Parallel Programming for Single-Source Shortest Path : OpenMP and CUDA

Minjian Liu
minjian.liu@jhu.edu
EN.601.620 Parallel Programming, Graduate Project

ABSTRACT

Single-source shortest path (SSSP) is one of the most common problems in graph analytics. Its applications include problems in road networks, electronic design, community detection, logistics, power grid contingency analysis and so forth. In this report, we explore two parallel programming ways (OpenMP and CUDA) for the SSSP problem, discuss about how the implementation details are different from each other and conduct an experiment to compare their performance on different sizes of graphs.

1. INTRODUCTION

Graph is frequently used as a model to represent information in the real world, for example, a social network can be represented as a graph with people being nodes and friendships being edges. And graph analytics is one of the growing applications in the parallel programming field. In graph analytics, it is frequently required to find the shortest path from one node to another, normally known as the single-source shortest path (SSSP) problem. Applications of SSSP include problems in road networks, electronic design, community detection, logistics, power grid contingency analysis and so forth. With a small graph, solving SSSP is easy and fast. However, the data in the real world are typically modeled as a graph with thousands or even millions of nodes and edges. Solving SSSP in such a graph requires parallel programming to achieve high performance analytics. Therefore, we will explore two different methods of parallel programming (i.e. OpenMP and CUDA) for SSSP in this report. OpenMP utilizes multiple CPUs to achieve parallelism while CUDA takes advantage of the GPU architecture to parallelize computing.

The remainder of this report is organized as follows: Section 2 reviews the background of SSSP. Section 3 describes the implementation details for SSSP. Section 4 conducts an experiment and discusses about the results. Finally, Section 5 gives a conclusion.

2. BACKGROUND

Given a graph $G = (N, E)$, Single Source Shortest Path (SSSP) aims to identify the fastest path from node A to node B through the graph. Algorithms such as the Bellman-Ford algorithm and the Dijkstra's algorithms are typically used to find solutions for SSSP problem.

The Bellman-Ford algorithm are used to solves the SSSP problem in general case in which edge weights may be negative [2]. Bellman-Ford basically focuses more on the edges of the graph, progressively decreasing an estimate distance on the weight of a shortest path from the source to each node of the graph until it achieves the actual shortest-path weight. It consists of N iterations, relaxing all edges in each iteration in arbitrary order. The running time of Bellman-Ford algorithm are $O(NE)$, where N means the number of nodes and E means the number of edges of a graph.

The Dijkstra's algorithm solves the SSSP problem on a weighted and directed graph with all edge weights are non-negative [2]. Dijkstra focuses more on the nodes of the graph, greedily selecting the closest node and getting its distance from the source. It checks every edge from the closest node to its neighbor and relaxes the edges, which repeats N times until it visits all nodes. The running time of Dijkstra depends on the data structure that stores the distance value. The running time is $O(N^2)$ when using an array, $O((N + E)lgN)$ when using a min-priority queue with a binary min-heap and $O(NlgN + E)$ when using a min-priority queue with a Fibonacci heap. We choose the dijkstra's algorithm to solve the SSSP problem in our implementations due to its better time complexity.

In addition, for better ustilizing the GPU architecture, these algorithms are typically transformed into the algebraic form to support matrix and vector operations. However, the algorithm transformation is beyond the scope of this report. For more information, please refer to Jeremy Kepner and John Gilbert's book "Graph Algorithms in the Language of Linear Algebra" [3].

3. IMPLEMENTATION

There are two standard ways to represent a graph $G = (N, E)$: adjacency list and adjacency matrix. In this project, in order to easier transform the graph to CUDA's CSC (compressed sparse column) topology, we use adjacency matrix to represent the graph. The graph created in this project is a connected undirected graph and the weight of each edge is randomly assigned. More details, please refer to the *graph_initialization()* function in the source code.

3.1 Quick Note for Variables

The following explains the specific purposes of some important variables in this project.

