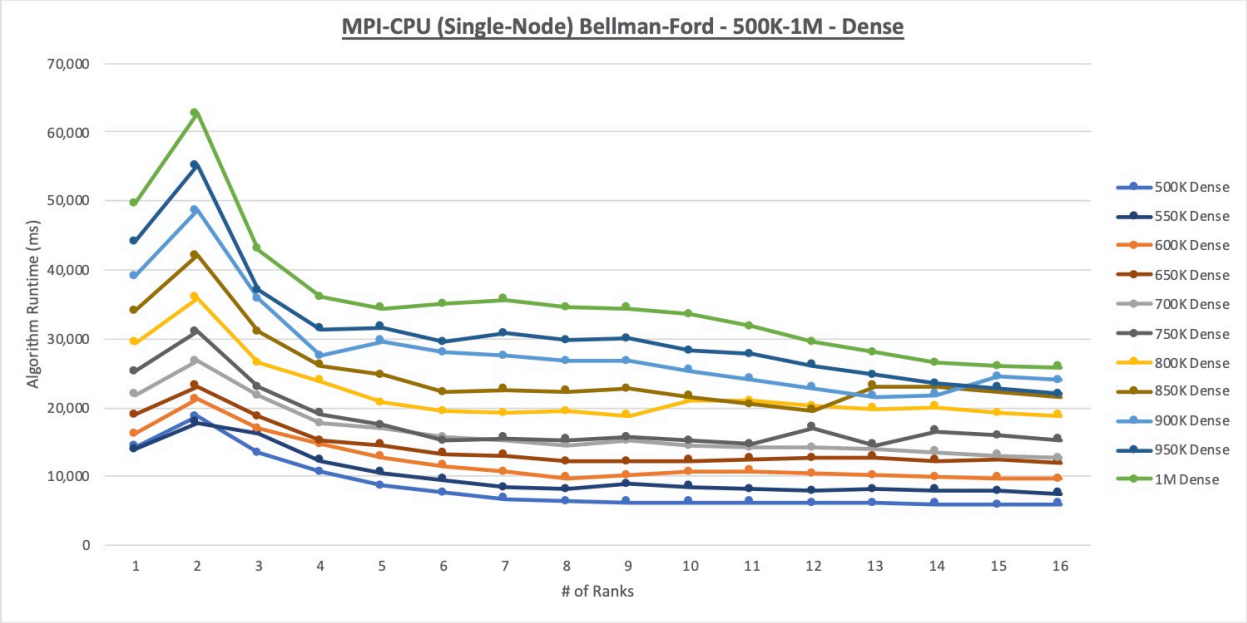

Figure E17. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Dense Runtime Chart

