Scaling Algorithms
for Distributed Max Flow

Hershel M. Safer

25 May 1988

Sloan Working Paper 2021-88
Scaling Algorithms for Distributed Max Flow

Hershel M. Safer

25 May 1988

Sloan Working Paper 2021-88
Scaling Algorithms for Distributed Max Flow

Hershel M. Safer
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
arpa: hersh@theory.lcs.mit.edu

25 May 1988

Abstract

We describe two distributed algorithms for the maximum flow problem; they are the first such algorithms to use scaling. One of these algorithms has communication complexity $O(nm + n^2 \log U)$ and time complexity $O(n^2 \log U)$. For the majority of real communication networks, this is one of the two best known distributed maximum flow algorithms. Although this algorithm uses a factor of $\log U$ more time than that of Goldberg and Tarjan, it sends a factor of $n$ fewer messages. Our second algorithm has a more distributed flavor than the first; it runs in the same time, but it uses more messages. Both algorithms are based on the serial algorithm of Ahuja and Orlin, but we modify their algorithm so that the time complexity is better than the message complexity.

Keywords: distributed computation, maximum flow, scaling
1 Introduction

Given a directed graph with capacitated arcs, the maximum flow problem asks how much flow can be sent in the graph from a particular node $s$ to another node $t$, without exceeding the arc capacities and while maintaining flow conservation at all other nodes. This is a well-known problem with many applications, some of which are discussed in [FF62][Law76] [Eve79][PS82][Tar83].

We are concerned with solving this problem as part of control protocols for communication networks and other distributed systems. Distributed solution methods are often preferred in such settings for many reasons, such as increased reliability and avoidance of the potentially prohibitive processing costs of centralized algorithms[LPV72]. Several distributed algorithms for the maximum flow problem have been proposed; some of the best are listed in Table 1. Each is an adaptation of a serial or a synchronous parallel algorithm. In the following, $n$ is the number of nodes, $m$ is the number of arcs, and $U$ is the largest capacity of an arc out of $s$.

Cheung's[Che83] algorithm is based on the serial breadth-first-search algorithm of Edmonds and Karp[EK72]. The analysis in that paper only proves message complexity $O(n^5)$ and time complexity $O(n^4)$. Using a better bound for the number of augmentation phases ($nm$ instead of $n^3$) and a better breadth-first-search algorithm ([AG87]), however, yields the complexities shown in the table.

Segall's[Seg82] algorithm is based on Dinic's[Din70] serial algorithm. The first algorithm of Marberg and Gafni[MG84] is an asynchronous version of the synchronous parallel algorithm of Shiloach and Vishkin[SV82]. Their second algorithm[MG87] is based on the serial algorithm of Cherkasky[Che77].

The algorithms of Awerbuch[Awe85b], Goldberg[Gol85], and Cherian and Maheshwari [CM87] are all based on synchronous parallel algorithms; the synchronizer described in [Awe85a] is used to adapt them to asynchronous systems. Awerbuch's algorithm uses the algorithm of Shiloach and Vishkin[SV82]; the other two use algorithms that appear in the same papers. The algorithm of Goldberg and Tarjan[GT87b] is also based on that of [Gol85], but uses a local synchronization procedure, rather than the global one used by

<table>
<thead>
<tr>
<th>Authors</th>
<th>Communication</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheung</td>
<td>$n^{2.6}m + nm^2$</td>
<td>$n^{2.6}m$</td>
</tr>
<tr>
<td>Segall</td>
<td>$nm^2$</td>
<td>$n^2m$</td>
</tr>
<tr>
<td>Marberg &amp; Gafni</td>
<td>$n^3$</td>
<td>$n^3$</td>
</tr>
<tr>
<td>Awerbuch</td>
<td>$n^3$</td>
<td>$n^2 \log n$</td>
</tr>
<tr>
<td>Goldberg</td>
<td>$n^3$</td>
<td>$n^2 \log n$</td>
</tr>
<tr>
<td>Goldberg &amp; Tarjan</td>
<td>$n^2m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Marberg &amp; Gafni</td>
<td>$n^2m/2$</td>
<td>$n^2m/2$</td>
</tr>
<tr>
<td>Cherian &amp; Maheshwari</td>
<td>$n^3$</td>
<td>$n^2 \log n$</td>
</tr>
<tr>
<td>This paper</td>
<td>$nm + n^2 \log U$</td>
<td>$n^2 \log U$</td>
</tr>
</tbody>
</table>

Table 1: Complexity of some distributed max flow algorithms.
the others.

The two algorithms presented in this paper are based on the serial algorithm of Ahuja and Orlin[AO87]. These algorithms are unique because they are the first distributed maximum flow algorithms that use scaling. Our first algorithm uses global scaling phases as in the serial algorithm. For networks in which the arc capacities satisfy \( \log U = O(\log n) \), which is true for most real communication networks, this algorithm is one of the two best known for this problem. Although the time complexity of this algorithm is a factor of \( \log U \) worse than that of [GT87b], its communication complexity is better by a factor of \( n \) (except that in very sparse networks the factor is \( \frac{m}{\log U} \)). Our second algorithm has a more distributed flavor; each node defines the phase locally. Although this algorithm uses many more messages than the first, both run in the same time. In both cases, we have modified the algorithm of [AO87] so as to improve the time complexity below the trivial bound of the message complexity.

This paper contains five more sections. Definitions and computing issues are introduced in section two and the serial algorithm is reviewed in section three. Our first algorithm is presented in section four, along with a proof of its correctness, an analysis of its complexity, and a discussion of its appropriateness in our model of computation. Our second algorithm is described and analyzed in section five. Finally, conclusions are discussed in section six.

2 Preliminaries

In this section we introduce definitions, notation, and our model of computation. We then explain how to measure an algorithm's complexity and discuss an assumption related to that measurement.

Definitions and Notation

A flow network is a directed graph \( G = (N, A) \) with node set \( N \) and arc set \( A \subseteq N \times N \). Two nodes are distinguished: \( s \) is the source and \( t \) is the sink. In addition, a capacity function \( u : A \to \mathbb{Z}^+ \) is defined, where \( \mathbb{Z}^+ \) is the set of non-negative integers. We allow the possibility that \( u(v, w) \neq u(w, v) \).

Let \( n \) be the number of nodes and \( m \) the number of arcs. We assume that \( G \) is connected and does not contain multiple directed arcs, so \( n - 1 \leq m \leq n^2 \). For convenience in presenting the algorithms we also assume that \((v, w) \in A \) iff \((w, v) \in A\), though this is not essential to their correct operation. Define the neighbor set of \( v \in N \) to be \( A(v) = \{ w \mid (v, w) \in A \} \) and its degree to be \( \delta_v = |A(v)| \).

We assume, without loss of generality, that all arcs \((v, s)\) and \((t, v)\) have zero capacity. Let \( U = \max \{ u(s, v) : (s, v) \in A \} \). For simplicity in stating time bounds, we assume that \( U \geq 2 \). We further assume that \( U \) is polynomially bounded in the size of the network, that is, that \( U = n^{o(1)} \). This is known as the similarity assumption; it is explained and justified later in this section.

Given a function \( f : A \to \mathcal{R}^+ \), where \( \mathcal{R}^+ \) is the set of non-negative reals, the excess at
node $v \in N \setminus \{s,t\}$ is
\[
e(v) = \sum_{w: (w,v) \in A} f(w,v) - \sum_{w: (v,w) \in A} f(v,w).
\]
By definition, $e(s) = e(t) = 0$.

A flow is a function $f: A \to \mathbb{R}^+$ that satisfies:

(capacity constraint) \quad f(v,w) \leq u(v,w) \quad \forall (v,w) \in A, \quad (1)

(flow conservation constraint) \quad e(v) = 0 \quad \forall v \in N. \quad (2)

The value of a flow $f$ is the net flow into the sink $\sum_{w: (v,t) \in A} f(v,t)$. A maximum flow (or max flow) is a flow with maximum value among all possible flows in $G$, and the maximum flow problem is to find a maximum flow.