- **graph**: An $N * N$ adjacency matrix. Each element (i, j) of the matrix indicates the weight of an edge from node i to node j (zero means no edge).
- **MIN_DEGREE**: An integer macro used to control the density of the graph. It means all nodes in the graph at least have **MIN_DEGREE** edges.
- **edge_degree_array**: An N-length array used to store the edge degree of the node with current index. When creating a graph, this array provides information to whether or not add more edges to the current node.
- **node_distance_array**: An N-length array used to store the distance from the source node to the node with current index (Initialized to infinity and would be updated by running the Dijkstra algorithm).
- **parent_nodes_array**: An N-length array used to store the predecessor of the node with current index in the shortest path (Initialized to -1 and would updated by running the Dijkstra algorithm).
- **visited_nodes_array**: An N-length array used to indicate whether or not the node with current index is visited in the Dijkstra algorithm (zero means not visited, one means already visited).
- **parent_nodes_results_matrix**: A $2 * N$ matrix used to store the results of **parent_nodes_array** for different implementations to check if their results are the same (one row for serial implementation, one row for OpenMP or CUDA implementation).
- **distance_results_matrix**: A $2 * N$ matrix used to store the results of **node_distance_array** for different implementations to check if their results are the same (one row for serial implementation, one row for OpenMP or CUDA implementation).

3.2 Serial Implementation

As the serial implementation serves as a baseline when comparing performance with OpenMP and CUDA implementations and provides correct results to check whether or not the OpenMP and CUDA versions are implemented properly, we do not use optimized data structure such as a min-priority queue with a Fibonacci heap for the implementation. This implementation follows the idea of Dijkstra's algorithm mentioned in the well-known algorithm book written by Thomas H. Cormen [2]. It uses a double for loops to processing the graph and to find the shortest path (refer to the *serial_sssp()* function for details).

The first for loop is the *extract_min()* function to greedily choose the closest node to relax by iterating through the **node_distance_array**. If the distance to the node with current index is less than the lowest distance, then sets the distance as the new lowest distance and returns the current index (i.e. the closest node index), then marks this node as visited.

The second for loop is to iterate through the graph matrix to check the neighbors of the current closest node. If a neighbor is not visited, calculate the distance from the source to the current closest node plus the neighbor's edge weight. If the new distance is less than the original distance from the source to the neighbor, then update the **node_distance_array** and the **parent_nodes_array** with the new distance and the index of the current closest node. This process is known as relaxation.

3.3 OpenMP Implementation

The OpenMP implementation basically parallelizes the two for loops (i.e. choosing the closest node and relaxation) of the serial implementation for multiple threads (refer to the *openmp_sssp()* function for details).

In terms of choosing the closest node, the serial implementation iterates the **node_distance_array** while the OpenMP implementation breaks down the iteration into several small iterations for multiple threads. Each thread works on its own small iteration to find the closest node. Then each thread will enter a critical section to update the global variable to ensure the global variable is the closest node index of the entire **node_distance_array**. Finally, it returns the global variable as the closest node index.

As for the relaxation, it similarly breaks down the iteration for multiple threads. Unlike the previous for loop, each thread relaxes different nodes and updates different indexes of the **node_distance_array** and the **parent_nodes_array**. So there are no data dependencies during the process. Finally, we set a barrier at the end to ensure all threads have updated the two arrays before getting the next closest node.

