Parallelized Single Source Shortest Path Algorithms

Zhenghua Chen, [redacted] Xiaoou Liu, [redacted] Arun Majumdar, [redacted]

1 Abstract

In this project, we implement and parallelize two algorithms, Bellman–Ford algorithm and Δ-stepping algorithm for Single-Source Shortest Path Problem (SSSP) and compare their performances with a non-parallelizable Dijkstra's algorithm when traversing through different directed graphs.

2 Introduction

SSSP is to find the path with minimum cumulative weights from the source vertex $s \in V$ to all other vertices in a given directed weighted graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The most famous algorithm is Dijkstra's algorithm. However, Dijkstra has a data dependency due to it needs to update the distance array serially. For large graphs in real problems, we need faster parallel algorithms.

3 Problem Statement

We selected Bellman-Ford and Δ-stepping algorithms to parallelize for this project. We implemented both OpenMP and CUDA versions of the two algorithms and compared their execution time with serial execution time respectively.

4 Bellman-Ford Algorithm

4.1 Serial pseudocode:

Given a graph G with V vertices and edges E:

```
1. Initialization:
        Create a distance array of size V to store the shortest distances. Set distance of all vertices
        to infinity. Set distance of source vertex to 0.
2. Repeat for i in (1, V-1):
        For all edges (u, v) in G:
            If distance(u) + weight (u, v) < distance(v):
```

distance (v) = distance (u) + weight (u, v)

The loop in step 2 reads all the edges in any order and guarantees that the shortest path with a maximum length of i from src to each node is found. Since each iteration i of the outer loop depends on the shortest path of length i-1 found in the previous iteration, the outer loop has loop-dependency and cannot be parallelized. However, the inner loop reads every edge in any order and can be parallelized. One important detail is that there's no critical section in the parallel inner loop even if the distance array is read and wrote at the same time by different threads since each edge would be read V-1 times in the outer loop.

4.2 Parallel pseudocode:

In OpenMP:

```
Step 2 in parallel: Repeat for i in (1, V-1):
        #pragma omp parallel for num_threads(8)
        For all edges (u, v) in G:
            If distance(u) + weight(u, v) < distance(v):
                distance (v) = distance (u) + weight(u, v)
```

In CUDA:

```
Step 2 in parallel: Repeat for i in (1, V-1):
        Run a kernel function:
            relax<<<1+(E-1)/1024, 1024>>>(d_edges, E, d_dist)
        which does:
            a.  find thread tid using blockDim.x * blockIdx.x +threadIdx.x;
            b.  if id is smaller than E:
                access one edge and its attributes u, v and weight using edges[tid]
                If distance(u) + weight(u, v) < distance(v):
                    distance (v) = distance (u) + weight(u, v)
```

4.3 Execution time in milliseconds for Bellman-Ford parallelization:

	Graph density	E=10	E=100	E=1k	E=10k	E=100k	E=1million
Serial	0.2	0.001	0.009	0.261	7.081	221.383	6715.286
	0.8	0.001	0.005	0.111	3.340	105.697	3369.093
OpenMP	0.2	0.469	1.356	3.642	8.989	87.317	1784.499
	0.8	0.472	1.213	1.556	5.425	47.1	908.041

	Graph density	E=1k	E=10k	E=100k	E=1million	E=2million
Serial	0.2	0	12	419	14151	37497
	0.8	0	6	215	6687	19324
CUDA	0.2	718	698	718	897	964
	0.8	710	711	704	786	910

When E gets larger than 10k/100k, OpenMP/CUDA parallel code would start to accelerate. When graph is denser, the execution time for both serial and parallel implementation is smaller since V is smaller (the outer loop iteration time is smaller and parallelization part is bigger) given the same E. For the same E, OpenMP serial code takes less time than CUDA code while OpenMP parallelization takes more time than CUDA parallelization.

