A Research on Parallelized Max Flow problem

HAINING XIE, University of Melbourne, Australia

Additional Key Words and Phrases: max flow, parallel computing

ACM Reference Format:
Haining Xie. 2022. A Research on Parallelized Max Flow problem. 1, 1 (September 2022), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 the 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 classes of algorithms. One is using the augmenting path, usually called Ford-Fulkerson, which uses the idea of interactive improvement. This approach needs to search for 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 searches 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 that pushes flows from the source. Generally, it is a group of assignment operations, which has no dependency on 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 maintains the newly discovered active vertices in a queue structure. There are two major designs of this approach. The first is using the copy of the excess flow value and label value during the iteration and updating 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

Author's address: Haining Xie, HAIXIE@student.unimelb.edu.au, University of Melbourne, Melbourne, Australia.

Manuscript submitted to ACM

the synchronization process between threads. Threads could work on their own copies and 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 into 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] have some mistakes. It uses the comparison between the actual label and a 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 correctness. Instead, the relabel condition should rely on the current state of the residual network, which is similar to the relabel condition of the 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.

2 PSEUDOCODE

2.1 The Max Flow Algorithm

The following is the main algorithm of the parallelized push relabel algorithm, which takes the idea of [2], which runs a global relabel algorithm after a certain amount of work has been done.

Algorithm 1 Parallelized Max Flow Algorithm

Require: n **processor**

for $v \in V$ **do in parallel**
  $h(v) = 0$
  $e(v) = 0$
  $v.acumExcess = 0$
end for
$h(s) = n$
for $(v, w) \in E$ **do in parallel**
  $f(v, w) = 0$
  $f(w, v) = 0$
end for
$PreFlow()$
$lastWork = 0$
while True do
  if $freq * lastwork > \alpha * n + m$ **then**
    $lastwork = 0$
    $GlobaleRelabel()$
    $newSet = \{\}$
    for $v \in activeSet$ **do in parallel**
      if $h(v) < n$ **then**
        $newSet.add(v)$
      end if
    end for
    $activeSet.swap(newSet)$
  end if
  if $activeSet == \{\}$ **then**
    break
  end if
  for $v \in activeSet$ **do in parallel**
    $v.discovered = []$
    $hcp(v) = h(v)$ ▷ hcp is the local copy of the height(label) of vertex
    $le = e(v)$ ▷ le is the local copy of the excess value of vertex
    $v.work = 0$

```
33:         while le > 0 do
34:             newLabel = n
35:             skipped = 0
36:             for (v, w) ∈ E_f do in parallel
37:                 if le == 0 then
38:                     break
39:                 end if
40:                 admissible = false
41:                 if hcp(v) == h(w) + 1 then
42:                     admissible = true
43:                 end if
44:                 win = false
45:                 if e(w) > 0 then
46:                     win = (h(v) == h(w) + 1||h(v) < h(w) − 1||(h(v) == h(w)&&v < w))
47:                     if admissible&&!win then
48:                         skipped = true
49:                         continue
50:                     end if
51:                 end if
52:                 if admissible&&c_f(v, w) > 0 then
53:                     dlt = min(c_f(v, w), e(v))
54:                     f(v, w) = f(v, w) + dlt
55:                     f(w, v) = f(v, w) − dlt
56:                     le = le − dlt
57:                     w.acumExcess = w.acumExcess + dlt          ▷ This should be perdormed atomicly
58:                     if w! = sink then
59:                         v.discovered.add(w)
60:                     end if
61:                 end if
62:                 if c_f(v, w) > 0&&E_f(v, w) > 0 then
63:                     newLabel = min(newLabel, h(w) + 1)    ▷ This is the modification part mentioned in the report
64:                 end if
65:             end for
66:             if le == 0||skipped then
67:                 break
68:             end if
69:             hcp(v) = newLabel
70:             v.work = v.work + v.outDegree + β                          ▷ Here β is 12
71:             if hcp(v) == n then
72:                 break
73:             end if
74:         end while
75:         v.acumExcess = le − e(v)
76:         if le > 0 then
77:             v.discovered.add(v)
78:         end if
79:     end for
```

```
    for v ∈ activeSet do do in parallel
        h(v) = hcp(v)
        e(v) = e(v) + v.acumExcess
        v.acumExcess = 0
    end for
    for v ∈ activeSet do
        lastWork+ = v.work
    end for
    newSet = {}
    for v ∈ activeSet && h(v) < n do
        newSet.add(v)
    end for
    activeSet.swap(newSet)
    for v ∈ activeSet do
        e(v) = e(v) + v.acumExcess
        v.acumExcess = 0
    end for
end while
```

2.2 The Preflow Operation

Algorithm 2 PreFlow

```
for (s, v) ∈ E do
    f(s, v) = c(s, v)
    f(v, s) = f(s, v)
    e(v) = f(s, v)
end for
```