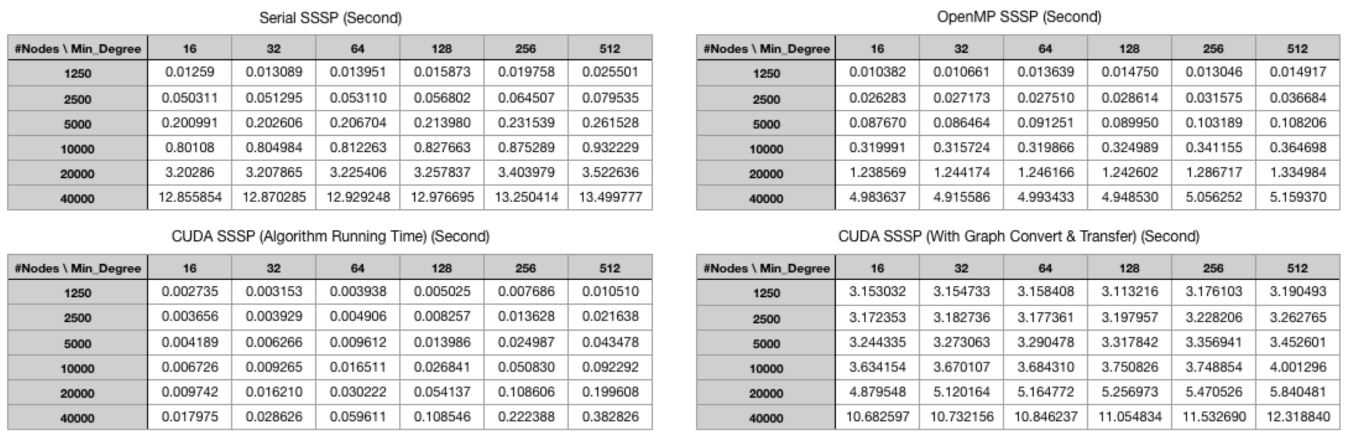
3.4 CUDA Implementation

The CUDA implementation mainly relies on the nvGraph library that was released in NVIDIA CUDA 8.0 aiming to harness the power of GPUs for linear algebra to handle the largest graph analytics problem. The core functionality of nvGraph library is a SPMV (sparse matrix vector product) using a semi-ring model with automatic load balancing for any sparsity pattern [1]. More details about the use of the semi-ring model and GPU graph analytics can be found at [3], [4] and [5].

A typical workflow for using nvGraph is described as follows: first calls *nvgraphCreate()* to initialize the library; then uploads graph data to the library through nvGraph's API including *nvgraphCreateGraphDescr()*, *nvgraphSetGraphStructure()*, *nvgraphAllocateVertexData()*, *nvgraphAllocateEdgeData()* and *nvgraphSetEdgeData()*; the graph data can be uploaded with the CSR (compressed sparse row) and CSC (compressed sparse column) format; afterward runs the graph algorithms on the graph data (e.g. *nvgraphSssp()* for single-source shortest path); at last calls *nvgraphDestroy()* to free resources used by nvGraph (refer to the the *cuda_sssp()* function for details).

Before entering the typical workflow, another important part of CUDA implementation is how to convert the standard graph representation model (i.e. adjacency matrix) to the CSR or CSC format. However, nvGraph does not directly provide such a API to let users easily convert adjacency matrix or adjacency list to the CSR or CSC format. We need to utilize another traditional storage format COO (Coordinate list), which consists of the following three parts:

- **source_indices**: An E-length array used to store the node index of the source of each edge in the graph.



Serial SSSP (Second)

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	0.01259	0.013089	0.013951	0.015873	0.019758	0.025501
2500	0.050311	0.051295	0.053110	0.056802	0.064507	0.079535
5000	0.200991	0.202606	0.206704	0.213980	0.231539	0.261528
10000	0.80108	0.804984	0.812263	0.827663	0.875289	0.932229
20000	3.20286	3.207865	3.225406	3.257837	3.403979	3.522636
40000	12.855854	12.870285	12.929248	12.976695	13.250414	13.499777

OpenMP SSSP (Second)

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	0.010382	0.010661	0.013639	0.014750	0.013046	0.014917
2500	0.026283	0.027173	0.027510	0.028614	0.031575	0.036684
5000	0.087670	0.086464	0.091251	0.089950	0.103189	0.108206
10000	0.319991	0.315724	0.319866	0.324989	0.341155	0.364698
20000	1.238569	1.244174	1.246166	1.242602	1.286717	1.334984
40000	4.983637	4.915586	4.993433	4.948530	5.056252	5.159370

CUDA SSSP (Algorithm Running Time) (Second)

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	0.002735	0.003153	0.003938	0.005025	0.007686	0.010510
2500	0.003656	0.003929	0.004906	0.008257	0.013628	0.021638
5000	0.004189	0.006266	0.009612	0.013986	0.024987	0.043478
10000	0.006726	0.009265	0.016511	0.026841	0.050830	0.092292
20000	0.009742	0.016210	0.030222	0.054137	0.108606	0.199608
40000	0.017975	0.028626	0.059611	0.108546	0.222388	0.382826

CUDA SSSP (With Graph Convert & Transfer) (Second)

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	3.153032	3.154733	3.158408	3.113216	3.176103	3.190493
2500	3.172353	3.182736	3.177361	3.197957	3.228206	3.262765
5000	3.244335	3.273063	3.290478	3.317842	3.356941	3.452601
10000	3.634154	3.670107	3.684310	3.750826	3.748854	4.001296
20000	4.879548	5.120164	5.164772	5.256973	5.470526	5.840481
40000	10.682597	10.732156	10.846237	11.054834	11.532690	12.318840

Figure 1: Running Time of Different SSSP Implementations

- **destination_indices**: An E-length array used to store the node index of the destination of each edge in the graph.
- **weights**: An E-length array used to store the weight of each edge in the graph.