A preflow is a function $f: A \to \mathbb{R}^+$ that satisfies (1) and a relaxation of (2):

(excess constraint) \quad e(v) \geq 0 \quad \forall v \in N. \quad (3)

In this context, $e(v)$ corresponds to the net flow into $v$. Given a preflow, a node with positive excess is active; otherwise, it is inactive. For $\Delta$ a power of two, if $\frac{\Delta}{2} < e(v) \leq \Delta$, then $v$ is $\Delta$-active; if $e(v) \leq \frac{\Delta}{2}$, then $v$ is $\Delta$-inactive. Define $\text{scaleup}(x)$ to be $2^{\lceil \log_2 x \rceil}$ for $x > 0$ and 0 otherwise.

The residual capacity of arc $(v,w)$ with respect to a preflow $f$ is $u_f(v,w) \equiv u(v,w) - f(v,w) + f(w,v)$. Let $A_f$ be the set of arcs with positive residual capacity. The network $G_f = (N, A_f)$ is the residual network.

The distance $d_G(v,w)$ from node $v$ to node $w$ in a network $G$ is the minimum number of arcs on a directed path in $G$ from $v$ to $w$. We also use distance approximations. A valid labeling is a function $d: N \to \mathbb{Z}^+$ that satisfies:

\[
d(t) = 0, \quad (4)
\]

(labeling constraint) \quad d(v) \leq d(w) + 1 \quad \forall (v,w) \in A_f. \quad (5)

If the label $d(v)$ of every node $v$ is equal to the distance $d_{G_f}(v,t)$ from $v$ to the sink in the residual network, then the labeling is exact. A node might have out-of-date information about a neighbor's label; node $v$ thinks that the label of its neighbor $w$ is $d_v(w)$.

All logarithms in this paper are to the base 2, unless indicated otherwise.

**The Model of Computation**

We want to solve a max flow problem on an asynchronous point-to-point (store-and-forward) communication network. The communication network is modeled as a flow network in a common fashion [Gal82][Seg83][Awe85a]; the nodes correspond to the processors and the arcs correspond to the bidirectional communication links. The source and sink are two processors that need to communicate. The arc capacities are some measure of communication channel capacity, such as available bandwidth.
The network topology is given and static; processors and links do not fail. Each processor is initially ignorant of the topology except for its own incoming and outgoing links, how they are paired, and their capacities. The processors need not have names, but each one knows if it is the source, the sink, or neither. The memory available at each processor to use for the algorithm is assumed to be proportional to the number of the processor's neighbors.

The processors do not have access to a global clock or to shared memory; they communicate only by sending and receiving messages on their adjacent links. The messages sent on a link arrive error-free and in the order they were sent, with arbitrary but finite delays; the receiving processor knows on which link it received each message. A processor may send messages to several of its neighbors simultaneously.

Measuring Complexity

We want to measure how efficiently an algorithm uses the resources of communication bandwidth and time. The following measures are commonly used [Gal82][Seg83][Awe85a]. The message complexity $M$ of an algorithm is the worst-case number of messages that the algorithm sends, where a message is allowed to contain at most $O(\log n)$ bits.

The time complexity $T$ of an algorithm is the worst-case elapsed time from the start of the algorithm until its finish. In order to make this definition more precise we adopt two assumptions. First, since we are primarily concerned with the effects of message passing, we assume that the processors have infinite computing power; this means that we can ignore the time used for local computation. Second, we assume that the maximum communication delay on a link is one unit of time. This assumption is strictly for analytical purposes, however; we only consider event-driven algorithms that work properly with arbitrary finite delays.

The Similarity Assumption

We assumed earlier that $U$ is polynomially bounded in $n$. Several authors have argued that this assumption makes sense in the context of serial computing [Gab85][GT87a][AOS7][AOT87][AMOT87]. One reason is that this condition is true for many practical and theoretical problems; the assumption allows reasonable comparisons between time bounds that include numerical parameters and those that do not. This is also true for distributed algorithms.

Another reason for adopting this assumption, which is also a carry-over from the serial setting, is that current computers need extra time to operate on exponentially large numbers. Although we assume unlimited computing power for the processors in our model, we do so with an implicit understanding that this power will not be "abused"; we are, after all, trying to model real systems.

An additional reason for adopting the similarity assumption holds in the distributed setting. All known max flow algorithms send messages that contain numbers relating to "flow amounts"; the only known upper bound on such numbers is $U$, or in some cases the largest of all of the arc capacities. We have assumed, however, that messages are bounded
in length by $O(\log n)$. In order for a flow quantity to be carried in a single message, then, we need $\log U = O(\log n)$; but this is just what the similarity assumption guarantees.

3 The Serial Excess-Scaling Preflow Algorithm

In this section we review the serial algorithm on which our algorithms are based. We first explain the basic preflow algorithm and then show how scaling can be used to improve its performance.

The Preflow Approach

The preflow algorithm pushes flow from node to node, going toward the sink, rather than on source–sink paths as in earlier augmenting path algorithms. When a bottleneck is encountered, some flow is backtracked and, if possible, rerouted.

The algorithm starts by sending as much flow as possible from $s$ to each of its neighbors, saturating the $(s,v)$ arcs and creating a positive excess at each neighbor $v$ for which $u(s,v) > 0$. The algorithm maintains a preflow that eventually becomes a flow; at the end, $s$ is still disconnected from $t$ in $G_f$, so the flow must be optimal.

The algorithm proceeds by pushing flow from a node $v$ with positive excess to a neighbor $w$ that “seems” to be closer to the sink than $v$ is. As much flow as possible is pushed subject to the preflow conditions, that is, $y = \min \{ u_f(v,w), e(v) \}$. If the residual capacity constrains $y$, the push is saturating; otherwise, it is non-saturating. A push from $v$ to $w$ can take two forms: either more flow is pushed along the unsaturated arc $(v,w)$, or else flow is returned along the non-empty arc $(w,v)$.

A valid labeling $d$ is used to identify nodes that are “seemingly” closer to the sink. A push from node $v$ goes to a neighbor $w$ such that

\[(\text{push choice rule})\quad d(v) = d(w) + 1 \text{ and } (v, w) \in A_f.\]  

(6)

The labeling is initialized with a backwards breadth-first search starting from $t$, except that $d(s)$ is set to $n$.

If $v$ has no neighbors that satisfy (6), then all $w$ such that $(v, w) \in A_f$ must satisfy $d(v) < d(w) + 1$. In order to continue, $v$ is relabeled; $d(v)$ is raised just enough to allow $v$ to push, that is,

\[(\text{relabeling rule})\quad d(v) \leftarrow \min \{ d(w) + 1 : (v, w) \in A_f \} .\]  

(7)

When a node $v$ becomes disconnected from $t$ in $G_f$, its label will eventually become greater than $d(s) = n$ and its excess will be returned to the source.

No more than $nm$ saturating pushes and $O(n^2m)$ non-saturating pushes are ever needed [GT87b]. Performing the pushes in a particular order, however, yields a tighter bound on the latter: $O(n^3)$ [Gol85] or $n^2m^{1/2}$ [CM87]. The total running time can also be reduced to $O(nm \log \frac{n^2}{m})$ by using advanced data structures to decrease the average time per push[GT87b].
Using Scaling

Another way to reduce the number of non-saturating pushes is to consider only the leading bits of the excesses[AO87]. The idea is to spread the excess evenly throughout the network, rather than allowing it to get bunched up so that it must be rerouted.

The excess-scaling preflow algorithm works in log \( U \) phases, which are defined by an excess bound \( \Delta \). The algorithm maintains the invariant

\[
\text{(excess bound constraint)} \quad e(v) \leq \Delta \quad \forall v \in N.
\]

If \( \frac{\Delta}{2} < e(v) \leq \Delta \), then node \( v \) has large excess and is \( \Delta \)-active; otherwise, it has small excess and is \( \Delta \)-inactive. Each push goes from a node with large excess to a node with small excess. Whenever all of the nodes have small excess, \( \Delta \) is replaced by \( \frac{\Delta}{2} \) and the next scaling phase begins.