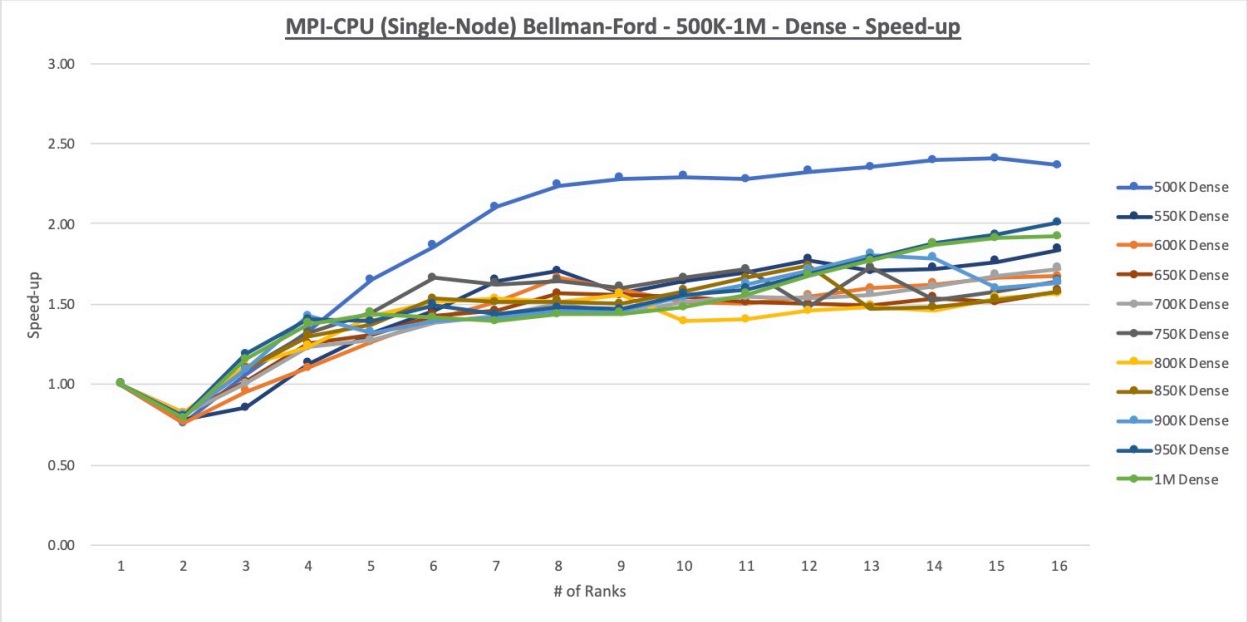

Figure E18. MPI-CPU (Single-Node) Bellman-Ford 500K-1M Dense Speed-up Chart

APPENDIX F

MPI-CPU FLOYD-WARSHALL DATA CHARTS

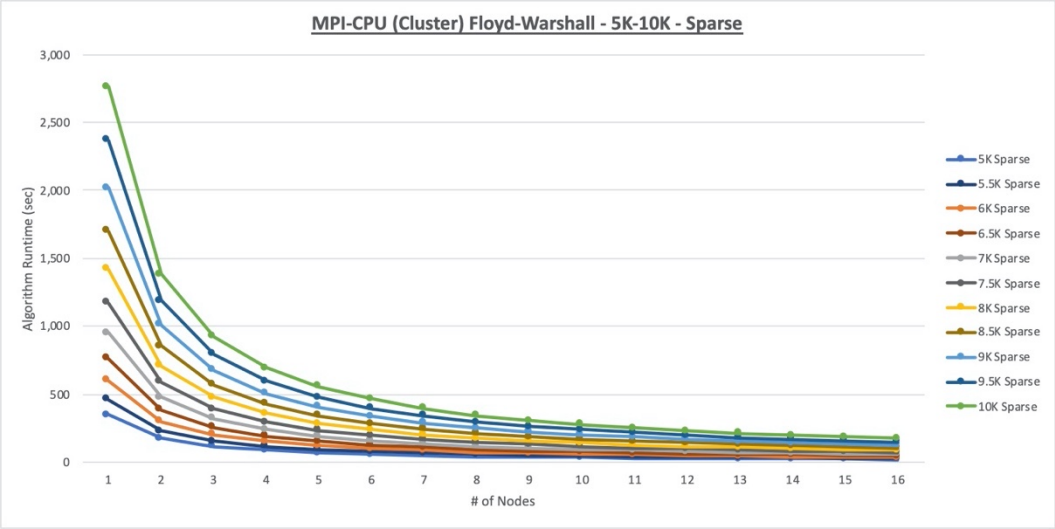

Figure F1. MPI-CPU (Cluster) Floyd-Warshall 5K-10K Sparse Runtime Chart

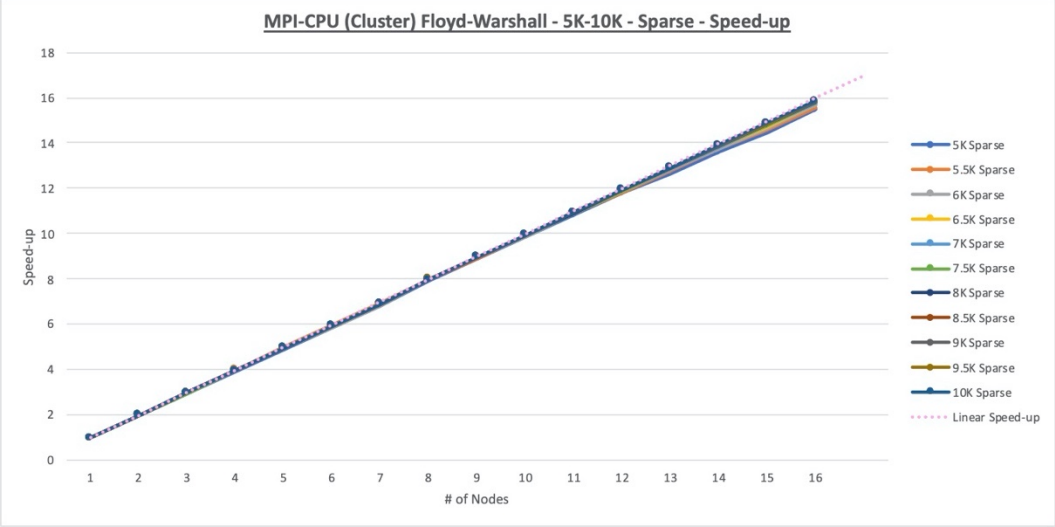

Figure F2. MPI-CPU (Cluster) Floyd-Warshall 5K-10K Sparse Speed-up Chart

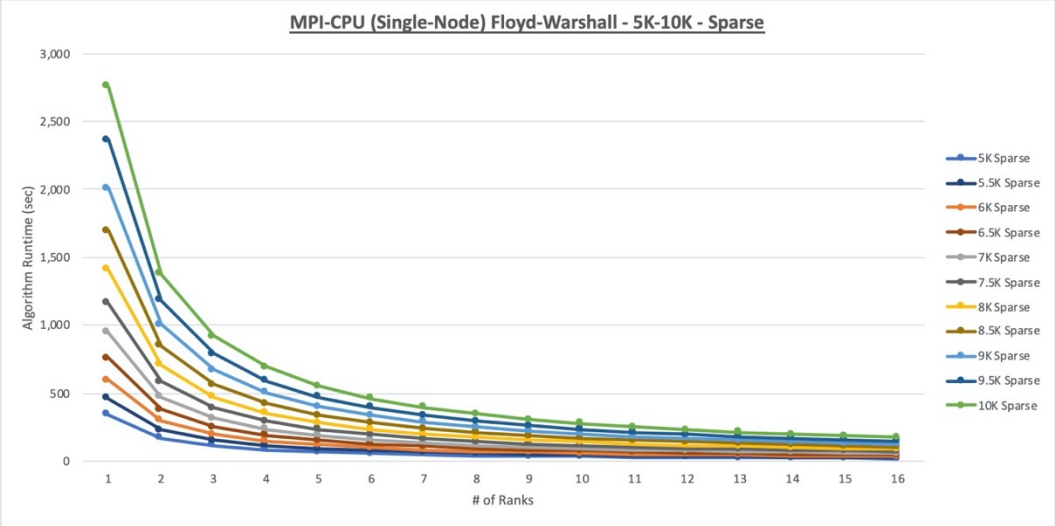

Figure F3. MPI-CPU (Single-Node) Floyd-Warshall 5K-10K Sparse Runtime Chart

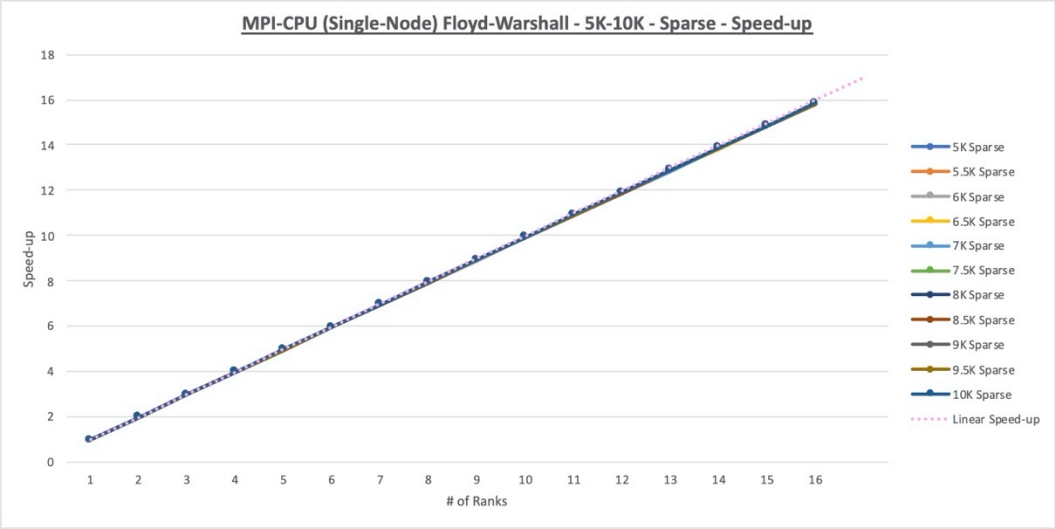

Figure F4. MPI-CPU (Single-Node) Floyd-Warshall 5K-10K Sparse Speed-up Chart

