# PARALLEL MAXIMUM FLOW SOLVER FOR MULTI-CORE MACHINES

by

Selçuk Cihan

B.S. in Computer Engineering, Boğaziçi University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2010

PARALLEL MAXIMUM FLOW SOLVER FOR MULTI-CORE MACHINES

APPROVED BY:

| | |
|---|---|
| Assoc. Prof. Can Özturan<br>(Thesis Supervisor) | ..... |
| Assoc. Prof. Haluk Bingöl | ..... |
| Prof. Haluk Rahmi Topçuoğlu | ..... |

DATE OF APPROVAL: 23.09.2010

# ACKNOWLEDGEMENTS

Selçuk Cihan is supported by TUBITAK BIDEB 2228 (followed by 2210) scholarship grant.

I would like to thank my advisor, Assoc. Prof. Can Özturan, for pointing me in the right direction and sharing his valuable opinion during the course of this study. I am also grateful to Ali Haydar Özer, providing me his LaTeX sources which assisted me in formatting this thesis.

# ABSTRACT

## PARALLEL MAXIMUM FLOW SOLVER FOR MULTI-CORE MACHINES

We provide a parallel algorithm for calculating maximum flow between two nodes, in a capacitated network. The algorithm we propose is based on push-relabel algorithm due to Goldberg and uses a modified first in first out selection strategy together with global relabeling heuristic. Our implementation targets multi-core processors, implements task stealing to balance load between multiple threads of execution and uses fast atomic variables for synchronization instead of costly general purpose locks. We compare our algorithm to other push-relabel based algorithms and demonstrate that it performs well in practice.

# ÖZET

## ÇOK ÇEKİRDEKLİ MİMARİLER İÇİN PARALEL EN BÜYÜK AKIŞ ÇÖZÜCÜ

Sığalı ağlarda, iki düğüm arasındaki en büyük akışı hesaplayacak, paralel bir algoritma sunuyoruz. Algoritmamız, Goldberg'in itele-tekrar-etiketle (push-relabel) algoritması temel alınarak geliştirilmiştir. Etkin düğüm seçimi için, paralel yürütülmeye uygun, değiştirilmiş bir "ilk giren ilk çıkar" (FIFO) yöntemi kullanılmaktadır. Algoritmamız, pratikte oldukça hız kazandıran global-tekrar-etiketleme (global relabeling) buluşsal (heuristic) yöntemini kullanmaktadır. Çok çekirdekli işlemcileri hedefleyen algoritmamız, görev çalma yöntemi ile, iş yükünü farklı iş parçacıklarına dağıtmaktadır. Çekirdeklerin ortak kullandığı belleğe erişimin senkronizasyonu için, hesaplama açısından pahalı genel amaçlı kilitler yerine, hızlı atomik (atomic) değişkenler kullanılmıştır. Algoritmamızı itele-tekrar-etiketle tabanlı seri algoritmalar ile karşılaştırdık ve başarımının iyi olduğunu gösterdik.

# TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . iii
ABSTRACT . . . . . iv
ÖZET . . . . . v
LIST OF FIGURES . . . . . viii
LIST OF TABLES . . . . . x
LIST OF SYMBOLS/ABBREVIATIONS . . . . . xii
1. INTRODUCTION . . . . . 1
1.1. Relation To Other Problems . . . . . 1
1.2. Solution Methods . . . . . 1
1.3. Motivation . . . . . 2
1.4. Problem Formulation . . . . . 3
2. GENERIC PUSH-RELABEL ALGORITHM . . . . . 4
2.1. Distance Labels . . . . . 4
2.2. Algorithm . . . . . 4
2.3. Example . . . . . 7
2.4. Improvements . . . . . 10
2.4.1. Active Node Selection Strategies . . . . . 10
2.4.2. Heuristic Methods . . . . . 10
3. RELATED WORK . . . . . 13
4. PARALLEL IMPLEMENTATION . . . . . 15
4.1. Intel's Threading Building Blocks . . . . . 16
4.2. POSIX Threads . . . . . 16
4.3. Synchronization . . . . . 16
4.4. Overview . . . . . 17
4.5. Data Structures & Naming Conventions . . . . . 18
4.6. Global Relabeling Implementation . . . . . 18
4.6.1. Basic Serial Algorithm . . . . . 19
4.6.2. Parallel Implementation . . . . . 22
4.6.3. Parallel Shared Queue Implementation . . . . . 24

4.7. Parallel Algorithm . . . . . 29
5. RESULTS . . . . . 34
5.1. Test Environment . . . . . 34
5.2. Evaluation On Synthetic Data . . . . . 35
5.2.1. Performance of Parallel Global Relabeling . . . . . 35
5.2.2. Overall Performance . . . . . 38
5.3. Evaluation On Vision Data . . . . . 41
5.3.1. Performance of Parallel Global Relabeling . . . . . 41
5.3.2. Overall Performance . . . . . 42
5.4. Comments On Parallel Implementation Efficiency . . . . . 44
6. CONCLUSION . . . . . 46
APPENDIX A: GENERATOR PARAMETERS FOR SYNTHETIC DATA . . 48
APPENDIX B: VISION DOMAIN GRAPH INSTANCES . . . . . 49
APPENDIX C: TEST RESULTS ON SYNTHETIC DATA . . . . . 50
APPENDIX D: TEST RESULTS ON VISION DATA . . . . . 53
REFERENCES . . . . . 57

# LIST OF FIGURES

Figure 2.1. Pseudo-code for push method . . . . . 5

Figure 2.2. Pseudo-code for relabel method . . . . . 5

Figure 2.3. Pseudo-code for generic push-relabel algorithm . . . . . 6

Figure 2.4. Example input and augmenting path operations . . . . . 8

Figure 2.5. Push-relabel algorithm example . . . . . 9

Figure 4.1. Data structures for Node and Arc . . . . . 18

Figure 4.2. Global relabeling heuristic example . . . . . 20

Figure 4.3. Pseudo-code for serial global relabeling . . . . . 21

Figure 4.4. Pseudo-code for parallel breadth-first traversal . . . . . 23

Figure 4.5. Data structures for parallel queue . . . . . 25

Figure 4.6. Pseudo-code for parallel_queue_pop . . . . . 26

Figure 4.7. Pseudo-code for parallel_queue_push . . . . . 27

Figure 4.8. Parallel pushing by two threads . . . . . 30

Figure 4.9. Pseudo-code for parallel maximum flow solver . . . . . 31

Figure 4.10. Pseudo-code for fprf_push . . . . . 32

Figure 4.11. Pseudo-code for fprf_relabel . . . . . 33

Figure 5.1. Speed-up of parallel global relabeling on synthetic graphs . . . . . 37

Figure 5.2. Speed-up on synthetic graphs . . . . . 39

Figure 5.3. Speed-up of parallel global relabeling on vision graphs . . . . . 41

Figure 5.4. Speed-up on vision graphs . . . . . 43

# LIST OF TABLES

Table 2.1. Push-relabel based implementations . . . . . 11

Table A.1. Parameters for genrmf-wide and genrmf-long types. . . . . 48

Table A.2. Parameters for washington-rlg-wide, washington-rlg-long and washington-line-moderate types. . . . . 48

Table A.3. Parameters for ac-dense type of graphs. . . . . 48

Table B.1. Number of nodes & number of arcs for vision instances. . . . . 49

Table C.1. Comparison on genrmf-wide family. Times in seconds. . . . . 50

Table C.2. Comparison on washington-rlg-wide family. Times in seconds. . . . 50

Table C.3. Comparison on ac-dense family. Times in seconds. . . . . 51

Table C.4. Comparison on genrmf-long family. Times in seconds. . . . . 51

Table C.5. Comparison on washington-rlg-long family. Times in seconds. . . . 52

Table C.6. Comparison on washington-line-moderate family. Times in seconds. 52

Table D.1. Comparison on BL06-gargoyle-lrg instance. Times in seconds. . . . 53

Table D.2. Comparison on BL06-gargoyle-med instance. Times in seconds. . . 53

Table D.3. Comparison on LB07-bunny-med instance. Times in seconds. . . . 54

Table D.4. Comparison on babyface.n6c100 instance. Times in seconds. . . . . 54

Table D.5. Comparison on babyface.n6c10 instance. Times in seconds. . . . . 55

Table D.6. Comparison on BL06-camel-lrg instance. Times in seconds. . . . . 55

Table D.7. Comparison on BL06-camel-med instance. Times in seconds. . . . 56

# LIST OF SYMBOLS/ABBREVIATIONS

| | |
|---|---|
| a | Length of a square grid, first parameter of genrmf generator |
| $A(v)$ | Set of arcs pointing out from node $v$ |
| b | Number of frames in the network, second parameter of genrmf generator |
| c1 | Maximum capacity for intra-frame arcs, third parameter of genrmf generator |
| c2 | Maximum capacity for inter-frame arcs, fourth parameter of genrmf generator |
| $c(<v, w>)$ | Capacity of arc $<v, w>$ |
| dim1 | Number of rows in the rectangular grid, second parameter for washington generator |
| dim2 | Number of columns in the rectangular grid, third parameter for washington generator |
| $d(v)$ | Distance label of node $v$ |
| $e(v)$ | Excess on node $v$ |
| fct | Specifies the graph type, first parameter for washington generator |
| $f(<v, w>)$ | Flow on arc $<v, w>$ |
| $m$ | Number of arcs |
| $n$ | Number of nodes |
| range | Maximum allowed capacity on arcs, fourth parameter for washington generator |
| $r(<v, w>)$ | Residual capacity of arc $<v, w>$ |
| $s$ | Source node |
| $t$ | Sink node |
| $<v, w>$ | Arc from $v$ to $w$ |

| | |
|---|---|
| API | Application Programming Interface |
| FIFO | First In First Out selection strategy |
| F_PRF | FIFO implementation with global relabeling |

| | |
|---|---|
| HI_PR* | Improved HL implementations with global relabeling and gap relabeling |
| HL | Highest Label selection strategy |
| H_PRF | HL implementation with global relabeling and gap relabeling |
| M_PRF | HL implementation with global relabeling |
| PRAM | Parallel Random Access Machine |
| Q_PRF | FIFO implementation with global relabeling and gap relabeling |

# 1. INTRODUCTION

Maximum flow problem is, finding a flow with the maximum amount, in a network with capacitated arcs, from a node designated as source to a node designated as sink. This problem, which can be formulated as a linear program, can be solved in polynomial time. We will present a parallel solver for the problem. First, we provide information about the problem. Then sequential algorithms and implementations are discussed. In Chapter 3, we discuss related work and parallel implementations. Our contribution is detailed in Chapter 4. At the end, we provide results of our experiments, discuss advantages/disadvantages of our implementation and conclude with future ideas.

## 1.1. Relation To Other Problems

Maximum flow problem is encountered in a variety of engineering fields such as network planning, and also as subproblems of other problems like minimum cost flow problem. Some of the applications related to maximum flow problem are scheduling problems, matrix rounding, bipartite matching and network connectivity problems. See [1, p. 169] for a comprehensive list of applications of the problem.

Maximum flow problem, is closely related to the problem of finding minimum cut in a network. *Max-flow min-cut theorem* establishes this relation. Amount of maximum s-t flow in a capacitated network equals minimum of capacity of cuts between s-t. A proof of this relation can be found in [1, p. 184]. With this duality, maximum flow problem can also be employed in applications of min-cut problem.

## 1.2. Solution Methods

Maximum flow problem has been studied extensively and there are various solution approaches. Some of these approaches are: labeling (Ford and Fulkerson), capacity scaling, shortest augmenting path, blocking flow, network simplex method and push-relabel [2] method. With the exception of network-simplex method, maximum flow

algorithms can be categorized into two: those that augment flow along paths (such as the labeling algorithm) and those that augment flow along arcs (such as the generic push-relabel algorithm). The latter type of algorithms are proved to have better worst-case complexity; although practical performances vary and heuristics are employed to obtain better practical performances. [1, p. 240] presents an extensive study of various maximum flow algorithms, together with running times and references.

Some of these methods have polynomial worst-case complexity, whereas others have pseudo-polynomial worst-case complexity. Shortest augmenting path algorithm of Edmonds and Karp [3] is a polynomial time algorithm with $O(nm^2)$ worst case complexity. The algorithm is a version of augmenting path algorithms and augments flow along shortest paths. Capacity scaling algorithm, due to Ahuja and Orlin [1, p. 210], runs in $O(nmlogU)$ time where $U = max(c(<v, w>))$. Another algorithm for maximum flow problem is push-relabel [4] algorithm, that we based our parallel implementation on. It yields practical implementations and has a worst case complexity of $O(n^2m)$.

