Parallel Dijkstra's Algorithm

Po-Kai Wang
Department of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
adkevin3307@gmail.com

Cheng-Tes Ho
Department of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
djjjimmyyy@gmail.com

1 ABSTRACT

We use OpenMP, POSIX Thread (Pthread), and C++ Thread to implement Dijkstra's Algorithm and analyze the result, finding out why the speedup is not close to ideal.

2 INTRODUCTION

The shortest path problem is finding the shortest path between nodes in a graph, which may represent the problem like road networks or Internet connection. The problem is important when the cost of connections is really expensive.

Finding the shortest path is a well-known problem in the algorithm area, and it already has many solutions for both single-source path and all-pairs shortest path. Although most of the solutions for the shortest path problem have already reduced time complexity, most of them will struggle in a large graph.

The algorithm we choose is Dijkstra's algorithm. The algorithm is a popular algorithm which can deal with single-source shortest path issue, so we are interested in speed up the algorithm. Moreover, we hope we can widely apply the parallel method to other similar algorithms.

The step of Dijkstra's algorithm is as follow, Given graph $G = (V, E)$, $visited =$. First, Extract the closest vertex, which needs to compare and select the minimal distance vertex u which is not in the visited set, then we put it in the visited set. Second, relax edges distance value, which updates the distances of vertices according to current closet vertex u, the formulate of the update is $distance[v] = min(distance[v], distance[v]+weight(u, v))$, repeat the above steps $V - 1$ times then the shortest path will be get.

The Time Complexity Extract minimum distance is $O(V^2)$, the time complexity of Relax edges is $O(E)$, so total Time complexity is $O(V^2)$

3 PROPOSED SOLUTION

$$\begin{aligned}
&d : distances \\
&w : weight \\
&gcv : globally\_closest\_vertex \\
&d[v] = min(d[v], d[gcv] + w[v, gcv])
\end{aligned} \tag{1}$$

Given a graph, Let $G = (V, E)$ be a directed graph, and the number of threads is T, first, we divide vertex into equal length vertex set and assign it to each thread. Each thread will get $\frac{|E|}{T}$ number of vertexes. Each thread identifies its local closest vertex in its set to the source vertex. Store this value in shared memory which is shared among threads.

Second, we use a binary tree reduction Figure 1 to select the globally closest vertex. Binary tree reduction is an algorithm that uses half of the activate thread to compare the items of shared memory. initially, the number of activated threads in our case will be the number of threads in the program, that is $|T|$, then it will be divided by two each iteration. We need to take care of the case that the number of the vertex is not the power of two, so we check whether the number of activated threads in this iteration is odd at the beginning of each iteration, if it is odd, we use first activate thread to handle last activate thread value, then the last activate thread will become inactivate, so the remaining number of the activated thread will become even. the iteration will repeat until the number of activated threads become one.

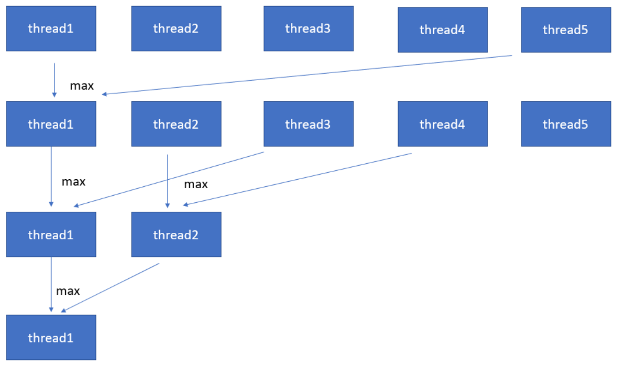

Figure 1: Reduction Compare

Finally, each process updates the value of vertexes in its local vertex set according to the new global closet vertex, let distances$[v]$ is the current distance from global to vertex v, then we can update distances$[v]$ by the formulate 1

The time complexity of the first step and the third step is $O(\frac{|V|}{T}$, it is less than the serial version which is $O(|V|)$ since we divided the for loop linear to threads. The workflow is like Figure 2.

The time complexity of the second step is $O(log(|V|))$ since we let not only master thread but also worker threads to compare to find the global minimum, and the problem size is divided by two each iteration, so the time formulate can be written like $t(T) = t(\frac{T}{2}) + O(1)$, according to the master theorem, the time complexity is $O(log(T))$, it is less than the serial version which is $O(|V|)$.

4 EXPERIMENTAL METHODOLOGY

In our experiments, we use the datasets from DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science) for the shortest path problem. We choose four areas in the USA, which are Florida, Colorado, San Francisco Bay Area, and New York City. The graphs' information are shown in Table 1.

