Optimizing and Parallelizing the Push-Relabel Algorithm

Ege Yurtsever - 2025

Introduction

To date, many of the asymptotically fastest maximum-flow algorithms are push-relabel algorithms, and the fastest actual implementations of maximum-flow algorithms are based on the push-relabel method. Moreover other flow problems, such as the minimum-cost flow problem, can be solved efficiently by push-relabel methods.[1] This report documents the process of testing different sequential optimization methods and approaches on the push-relabel algorithm, and then parallelizing it.

Generic Goldberg-Tarjan Push-Relabel Algorithm

The push-relabel algorithm maintains a *preflow*, where flow conservation is relaxed and excess flow is allowed at intermediate vertices. Initially, the source node pushes as much flow as possible to its neighbors, saturating the outgoing edges. Throughout the algorithm, flow is pushed from vertices with excess to neighboring vertices if the neighboring vertex is at a lower "height"; this is the **push** operation. If no such push is possible, the vertex's height is increased—this is the **relabel** operation. These two operations are applied iteratively when applicable, on the vertices with excess flows until no more valid pushes or relabels are possible.

GENERIC-PUSH-RELABEL(G)

1- INITIALIZE-PREFLOW(G,s)
2- while there exists an applicable push or relabel operation
3- do select an applicable push or relabel operation and perform it[2]

The generic push-relabel algorithm, as seen above, does not specify the method of selection and ordering of the applicable operations, and leaves the implementation flexible. Because of this, sometimes it is referred to as the Push-Relabel Method instead of an algorithm. Therefore, there are lots of implementation strategies available with different time complexities.

Definition: Discharge Operation

It is beneficial to define the discharge operation before diving any deeper for simplification. The **discharge** operation encapsulates the push-relabel process: for a given vertex with excess flow, discharge repeatedly attempts to push flow to its neighbors, and performs a relabel when no more pushes are available. Discharge continues until the vertex no longer has excess flow.

Definition of the discharge operation reduces the push–relabel algorithm to repeatedly selecting an active node to discharge.

Active Node Selection Strategies

Short explanations of the most popular strategies are given below for context:

---

[1] Cormen, Leiserson, *Introduction to Algorithms (2nd ed.), 669.*
[2] ibid.

Relabel-to-Front

This is a variant of the generic push-relabel algorithm that maintains a list of all vertices and processes them in order. If a vertex's label increases during discharge, it is moved to the front of the list, hence the name. This adaptive ordering can lead to faster convergence on some graphs by clustering work around recently updated regions.

Highest-Label

The algorithm always selects the active node with the highest current height. The motivation is to prioritize nodes that are "closer" to being able to push flow toward the sink. It can reduce the number of relabel operations by aggressively pushing flow from the top down.

FIFO (First-In, First-Out)

The algorithm maintains a queue of active nodes. When a node becomes active, it is appended to the end of the queue. The algorithm always discharges the node at the front. If a node is still active after discharge, it is re-added to the end. It tends to spread the flow broadly, avoiding extreme label heights early in the execution.

Time Complexity of Push-Relabel (Sequential)

The generic push–relabel algorithm has $O(V^2E)$ time complexity.
The Relabel-to-Front variant has $O(V^3)$ time complexity.
The FIFO variant also has $O(V^3)$ time complexity.
The Highest-Label variant has $O(V^2\sqrt{E})$ time complexity.

As it is observed that the FIFO variant outperforms the Highest-Label variant on modern, large maximum flow problem instances, this report focuses on the FIFO variant and continues working on it for optimization and parallelization.[3]

Implementation of the Basic FIFO variant

The FIFO variant of the push-relabel algorithm is implemented and tested for correctness on various graphs. The algorithm performed much more iterations than expected when tested with dense flow networks, this clearly led to the conclusion that an aggressive heuristics implementation would be a necessity later on for the best possible performance. Push and Relabel (thus Discharge) operations implemented in this version are not to be touched till parallelization, as the additional heuristics do not require changing the core structure of the algorithm.

