# Implementation and Analysis of Parallel Shortest Path Graph Algorithms

Brandon Autrey, Anthony Bolton, Ryan Harrigan

***Abstract*—Finding the shortest path is very important in many applications. This is computed by graph algorithms which can be computationally intensive on large graphs or datasets. Computational speed is important for real-time applications such as pathing for games, traffic analysis for GPS systems, and geodetics. In this paper we explore the parallelization of A*, Dijkstra, and Bellman-Ford using OpenMP on an SMP architecture. The performance of these algorithms are compared against their serial versions to show their parallelization efficiency. These algorithms are then experimented on real datasets and generated datasets to show the shortest path.**

***Index Terms*—OpenMP, SMP, graph algorithms, shortest path, multithreading, Dijkstra's, A*, Bellman-Ford, pathfinding, genetic algorithm**

## I. Introduction

THERE are many ways for graphs to be used. Depending on the purpose of the program, the same data can offer different insights for the user. As there are various ways to represent data, a slurry of algorithms are available for processing. This is interesting because the effectiveness of algorithms are dependent on data and vice versa.

Determining information from graphs is usually accomplished by one of two general approaches. Read-oriented processing usually does not alter vertices/edges. They usually run faster because writing values pauses the traversing process. Path processing is different because a separate memory-space is required to compute some values for decision-making. Many algorithms have been imagined to traverse graphs for different reasons. Such algorithms include Dijkstra's, A*, Prim's, and Floyd-Warshall's. These algorithms were designed for a serial environment. When introduced to parallelism, many difficulties arise.

Read-oriented processing is fair because it usually splits the graph into equal parts for the cores to compute operations in parallel. Read-oriented processing generally does not have contention when performing operations from vertex/edge values because the outputs of threads do not conflict with other thread's input. Shortest path processing, however, introduces most of the difficulty. Threads must share read/write access to shared nodes/edges and their synchronization tends to be serial in nature. Especially in dense graphs, shortest path processing has contention when updating adjacency lists. Both approaches have a variety of applications, but the prevalent concerns are focused on defining areas of possible contention when considered for parallelism.

Parallelizing graph search algorithms is challenging. Challenges are often addressed by the irregularity of data, locality of data in terms of the shared global data among threads, and the partitioning of data among threads. Irregular data causes latency for datum identification. Data locality is difficult to predict, especially when predefined code is absent, because locality often requires computing relationship among many other nodes/edges. Partitioning of data computation requires careful definitions of memory access. Determining the grain of parallelism and boundaries of dependent node/edge values tend to make programming complex because of these general challenges.

This paper will juxtapose serial and parallel implementations of Dijkstra's, A*, Floyd-Warshal's, and genetic algorithms. These algorithms have unique properties in parallel environments and can improve many real-world applications. To demonstrate of these algorithms, this paper will experiment on graph generation for games, road maps of the United States, and LiDAR terrain data. These datasets also have different properties and such will show various runtimes.

Implementations in this paper will often use OpenMP. OpenMP is a set of commands that was intended to ease parallelization. Just like POSIX threads, it is formatted to work on SMP machines, however, it can be incorporated in distributed environments. For the sake of brevity, implementations of algorithms will be limited to SMP environments.

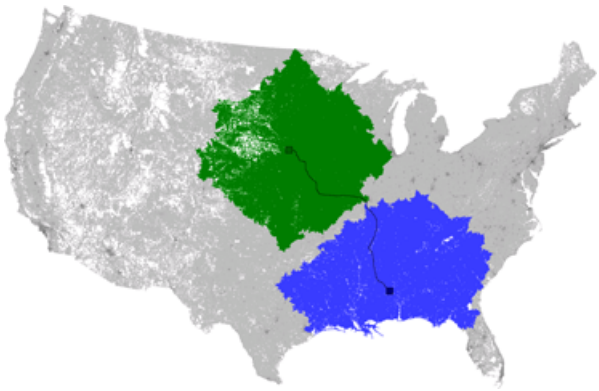
OpenMP takes advantage of global memory available to the computing nodes; allowing each node to be updated as shared memory changes. An OpenMP application begins with the master thread. When the program encounters a parallel region construct, OpenMP automatically creates and distributes threads to computing nodes, overhead requiring time before parallel computation begins. This time must be considered for time performance and is strongly suggested that the parallel sections have large demand [1]. At the end of the parallel region, the created threads are stopped and the master thread continues running. OpenMP specifies parallel regions by pragmas, a directive based standard, so that the source code combines serial and parallel code. The program be compiled with or without the /openmp compiler option. If it is compiled without it, then the pragmas are ignored and the program behaves as entirely serial.