Figure 2: Block Diagram

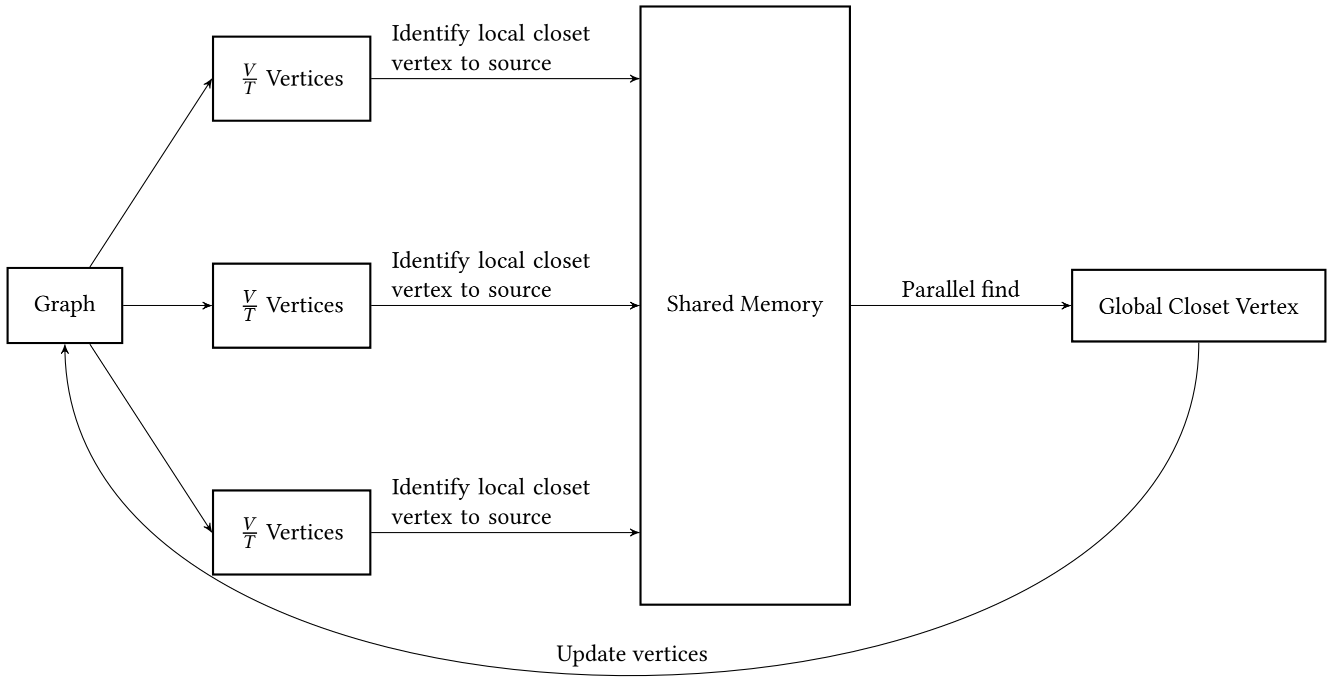

Table 1: Graph Information

Area	# of Nodes	# of Edges	Min # of Edges	Max # of Edges	Avg # of Edges
Florida	1,070,376	2,712,798	1	8	2
Colorado	435,666	1,057,066	1	8	2
San Francisco Bay Area	321,270	800,172	1	7	2
New York City	264,346	733,846	1	8	2

We implement the serial, OpenMP, thread, and Pthread version for Dijkstra's Algorithm, and we test each method with the datasets mentioned above. For evaluating program performance, we set programs with 2, 4, 8, 12 threads and compare their execution time by getting each version of Dijkstra's Algorithm 5 times' average.

Our testing environment is on x86_64 GNU/Linux, which CPU is AMD Ryzen 5 3600 6-Core Processor. There are 2 threads per core in the CPU, which means the maximum number of threads that can support by hardware is 12 threads. The memory is 16 GB, enough to load the entire graph into the memory.

5 EXPERIMENTAL RESULTS

5.1 OpenMP

We first measure normal, padding, and no reduction version of openMP, no reduction version is the method we proposed above but it uses atomic operations to find global maximum vertex rather than binary tree reduction padding version is the normal version but shared memory is padded to reduce false sharing. Result is show at Table 3.

It is no surprise that the padding version is faster than the normal version. No reduction version is normal version without binary tree reduction, instead, it uses atomic operation to find global maximum, it is slightly faster than the normal version, it is understandable since reduction time complexity is $O(logn)$, it will be fast when n is large but in our case, the number of thread is only less than or equal 12.