APPENDIX G

DIJKSTRA GPU DATA CHARTS

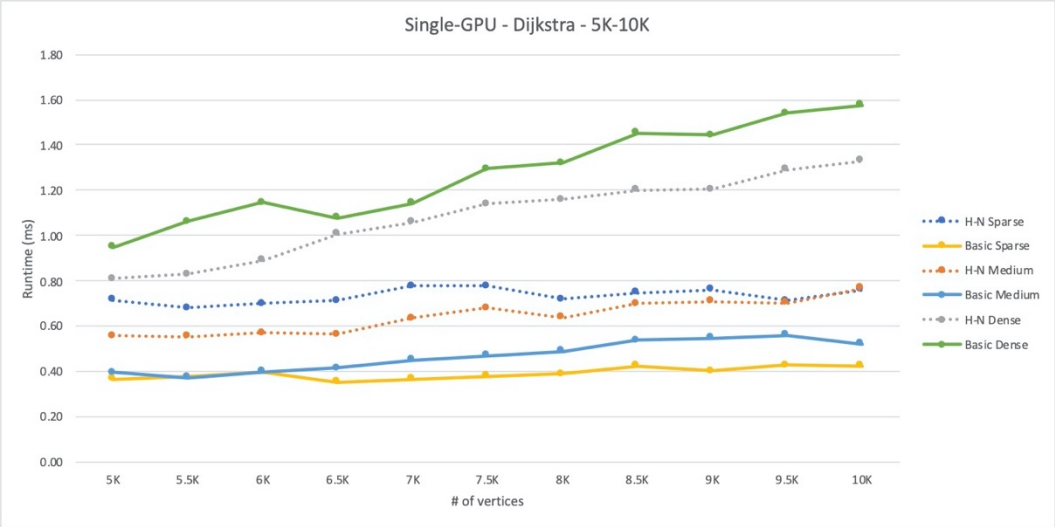

Figure G1. Single-GPU Dijkstra 5K-10K Runtime Chart

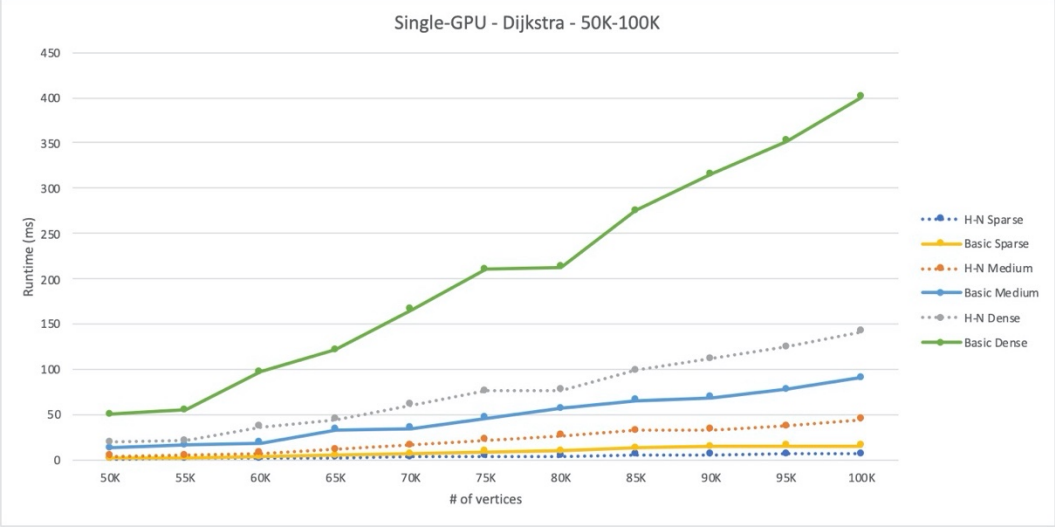

Figure G2. Single-GPU Dijkstra 50K-100K Runtime Chart

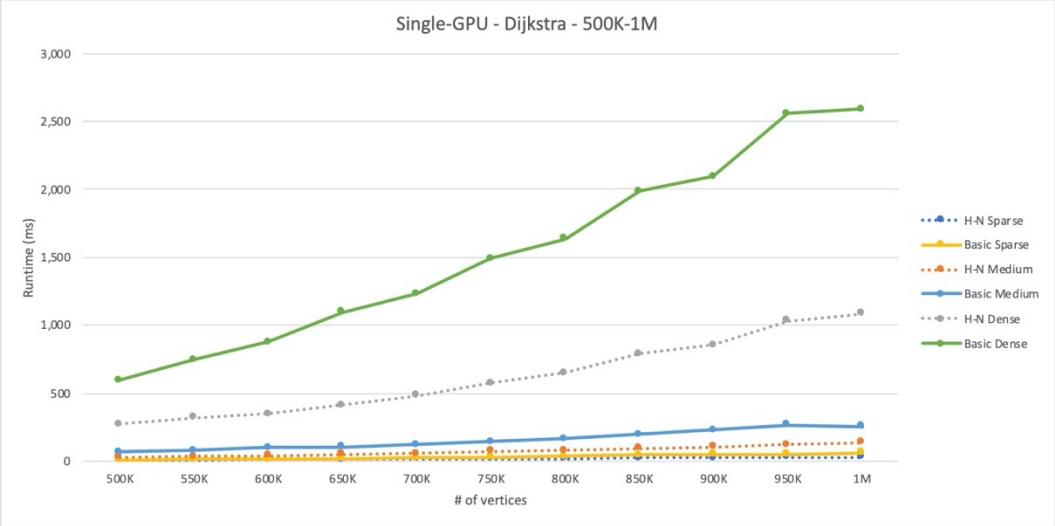

Figure G3. Single-GPU Dijkstra 500K-1M Runtime Chart

APPENDIX H

MPI-GPU (CLUSTER) BELLMAN-FORD DATA CHARTS

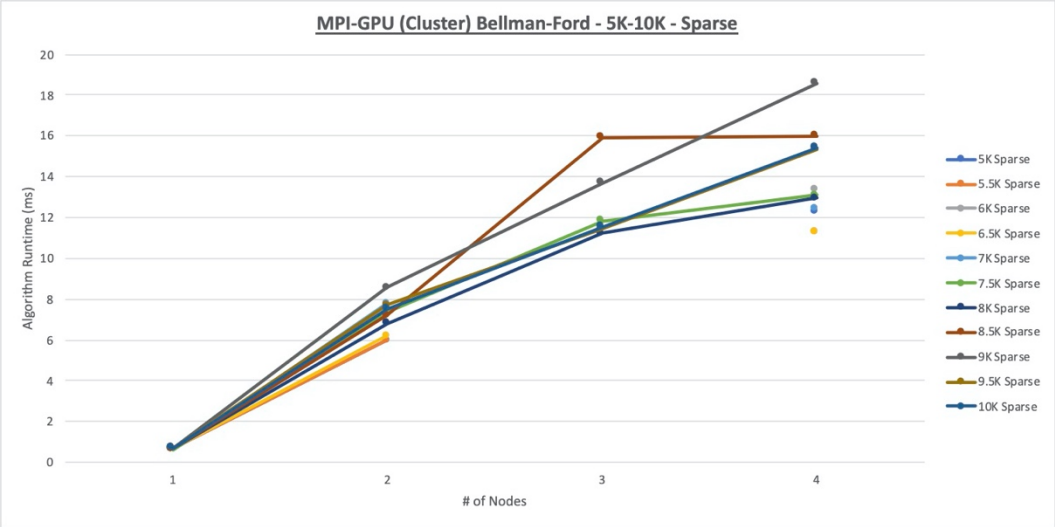

Figure H1. MPI-GPU (Cluster) Bellman-Ford 5K-10K Sparse Runtime Chart

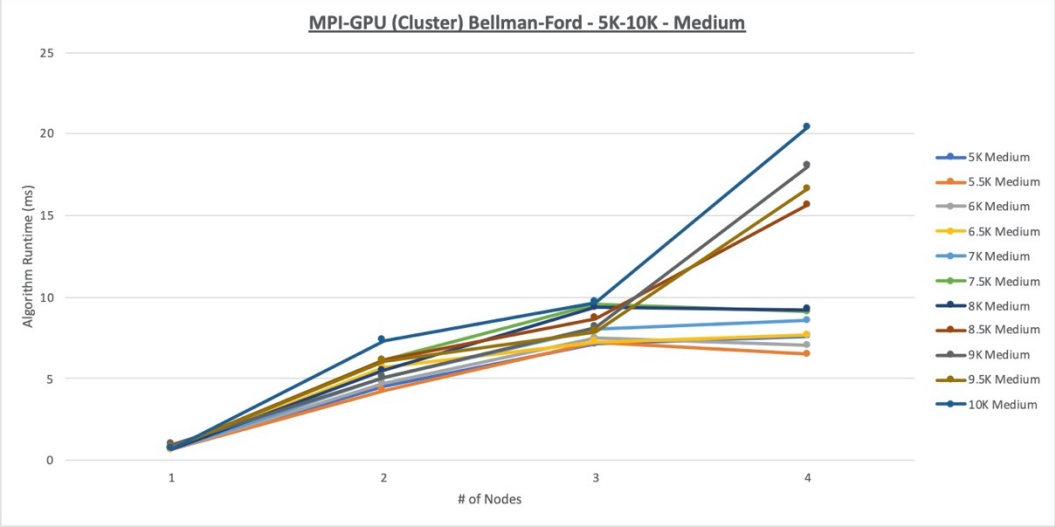

Figure H2. MPI-GPU (Cluster) Bellman-Ford 5K-10K Medium Runtime Chart