2.3 the GlobalRelabel Algorithm

Algorithm 3 GlobalRelabel

```
for v ∈ V do do in parallel
    h(v) = n
end for
h(sink) = 0
Queue = [sink]
while !Queue.isEmpty() do
    for v ∈ Queue do do in parallel
        v.discovered = []
        for (v, w) ∈ E_f && w!=s && c_f(v, w) > 0 do
            if w! = sink && h(w) == n then
                h(w) = h(v) + 1
                v.discovered.add(w)
            end if
        end for
    end for
    newQueue = {}
    for v ∈ Queue do
        for w ∈ v.discovered do
            newQueue.add(w)
        end for
    end for
    Queue.swap(newQueue)
end while
```

3 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. The n is the number of vertices. only the number of vertices is used because the time complexity of this algorithm only relates to the number of vertices.

$$speedupfactor_n = BaselineExecutionTime/CurrentExecutionTime$$

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 real implementation, which takes advantage of the hardware features and OpenMP features. The first is the cache miss count. It could be defined as LC_p. Here p refers to a specific max problem. 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. Next experiment is to measure the influence of the thread's position. This report measures the execution time of using different threadcbinding policies. Mainly the close and the spread policy is being tested.

4 EXPERIMENT

Fig. 1. The speed-up factor of the parallelized computation among different node pairs. The testing problem contains 100 nodes. After resolving the anti-parallel edges, the actual nodes number is around 2000.

Fig. 2. The speed-up factor of the parallelized computation among different node pairs. The testing problem contains 300 nodes. After resolving the anti-parallel edges, the actual nodes number is around 8000.

The first experiment is about the speed-up of computing max flow of different pairs. The experiment shown in Figure 1 is based on a graph with 100 nodes, densely connected. Here 100 nodes mean 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 algorithm only deals with the normalized graph. Figure 2 is based on a 300-node problem, which may have around 8000 nodes after normalization to the standard form. The size of the dataset of the above two experiments is 20.

Figure 3 is the execution time of parallelized max flow problem using different threads. The testing graphs have 100000 nodes and nearly 500000 edges each. The time unit is second.

Figure 4 shows the performance of the parallelized max flow algorithm when the edges increase.

Fig. 3. The speed-up factor of the parallelized max flow algorithm, which is tested on 100000 nodes graph.

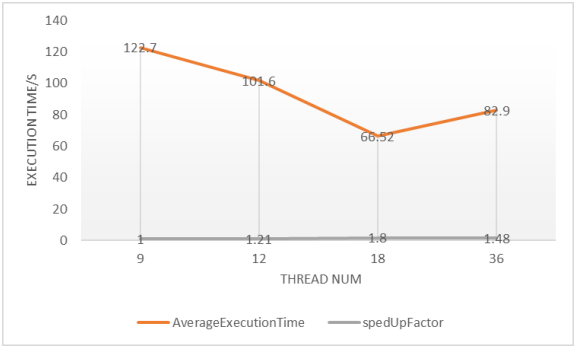
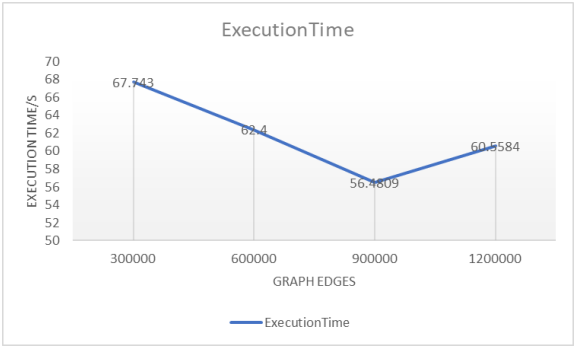


Fig. 4. The parallelized max flow algorithm being tested on graphs with different edge numbers. The vertices count of these tests are all 100000. The threads used are 18.



Thread distribution	Average Instruction speed/(instruction per cycle)
4 outer master thread, 2 inner slave thread	0.57
2 outer master thread,1 inner slave thread	2.68
Single thread	2.66

Table 1. Instruction speed with different threads distribution on the same problem.

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 the max flow algorithm.