The value of \( \Delta \) is initially scaleup(\( U \)). This guarantees that (8) is satisfied when the algorithm begins, since the initializing pushes saturate the \((s, v)\) arcs. Throughout the algorithm, push quantities are chosen to ensure that (8) remains true; a push from \( v \) to \( w \) sends \( \min\{e(v), u_f(v, w), \Delta - e(w)\} \) units of flow.

The reason that this works well is that no more than \( O(n^2) \) non-saturating pushes are needed in each phase, and the total number of saturating pushes is still \( O(nm) \). As a result, the algorithm's complexity is \( O(nm + n^2 \log U) \), which, under the assumption of similarity, is better than other preflow algorithms, except for a recent improvement[AO87].

4 A Distributed Algorithm with Global Phases

We start this section by describing the distributed algorithm with coordinated global phases. We then prove that it works correctly and analyze its complexity. Finally, we examine the appropriateness of the assumption that local computation requires negligible time.

The Algorithm

Our first distributed algorithm is almost the same as the serial excess-scaling preflow method, with two important exceptions. The first difference is at a global level: several nodes can be pushing at once. The second difference is at a local level: a node can send multiple pushes simultaneously. The algorithm is precisely specified in Figure 1 to Figure 7, and is informally described in the remainder of this subsection.

Although creating a distributed form of a serial algorithm that uses only local information can be a routine exercise, some aspects can be potentially troublesome. Two such details were of concern in this case. First, since a node can have out-of-date information about its neighbors’ labels, it may send some inappropriate pushes. Second, the node that controls the phases must find out when all of the nodes have small excess so that it can initiate a new phase. Each of these could conceivably blow up the communication and time complexities. We show how to do both without increasing the complexities by more than a constant factor.
Algorithm GLOBAL for processor \( v \)

The subscript \( v \) indicates that the value is local to processor \( v \). The term "link \( g \)" represents either an adjacent link or the processor at the other end of the link, as appropriate. Although each anti-symmetric pair of links is often considered as a single link, occasionally a particular direction must be specified. The symbols \( g_{in} \) and \( g_{out} \) are used for this purpose.

<table>
<thead>
<tr>
<th>Messages</th>
<th>Scalar Variables</th>
<th>Array Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG(Start-Phase, ( \Delta ))</td>
<td>( d_v ) distance label</td>
<td>( d_v(g) ) dist. labels, ( g \in N )</td>
</tr>
<tr>
<td>MSG(Push, ( e, \Delta ))</td>
<td>( e_v ) excess</td>
<td>( M_v(g) ) stored pushes, ( g \in N )</td>
</tr>
<tr>
<td>MSG(Accept, ( y, q ))</td>
<td>( \Delta_v ) excess bound</td>
<td>( f_v(g) ) flows, ( g \in A )</td>
</tr>
<tr>
<td>MSG(Re-label, ( d ))</td>
<td>( y_v ) flow increment</td>
<td>( r_v(g) ) residuals, ( g \in A )</td>
</tr>
<tr>
<td>MSG(Quiet)</td>
<td>( x_v ) min. amt. left to send</td>
<td>( h_v(g) ) amt. sent to ( g \in N )</td>
</tr>
<tr>
<td></td>
<td>( z_v ) no. outstanding pushes</td>
<td>( \beta_v ) bfs tree children</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_v ) adjacent links</td>
<td></td>
</tr>
<tr>
<td>( a_v ) no. activity-tree children</td>
<td></td>
</tr>
<tr>
<td>( u_v(g) ) capacities, ( g \in A )</td>
<td></td>
</tr>
<tr>
<td>( b_v ) bfs tree parent</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The GLOBAL algorithm, beginning: Notation.

Furthermore, when a serial algorithm is adapted for distributed use, the communication complexity can generally be reduced as low as the time complexity of the serial algorithm, as it is in this case. If the adaptation is done mechanically, however, then the time complexity, which is never worse than the message complexity, will also probably not be any better. We show how to modify the algorithm so as to improve its time complexity below this trivial bound. Notation for the formal statement of the algorithm appears in Figure 1.

Overview: The sink node \( t \) controls the scaling phases. After \( t \) announces the start of a new phase, each node with large excess independently tries to push. This continues until all nodes have small excess. If node \( v \) pushes to a node \( w \) with small excess, then \( w \) accepts as much flow as it can. If, however, \( w \) has large excess, then \( v \) must wait until \( w \) has pushed enough flow so that \( e(w) \) is small. Some time after all of the nodes have small excess, \( t \) finds out and starts a new phase.

Transitions: A node only performs an action in response to a message from another node, except for the very first action of initiating the algorithm. Receipt of each kind of message triggers a particular subroutine. After completing the subroutine, the node may be ready to start a new operation; for example, a node that receives a push may find itself with large excess and be able to send a push. The procedure What-Next, which appears in Figure 2, is called to determine which action is appropriate (line 36, line 40, line 45, line 55).

Pushing: The details of pushing appear in Figure 3. The amount sent in a push from \( v \) to \( w \) depends on both \( e(v) \) and \( e(w) \). Since \( v \) does not know the value of \( e(w) \), the Push
Procedure *What-Next*:

1. if \((e_v > \frac{\Delta u}{2})\) then
2. \[ x_v \leftarrow \frac{\Delta u}{2} \]
3. Perform procedure *Push*
4. else if \((\exists g \in A_v \ni M_v(g) \neq \text{null})\) then
5. Perform procedure *Accept-A-Push*
6. else if \((a_v = 0)\) and \((v \notin \{s, t\})\) then
7. send MSG(Quiet) on link \(p_v\)
8. \[ p_v \leftarrow \text{null} \]

Figure 2: The GLOBAL algorithm, continued: *What-Next*.

message includes the value of \(e(v)\), and \(w\) determines how large the push will be. It then informs \(v\) of this decision (see "Accepting a Push").

Whenever \(v\) pushes, it sends out at least \(\frac{\Delta}{2}\) units of flow. If \(v\) has an incident arc with residual capacity of at least this amount that goes to an appropriately labeled neighbor \(w\), then \(v\) tries to push \(e(v)\) units to \(w\) (line 10). Node \(w\) will be able to accept at least \(\frac{\Delta}{2}\) of these units (Lemma 3), so \(v\) will have successfully pushed enough flow.

If, however, all of the arcs in the residual network that go from \(v\) to appropriately labeled neighbors have residual capacity smaller than \(\frac{\Delta}{2}\), then \(v\) will push on several arcs simultaneously. It determines the pushes that it will send by considering its appropriately labeled neighbors \(w\) one at a time. Rather than trying to send \(e(v)\) units to \(w\), though, \(v\) will only try to send \(u_f(v, w)\) units. After generating enough such messages to send out a total of at least \(\frac{\Delta}{2}\) units of flow, \(v\) simultaneously sends all of the Push messages.

This description ignores two contingencies: relabeling and rejected pushes. Both are considered later in this subsection.

**Accepting a Push:** The procedures involved in accepting a push are shown in Figure 4. When \(w\) receives a Push message from \(v\), it may not be able to process the message immediately for two reasons (line 25): \(w\) may be waiting for a response to a Push that it sent, or \(w\) may not have received the current Start-Phase message. It recognizes the latter case by comparing its value of \(\Delta\) to the one contained in the Push message. If \(w\) cannot process the Push message right away, it stores the message (line 24); when \(w\) finds out about its own Push or receives the Start-Phase message, it will process the stored Push (line 40, line 45, line 55).

To process a Push, \(w\) determines the maximum allowable push amount (line 29). It then informs \(v\) of this amount in an Accept message (line 33, line 35). Both \(v\) and \(w\) update the flows, residuals, and excesses (line 37, line 30).

