Universiteit Leiden
Opleiding Informatica

Utilizing a tuple-based optimization framework

for graph algorithms

Name:	L.J. Peters
Date:	Thursday 5th January, 2017
1st supervisor:	Prof. Dr. H.A.G. Wijshoff
2nd supervisor:	Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

As graph problems are among the most common in various scientific research fields, being able to solve them quickly is wildly applicable. In this thesis, we apply a tuple-based optimization framework to the max flow min cut problem to improve performance in existing algorithms. Using this framework, we create several parallel implementation of the Push-Lift algorithm and test its performance on several relevant data sets.

Contents

Chapter 1

Introduction

Graph algorithms are in the most commonly used algorithms in computer science. Most general real world problems can be solved by converting the problem to a graph problem and solving that using a known algorithm. As such, optimizations in graph algorithms are optimizations to various fields in computing.

In this thesis, we attempt to optimize graph algorithms using the forelem language as described in [Rie14]. For this thesis, we have looked at the max flow min cut problem as an example graph problem.It was first formulated in 1954 as a way of modelling Soviet railway flow [HR55]. The problem asks what, given a graph (V, E) in which every edge has a flow capacity c, is the maximum flow from a given source to a given sink. For example, the in the graph shown in Figure 1.1 has a maximum flow of 14, utilizing the capacities as shown.

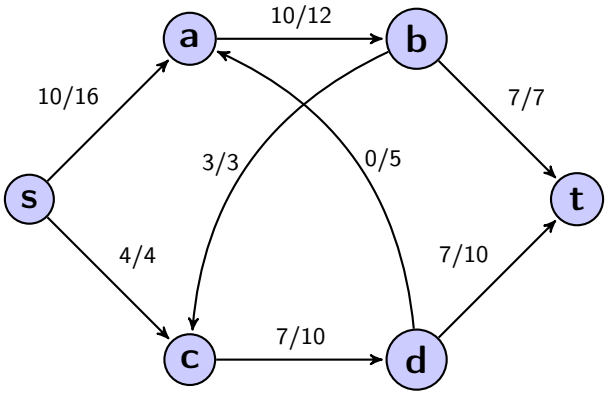

Figure 1.1: A simple graph with flows.

It should be noted that the flow configuration for the maximum flow is not necessarily unique, although in this example it is. The only requirement is that for each node (excluding the source and the sink) the incoming flow is equal to the outgoing flow.

1.1 Current solutions

It should be noted that the max flow problem is not an unsolved problem. There have been several implementations, with various time and memory complexities. Early algorithms used the augmenting paths [Hono8] and construct additional routes from sources to sinks, most notably the Ford Fulkerson algorithm (see subsection 1.1.1).

Goldberg et al. produced a different result by constructing a preflow instead of a valid flow [GT86], and pushing as much flow from the source to the sink, see subsection 1.1.2. This thesis will attempt to work with the Push Lift algorithm proposed by Bo Hong [Hono8], which allows a lock-free parallel execution of the algorithm.

1.1.1 Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm [FF56] computes for a given directed graph of edges (u, v, c) with u and v nodes and c the capacity for the edge, the maximum "flow" from a given source s to a sink t, given that the flow of an edge e can never exceed its capacity. The algorithm assumes that

- there are no parallel edges, meaning that

$$\forall e_1(u_1, v_1, c_1), e_2(u_2, v_2, c_2) : e_1 \neq e_2 \implies u_1 \neq u_2 \wedge v_1 \neq v_2$$

  , and

- there are no self loops, meaning that

$$\nexists e(u, v, c) : u = v$$

The former condition is easily satisfied, as we can just merge two parallel edges by adding their respective capacities. The latter we can satisfy by can completely ignoring any self loops. This is valid, as self loops do not contribute to the total flow.

The algorithm works as follows. We define $f(u, v)$ and $c(u, v)$ as the flow and capacity respectively from u to v. For all u and v, we initialize $f(u, v)$ as zero. For all edges (u, v, c), we define $c(u, v) = c$ and for all other combinations, we define $c(u, v) = 0$.

Algorithm 1 A pseudo-code representation of the Ford-Fulkerson algorithm

```
function FindPath(source, destination, path)
    if source = destination then
        return path
    else
        for all e : e ∈ E ∧ e.u = source ∧ e ∉ path ∧ f(e.u, e.v) < c(e.u, e.v) do
            newPath ← FindPath(e.v, destination, path ∪ e)
            if newPath ≠ ∅ then
                return newPath
            end if
        end for
        return ∅
    end if
end function

while there is a useful path from s to t do
    path ← FindPath(s, t, ∅)
    δ ← min(c(e.u, e.v) − f(e.u, e.v) ∀e ∈ path
    for all e ∈ path do
        f(e.u, e.v) ← f(e.u, e.v) + δ
        f(e.v, e.u) ← f(e.v, e.u) − δ
    end for
end while
maxflow ← Σ_{e:e.v=t} f(e.u, e.v)
```

1.1.2 Push-Relabel algorithm

We define the set of nodes, V, and the set of edges E. Furthermore, we define the function $\Delta(u)$ which denotes the current excess flow[1] at node u, and $F(u, v)$ and $C(u, v)$, which denote the current flow and the flow capacity from u to v respectively. Both start at 0 if the edge does not exist.

