Parallel Maximum-Flow Algorithms
on a Transputer Hypercube

Claudia F. Portella †
Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ
Caixa Postal 68511
21945 Rio de Janeiro - RJ, Brazil

Valmir C. Barbosa ‡
IBM Rio Scientific Center
Caixa Postal 4624
20071 Rio de Janeiro - RJ, Brazil

Address for correspondence:
Dr. Valmir C. Barbosa
IBM Rio Scientific Center
Caixa Postal 4624
20071 Rio de Janeiro - RJ
Brazil
tel.: +55-21-271-2303
fax: +55-21-271-2797
Internet: valmirb@riovmsc.vnet.ibm.com
Bitnet: valmirb@riosc

† Supported by a scholarship from CAPES, Ministry of Education, Brazil.
‡ On leave from COPPE/UFRJ, where he performed research for this paper.
Abstract. We consider the problem of determining the maximum flow in a flow network, and three distributed parallel algorithms that solve it. These are Awerbuch’s algorithm and two versions of Goldberg and Tarjan’s algorithm. Awerbuch’s algorithm and one of Goldberg and Tarjan’s are synchronous, while the other of Goldberg and Tarjan’s are asynchronous. In this paper we report on an experimental evaluation of the three algorithms on an 8-node T-800 Transputer hypercube. For this evaluation we have employed Awerbuch’s synchronizer $\sigma$ to synchronize the synchronous algorithms, thereby adding to the interest of the investigation, as the three algorithms end up with the same asynchronous-time and message complexities. Experiments were performed on various classes of flow networks, each emphasizing different aspects of relevance. The overall trend in the performance of the three algorithms is that Awerbuch’s tends to do better, and between Goldberg and Tarjan’s the synchronous algorithm is the fastest.

Keywords. Maximum-flow algorithms, distributed parallel algorithms, synchronous algorithms, asynchronous algorithms, hypercubes.

1. Introduction

Consider the directed graph $G = (V, E)$ and for all $u, v \in V$ let a nonnegative capacity $c(u, v)$ be such that $c(u, v) = 0$ if $(u, v) \notin E$. Throughout this paper, we let $n = |V|$ and $m = |E|$. Assume that $G$ has two distinguished vertices, a source $s$ and a sink $t$, such that every other vertex in $V$ lies on a directed path from $s$ to $t$ ($G$ is therefore connected). $G$ is called a flow network, and a flow in $G$ is a function $f : V \times V \rightarrow \mathbb{R}$ satisfying the following three properties:

(i) $f(u, v) \leq c(u, v)$ for all $u, v \in V$;

(ii) $f(u, v) = -f(v, u)$ for all $u, v \in V$;

(iii) $\sum_{v \in V} f(u, v) = 0$ for all $u \in V - \{s, t\}$.

The value of a flow $f$, denoted by $F$, is

$$F = \sum_{v \in V} f(s, v).$$

The maximum-flow problem asks for a flow $f$ of maximum value, and can be solved by a variety of algorithms (see [8] for basic concepts and [1.15] for two very good surveys).

For $u, v \in V$, the residual capacity of $(u, v)$ given a flow $f$ is

$$c_f(u, v) = c(u, v) - f(u, v).$$

2
The residual network of $G$ given $f$ is the directed graph $G_f = (V, E_f)$, where $E_f$ is the set of pairs $(u, v) \in V \times V$ such that $e_f(u, v) > 0$. A directed path from $s$ to $t$ in $G_f$ is called an augmenting path. Finally, if $f$ does not satisfy property (iii), but rather the weaker property that

$$\sum_{r \in V} f(u, r) \leq 0$$

for all $u \in V - \{s, t\}$, then it is called a preflow. In this case, there is an excess flow coming into $u$, denoted by $\epsilon_f(u)$ and given by

