A Research on Parallelized Max Flow problem

Haining Xie

Introduction

This research report aims to illustrate an efficient parallel algorithm to find the minimum of maximum flows between all pairs of vertices within a weighted directed graph. It is easy to divide this problem into two parts. The first part is to compute max flow of different vertex pairs in parallel. The second part is to compute the max flow between two vertices in the graph and parallelize this process.
It is obvious to see that there is no dependency between calculating max flow between different pairs. These sub-problems just share the same graph structure. To parallelize this, just allocate multiple threads to calculate a disjoint part of the sub-problem set.
To solve the second part, we need to analyze the existing sequential algorithms of max flow and find their suitable part to parallelize. There are mainly two classed of algorithms. One is using the augmenting path, usually called Ford-Fulkerson, which uses the idea of interactive improvement . This approach needs to search an augmenting path from the whole residual network in each iteration. The other is the push-relabel algorithm, which maintains a preflow during the iterative improvement. Different from the former algorithm, it works on the vertex and search its neighbors in the residual network. This localized manner makes it a good candidate for parallelization.
This research report uses the idea of the push-relabel algorithm. Mainly, the push-relabel algorithm defines three operations, the preflow process, the flow pushing and the vertex relabeling. The preflow process is the initialization process which pushes flows from the source. Generally, it is a group of assignment operations, which has no dependency with each other. So it is easy to parallelize. The flow pushing is performed on current active vertices. There is no restricted order of pushing. But this only guarantees the algorithm could run in polynomial time complexity. According to[1], the FIFO processing order could help to achieve the time complexity of O(n^3). No specific order means the active vertices could be processed parallelized. Further, to utilize the FIFO processing improvement, this research report used the approach proposed by[2], which performs the push operation concurrently on all current active vertices and maintain the newly discovered active vertices in a queue structure. There are two major design of this approach. The first is using the copy of the excess flow value and label value during the iteration and update these two values only after the pushing iterations of current active vertices. The first usage of this design is to let the algorithm make progress during iterations, which avoids flow bouncing between vertices permanently. The second usage is to reduce the synchronization process between threads. Threads could work on their own copies and do not interfere with each other. The second design is the conflict resolving policy. This resolves the case that two vertices are activated at the same time and try to push flows to each other. With the resolving policy, the result of parallelized pushing operations can be mapped to a result after sequential operations. The relabel operations are separated in different parts. In the pushing phase, it actively records the possible new label. Then the final relabel is done after the pushing iterations. However, the relabeling methods introduced by [2]has some mistakes. It uses the comparison between the actual label and copy of the label as the condition to decide if the new label of the current vertex should be updated. This could work in most cases. However, this is not sufficient to guarantee the correctness. Instead, the relabel condition should rely on the current state of the residual network, which is similar to the relabel condition of relabel-to-front algorithm[3]. This research report corrects this part in the pseudocode. In addition, this research also makes use of the improved version of global relabelling introduced by prsn[2], which uses a parallelized BFS to correct the label of vertices.

The pseudocode:

```
procedure PushRelabel()
   for v in V do in parallel:
      h(v) = 0
                  e(v) = 0
                  v.addExcess =0
         h(s) = n
         for (v,w) in E do in parallel
                  f(v,w) = f(w,v) =0
         for (s,v) in E
                  f(s,v) = c(s,v)
                  s(v,s) = c(s,v)
                  e(v) = c(s,v)
         workGR = Max
         while true
                  if freq*workGR > a*n+m
                           workGR =0
                           Globalrelabel()
                           activeSet = [v | v in activeSet, h(v)<n]
                  if activeSet.isEmpty break
                  for v in activeSet
                           v.discovered = []
                           h_cp(v) = h(v)
                           le = e(v)
                           v.work =0
                           while le >0
                                    newL = n
                                    skipped = false
                                    for (v,w) in residual net do in parallel
                                             if le ==0 break
                                             admissible = (h_cp(v) == h(w)+1)
                                             if e(w) >0
                                                      win = (h(v)==h(w)+1 || h(v)<h(w)-1 ||
(h(v) == h(w) && v<w))
                                                      if admissible and !win
                                                               skipped = true
                                                               continue
                                             if admissible && cf(v,w)>0
                                                      dlt = min(cf(v.w),e(v))
                                                      f(v,w) += dlt
                                                      f(w,v) +=dlt
                                                      e -= dlt
                                                      w.addExcess += dlt
                                             if w != sk
                                                      v.discovered.add(w)
                                    if cf(v,w) >0 && (w,v) in residual net
                                             newL = min(newL,h(w)+1)
                           if le == 0 || skipped break
                           h_cp(v) = newL
                           if h_cp(v) == n
                                    break
                  v.addExcess = le-e(v)
                  if le
                           v.discovered.add(v)
```

```
for v in activeSet
        h(v) = h_cp(v)
        e(v) += v.addExcess
        v.addExcess =0
        activeSet = Concatnate([v.discovered | v in activeSet,h(v)<n])
for v in activeSet
        e(v) += v.addExcess
        v.addExcess =0
```