A new property that is used, is the height function $H(u)$, which holds the current height for a given node. This height function is used to enforce an ordering.

The algorithm constsists of two operations. The Push operation pushes excess flow from a node u to a connected node v. It is allowed only when $H(u) = H(v) + 1$. If no valid push is possible, a node must be relabeled with the Relabel operation. This method increases $H(u)$ to the minimum level so that a new push is allowed.

The algorithm starts by performing a satisfying push on the source node, which fully uses all outgoing edges on that node. After that, the algorithm attempts to either push the excess flow to the sink, or, if it is not possible to push, relabels the node. When there are no longer any nodes with an excess flow, the algorithm is finished and the maximum flow can be calculated by summing the flow coming into the sink t or the flow going out of the source s. A pseudo-code implementation can be seen in Algorithm 2.

[1] created by the preflow

Algorithm 2 A pseudo-code implementation of the generic Push-Relabel algorithm

function PUSH($e, amount$)
  $\delta \leftarrow \min(amount, C(e.u, e.v) - F(e.u, e.v))$
  $\Delta(e.u) \leftarrow \Delta(e.u) - \delta$
  $\Delta(e.v) \leftarrow \Delta(e.v) + \delta$
  $F(e.u, e.v) \leftarrow F(e.u, e.v) + \delta$
  $F(e.v, e.u) \leftarrow F(e.v, e.u) - \delta$
end function

function RELABEL(u)
  $H(u) \leftarrow \min(H(v) : F(u, v) < C(u, v)) + 1$
end function

for all $e : e \in E \wedge e.u = s$ **do**
  PUSH(e, ∞)
end for

$H(s) \leftarrow |V|$
$H(V \setminus s) \leftarrow 0$

while $\exists u : u \in V \setminus t \wedge \Delta(u) > 0$ **do**
  if $\exists e : e \in E \wedge e.u = u \wedge C(e.u, e.v) > F(e.u, e.v) \wedge H(e.u) = H(e.v) + 1$ **then**
    PUSH($e, \Delta(u)$)
  else
    RELABEL(u)
  end if
end while

$maxflow \leftarrow \sum_{e:e.v=t} F(e.u, e.v)$

1.2 The forelem language

For this research, we will use the forelem framework [Rie14] to represent a general algorithm for the max flow problem. In this framework the data is expressed as several unordered sets of tuples and the algorithm can be expressed as an unordered loop with some operations over those tuples.

Once an algorithm is formatted that way, it can be materialized, transformed and concretisised into a regular programming language (such as C or Python) to be compiled and executed normally. We will discuss a number of possible the transformations here. The complete discussion can be found chapter 9 of [Rie14] but is out of the scope of this thesis.

Algorithm 3 forelem algorithm for spare matrix dense vector multiplication

```
for (i = N; i ≥ 1; i−−)
{
    int sum = 0;
    forelem (j; j ∈ pA.row[i])
        sum += B[A[j].col] * A[j].value;

    C[i] = sum;
}
```

Algorithm 4 *forelem* algorithm for summing the products of a set of tuples

```
sum = 0
forelem (i; i ∈ pA)
    sum = sum + A[i].part1 * A[i].part2
```

First, we look at Algorithm 3. Here, we have two dense vectors `B` and `C`, and a tuple representation of a matrix `A(column, row, value)`. We want to compute `A * B` and store the result in `C`. To do this, we iterate (in reverse) over `C` and process the tuples of the relevant row in `A` (i.e. for which the row number is `i`). In order to generate executable code from this forelem-representation, we materialize these subsets into some structure. A possible solution is to create some multidimensional array, for which the outer index is our row number. However, since not every row will have the same number of non-zero elements in each row, this will lead to problems. Instead, we can create some indirection of this, which will allow for varying length arrays within the outer array. This is what we will be looking into in this thesis. The generalization of this is called Loop dependent materialization. However, since the other forms are not that applicable to this thesis, we will not be addressing them.

Another thing we may change is the structure of the tuples themselves. Consider Algorithm 4. Here, our tuples consist of two numbers, and we want to know the sum of their products. We can store them just as that: an array of tuples. However, it may be beneficial to split our array of tuples into two arrays of the two parts. Doing so may improve cache performace, depending on the access pattern. This procedure is called

structure splitting.