$$\epsilon_f(u) = -\sum_{r \in V} f(u, r).$$

In this paper we describe the results of an experimental evaluation of three distributed parallel algorithms for the maximum-flow problem. One of them is due to Awerbuch [4] and the other two are versions of a generic algorithm of Goldberg and Tarjan [14]. The interest in performing an experimental comparison of these three algorithms stems primarily from two motivations. First, Awerbuch’s algorithm employs the concept we just discussed of an augmenting path, originally introduced by Ford and Fulkerson [8,13], and of a layered residual network, introduced by Dinic [11,12], while Goldberg and Tarjan’s two algorithms are based on Karzanov’s concept of a preflow [8,17] we also discussed. As a consequence, our experimental study is in the first place promoting a comparison of two fundamentally different techniques for finding maximum flows. Secondly, two of these algorithms (Awerbuch’s and one of Goldberg and Tarjan’s) were given under the assumption of a synchronous model of distributed computation, while the other utilizes an asynchronous model. Interestingly, as we discuss in more detail in Section 2, once the three algorithms are adapted to run in the same computational environment, the resulting computational complexities are the same for all three, which supports even further the interest in a comparative experimental evaluation.

The following is how the balance of the paper is organized. Section 2 contains a description of the synchronous and asynchronous models of distributed parallel computation and a discussion of the complexities of the three maximum-flow algorithms under study. In Section 3 we provide a description of the three algorithms, and in Section 4 introduce the computational environment utilized in performing our experimental evaluation. This environment embodies an S-node T-800 Transputer hypercube [2] and a modular software package for performing point-to-point and group communication functions [5]. Section 5 contains a detailed description of the experiments performed and their results. We then give in Section 6 a summary of conclusions.
2. Models of Computation

When designing algorithms for distributed-memory machines, in which processors must communicate by message passing, it is customary to adopt two complexity measures, one that accounts for the number of messages exchanged until the end of the algorithm, and one that accounts for the time elapsed in the same period. There are essentially two models of distributed computation that can be used when the target machine offers point-to-point communication, namely the fully synchronous (synchronous, for short) and fully asynchronous (asynchronous, for short) models [18].

In the synchronous model, processors can be thought of as having access to a global clock that drives their computation in lock step. This global clock issues pulses, and at each pulse a processor may perform local computation (assumed to take no time) and send messages to its neighbors (i.e., the other processors that can be reached via a point-to-point communication channel). The delay that these messages suffer to be delivered is in the synchronous model no greater than the duration of a clock pulse. So a message sent by a processor to one of its neighbors at the beginning of a clock pulse is guaranteed to arrive before the beginning of the next clock pulse. Both complexity measures have a simple definition under the synchronous model. The message complexity requires that we count the total number of messages exchanged between neighbors, while the time complexity is given simply by the number of clock pulses necessary for the algorithm to terminate.

The asynchronous model is more realistic, inasmuch as it bears closer resemblance to most of the actual distributed-memory parallel machines. Processors are in this model driven by local, independent clocks. In addition, messages sent between neighbors are delivered within a finite, although unpredictable, time. Evaluating the message complexity of an algorithm under this model is as simple as in the synchronous case: one simply counts the total number of messages exchanged. The time complexity, on the other hand, requires a slightly more elaborate definition. In this context, though, it suffices that we state that the time complexity under an asynchronous model is the length of the longest “causal chain” in the computation, where each element in this chain is to be understood as the reception of a message and, as a consequence thereof, the sending of one or more messages by a processor (this notion is based upon the observation that, in an asynchronous model, a processor participates in an algorithm either by initiating spontaneously or by responding to the reception of messages).

It is usually considerably simpler to design an algorithm under the assumption of a synchronous model than an asynchronous one. However, as the great majority of distributed-memory parallel machines is in fact constituted by asynchronous machines, one needs a mechanism to perform the “translation” to the asynchronous environment of the algo-
rithm originally designed to operate under the assumptions of a synchronous model. Such a mechanism is the so-called *synchronizer*, introduced and developed by Awerbuch [3]. The essential property that a synchronizer seeks to preserve is that no processor will proceed to a further pulse in its computation before every message sent to it during the previous pulse by its neighbors has indeed reached it. Under very general assumptions, a synchronizer utilizes two basic mechanisms to ensure that the aforementioned property is preserved. First, every message is acknowledged, so that a processor can detect that every message sent by it during a certain pulse of the computation was received, at which moment the processor is said to be *safe* with respect to that pulse. Secondly, a processor may only proceed to the next pulse when it detects that all of its neighbors are safe with respect to the current pulse.