## II. Related Work

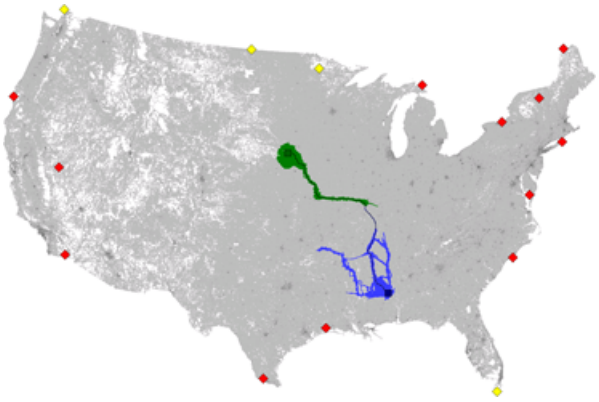
Previously an attempt to parallelize Dijkstra's using OpenMP gained a 10% speedup by making it parallel. This algorithm is a difficult to parallelize to gain a significant speedup because lots of time is being spent on parallelization and synchronization than is actually being spent on the execution of code. The algorithm is not suited for parallelism because

it relies on priority queues [2]. Since the priority queue is the bottleneck in the parallelization of this algorithm, this paper will attempt an implementation of parallelizing the priority queue.

Without even parallelizing the code, there are some optimizations that can be made to the A* algorithm. For example, research has been done discussing the use of precomputed paths for commonly used paths, or using particular landmarks and precalculating paths between them. Additionally, A* can be modified to be a bidirectional search, wherein two searches are conducted "simultaneously" starting at both the origin node and the target node, branching out until they meet in the middle. The word "simultaneously" is in quotes because while it may appear like the bidirectional search is happening concurrently, with only one thread and/or processor the calculations are still happening serially, alternating between calculating the path from the start node and the path from the target node. Andrew Goldberg, principal researcher at Microsoft Research Silicon Valley, along with other colleagues have published several papers on improving various shortest path algorithms [3]. They have used data sets like the North American road network (which is almost 30 million nodes) so it is clear why it is important that they optimize the time efficiency of their shortest path algorithms. They have shown a variety of algorithms traversing this network with randomly selected "origin" and "destination" nodes, and it shows the variety of routes by which these algorithms traverse the network. Clearly, some of them are not very efficient because they visit nodes that do not need to be considered in the first place. The image below is that of a bidirectional search performed with Dijkstra's algorithm with the green area emanating from the origin node and the blue area emanating from the target node [4]. Once they find a meeting point, the path has been calculated and the search stops.



The A* algorithm is essentially Dijkstra's algorithm with an admissible heuristic (also called an optimistic heuristic) for searching, which means it never overestimates the cost of reaching the goal - it's always an underestimate of the cost or the exact cost (this paper explores the Euclidean distance as the cost) [5]. The following image is the same network but now with a bidirectional A* algorithm performed along with precalculated landmarks [5].



Though this implementation of the A* algorithm also involves precalculated landmarks, the reader can see that a great deal of computation and "unnecessary nodes" are ignored when using this P2P (point-to-point) search algorithm versus a simple Dijkstra's.

## III. Parallel Algorithm Analysis

### A. Parallel Dijkstra Algorithm

The Dijkstra's algorithm finds the shortest path from the starting vertex to other vertices in the weighted graph. This algorithm keeps two sets, the set of nodes it has already checked and the set of nodes that it has not checked. When searching the algorithm searches every neighboring vertex from the previously checked vertex. This results in checking every vertex in a circle almost, until the destination vertex is found.