Finally, not every element of the tuple may be necessary for the computation. Looking again at Algorithm 4, we can imagine that the tuples in `A` had an additional element, `part3`, which is not used in the computation. We can optimize this by leaving it out of the tuples, and creating a new array containing only the relevant parts, which would be smaller. We call this horizontal iteration space reduction. The smaller array might then improve cache performance. However, since in our algorithm all data is needed in some way, we chose not to implement this. Small improvements may be had by applying this to small local loops, but this would be difficult to implement manually in the absense of an automated compiler.

Chapter 2

Implementation

2.1 Push-lift

The algorithm we consider the most suitable for optimization, is the Push Lift algorithm by Bo Hong et al. This is because it can be executed in parallel as it was designed that way and because it requires no locking between workers during execution. It uses the same preflow mechanics as the Push-Relabel [GT86] but has a few slight differences.

The algorithm considers a as a collection of nodes V and a capacity function $c(u, v)$ indicating the remaining capacity from u to v. Like the Push-Relabel algorithm, Push-Lift first establishes a preflow, in which some nodes have more incoming flow than outgoing flow. This is done by using each edge going out of a source to its full capacity.

The Push-Lift algorithm also has a similar height function H. At the start of the algorithm, the height function is initialized as follows:

$$H(u) \leftarrow \begin{cases} |V| & \text{if } u \text{ is a source} \\ 0 & \text{otherwise} \end{cases}$$

In order to construct a valid flow, the algorithm then looks at each active node and attempts to *push* some of the excess incoming flow to outgoing edges. A node u must push to its lowest neighbour v for which there is still capacity left in the connecting edge, and $H(u) > H(v)$. This is different from the Push-Relabel algorithm, where a push is only allowed from u to v when $H(u) = H(v) + 1$, with no further preference over which node to choose.

If no such push is possible, the node must be lifted. A *lift* operation changes the height of node u, such that $H(u) \leftarrow \min\{H(v)|c(u,v) > 0\} + 1$, that is, to the minimum of all neighbours of u that have capacity in the edges plus 1, so that in the next iteration, a push is possible.

Nodes are distributed over workers, which can work on either one node or a set of nodes. The algorithm terminates when there are no longer nodes with more incoming flow than outgoing flow, bar the sink nodes. This algorithm can find the maximum flow in $O(|V|^2|E|)$ operations [Hono8].

2.2 *forelem* implementation

To implement the above algorithm in forelem, we need to structure our data as sets of tuples. As the algorithm considers properties on edges, being the remaining capacity they have, and nodes, being their height and current excess flow.

It then makes sense to consider two sets of tuples. One of edges E containing tuples (u, v, c, f) with the edge source, sink, capacity, and current flow, and one of nodes V containing tuples (u, h, e), node, height, and excess. The last has one record per u, and is (for integer u) probably better off being an array or a map. In the following implementation, we will consider it as such.

We use a capacity and a remaining capacity for the flow, so that we can combine $c(u, v)$ and $c(v, u)$ in one tuple, instead of 2. This allows the algorithm to be simpler. Also, this circumvents the limitation that it is very difficult in *forelem* to work on one tuple, and then to modify another.[1]

The finalized implementation for the Push-Lift algorithm can be found in Appendix A.

2.3 DAS-4 and MPI

We chose the DAS-4 as our experimentation platform. This means any parallelization would be done using MPI. This poses a slight challenge, as Hong's Push-Lift was previously engineered for and profiled on a shared memory system. MPI has no such thing, and instead relies on message passing for communication.

To work around this, we implement a solution in which each worker has knowledge of the updates that are relevant to that worker.

Firstly, we need to divide the work. This is done by modulo division, such that every worker has all nodes $u : u \mod workerCount = workerNo$. Then, we need to know what information is important which nodes.

[1] It should be noted however, that this is not impossible. Instead, you could make a *forelem* loop that you know will only match a single tuple. This does add a lot of loops, with a lot of variations that will not be efficient.

For push operations, this is obvious, as both the endpoints of the edge being pushed over need to know what is happening. This means that we need pass at most one message, because at least one of the nodes is owned by the worker performing the operation. When a push operation is performed, a check is done to see if the destination has the same owner as the origin, and if not, a tuple (u, v, δ) is sent to communicate the push happening.

For lift operations, this is slightly more complicated. As any worker that could theoretically push to a given node needs to know that node's height, all information must be pushed there as well. To implement this, we precompute the set of neighbours for each node, and store those sets. When a lift operation is performed on u, all nodes $v : (u, v, c, f) \in E$ are sent the increment in height. This ensures that multiple consecutive lift operations do not interfere.

Finally, we need to ensure termination. In a shared memory implementation, one could regularly check the current excess at every node, and terminate if there are no more active nodes. Because we attempt to minimize communication, no single worker will necessarily know all excess flows, and more importantly, no single worker can be sure there are no messages in transit that will change this. To deal with this, we implement Safra's algorithm.