Methodology

The experiments are mainly divided into two parts. The first part is measuring the speed up of this parallelization algorithm. The speed up could be defined using the following formula. However, this measurement method has its own scope, which only suits the problem with huge size, where the context switching time and threads synchronization could be ignored. In small scope, this report utilizes the instruction speed to illustrate the performance, which could be defined with the following formula.
speed up factor = Baseline executionCurrent execution time
In small scope problems, threads are usually idle waiting for each other. This factor could show how idle threads are.
The other part of the experiment is based on tricks used in the real implementation, which takes advantage of the hardware features and openmp features. The first is the cache miss count. The reason for not using cache miss rate is that the whole experiment is performed on the whole algorithm, which is limited by the profiling tool. In this scope, the total cache miss rate may be similar to each other. However the cache miss count has huge differences. Then the next experiment is to measure the influence of the thread's position. In this report, it measures the execution time of using different binding policies.

Experiment

The first experiment is about the speed up of computing max flow of different pairs. The experiment of figure 1 is based on a graph with 100 nodes, densely connected. Here 100 node means the initial size of the problem. In this graph, there are anti-parallel edges. After converting to normal form, the average number of nodes is 2000. The figure 2 is based on a 300 node problem, which may have around 8000 nodes after normalization to the standard form.

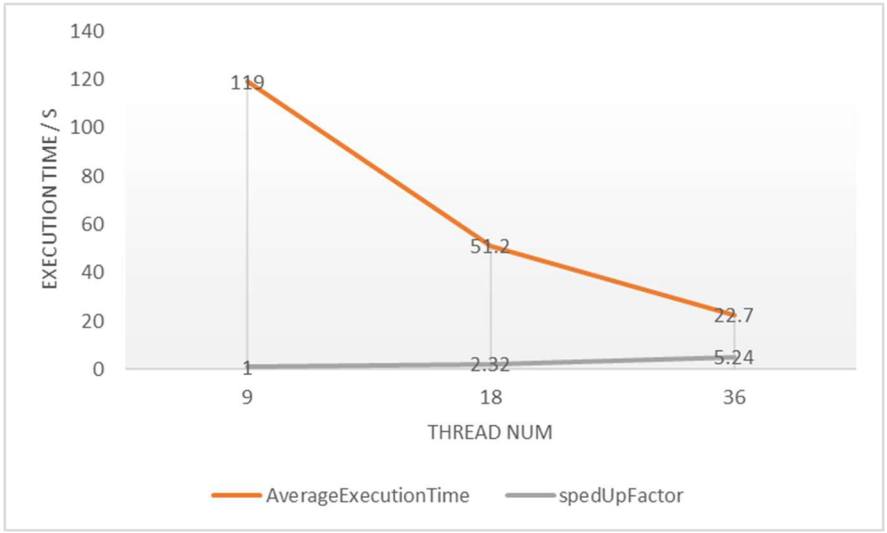

Figure 1

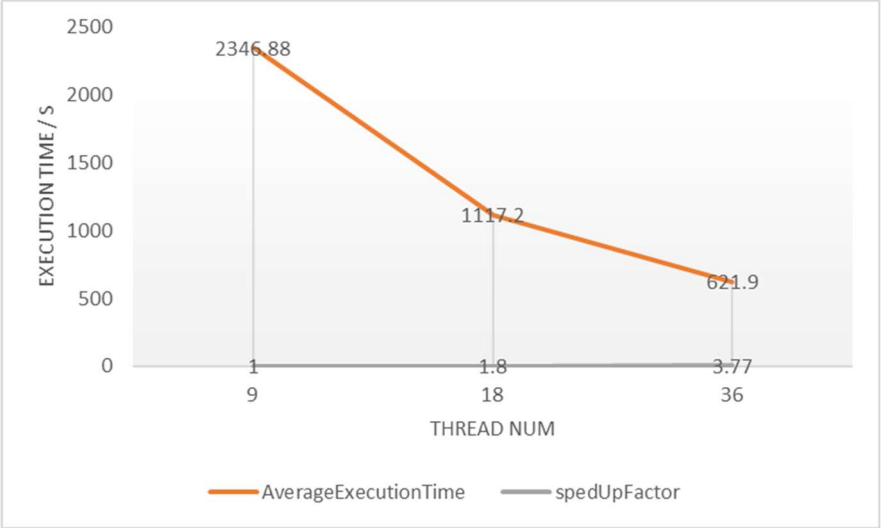

Figure 2

The figure3 is the execution time of parallelized max flow problem using different threads. The testing graphs has 100000 nodes and nearly 500000 edges each. The time unit is second.