This description does not discuss the possibility, which is considered later in this subsection, that a push might have to be rejected.

**Relabeling:** Relabeling is described in Figure 3 and Figure 5. Node \(v\) only pushes to appropriately labeled neighbors, that is, those that it thinks satisfy (6) (line 10, line 14).
Procedure \textit{Push}:

\begin{verbatim}
while \((x_v > 0)\) do
  if \((\exists g \in A_v \exists (r_v(g_{out}) \geq \frac{A_v}{2})\) and \((d_v = d_v(g) + 1)\) \textbf{then}
    Choose such a \(g\). /* Large push—try to send all excess*/
    \(y_v \leftarrow e_v - \frac{A_v}{2} + x_v\), \(x_v \leftarrow x_v - y_v\), \(z_v \leftarrow z_v + 1\), \(h_v(g) \leftarrow y_v\)
    queue MSG(Push, \(y_v, \Delta_v\)) on link \(g\)
  else if \((\exists g \in A_v \exists (r_v(g_{out}) > 0)\) and \((d_v = d_v(g) + 1)\) \textbf{then}
    Choose such a \(g\). /* Small push—only send enough to saturate arc*/
    \(y_v \leftarrow r_v(g_{out})\), \(x_v \leftarrow x_v - y_v\), \(z_v \leftarrow z_v + 1\), \(h_v(g) \leftarrow y_v\)
    queue MSG(Push, \(y_v, \Delta_v\)) on link \(g\)
  else /* Must relabel*/
    Simultaneously send queued Push messages
    \(d_v \leftarrow \min\{d_v(g) + 1 : r_v(g_{out}) > 0\}\)
    Simultaneously \(\forall g \in A_v\), send MSG(Relabel, \(d_v\)) on link \(g\)
\end{verbatim}

endwhile

Simultaneously send queued Push messages

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3}
\caption{The GLOBAL algorithm, continued: Pushing.}
\end{figure}

Recall, however, that \(v\) might have out-of-date information about its neighbors' labels, so it cannot be sure which of its neighbors actually satisfy (6); this leads to the phenomenon of rejected pushes, which is discussed later in this subsection.

If \(v\) is trying to push but has no neighbors that it thinks satisfy (6), then \(v\) relabels itself (line 20) and informs its neighbors of its new label by simultaneously sending them Relabel messages (line 21). While these messages are in transit, \(v\)'s neighbors have out-of-date information about \(v\)'s label.

Note that \(v\) may have to relabel while it is in the middle of generating a sequence of pushes; in fact, \(v\) may have to relabel several times during such a sequence. Since \(v\) simultaneously sends the Push messages that it has already generated (line 19) before sending the Relabel messages, this does not cause a problem.

\textbf{Rejected Pushes}: Occasionally node \(v\) may send a Push message to a node \(w\) while a relabel message is in transit from \(w\) to \(v\). This means that \(v\) used out-of-date information about \(d(w)\), and should not have sent the Push message to \(w\). Such a Push message must not be processed.

Node \(w\) recognizes this situation when it checks \(d(v)\) before processing a push (line 28) and simply ignores the message. Note that \(w\) must know the correct value of \(d(v)\); since messages on an arc arrive in order, the last Relabel message from \(v\) must have arrived before the Push message.

When \(v\) receives the Relabel message before receiving an Accept message (line 42), it realizes its error and cancels the local effects of the push. The required actions are shown in Figure 5. If enough flow is rejected so that \(v\) has not actually pushed at least \(\frac{A_v}{2}\) units, \(v\)
On receipt of MSG(Push, e, Δ) on link g: /* $d_v + 1 \geq d_g = d_v(g)$ */

\[ M_v(g) \leftarrow \text{MSG(Push, } e, \Delta)/* \text{Store push} */ 

if ((z_v = 0) and (Δ = Δ_v)) then perform procedure Accept-A-Push

Procedure Accept-A-Push:

Choose a stored push, say $M_v(g) = \text{MSG(Push, } e, \Delta)$.

\[ M_v(g) \leftarrow \text{null} \]

if ($d_v(g) = d_v + 1$) then /* o.w., previous Relabel message cancels push at g. */

\[ y_v \leftarrow \min \{v, r_v(g_{in}), \Delta_v - e_v\} /* \text{Maintain preflow conditions} */ 

Update $f_v(g_{in})$, $f_v(g_{out})$, $r_v(g_{in})$, $r_v(g_{out})$, and $e_v$.

if ($p_v = \text{null}$) then /* $v$ not yet in activity tree. */

\[ p_v \leftarrow g \]

send MSG(Accept, $y_v$, 1) on link g

else /* $v$ already in activity tree. */

send MSG(Accept, $y_v$, 0) on link g

Perform procedure What-Next

On receipt of MSG(Accept, $y$, $q$) on link g: /* $h_v = g$ */

Update $f_v(g_{in})$, $f_v(g_{out})$, $r_v(g_{in})$, $r_v(g_{out})$, and $e_v$.

\[ z_v \leftarrow z_v - 1, \quad h_v(g) \leftarrow 0 /* \text{This push is over} */ 

if ($q = 1$) then $a_v \leftarrow a_v + 1 /* \text{Increment count of activity-tree children} */$

if ($z_v = 0$) then perform procedure What-Next /* All pushes acknowledged */

Figure 4: The GLOBAL algorithm, continued: Accepting a Push.

resumes pushing where it left off. This might not be necessary, however; the rejected push may have carried a small amount of flow, but the final push may have been large, so that more than $\frac{\Delta}{2}$ units of flow were pushed.

Initialization: The algorithm performs two kinds of initialization: one occurs at the beginning of the algorithm, and the other at the start of each phase. Both are described in Figure 6. The algorithm is initialized by running a distributed breadth-first search (dfs) algorithm to build a backwards dfs tree rooted at $t$ (line 46). The algorithm of [AG87] is efficient enough that this step is not a bottleneck. Also, the value of $U$ is communicated to $t$, and the arcs out of $s$ are saturated as in the serial algorithm (line 47).

The dfs tree serves two purposes. First, it is used to initialize each node's local variables at the beginning of the algorithm. Second, it is used to start each phase; $t$ broadcasts the new value of $\Delta$ along the dfs tree (line 53). When node $v$ receives this broadcast, it determines what to do by executing the procedure What-Next that was discussed earlier in this section under "Transitions". Note that the dfs tree remains fixed throughout the algorithm—it does not change in response to changes in labels, flows, or the structure of the residual network.
On receipt of MSG(Rlabel, d) on link g:

41 \[ d_v(g) \leftarrow d \] /* Update label estimate */
42 if \( h_v(g) > 0 \) then /* This message cancels the push to g */
43 \[ z_v \leftarrow z_v - 1, \quad x_v \leftarrow x_v + h_v(g), \quad h_v(g) \leftarrow 0 \]
44 if \( x_v > 0 \) then perform procedure Push /* Must push at least \( \frac{\Delta}{2} \) */
45 else if \( z_v = 0 \) then perform procedure What-Next

Figure 5: The GLOBAL algorithm, continued: Relabel Messages.

Initialization:

46 Build bfs tree from t; each node v learns \( b_v, \beta_v, d_v, \) and \( d_v(g) \), and performs
47 \[ e_v \leftarrow 0, \quad \Delta_v \leftarrow 0, \quad z_v \leftarrow 0, \quad p_v \leftarrow \text{null}, \quad a_v \leftarrow 0 \] /* Init local variables */
48 \( \forall g \in A_v, \quad f_v(g_{\text{in}}) \leftarrow 0, \quad f_v(g_{\text{out}}) \leftarrow 0, \quad r_v(g_{\text{in}}) \leftarrow u_v(g_{\text{in}}), \quad r_v(g_{\text{out}}) \leftarrow u_v(g_{\text{out}}), \quad h_v(g) \leftarrow 0, \quad M_v(g) \leftarrow \text{null} \)
49 \( p_s \leftarrow s, \quad d_s \leftarrow n, \quad p_t \leftarrow t \)
47 At node s, determine \( U \) and send it to node t. \( \forall g \in A_s, \) saturate link \( g_{\text{out}} \).
48 At node t, \( \Delta_t \leftarrow 2 \cdot \text{scaleup}(U) \) and send MSG(Start-Phase, \( \Delta_t \)) to itself.