2.3.1 Safra's algorithm

Safra's algorithm [Fok13] works by repeatedly passing along the workers a token. It works by passing a token along the workers. The token is filled with a color, black or white, and a message counter. If the message counter indicates there are no messages in transit, and the color indicates there are no more active workers.

This way, Safra's algorithm ensures that every node is inactive, and that there is no way a node could become active again. Its overhead is minimal, as the token is only passed when a node becomes inactive, so it does not interrupt execution when a worker is busy. After the entire algorithm is finished, the token is passed at most 2 times through every node.

2.4 Generating implementations

Due to time constraints, it was not possible to create a program that could materialize the *forelem* implementation of Push-Lift (see appendix A) into a compilable or runnable algorithm. Instead, we chose to implement the algorithm by hand and then vary the way we handle the indices for the graph structure. By materializing for the origin node of edges, we arrive at a structure similar to adjacency lists. These are illustrated as follows:

	Implementation 1	Implementation 2	Implementation 3
Retrieve remaining capacity for edge	$O(B)$	$O(\log \lvert E \rvert)$	$O(\lvert E \rvert)$
Retrieve node excess / height	$O(1)$	$O(\log \lvert V \rvert)$	$O(1)$
Determine node neighbours	$O(B)$	$O(B + \log \lvert V \rvert)$	$O(\lvert E \rvert)$

Table 2.1: Various complexities for the proposed implementations.

1. For the first implementation, we use nested varying length arrays. Height and excess are stored per-node in a second array as tuples.

2. The second implementation we constructed uses a sparse array (implemented as `map`, backed by a red-black tree) as the data structure for the capacities.

3. Our final implementation was a naive one, with all edge stored a an array of tuples, with the index set computed within the loop. This would serve as a baseline, but it did not complete any of the test runs within the allowed time limit of 120 minutes[2] and will therefore be excluded from the rest of this thesis.

This in itself provides us with some complexities for some operations associated with the algorithm. These complexities are shown in Table 2.1. Here, $|V|$ denotes the number of nodes, $|E|$ the number of edges, and B the branching factor of the graph, which is the average number of edges per node.

From the complexities itself we can see that the first implementation has a trade-off compared to the second one. In the former we can find a particular edge in our data in linear time with respect to the branching factor, while for the latter we need a logarithmic time with respect to the total number of edges. In the next chapter we will see that the average branching factor in data sets is slightly larger than the binary logarithm of the number of nodes. However, the cache access pattern for the first implementation is better (sequential scan as opposed to random access) so this difference may not be insignificant.

[2]On the DAS-4, the job time limit is 120 minutes, but exceptions may be made outside of office hours. For the experiment to be repeated, however, shorter execution times were preferred.

Chapter 3

Experiments

As a platform, we used the DAS-4 and its MPI implementation. We run algorithm with 8 instances per physical node, which is default for the DAS-4. The DAS scheduler is responsible for placing algorithms on nodes. To circumvent jitter in timings results, we repeat each experiment 5 times. We vary the number of worker nodes in our experiment. Due to limitations of the platform, we can either run a multiple of 8 workers or less than eight workers. This is because DAS places 8 workers on a single physical platform.

To test the previously described implementations of the Push Lift algorithm, we run it repeatedly in different configurations on several different data sets. The algorithm is run with 1, 2, 4, 8, 16, 24, 32, ..., 120 (powers of two up to eight, then multiples of 8 up to 120)[1] workers to see whether there is any speed up when running the algorithm in parallel.

3.1 Experiment data

For our experiment data, we used real-world graphs as well as generated graphs. For each graph we chose a specific source and sink for our experiments so that we could easily repeat and average results. An overview of the used graphs and configuration used can be found in Table 3.1. Not all configurations are shown, however.

The real-world sample graphs were obtained from the University of Florida Sparse Matrix Collection [DH11]. Although the collection is mostly unrelated to graphs, some matrices are in fact regular (natural) networks. We looked for well-connected graphs weighted graphs of a size for which max-flow computations take a significant but not too large time to complete. From the collection, we chose two graphs: `vanHeukelem/Cage11`,

[1] Even though the DAS cluster has enough nodes available for 128 workers, there was an outage preventing us from using the entire cluster. Thus, one node remained unused.

Name	Description	From	Nodes	Edges	Source	Sink
cage11	DNA electrophoresis	`vanHeukelum/cage11`	39082	559722	1361	28129
internet	Connectivity of internet routers	`Pajek/internet`	124651	207214	94268	1046
rmat	Scale-free random graph	GTgraph R-MAT	30000	5000000	89872	59366
ssca2	Hierarchically clustered random graph	GTgraph SSCA2	32768	1570139	21264	7066

Table 3.1: An overview of the data sets and configurations used.

a biological network, and `Pajek/Internet`, a network topology of internet routers.

