Parallelizing Dijkstra's Algorithm: OpenMP and CUDA

Alex Wong

Department of Electrical and Computer Engineering

Boston University College of Engineering

1 INTRODUCTION

1.1 GRAPHS

A graph is a data structure consisting of a set of vertices or nodes[1] {**V**}, and a set of edges {**E**} connecting them. The edges can have weights associated with them, and the nodes can be containers holding multiple variables. As such, graphs are frequently used to represent real-world information; for example, a datacenter network could be represented as a graph, with each vertex being a machine, edges between the vertices indicating connections and weights representing the bandwidth, latency, cable length, or any combination of the three.

In graph analysis, it is frequently required to find the shortest path from one vertex to another. With small graphs this can be easy, but with graphs with more vertices and larger degrees (the number of edges that touch a vertex) analysis can be computationally intensive. For time-critical applications (i.e. load balancing on a network to send a packet to another machine on the least congested path), rapid computation of the shortest path is highly desired. In many cases, this means parallelizing the computation amongst multiple processors.

1.2 Dijkstra's ALGORITHM

Dijkstra's Algorithm is a single-source, shortest path (SSSP) algorithm. In essence, it takes a graph and a starting vertex, and finds the shortest path to every other vertex (though it is frequently used to find a shortest path to only one other vertex, it does not require significant additional time to compute the shortest path to every other vertex due to the nature of the algorithm; as such, in many implementations it simply computes all the shortest paths from one vertex to another).

The algorithm functions by greedily selecting the closest node and getting its distance from the source. It then checks every edge from the closest node to its neighboring vertices, and if the distance to the closest node from the source, plus the weight of the edge from the closest node to the neighbor is less than the distance from the source to the neighbor it updates the distance of the neighbor. This repeats $|\mathbf{V}|$ times until all nodes are "relaxed"; the shortest path can then be found by following the predecessor

[1] Node and vertex are used together in this report, but both reference the same object in a graph (an object that is connected to other objects via edges).

matrix (where elements are indexed by vertex, and each element is the previous node on the shortest path), and the distance can be found by checking the node itself.

Dijkstra's algorithm is currently the fastest SSSP algorithm, capable of running in O(|**E**| + |**V**|*log(|**V**|)) time if certain data structures are utilized; as such, Dijkstra's is a frequently-used algorithm for everything from packet routing to GPS navigation. However, Dijkstra's algorithm works on several assumptions. For one, Dijkstra's algorithm does not work on graphs with negative-weight edges; since it visits each vertex once and only once, a negative-weight edge could cause the algorithm to return the incorrect shortest path.

1.3 NOTES ON THIS IMPLEMENTATION

This implementation of Dijkstra's algorithm does not take advantage of optimized data structures; as such, the running time is $O(|\mathbf{V}|^2)$. This implementation also makes use of an adjacency matrix representation of the graph; an element [U][V] represents the weight between vertices U and V. The graph in this algorithm is also undirected and fully connected. That is, any vertex can be reached from any other vertex. All code-implementations of Dijkstra's algorithm (serial, OpenMP, CUDA) make use of the following shared arrays and matrices (CUDA variables are denoted by a "gpu_" prefix):

- graph: A VxV-sized matrix. Each element (U, V) denotes the weight of an edge from U to V; if there is no edge, then the weight is zero.
- node_dist: A V-length array initialized to infinity and updated by Dijkstra's algorithm. Each element V denotes the distance from the source node to V.
- parent_node: A v-length array initialized to -1. Each element V denotes the previous node on the shortest path from the origin to V; the value -1 does not correspond to any node, and is thus considered to be NULL.
- edge_count: A V-length array initialized to 0. Each element denotes the degree of the indexed node (the number of edges that touch the node). This is not used in Dijkstra's itself, but is instead used in the construction of the graph to ensure it is connected and the degree requirements are met.
- visited_node: A V-length array initialized to 0. This determines whether or not a node has been visited in Dijkstra's algorithm, and is in essence a pseudo-boolean array.
- pn_matrix and dist_matrix: 4xV matrices, where each row is a parent_node or node_dist array for a particular implementation of Dijkstra's algorithm. This is used to ensure that all implementations return the same results for a shared start vertex.

2 IMPLEMENTATIONS

2.1 SERIAL (SINGLE-CORE)

The serial version of Dijkstra's algorithm was not designed to be particularly efficient, as its main role is to serve as a baseline to the OpenMP and CUDA variants. It uses a double for-loop to run Dijkstra's algorithm. On each iteration of the first for-loop, the program finds the closest node by iterating through the node_dist array; if the distance to a node is less than the previous lowest distance, it sets that as the new lowest distance and remembers the index (i.e. the vertex with the lowest distance). At the end the

node returned is marked as visited. The program then runs a second for-loop, this time iterating through the graph matrix with the row indexed to the closest node. On each iteration, the program checks if there exists an edge to the other node and if the other node is unvisited. If both conditions are met, it then checks to see if the distance from the source to the current node plus the distance from the current node to the neighbor results in a lower distance than the neighbor's current distance, and if so updates the neighbor's distance to reflect the lower value and sets the predecessor of the neighbor to be the current node.