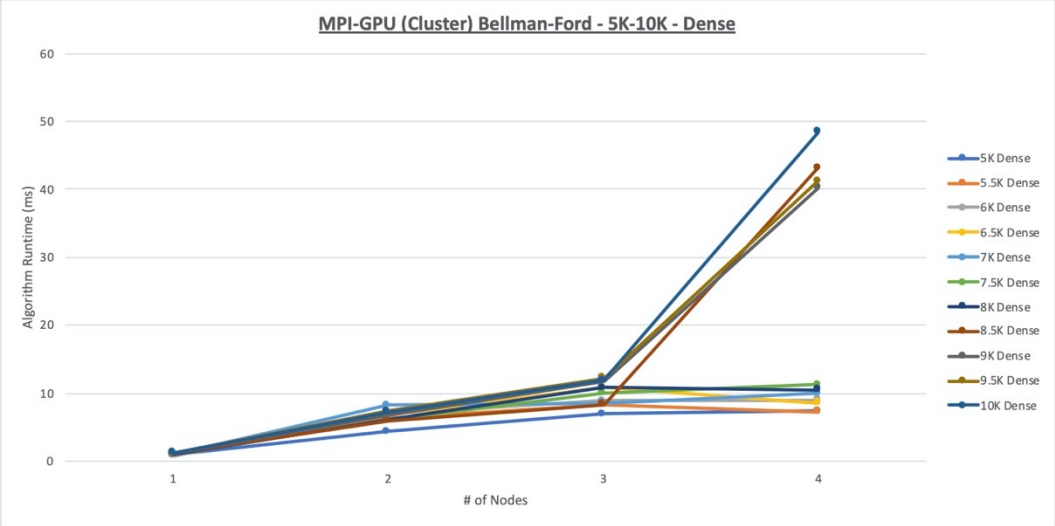

Figure H3. MPI-GPU (Cluster) Bellman-Ford 5K-10K Dense Runtime Chart

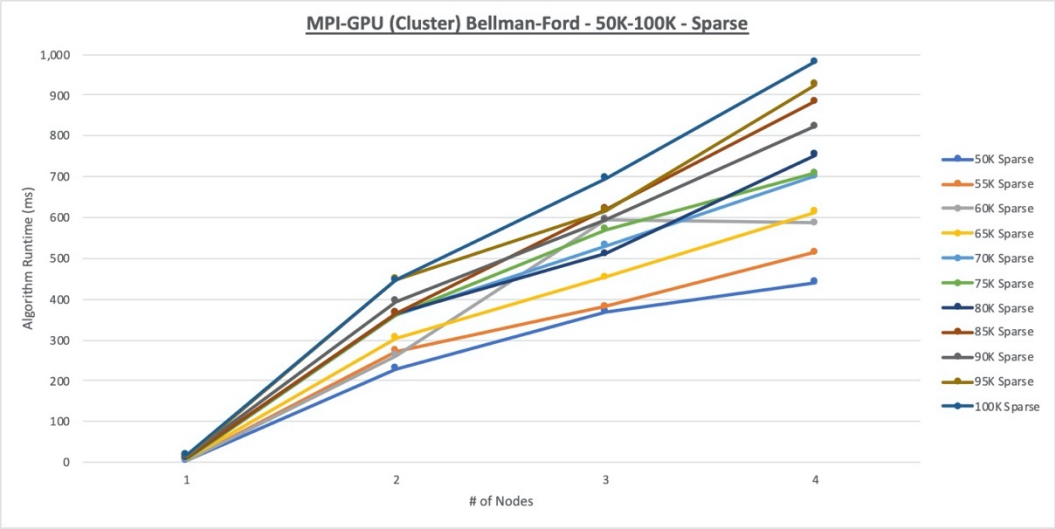

Figure H4. MPI-GPU (Cluster) Bellman-Ford 50K-100K Sparse Runtime Chart

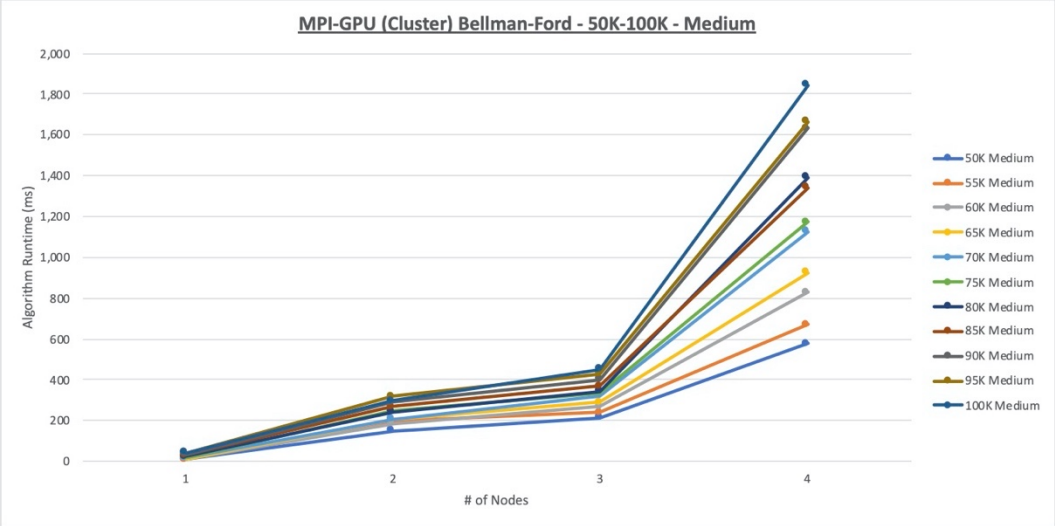

Figure H5. MPI-GPU (Cluster) Bellman-Ford 50K-100K Medium Runtime Chart

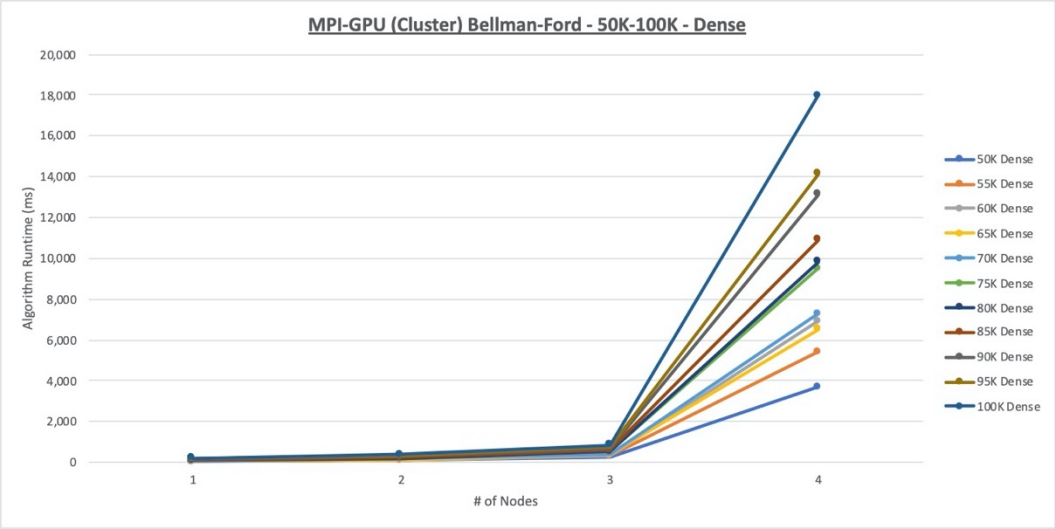

Figure H6. MPI-GPU (Cluster) Bellman-Ford 50K-100K Dense Runtime Chart

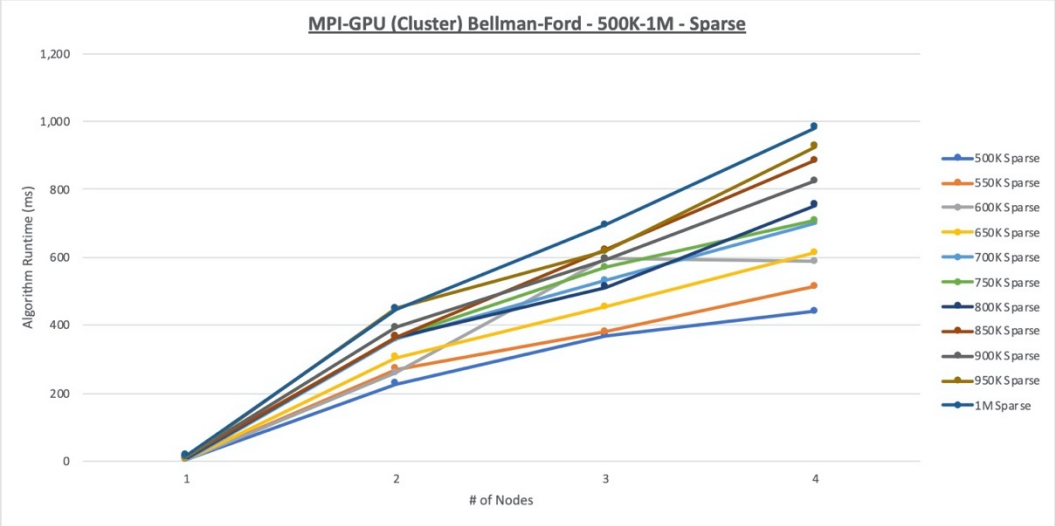

Figure H7. MPI-GPU (Cluster) Bellman-Ford 500K-1M Sparse Runtime Chart

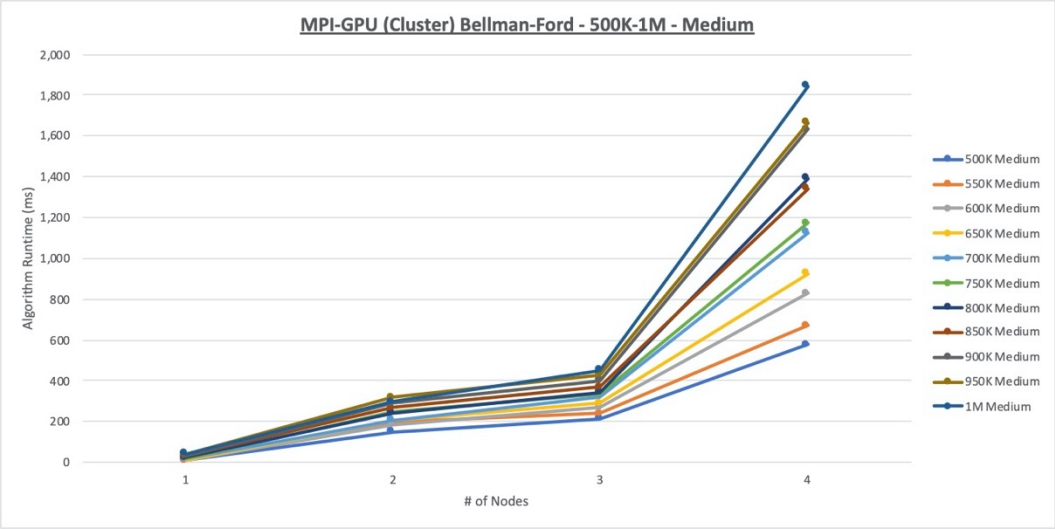