To generate graphs, we looked to the GTgraph [BM06] toolkit. This program can generate graphs using any of four algorithms:

random allows us to specify a number of nodes n and a number of edges m, after which it iteratively picks two random nodes and adds an edge between them.

Erdős-Renyi takes a number of edges n and a probability p of an edge between any pair of nodes existing. Then it iterates over all pairs of nodes and generates an edge with probability p. This is mostly equivalent to the **random** generator when $p = \frac{m}{n \times (n-1)}$.

R-MAT generates graphs that follow a power law for the degree distribution of the nodes. This means that for any degree k, $P(k) \propto k^{-\gamma}$.[2] This is a property commonly encountered in various networks [New05].

SSCA2 generates locally clustered graphs, meaning it creates graphs that are very well connected in small communities with little connections between them.

Because the both the random and Erdős-Renyi algorithms do not produce graphs similar to those observed in real scenarios, we only consider graphs generated by the R-MAT and SSCA2 algorithms.

For each graph, we searched for a combination of source/sink that took a reasonable time to compute with one worker instance, in order to repeat the experiment and also had a non-zero final result so we could actually check the result. The resulting configurations can be found in Table 3.1.

3.2 Expectations

Even with the optimizations described in the previous chapter, the algorithm needs a lot of communication between workers about its current state. Therefore we will only see any real performance gains when the workers have to rarely wait for input and are mostly busy. This is not an ideal scenario.

[2] In directed graphs nodes have two degrees, k_{in} and k_{out} because the number of edges coming in is not necessarily the same as the number of edges going out. The power law distribution then simply applies to both separately.

Graph	# workers	Implementation 1 μ	Implementation 1 σ	Implementation2 μ	Implementation2 σ
cage11	1	2872.746	36.633		
	4	2963.930	45.048		
	8	2264.534	20.725	5091.604	12.941
	16	1887.338	13.634	3540.866	21.292
	64	2834.964	55.413	2850.964	43.792
	120	2651.752	53.096	2689.112	57.354
internet	1	309.461	0.479	1570.532	6.083
	4	283.967	1.980	718.834	12.566
	8	201.618	0.851	482.113	8.827
	16	182.930	0.695	403.690	2.906
	64	167.777	1.619	314.809	2.856
	120	145.779	0.560	242.971	5.694
rmat	1	19.911	0.052	76.587	0.141
	4	14.728	0.100	36.691	0.169
	8	14.793	0.147	31.797	1.293
	16	16.216	0.681	27.546	0.727
	64	18.105	0.542	25.952	0.239
	120	19.138	0.495	26.322	0.427
ssca2	1	97.455	0.732	556.118	6.299
	4	83.914	0.642	244.998	5.372
	8	71.072	0.614	166.594	1.978
	16	82.053	3.751	124.637	1.112
	64	165.327	2.249	160.869	2.705
	120	138.342	1.826	134.720	1.997

Table 3.2: Mean and standard deviation for algorithm run times in several configurations.

Furthermore, since we run the algorithm with 8 workers on a single physical DAS node, the communication latency increases greatly after scaling past 8 workers. We expect to see this gap in the performance as well.

3.3 Results

After running every implementation of a varying amount of MPI workers, we can compare the execution time of each algorithm. For the results, we have looked at algorithm execution time only. An overview of that can be seen in Table 3.2. It should be noted that the standard deviation on the runtime is fairly low compared to the actual runtime. This shows that despite network jitter and other external factors, the run time of the algorithm is rather consistent

Missing entries in the table (and the following graphs) are configurations that failed to complete within the time limit of two hours.

We have decided to leave out the algorithm initialization time (i.e. initializing MPI and reading in the data) as this is unrelated to the problem being solved, and it would only serve to hide potential gains. Furthermore, as shown in Figure 3.1, the cost of initializing does not increase that much with the number of workers, so even at larger scale this should not be a problem.