### 1.3. Motivation

Maximum flow problem finds extensive use in many different algorithms. Any improvement in solving the maximum flow problem would result in practical improvements in various other algorithms that need to solve the maximum flow problem. With the advances in multi-core hardware and supporting software, we believe the maximum flow problem would benefit from parallel implementations. However, devising a parallel solver for the maximum flow problem, which performs well in practice on different types of graphs, is not straightforward. Some problem instances lack parallelism and on some problems, synchronization overhead (locking of nodes, arcs) may result in performance degradation [5]. We present a parallel implementation that uses fast atomic variables instead of costly general purpose locks.

Another point of interest is that best sequential implementations of maximum flow solvers evolved over time. There are useful heuristics that improve performance

drastically, and some of the previous work on parallel maximum flow solvers did not know about these heuristics.

### 1.4. Problem Formulation

The input to maximum flow problem is a capacitated network (a directed graph with capacities assigned to each arc) $G = (V, A)$, where $V$ is the set of nodes and $A$ is the set of arcs. We denote $|V|$ as $n$ and $|A|$ as $m$. An element $<v, w> \in A$ denotes an arc from node $v$ to node $w$. There are two specially designated nodes named *source* and *sink*, denoted by $s$ and $t$ respectively. Each arc $<v, w>$ has a nonnegative capacity assigned to it via a real-valued function $c : A \rightarrow \mathbb{R}_0^+$, where $c(<v, w>) = c_{v,w}$ and $c_{v,w} \geq 0$.

The problem is to calculate a flow from $s$ to $t$ which has maximum amount amongst all other $s$-$t$ flows. A flow from $v$ to $w$ is defined as a real-valued function $f : A \rightarrow \mathbb{R}$, where $f(<v, w>) = f_{v,w}$. A flow should honor the capacity constraints, $f_{v,w} \leq c_{v,w} \forall <v, w> \in A$. Formally, the problem can be formulated as a linear program as follows.

$$\text{maximize } F$$

$$\text{subject to}$$

$$\sum_{w:<v,w>\in A} f_{v,w} - \sum_{w:<w,v>\in A} f_{w,v} = \begin{cases} F & \text{for } v = s, \\ 0 & \forall v \in V \setminus \{s, t\}, \\ -F & \text{for } v = t \end{cases} \tag{1.1}$$

$$0 \leq f_{v,w} \leq c_{v,w} \forall <v, w> \in A \tag{1.2}$$

where $f_{v,w}$ are the decision variables. Equation 1.1 establishes mass balance constraints. Equation 1.2 gives capacity constraints. A function $f(<v, w>) = f_{v,w}$ is called a *flow*, if equations 1.1 & 1.2 are satisfied. Scalar value $F$ corresponding to a flow, is the value of that flow.

# 2. GENERIC PUSH-RELABEL ALGORITHM

Generic push-relabel algorithm, due to Goldberg [4], is the basis of many practical implementations. The algorithm is generic, in that basic operations of the algorithm can be applied in any order. That is why, there are many derivatives of the algorithm and the algorithm was a basis for our parallel implementation also. As the algorithm proceeds, capacities on arcs are updated such that they are equal to the remaining capacities, to reflect the current flow on each arc. The graph with updated capacities is then called a *residual graph*.

## 2.1. Distance Labels

A major improvement in theoretical performance of maximum flow algorithms was achieved, as distance labels were introduced by Goldberg [4] in their push-relabel algorithm. Distance labels can be considered as counterpart to layered networks used in previous algorithms. Distance labels are assigned to each node and are used to ensure that flow is always pushed along an arc which is a part of the shortest path to sink. The sink is assigned a distance label of zero. A node which is $d$ hops from the sink would have a distance label of $d$. We denote the distance label of a node $v$ as $d(v)$. Distance labels are updated to reflect the changes in the residual graph as the algorithm executes, thus shortest paths are also updated.

## 2.2. Algorithm

Push-relabel algorithm pushes flow along arcs, as opposed to the augmenting path algorithms which push flow along paths. At intermediate steps of push-relabel algorithm, there are intermediate nodes for which incoming flow and outgoing flow are not equal. This intermediate flow is called a *preflow*. If incoming flow and outgoing flow are not equal on a node, the node is said to have *excess* on it.

Push-relabel algorithm pushes excess from nodes towards the sink in a local

fashion, that is along arcs that are part of the shortest path to sink in the residual graph, until the sink is not reachable in the residual graph. At that point, the excess that cannot be pushed to the sink is returned back to the source to convert the preflow to an actual flow. A detailed description of generic push-relabel algorithm can be found in [2] and [4].

There are two basic operations called push and relabel. The algorithm performs these basic operations in any order until none of them are applicable, and algorithm terminates finding the maximum flow. There are two definitions that are used in the algorithm. First, we call a node *active* if its distance label is finite and it has excess. Source and sink are never active. Second, we call an arc $<v, w>$ *admissible* if it has residual capacity and $d(v) = d(w) + 1$. We give a pseudo-code of the generic algorithm in Figure 2.3, with the basic operations of push and relabel in Figures 2.1 and 2.2 respectively.

**Method push($<v, w>$)**

**Require:** Arc $<v, w>$

**Ensure:** Maximum possible amount of flow is pushed on arc $<v, w>$

$value \leftarrow min(e(v), r(<v, w>))$

$e(v) \leftarrow e(v) - value$

$e(w) \leftarrow e(w) + value$

$r(<v, w>) \leftarrow r(<v, w>) - value$

$r(<w, v>) \leftarrow r(<w, v>) + value$

Figure 2.1. Pseudo-code for push method

**Method relabel($v$)**

**Require:** Node $v$

$d(v) \leftarrow min(\{d(i) + 1 : <v, i> \in A(v) \wedge r(<v, i>) > 0\} \cup \{\infty\})$

Figure 2.2. Pseudo-code for relabel method

**Generic push-relabel algorithm**

**Require:** Directed graph $G(V, A)$, source node $s$ and sink node $t$ and arc capacities $c(<v, w>)$ for each $<v, w> \in A$

**Ensure:** Maximum flow from $s$ to $t$

**for all** $v \in V$ **do**
  $d(v) \leftarrow 0$
  $e(v) \leftarrow 0$
**end for**
$d(s) \leftarrow n$
**for all** $<v, w> \in A$ **do**
  $r(<v, w>) \leftarrow c(<v, w>)$
**end for**
**for all** $<s, v> \in A(s)$ **do**
  $r(<s, v>) \leftarrow 0$
  $r(<v, s>) \leftarrow c(<s, v>)$
  $e(v) \leftarrow c(<s, v>)$
**end for**
**while** there is an active node $v$ in the network **do**
  **if** there is an admissible arc $<v, w> \in A(v)$ **then**
    **call** $push(<v, w>)$
  **else**
    **call** $relabel(v)$
  **end if**
**end while**

Figure 2.3. Pseudo-code for generic push-relabel algorithm

## 2.3. Example

In order to better understand the generic push-relabel algorithm, we present an example execution of the algorithm. For the example, we have a directed graph with five nodes as shown in Figure 2.4a. It is easy to immediately see that the maximum amount of flow that can be sent from $s$ to $t$ in this small graph is four units.

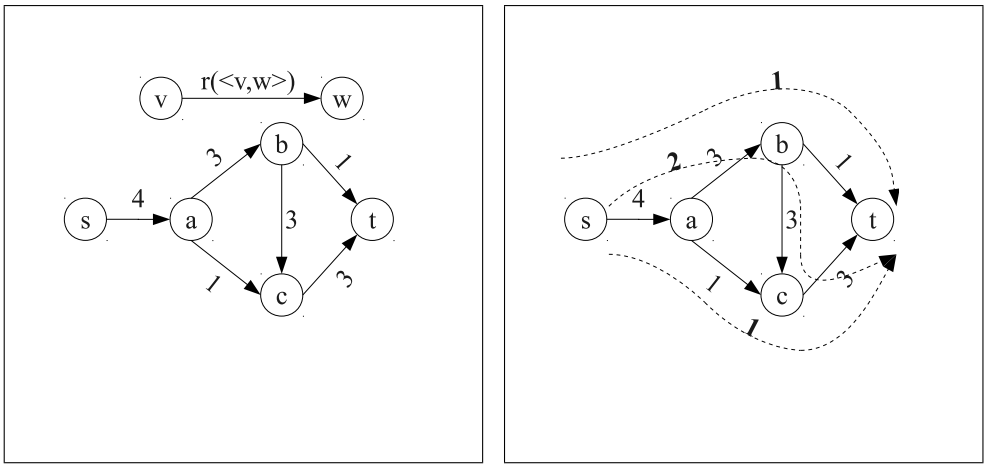
Firstly, in Figure 2.4b, we show the solution steps of the augmenting path algorithm so as to better present the advantages of push-relabel based algorithms against augmenting path algorithms. One unit of flow is augmented along $s-a-b-t$, one unit of flow is augmented along $s-a-c-t$ and two units of flow are augmented along $s-a-b-c-t$. Notice that it takes a total of ten arc traversals to send flow along the three augmenting paths. Furthermore, we see that $<s,a>$ is shared by the three augmenting paths and hence this arc is traversed three times.

Generic push-relabel algorithm example is in Figure 2.5. In Figure 2.5a, initial distance labels are assigned as shortest path distances to $t$. Distance label of $s$ is by default equal to the number of nodes in the graph, five in this case. Initially in Figure 2.5b, a flow of four units is pushed from $s$ to $a$, to saturate $<s,a>$. Node $a$ is the only active node at this point, with an excess of four units; basic operations will be applied to $a$. Both $<a,b>$ and $<a,c>$ are saturated by pushing three and one unit of flow respectively, in Figure 2.5c. Nodes $b$ and $c$ become active and node $a$ is not active anymore. Then, push operation is applied on $<b,t>$ and $<c,t>$ in Figure 2.5d. At this point, the only active node is $b$ and there are two arcs $<b,c>$ and $<b,a>$ emanating from $b$. However, flow cannot be pushed on these arcs since $d(c)=d(b)$ and $d(a)>d(b)$, the arcs are not admissible. Therefore, a relabeling operation is applied on node $b$, relabeling it to have a distance label of two. Flow can be pushed along $<b,c>$ now, as depicted in Figure 2.5e, since $d(c)+1=d(b)$. In the last iteration, in Figure 2.5f, excess on $c$ is pushed towards $t$. There are no active nodes and the algorithm terminates with an excess of four units on $t$, that is the amount of maximum flow.

The generic push-relabel algorithms example traversed a total of seven arcs in the

push operations. Even in this small example, we can see that push-relabel algorithm performs less arc traversals compared to augmenting path algorithm. Push-relabel based algorithms are more efficient than augmenting path based algorithms in general. The main reason is that, push-relabel method works locally on each arc, whereas augmenting path method finds paths from source to sink and sends flow along these paths. Obviously, arcs which are shared by many augmenting paths are traversed more than once and this results in more arc traversals.

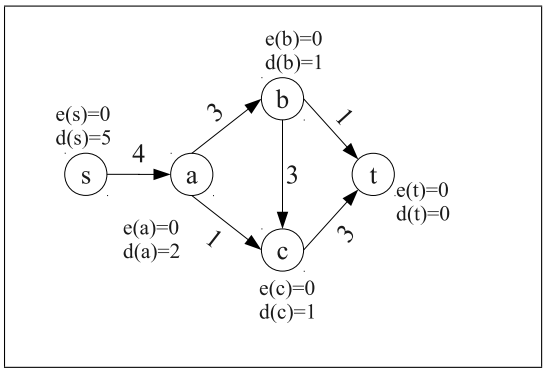
Another point of interest in comparing push-relabel method against augmenting path method is that push-relabel method yields more parallelism than augmenting path method. In the augmenting path example, since $<s, a>$ is shared by the three augmenting paths, these paths cannot be used to push flow concurrently. Flow has to be augmented sequentially on these paths. However, in push-relabel example, the two push operations of Figure 2.5c can be applied in parallel as well as the two push operations of Figure 2.5d.