Figure H8. MPI-GPU (Cluster) Bellman-Ford 500K-1M Medium Runtime Chart

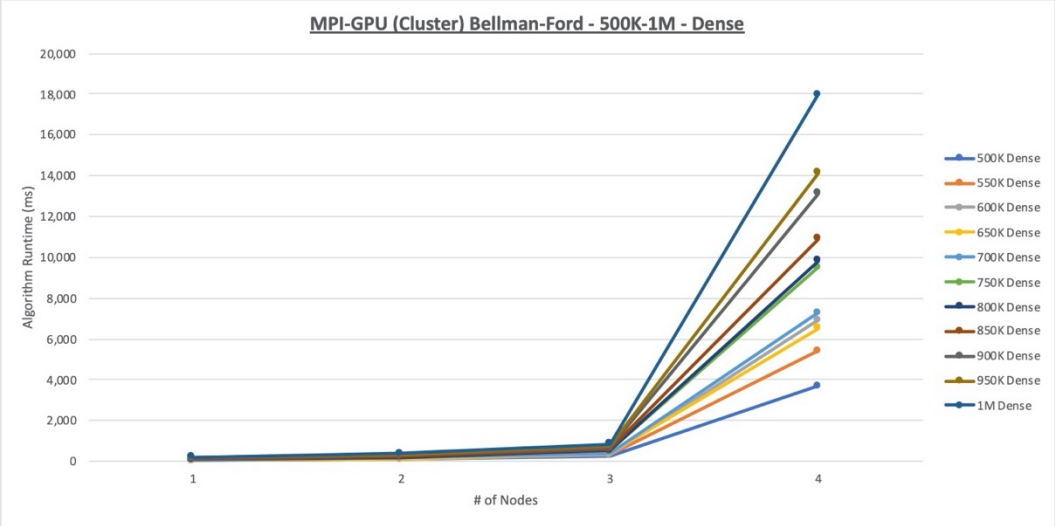

Figure H9. MPI-GPU (Cluster) Bellman-Ford 500K-1M Dense Runtime Chart

APPENDIX I

MPI-CPU (SINGLE-NODE) BELLMAN-FORD DATA CHARTS

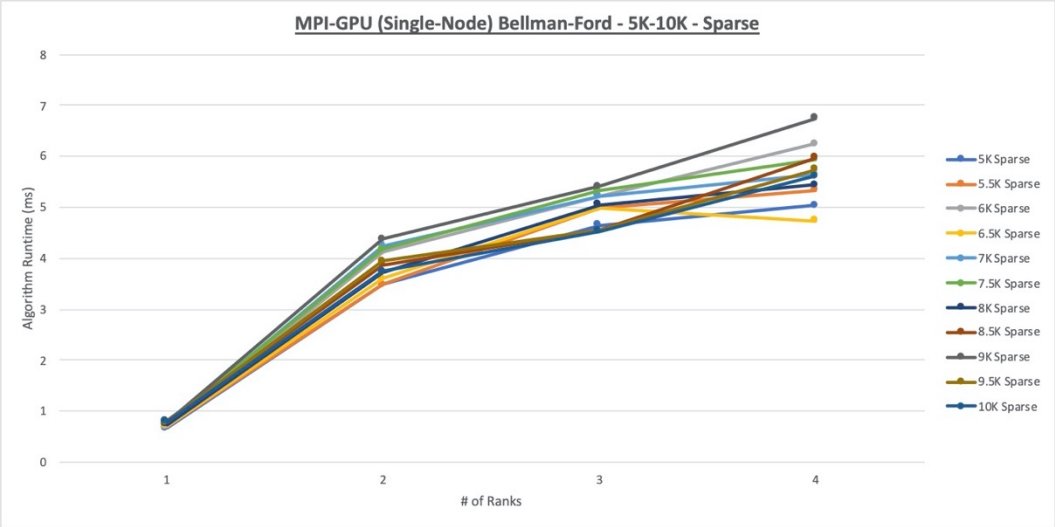

Figure I1. MPI-GPU (Single-Node) Bellman-Ford 5K-10K Sparse Runtime Chart

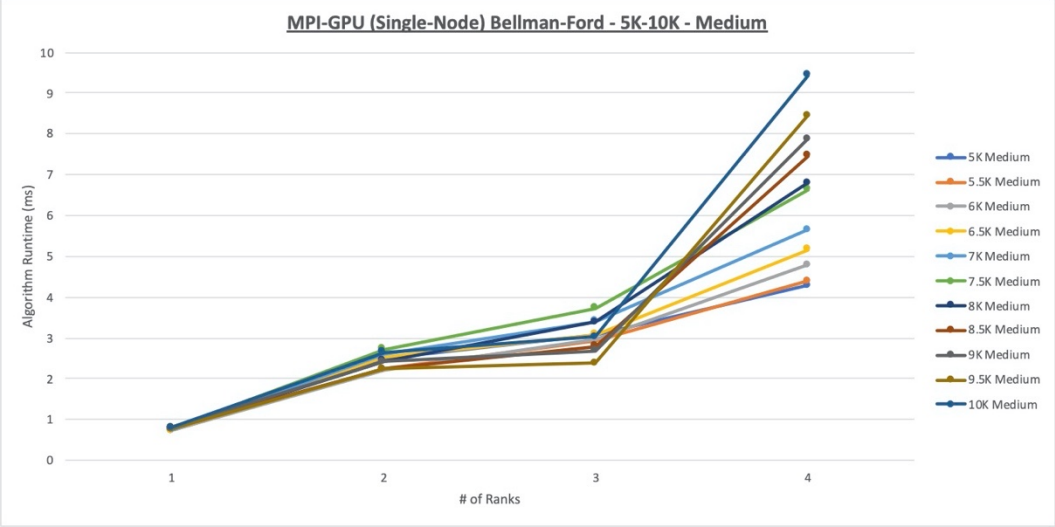

Figure I2. MPI-GPU (Single-Node) Bellman-Ford 5K-10K Medium Runtime Chart

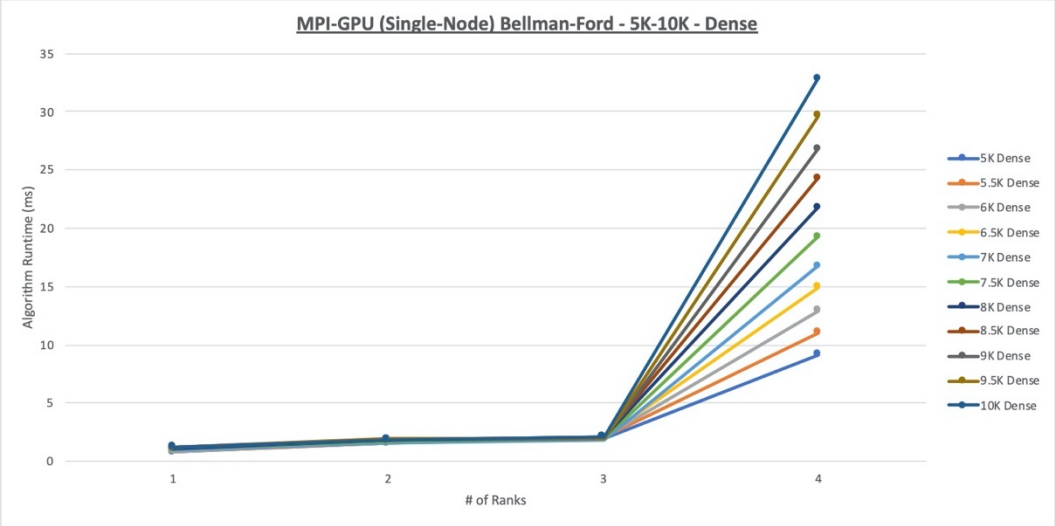

Figure I3. MPI-GPU (Single-Node) Bellman-Ford 5K-10K Dense Runtime Chart

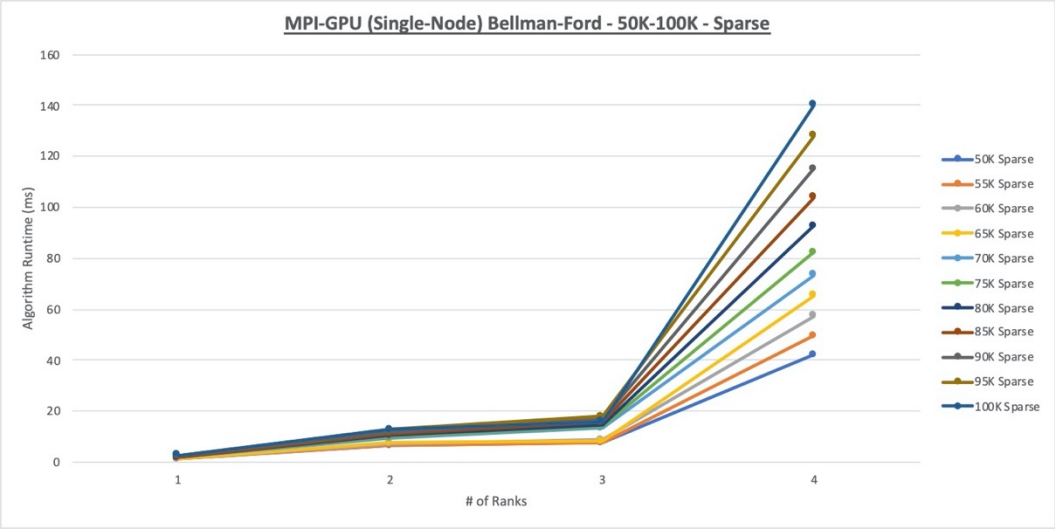