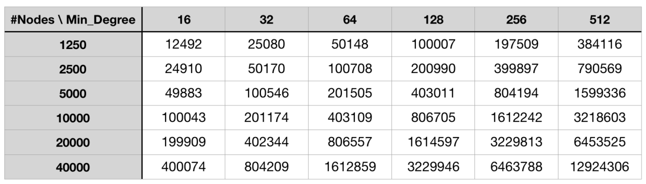
Thus, we need to iterate the graph matrix first to create the corresponding **source_indices**, **destination_indices** and **weights**, then use *nvgraphConvertTopology()* to convert the COO format to the CSR or CSC format. After that, enters the typical workflow to run the graph algorithms.

4. EXPERIMENT

The experiment is conducted at a p2.xlarge instance deployed by the Amazon Web Service (AWS), with 61 GiB memory, an Intel Xeon E5-2686 v4 (Broadwell) processor having 4 vCPUs and 2.3GHz clock speed, and a NVIDIA K80 GPU having 2496 parallel processing core and 3.7 CUDA compute capability.

4.1 Experiment Design

The experiment aims to compare the performance of three SSSP implementations (i.e. serial, OpenMP and CUDA) that runs on different sizes of graphs. Figure 2 shows the number of nodes and the number of edges of all graphs that are tested in this experiment. We use **MIN_DEGREE** per node to control the density of the graph (total number of edges). In the source code, to ensure reproducibility of the experiment, we do not set the srand(), so the rand() seed is set as if srand(1) were called at program start.



#Nodes \ Min_Degree	16	32	64	128	256	512
1250	12492	25080	50148	100007	197509	384116
2500	24910	50170	100708	200990	399897	790569
5000	49883	100546	201505	403011	804194	1599336
10000	100043	201174	403109	806705	1612242	3218603
20000	199909	402344	806557	1614597	3229813	6453525
40000	400074	804209	1612859	3229946	6463788	12924306

Figure 2: Graph Information: number of nodes and number of edges

4.2 Result and Discussion

In the CUDA implementation, we are required to convert the graph data into the CSC format and transfer it to the GPU memory, which takes certain amount of time to complete. However, in the real world applications, we may keep the graph data in the memory for a long time and repeatedly run graph algorithms on it. So we focus more on the discussion about the algorithm running time and also provide the total running time of the CUDA implementation (including time for graph convert & transfer) as a reference.

Figure 1 shows the specific data regarding to the running time of Different SSSP Implementations and Figure 3 illustrates how trends look like in terms of the number of nodes and the MIN_DEGREE per node.

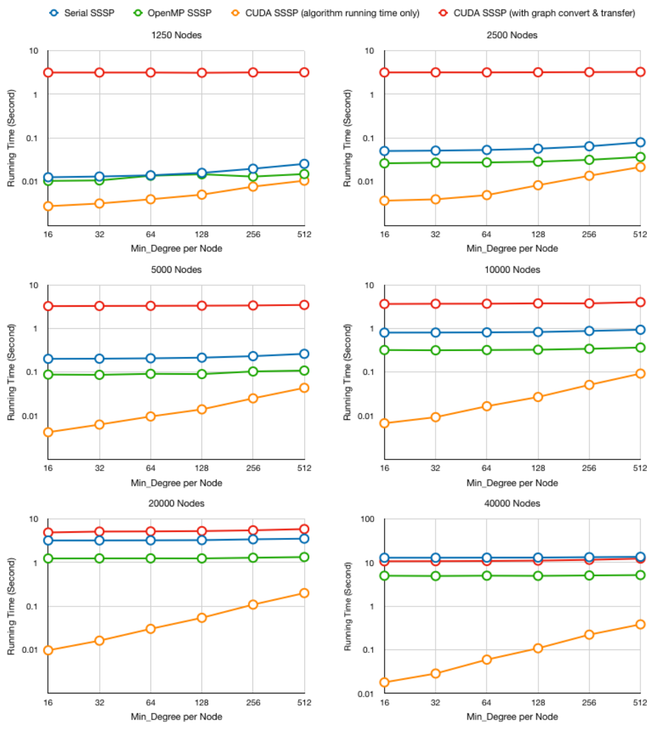
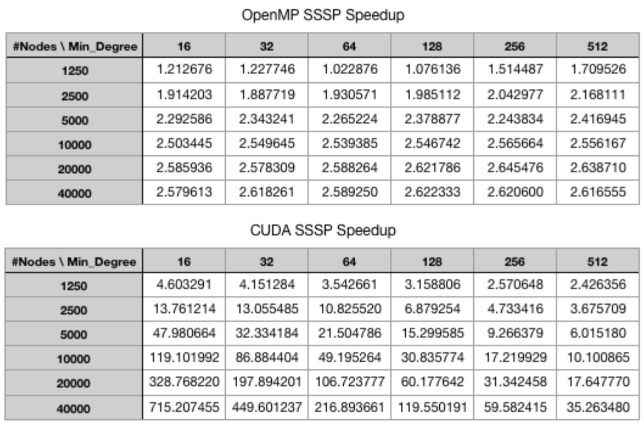