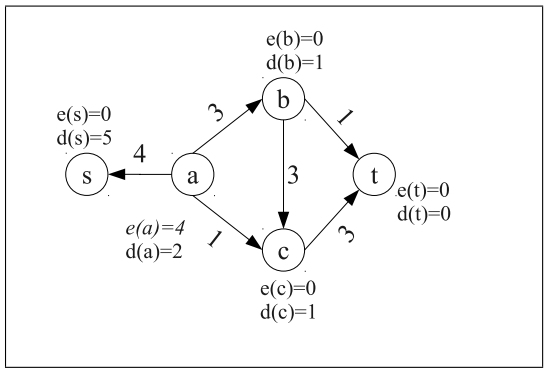

(a) Example input graph

(b) Augmenting along $s-a-b-t$, $s-a-c-t$ and $s-a-b-c-t$

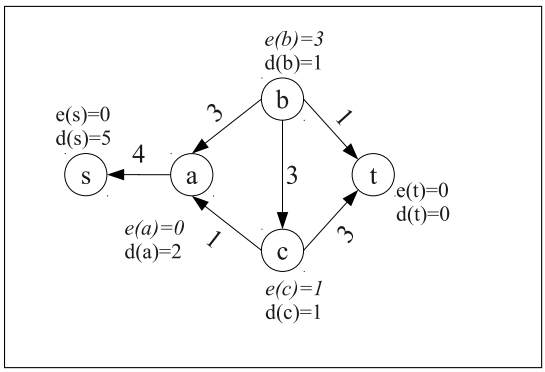
Figure 2.4. Example input and augmenting path operations



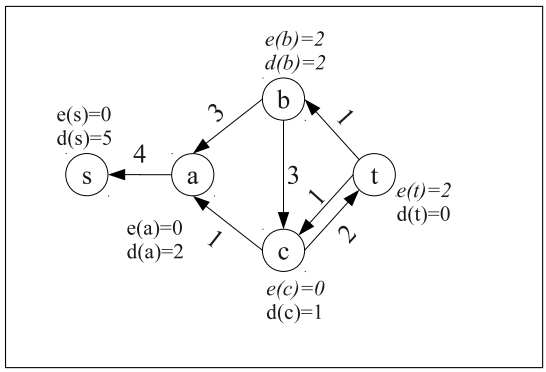

(a) Initial labelings computed



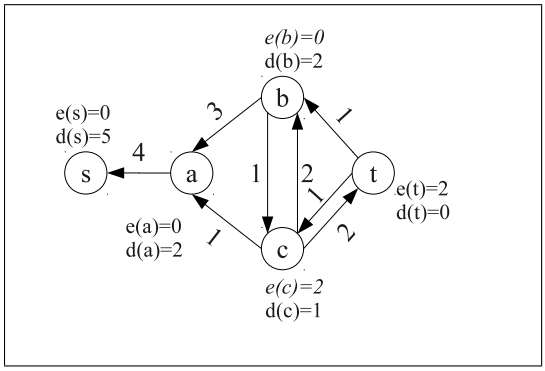

(b) Push on $<s, a>$



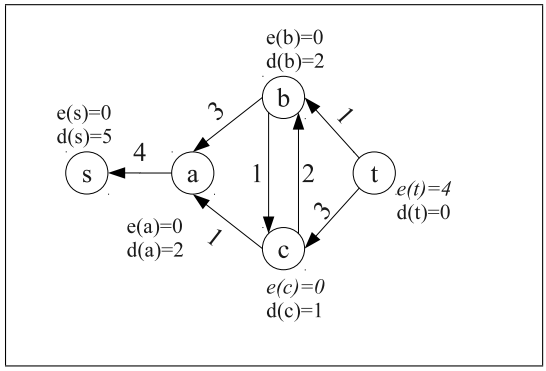

(c) Push on $<a, b>$ and $<a, c>$




(d) Push on $<b, t>$, $<c, t>$ and relabel $b$




(e) Push on $<b, c>$




(f) Push on $<c, t>$

Figure 2.5. Push-relabel algorithm example

## 2.4. Improvements

Generic push-relabel algorithm runs in $O(n^2m)$ time. However, generic implementation of push-relabel algorithm does not perform well in practice. Improvements in practical performance can be achieved by using different active node selection methods and heuristics.

### 2.4.1. Active Node Selection Strategies

One practical improvement can be made by using a selection strategy for choosing which node to push flow from. Two such strategies, FIFO selection and HL selection, are presented in [6]. With FIFO selection strategy, a global queue is maintained and nodes that become active are pushed to the tail of the queue. When the algorithm needs an active node to process, it is always retrieved from the head of the queue. FIFO selection strategy yields a $O(n^3)$ time algorithm. HL selection strategy ensures that the algorithm always proceeds with the active node that has the highest distance label. Worst case complexity of HL selection strategy is shown to be $O(n^2\sqrt{m})$ [7]. Notice that these theoretical bounds are better than previous augmenting path algorithms for dense graphs. Practically, most of the time, HL selection strategy performs better than FIFO selection [8].

### 2.4.2. Heuristic Methods

Yet another improvement in practical behaviour of push-relabel algorithms can be achieved by using heuristics. Two such heuristics, global relabeling and gap relabeling, are treated in [8].

As described in [8], global relabeling heuristic is a very useful addition to both FIFO and HL selection strategy. After each $n$ relabeling operations, all distance labels are computed from scratch. The best labeling is calculating shortest paths from each node to the sink. This can be done in linear time with a backwards breadth-first search on the reverse of residual graph, starting from sink. Since global relabeling is

an expensive operation compared to the basic operations of push and relabel, global relabelings are performed periodically.

Gap relabeling heuristic comes in handy for HL selection strategy. The idea is that at certain points in the execution of the algorithm, there may be an integer $g$, $0 \leq g \leq n$, such that there are no nodes with distance label $g$, but there are nodes with distance label greater than $g$. Distance labels of such nodes can be increased to $n$, since the sink is not reachable from them in the residual graph. This reasoning can be followed from the fact that, in order for the nodes with distance label $d$, where $g < d$, to be connected to the sink, there has to be at least one node for each distance label $l$ where $0 < l \leq g$. And there is no node with distance label $g$.

Table 2.1 presents various implementations [8] of push-relabel algorithm. Among these implementations, F_PRF and Q_PRF both use FIFO selection strategy and global relabeling heuristic, with Q_PRF employing gap relabeling heuristic additionally. On the other hand, M_PRF and H_PRF use HL selection strategy, with H_PRF incorporating gap relabeling heuristic in addition to global relabeling heuristic used by both. Goldberg's code for F_PRF, H_PRF and others can be found in [9].

Table 2.1. Push-relabel based implementations

| | global relabeling | gap relabeling | selection |
|---|---|---|---|
| F_PRF | yes | no | FIFO |
| Q_PRF | yes | yes | FIFO |
| M_PRF | yes | no | HL |
| H_PRF | yes | yes | HL |

Among these implementations, F_PRF and H_PRF are the promising ones. That is because, incorporating gap relabeling heuristic in F_PRF does not result in significant gain (hence Q_PRF is not interesting) and H_PRF's performance degrades if gap relabeling is not employed (that is, M_PRF implementation is not compelling) [8].

Although H_PRF is claimed to perform better than F_PRF, our experiments found instances of graphs on which F_PRF performs as good as H_PRF or even better.

# 3. RELATED WORK

Various parallel implementations and algorithms have been proposed in the literature. Some of these implementations are targeting distributed memory architectures, such as the one in [2]. However, we are interested in the shared memory parallel implementations. We noticed that, the previous work on parallel implementations of push-relabel algorithm either dates back to previous decades where multi-core architectures were not as advanced or that the performance evaluation of such work lacked comparisons regarding actual speed-up (against the best known sequential implementations). In this chapter, we discuss some previous studies that are relevant to our work.

A parallel implementation on PRAM model is discussed in [2]. Their algorithm works in pulses. A pulse is a three stage execution in which each stage is performed in parallel. In the first stage, flow is pushed and residual graph is modified, however flow values are not added as excess to nodes that received flow until the last stage. In the second stage, relabeling is performed. Finally, in the last stage, flow pushed in stage one is added as excess to nodes that received flow. Parallelism is achieved by running pulse for each active node in parallel, that is why a queue is not needed to buffer active nodes. This algorithm has a theoretical time complexity of $O(n^2 logn)$ with $O(n)$ processors. This work only discusses the theoretical complexity and do not measure practical performance. However, parallel implementations in general suffer from issues such as the drawbacks of hardware (PRAM architecture is only theoretical).

Another shared memory parallel implementation can be found in [5]. They provide a way to concurrently perform global relabeling heuristic. Queue of active nodes is divided into two parts, a shared queue and local queues for each processor. Local queue is divided into two as an in-queue and an out-queue. Each processor takes nodes to process, from their in-queues and outputs newly active nodes to their out-queues. If in-queue of a processor is empty, the processor fetches a batch of vertices from the shared queue. If out-queue of a processor is full, the queue is flushed to the shared

queue. Global relabeling is handled by making each processor collaborate. Our work is related to this algorithm; both use FIFO active node selection strategy and both employ global relabeling heuristic. This algorithm uses locks; on the other hand, our algorithm was designed to minimize the high cost of locking by using atomic variables. The significant innovation in our work is the use of atomic variables. Furthermore, we do not use a global queue of active nodes, we only use local queues in each thread and implement task-stealing to balance load. Finally, our algorithm does not concurrently perform global relabeling with the basic operations, so as to reduce synchronization overhead stemming from the need for extensive locking of nodes and arcs.

Another point of interest is that multi-core architectures have evolved. The work in [5] was experimented on Sequent Symmetry S81 with 20 Intel 16Mhz 80386 processors, where each CPU has 64 kilobytes of cache. The system ran on DYNIX 3.0, with 32 megabytes of memory. We, on the other hand, conducted our experiments on 8 cores (two off-the-shelf quad-core CPUs), with 16 gigabytes of memory. The advance in computers, of course, means that we can test our implementation on bigger graphs. Our test cases require hundreds of millions of push-relabel operations to take place. Also, better sequential implementations were developed since this work.

A parallel implementation of push relabel algorithm with gap relabeling heuristic is presented in [10]. Their implementation is experimented on Sun E4500 which is a parallel machine with 14 UltraSPARC II 400MHz processors and 14 gigabytes of memory. A modified HL selection strategy is used in their work. Locking is used to provide synchronization. Results and discussion in this work are about the impact of cache miss and locality on the running times. The implementation is not compared to the best known sequential implementations, only relative speed-up results are provided.

# 4. PARALLEL IMPLEMENTATION

Parallelization of push-relabel algorithm for multi-core machines is not trivial. Complex synchronization is needed to prevent multiple threads interfering with each other. Due to synchronization overhead and structure of different classes of input graphs, developing scalable parallel implementations is hard. We try to tackle these issues and provide a practical implementation that will run considerably faster than the best serial implementations, on modern day multi-core computers.

Amongst several approaches, push-relabel based algorithms with certain heuristics are known to be superior in practice [8]. Not all algorithms provide polynomial worst-case complexity and some of these algorithms do not behave well in practice. We have based our parallel algorithm on Goldberg's push-relabel algorithm, with FIFO selection strategy and global relabeling heuristic [8]. The main reason we chose push-relabel algorithm to parallelize is that it offers more potential for parallelism (compared to augmenting path algorithms) by employing local push operations on arcs. In case of augmenting path algorithms, flow is sent through paths (multiple arcs, connected to each other). This results in loss of parallelism: when a thread works on pushing through a path, no other thread will be able to use the arcs that are part of that path.

Although practically HL selection strategy performs better, we used a FIFO selection strategy (each thread has its own FIFO and task stealing balances the load). Implementing a parallel version of HL selection would involve much overhead in terms of synchronization, compared to FIFO selection. Also the gap in performance of FIFO and HL selection is small. Hence, we used a slightly modified FIFO selection strategy that accounts for synchronization needs. We will be comparing our implementation to the work in [8].

We evaluated Intel's Threading Building Blocks (TBB) [11] and POSIX Threads (pthreads) [12] to implement our parallel algorithm. In the following sections, we study TBB and pthreads in detail. Then we give an overview of our overall solving approach,

followed by the details of our implementation.

### 4.1. Intel's Threading Building Blocks

TBB is a parallel programming library. It is implemented in C++, using templates. TBB offers various constructs such as *parallel for* and *parallel do*, that allow easy parallelization of certain tasks. Apart from parallel algorithm constructs, there are constructs that enable developers to create synchronized variables (atomic). We use TBB's atomic constructs in our parallel algorithm, for synchronizing access to shared variables.

### 4.2. POSIX Threads

Pthreads is a threading API that finds widespread use. Note that pthreads is a specification, not an implementation [13]. It is lower level than the API of TBB, and it does not have any high level constructs like the *parallel for* of TBB. There are methods for creating threads, for manipulating mutexes and condition variables in pthreads. We used pthreads to implement our parallel algorithm with our own task stealing mechanism. Using pthreads, one has to focus on threads rather than tasks, as opposed to TBB. This requires more effort to establish a parallel implementation, but provides flexibility.