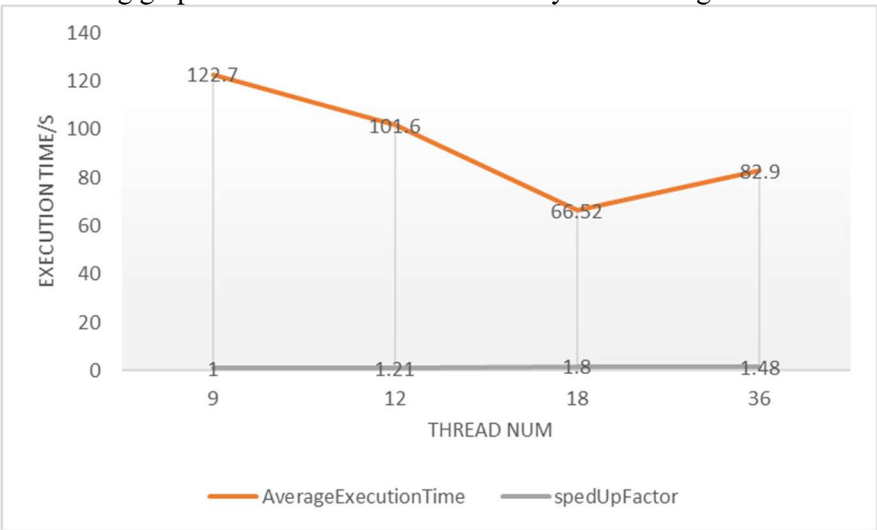

Figure 3

Table 1 is the measurement on a problem with 50 nodes. Outer thread means how many threads are used for the first part of parallelization, the computation of max flow of different pairs of nodes. The inner thread means how many threads are used for the parallelization of max flow algorithm.

Table 1

Thread distribution	Average Instruction speed
4 outer master thread, 2 inner slave thread	0.57
2 outer master thread,1 inner slave thread	2.68
Single thread	2.66

The following figure 4 is a sample of measurement of instruction speed, where threads are idle in most of the time.

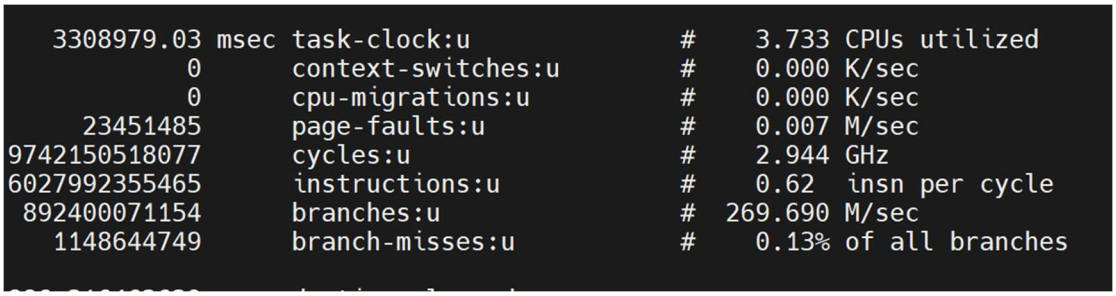

Figure 4

The following table 2 is the comparison of L1 cache load misses. The first row uses an extra structure to store elements, which avoids skip indexes in the matrix. The other one just operates directly on the matrix.

Table 2

Matrix operation	Average L1 cache load miss
Use optimized matrix operation	23906147
Direct use	3882807768

The table 3 shows the average delay between using or not using the cache optimized structure.

Table 3

	Execution time
Direct use	8.6s
Use cache optimized structure	1.2s

The following table 4 illustrates the execution on different size of problems when applying different policies.

Table 4

	10000 Node problem	100000 node problem
Close policy (18 threads)	0.56	65.13 s
spread policy (18 threads)	0.61	67.81 s

Discussion and Conclusion

The first experiment of measuring the speed up has been divided into two parts. The first part is measuring the speed up of parallelized computation of different pairs. The second part is measuring the speed up of the parallelized max flow algorithm. There are several reasons to measure these two parts separately. The first reason is that measuring these two parts together may make the contribution of performance improvement unclear. The second part is the physical limit from the experiment environment. Because the total computing complexity

could be expressed as O(n^3*O(f)). The f refers to the time complexity of the max flow algorithm and n refers to the number of vertices. In this report, the time complexity of the max flow algorithm is O(n^3). The parallelized max flow algorithm needs a graph with a large number of vertices, which usually means more than 10000 nodes, to show its performance speed up. Let's say it take 0.1 second to solve this single max flow problem. To calculate all pairs, it could take tens of hours. In this case, experimenting with the gained performance in a larger graph is almost possible. The other physical limitation is the memory. Even though each thread could share the same graph structure in the memory, they still need memory to store flow network and residual network, which is the same size as the original graph. This memory requirement also beyonds the memory limit.