On receipt of MSG(Start-Phase, \( \Delta \)) on link g: /* \( g = b_v \) */

49 \( p_o \leftarrow g, \quad \Delta_v \leftarrow \Delta \) /* Join the activity tree */
50 if \( v = t \) then /* Begin a new phase */
51 \( \Delta_t \leftarrow \frac{\Delta_t}{2} \)
52 if \( \Delta_t < 1 \) then Done
53 Simultaneously \( \forall g \in \beta_v \), send MSG(Start-Phase, \( \Delta_v \)) on link g
54 \( a_v \leftarrow |\beta_v| \) /* Init count of activity-tree children */
55 Perform procedure What-Next

Figure 6: The GLOBAL algorithm, continued: Initializations.

Ending: The algorithm has two kinds of endings, just as it has two kinds of initializations: the end of the algorithm and the end of each phase. The former occurs when \( t \) recognizes that a phase has ended and that the next phase would have \( \Delta < 1 \) (line 52), as shown in Figure 6.

The latter occurs when \( t \) recognizes that all of the nodes have small excess. For this purpose the algorithm uses the activity tree described in [DS80]. The activity tree contains all of the nodes that are \( \Delta \)-active, as well as some that are \( \Delta \)-inactive. When \( t \) is the only node in the activity tree, it knows that the phase has ended. The rest of this subsection contains a more detailed description of the activity tree that some readers may want to skip.

To understand the operation of the activity tree, think of nodes as having three possible states: asleep, awake, and busy. Nodes that are \( \Delta \)-active are "busy" pushing flow. Nodes that are not busy, but have, either directly or indirectly, caused another node to become
On receipt of MSG(Quiet) on link g: /* \( v = p_g \) held when message was sent. */
56 \( a_v \leftarrow a_v - 1 \) /* Decrement count of activity-tree children */
57 if \( ((a_v = 0) \text{ and } (z_v = 0)) \) then /* Leaf of activity tree and not pushing */
58 if \( (v \neq t) \) then
59 send MSG(Quiet) on link \( p_v \) /* Drop out of activity tree */
60 \( p_v \leftarrow \) null
61 else /* \( v = t \) */
62 send MSG(Start-Phase, \( \Delta_t \)) to itself /* Start new phase */

Figure 7: The GLOBAL algorithm, concluded: Quiet Messages.

busy, are awake. Note that a node must be busy before it becomes awake. All other nodes are asleep.

The activity tree contains all of the nodes that are either awake or busy. The leaves of the activity tree are all busy. The activity-tree parent of node \( v \) is the node that most recently made \( v \) go from the sleeping state to the busy state. Each node remembers how many activity-tree children it has and which node is its activity-tree parent.

Node \( w \) enters the activity tree if it is asleep when it receives a Push message from a node \( v \); \( v \) becomes the activity-tree parent of \( w \) (line 32). If \( w \) is already awake when it receives a Push from \( v \), then the activity-tree parent of \( w \) remains unchanged. Node \( w \) informs \( v \) which of these cases holds by setting a bit in the Accept message (line 33, line 35, line 39).

If \( w \) joins the activity tree but finds that it still has small excess, it immediately drops out of the activity tree by sending a Quiet message to its activity-tree parent (line 6). Otherwise, \( w \) pushes and becomes an internal node of the activity tree. When an awake node receives a Quiet message from each of its activity-tree children, it becomes a leaf of the activity-tree that is not busy, so it drops out of the activity tree in the same way (line 57); the details appear in Figure 7.

We have omitted some facts from the above description. The activity tree is re-initialized in each phase by the Start-Phase messages. As a result, the original definitions of the three states were not quiet precise. Note also that a node can enter and leave the activity tree several times and have several parents during each phase.

Correctness

We now prove that the Global algorithm finds a correct solution, if it terminates. The fact that it terminates is proved in the next subsection. The main theorem is preceded by several lemmas about basic properties of the algorithm. Citations are to proofs of similar claims for related serial algorithms; GT refers to [GT87b] and AO refers to [AOS87].

The first two lemmas show why valid labels are interesting: they are lower bounds on the distances to the sink in the residual graph.
Lemma 1 (GT) At each \( v \in N \), the condition \( d(v) \leq d_w(w) + 1 \) is maintained for all neighbors \( w \) such that \( (v, w) \in A_f \). Furthermore, \( d(v) \) is never decreased, and is strictly increased whenever \( v \) is relabeled. In addition, \( d(s) = n \) and \( d(t) = 0 \) throughout the algorithm.

Proof. The first claim is initially true (line 46). Now consider the first time that one of the first two claims becomes false. This can happen because of two operations: a node relabels (line 20), or a node updates its estimate of a neighbor's label (line 41).

When \( v \) relabels, the tests on line 10 and line 14 guarantee that the new value of \( d(v) \) is strictly larger than the old one. Furthermore, when \( w \) updates \( d_w(v) \), it must increase \( d_w(v) \), since \( d(v) \) is non-decreasing and messages on a link arrive in order. Hence this operation also maintains the claims.

Initially \( d(s) = n \) and \( d(t) = 0 \) (line 46). A node can only relabel if it is trying to push, and must call Procedure Push the first time from line 3; but \( s \) and \( t \) are prevented from doing so by the test on line 1, since \( e(s) = e(t) \equiv 0 \).

Lemma 2 (GT) The algorithm maintains a valid labeling, and \( \forall v \in N, d(v) \leq d_{G_f}(v, t) \).

Proof. The labels are initially exact, and therefore valid, except for \( d(s) \) (line 46); but the \( (s, v) \) arcs are immediately saturated. So consider the first time that (5) is violated. Two operations could make this occur: relabeling and pushing. Labels are non-decreasing (Lemma 1); so \( d_w(w) \leq d(w) \), and relabeling (line 20) maintains a valid labeling.

When \( v \) pushes to \( w \), \( A_f \) can lose \( (v, w) \) and/or gain \( (w, v) \). Removing \( (v, w) \) just removes a constraint of (5), and cannot make the labeling invalid. Furthermore, \( d(w) = d(v) - 1 \), so adding \( (w, v) \) does not make the labeling invalid.

Now let \( v = w_1 - w_2 - \ldots - w_k = t \) be a \( v \)-t path in \( G_f \); it has length \( (k - 1) \). Since the labeling is valid, the labels satisfy \( d(w_{k-1}) \leq d(w_k) + 1 = d(t) + 1 = 1, d(w_{k-2}) \leq d(w_{k-1}) + 1 \leq 2, \ldots, d(v) = d(w_1) \leq d(w_2) + 1 \leq k - 1 \). So \( d(v) \) is a lower bound on the length of any \( v \)-t path in \( G_f \), including the shortest one.

We now show why \( \Delta \) is called the "excess bound".

Lemma 3 (AO) During a phase with excess bound \( \Delta \), the excess at each \( v \in N \) satisfies \( e(v) \leq \Delta \). If the phase terminates, then at termination \( e(v) \leq \frac{\Delta}{2} \). Each non-saturating push during the phase sends at least \( \frac{\Delta}{2} \) units of flow.

Proof. The first phase starts with \( e(v) \leq \Delta = \text{scaleup}(U) \) (line 46, line 47), and the same is inductively true of each subsequent phase because of the termination condition and line 51. If the \( \Delta \) bound holds before a push, it remains true after the push (line 29). If the phase terminates, then because of the operation of the activity tree, the results in [DS80] imply that \( e(v) \leq \frac{\Delta}{2} \).