Figure I4. MPI-GPU (Single-Node) Bellman-Ford 50K-100K Sparse Runtime Chart

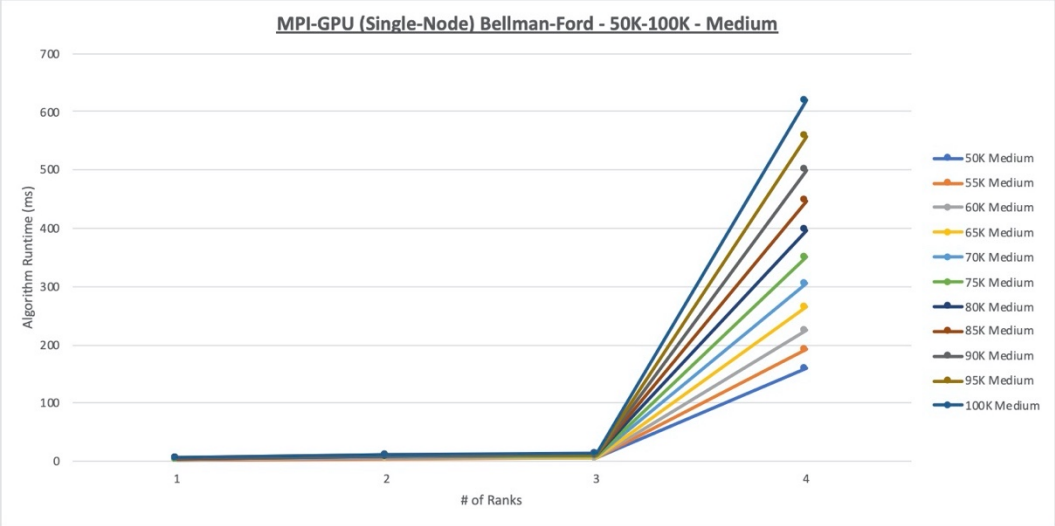

Figure I5. MPI-GPU (Single-Node) Bellman-Ford 50K-100K Medium Runtime Chart

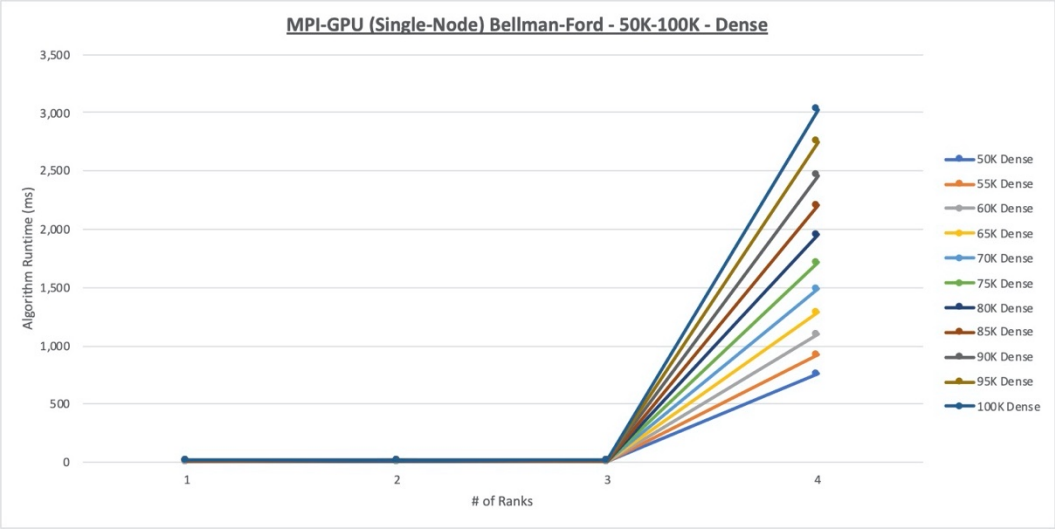

Figure I6. MPI-GPU (Single-Node) Bellman-Ford 50K-100K Dense Runtime Chart

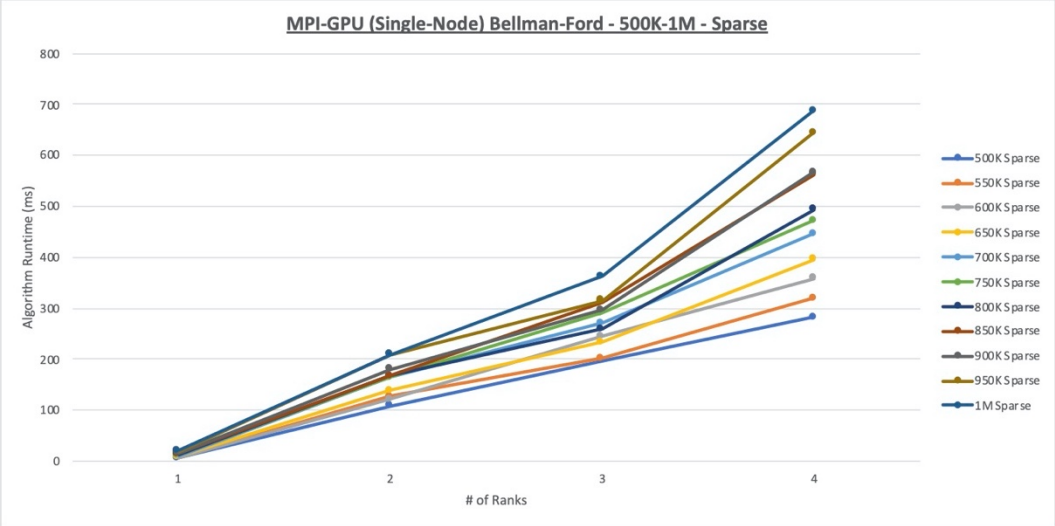

Figure I7. MPI-GPU (Single-Node) Bellman-Ford 500K-1M Sparse Runtime Chart

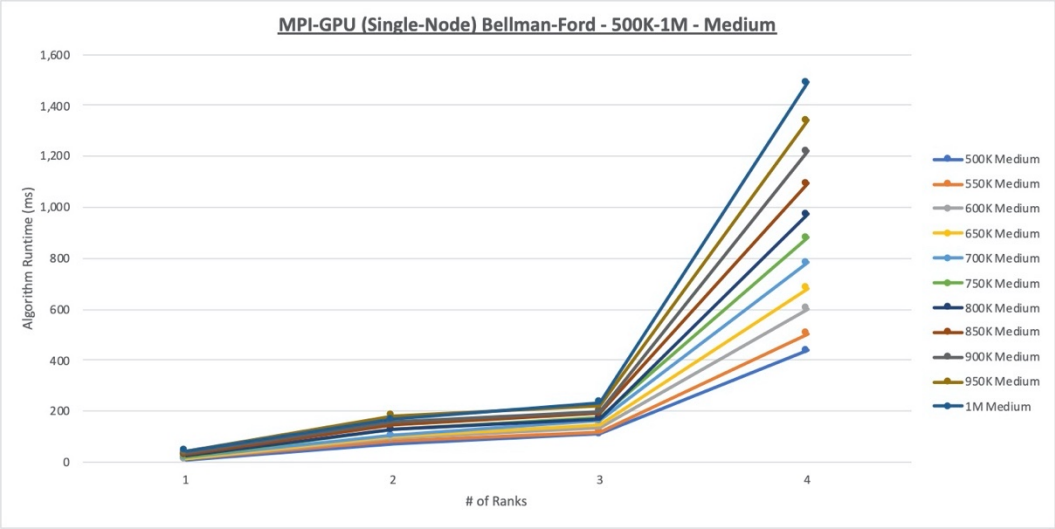

Figure I8. MPI-GPU (Single-Node) Bellman-Ford 500K-1M Medium Runtime Chart

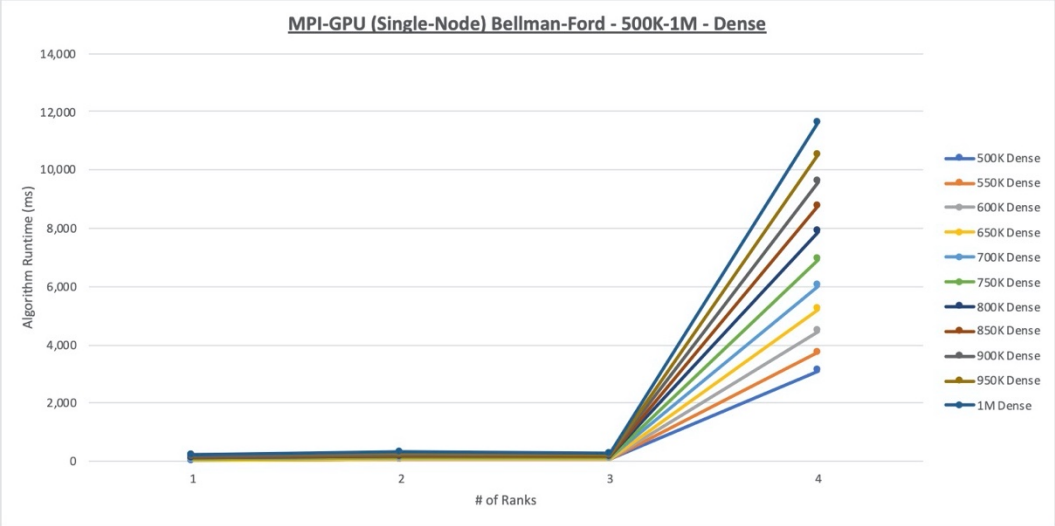

Figure I9. MPI-GPU (Single-Node) Bellman-Ford 500K-1M Dense Runtime Chart