5 Δ-stepping Algorithm

Δ-stepping is an algorithm that combines the advantages of Dijkstra and Bellman-Ford. It can balance the parallelism and work efficiency. We divide all edges into light and heavy edges based on the value of Δ. When the weight is less than or equal to Δ, it belongs to the light edge and vice versa. In Bellman-Ford algorithm, we need to relax all the edges in each loop. However, some edges that have longer distance don't need to relax every time, because they will not change the results in the first few loops. Thus, the light edges have a high relax frequency than the heavy ones. Compared to the Bellman-Ford algorithm, Δ-stepping algorithm can reduce the number of edges relax.

Bucket is a priority queue data structure type. It is grouped by vertices temporary distance labels. Which bucket we put the vertex in depends on the distance of this vertex and Δ, like a hash function. A vertex in the bucket means its shortest path has not yet been found and it still needs to be relaxed. This is similar to the Dijkstra algorithm. When Δ goes to 0, this algorithm is equivalent to Dijkstra algorithm. When Δ goes to infinity, this algorithm is equivalent to Bellman-Ford algorithm.

5.1 Pseudo code for the serial solution algorithm

```
Algorithm 1: Serial $\Delta - stepping\ algorithm$
Foreach v $\in V, dis[] := \infty$
Bucket[0][0]:=0; dis[0]:=0;
j:=0;
while $\neg isEmpty(B)$ do
    $Rl[]:=\emptyset; Rh[] := \emptyset$;
    while $\neg isEmpty(B[j])$ do
        Rl:=find(B[j],light);
        Rh:=find(B[j],heavy);
        $B[j]:=\emptyset$;
        relaxRequests(Rl);
    relaxRequests(Rh);
    j:=j+1;
```

```
Algorithm 2: relaxRequests(w, d)
if $d < dis[w]$ then
    dis[w] <- d;
    B[dis[w] $\div\Delta$]$< - B[dis[w] \div \Delta] \setminus$ {w};
    B[d $\div\Delta$]$< - B[d \div \Delta] \cup \{w\}$;
```

Algorithm 1 is the pseudo-code of the serial Δ-stepping algorithm. The dis array is the temporary shortest distance from the source vertex to all other vertices. The bucket queue contains all the vertices that still need to be relax. The Rl array is the set of all light edges. The Rh array is the collection of all the heavy edges. This algorithm firstly initializes the dis array to infinity and add source vertex 0 to the dis array and bucket. Secondly, it divides all the edges of all the vertices in the first bucket into light list and heavy list, then empty this bucket. Next, relax all the light edges. Repeat the inner loop until the first bucket is empty. Then relax heavy edges just one time. Thirdly, it repeats the outer loop of the next bucket until all the buckets are empty. The dis array is the final shortest path result. Algorithm 2 is the pseudo-code of the relaxRequests function. If the new distance is less than the old distance, the dis array is changed. Then the old distance vertices in the bucket are deleted and the new distance vertices in the bucket are added.

There are three parts that can be parallelized. The first is to build request lists. Each thread can handle one vertex in the bucket. The second is to relax light edges list. Each thread can relax one edge it the list. The third is to relax heavy edges list.

5.2 OpenMP Implementation

Algorithm 3: OpenMP Implementation

```
1  Foreach v $\in V, dis[] := \infty$
2  Bucket[0][0]:=0; dis[0]:=0;
3  j:=0;
4  while $\neg isEmpty(B)$ do
5      $Rl[]:=\emptyset; Rh[] := \emptyset$;
6      while $\neg isEmpty(B[j])$ do
7          #pragma omp for private (k)
8          Rl:=find(B[j],light);
9          Rh:=find(B[j],heavy);
10         $B[j]:=\emptyset$;
11         #pragma omp for
12         relaxRequests(Rl);
13     #pragma omp for
14     relaxRequests(Rh);
15     j:=j+1;
```

Algorithm 3 is the pseudo-code of OpenMP implementation. We use three parallel for in lines 7,11 and 13. In the relaxRequests function, we need to read and write to the same variable Bucket, which has read and write conflicts. There are two solutions to solve this problem. One is to use critical directive. The other is to use vector structure in C++. Vector can protect the same variable from being modified at the same time to ensure accuracy.