Figure 3: Line Chart for Running Time of Different SSSP Implementations

The experiment results are influenced by two factors: the number of nodes and the density (**MIN_DEGREE** per node). The number of nodes has a significant effect on all three implementation regarding to the running time, while the **MIN_DEGREE** per node affects more on the CUDA implementation but has little effect on the serial and OpenMP implementations. The reason for that is because we do not use optimized data structure in the serial and OpenMP implementations, so the time complexity of serial and OpenMP SSSP are both $O(N^2)$. Then for the CUDA implementation, the algebraic SSSP algorithm uses $O(N)$ dense vector, sparse matrix multiplications, so the running time is $O(NE)$ comparisons and additions [3].

However, discussing the time complexity is not very useful when the OpenMP implementation only has 4 vCPUs for parallel processing but the CUDA implementation has 2496 cores for parallel processing. With more parallel processing cores, the algorithm running time of CUDA SSSP is far less than the OpenMP SSSP, even though it does take certain amount of time for graph convert and transfer. Nonetheless, the gap between the total running time of CUDA SSSP and that of the OpenMP SSSP is getting smaller when the graph grows larger.


OpenMP SSSP Speedup

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	1.212676	1.227746	1.022876	1.076136	1.514487	1.709526
2500	1.914203	1.887719	1.930571	1.985112	2.042977	2.168111
5000	2.292586	2.343241	2.265224	2.378877	2.243834	2.416945
10000	2.503445	2.549645	2.539385	2.546742	2.565664	2.556167
20000	2.585936	2.578309	2.588264	2.621786	2.645476	2.638710
40000	2.579613	2.618261	2.589250	2.622333	2.620600	2.616555

CUDA SSSP Speedup

#Nodes \ Min_Degree	16	32	64	128	256	512
1250	4.603291	4.151284	3.542661	3.158806	2.570648	2.426356
2500	13.761214	13.055485	10.825520	6.879254	4.733416	3.675709
5000	47.980664	32.334184	21.504786	15.299585	9.266379	6.015180
10000	119.101992	86.884404	49.195264	30.835774	17.219929	10.100865
20000	328.768220	197.894201	106.723777	60.177642	31.342458	17.647770
40000	715.207455	449.601237	216.893661	119.550191	59.582415	35.263480

Figure 4: Speedup of OpenMP and CUDA SSSP Implementations

In addition, the time complexity is helpful when discussing the speedup. As shown by the Figure 4, the best speedup of the OpenMP implementation is about 2.6 (with 0.65 parallel efficiency) when handling the graph with 20000 or 40000 nodes and 256 or 512 **MIN_DEGREE** per node. The speedup trend is getting better when having more nodes in the graph as well as having more edges. For the CUDA implementation, the best speedup is about 715 (with 0.29 parallel efficiency) when dealing with the graph with 40000 nodes and 16 **MIN_DEGREE** per node. The speedup trend is getting better with more nodes but getting worse with more edges.

5. CONCLUSION

In this project, two parallel implementations (OpenMP and CUDA) for the SSSP problem have been explored. The experiment conducted to compare the performance of the SSSP implementations shows both parallel implementations can help to reduce the running time. However, the CUDA SSSP needs take certain amount of time for graph convert and transfer. If the graph data is represented by the standard ways (i.e. adjacency list or adjacency matrix) and the user does not need to keep the graph data in memory for repeatedly running SSSP, then the OpenMP parallel implementation may be the best choice to improve performance. If the graph data is represented in COO, CSC or CSR format and the user needs to run SSSP on the graph data over and over again, then go for the CUDA implementation which can largely improve the performance due to taking advantage of the GPU architecture.

6. REFERENCES

[1] nvGraph libraray user's guide. `https://docs.nvidia.com/cuda/pdf/nvGRAPH_Library.pdf`.

[2] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[3] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[4] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[5] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3, 2017.