### 4.3. Synchronization

Any parallel implementation targeting shared-memory computers should use synchronization for accessing shared variables. There are different ways to synchronize access to shared variables. We are interested in two such ways in our work, one is using mutexes and the other is atomic variables. Atomic variables provide synchronization at a smaller cost in terms of time, compared to mutexes. There is a trade-off, however, using atomic variables require certain patterns and are less applicable compared to mutexes. Our implementation is using atomic variables instead of mutexes. Atomic variables we use are of type tbb::atomic<int>, of Intel's TBB library.

TBB and pthreads can coexist in the same implementation. We needed TBB's abstraction of atomic variables and low level API of pthreads. That is why we incorporated both TBB and pthreads in our implementation.

### 4.4. Overview

At the beginning, we parse the input file and create internal data structures, which is the serial part of our implementation. Then comes the parallel part, in which each thread gets active nodes from its local queue and processes (applying push/relabel operations) them. When a thread causes a node to become active, that node is pushed to the thread's local queue. There is no global FIFO queue of active nodes. When the local queue of a thread becomes empty, the thread signals this to other threads and waits for some thread to donate active nodes.

Global relabeling heuristic is run periodically. Since global relabeling is an expensive operation compared to the basic operations of push and relabel, global relabelings are performed periodically. After each $n$ relabeling operations, all distance labels are computed from scratch by global relabeling. The best labeling is, calculating shortest paths from each node to the sink. When global relabeling is decided, all threads synchronize at a barrier and participate in global relabeling together. Algorithm terminates when there aren't any active nodes, that is, each local queue is empty.

We noticed that global relabeling heuristic constitutes about 30% of overall running time of the sequential implementations. This is in accordance with [5], in which they claim as much as 40% of overall running time is spent for global relabeling. Therefore, parallelization of global relabeling heuristic was crucial, as otherwise we would not obtain speed-up above three. Our overall implementation can be divided into two distinct parts: parallel global relabeling and parallel push-relabel operations.

## 4.5. Data Structures & Naming Conventions

We prefix global variables with *g_*, as a naming convention. We have two structures for data representation, Node and Arc, presented in Figure 4.1. Input graph is represented by an array of struct Node and an array of struct Arc. Both Node array and Arc array are global variables that are accessible to each thread. We denote Node array as *g_nodes* and Arc array as *g_arcs* in the pseudo-codes. The length of *g_nodes* array and *g_arcs* array (number of nodes and arcs in the graph) are denoted as *g_n* and *g_m* respectively. Source and sink nodes are denoted by *g_s* and *g_t* respectively. Number of threads present is denoted by *g_threadCount*. We refer to each thread with an index $t$, where $0 \leq t < g_threadCount$.

Node:

- *atomic int* e (excess)
- *atomic int* d (distance label)
- *Arc pointer* adj (adjacency list, linked list of Arcs)
- *Arc pointer* cur (current arc to push from)

---

Arc:

- *int* w (tail node of this arc)
- *int* cap (capacity, constant)
- *atomic int* flow (current flow on the arc)
- *Arc pointer* next (pointer to next arc in adjacency list)
- *Arc pointer* other (pointer to reverse arc)

Figure 4.1. Data structures for Node and Arc

## 4.6. Global Relabeling Implementation

Global relabeling is essentially a breadth-first traversal on an unweighted graph to determine shortest path distances from each node to the sink. This can be done in linear time with a backwards breadth-first search on the reverse of residual graph,

starting from sink.

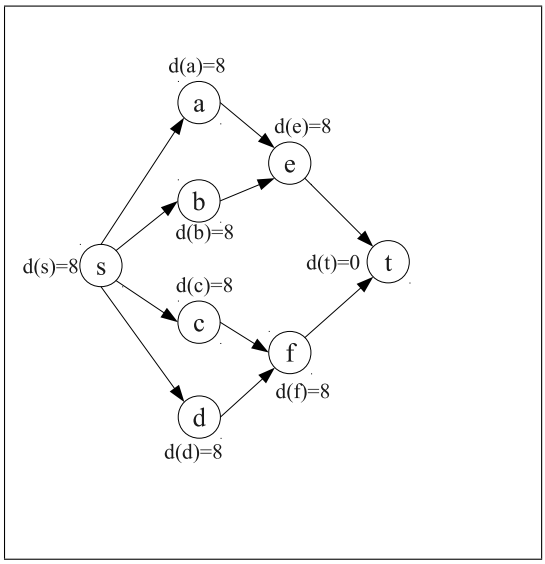
Global relabeling is implemented by running a breadth-first traversal starting from sink, on the reverse of the residual graph. After the traversal, all nodes that can be used to reach to the sink are relabeled with shortest path distances to the sink. First, we give the basic serial algorithm for performing global relabeling. Then we present our parallel global relabeling implementation.

### 4.6.1. Basic Serial Algorithm

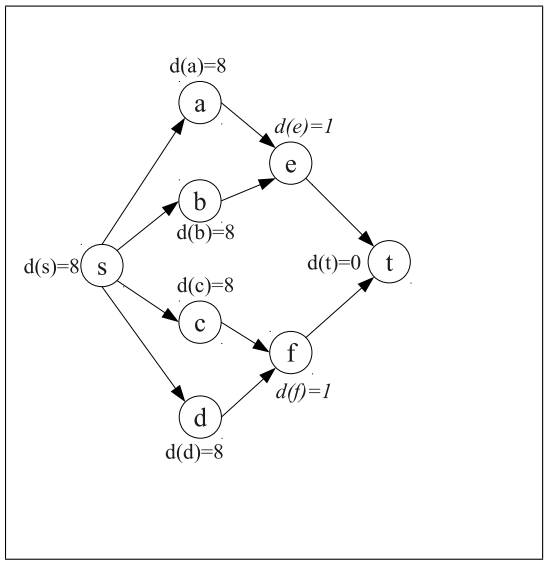
We describe global relabeling heuristic with an example in Figure 4.2, before going into the details of its implementation. Since there are eight nodes, initially each node is assigned a distance label of eight, except the sink node $t$ which always has a distance label of zero. Initial distance labels are depicted in Figure 4.2a. In the first iteration, in Figure 4.2b, nodes that are one hop away from the sink are relabeled to have a distance label of one. Next, nodes that are two hops away from the target are relabeled, with a distance label of two, in Figure 4.2c. Each node except the source and the sink are relabeled and the algorithm terminates.

Notice that, there is a potential of parallelism in global relabeling operation. Nodes that are at the same distance from the sink can be relabeled concurrently. We exploit this parallelism in our parallel global relabeling implementation described in Subsection 4.6.2. In the example of Figure 4.2, nodes $a$ and $b$ would be relabeled by one thread, while concurrently nodes $c$ and $d$ would be handled by another thread.

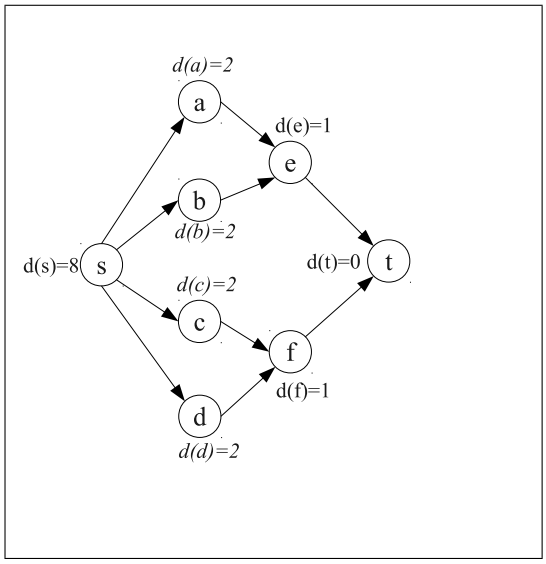
Serial implementation of global relabeling is straight-forward. Pseudo-code of the basic serial algorithm that performs global relabeling is given in Figure 4.3.




(a) Example input graph




(b) Nodes $e$ and $f$ are relabeled




(c) Nodes $a$, $b$, $c$ and $d$ are relabeled

Figure 4.2. Global relabeling heuristic example

**Serial global relabeling implementation**

**Require:** Residual graph

```
for all node v in g_nodes do
    v.d ← g_n
end for
g_nodes[g_t].d ← 0
initialize traversal queue tq with g_t
while tq is not empty do
    v ← pop from tq
    wd ← v.d + 1
    for all arc in v.adj do
        w ← arc.w
        other ← arc.other
        if w.d == g_n && other.cap − other.flow > 0 then
            w.d ← wd
            push w to tq
        end if
    end for
end while
```

Figure 4.3. Pseudo-code for serial global relabeling

### 4.6.2. Parallel Implementation

Parallelization is achieved, by relabeling all vertices having the same distance to the sink, in parallel. Thus, our global relabeling algorithm performs $k$ steps, each of which is executed in parallel by many threads, where $k$ is the largest distance label excluding the label of the source node. There are synchronization barriers at the end of each parallel step. We have a special shared queue data structure, presented in detail in Subsection 4.6.3, that allows for parallelization of breadth-first traversal with considerably small synchronization overhead.

Local queues of each thread is emptied at the beginning of global relabeling. As a thread traverses the graph, relabeling nodes, it also pushes the active nodes to its queue. Sink is relabeled to zero. All other nodes, including the source, have a distance label of $n$ (number of nodes) in the beginning.

Parallel algorithm is somewhat similar to the basic algorithm; a queue is used to ensure breadth-first traversal and each thread executes the loop in Figure 4.4 concurrently. At the first step (step 0), distance label of the sink is set to zero. At each step $i$, distance labels of nodes that are $i$ hops away from the sink are updated in parallel by each thread.

```
Method fprf_global_relabeling(t)
Require: Thread t
  nextlevel ← 0
  while g_hasmore do
    nextlevel ← nextlevel + 1
    _v ← call parallel_queue_pop(t)
    while _v ≠ −1 do
      Node *v ← g_nodes + _v
      for all a in v-> adj do
        _w ← a-> w
        Node *w ← g_nodes + _w
        if (a-> other-> cap) > (a-> other-> flow) && w-> d > nextlevel then
          original ← w-> d.fetch_and_store(nextlevel)
          if original > nextlevel then
            call parallel_queue_push(t, _w)
          end if
        end if
      end for
      _v ← call parallel_queue_pop(t)
    end while
    SYNCHRONIZATION BARRIER
    if t = 0 then
      g_hasmore ← call parallel_queue_revert()
    end if
    SYNCHRONIZATION BARRIER
  end while
```

Figure 4.4. Pseudo-code for parallel breadth-first traversal

### 4.6.3. Parallel Shared Queue Implementation

Our parallel shared queue structure, allows a parallel breadth-first traversal to be implemented with little synchronization overhead. The API for our shared queue contains the following methods:

- void parallel_queue_init (n, p)
- void parallel_queue_uninit ()
- int parallel_queue_pop (t)
- void parallel_queue_push (t, v)
- int parallel_queue_revert ()

where $n$ denotes the number of nodes, $p$ is the number of threads, $t$ is a thread ID and $v$ is a node. Notice that push and pop methods require thread ID $t$, so that the operation is applied on the appropriate in-queue or out-queue.

Each thread has a logical in-queue and a logical out-queue, all of which use the same underlying array for storage. In addition to the logical queue data structures, parallel shared queue uses a contiguous underlying array for storage. Out-queue is used for pushing newly encountered nodes, which are yet to be relabeled. In-queue is used for popping nodes, so that they can be relabeled. Members of the two logical queue data structures are presented in Figure 4.5, where pointers point to certain parts of the underlying array. Out-queue of thread $t$ is accessed by $g_outq[t]$ and in-queue of thread $t$ is accessed by $g_inq[t]$ in the pseudo-codes.

Space complexity of this data structure is $O(p \cdot n)$ where $p$ is the number of threads and $n$ is the number of nodes. We need $O(p \cdot n)$ space, since it might be the case that any of the threads push all $n - 1$ nodes (excluding sink node) in a single parallel step. That is, each out-queue should be able to hold $n - 1$ nodes. Therefore, storage requirement for a single thread is $O(n)$, since there are $n$ nodes in total and duplicate items are not allowed in the queues.

Underlying storage can be thought of as $p$ chunks of size $n$. Out-queues and in-queues are maintained such that there is no need for synchronizing access, since the queues never overlap on the underlying array. Out-queue of thread $i$ is always using chunk $i$ and never uses other threads' chunks. Therefore, there is no contention among threads for push operations. As for the in-queues, an in-queue may span more than one chunk. Furthermore, one chunk may provide storage for more than one in-queue.