2.2 OpenMP (Multiple Threads)

The OpenMP implementation is based off the serial version with several key changes to increase performance. The serial code was divided into two parallelized parts; the first part is the closestNodeOMP function. In the serial version, this sifted through the entire node_dist array element-by-element to find the closest node. The OpenMP implementation parallelizes this by breaking down the inner for-loop between multiple threads. Each thread finds the minimum of a subset of node_dist, essentially splitting the work amongst themselves. The threads then update the global closest node, which is marked with a critical pragma to ensure no data hazards exist.

The second part is the relax function. Again, the work within the for-loop is split among multiple threads, with each thread relaxing to a different vertex. Since there are no data dependencies, all that is needed is a barrier at the end in order to ensure all threads have finished relaxing edges so that a new closest vertex can be found.

2.3 CUDA

Threads in CUDA can be synchronized with a __syncthreads() function call; however, this only applies to the threads within a block. As there is no function call to sync *all* the threads amongst all blocks and in the entire grid, the only solution is to split the kernel when a global thread synchronization is necessary and call a new kernel to continue. Though it would be ideal to have a single device kernel for Dijkstra's algorithm, it would only allow for a single block of threads to be used, which would massively impact any performance improvement. Thus, the CUDA implementation is split into two separate kernels to ensure global synchronization.

The first step in the CUDA implementation is the closestNode() kernel, which returns the index of the node with the smallest distance from the source that has not been visited. This is in essence a reduction operation performed on an array (node_dist) with additional conditions; since this step is performed |V| times, it is a good target for parallelization.

The next step is the relax kernel. |**V**| threads are spawned, one for each vertex, and the relax operation is performed on these vertices from a common source vertex (the closest node previously found by the closestNode() function). As there are no data hazards involved in this step (the only variables involved are a read-only source vertex, and a neighbor vertex that is unique to each thread) there is no need for explicit synchronization.

3 RESULTS:

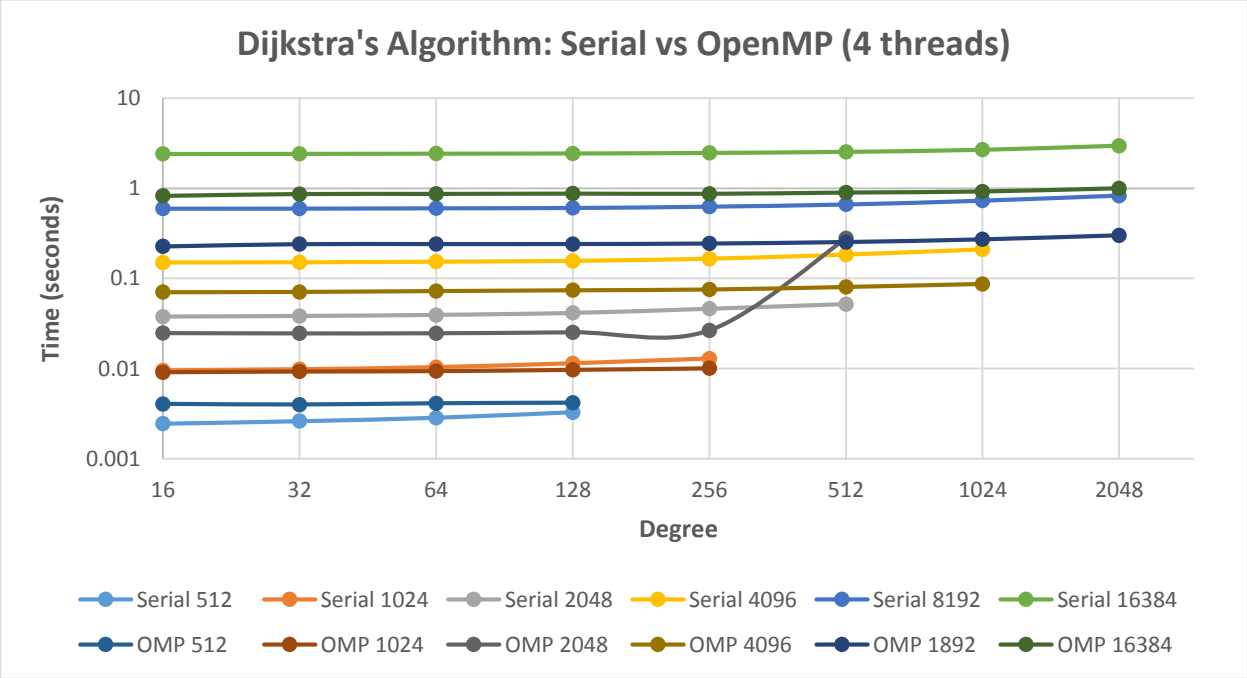
3.1 GENERAL NOTES:

For all iterations, the max weight of each edge was set large enough to ensure that there would be one distinct shortest-path solution for each graph. That is, there would not be a chance of having two paths to a node, each with equal length, or if so then the number of equal-length paths would be very small. If two paths of equal length did exist, then it is possible that the OpenMP implementation would return different predecessor values depending on which thread most recently relaxed to that vertex. The random seed was also fixed each time to ensure reproducibility, and edge values consisted of floats. Each iteration varied the number of vertices and the degree of each vertex.

The OpenMP results were obtained on a computer with an 8-core AMD FX-8150 CPU, clocked at 3.611GHz. All results were returned as the number of cycles needed to execute the program; additional error checking was conducted, but this was done after the timer had stopped.

The CUDA results were obtained on a different workstation. The serial results were executed on a dual-core Intel Xeon W3505 CPU, clocked at 2.53GHz. The CUDA results were executed on an Nvidia GeForce GTX 650Ti, with 1GB of DDR5 memory clocked at 2.7GHZ and a 128-bit memory bus (86.4GB/s). The card contains 768 CUDA cores (streaming processors) clocked at 928MHz.

3.2 OPENMP RESULTS:

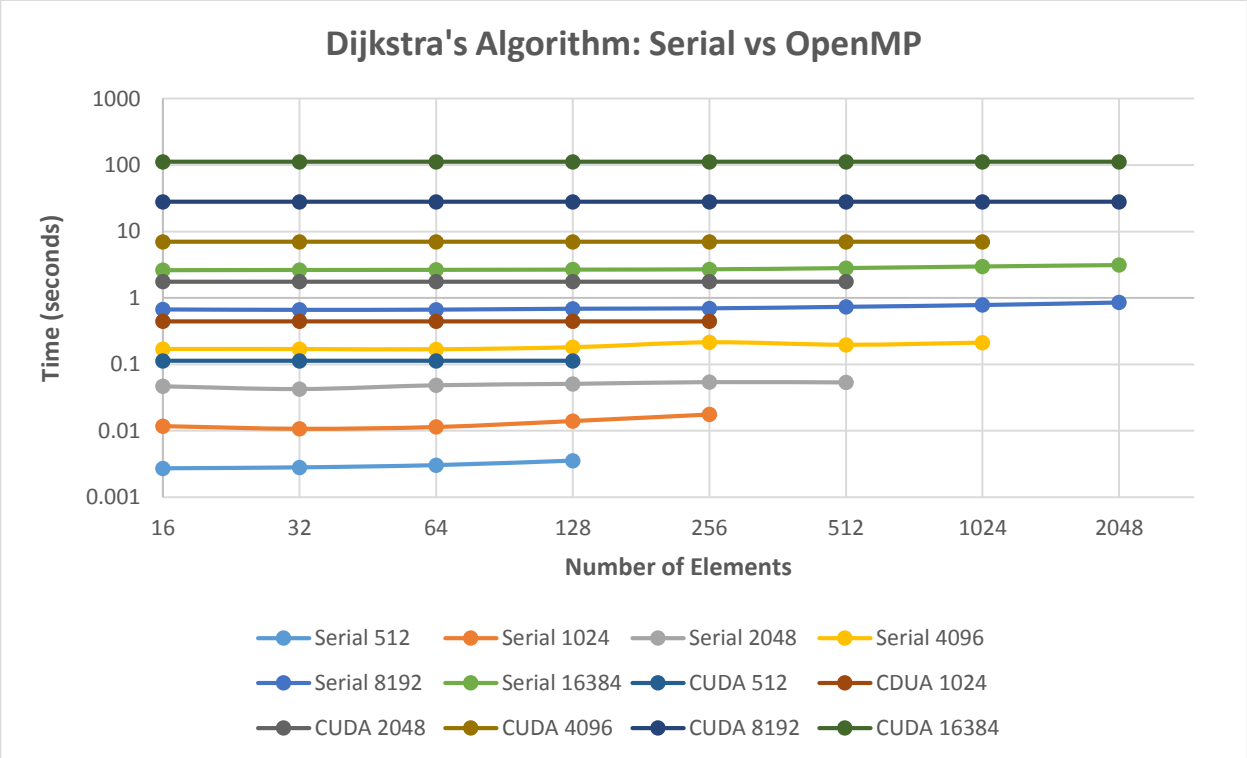


Average Dijkstra's Execution Time (seconds)