There are two types of overhead that one necessarily incurs when utilizing a synchronizer, namely the time and the communication overheads. The time overhead refers to the additional amount of time (in the asynchronous sense) introduced per pulse of the synchronous algorithm. The communication overhead is the additional number of messages introduced per pulse of the synchronous algorithm. Different synchronizers are obtained as one trades off these two types of overhead. One of the simplest synchronizers, the so-called synchronizer $\alpha$, is also the one with lowest time overhead, $O(1)$, although a message overhead proportional to the number of point-to-point communication channels is incurred. Under the synchronizer $\alpha$, a processor, upon becoming safe, simply reports this fact to its neighbors. Upon receiving similar reports from all of its neighbors, it then proceeds to the next pulse.

The communication overhead of synchronizer $\alpha$ can be significantly reduced in many cases. Suppose for example that our synchronous algorithm is such that every processor sends exactly one message to every one of its neighbors in most of the pulses. In this case no acknowledgement messages are any longer needed, and the functioning of synchronizer $\alpha$ is simplified to the following. In a pulse at which no message would be sent to a certain neighbor, send it a special signal. In order to proceed to the next pulse, a processor waits to receive one message of the computation or a special signal from all of its neighbors. As we shall see in the sequel, this may be the case with our maximum-flow algorithms.

In the remainder of this section, and throughout Section 3, let us suppose that our algorithms for maximum-flow computation utilize distributed models that mimic exactly the structure of the graph $G$. In other words, there is one processor per vertex in $V$ and one bidirectional communication channel per edge in $E$. The per-pulse overheads of synchronizer $\alpha$ are then $O(1)$ for asynchronous time and $O(m)$ for number of messages.

The following are the maximum-flow algorithms we investigate in this paper, along
with their time and message complexities.

- **Algorithm SA** (Awerbuch [4]): $O(n^2)$ synchronous time and $O(n^3)$ messages;

- **Algorithm SGT** (Goldberg and Tarjan [14]): $O(n^2)$ synchronous time and $O(n^3)$ messages;

- **Algorithm AGT** (Goldberg and Tarjan [14]): $O(n^2)$ asynchronous time and $O(mn^2)$ messages.

Algorithms SA (Synchronous-Awerbuch) and SGT (Synchronous-Goldberg-Tarjan) are given for a synchronous model of computation, while algorithm AGT (Asynchronous-Goldberg-Tarjan) assumes an asynchronous model. Suppose we apply synchronizer $a$ to algorithms SA and SGT. As this synchronizer introduces a constant asynchronous-time overhead per pulse of the synchronous algorithm, the resulting algorithms, call them $\alpha$SA and $\alpha$SGT, respectively, have an $O(n^2)$ asynchronous-time complexity. On the other hand, considering the $O(m)$ message overhead introduced per pulse of the synchronous algorithm by synchronizer $a$, both $\alpha$SA and $\alpha$SGT have a message complexity given by $O(n^3) + O(n^2)O(m)$, that is, $O(mn^2)$, as $G$ was in Section 1 assumed to be connected. We then have two additional algorithms for operation under the asynchronous model:

- **Algorithm $\alpha$SA**: $O(n^2)$ asynchronous time and $O(mn^2)$ messages;

- **Algorithm $\alpha$SGT**: $O(n^2)$ asynchronous time and $O(mn^2)$ messages.

Interestingly, all three of algorithms $\alpha$SA, $\alpha$SGT, and AGT have the same asynchronous-time and message complexities. We proceed in the next section to a brief description of them.

### 3. Three Distributed Parallel Algorithms

As we remarked in the previous section, the following three algorithms are described under the assumption of one processor for each vertex in $V$ and one bidirectional communication channel for each edge in $E$. We shall then refer to a processor and its corresponding vertex, as well as a channel and its corresponding edge, interchangeably.