Pseudo-Code:

Notation:
G = (V, E): Flow network with vertices V and edges E
source: The source vertex
sink: The sink vertex

---

[3] Baumstark, N.; Blelloch G.; Shun J. *Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm*, (Carnegie Mellon University, Karlsruhe Institute of Technology, 2001).

n: Number of vertices in G
capacity(u, v): Capacity of edge (u, v)
flow(u, v): Current flow on edge (u, v)
excess(v): Excess flow at vertex v
height(v): Height (label) of vertex v
Q: FIFO queue of active vertices
in_queue(v): Boolean indicating if vertex v is in queue

Algorithm:

```
function FIFO_MaxFlow(G, s, t):
	Initialize all e(v) = 0, h(v) = 0
	h(s) = |V|

	// Preflow
	for each edge (s,v):
	f(s,v) = c(s,v)
	f(v,s) = -c(s,v)
	e(v) += c(s,v)
	if e(v) > 0 and v ≠ s and v ≠ t: add v to Q

	// Main loop
	while Q not empty:
	u = Dequeue(Q)

	// Discharge u
	while e(u) > 0:
		// Try to push
		if exists an edge (u,v) where f(u,v) < c(u,v) and h(u) = h(v) + 1:
		// Push operation
		d = min(e(u), c(u,v) - f(u,v))
		f(u,v) += d
		f(v,u) -= d
		e(u) -= d
		e(v) += d
		if e(v) > 0 and v ≠ s and v ≠ t and v not in Q: add v to Q
		else:
		// Relabel
		h(u) = 1 + min{h(v) | (u,v) ∈ E, f(u,v) < c(u,v)}

	if e(u) > 0 and u ≠ s and u ≠ t: add u to Q

	return sum of f(s,v) for all v
```

Optimizing the Algorithm Sequentially: Global Relabeling Heuristics

One of the most impactful optimizations for the push-relabel algorithm is the use of ***global relabeling***, a heuristic designed to improve the accuracy of the height function used to guide push operations. In the basic push-relabel method, node heights are updated locally through relabel operations based on neighbors' heights as mentioned, which may cause outdated or suboptimal height assignments over

time. The global relabeling heuristic addresses this by periodically recalculating the exact shortest-path distance from every vertex to the sink, effectively reinitializing the height labels with more globally accurate values. This is typically achieved through a ***backward breadth-first search (BFS)*** starting from the sink node. Since pushing is only allowed from higher to lower heights, maintaining a more accurate height landscape ensures flow is pushed more directly toward the sink, reducing unnecessary relabels and inefficient pushes.

While global relabeling brings significant overhead to the algorithm, it dramatically improves the overall performance. Moreover it is known to be beneficial to run an initial global relabeling at the beginning of each run, as it usually gives better performance. The global relabeling method implemented for the sequential version of the push-relabel can be seen below:

```
void
PushRelabelParallel::globalRelabel(std::vector<std::vector<FlowNetwork::Edge>>&
graph, std::vector<int>& height, int source, int sink, int n) {
       height.assign(n, n);
       height[sink] = 0;
       std::queue<int> q;
       q.push(sink);
       std::vector<bool> vis(n, false);
       vis[sink] = true;
       while (!q.empty()) {
          int v = q.front(); q.pop();
          for (auto& e : graph[v]) {
              int u = e.to;
              int rev = e.rev;
              if (u >= 0 && u < n && rev < graph[u].size()) {
                auto& rev_e = graph[u][rev];
                if (rev_e.capacity - rev_e.flow > 0 && !vis[u]) {
                        vis[u] = true;
                        height[u] = height[v] + 1;
                        q.push(u);
                }
              }
          }
       }
       height[source] = n;
}
```

The sequential version has been tested with graph sizes 1000 and 2000, edge probabilities 0.1 and 0.5, and max flow capacity of 100 per vertice. The global relabeling heuristics implementation with one global-relabel per ***n*** (graph size) iterations averaged a ~ **46x** speedup when compared with the standard FIFO version.

Timing - Cost Analysis of the Sequential Algorithm

Problems with observing significant performance gains while parallelizing the main components of the algorithm (push, relabel) were experienced in previous approaches (without global relabel). In order to set the parallelization goals and expectations better, a cost analysis is made on the sequential version, before any parallelization.
The algorithm is tested with the graph size n = 10000, edge probability 0.3, and max capacity per vertice = 100. Each test is run 3 times and the average is taken into account. The average cost analysis of the mentioned runs are given below:

Operation	Timing (ms)	Timing Percentage
Push Operations	20.92	3.37%
Relabel Operations	18.25	2.94%
Queue Management	3.88	0.62%
Global Relabels	495.63	79.73%
Other	82.93	13.34%

It is clearly observed that most of the time is spent during the Global Relabel operations. The Global Relabeling is currently utilizing a sequential backwards breadth-first search, this can be parallelized to reduce the bottleneck and improve the timings.