Vertices	512	1024	2048	4096	8192	16384
Serial	**0.002791**	0.010812	**0.042394**	0.166766	0.655626	2.537073
OpenMP	0.004078	**0.009489**	0.067253	**0.075668**	**0.251889**	**0.887887**

The results are influenced by two factors: the number of vertices, and the degree of each vertex. The number of vertices has a more direct effect on execution times, while the degree has less of an influence with lower values, but a noticeable effect at larger values. The results of the OpenMP implementation of Dijkstra's algorithm are in line with expectations. For graphs with fewer vertices and lower densities (VERTICES < 1024, DEGREE < 256), the overhead involved with creating the threads exceeds any improvement in performance, leading to longer execution times versus the single-thread, single-core implementation. However, as the number of vertices grows the improvement is more noticeable. With 4096 vertices, the OpenMP implementation is 2.204 times as fast as the serial implementation; with 8192, OpenMP is now 2.603 times as fast, and with 16384 vertices OpenMP is 2.857 times as fast. Though these graphs are relatively small in comparison to examples done by other researchers, the results indicate that Dijkstra's algorithm can be parallelized to improve performance, and it is expected that as the graphs get denser and grow in the number of vertices that OpenMP will have a larger impact.

3.3 CUDA RESULTS:



Average Dijkstra's Execution Time (sec)

Vertices	512	1024	2048	4096	8192	16384
Serial	**0.003032**	**0.013137**	**0.049391**	**0.187544**	**0.720557**	**2.777463**
CUDA	0.113114	0.444623	1.761672	7.012428	27.983197	111.812742

During the course of the experiment, several issues arose with the program. In all implementations, the resulting parent_node and node_dist arrays are copied into a matrix for recordkeeping; this allows the program to check for consistency. If both the programs execute correctly, then the parent_node arrays should be equal; the node_dist arrays may vary in value due to the nature of floating-point numbers, but this should also be reflected in the parent_node array. Interestingly enough, while the node_dist arrays were identical the parent_node arrays were corrupted. This occurred *only* with the NVCC compiler; when the exact same code for the serial and OpenMP implementations was compiled and executed with the GCC compiler there were no errors. Upon comparing the arrays, it became clear that the parent_node array for the OpenMP was corrupted with randomized data that could not have been generated by the program itself. Attempts were made to debug this, but in the end the OpenMP version was left out of the CUDA file; as such, the CUDA results are compared only to the serial implementation, though both the serial and CUDA versions use the same graph.

There were also additional issues with the CUDA implementation of the closestNode() function. Originally a version using the CUDA reduction template was used, but as with the OpenMP implementation there were issues regarding memory corruption when this function was used. As a result, a backup function was tested that was essentially the serial implementation rewritten for a single-thread, single-block kernel. Though abysmally inefficient, this provided a fully functional closestNode() implementation that gave the expected results, and thus this was used instead.

As a result of these issues, the results from the CUDA implementation are less than ideal. The CUDA implementation is less affected by the degree of the vertex, primarily due to the fact that each thread works on a single variable in parallel; with the serial and OpenMP implementations, each thread works on multiple vertices and divergence can cause performance penalties. However, the CUDA implementation was clearly worse than the serial version for all vertices and degrees; in the extreme case of 16,384 vertices CUDA was 50x slower than the serial implementation, and by extension almost 150x slower than the OpenMP implementation.

The performance penalty is almost certainly a result of the closestNode() function. When the number of vertices doubles, the first for-loop must run for twice the number of iterations, and the closestNode() function must operate on twice as many elements; in essence, every time the number of vertices doubles the closestNode() function must do four times as much work overall. This trend is clearly reflected in the table of average times; with each doubling of the vertices the time for the CUDA implementation roughly quadruples.

4 CONCLUSION AND NEXT STEPS

Graph algorithms are frequently used for all sorts of computations; as a result, improving the performance of graph algorithms is highly desired. However, due to the nature of many graph algorithms it is difficult to parallelize them; branch divergence and data hazards are two prime examples of this.

The results of these experiments indicate that while difficult, it is possible to parallelize Dijkstra's algorithm to improve performance. A naïve OpenMP implementation was able to improve execution time by almost a factor of 3 for small graphs, and shows promise for larger graphs with even better improvements possible as more threads are used. However, the CUDA implementation suffers from extreme performance degredation. While the relax() function is easily parallelized, the reduction function requires some more work in order to increase its efficiency. Due to compiler issues, this was unable to take place; however, with a more efficient reduction function it is likely that CUDA will easily be able to beat the serial implementation and OpenMP.