**Algorithm $\alpha$SA**

Algorithm SA is an adaptation to a distributed model of the algorithm given by Shiloach and Vishkin [19] as a parallel version of Dinic’s algorithm [11] for a shared-memory synchronous model [16]. It starts with any flow, for example by assigning zero flow to all edges, and then builds on top of it until a maximum flow is found.
The essence of algorithm SA is the following. It proceeds in iterations, and at each iteration a layered residual network is built. A maximal flow is then found on this network and then added to the cumulative flow that is maintained throughout the iterations. A flow in the layered residual network is said to be *maximal* when it is equal to the residual capacity of at least one edge on every $s$-to-$t$ path. When a layered residual network that includes $t$ can no longer be found the flow is maximum and the algorithm terminates.

The layered residual network is built at each iteration as follows. Let $f$ be the cumulative flow obtained at the end of the previous iteration (the initial flow, for the first iteration). The source $s$ is included in the first layer and a process similar to breadth-first-search is started to determine the subsequent layers. For $k > 1$, the $k$th layer contains every vertex $v$ such that $v$ is not in any of the previous $k-1$ layers and, furthermore, there exists a vertex $u$ in the $k-1$st layer such that $c_f(u, v) > 0$. The synchronous distributed algorithm to build the layered residual network is then very simple. For $k \geq 1$, the $k$th layer is determined at the $k$th pulse as follows. Those vertices $u$ belonging to the $k-1$st layer send a message to their neighbors $v$ such that $c_f(u, v) > 0$. In the next pulse, $v$ replies positively or negatively to $u$, depending on whether it had already been included in a layer at any of the previous pulses.

Once the layered residual network has been constructed based on a flow $f$, a maximal flow on it is determined by a process that is started at $s$ by assigning to each $(s, v) \in E_f$ a flow equal to $c_f(s, v)$, thereby providing $v$ with a positive excess flow. This process continues on to the succeeding layers, and along the way the excess flow at the vertices is either pushed to the next layer or returned to the previous one (and then possibly re-routed through other edges). Termination occurs when no vertex can take any additional flow. The synchronous distributed algorithm to find a maximal flow on the layered residual network works by sending flow between neighbors in the form of messages. Whenever flow is received at $v$ from $u$ on edge $(u, v)$, the amount of flow received is pushed onto a stack along with the identification of its sender, $u$. At each pulse, a vertex $v$ may receive flow from $u$ on edge $(u, v)$ or on edge $(v, w)$ (this is returned flow). At the beginning of the next pulse, all the flow $v$ received is either sent to the succeeding layer, if at all possible, or returned to the previous one, in this case by popping the amount of flow to be returned and its destination off the stack. In case no more flow can be sent to the next layer, $v$ informs its neighbors in the preceding layer that it is *blocked*, so no further attempts will be made to send flow to it in the remainder of the iteration.

This completes our brief description of algorithm SA. As we mentioned earlier, obtaining $aSA$ is now a simple matter: if, at any pulse, no message is sent in SA by a vertex to one of its neighbors, then in $aSA$ a special signal is sent.
Algorithm $\alpha$SGT

Algorithm SGT is conceptually a lot simpler than algorithm SA. It works with the notion of preflows, and continually tries to push excess flows along the edges of the residual network that the vertices estimate to be on the shortest paths to $s$ or $t$. The algorithm starts with a preflow $f$ such that $f(s, v) = c(s, v)$ for all $(s, v) \in E$ and $f(u, v) = 0$ for all $(u, v) \in E$ with $u \neq s$.

Every vertex $u$ maintains an estimate $d(u)$ of its shortest distance to either $s$ or $t$ in the residual network. Any initial values for these estimates will do, as long as $d(s) = n$, $d(t) = 0$, and $d(u) \leq d(v) + 1$ for all $(u, v) \in E$. In the first pulse of the algorithm these estimates are exchanged between neighbors. At all times during the execution of the algorithm, these estimates are such that either $d(u)$ is a lower bound on the distance from $u$ to $t$, if $d(u) < n$, or $d(u) - n$ is a lower bound on the distance from $u$ to $s$, if $d(u) \geq n$.