A non-saturating push from \( v \) to \( w \) sends either \( e(v) \) or \( \Delta - e(w) \) units (line 29). In the first case, \( e(v) > \frac{\Delta}{2} \) (line 1). In the second case, \( e(w) \leq \frac{\Delta}{2} \) (line 1), so \( \Delta - e(w) \geq \frac{\Delta}{2} \).

We can now prove that an optimal solution is found at termination.
Theorem 4 (GT, AO) If the algorithm terminates, then it has found a maximum flow.

Proof. An initial preflow is constructed on line 46 and line 47, and the preflow conditions are maintained each time a push amount is chosen (line 29). If the algorithm terminates, then $\forall v \in N$, the excesses satisfy $e(v) < 1$ (Lemma 3, line 52). All excesses are integral and non-negative, so they must be zero, and the final preflow is a flow. But $d(s) = n$ (Lemma 1), so $s$ is disconnected from $t$ in $G_f$ (Lemma 2). This is a classical optimality condition for the max flow problem [FF62].

Complexity

We now prove that the algorithm terminates and analyze its complexity. The first lemma will prove useful later.

Lemma 5 (GT) $\forall v \in N, e(v) > 0 \Rightarrow G_f$ contains a directed path from $v$ to $s$. In addition, $0 \leq d(v) < 2n$.

Proof. By [FF62, section 2.2], the flow in any preflow can be decomposed into three parts in $G$: flow on directed $s-t$ paths, flow on directed cycles, and flow on directed paths from $s$ to active nodes. If $e(v) > 0$, then $G$ contains a directed $s-v$ path with positive flow. The reverse of each arc on this path is in $A_f$, and these arcs form a directed $v-s$ path in $G_f$.

Initially $d(v) \geq 0$ (line 46), and it can never decrease (Lemma 1). The label can only change when $e(v) > 0$ (line 1); but then $G_f$ contains a $v-s$ path. Since $d(s) = n$ (Lemma 1), an argument similar to that in Lemma 2 shows that $d(v) \leq d(s) + (n - 1) = 2n - 1$.

The next three lemmas bound the number of push and relabel operations.

Lemma 6 (GT) The algorithm performs at most $2n^2$ relabel operations, sends at most $4nm$ Relabel messages, and sends at most $4nm$ rejected Push messages.

Proof. For all $v \in N$, $0 \leq d(v) < 2n$ (Lemma 5). Since labels never decrease (Lemma 1), $v$ is relabeled at most $2n$ times, and at most $2n^2$ relabels are performed in total. When $v$ relabels, it sends $\delta_v$ Relabel messages, so the total number of Relabel messages is at most $\sum_{v \in N} 2n|A(v)| = 4nm$. Finally, each rejected Push is canceled by a Relabel message, and no Relabel message can cancel more than one Push (though a relabel operation at $v$ can cancel $\delta_v$ pushes), so at most $4nm$ pushes are rejected.

Lemma 7 (GT) The algorithm performs at most $nm$ saturating pushes.

Proof. If $v$ sends a saturating push to $w$ at time $t_1$, then $d^1(v) = d^1(w) + 1$, and $(v, w)$ is removed from $A_f$. Such a push cannot occur again until $(v, w)$ is back in $A_f$ (line 10, line 14); this only happens after $w$ pushes to $v$ at time $t_2 > t_1$, at which time $d^2(w) = d^2(v) + 1$. Since $d^2(v) \geq d^1(v)$ (Lemma 1), it follows that $d^2(w) \geq d^1(w) + 2$. Since $d(w)$ must increase by at least 2 between each two consecutive saturating pushes from $v$ to $w$, at most $n$ such saturating pushes can occur (Lemma 5); so the total number of saturating pushes is at most $nm$. 


Lemma 8 (AO) The algorithm performs at most $8n^2$ non-saturating pushes during any phase.

Proof. Consider the potential function $\Phi = \sum_{v \in N} \frac{d(v)}{\Delta} e(v)$. Since $0 \leq \frac{e(v)}{\Delta} \leq 1$ (Lemma 3), and $0 \leq d(v) < 2n$ (Lemma 5), $\Phi$ always satisfies $0 \leq \Phi \leq 2n^2$. At the beginning of a phase, then, $\Phi \leq 2n^2$, and at the end of a phase, $\Phi \geq 0$. Two operations affect the value of $\Phi$ during a phase: relabeling and pushing.

When a relabel operation increases $d(v)$ by an amount $p$, $\Phi$ goes up by $p \frac{e(v)}{\Delta} \leq p$; so the increase in $\Phi$ due to relabeling $v$ is at most $2n$ (Lemma 5), and the total increase in $\Phi$ due to relabeling is at most $2n^2$.

Every push decreases $\Phi$. Each non-saturating push sends at least $\frac{\Delta}{\Delta}$ units of flow (Lemma 3) from a node with label $d(v)$ to a node with label $d(v) - 1$, and so decreases $\Phi$ by at least $\frac{1}{2}$. Hence the number of non-saturating pushes is no more than $\frac{2n^2 + 2n^2}{1/2} = 8n^2$.

Finally we can analyze the algorithm's complexity and prove that it terminates.

Theorem 9 The algorithm sends at most $M = O(nm + n^2 \log U)$ messages.

Proof. Count the messages of each kind separately:

Initialization: $O(n^{1.6})$ [AG87].
Start-Phase: The broadcast uses $(n - 1)$ messages per phase, or $O(n \log U)$ in total.
Relabel: At most $4nm$ (Lemma 6).
Rejected Push: Charged to the Relabel message that canceled it.
Non-Saturating Push: $O(n^2 \log U)$ (Lemma 8).
Saturating Push: At most $nm$ (Lemma 7).
Accept: Charged to the Push message that it answers.
Quiet: Charged to the Start-Phase or Push message that most recently caused the sending node to join the activity tree.

So the total number of messages is $O(nm + n^2 \log U)$.

Theorem 10 The algorithm terminates.

Proof. The number of messages is bounded (Theorem 9) and all of the messages are delivered in finite time, so the only reason that the algorithm might not terminate is that some messages are not processed. The only messages that are delayed, however, are Push messages. Suppose that $v$ sends a Push message to $w$, but $w$ does not process the message; this can happen either because $w$ has not yet received the current Start-Phase message or because $w$ is waiting acknowledgement for one or more Push messages that it has sent (line 25).

The former situation cannot persist, so suppose that the second situation occurs. In order for this condition to continue indefinitely, $w$ must be unable to push enough of its
excess. But $G_f$ contains a $w$–$s$ path (Lemma 5), and $s$ only delays Push messages because of the first possibility mentioned above, which cannot last. By induction on the length of the $w$–$s$ path, then, $w$ must be able to dispose of its excess. So the $v$–$w$ Push cannot be delayed indefinitely, and the algorithm must terminate. ■

**Theorem 11** The algorithm requires at most $T = O(n^2 \log U)$ units of time.

**Proof.** The algorithm is never deadlocked, and messages are always in transit (Theorem 10). We allocate the algorithm’s elapsed time to each kind of message in order; so the time allocated to rejected Push messages, for example, is the time during which such messages are in transit, but no Initialization, Start-Phase, or Relabel messages are in transit. Note that since the communication delay is at most one unit of time, the time allocated to a given kind of message can be no greater than the number of messages of that kind.

- **Initialization:** $T \leq M = O(n^{1.6})$.
- **Start-Phase:** $T \leq M = O(n \log U)$.
- **Relabel:** $O(n^2)$ (Lemma 6).
- **Rejected Push:** Charged to the time for the corresponding Relabel message.
- **Non-Saturating Push and corresponding Accept:** $T \leq M = O(n^2 \log U)$.
- **Saturating Push and corresponding Accept:** $O(n^2 \log U)$ (see below).
- **Quiet:** Charged to the time for another message (see below).

So the algorithm spends $O(n^2 \log U)$ units of time in total if the above claims for Quiet and saturating Push messages can be justified, which we do now.