Parallelizing the Global Relabeling Heuristics

The main structure of the sequential global relabel is kept intact. Initialization part is parallelized as the method is called multiple times in a push-relabel run. Each thread is assigned a portion of the current BFS ***frontier***. For each node ***v*** in the frontier, the thread examines each incoming residual edge of ***v***. If a neighbor has not yet been visited (***height[u] == n***) and there is residual capacity from ***u*** to ***v***, the thread attempts to update ***u***'s height to ***current_level + 1***. Threads independently collect newly discovered nodes (***u***) in their local frontiers, avoiding contention. Afterwards these local frontiers are merged into the global ***next_frontier*** before the next BFS level starts. The process continues until no more nodes can be labeled. The parallelized global relabeling method implemented for the sequential version of the push-relabel can be seen in the pseudo-code below:

```
procedure GLOBAL_RELABEL(graph, height, source, sink, n)
	parallel for i in 0 to n-1 do
		height[i] ← n
	end parallel for
	if n > 0 then
		height[sink] ← 0
	end if
```

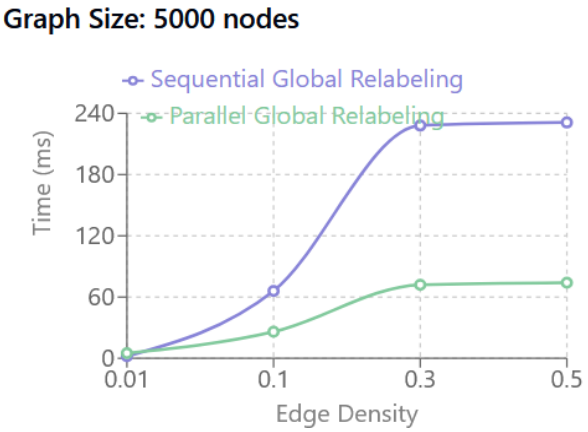
```
        frontier ← [sink] if n > 0 else []
        current_level ← 0

        num_threads ← max_threads()
        local_frontiers[0..num_threads-1] ← empty lists

while frontier is not empty do
         clear all local_frontiers
         next_frontier ← empty list

         parallel for each node v in frontier do
                 for each edge (v → u) in graph[v] do
                   if residual capacity of reverse edge (u → v) > 0 and
height[u] = n then
                     critical section:
                          if height[u] = n then
                            height[u] ← current_level + 1
                            add u to thread's local_frontier
                          end if
                     end critical section
                   end if
                 end for
         end parallel for

         merge all local_frontiers into next_frontier
         frontier ← next_frontier
         if frontier is not empty then
                 current_level ← current_level + 1
         end if
end while

        if n > 0 and source ≠ sink then
               height[source] ← n
        end if
end procedure
```

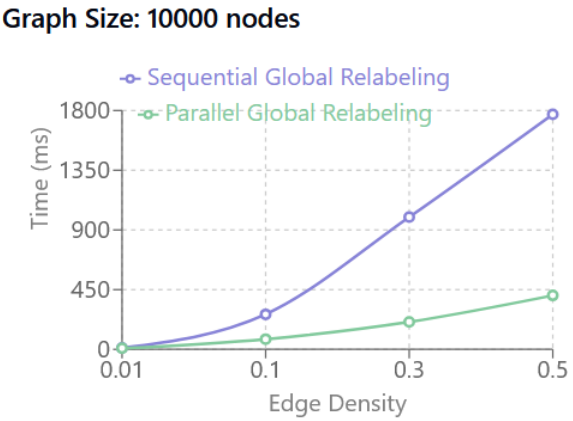
Testing Global Relabeling Variants

The parallel global relabeling heuristics is tested against the sequential variant for performance observation, and to consider a hybrid approach later. Results of the testing and the plots are given below: (Parallel runs used 8 threads)

Graph Size	Edge Density	Sequential Time (ms)	Parallel Time (ms)
1000	0.01	0	5
1000	0.1	1	5

1000	0.3	7	3
1000	0.5	8	2
5000	0.01	2	2
5000	0.1	66	26
5000	0.3	228	72
5000	0.5	231	79
10000	0.01	24	16
10000	0.1	342	116
10000	0.3	1081	177
10000	0.5	1772	406





It is observed from the data that the parallel global relabeling heuristics start to outperform the sequential version as early as around $n = 1000$, for $d = 0.3$. This suggests that it can be beneficial to keep the sequential version in-use for really small flow networks, however for most cases the parallel version heavily outperforms. With this observation, a hybrid approach that uses the sequential global relabel for small graphs and parallel version for the rest is implemented.