At each pulse of SGT the active vertices attempt to get rid of their excess flow by pushing it in the direction of $s$ or $t$. Letting $f$ be the preflow at the end of the previous pulse (the initial preflow, in the first pulse), a vertex $u$ is said to be active if $u \in V - \{s, t\}$ and $\epsilon_f(u) > 0$. An active vertex $u$, at the current pulse, first sends an amount of flow equal to

$$\min\{\epsilon_f(u), \epsilon_f(u, v)\}$$

to a neighbor $v$ such that $d(u) = d(v) + 1$ and $\epsilon_f(u, v) > 0$, and updates $f$ (consequently, $\epsilon_f(u)$ and $\epsilon_f(u, v)$) accordingly. This is repeated until either $\epsilon_f(u) = 0$ or $\epsilon_f(u, v) = 0$ for all $v$ such that $d(u) = d(v) + 1$. If after this $\epsilon_f(u) > 0$, then $d(u)$ is updated to

$$\min_{v | \epsilon_f(u, v) > 0} \{d(v) + 1\},$$

and this value, if different from the previous one, is sent to $u$'s neighbors. The pulse is terminated by adding to $\epsilon_f(u)$ all the flow received during the pulse. The algorithm terminates when no vertices are any longer active.

As in the case of the previous algorithm, the use of special signals makes it straightforward to obtain $\alpha$SGT from SGT.

Algorithm AGT

The essential difficulty with algorithm SGT in an asynchronous environment is that the condition that $d(u) = d(v) + 1$, necessary for $u$ to send flow to $v$, cannot be trivially ensured, as the values of $d(u)$ and of $d(v)$ may differ substantially at $u$ and $v$. The solution adopted when proposing algorithm AGT has been that every flow sent between two vertices $u$ and $v$ must carry the value of $d(u)$ and be explicitly accepted or rejected by $v$, and only then additional flow may be sent.
When \( v \) receives flow from \( u \) and verifies that in fact \( d(u) = d(v) + 1 \), then the flow is accepted and this is reported back to \( u \). If, on the other hand, \( d(u) \neq d(v) + 1 \), then the flow is rejected and this is reported back to \( u \) along with the value of \( d(v) \). Upon receiving this rejection message, \( u \) updates \( e_f(u), c_f(u,v), d(v) \), and possibly \( d(u) \). Whenever \( d(u) \) changes, its new value is reported to all of \( u \)'s neighbors.

4. Implementation Environment

The experimental evaluation we report on in this paper was carried out on an NCP-I machine, which is a Transputer-based parallel machine built at COPPE/UFRJ [2]. The prototype we used has currently eight T-800 processors interconnected as a hypercube.

All the programs for maximum-flow calculation were written in Occam2 [7]. If, on the one hand, Occam2 offers very adequate programming structures for the development of message-driven programs, on the other hand some of its peculiarities make it a potentially very difficult endeavor to carry out the development of a program to the end. One issue of particular relevance in this respect is the fact that all interprocess communication channels are in Occam2 assumed to have zero capacity, meaning that the end processes of a channel have to synchronize to exchange information. Readily, such a mechanism is very prone to the occurrence of deadlocks, thence what we believe to be the main difficulty with Occam2 programming.

In writing the programs for maximum-flow calculations we have adopted the methodology for Occam2 programming presented in [6], according to which the program is initially specified as if all interprocess communication channels had positive (in fact, as large as needed) capacities, and then rearranged by a series of potentially automatic procedures whereby the actual zero capacity of channels is employed in a way that no deadlock is any longer possible.