Maintenance is performed by *parallel_queue_revert*, which scans each in and out queue and flushes out-queue contents to in-queues. This needs to be done in serial by a single thread, while other threads are waiting on the barrier. Benefit of the maintenance step is that, pop and push methods can be called in parallel with no synchronization cost. Notice that, this parallel queue is only used in global relabeling step and is not related to the active node queues which are local to each thread.

in-queue:

- *int pointer* head (moving head of an in-queue)
- *int pointer* tail (moving tail of an in-queue)
- *int* direction (direction in which an in-queue shrinks)

---

out-queue:

- *int pointer* head (moving head of an out-queue)
- *int* direction (direction in which an out-queue grows)
- *int* count (number of items)
- *int pointer* fixedhead (pointer to head of a chunk)
- *int pointer* fixedtail (pointer to tail of a chunk)

Figure 4.5. Data structures for parallel queue

An in-queue's head is where items are popped from, it points to the head of the in-queue. An in-queue is empty if its head equals its tail. In other words, the item at the tail of an in-queue does not belong to that in-queue, tail points to one position further from the last item in an in-queue. Direction field indicates the direction towards which head pointer moves as items are popped and gets the value -1 or +1. Notice that an in-queue is only used when popping items and in-queues are arranged such that each of them use a distinct part of the underlying array without any overlapping region. Figure 4.6 describes parallel_queue_pop method.

**Method parallel_queue_pop($t$)**

**Require:** Thread $t$

**Return:** Head of the queue or -1 if the in-queue is empty

**if** $inq[t].head = g_inq[t].tail$ **then**
  **return** -1
**end if**
$v \leftarrow *(g_inq[t].head)$
**while** $v < 0$ **do**
  $*(g_inq[t].head) \leftarrow 0$
  $g_inq[t].head \leftarrow g_inq[t].head - (v * g_inq[t].direction)$
  **if** $inq[t].head = g_inq[t].tail$ **then**
    **return** -1
  **end if**
  $v \leftarrow *(g_inq[t].head)$
**end while**
$*(g_inq[t].head) \leftarrow 0$
$g_inq[t].head \leftarrow g_inq[t].head + g_inq[t].direction$
**return** $v$

Figure 4.6. Pseudo-code for parallel_queue_pop

Out-queue's are used for pushing new items. Head pointer points to the location where an item will be pushed. Direction is used for the same purpose as in an in-queue,

here it shows the direction towards which an out-queue grows. Count is a counter of the items in an out-queue and fixedhead and fixedtail point to the beginning and end of the chunk which belongs to the owner thread of that out-queue. Notice that an out-queue is only used when pushing items and out-queues do not share common storage on the underlying array. Figure 4.7 describes *parallel_queue_push* method.

**Method parallel_queue_push**$(t, v)$

**Require:** Thread $t$, item $v$

**Ensure:** Item $v$ is pushed to the out-queue of thread $t$

$*(g_outq[t].head) \leftarrow v$

$g_outq[t].head \leftarrow g_outq[t].head + g_outq[t].direction$

$g_outq[t].count \leftarrow g_outq[t].count + 1$

Figure 4.7. Pseudo-code for parallel_queue_push

Next, we describe the maintenance method *parallel_queue_revert*. This method transforms out-queues into in-queues with some pointer manipulations, with no memory copying. Prerequisite of this method is that the in-queues are all empty. After this method executes, out-queues will be empty and directions of in and out queues will be flipped. Contents of the underlying array in a small example run would reveal more than a pseudo-code involving pointer arithmetic. We describe the execution of revert method, in a problem with four nodes and two threads.
After thread #0 first pushes node 10 and then node 20 onto its out-queue, contents of the array is as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 20 | 10 | 0 | 0 | 0 | 0 |

At this point, out queues and in queues have the following contents:

| g_outq[0].head | 1 |
|---|---|
| g_outq[0].direction | -1 |
| g_outq[0].count | 2 |
| g_outq[1].head | 7 |
| g_outq[1].direction | -1 |
| g_outq[1].count | 0 |

| g_inq[0].head | 0 |
|---|---|
| g_inq[0].tail | 0 |
| g_inq[0].direction | 1 |
| g_inq[1].head | 4 |
| g_inq[1].tail | 4 |
| g_inq[1].direction | 1 |

After calling revert, the array contents is as follows:

| 0 | 0 | 20 | 10 | 0 | 0 | 0 | -4 |
|---|---|---|---|---|---|---|---|

And out queues and in queues have the following contents:

| g_outq[0].head | 0 |
|---|---|
| g_outq[0].direction | 1 |
| g_outq[0].count | 0 |
| g_outq[1].head | 4 |
| g_outq[1].direction | 1 |
| g_outq[1].count | 0 |

| g_inq[0].head | 2 |
|---|---|
| g_inq[0].tail | 1 |
| g_inq[0].direction | -1 |
| g_inq[1].head | 7 |
| g_inq[1].tail | 2 |
| g_inq[1].direction | -1 |

Notice that the two items are fairly shared between the two threads after revert. Negative values in the array ($-4$ in this case) are not actual items to be popped. These negative values are used to span a single in-queue among multiple chunks. Although thread #1's in-queue has just one item in it, it spans two chunks. Let's look at what happens when thread #1 tries to pop an item from its in-queue. Its in-queue's head pointer points to 7, in which a value of -4 is encountered. This value is used to jump the pointer to point to 3 ($7 - 4 = 3$). Now head points to the value 10 and 10 is returned from the pop method. Next time thread #1 tries to pop an item, it will see its in-queue's head to be equal to its in-queue's tail and will return -1 indicating that the queue is empty.

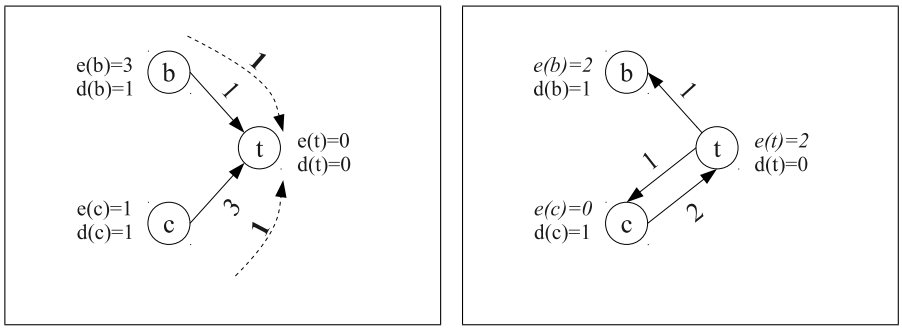
Maintenance method *parallel_queue_revert* runs in $O(p)$ time, since it scans out-queues and converts them to in-queues without moving the actual items around.

## 4.7. Parallel Algorithm

Our algorithm uses global relabeling and a FIFO selection strategy that is local to each thread. Each thread maintains a queue of active nodes and runs the main loop described in Figure 4.9. A thread gets nodes from its queue and pushes nodes that become active to its queue. Only when a thread's queue becomes empty, that thread waits until another thread sees it and transfers tasks to the waiting thread. Notice that, we continue to push from a node until either no excess remains at that node or the node cannot be used to push more flow since its arcs are saturated. In the latter case, a relabel operation is performed for that node and push operation again takes place, if possible.

Basic operations push and relabel do not take place while global relabeling is going on. That explains the synchronization barriers before and after global relabeling.

Next, we define in detail two basic operations, push and relabel. Push operation sends as much flow as possible through arcs emanating from a node. Pseudo-code for push operation is in Figure 4.10. At any point in the execution of the algorithm, we guarantee through synchronization via atomic variables, that a node appears in at most one queue. Therefore, we do not need to lock nodes when pushing flow out of them. When pushing out from a node $v$, $v$'s excess value will change, and that value is also possibly needed by other threads that may push towards $v$. That is why, *Node.e* is made atomic. Notice, also, *Arc.flow* is an atomic variable. The reason for this is that, when a thread is pushing in one direction, another thread might be pushing in the opposite direction. A visual depiction of parallel pushing is given in Figures 4.8a and 4.8b, in which two threads concurrently push flow to the sink. Since excess values are atomic variables, multiple threads can safely update the excess at the same time, there is no need to lock any nodes.




(a) Flow is pushed on $<b,t>$ and $<c,t>$ concurrently by two threads (b) Excess at $t$ is updated atomically by each thread

Figure 4.8. Parallel pushing by two threads

Relabel operation is described in detail in Figure 4.11. When a node is being relabeled, obviously its label is subject to change. At the same time, a thread might be pushing flow towards this node, hence needs to check the label. For this reason, we make *Node.d* an atomic variable.

Next, we describe how task-stealing is implemented. A thread that has no task left in its queue signals this by setting an idle flag which is an atomic variable. Each thread has its own idle flag and these flags are accessible from all threads. An idle flag is set only by the corresponding idle thread, and reset by another thread which has transferred a task from its own queue to the idle thread's queue. At each iteration in a thread's main loop, other threads are checked to see if there is an idle thread waiting for a task transfer. If there is an idle thread, and if a thread can atomically reset the corresponding idle flag, then that thread wins permission to transfer the task. We choose the oldest task (front of the queue) to transfer.

It is guaranteed that only a single thread transfers a task to an idle thread at any time. This is ensured by using *fetch_and_store* method of TBB's atomic variables. In particular, *idle.fetch_and_store(0)* will store the value 0 in idle and return idle's previous value. By checking the return value, we can make sure if the call changed idle's value from 1 to 0 or from 0 to 0. If the call changed idle's value from 1 to 0, then the thread making the call can safely transfer a task to the idle thread, knowing

that no other thread will attempt to transfer a task to the idle thread. If, on the other hand, the call returned 0; this means the value of idle was already reset to 0, that is, another thread got permission to transfer.

```
Parallel maximum flow solver(t)
Require: Thread t
Ensure: Maximum flow from g_s to g_t
  while true do
    if labelingcounter > g_n then
      SYNCHRONIZATION BARRIER
      call fprf_global_relabeling(t)
      SYNCHRONIZATION BARRIER
      if activenodecount = 0 then
        break
      end if
    end if
    if t.queue is empty then
      wait until a task is transferred to this thread
    end if
    transfer a task to an idle thread if possible
    v ← pop from t.queue
    d ← v.d
    while d < g_n do
      if (call fprf_push(t, v)) = 0 then
        break
      end if
      d ← call fprf_relabel(v)
    end while
  end while
```

Figure 4.9. Pseudo-code for parallel maximum flow solver

```
Method fprf_push(t, v)
Require: Thread t, Node v
Return: push status: 1 for push more, 0 otherwise)
  for all arc in v.adj do
    if arc.cap − arc.flow > 0 && arc.w.d = v.d − 1 then
      flow ← MIN(arc.cap − arc.flow, v.e)
      arc.flow ← arc.flow + flow
      arc.other.flow ← arc.other.flow − flow
      oldE ← arc.w.e.fetch_and_add(flow)
      if oldE = 0 && arc.w.d > 0 then
        push arc.w to t.queue
      end if
      v.e ← v.e − flow
      if v.e = 0 then
        return 0
      end if
    end if
  end for
  return 1
```

Figure 4.10. Pseudo-code for fprf_push

```
Method fprf_relabel(v)
Require: Node v
Return: new label for v
  v.d ← g_n * 2
  wd ← g_n * 2
  for all arc in v.adj do
    if arc.cap − arc.flow > 0 && arc.w.d < wd then
      wd ← arc.w.d
    end if
  end for
  wd ← wd + 1
  if wd < g_n * 2 then
    v.d ← wd
  end if
  return wd
```

Figure 4.11. Pseudo-code for fprf_relabel

# 5. RESULTS

## 5.1. Test Environment

We ran all our tests on CentOS release 5.3 with 2 Intel Xeon X5355 @2.66GHz quad-core CPUs and 16 gigabytes of RAM. Our test setup is fair. Each binary is run on the *same* computer with the *same* resources. Furthermore, binaries are generated on the same computer with the *same* compiler and optimization flags. We used GCC [14], version 4.1.2 20080704 (Red Hat 4.1.2-48) as compiler. Optimization level we used is O4. TBB version 2.2 and pthreads version NPTL 2.5 is used.

Input graphs we used are divided into two categories: synthetic graphs created using generators and graphs encountered in vision, that mimic real-life data. For the synthetic benchmark data, we used three different generators to generate various types of input graphs. These generators, namely "genrmf", "washington" and "acyclic-dense", can be found in [15].

Graph instances from computer vision domain are present at [16]. We present the size of these vision domain graphs in Table B.1. In various image processing related studies, these graphs are encountered. Finding minimum cut of a graph (hence maximum flow), is a basic building block of many image processing related algorithms and thus fast computation of maximum flow on these graphs affect other algorithms that depend on this computation [16]. It is important that our parallel implementation runs considerably faster than the best serial implementations on these real-life problems, which results show it does.