**Quiet:** Recall that a node $v$ might drop out of the activity tree and send a Quiet message immediately upon joining the tree, or else it might send other messages first. In the first case, the time for the Quiet message is charged to the time for the Start-Phase or Push message that brought $v$ into the activity tree. In the second case, the time for the Quiet message is charged to the time for the first of the other messages.

Note that this two case approach is needed because the first possibility fails when several Push messages that are sent simultaneously from a single node all add nodes to the activity tree; the newly added nodes may not return their Quiet messages at the same time.

**Saturating Push and corresponding Accept:** We will show that within every 2 unit interval of the time allocated to the saturating Push messages and the corresponding Accept messages, at least $\frac{\Delta}{2}$ units of flow are sent in saturating pushes, accepted, and acknowledged. A potential function argument similar to that in Lemma 8 shows that at most $O(n^2)$ such intervals can occur during each phase; so at most $O(n^2 \log U)$ units of time are allocated to saturating Push messages.

Think of a saturating Push message sent from $v$ to $w$ and the resulting Accept message sent from $w$ to $v$ as having a tag $v$. At a time when any of these messages is in transit, some
of them have a tag $v_{\text{min}} = \arg \min \{ d(v) : v \text{ is a tag} \}$. Since no node $w$ with $d(w) < d(v_{\text{min}})$ is pushing, and no Start-Phase messages are still in transit (recall that we are allocating time in order), Push messages from $v_{\text{min}}$ cannot be delayed; so $v_{\text{min}}$ receives Accept messages for each of these Pushes within 2 units of time after sending the Pushes.

If all of the Push messages from $v_{\text{min}}$ are accepted, then within 2 units of time, $v_{\text{min}}$ has disposed of $\frac{\Delta}{2}$ units of flow. Suppose, however, that some of them are rejected. If only a small amount of flow is rejected, so that the total amount accepted is at least $\frac{\Delta}{2}$ units, then the rejections do not cause a problem (line 44). Otherwise, $v$ will try to push again; but the time for sending the Push messages and their acknowledgements is bounded, and can be charged to the Relabel messages. So in at most 2 units of the time that is allocated to saturating Push messages, at least $\frac{\Delta}{2}$ units of flow are indeed sent and accepted. ■

Free Local Computation

The definition of time complexity assumes that the time required for local computation is negligible compared to the length of a communication delay. We now discuss whether this assumption is reasonable for our algorithm by determining the amount of local computation required for each message transmission. Note that a processor may send messages to several neighbors simultaneously in a single transmission. We therefore consider a reasonable amount of local processing to be $O(\delta_v)$ for sending a transmission and $O(1)$ for receiving a message.

This algorithm can be implemented so as to satisfy these conditions. When node $v$ receives a Start-Phase message, it puts each of its incident arcs $(v, w)$ in one of three doubly linked lists. The first set of arcs are those on which $v$ can push a large amount of flow, that is, $d_v(w) = d(v) - 1$ and $u_f(v, w) \geq \frac{\Delta}{2}$. The second set of arcs are those on which $v$ can only push a small amount of flow, that is, $d_v(w) = d(v) - 1$ and $0 < u_f(v, w) < \frac{\Delta}{2}$. The third set of arcs are those on which $v$ cannot push, that is, $d_v(w) > d(v) - 1$ or $u_f(v, w) = 0$. Since the lists are doubly linked, each update can be performed in $O(1)$ time. We now examine the work required for sending and receiving each kind of message.

**Start-Phase:** A transmission requires building the doubly-linked lists and queueing the messages, which can be done in $O(\delta_v)$ time. Receiving a message requires $O(1)$ time to update variables.

**Push:** Each message requires $O(1)$ time to choose an arc, determine the amount of flow to push, and queue the message, so a transmission requires $O(\delta_v)$ time. Receiving a message requires $O(1)$ time to store it.

**Accept:** A transmission uses $O(1)$ time to process a Push message. A reception uses $O(1)$ time to update local variables.

**Relabel:** A transmission requires $O(\delta_v)$ time to queue the messages. Receiving a message requires $O(1)$ time to update the label and possibly cancel a Push.

**Quiet:** Sending and receiving each require $O(1)$ time.
5 A Distributed Algorithm with Local Phases

In this section we describe the algorithm with local phases and analyze its complexity. We omit the proofs of correctness and termination; they are similar to those for the previous algorithm.

The Algorithm

This algorithm is similar to the algorithm with global phases, and we describe here how it differs from the first algorithm. Much of the code is the same as that for the first algorithm, and we show the entire algorithm in Figure 8.

The basic difference is that this algorithm does not use global scaling phases; rather, each node determines locally what it thinks is the current value of $\Delta$. Node $v$ sets $\Delta(v) \leftarrow \text{scaleup}(e(v))$, so that if $v$ is active, it thinks that it has large excess and tries to push. For purposes of analysis, define $\Delta^* = \max \{ \Delta(v) : v \in N \}$. Note that no node knows the value of $\Delta^*$.

When $w$ receives a Push from $v$, $w$'s inclination will be to make $v$ wait, since $w$ thinks that $e(w)$ is large. If $\Delta(v) \leq \Delta(w)$, then this is what happens. If $\Delta(v) > \Delta(w)$, however, then $w$ realizes that it was mistaken about the value of $\Delta^*$ (line 19). In this case, $w$ accepts the Push immediately and updates its value of $\Delta(w)$. Notice that the update is done in such a way that $\Delta^*$ never increases (line 28), even though the local value of $\Delta(w)$ can increase.

Complexity

We now analyze the complexity of this algorithm. The message complexity, $O(n^2 U)$, is not very attractive. The number of messages can be so large because when $\Delta^*$ is large, a lot of small non-saturating pushes can be wasted by nodes $v$ with $\Delta(v) \ll \Delta^*$.

As discussed in [Bou88], however, in some situations the criteria for assessing distributed algorithms include the degree to which the algorithm has a distributed character. The discussion singles out a centralized monitor, such as the phase controller in the algorithm with global phases, as a feature that can be undesirable. If communication bandwidth is less of a constraint than the distributed character of the system, then the present algorithm may be preferable.

The following theorem shows that this algorithm's time complexity is the same as that of the first algorithm, despite this algorithm's large message complexity. We omit the proofs of analogues of lemmas in the previous section, but note that their proofs are similar to those already presented.

**Theorem 12** The algorithm requires at most $T = O(n^2 \log U)$ units of time.

**Proof.** At any time, consider only the messages from nodes $v$ with $\Delta(v) = \Delta^*$; some such message is always in transit. Let $\tau$ be any power of 2 such that $\tau \leq \text{scaleup}(U)$; a proof similar to that of Theorem 11 shows that $\Delta^* = \tau$ for at most $O(n^2)$ units of time, so the entire algorithm terminates in $O(n^2 \log U)$ units of time. ■
Algorithm LOCAL for processor \( v \)

<table>
<thead>
<tr>
<th>Messages</th>
<th>Scalar Variables</th>
<th>Array Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG(Push, ( e, \Delta ))</td>
<td>( d_v ) distance label</td>
<td>( d_v(g) ) dist. labels, ( g \in N )</td>
</tr>
<tr>
<td>MSG(Accept, ( y, q ))</td>
<td>( e_v ) excess</td>
<td>( M_v(g) ) stored pushes, ( g \in N )</td>
</tr>
<tr>
<td>MSG(Relabel, ( d ))</td>
<td>( \Delta_v ) excess bound</td>
<td>( f_v(g) ) flows, ( g \in A )</td>
</tr>
<tr>
<td>MSG(Quiet)</td>
<td>( y_v ) flow increment</td>
<td>( r_v(g) ) residuals, ( g \in A )</td>
</tr>
<tr>
<td></td>
<td>( x_v ) min. amt. left to send</td>
<td>( h_v(g) ) amt. sent to ( g \in N )</td>
</tr>
<tr>
<td></td>
<td>( z_v ) no. outstanding pushes</td>
<td>( \beta_v ) bfs tree children</td>
</tr>
</tbody>
</table>