This methodology, in essence, consists of three main steps along which the various sources of improper behavior are dealt with. Here we give a very brief description of these main constituents. The program given as input, written as if channels had sufficient capacity to accommodate the necessary message traffic among processes, is viewed as a topmost layer in the final software. The first step is to calculate the actual number of buffers that, according to certain optimization criteria, will ensure the absence of deadlocks even if some channels are assigned less capacity than needed. This set of buffers is then included in the Occam2 program and constitutes its second layer. At this point, before the third layer is introduced, comes the second step, which is a task allocation or data partitioning step. The result of this step is a decision of which processors will run which parts of the program built so far. The third step, to finalize, is the inclusion of the third (and
lowest) layer, which is a set of communication procedures designed to provide deadlock-free routing among the processors of the system, and several other group communication functions [5]. Of course this methodology is not to be used blindly (except possibly for very large programs), and functions mainly as a set of guidelines to be observed.

The resulting Occam2 program is guaranteed to be deadlock-free if the original program (written under the sufficient-capacity assumption) is deadlock-free (that is, no deadlock is added in transforming the original program into an actual Occam2 program). In this final program each processor lodges a set of Occam2 processes. Each of these processes can communicate with any of the other processes it was originally designed to communicate with, and no deadlock occurs in this communication. The same continues to hold if messages are sent between processes that do not reside in a same processor. Such messages are routed automatically to the proper destination, and no deadlock arises in this phase either.

We now give a brief description of how these techniques were employed in writing our programs for maximum-flow calculations. The first issue we discuss is the partitioning of the graph $G$ among the processors. Recall from previous sections that we described the maximum-flow algorithms as if there were one processor per vertex in $V$ and one bidirectional communication channel per edge in $E$. After $G$ is partitioned among the processors this view is no longer valid. The result of this partitioning is the structure of the initial Occam2 program (a "task graph"), in which there is only one process per processor of the actual machine (eight in our case). Each of these processes is responsible for a subset of $V$, and between two processes a communication channel exists if and only if they are responsible for vertices that are connected to each other in $G$. For all the results we report on in the next section the partition of $G$ was done arbitrarily and uniformly among the processors, except for the vertices $s$ and $t$, both allocated to the processor that is in NCP-I connected to the host machine (this is the root processor).

In the implementation of the two synchronous algorithms, $\alpha$SA and $\alpha$SGT, synchronization messages are only necessary between processors that lodge neighboring vertices of $G$. In these two cases, in addition, global termination is implemented by the process residing in the root processor as well, to which the other processors report at the end of every pulse. When this process detects termination of the maximum-flow computation, it then broadcasts a termination order to the others.

The asynchronous algorithm, AGT, in its turn, employs Dijkstra and Scholten's algorithm [9] for detecting global termination. This termination detection approach is well suited to the so-called diffusing computations, which are distributed computations in which processors do some effective computation upon the receipt of a message only (except pos-
sible at the beginning). This is clearly the case of algorithm AGT, so the approach is applicable. The algorithm of Dijkstra and Scholten requires that all messages be acknowledged (these acknowledgements are not to be confused with the messages used in algorithm AGT for accepting and rejecting flow), and maintains a dynamic tree on the graph formed by the interconnected processors along which termination signals are sent. When these signals reach the processor that originated the computation (in our case the root processor, since it lodges vertex s), this processor knows that the computation has terminated globally and may then do a broadcast of this.

5. Experimental Evaluation

In this section we describe the results of our experiments with algorithms aSA, aSGT, and AGT within the environment described in the previous section. Our results are partitioned into two main groups. In the first group we show the behavior, when running on eight processors, of the three algorithms when presented with various types of flow networks. The second group of results shows the speedup of the three algorithms for two, four, and eight processors. All data in Figures 1 through 7 correspond to averages over eight instances of the same problem. In these figures, T stands for the time for completion of the maximum-flow computation.

The tests whose results we report were carried out following the guidelines found in the documentation of the First DIMACS International Algorithm Implementation Challenge [10].