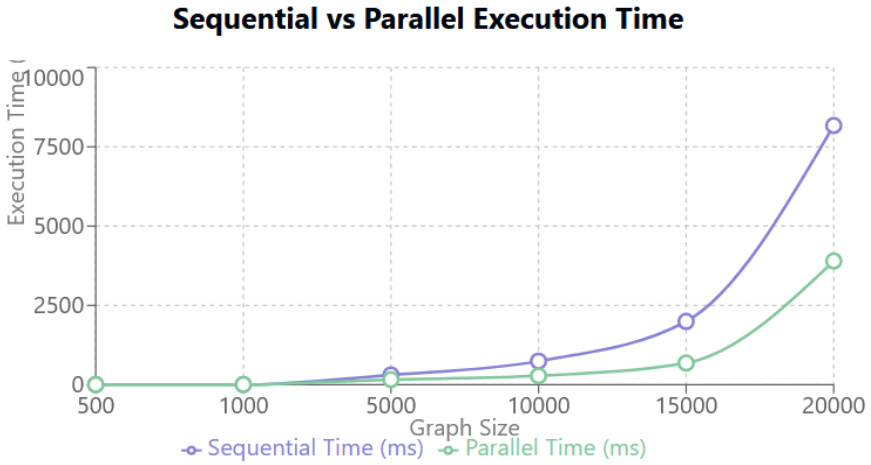
Hybrid Global Relabeling

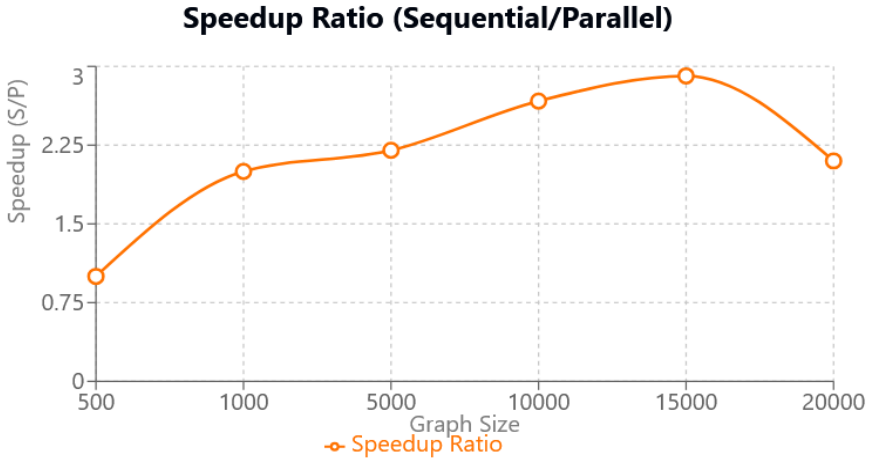
A hybrid variant, where the it uses the sequential version for smaller networks, and the parallel version otherwise is created. In order to tune the threshold, the testing above is used as reference and upon further tweaking it is set as $n = 1300$.

Speed-up After Global Heuristics Parallelization

The push-relabel parallel version is tested against the sequential version, in order to see the impact of parallelizing the heuristics. The test results are given below: (Edge density = 0.3 and threads: 8 for all)

Graph Size	Sequential Time (ms)	Parallel Time (ms)	Speedup (S/P)
20000	8172	3897	2.10
15000	1994	684	2.91
10000	743	278	2.67
5000	303	138	2.20
1000	4	2	2.00
500	2	2	1.00





Parallelizing the Push and Relabel Operations

Even though it is expected to not be dramatically impactful like parallelizing global heuristics (as the sequential cost of push/relabel are small compared to the heuristics), there are still performance improvements possible on dense graphs upon parallelizing the push and relabel operations (especially the push). It is expected that the parallelized push and relabel operations might not perform any better, possibly even worse compared to the sequential versions when tested on **sparse** graphs (this expectation is due to previous failed attempts of parallelization of a different version). It is expected to perform better on large and dense graphs when compared with the sequential push/relabel version.