We collected two different measurements from our tests. First, we present how much parallelism is achieved by our parallel implementation by providing speed-up factors (relative to running our implementation with a single thread). Then, we refer the reader to the tables found in Appendix C and Appendix D. These tables provide a comparison of our implementation to the best sequential implementations, by

presenting average running times excluding input parsing. Input parsing times of our implementation and the sequential implementations were more or less the same.

## 5.2. Evaluation On Synthetic Data

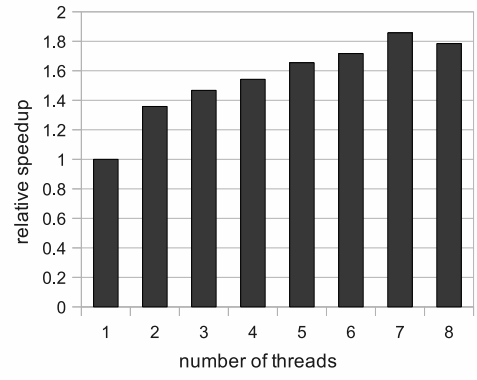
We followed a similar path to [5] in experiment design. For a given input graph class, we generated 20 instances by supplying different seed values for the random number generator. Then, we ran each implementation three times on each instance. We then calculated the average running times for each input class, calculated from 20 x 3 = 60 samples. Furthermore, we calculated relative speed-up for our parallel global relabeling implementation described in Subsection 4.6.2. Standard deviation of running times for each input set was calculated below 15% of the mean.

We used six different graph types, from the three generators. Running genrmf generator, we generated genrmf-wide and genrmf-long graphs with the parameters given in Table A.1. The number of nodes (N) and the number of arcs (M) are presented also. Using washington generator, we generated three different types of graphs. Parameters for washington-rlg-wide, washington-rlg-long and washington-line-moderate types can be found in Table A.2. For generating ac-dense types of graphs, we used size parameter as 8192 (first parameter supplied) and capacity parameter as 10000 (second parameter supplied) as demonstrated in Table A.3. For each type of graph, we used seeds in the range [0, 19] for generating 20 different instances.

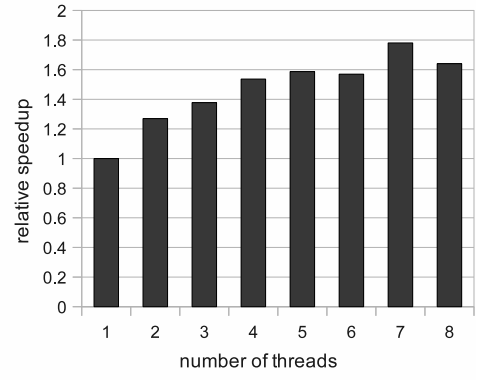
### 5.2.1. Performance of Parallel Global Relabeling

We measured the performance of our parallel global relabeling algorithm. We managed to get a speed-up of about two on certain graphs. This speed-up governs 30% of execution (since global relabeling constitutes about 30% of the sequential algorithm). Results are presented in Figure 5.1. Notice that, our parallel implementation will spend about 30%/2 = 15% of the total execution time for global relabeling. This means that, we cannot obtain an overall speed-up of more than about six, which can be seen from our overall speed-up results presented in Subsection 5.2.2. For long types of

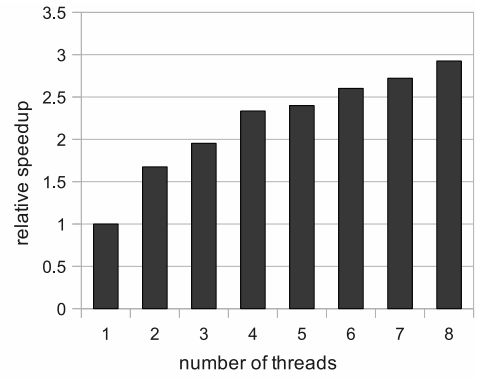
graphs (genrmf-long and washington-rlg-long families), there is no gain from parallel implementation. On the contrary, overhead of synchronization causes speed-up to be slightly less than one.



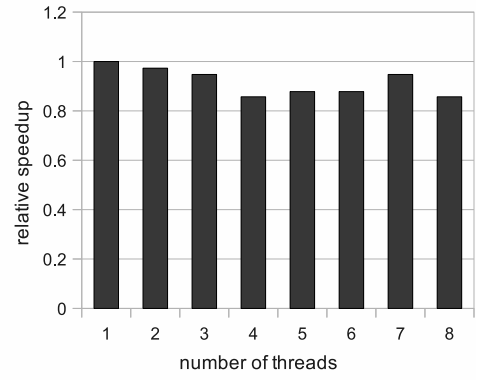

(a) genrmf-wide family



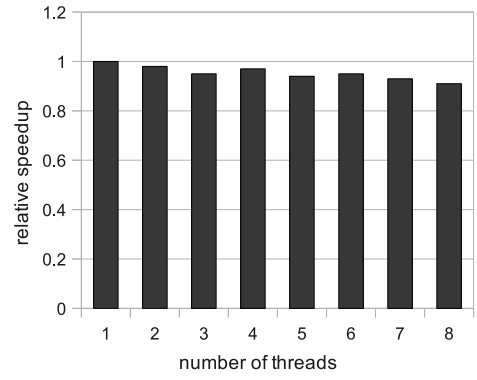

(b) washington-rlg-wide family



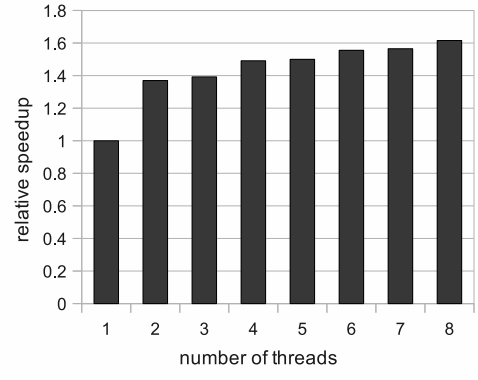

(c) ac-dense family




(d) genrmf-long family




(e) washington-rlg-long family

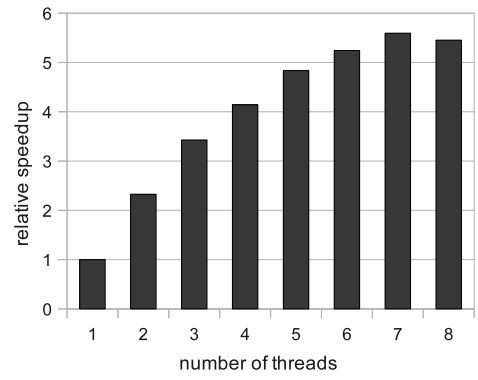



(f) washington-line-moderate family

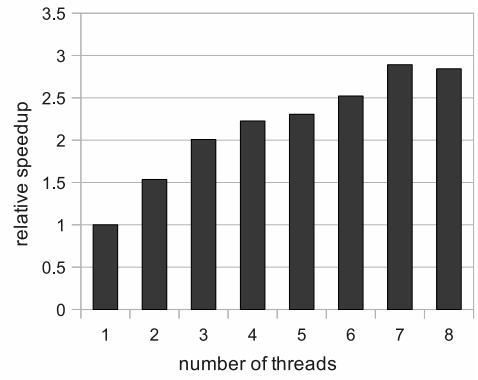
Figure 5.1. Speed-up of parallel global relabeling on synthetic graphs

### 5.2.2. Overall Performance

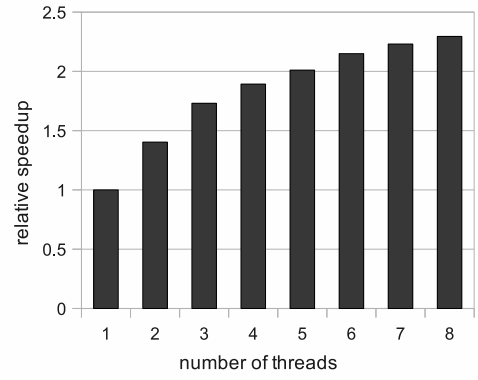
Overall performance and the parallelism achieved are presented here. We excluded input parsing (which is performed serially) times while measuring speed-up factors. We achieved speed-up of about five on certain types of graphs. Relative speed-up factors for genrmf-wide, washington-rlg-wide, ac-dense, genrmf-long, washington-rlg-long and washington-line-moderate are given in Figures 5.2a, 5.2b, 5.2c, 5.2d, 5.2e and 5.2f respectively.



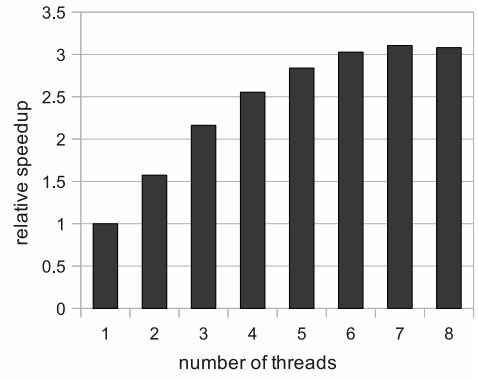

(a) genrmf-wide family



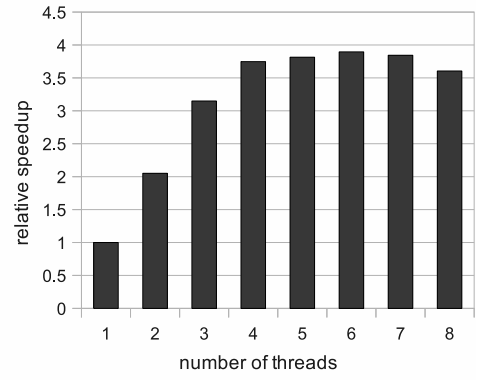

(b) washington-rlg-wide family



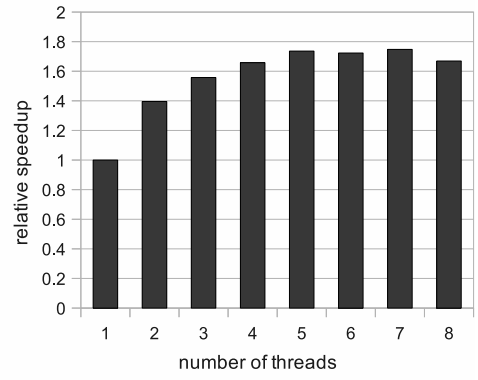

(c) ac-dense family




(d) genrmf-long family




(e) washington-rlg-long family




(f) washington-line-moderate family

Figure 5.2. Speed-up on synthetic graphs

Note that, different types of graphs yield different parallelism. The case of washington-line-moderate type of graphs need explaining. If we look at the execution of our algorithm on washington-line-moderate types of graphs, parsing dominates the overall execution. This is not because parsing washington-line-moderate types of graphs are any different than other types. The reason is, push-relabel based algorithms perform exceptionally well in practice on these types of graphs; thus, making parsing time appear large compared to the solving time. Hence, about 90% of execution is spent for parsing washington-line-moderate type of inputs. Since we have memory and time constraints (we cannot test on arbitrarily large graphs, and our washington-line-moderate input graph is considerably large), we were forced to take our measurements on an execution for which about 3-4 seconds were parallel and about 40 seconds were serial time spent for parsing. Although we exclude parsing times from our speed-up measurements, a parallel execution of 3-4 seconds is not significant enough to achieve considerable speed-up, due to parallelization overhead.

Notice that we ran our tests on a hardware with eight cores. That is why, as number of threads approach eight, increase in speed-up diminishes. In most of the results, performance of running with eight threads is actually slightly worse than running with seven threads. The reason for this is, our tests are not the only processes that are using the computer resources. There are other processes, in a time-shared system, so most of the time one core is actually serving processes other than our implementation.