We can see that the speedup glow up slowly when the number of threads is beyond 4 threads. We can find some reason why the speedup is slow after 4 threads from Table 2. First, context switch increase rapidly, we thought that our platform has 12 thread, though, it only has 6 cores, which means if the number of thread is larger than 6, it needs to do the context switch to let every thread be done, context switch needs CPU scheduling times, load and store the register from PCB, and jump to the PC, it is a large overhead, second, increasing of context switch let to increasing of CPU migration,

Table 2: OpenMP Analysis

	1 Thread	2 Threads	4 Threads	8 Threads	12 Threads
Time	72.6	44.8	26.7	24.5	21
Page Faults	5275	5275	5280	5288	5297
CPU Migrations	7	7	32	115	438
Context Switches	5834	6148	7607	14,754	30,319
Cache Misses	155,321,333	96,508,940	105,331,839	190,194,510	242,000,827
Cache References	4,946,275,072	4,370,122,372	3,288,139,266	3,172,426,637	3,336,326,862
Instructions	841,363,340,905	841,363,340,905	843,773,918,880	848,384,339,189	865,240,316,477

All of measurements are only calculate on normal OpenMP version.

Table 3: OpenMP Execution Time (seconds)

# of Threads	Normal	Padding	No Reduction
1	72.6	74.5	71
2	44.8	35.4	42.3
4	26.7	20.7	23.1
8	24.5	18.9	21.7
12	21	19	19.8

third, cache miss increases rapidly, if cache miss happened, CPU will need to get the data from memory, which is 10 times slower than cache normally, we do an experiment between padding and non-padding version.

We also test 12 threads cache misses, where the normal version cache misses 235,032,974 times and padding version cache misses 228,242,059 times. So, we can see that no padding version cache miss is higher than padding version, we can said that the cache miss increaseing is cause by false sharing.

5.2 C++ Thread and POSIX Thread

All known OpenMP implementations use a thread pool so the full cost of threads creation and destruction is not incurred for each parallel region. To compare with the OpenMP version, we implement the C++ thread and the POSIX thread version. The C++ thread version creates threads and joins them after finish works. And the POSIX thread version creates threads and detaches them instead of joins them, and if threads need to wait for each other, we use a barrier to synchronizes them.

As mentioned above, we can analyze the efficiency of join and barrier, and the result that is shown at Table 5 as several points can discuss. First, we can find that the C++ thread version with lock is faster than the POSIX thread version in 2 threads, but when more threads we use, the C++ thread version becomes slower. Because of this result, we think the overhead of joining is larger than the barrier when the amount of threads is larger. Second, we can easily find that all three methods' fast time is at 4 threads, and in Table 4, we can see the more thread we use, the more waiting time consume. Because of this result, we think the overhead of both joining and barrier affects a lot, especially we have a huge loop number. Third, we can find that the reduction version is slower than the lock version, and we think the reason is the same as the reason already mentions when analyzing the OpenMP version.

Table 4: POSIX Thread Barrier Time (seconds)

Thread Id	2 Threads	4 Threads	8 Threads	12 Threads
0	12.6801	11.4507	6.5479	**10.6137**
1	**3.1549**	12.6662	23.6264	43.4708
2	-	8.1953	19.5297	34.7185
3	-	**6.1430**	17.0812	31.4165
4	-	-	15.2573	29.6719
5	-	-	13.7105	27.8492
6	-	-	9.9289	25.5766
7	-	-	**6.1337**	22.8032
8	-	-	-	20.4435
9	-	-	-	17.8874
10	-	-	-	14.5568
11	-	-	-	10.8061

6 RELATED WORK

Since Dijkstra's algorithm is a complex graph algorithm that has low computation to memory access ratio and requires a lot of synchronizations, the following concerns need to be answered for solution on parallel hardware:

- Optimize memory accesses
- Minimize synchronizations
- Exploit parallelism in the algorithm

Following are some of the related works for parallelization of Dijkstra's algorithm:

- Graph partitioning into neighboring sub-graphs: kd-tree is utilized in [1] in order to generate arc labels that mark, for each arc, possible target nodes of a shortest paths that start with this arc. Dijkstra's algorithm can then be restricted to arcs whose label mark the target node of the current search, because a sub-path of a shortest path is also a shortest path. Partitioning algorithms from computational geometry are utilized in combination with arc labels. But, the graph partitioning is a NP-Hard problem and hence may be impractical for large graphs.
- Delta-stepping algorithm: [2] divides original Dijkstra's algorithm into a number of phases that can execute in parallel. This algorithm is an approximate bucket implementation of Dijkstra's original algorithm. For random graphs with uniformly distributed edge weights, this algorithm runs in sub-linear time with linear average case work. [3] shows 30x