Constants

- \( A_v \) adjacent links
- \( p_v \) activity-tree parent
- \( u_v(g) \) capacities, \( g \in A \)
- \( a_v \) no. activity-tree children
- \( b_v \) bfs tree parent

Initialization:

1. Build bfs tree from \( t \); each node \( v \) learns \( b_v, \beta_v, d_v, \) and \( d_v(g), \) and performs
2. \( e_v \leftarrow 0, \Delta_v \leftarrow 0, z_v \leftarrow 0, p_v \leftarrow \text{null}, a_v \leftarrow 0 \)
3. \( \forall g \in A_v, f_v(g_{\text{in}}) \leftarrow 0, f_v(g_{\text{out}}) \leftarrow 0, r_v(g_{\text{in}}) \leftarrow u_v(g_{\text{in}}), r_v(g_{\text{out}}) \leftarrow u_v(g_{\text{out}}), h_v(g) \leftarrow 0, M_v(g) \leftarrow \text{null} \)
4. \( p_s \leftarrow s, d_s \leftarrow n, p_t \leftarrow t \)
5. At node \( s, \) determine \( U. \)
6. \( \forall g \in A_s, \text{send} \) MSG(Push, \( r_s(g_{\text{out}}), \text{scaleup}(U) \)) on link \( g. \)

Procedure Push:

3. \( \text{while} \ (x_v > 0) \) do
4. \( \text{if} \ (\exists g \in A_v \exists (r_v(g_{\text{out}}) \geq \frac{\Delta_v}{2}) \) and \((d_v = d_v(g) + 1)) \) then
5. \( \text{Choose such a } g. \)
6. \( y_v \leftarrow e_v - \frac{\Delta_v}{2} + x_v, x_v \leftarrow x_v - y_v, z_v \leftarrow z_v + 1, h_v(g) \leftarrow y_v \)
7. \( \text{queue} \) MSG(Push, \( y_v, \Delta_v \)) on link \( g \)
8. \( \text{else if} \ (\exists g \in A_v \exists (r_v(g_{\text{out}}) > 0) \) and \((d_v = d_v(g) + 1)) \) then
9. \( \text{Choose such a } g. \)
10. \( y_v \leftarrow r_v(g_{\text{out}}), x_v \leftarrow x_v - y_v, z_v \leftarrow z_v + 1, h_v(g) \leftarrow y_v \)
11. \( \text{queue} \) MSG(Push, \( y_v, \Delta_v \)) on link \( g \)
12. \( \text{else} /* \text{Must relabel} */ \)
13. \( \text{Simultaneously send} \) queued Push messages
14. \( d_v \leftarrow \min \{ d_v(g) + 1 : r_v(g_{\text{out}}) > 0 \} \)
15. \( \text{Simultaneously} \ \forall g \in A_v, \text{send} \) MSG(Relabel, \( d_v \)) on link \( g \)
16. \( \text{endwhile} \)
17. \( \text{Simultaneously send} \) queued Push messages

On receipt of MSG(Push, \( e, \Delta \)) on link \( g: \ /* d_v + 1 \geq d_g = d_v(g) */ \)

18. \( M_v(g) \leftarrow \text{MSG(Push, } e, \Delta) \)
19. \( \text{if} \ (\Delta > \Delta_v) \text{ then} \) perform procedure Accept-A-Push

Figure 8: The LOCAL algorithm, beginning.
On receipt of MSG(Relabel, d) on link g:
20 \[ d_v(g) \leftarrow d \]
21 if \((h_v(g) > 0)\) then /* This message cancels the push to g*/
22 \[ z_v \leftarrow z_v - 1, x_v \leftarrow x_v + h_v(g), h_v(g) \leftarrow 0 \]
23 if \((x_v > 0)\) then perform procedure Push
24 else if \((z_v = 0)\) then perform procedure What-Next

Procedure Accept-A-Push:
25 Choose a stored push, say \(M_v(g) = MSG(Push, e, \Delta)\), with \(\Delta > \Delta_v\).
26 \(M_v(g) \leftarrow \text{null} \)
27 if \(((d_v(g) = d_v + 1) \text{ or } (g = s))\) then /* o.w., prev. Relabel cancels push at g.*/
28 \[ y_v \leftarrow \min\{e, r_v(g_{in}), \Delta - e_v\} \]
29 Update \(f_v(g_{in}), f_v(g_{out}), r_v(g_{in}), r_v(g_{out})\), and \(e_v\).
30 if \(v \notin \{s, t\}\) then \(\Delta_v \leftarrow \text{scaleup}(e_v)\)
31 if \((p_v = \text{null})\) then /*v not yet in activity tree.*/
32 \(p_v \leftarrow g\)
33 send MSG(Accept, \(y_v, 1\)) on link \(g\)
34 else /*v already in activity tree.*/
35 send MSG(Accept, \(y_v, 0\)) on link \(g\)
36 Perform procedure What-Next

On receipt of MSG(Accept, y, g) on link g: /* \(h_v = g\)*/
37 Update \(f_v(g_{in}), f_v(g_{out}), r_v(g_{in}), r_v(g_{out})\), and \(e_v\).
38 if \((v \neq s)\) then \(\Delta_v \leftarrow \text{scaleup}(e_v)\)
39 \[ z_v \leftarrow z_v - 1, h_v(g) \leftarrow 0 \]
40 if \((g = 1)\) then \(a_v \leftarrow a_v + 1/*v = p_g*/\)
41 if \((z_v = 0)\) then perform procedure What-Next

On receipt of MSG(Quiet) on link g: /* \(v = p_g\) held when message was sent.*/
42 \(a_v \leftarrow a_v - 1\)
43 if \((a_v = 0)\) then
44 if \((v \neq s)\) and \((e_v = 0)\) then
45 send MSG(Quiet) on link \(p_v\)
46 \(p_v \leftarrow \text{null} \)
47 else /*\(v = s\)*/
48 Done

Figure 8: The LOCAL algorithm, continued.
Procedure *What-Next*:

49  if (\( \exists g \in A_v \ni (M_v(g) = MSG(Push, e, \Delta) \neq null) \) and (\( \Delta > \Delta_v \))) then
50       \( x_v \leftarrow \frac{A_v}{2} \)
51       Perform procedure *Accept-A-Push*
52   else if \( (e_v > 0) \) then
53       Perform procedure *Push*
54   else if \( ((a_v = 0) \text{ and } (v \notin \{s,t\})) \) then
55       send MSG(Quiet) on link \( p_v \)
56       \( p_v \leftarrow null \)

**Figure 8:** The LOCAL algorithm, concluded.

## 6 Conclusions

We have described two distributed max flow algorithms. These algorithms are unique because they are the first to use *scaling*. Under the similarity assumption, which generally holds for real communication networks, our first algorithm is one of the two best known in terms of both communication and time complexity. Although the algorithm of [GT87b] achieves a factor of \( \log U \) better time complexity, this algorithm sends a factor of \( n \) fewer messages.

The second algorithm has the same time complexity as the first one. Although this algorithm uses a large number of messages, it may be desirable in some situations. For both algorithms, we have reduced the time complexity below the trivial upper bound of the message complexity.

One question that we leave open is whether the time bound of the first algorithm can be improved even further by exploiting parallelism between the nodes. Another open question is whether the second algorithm can be modified so as to substantially improve its message complexity without destroying its distributed character.

## Acknowledgements

It is a pleasure to acknowledge the help that my colleagues have provided with this research. I thank Jim Orlin for introducing me to scaling, Baruch Awerbuch for suggesting that I apply it to distributed algorithms, and both of them and Ravi Ahuja for many stimulating and helpful discussions. Thanks are due to Jonathan Eckstein for suggesting that I analyze the local computation time. I also thank Jim, Ravi, and Rob Freund for comments that led to improvements in the presentation of this paper.

This research was supported in part by an NSF Presidential Young Investigator Grant to James B. Orlin.
References


REFERENCES