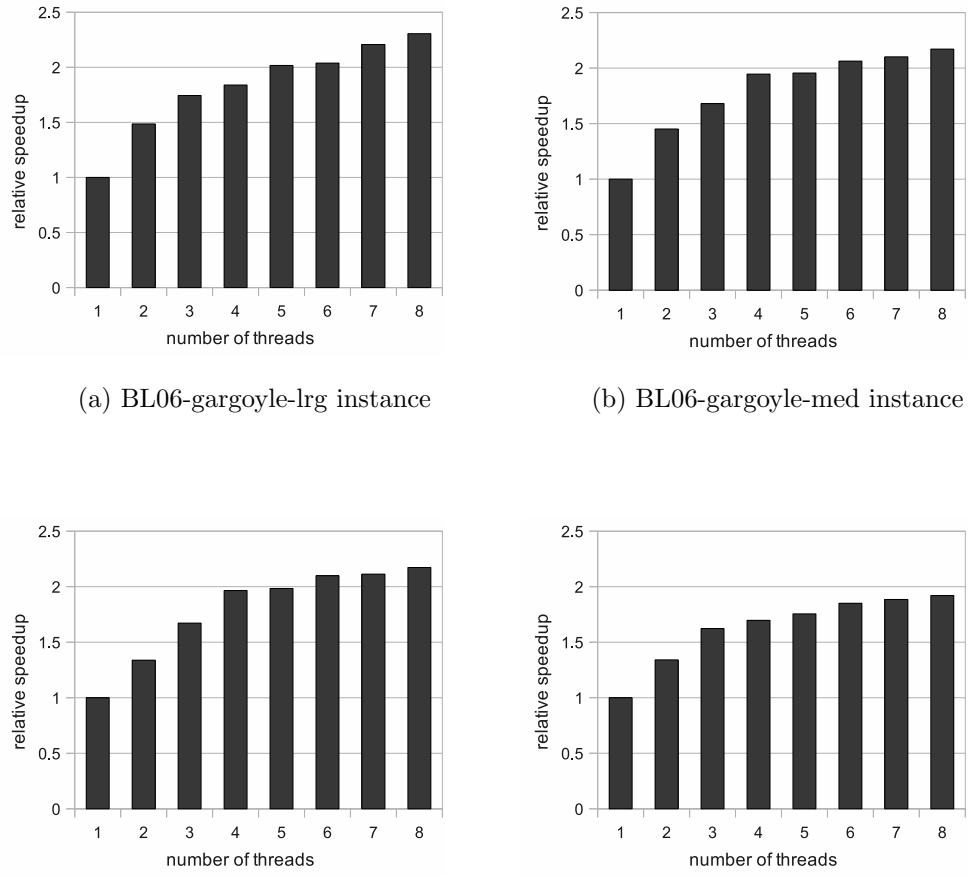
Next, in Tables C.1, C.2, C.3, C.4, C.5 and C.6, we compare practical performance of our parallel implementation to Goldberg's F_PRF, H_PRF, HI_PR, HI_PRO and HI_PRW implementations (HI_PR* implementations are derivatives of H_PRF, with heuristic parameters adjusted for better performance on certain instances) [9]. For washington-rlg-long and genrmf-long types of graphs, implementations that employ gap relabeling heuristic are superior. For both types of graphs, implementations with gap relabeling detected many gap nodes per gap and saved time by not inspecting those gap nodes. Notice that our parallel implementation is, overall, superior to the serial implementations. The execution times are directly proportional to the number of basic operations.

## 5.3. Evaluation On Vision Data

For tests on the data from [16], we used a slightly different experiment setup. For a given instance, we ran each implementation 20 times. Notice that there isn't any randomness in these instances. We then presented the average running times for each input class, calculated from 20 samples. Standard deviation for each input set was calculated below 15% of the mean.

### 5.3.1. Performance of Parallel Global Relabeling

Performance of our parallel global relabeling implementation on vision domain graphs are similar to that of synthetic data. A speed-up of about two is observed, in Figure 5.3. Notice again that, this speed-up concerns about 30% of overall execution.



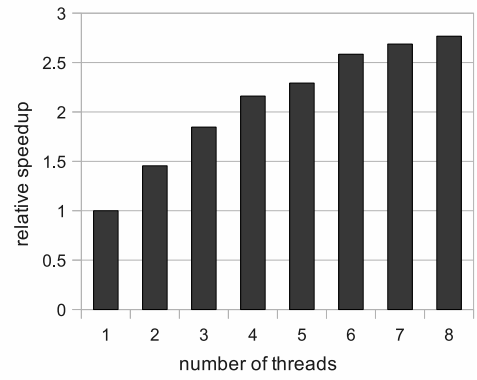

(a) BL06-gargoyle-lrg instance

(b) BL06-gargoyle-med instance

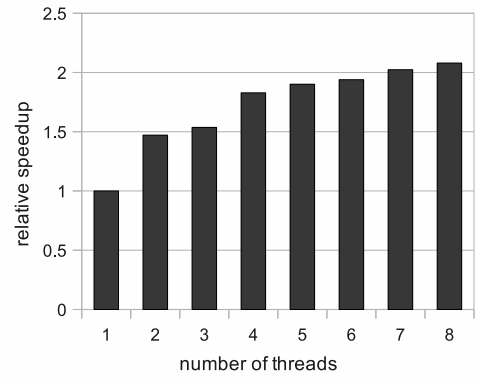
(c) LB07-bunny-med instance

(d) babyface.n6c100 instance

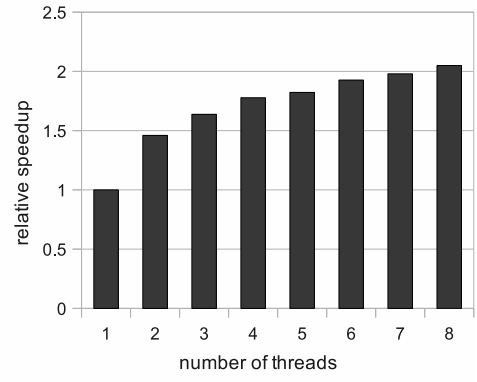
Figure 5.3. Speed-up of parallel global relabeling on vision graphs




(e) babyface.n6c10 instance




(f) BL06-camel-lrg instance



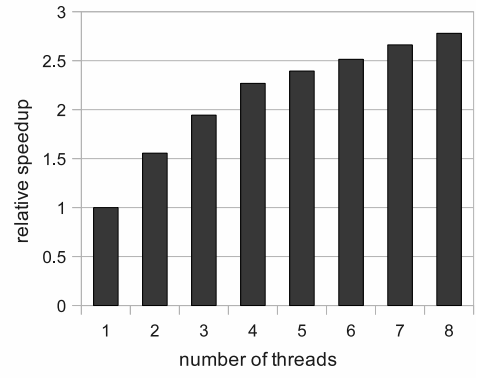

(g) BL06-camel-med instance

Figure 5.3. Speed-up of parallel global relabeling on vision graphs (continued)

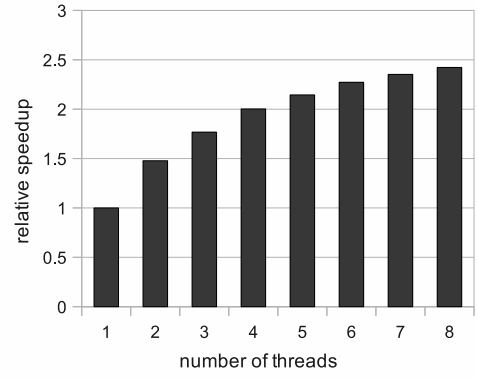
### 5.3.2. Overall Performance

Relative speed-up factors (again input parsing times are excluded, which is performed serially) for BL06-gargoyle-lrg, BL06-gargoyle-med, LB07-bunny-med, babyface.n6c100, babyface.n6c10, BL06-camel-lrg and BL06-camel-med are given in Figures 5.4a, 5.4b, 5.4c, 5.4d, 5.4e, 5.4f and 5.4g respectively.

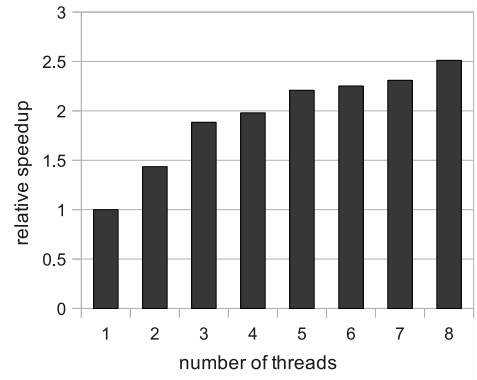
Comparison of our implementation to sequential implementations on vision instances can be found in Tables D.1, D.2, D.3, D.4, D.5, D.6 and D.7 respectively, for BL06-gargoyle-lrg, BL06-gargoyle-med, LB07-bunny-med, babyface.n6c100, babyface.n6c10, BL06-camel-lrg and BL06-camel-med instances.



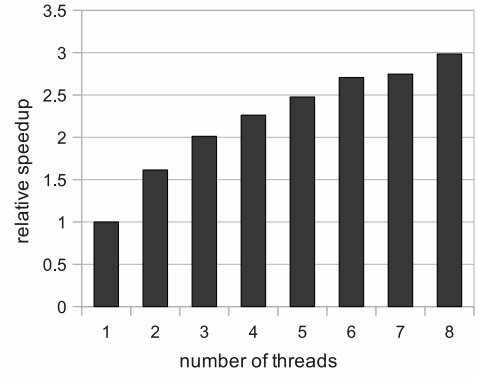

(a) BL06-gargoyle-lrg instance




(b) BL06-gargoyle-med instance

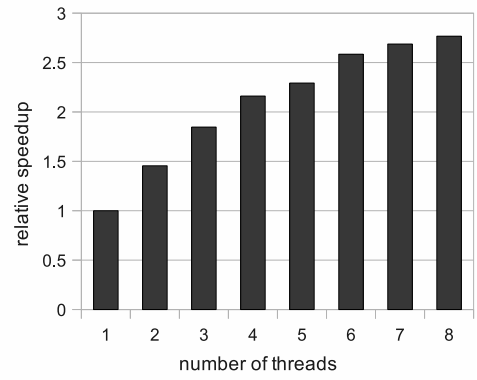



(c) LB07-bunny-med instance

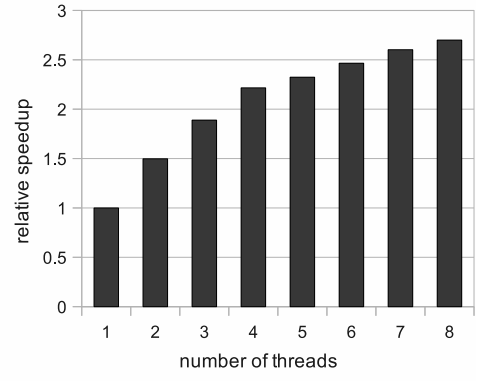



(d) babyface.n6c100 instance

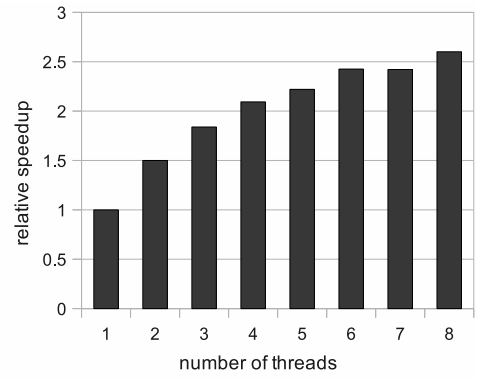
Figure 5.4. Speed-up on vision graphs




(e) babyface.n6c10 instance




(f) BL06-camel-lrg instance




(g) BL06-camel-med instance

Figure 5.4. Speed-up on vision graphs (continued)

## 5.4. Comments On Parallel Implementation Efficiency

Results regarding relative speed-up, presented in Subsections 5.2.2 and 5.3.2, show that the efficiency of our parallel implementation varies over different classes of input graphs. The main reason for this varying levels of speed-up is not cache related. On some graphs, our modified FIFO selection strategy (local queues for each thread) requires more basic operations, for the algorithm to terminate. That is because, global FIFO order cannot be maintained; hence, there is a trade-off between using multiple threads (thereby altering global FIFO order) and performing more basic operations. Since running times are directly proportional to the number of basic operations performed, an increase in the number of basic operations result in a lesser speed-up, than

what our implementation can achieve given that the number of basic operations do not increase.

Another contributing factor for varying speed-up levels is, certain graphs do not yield much parallelism for our implementation. That is, in the course of an execution, amount of active nodes is limited. There is not enough work to distribute among the threads and utilize all the threads.

# 6. CONCLUSION

We presented a parallel implementation for solving maximum flow problem in capacitated networks. Our implementation targets multi-core computers and uses FIFO selection strategy local to each processor. A parallel version of global relabeling heuristic is incorporated into the implementation. Our implementation uses separate queues for each processor and a task stealing mechanism to balance load. Main contribution of our work is that, we make extensive use of atomic variables to avoid explicit locks, this leads to a faster implementation. Synchronization overhead is minimized, thus available parallelism is better utilized.

In order to evaluate practical performance of our parallel implementation, we used Goldberg's various serial implementations in our comparisons. Since there is no single winner among these implementations, we included each of them in our experiments. Even then, our parallel implementation is shown to perform better overall. Our code outperforms its serial counterpart F_PRF of Goldberg on each graph instance we experimented. Except long types of graphs (washington-rlg-long and genrmf-long), our parallel implementation is faster than other implementations. Our parallel implementation performs exceptionally well on the vision data of [16]. Experiments suggest our implementation performs about twice as good as the fastest serial implementations. These results establish that our parallel implementation is robust and can be used as a replacement for the serial implementations of push-relabel based solvers on multi-core machines.