Table 5: C++ Thread and POSIX Thread Execution Time (seconds)

Method	2 Threads	4 Threads	8 Threads	12 Threads
C++ Thread (reduction)	82.4959	84.3278	139.707	178.736
C++ Thread (lock)	74.3632	60.4873	119.444	130.994
POSIX Thread (lock)	48.8485	39.2458	51.7744	70.5366

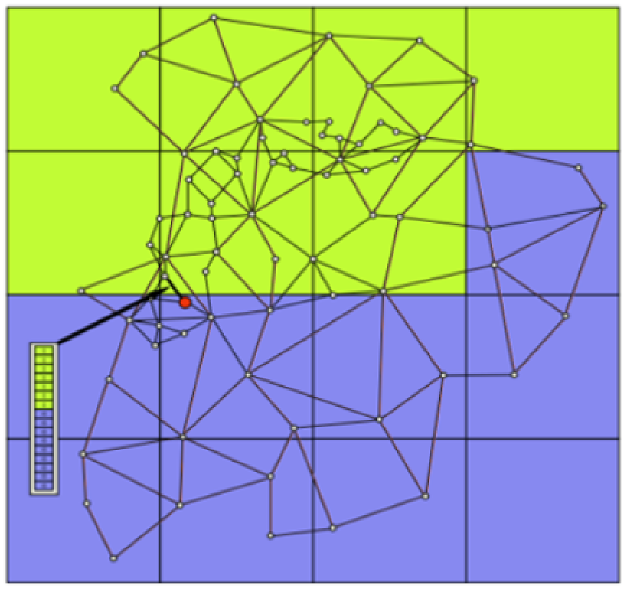

Figure 3: Arc-Flag

improvements over a high-end Shared memory platform, also displaying good vertical scalability.

- GPU-based implementation: It has been observed in [4] that GPU-based solution is worse performing than the sequential implementation for graphs with more than a million vertices. Moreover, the device memory is not enough to handle large graphs.
- Intel Many Core Integrated (MIC) architecture: [5] implements a asynchronous version on the shared memory architecture, with aim to have cache optimizations. The work achieved the highest performance yet for Dijkstra's algorithm on Intel Xeon Phi processor by minimizing irregular memory access patterns.

6.1 Arc-Flag Approach

Here we separately mention about Arc-Flag (show at Figure 3), this approach is based on a partition of the graph into node sets $R_1, R_2, ..., R_k$ which we call regions. For each *arc A* in the graph, it store a flag for each region $R_i (0 < i \leq k)$). This flag is set to TRUE if A is on a shortest path to at least one node in R_I or if A lies in R_i.

A shortest path search from a node s to a node t in region R_j can now be conducted using a Dijkstra's Algorithm that only traverses *arc A* where $fA(j)$ is TRUE. Note that all shortest paths entering a region R_i have to use some *arc A* that crosses the border of R_i. Now for each of those crossing arcs a shortest path tree in the reverse graph is computed starting at *arc A*. All arcs in this a–rooted reverse shortest path tree obtain the value TRUE in their flag-vector at position i. Doing this for all arcs entering R_i one can fill up all entries at the i-th component of the flag-vector of all arcs in G.

7 CONCLUSIONS

In our experiments, we find that OpenMP implementations probably have optimization on the barrier. And in our research, we have two main contributions.

- We prove that join and barrier play a role when using many times.
- Our parallel strategy is useful.
  Most of the time is wasted on search the shortest distance from the source.

In the research, we only analyzed execution time on the CPU, so we can next try to apply our parallel strategy to GPU and test efficiency. The C++ library, boost, has API for Dijkstra's Algorithm, so we test the boost version's execution time and it only calculates about 0.05 seconds for New York City graph. Because of the boost version, Dijkstra's Algorithm may have a better parallel algorithm to do so.

REFERENCES

[1] Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 189–202. Springer, Heidelberg (2005). https://doi.org/10.1007/11427186_18

[2] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm.J. Algs.,49(1):114–152, 2003.

[3] Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R.: Parallel shortest path algorithms for solving large-scale instances. In: Dimacs Implementation Challenge - The Shortest Path Problem, vol. 74, pp. 249–290 (2011)

[4] Harish, P., Narayanan, P.J.: Accelerating Large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77220-0_21

[5] Zhang, W., Zhang, L. and Chen, Y., 2018, November. Asynchronous parallel Dijkstra's algorithm on intel xeon phi processor. In International Conference on Algorithms and Architectures for Parallel Processing (pp. 337-357). Springer, Cham.