### B. Parallel A* Algorithm

The A* algorithm is a popular pathfinding algorithm used to find the shortest distance between two points and is actually a modification of Dijkstra's algorithm. The A* algorithm uses a heuristic based on estimating the distance from any searched node to the desired target node, and the A* algorithm can be made to be Dijkstra's algorithm simply by setting the heuristic cost to zero for all the nodes. The A* algorithm requires a priority queue for the optimal choices currently held to be possible paths from the origin node to the target node, and when one path 'deeper' in the queue is discovered to be shorter than the shortest path so far, it climbs to the head of the priority queue. Shortest paths algorithms like A* are types of combinatorial optimization problems that can be useful for real-world applications like route planning for shipping paths. Using multiple threads and altering the algorithm to work more efficiently in a parallel environment, we will show that the parallelized version of the A* algorithm is more efficient and can be used to compute the shortest path very quickly for large sets of data. For efficiency of runtime, it is important that your graph be connected (i.e., that all your nodes are connected to one another by way of some path in the graph and that no nodes are isolated) or that at least your start node and your end node are part of the same "connected component" of the graph. If your start node and end node are NOT part of the same connected component, your search (bidirectional or otherwise) will search through the ENTIRE search space

within the connected component(s) of the graph that has your starting node (and within the full connected component that has your ending node, if doing a bidirectional search) - a very costly operation. The pseudocode for the serial version implementation of the A* algorithm used in this paper is as follows:

For the purposes of parallelization, it will be necessary to create a concurrent priority queue data structure, as well as a parallelized dynamic array (Java was utilized, so for the serial version of the algorithm, ArrayList was used for this purpose). A trivial way to parallelize this algorithm a bit would involve parallelizing a bidirectional search involving only two processors and two threads, initially. One thread will be responsible with searching starting from the origin node, and the other thread will be responsible for searching starting with the destination node. When one of the threads has a path that exceeds half the heuristic distance to the other thread's origin node, it will begin to check whether the nodes it's adding to the path have also been added to the other thread's path.

Additionally, the for-loop for searching a node's neighbors can be parallelized easily in OpenMP with their pragma-for-loop abilities, so it will take the threads available to your computer and put them to work in the for-loop. In general, the work of the for-loop can be divided amongst individual threads waiting to perform calculation-work in a thread pool, and then when their calculation is complete, they can return to the thread pool, awaiting being used for the next for-loop of either of the bidirectional searches. Additionally, there is the possibility of having a separate thread go through the combined set of nodes included so-far in the two paths (since a bidirectional search will be performed) to stop the searches once it encounters a node that states it is in both thread's paths. It is the attempt of this paper to show that these changes to the algorithm will result in an optimized runtime of the algorithm on large datasets.

### C. Parallel Bellman-Ford Algorithm

## IV. Experimentation and Testing

## V. Conclusions

** A recap of the algorithms and how their parallelized versions are different from their serial counterparts, and then a discussion of how our implementations were similar to and differed from publicly known parallelized versions. **

## VI. Future Directions

** OpenMP vs CUDA or OpenCL OpenMP + MPI to increases parallelism **

## Appendix

### A. Completed

- Figuring out how to share the graph for all threads
- Learning OpenMP

### B. Completed Tasks

- Serial version of A* algorithm
- Read articles on parallel algorithms, OpenMP, and parallel programming

### C. Remaining Tasks

- Parallelize all three algorithms
- Write working serial Dijkstra and Bellman Ford
- Find and write other working parallel algorithms
- Incorporate into real applications
  - Generated graphs
  - Roadmaps/City traffic (real data)
  - Terrain Maps (real data)
- Experiment and document results
- Make graphs comparing performance

## References


[1] Cyberinfrustructure tutor. NCSA. [Online]. Available: cututor.org

[2] A. E. K. I. L. E. N. N. Nadira Jasika, Naida Alispahic, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," in *MIPRO, 2012 Proceedings of the 35th International Convention*, May 2012, pp. 1811–1815.

[3] R. F. W. Andrew V. Goldberg, Haim Kaplan, "Reach for a*: Efficient point-to-point shortest path algorithms," Microsoft Corporation, Technical Report, October 2005.

[4] J. Chang. (2009, July) Making the shortest path quicker. Microsoft. [Online]. Available: http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx

[5] R. Eranki. (2002) Pathfinding using a* (a-star): Traversing graphs and finding paths. [Online]. Available: http://web.mit.edu/eranki/www/tutorials/search/