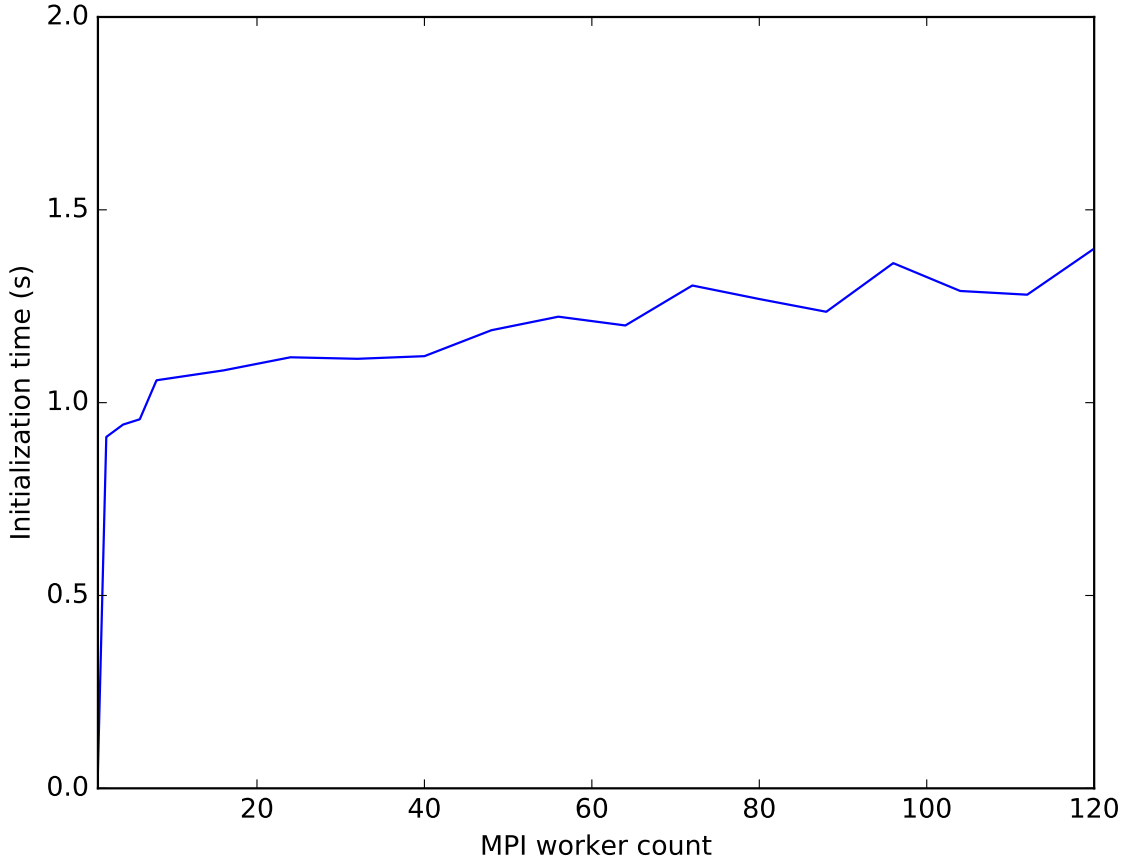

Figure 3.1: Cost of initialization as a function of the number of workers. This graph shows an average over all trial runs in this thesis, not one specific graph.

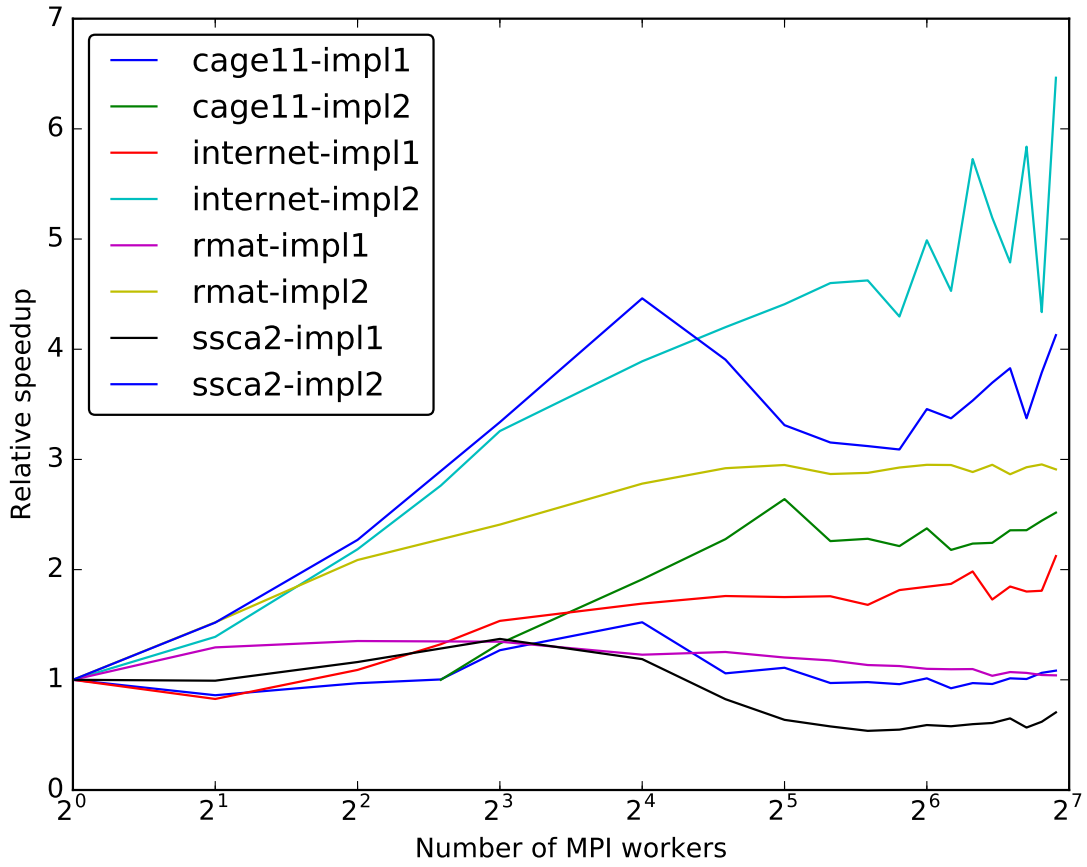

Figure 3.2: Relative speed up when running a specific route on the datasets.