Instead of letting all threads work as much as possible with little synchronization, and ensuring correctness with reliance on atomic operations, a more synchronous approach was preferred. This is due to the fact that while the overhead from the parallelization causes poor performance on sparse graphs, as the graph density and thread count increases the contention incredibly shoots up. This approach might work better if tuned specifically for certain graphs, but it is not a preferable general solution. While the parallelization of the push-relabel might sound easy, keeping the contention low is a challenging task. The parallelization strategy used is explained below:

In each iteration, a batch of active vertices is collected from a global queue into a temporary list. Vertices are distributed from this batch among multiple threads. Each thread processes these vertices in their thread-local storage, and atomically discharges these vertices. Status of a queue being in an active queue is updated atomically, if necessary. After threads finish their work, all thread-local storage is merged into the global queue appropriately. At last a final check for a vertice still having excess after a relabel is checked, and then an iteration is complete. The algorithm runs till the active queue is empty or max iteration limit is reached. Pseudo-code of the main loop can be seen below:

```
procedure MaxFlow_ActiveParallel(G, s, t, num_threads)
  initialize height[], excess[], in_queue[], active_queue from preflow at s
  globalRelabel(G, height)
  rebuild active_queue and in_queue from atomic_excess

  iterations ← 0
  total_local_relabels ← 0
  relabel_since_last ← 0
  global_freq ← |G.V|

  while !active_queue.empty() and iterations < MAX_ITERATIONS do
    if relabel_since_last ≥ global_freq then
      globalRelabel(G, height)
      relabel_since_last ← 0
      rebuild active_queue and in_queue
      if active_queue.empty() then break
    end if

    // Form batch of the currently active nodes
    current_batch ← []
    while !active_queue.empty() do
      u ← active_queue.pop()
      current_batch.append(u)
    end while

    if current_batch.empty() then
      iterations ← iterations + 1
      continue
    end if

    // Parallel discharge
    batch_relabels ← 0
```

```
    parallel for each thread tid in [0...num_threads-1] do
      clear thread_local_newly_active[tid]

      for each u in current_batch assigned to this thread do
        if atomic_excess[u] > 0 and height[u] < |G.V| then
          d ← discharge_Atomic(G, atomic_excess, height,
                               thread_local_newly_active[tid], u, s, t)
          batch_relabels ← batch_relabels + d
        end if

        Critical:
          in_queue[u] ← false
      end for
    end parallel

    total_local_relabels ← total_local_relabels + batch_relabels
    relabel_since_last ← relabel_since_last + batch_relabels

    // Merge new active nodes
    for tid from 0 to num_threads-1 do
      for each v in thread_local_newly_active[tid] do
        if !in_queue[v] and atomic_excess[v] > 0 and height[v] < |G.V| then
          active_queue.push(v)
          in_queue[v] ← true
        end if
      end for
    end for

    // Re-activate batch nodes if still with excess
    for each u in current_batch do
      if !in_queue[u] and atomic_excess[u] > 0 and height[u] < |G.V| then
        active_queue.push(u)
        in_queue[u] ← true
      end if
    end for

    iterations ← iterations + 1
  end while

  return atomic_excess[t]
end procedure
```

Test Results After Push and Relabel Parallelization

Testing Results of the Parallel Version (all), only Global Relabel Parallel Version (oGRP), and the sequential version can be seen below:

Edge Probability 0.3, 8 threads (Dense Graph):

Graph Size	Sequential	Parallel (GRH)	Parallel (All)
40000	63677	23164	16459
30000	13885	8762	6851
25000	12373	7434	5572
20000	7990	4030	3132
15000	1780	918	912
10000	863	304	314
5000	315	166	167
1000	3	5	4

On dense graphs, it is observed that even for small sizes the fully parallel version does not perform bad compared to the sequential discharge version. Much better performance is observed for large and dense graphs, however any larger graph is not testable considering the host machine. (it took 34 minutes to calculate max flow on graph size 60000 sequentially and experienced several memory errors afterwards)

Edge Probability 0.01, 8 threads (Sparse Graph):

Graph Size	Sequential	Parallel (GRH)	Parallel (All)
10000	53	48	83
20000	165	110	103
30000	712	378	455
40000	809	338	389
55000	2503	1035	906
70000	4597	1929	1557
100000	10951	4390	4489

It is observed above that the discharge-parallelized version only shows reliable improvements for medium-sized dense graphs. However it fails to reliably provide speedup on sparse graphs (only shown speedup around 55000-70000 and didn’t scale properly). This concludes that the discharge parallelization is only suitable for dense flow network applications and the sequential discharge is viable for sparse graphs. This concludes the parallelization of the push-relabel algorithm.