After illustrating the above issue, we could focus on the performance improvement. To the parallelized computation of different pairs, it is easy to see the speed up increase linearly with the thread number with a suitable graph size. However, in a small graph size, the total performance degrades. Multiple threads may need hundreds of execution time than a single thread. To address this problem, this report measures the instruction operation speed, which will be covered in later experiment section.

Then let's focus on the performance of parallelized max flow algorithm. According to figure 1, with the fixed size of the graph, the speed up factor increases nearly linearly in a certain range. In figure 3, which has the largest graph in this experiment, this relationship is prominent. It could infer that with a large graph, the scalability of this parallelized algorithm is stable and predictable. This feature could also be inferred from the algorithm itself. Different from other parallelized max flow algorithms, this algorithm is a synchronized algorithm. Threads are synchronized after each parallel section. It is simple and the synchronization complexity will just increase linearly with the thread count. However, we should notice that, in figure 3, the speed up factor first increases linearly and then begins to stagnate, or even goes down. This is because the work shared by each thread becomes smaller when the number of threads increases. Each thread finishes its own work and waits for each other. The following part will illustrate the reason.

With table 1, we can see the single thread instruction execution speed is much higher than than multiple threads. This is measured by the perf tool provided by the linux system. In theory, it is easy to illustrate. Threads are created at different time and has its own unique context. According to the implementation, some of them may be pulled from the idle thread pool while others may be create on demand. They start at different time and may share different amounts of work. Some of them may finish early and wait for others. The whole waiting time depends on the slowest thread, due to the fork join model. This delays the whole process. So the instruction execution speed indicated by the perf tool is lower than the single thread case. However, this could only illustrate why the instruction speed is lower, it doesn't address why the total execution time is longer. One possible reason is that the OS may hang up the idle thread and then wake it up when the slowest thread catched up. This hang up and wake up behavior may cause extra time wasted.

The next experiment is about the cache problem. This report uses an extra space to store neighbors of vertices during the breadth first search process of the global relabel phase. The tuition of this trick is that the hardware may try to prefetch some of the neighboring data to its local cache. The speed of this cache is much faster than the memory accessing speed. In some sequential operations on the matrix, this feature will increase the total performance. However, in the reverse case, unpredicted indexing will cause the cache miss. The classical example of this problem is the indexing order on the matrix. The operations in the global relabel phase face this scene. To illustrate this could help, this report tests different strategies on the same problems. One measure is based on the execution time. According to table 2, the algorithm that does not utilize this feature costs a significant amount of time more than the one that uses. Furthermore, it could be seen that this phenomenon is more and more obvious when the graph gets larger. To check if it is the problem of the cache miss. The experiment uses the perf tool to monitor the L1 cache miss number of two different cases. There are two main reasons that the cache missing rate is not used. The first one is that the overall cache miss rate does not differ too much in cases that we could measure. Because in these cases, the matrix

operation part is just a small part of the problem. The second one is that using the perf tool to monitor the cache miss may cause the whole process to operate in a much longer period of time. So only small graphs could be used. With small graphs, in most cases, the cache miss rate does not differ too much. However, the cache miss number is obviously large in an unoptimized algorithm. In large graph case, this problem is much more serious. For example, the larges graph that we used has 100000 nodes. If we represent it using a matrix, a single row may cost more than one hundred megabytes. This exceed the capacity of common L1 or L2 cache.

The last experiment is on the thread allocation position. In this experiment, we used two strategies. The first one is close binding while the other is spread binding. However, according to the test, seen from table 4, these two modifications do not influence the execution time too much. The test is performed on two sockets. More sockets are limited by the account quota. The reason may be this algorithm's locality feature. Though threads on different sockets have different memory accessing speed, the core's own cache may make this influence non-obvious.

With the above discussion and experiment, it can be seen that this parallelized algorithm has a promising speed up to the sequential version and it is suitable to scale up. It also shows that optimized matrix operation may also improve the performance.

Reference

[1] Anderson, Richard, and Joao C. Setubal. "A parallel implementation of the push-relabel algorithm for the maximum flow problem." *Journal of parallel and distributed computing* 29.1 (1995): 17-26.

[2] Baumstark, Niklas, Guy Blelloch, and Julian Shun. "Efficient implementation of a synchronous parallel push-relabel algorithm." *Algorithms-ESA 2015*. Springer, Berlin, Heidelberg, 2015. 106-117.

[3] Shaffer, Clifford A. *A practical introduction to data structures and algorithm analysis*. Upper Saddle River, NJ: Prentice Hall, 1997.