Fig. 5. The sample of testing instructions speed.

```
   3308979.03 msec task-clock:u              #    3.733 CPUs utilized
            0      context-switches:u        #    0.000 K/sec
            0      cpu-migrations:u          #    0.000 K/sec
     23451485      page-faults:u             #    0.007 M/sec
9742150518077      cycles:u                  #    2.944 GHz
6027992355465      instructions:u            #    0.62  insn per cycle
 892400071154      branches:u                #  269.690 M/sec
   1148644749      branch-misses:u           #    0.13% of all branches
```

The following Figure 5 is a sample of measurement of instruction speed, where threads are idle in most of the time. The following Table 2 is the comparison of L1 cache load misses. The first row uses an extra structure to store elements, which avoids skipping indexes in the matrix. The other one just operates directly on the matrix. The Table 3 shows the

Optimization Method	Log_{10}(cache miss times)
Using extra space avoid jumping matrix index	7.38
Null	9.59

Table 2. The cache miss times of different indexing policies. One is using extra memory to record neighbors during the common matrix reading index. It then uses these record to help finishing global relabel. While the other method is using matrix index directly, which is normal operation in small size problems.

average delay between using or not using the cache optimized structure. The following Table 4 illustrates the execution on different size of problems when applying different policies.

Optimization Method	Execution Time/s
Using extra space avoid jumping matrix index	8.6
Null	1.2

Table 3. The execution time comparison with different matrix operations. These data are measured on the measurable problem, which is defined in the discussion section.

binding Policy	Average Execution Time/s	standard deviation
close	47.14	2.47
Spraed	46.6	2.19

Table 4. The performance difference when using different thread allocation policies. The test is based on the problem size of 300 nodes, which usually has around 8000 nodes after converting to the standard form. the number of thread is 36.

5 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 of the experiment environment. Because the total computing complexity could be expressed as $O(n^2 * O(f))$. 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 takes 0.1 seconds to solve this single max flow problem. Calculating all pairs, 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 memory. Even though each thread could share the same graph structure in the memory, they still need memory to store the flow network and residual network, which is the same size as the original graph. This memory requirement is also beyond the memory limit.

After illustrating the above issue, we could focus on performance improvement. To the parallelized computation of different pairs, it is easy to see the speed up increase linearly with the thread number with suitable graph size, which could be seen from Figure 1 and 2. However, in a smaller graph size, the total performance degrades. The small graph size usually means graph with less than 50 nodes. One thing needs attention here is that the above tests are on problems of 100 nodes and 300 nodes. But to use the algorithm, we still need to convert them into a standard form, which is required by most of the max flow algorithms. The standard form has no anti-parallel edges. In small graph, multiple threads may need hundreds of execution time than a single thread. To address this problem, this report measures the instruction operation speed, instead of using execution time directly, which will be covered in a later experiment section.

Then let's focus on the performance of parallelized max flow algorithm. According to Figure 3, with the fixed size of the graph, the speed-up factor increases nearly linearly in a certain range. This relationship is prominent in Figure 3, which is tested under the largest graph in this experiment, 100000 vertices. 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 synchronous algorithm. Threads are synchronized after each parallel section. It is simple and the synchronization complexity will just increase linearly with the thread counts. However, we should notice that 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. With a larger graph, the speed-up factor may keep its linear increasing trend. Each thread finishes its own work and waits for each other. The following part will illustrate the reason. It is the same thread idle problem.

Before going to the analysis of the idle problem. We need to take a look at Figure 4, which shows how the algorithm performs when edges increase. the tests are performed on graphs with the same vertices number. It can be seen the total execution time does not change much with the increasing edges, which fits the time complexity model of this algorithm.

With Figure 5 and Table 1, we can see the single thread instruction execution speed is much higher than the case with 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 have their own unique context. According to the implementation, some of them may be pulled from the idle thread pool while others may be created on demand. They start at a 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 caught up. This hang-up and wake-up behavior may cause extra time wasted.

The next experiment is about the cache problem. This algorithm used by 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, unpredictable indexing or distant index 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, this report tests different strategies for the same problems. One measure is based on the execution time. According to Table 3, the algorithm that does not utilize this feature costs a significant amount of time

more than the one that uses it. Furthermore, 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 for a much longer period of time. So only small problems could be used. On large graph, using perf to monitor will cost more than 12 hours to measure. With small graphs, in most cases, the cache miss rate does not differ too much. However, the cache miss number is obviously large in the unoptimized algorithm, which could be seen from Table 2. Due to the cache miss counts are huge, the log function is used to show the result. In large graph cases, this problem is much more serious. For example, the largest 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 exceeds the capacity of a 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 in 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 and the graph size. The reason may be this algorithm's locality feature. Though threads on different sockets have different memory accessing speeds, 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 performance.

REFERENCES

[1] Richard J. Anderson and João C. Setubal. 1995. A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem. *J. Parallel Distributed Comput.* 29, 1 (1995), 17–26. https://doi.org/10.1006/jpdc.1995.1103

[2] Niklas Baumstark, Guy Blelloch, and Julian Shun. 2015. Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm. (July 2015). arXiv:1507.01926 [cs.DS]

[3] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., and Stein Clifford. 2009. Eintroduction to Ealgorithms Third Edition. https://doi.org/10.1.1.708.9446