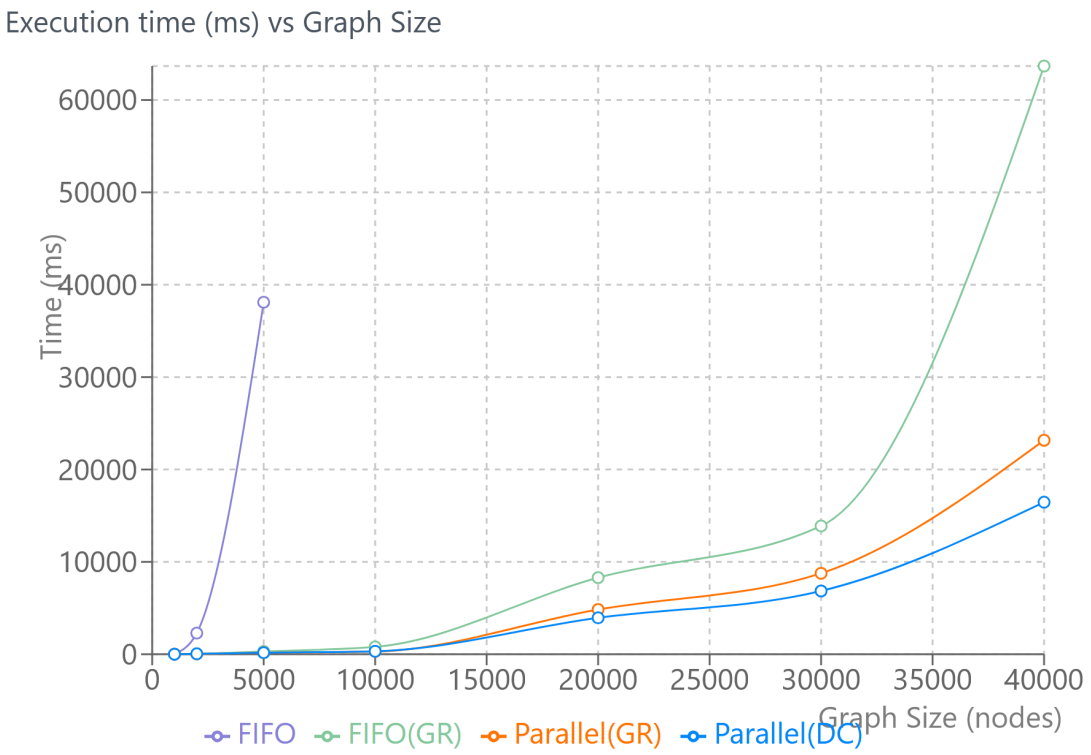
Final Test Results:

Final comparison between the different versions discussed in this report can be seen below:

FIFO: The standard FIFO Push-Relabel without extra heuristics
FIFO(GR): Final sequential version with global relabeling
Parallel(GR): Parallelized heuristics with sequential discharge
Parallel(DC): Parallelized heuristics with parallelized discharge

Edge density = 0.3, threads = 8 for the final tests;
(FIFO becomes untestable for the graph sizes above 5000 due to incredible time cost)

Graph Size	FIFO	FIFO(GR)	Parallel(GR)	Parallel(DC)
1000	6	5	5	7
2000	2290	43	32	42
5000	38109	305	168	164
10000	-	793	295	320
20000	-	8289	4842	3941
30000	-	13885	8762	6851
40000	-	63677	23164	16459



A speedup plot of the parallel versions, compared with the FIFO(GR) is given below:

It is visible that as the graph sizes increase, the speedup of the parallel versions also tend to increase. However, the test machine is not capable of reliably running bigger max flow tests while ensuring the correctness of the result, due to both memory and performance limits.

REFERENCES

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms (2nd ed.)*, (The MIT Press, 2001), 669.
Baumstark, N.; Blelloch G.; Shun J. *Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm*, (Carnegie Mellon University, Karlsruhe Institute of Technology, 2001).