APPENDIX J

BELLMAN-FORD SINGLE-GPU DATA CHARTS

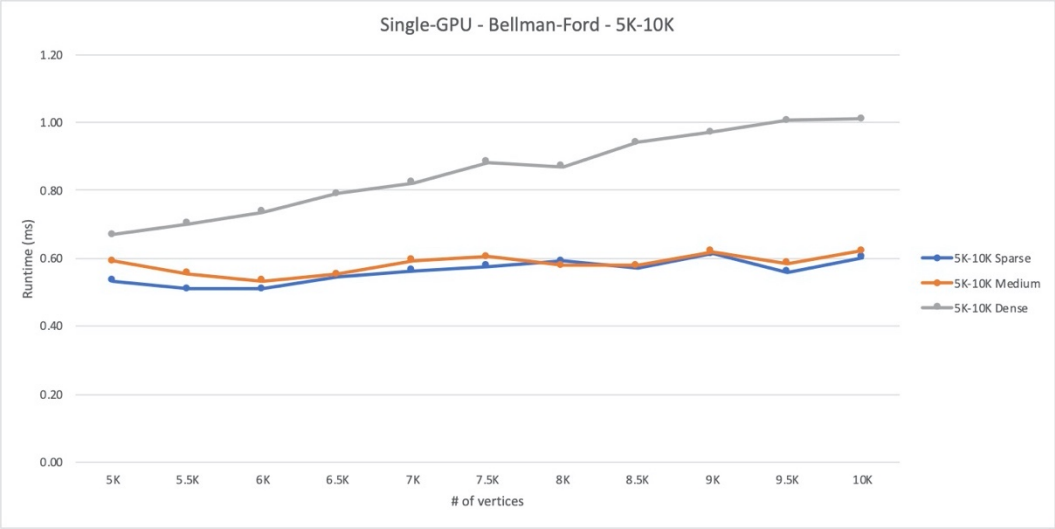

Figure J1. Single-GPU Bellman-Ford 5K-10K Runtime Chart

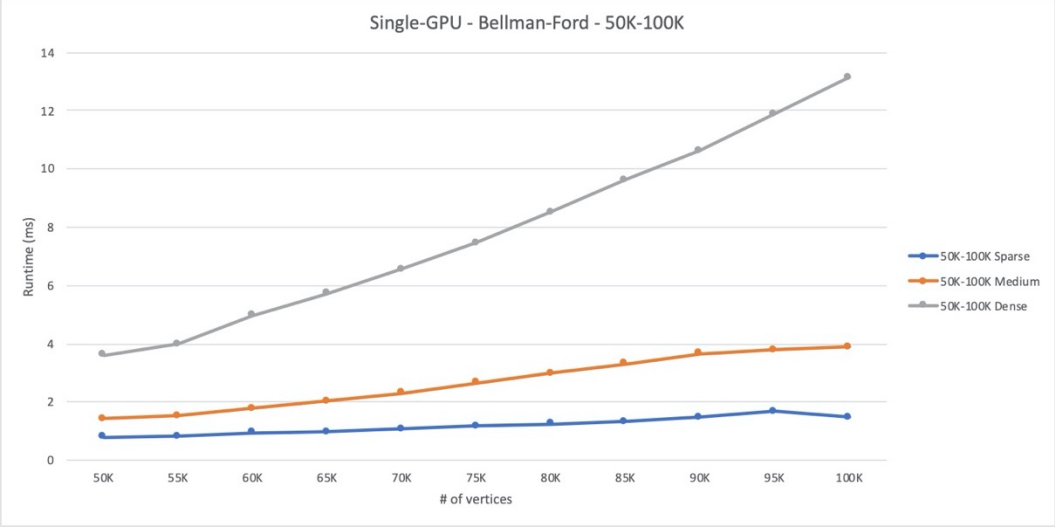

Figure J2. Single-GPU Bellman-Ford 50K-100K Runtime Chart

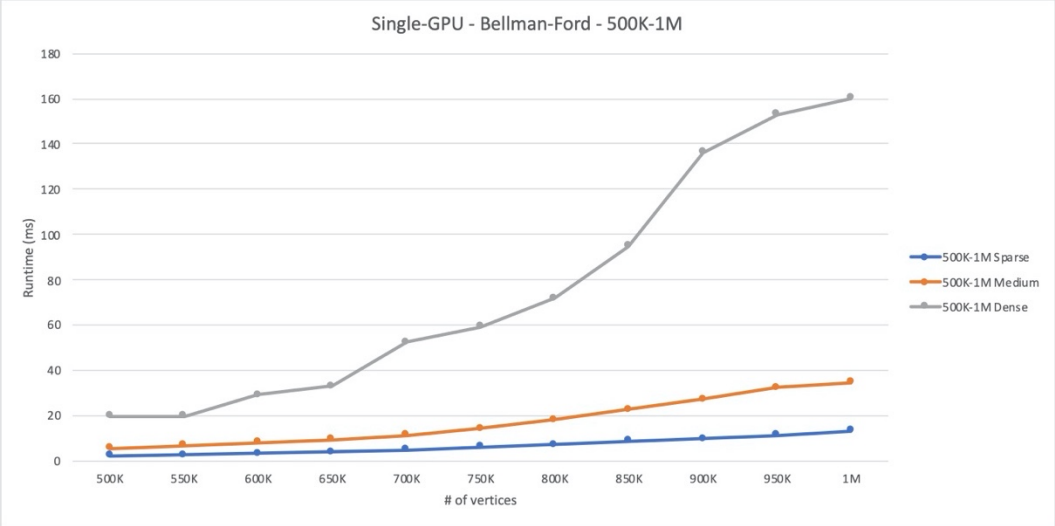

Figure J3. Single-GPU Bellman-Ford 500K-1M Runtime Chart

APPENDIX K

FLOYD-WARSHALL GPU DATA CHARTS

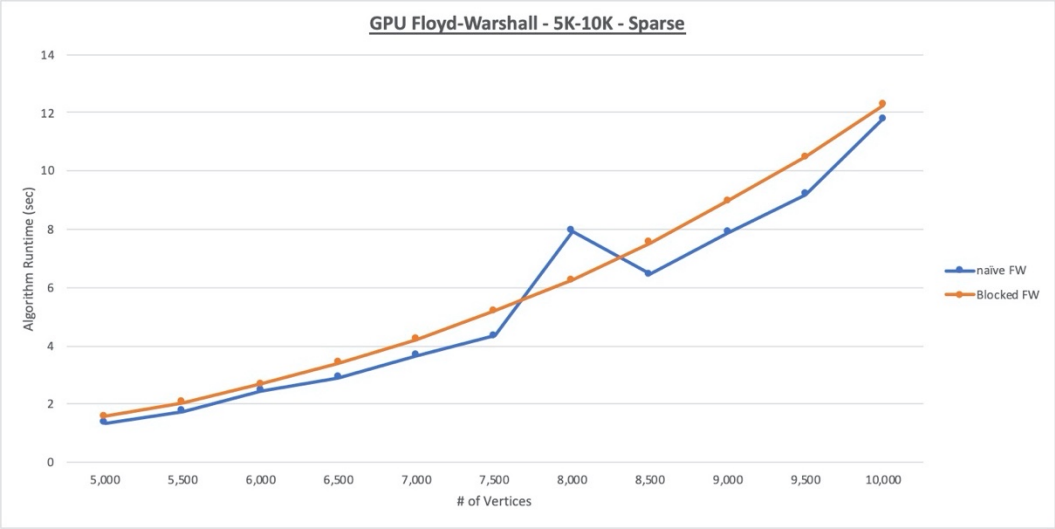

Figure K1. GPU Floyd-Warshall 5K-10K Sparse Runtime Chart

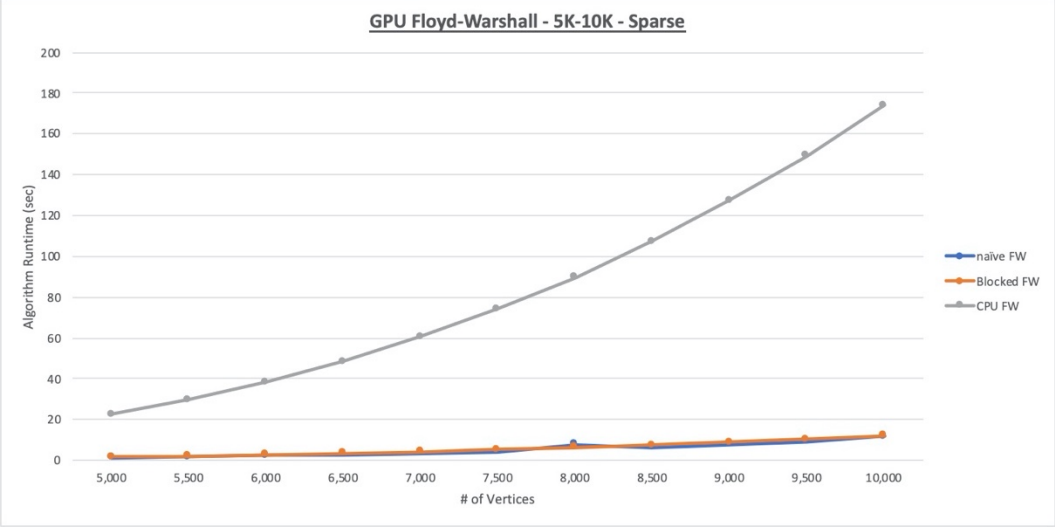

Figure K2. GPU Floyd-Warshall 5K-10K Sparse with 5K-10K CPU overlay Runtime Chart