3.3.1 Relative speed up

What we were most interested in, was the possibility of a speed up when running the algorithm in parallel. In Figure 3.2, we see the relative speed up of both working implementations on the various graphs. In this figure, we define the relative speed up as the time it took the algorithm to complete (on average) relative to the time it took that that implementation *with the least number of workers to complete within two hours*. In other words, in the case of `cage11-impl2`, which first succeeded with 8 workers, the speed up is defined relative to that rather than to the result with 1 workers.

The worker-axis is scaled logarithmically to highlight the performance gains within the first physical node, before external communication needs to occur. Most trends indeed seem to flatten at 8 workers, which is to be expected since the communication latency increases at that point.

As expected, the figure shows very little speed up across the board. In particular, implementation 1 hardly shows any performance gains at all while implementation 2 improves 3-5 fold while using 120 cores. This most likely is due to the massive overhead in communication.[3]

[3]In fact, the results produced by Hong et al [Hono8] on a shared memory implementation prove this, as shared memory has a lower

3.3.2 Comparing implementations

Even though Figure 3.2 suggests that implementation 2 scales better when run in parallel than implementation 1, this is hardly fair. As seen in Table 3.2, the run time of the former is always longer than that of the latter.

In Figure 3.3 we show the run times of the different implementations on the same graph, and we can immediately see that, even without any speed up, the first implementation still is faster. The second merely gets closer in performance. This tells us that, for the second implementation, there is significantly more time being spent on computation and not communication. Only when the majority of time is being spent in communication (the networking systems were the same for both) rather than the actual algorithm, the run times start to get similar.

overhead than a message passing implementation such as this one.

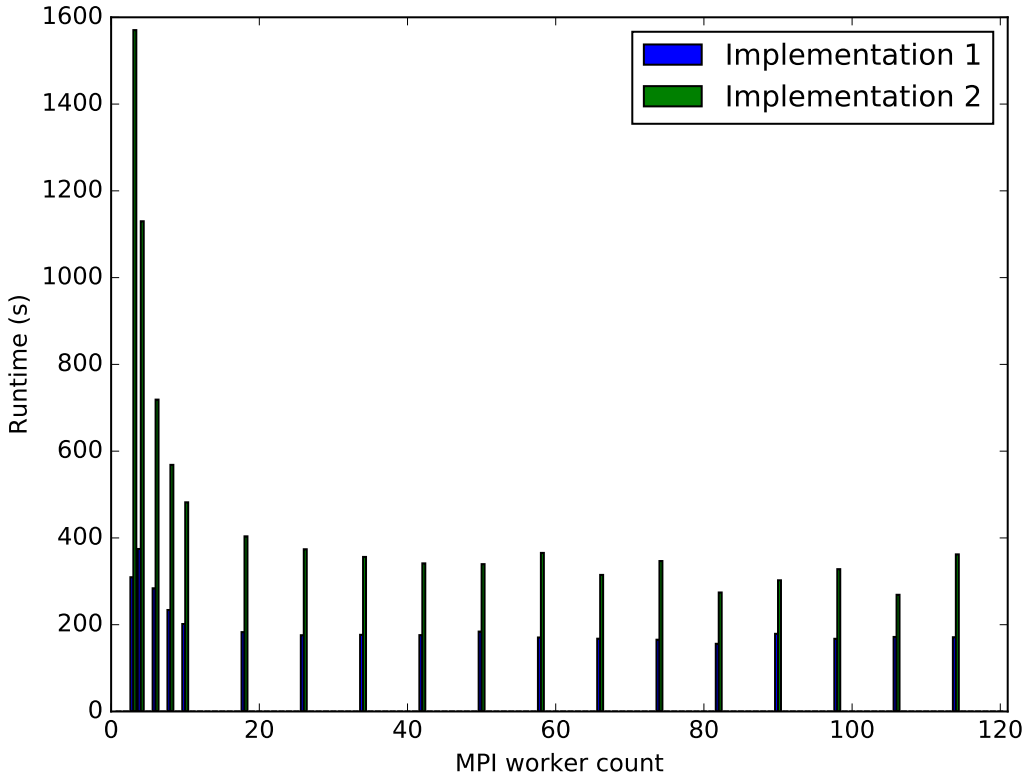

Figure 3.3: Comparison of implementations by execution time on the `internet` data set.