5.3 CUDA Implementation

Algorithm 4: CUDA Implementation

```
1  kernel launch configuration, allocating and copying device memory
2  Foreach v $\in V, dis[] := \infty$
3  Bucket[0][0]:=0; dis[0]:=0;
4  j:=0;
5  while $\neg isEmpty(B)$ do
6      $Rl[]:=\emptyset; Rh[] := \emptyset$;
7      while $\neg isEmpty(B[j])$ do
8          Launch request kernel function
9          $B[j]:=\emptyset$, then copy CPU dataB[][] to the GPU memory
10         Launch relax kernel function in Rl
11         Copy GPU data d_S_Rl[] back to the CPU S_Rl[]
12         Serial relax function in S_Rl[], then copy CPU B[][] and dis[] to the GPU d_B[][]
             and d_dis[]
13     Launch relax kernel function in Rh
14     Copy GPU data d_S_Rh[] back to the CPU S_Rh[]
15     Serial relax function in S_Rh[], then copy CPU B[][] and dis[] to the GPU d_B[][] and
         d_dis[]
16     j:=j+1;
```

Algorithm 4 is the pseudo-code of CUDA implementation. CUDA is implemented in the same way as OpenMP. However, there are four caveats. The first is it requires to use cudaMemcpy2D to copy the two-dimensional array Bucket to CUDA, because in CUDA the two-dimensional array is scattered rather than continuous area like in CPU. Secondly, we launch three kernel functions in line 8, 10 and 13. Some variables, such as dis[] and B[][], need to be updated in the CPU, while others, such as d_S_Rl[] and d_S_Rh, need to be updated in CUDA. Thus, we need to pay more attention to data synchronization. Thirdly, this implementation also has data race like in OpenMP. However, CUDA does not support the vector structure. The only thing we can do in CUDA is to find out the edges that need to relax, then we do the relax function in CPU serially. Compared to Bellman-Ford algorithm, we can't parallel every relax function, but it's still much more efficient. Finally, as Δ increases, the running time will increase first and then decrease. There is different best of Δ for different graphs. Currently, some papers show how to automatically find the best of Δ in different graphs and propose the ρ-stepping algorithm. In my implementation, I use a for loop to find the Δ with the shortest running time.

5.4 Δ-Stepping Implementation Results

Table 1: The running time of Δ-stepping algorithm in milliseconds.

E	10	100	1,000	10,000	100,000	1,000,000
*Density = 0.2						
Serial	0	1	4	46	454	5164
OpenMP	0	0	2	16	64	704
CUDA	15	36	103	345	1030	\
*Density						
Serial	0	0	4	44	436	5034
OpenMP	0	0	2	11	58	635
CUDA	14	21	112	258	961	\

We can obviously see that the speedup of OpenMP implementation is about 7 on the million-edge graph. It has a nice parallel efficiency. The reason why CUDA on the million-edge graph has no result is that it exceeds the Bucket maximum range. The global memory in CUDA is limited, so the 2D Bucket can't be so huge. Moreover, CUDA runs longer than serial because it needs to copy data between CPU and GPU which is a waste of time. In the future, we can think about how to optimize CUDA implementation.

6 Conclusion:

1. As the size of graph increases, the speedup increases. Parallelism is better for the large graphs. (Gustafson's Law)

2. Δ-stepping takes less time than Bellman-Ford when the number of edges is equal to one hundred thousand in OpenMP. The reason why CUDA implementation of Δ-stepping takes more time is because of updating global memory frequently and data synchronization between the CPU and GPU. It's much overhead. CUDA implementation of Δ-stepping algorithm needs to be optimized. Therefore, the OpenMP implementation of Δ-stepping algorithm is the best implementation in our experiments.