The first set of experiments is aimed at investigating the behavior of the three algorithms when subject to flow networks of different densities. An instance in this case is generated by arranging the vertices in \( V = \{s, t\} \) into an \( r \times r \) square matrix. An edge \((s, v)\) is then created for every vertex \( v \) in the first column of the matrix, and similarly an edge \((u, t)\) for every vertex \( u \) in the last column of the matrix. Inside the matrix, an edge \((u, v)\) is created between every vertex \( u \) in column \( c \) and \( od \) vertices \( v \) in column \( c + 1 \), for all \( 1 \leq c \leq r - 1 \). The density of the flow network is then given by \( od \), the out-degree of a vertex in \( V - \{s, t\} \). For an edge \((u, v)\) inside the matrix, we let \( c(u, v) \) be an integer value chosen randomly between 0 and \( c^* \), a maximum capacity parameter. Edges emanating from \( s \) or converging to \( t \) are assigned capacity \( 3c^* \).

We show in Figure 1 the behavior of the three algorithms for \( r = 5, 8, 11, 16, 22 \) and \( c^* = 10^4 \). For each value of \( r \) we have \( od \in \{2, 4, \ldots, 2^{\log_2 r}\} \). The overall behavior depicted in Figure 1 for the three algorithms is that they tend to perform better as the density increases for a same value of \( n - 2 \). This is so because in these situations less and less flow has to be returned toward \( s \). An exception to this behavior occurs for some
values of $n - 2$ and very low densities, where a slight increase in $T$ is observed. This can be accounted for by the observation that, in situations of very low density, the flow to be returned toward $s$ is identified quite early in the computation, so the algorithms tend to be faster.

Most of the data shown in Figure 1 indicate a clear superiority of $\alpha$SA over $\alpha$SGT, and of this one over AGT. There is also an indication that, for large graphs with high density, the three algorithms may behave approximately equivalently to one another. Figure 2 shows an enlargement of Figure 1 for $n - 2 = 256$ (with additional values of $od$).

![Figure 1](image)

**Figure 1.** Performance under varying density

Additional experiments have been performed to see how the algorithms behave when presented with flow networks of very high density (in the experiments we just reported the maximum out-degree is bounded by $r$, so $G$ cannot get truly dense). In our experiments, such a dense network has been obtained by numbering the vertices from 0 through $n - 1$ and then creating an edge from vertex of number $i$ to each of the vertices of numbers
The relative performance of the three algorithms has in this case been the same as in the previous density-related cases. In addition, $T$ increases steadily as $n$ increases.

In the second set of experiments we investigate the behavior of the algorithms when the graph’s length and width vary. As with the previous set of experiments, the vertices in $V \setminus \{s, t\}$ are arranged in a matrix, this time a $w \times \ell$ matrix, where the number of rows, $w$, is $G$’s width and the number of columns, $\ell$, its length. An edge $(s, v)$ exists for every $v$ in the first column and an edge $(u, t)$ exists for every $u$ in the last column. Inside the matrix, an edge $(u, v)$ exists for all $u$ in column $c$ and three vertices $v$ in column $c + 1$, where $1 \leq c \leq \ell - 1$. Edges $(u, v)$ inside the matrix are assigned an integer capacity $c(u, v)$ chosen randomly between 0 and a maximum $c^*$. All other edges receive capacity $3c^*$.

In Figure 3 we show the behavior of the three algorithms for $n = 2 = 16, 32, 64, 128, 256, 512$ and $c^* = 10^4$. For each value of $n = 2$ we have $\ell \in \{2, 4, \ldots, (n - 2)/4\}$. Figure 3 indicates that in general the three algorithms tend to perform more slowly as the graph’s length is increased for a same value of $n - 2$. This is expected in view of the fact that it takes longer for flow to reach $t$ from $s$ as $\ell$ becomes larger. An exception to this
general observation is the case of $\alpha$SA, which seems to perform better as $\ell$ is increased. The reason for this is that, as $\ell$ is increased, $n$, the graph's width, is decreased as $n$ is kept constant. Consequently, the number of layered residual networks built by $\alpha$SA tends to decrease and the algorithm becomes faster. Evidence for this conclusion is provided by Figure 4, where, for the same range of parameters used in Figure 3, we show the number of layered residual networks built by $\alpha$SA. What Figure 4 shows is that this number tends to be greatest for a given value of $n - 2$ when $\ell$ is such that the matrix is approximately square.