A few points need mentioning, concerning the performance of our implementation on synthetic washington-rlg-long and genrmf-long types of graphs. For long types of graphs, gap relabeling heuristic seems to boost performance. Another factor that explains why on long types of graphs our implementation performs slower, is that HL selection strategy yields a faster algorithm in practice for certain graphs (compared to FIFO selection strategy). Furthermore, we have experimented in our parallel global relabeling implementation, that these long types of graphs yield less parallelism. Even

though we achieved considerable speed-up over F_PRF on long types of graphs, serial implementations employing gap relabeling heuristic perform better than our parallel implementation. For further study, bottlenecks in our implementation can be identified empirically for different graph families.

Although combinatorial problems might not allow immediate parallelism, this area of research is promising with the recent advance in multi-core processors. There are various serial algorithms for solving maximum flow problem, with push-relabel based algorithms being the most practical. These algorithms can be studied to devise different types of parallel implementations for the maximum flow problem. In particular, our atomic variable usage can be employed to devise a parallel implementation making use of gap relabeling heuristic which improves practical performance further.

# APPENDIX A: GENERATOR PARAMETERS FOR SYNTHETIC DATA

Table A.1. Parameters for genrmf-wide and genrmf-long types.

| parameter | a | b | c1 | c2 | N | M |
|---|---|---|---|---|---|---|
| genrmf-wide | 400 | 16 | 1 | 10,000 | 2,560,000 | 12,614,400 |
| genrmf-long | 32 | 1,024 | 1 | 10,000 | 1,048,576 | 5,110,784 |

Table A.2. Parameters for washington-rlg-wide, washington-rlg-long and washington-line-moderate types.

| parameter | fct | dim1 | dim2 | range | N | M |
|---|---|---|---|---|---|---|
| washington-rlg-wide | 2 | 32,768 | 64 | 10,000 | 2,097,154 | 6,258,688 |
| washington-rlg-long | 2 | 64 | 65,536 | 10,000 | 4,194,306 | 12,582,848 |
| washington-line-moderate | 6 | 65,536 | 4 | 128 | 262,146 | 33,521,502 |

Table A.3. Parameters for ac-dense type of graphs.

| size | capacity | N | M |
|---|---|---|---|
| 8,192 | 10,000 | 8,192 | 33,550,336 |

# APPENDIX B: VISION DOMAIN GRAPH INSTANCES

Table B.1. Number of nodes & number of arcs for vision instances.

| graph | number of nodes | number of arcs |
|---|---|---|
| BL06-gargoyle-lrg | 17,203,202 | 86,175,090 |
| BL06-gargoyle-med | 8,847,362 | 44,398,548 |
| LB07-bunny-med | 6,311,088 | 38,739,041 |
| babyface.n6c100 | 5,062,502 | 30,386,370 |
| babyface.n6c10 | 5,062,502 | 30,386,370 |
| BL06-camel-lrg | 18,900,002 | 93,749,846 |
| BL06-camel-med | 9,676,802 | 47,933,324 |

# APPENDIX C: TEST RESULTS ON SYNTHETIC DATA

Table C.1. Comparison on genrmf-wide family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 144.56 | 608,354,883 | 324,971,087 | 42 |
| F_PRF | 472.2 | 604,329,223 | 265,709,785 | 104 |
| H_PRF | 168.94 | 357,932,622 | 210,455,046 | 82 |
| HI_PR | 153.85 | 345,341,109 | 203,143,194 | 81 |
| HI_PRO | 152.9 | 342,331,384 | 201,709,409 | 80 |
| HI_PRW | 146.9 | 331,505,857 | 195,644,032 | 78 |

Table C.2. Comparison on washington-rlg-wide family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 34.15 | 126,367,606 | 52,292,047 | 17 |
| F_PRF | 64.07 | 92,457,946 | 30,672,323 | 15 |
| H_PRF | 41.42 | 91,626,681 | 37,085,639 | 18 |
| HI_PR | 45.35 | 92,383,060 | 37,477,319 | 19 |
| HI_PRO | 44.96 | 92,311,356 | 37,445,503 | 19 |
| HI_PRW | 44.65 | 91,140,000 | 36,790,682 | 19 |

Table C.3. Comparison on ac-dense family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 20.61 | 278,379 | 128,737 | 6 |
| F_PRF | 36.29 | 202,765 | 74,208 | 9 |
| H_PRF | 19.18 | 134,287 | 46,827 | 6 |
| HI_PR | 32.86 | 181,995 | 52,971 | 8 |
| HI_PRO | 27.9 | 133,111 | 46,101 | 7 |
| HI_PRW | 34.04 | 182,576 | 53,466 | 8 |

Table C.4. Comparison on genrmf-long family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 17.28 | 59,429,217 | 17,250,374 | 16 |
| F_PRF | 34.68 | 66,248,774 | 17,965,208 | 17 |
| H_PRF | 11.8 | 33,646,761 | 26,848,312 | 26 |
| HI_PR | 10.2 | 33,412,519 | 26,636,933 | 27 |
| HI_PRO | 10.19 | 33,448,035 | 26,670,890 | 27 |
| HI_PRW | 11.34 | 37,272,662 | 29,120,180 | 29 |

Table C.5. Comparison on washington-rlg-long family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 74.71 | 416,787,522 | 61,070,981 | 7 |
| F_PRF | 156.08 | 455,468,236 | 36,393,644 | 9 |
| H_PRF | 2.22 | 17,066,286 | 2,757,994 | 1 |
| HI_PR | 2.52 | 17,078,084 | 2,764,515 | 2 |
| HI_PRO | 2.43 | 17,066,286 | 2,758,057 | 2 |
| HI_PRW | 2.63 | 17,435,751 | 2,773,932 | 2 |

Table C.6. Comparison on washington-line-moderate family. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 3.09 | 1,619,893 | 371,663 | 2 |
| F_PRF | 1.81 | 970,826 | 170,807 | 1 |
| H_PRF | 0.92 | 473,983 | 12,576 | 1 |
| HI_PR | 2.53 | 473,904 | 12,536 | 2 |
| HI_PRO | 2.53 | 473,983 | 12,576 | 2 |
| HI_PRW | 2.54 | 473,912 | 12,550 | 2 |

# APPENDIX D: TEST RESULTS ON VISION DATA

Table D.1. Comparison on BL06-gargoyle-lrg instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 139.35 | 481,472,484 | 185,776,642 | 10 |
| F_PRF | 361.42 | 515,678,942 | 137,625,624 | 9 |
| H_PRF | 327.64 | 593,231,209 | 288,901,272 | 17 |
| HI_PR | 252.31 | 573,404,232 | 277,538,216 | 17 |
| HI_PRO | 316.25 | 573,404,232 | 277,538,311 | 18 |
| HI_PRW | 235.23 | 545,866,547 | 252,860,479 | 16 |

Table D.2. Comparison on BL06-gargoyle-med instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 62.45 | 224,193,550 | 89,910,873 | 9 |
| F_PRF | 151.7 | 228,801,386 | 61,931,541 | 8 |
| H_PRF | 146.5 | 250,236,488 | 124,966,936 | 15 |
| HI_PR | 117.77 | 258,291,346 | 129,415,650 | 15 |
| HI_PRO | 144.41 | 258,291,346 | 129,415,725 | 16 |
| HI_PRW | 108.12 | 234,920,307 | 113,515,312 | 14 |

Table D.3. Comparison on LB07-bunny-med instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 22.65 | 68,152,847 | 28,370,124 | 5 |
| F_PRF | 36.08 | 59,696,211 | 25,244,356 | 5 |
| H_PRF | 42.8 | 61,505,463 | 35,620,715 | 6 |
| HI_PR | 46.83 | 62,132,193 | 35,947,981 | 7 |
| HI_PRO | 47.02 | 63,901,025 | 35,923,078 | 7 |
| HI_PRW | 63.93 | 114,232,037 | 59,410,754 | 11 |

Table D.4. Comparison on babyface.n6c100 instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 27.68 | 102,149,201 | 45,417,102 | 8 |
| F_PRF | 51.67 | 66,512,855 | 25,710,368 | 6 |
| H_PRF | 70.89 | 144,081,054 | 67,919,846 | 14 |
| HI_PR | 69.96 | 144,727,796 | 68,305,595 | 15 |
| HI_PRO | 69.76 | 145,046,119 | 68,172,555 | 15 |
| HI_PRW | 65.3 | 137,095,408 | 63,109,261 | 14 |

Table D.5. Comparison on babyface.n6c10 instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 24.06 | 68,909,744 | 31,885,082 | 8 |
| F_PRF | 51.7 | 60,133,237 | 25,339,596 | 6 |
| H_PRF | 52.86 | 108,512,380 | 55,665,530 | 11 |
| HI_PR | 52.27 | 107,922,330 | 55,655,142 | 12 |
| HI_PRO | 51.9 | 107,959,242 | 55,596,317 | 12 |
| HI_PRW | 54.5 | 111,159,170 | 57,578,572 | 13 |

Table D.6. Comparison on BL06-camel-lrg instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 163.84 | 518,414,986 | 201,545,792 | 10 |
| F_PRF | 318.19 | 438,198,557 | 151,200,025 | 9 |
| H_PRF | 401.1 | 830,152,726 | 392,764,654 | 21 |
| HI_PR | 399.37 | 846,148,257 | 404,634,873 | 22 |
| HI_PRO | 431.78 | 846,148,257 | 404,634,929 | 23 |
| HI_PRW | 389.83 | 848,117,029 | 397,136,601 | 22 |

Table D.7. Comparison on BL06-camel-med instance. Times in seconds.

| | time | # of pushes | # of relabels | # of global relabelings |
|---|---|---|---|---|
| our impl. | 72.49 | 240,321,840 | 98,548,804 | 9 |
| F_PRF | 132.17 | 194,376,600 | 67,737,622 | 8 |
| H_PRF | 163.62 | 337,977,034 | 159,339,534 | 17 |
| HI_PR | 163.42 | 350,016,127 | 168,312,018 | 18 |
| HI_PRO | 178.47 | 350,016,127 | 168,312,066 | 19 |
| HI_PRW | 150.38 | 336,872,618 | 158,178,409 | 17 |

# REFERENCES


1. Ahuja, R. K., T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, New Jersey, NJ, USA, 1993.

2. Goldberg, A. V., *Efficient Graph Algorithms for Sequential and Parallel Computers*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

3. Edmonds, J. and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of ACM*, Vol. 19, pp. 248–264, 1972.

4. Goldberg, A. V., "A New Max-Flow Algorithm", Technical report, Laboratory for Computer Science, MIT, Cambridge, MA, USA, 1985.

5. Anderson, R. J. and J. C. Setubal, "On the Parallel Implementation of Goldberg's Maximum Flow Algorithm", *Proceedings of the fourth annual ACM symposium on Parallel algorithms and Architectures*, 1992.

6. Goldberg, A. V. and R. E. Tarjan, "A New Approach to the Maximum Flow Problem", *Journal of the Association for Computing Machinery*, Vol. 35, pp. 921–940, 1988.

7. Cheriyan, J. and S. N. Maheshwari, "Analysis of Preflow Push Algorithms for Maximum Network Flow", *SIAM Journal on Computing*, Vol. 18, No. 6, pp. 1057–1086, 1989.

8. Cherkassky, B. V. and A. V. Goldberg, "On Implementing Push-Relabel Method for the Maximum Flow Problem", *Algorithmica*, Vol. 19, pp. 390–410, 1997.

9. "Andrew Goldberg's Network Optimization Library", `http://www.avglab.com/andrew/soft.html`, 2009.

10. Bader, D. A. and V. Sachdeva, "A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic", *Proc. 18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, 2005.

11. "Intel Threading Building Blocks", `http://www.threadingbuildingblocks.org/`, 2009.

12. "POSIX Threads Programming", `https://computing.llnl.gov/tutorials/pthreads/`, 2010.

13. Silberschatz, A., P. B. Galvin, and G. Gagne, *Operating System Concepts*, John Wiley & Sons, Inc., New Jersey, NJ, USA, 2005.

14. "GCC, the GNU Compiler Collection - GNU Project- Free Software Foundation (FSF)", `http://gcc.gnu.org/`, 2010.

15. "Synthetic Maximum Flow Families", `http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm`, 2009.

16. "Computer Vision at Western - Max-flow problem instances in vision", `http://vision.csd.uwo.ca/maxflow-data/`, 2010.