When looking at the complexities as shown in Table 2.1, we can see that the second implementation will only do better if the sizes of our graph $\log |V|$ and $\log |E|$ are significantly less important than the branching factor B. As the average branching factor is the number of edges divided by the number of nodes, this is unlikely, as these are usually of similar order of magnitude.

Additionally, the memory access pattern of trees is less friendly to memory caches than that of arrays, which also does not help in favour of the second implementation.

Chapter 4

Conclusion

In this thesis, we have looked at the max-flow min-cut problem and several algorithms that solve it. We have then taken the Push-Lift algorithm and implemented it in the *forelem* framework. We have then derived several parallel implementations and tested them against various graphs.

In our experiments, we have seen that — even though one implementation is significantly better than the other — both implementations do not scale well when run in parallel, even when using larger numbers of workers. This can be attributed the poor locality of the algorithm, requiring a lot of communication to run.

As communication appears to be the bottleneck, it is also the first site to improve. When splitting work between workers, nodes have been divided by their node number. If a more local subdivision could be made, (for instance, by using graph clustering detection techniques), less communication between workers would be necessary, greatly reducing the overhead. Workers only need to communicate between each other when information changes on the boundary between them. If those boundaries were minimized, so would the needed communication.

The communication layer itself could be reworked to be more efficient. Currently, it each *push* and *lift* operation is transferred separately, incurring MPI overhead for every operation communicated. Combining the results and possibly sending them asynchronously could also improve performance.

Most importantly, more work can be done in automating the translation from *forelem* to code. In this thesis, we made an approximation by hand, but an automated translation could create more equivalent implementations and try more variations. The tested implementations varied only in underlying data structure, but data distribution could also easily adapted. This would also allow the lessons learned here to be applied to other problems.

Appendix A

forelem implementation for Push-Lift

The following code listing show the forelem specification for Push-Lift used in the experiment of this thesis.

The global variables assumed to be present are:

- *source* the source node
- *sink* the sink node
- V, E the sets of tuples described above.

After the algorithm complettes, `flowOut` contains the maximum flow from `source` to `sink`.

```
1  // Initialization
2  flowOut = 0;
3
4  // Initialize each height as 0
5  forelem (i; i ∈ pV) {
6      V[i].h = 0;
7  }
8
9  V[source].h = |V|;
10
11 // Perform the initial push
12 forelem (i; i ∈ pE.u[source]) {
13     V[e[i].v].e += e[i].c;
14     e[i].f = e.[i].c;
15 }
```

```
16
17 // While there still is an element with excess
18 whilelem (i ∈ pV : V[i].e >0) {
19     if (V[i].u == sink) {
20         flowOut += V[i].e;
21         V[i].e = o;
22         continue;
23     }
24
25     // Determine the smallest pushable edge
26     edge = null;
27     minHeight = V[i].h;
28
29     // Consider original edges
30     forelem (j ∈ pE.u[V[i].u]) {
31         if (E[j].f < E[j].c && V[E[j].v].h < minHeight) {
32             edge = j;
33             minHeight = V[E[j].v].h;
34         }
35     }
36
37     // Consider backtracking
38     forelem (j in ∈ pE.v[V[i].u]) {
39         if (E[j].f > o && V[E[j].v].h < minHeight) {
40             edge = j;
41             minHeight = V[E[j].u].h;
42         }
43     }
44
45     if (edge != null) {
46         // Perform a push
47         if (E[edge].u == V[i].u) {
48             delta = min(V[i].e, E[edge].c − E[edge].f)
49             E[edge].f += delta
50         } else {
```

```
51              delta = min(V[i].e, E[edge].f)
52              E[edge].f -= delta
53          }
54          V[i].e -= delta;
55      } else {
56          // Need to lift, find a new height.
57          newHeight = ∞;
58          forelem (j in ∈ pE.v[V[i].u]) {
59              if (E[j].f > 0) {
60                  newHeight = min(newHeight, V[E[j].v].h)
61              }
62          }
63          forelem (j in ∈ pE.u[V[i].u]) {
64              if (E[j].f < E[j].c) {
65                  newHeight = min(newHeight, V[E[j].u].h)
66              }
67          }
68
69          V[i].h = newHeight + 1;
70      }
71  }
```

Bibliography

[BM06] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[DH11] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011.

[FF56] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[Fok13] W. Fokkink. *Distributed Algorithms: An Intuitive Approach*, chapter 6, pages 42–44. MIT Press, 2013.

[GT86] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, 1986.

[Hon08] Bo Hong. A lock-free multi-threaded algorithm for the maximum flow problem. In *Parallel and Distributed Processing*, 2008.

[HR55] T.E. Harris and F.S. Ross. Fundamentals of a method for evaluating rail net capacities. 1955.

[New05] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[Rie14] K.F.D. Rietveld. *A Versatile Tuple-Based Optimization Framework*. PhD thesis, Leiden University, 2014.