As in the previous set of experiments, here too there seems to be a superiority of $\alpha$SA over $\alpha$SGT, and of $\alpha$SGT over AGT. For very wide networks (very low $\ell$), $\alpha$SA and $\alpha$SGT seem to perform equivalently to each other. Also, as $n$ and $\ell$ both increase, apparently $\alpha$SGT and AGT tend to come closer to each other in their performances.

![Graph showing performance under varying width and length](image)

**Figure 3.** Performance under varying width and length

In a third set of experiments, we have investigated the influence of the value $c^*$ of the
Figure 4. Number of layered residual networks built by αSA under varying width and length

maximum capacity on the performance of the three algorithms. For such instances were generated in entirely the same fashion as in the second set of experiments. The results of those experiments have been that, for $c^* = 10^2$ and $c^* = 10^8$, the difference in the performance of the three algorithms seems to be negligible.

The remaining three figures, Figures 5 through 7, show the speedup on two, four, and eight processors of the algorithms αSA, αSGT, and AGT, respectively. (The notion of speedup we have employed considers the time on one processor of the same algorithm that is run on more processors.) The input graphs to the algorithms have been in this case the same as for the first set of experiments, with $r = 5, 8, 11, 16$ and $od = 4$. As one notices, both αSA and αSGT tend to do well as $n$ increases. This is not the case, though, with AGT, whose speedup is much less than the ideal, although it too increases with $n$. 

15
6. Concluding Remarks

We have in this paper described an experimental evaluation of three distributed parallel maximum-flow algorithms, namely Awerbuch's algorithm and two versions of Goldberg and Tarjan's algorithm. Awerbuch's algorithm and one of Goldberg and Tarjan's were originally given under the assumption of a synchronous model of distributed computation (unlike the other of Goldberg and Tarjan's algorithms, given for an asynchronous model); so before an implementation can be carried out they have to be synchronized.

The interest in comparing these three algorithms stems primarily from two sources. First, Awerbuch's algorithm employs the concepts of augmenting paths and layered residual networks, while Goldberg and Tarjan's algorithms are based on the notion of preflow. We are then promoting the comparison of two very different classes of algorithms for maximum-flow computation. Secondly, if when synchronizing the two synchronous algorithms we employ Awerbuch's synchronizer $\alpha$, then the resulting asynchronous algorithms have the same asynchronous-time and message complexities, and these are in turn the same as Goldberg and Tarjan's asynchronous algorithm's. A comparative study of the
three becomes then especially interesting, inasmuch as it involves evaluating how far their predicted asymptotic performance is from their behavior in practice.

We have named the three algorithms αSA, αSGT, and AGT, respectively for Awerbuch's α-synchronized algorithm, Goldberg and Tarjan's α-synchronized algorithm, and Goldberg and Tarjan's asynchronous algorithm. Our experiments have all been carried out on an 8-node T-800 Transputer hypercube programmed in Occam2.

Experiments have been performed in which the algorithms were presented with different classes of graphs, each emphasizing one important aspect of flow networks. In general we have observed αSA to perform better than αSGT, and the latter to perform better than AGT.

There are two important observations concerning this general trend in the behavior of the three algorithms. The first observation is that αSA and αSGT have benefited greatly from the use of synchronizer α because the relation of problem size to number of processors \((n/8)\) has in our experiments tended to stay relatively high. In such situations, in almost every pulse there is flow to be shipped among the processors, so the communication overhead introduced by synchronizer α is negligible. Had \(n/8\) had a generally low value.
there would have been more pulses in which more processors would have no flow to ship, so the message overhead of synchronizer \( \alpha \) would begin to show.

The second important observation concerns the behavior of Goldberg and Tarjan's algorithms with respect to Awerbuch's. The versions of the former algorithms we used are derived from a generic algorithm that Goldberg and Tarjan gave. Other versions, of better asymptotic complexities or even exploiting other properties of the general method, can be obtained and may behave significantly differently from the ones we implemented.
References