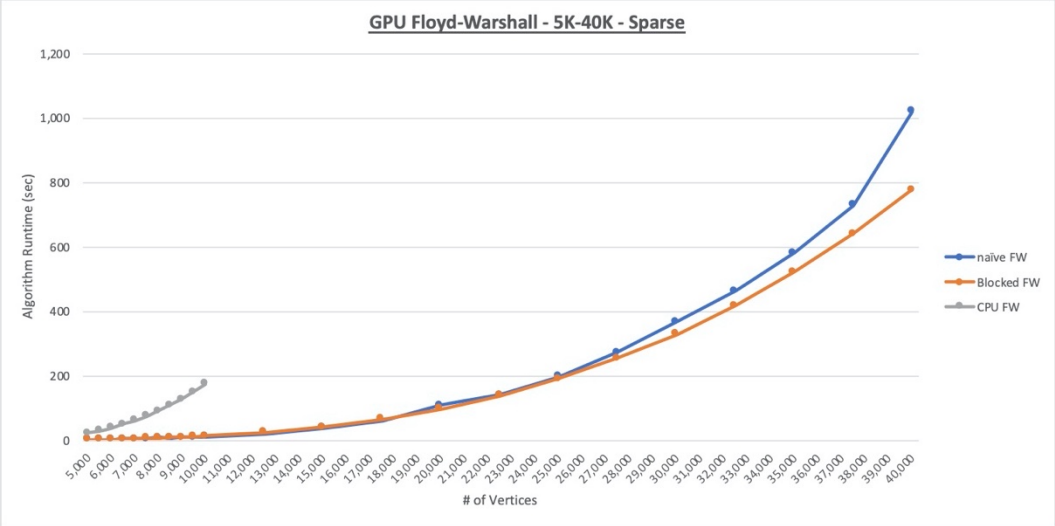

Figure K3. GPU Floyd-Warshall 5K-40K Sparse with 5K-10K CPU overlay Runtime Chart

APPENDIX L

APSP-VIA-SSSP GPU DATA CHARTS

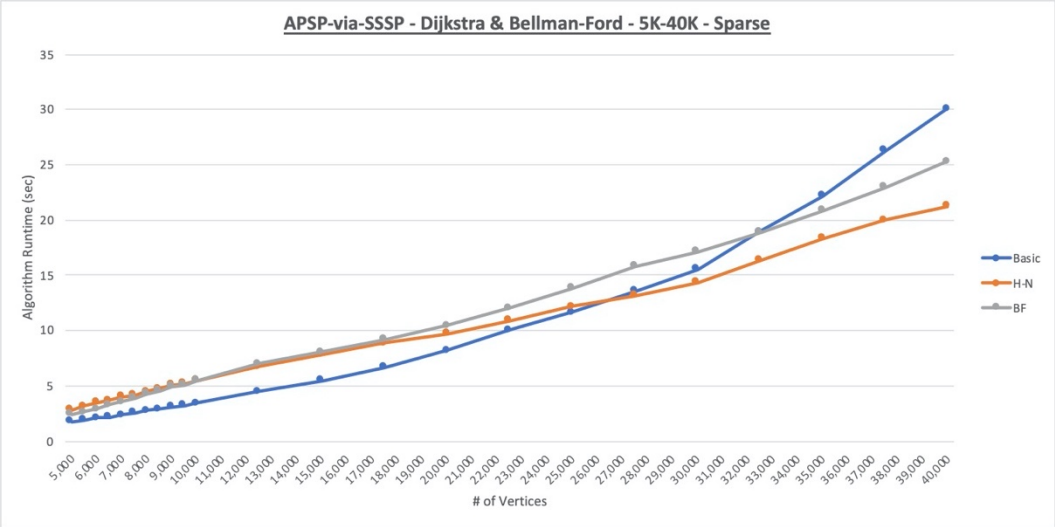

Figure L1. APSP-via-SSSP Dijkstra & Bellman-Ford 5K-40K Sparse Runtime Chart

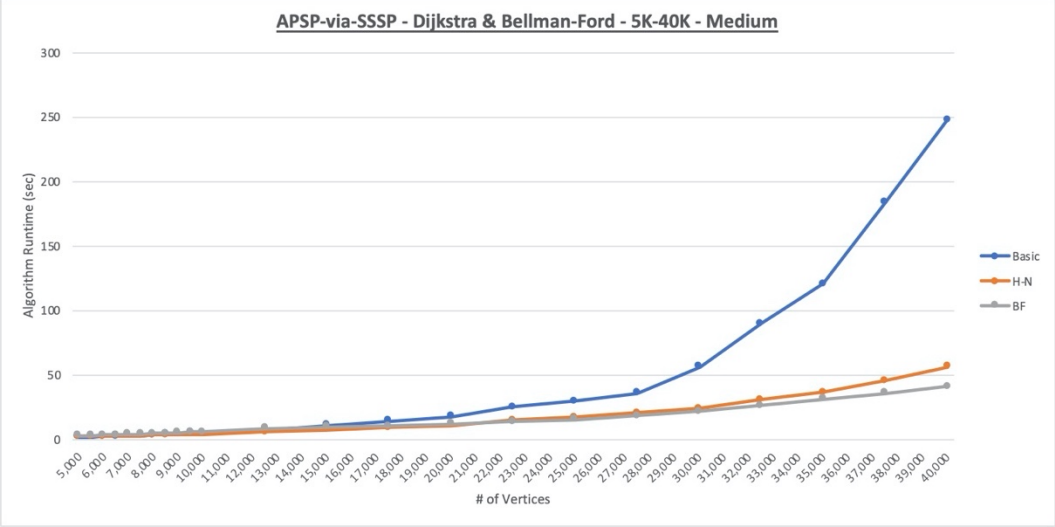

Figure L2. APSP-via-SSSP Dijkstra & Bellman-Ford 5K-40K Medium Runtime Chart

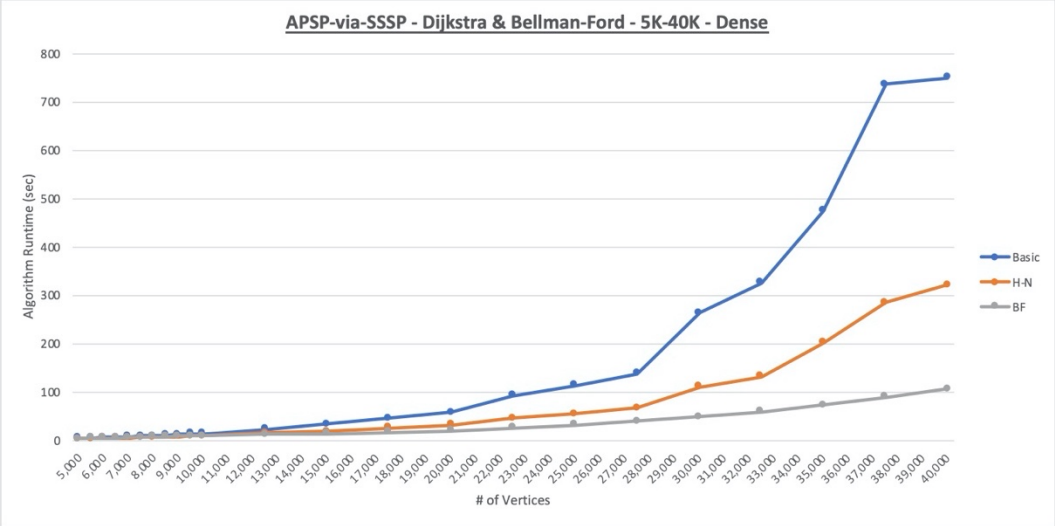

Figure L3. APSP-via-SSSP Dijkstra & Bellman-Ford 5K-40K Dense Runtime Chart

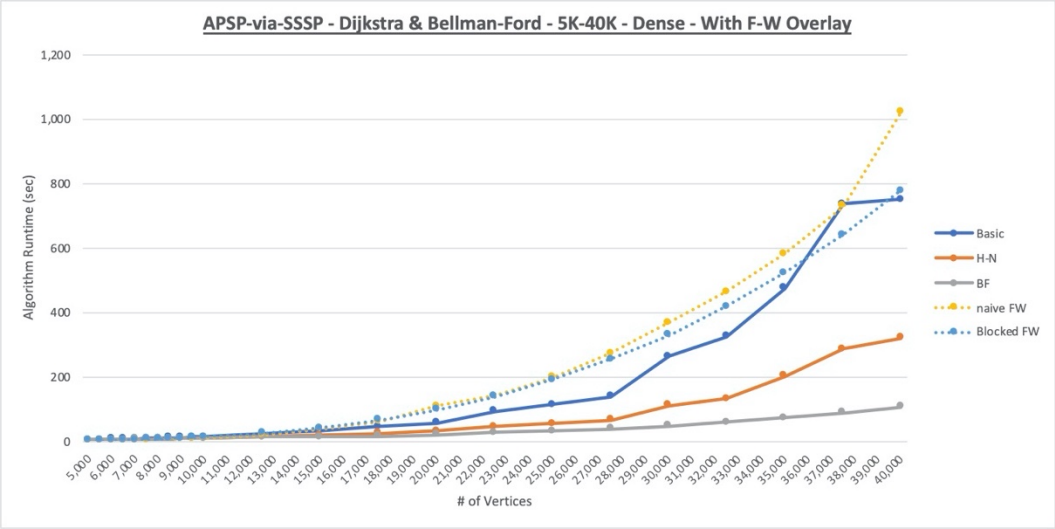

Figure L4. APSP-via-SSSP Dijkstra & Bellman-Ford 5K-40K Dense With F-W Overlay Runtime Chart