DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Implementing the Push-Relabel Method for the Maximum Flow Problem on a Connection Machine

FARID ALIZADEH AND ANDREW V. GOLDBERG

December 1992

1. Introduction

In this paper we study the recent *push-relabel* [**9, 13**] class of algorithms for the maximum flow problem in capacitated directed networks. Several researchers [**1, 5, 7, 15**] concluded that their sequential implementations of push-relabel algorithms are superior to previously used codes. (For more recent results, see [**3, 21**].) We focus on the behavior of the push-relabel method in massively parallel environments. For a related work in the context of minimum-cost flow problem, see [**20**].

The push-relabel method is the first theoretically efficient algorithm for the maximum flow problem that is potentially practical; the previous algorithm of Shiloach and Vishkin [**23**], based on Dinitz' blocking flow method [**8**], requires the amount of memory quadratic in the number of nodes in the input network.

The push-relabel method uses two operations, *push* and *relabel*, to manipulate current flow and distance labeling functions. In the parallel implementation the algorithm pushes simultaneously from all nodes to which the push operation applies; we refer to this method of pushing as *parallel push*. Similarly, *parallel relabel* applies relabeling operation to all eligible nodes simultaneously.

1991 *Mathematics Subject Classification.* Primary 90B10; Secondary 68Q20.
The first author was supported in part by by the Air Force Office of Scientific Research grant AFOSR-87-0127, the Minnesota Supercomputer Institute and NSF postdoctoral grant number CDA-9211106. The second author was supported in part by ONR Young Investigator Award N00014-91-J-1855, NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T and DEC, Stanford University Office of Technology Licensing, a grant form Mitsubishi Electric Laboratories, and a grant from the Powell Foundation.

The push-relabel algorithm works regardless of the order in which the push and relabel operations are applied. It also works with any correct labeling procedure. This robust nature of the algorithm allows us to experiment with various strategies for ordering push and relabel operations, and with a variety of relabeling procedures. All of the push and relabel operations are suitable for parallelization. The goal is to determine the ordering and relabel procedures most suitable for a practical parallel implementation.

It is well-known that the maximum flow problem is P-complete [**14**]. Therefore, for any proposed parallel algorithm one may find instances of the problem which take at least linear time in the the size of input network, unless P=NC. Our goal is to find techniques that perform substantially better than linear time on most networks in a fine grain parallel environment.

This paper is a report on experiments with various strategies for the parallel push-relabel algorithm. The experiments were conducted on a Connection Machine [**16**] model CM-2, with 32K processors (K=1024). Each processor can directly access 1 megabits of local memory, so the entire system has 4 gigabytes of memory. The starting point for our work was the Connection Machine implementation of the algorithm described in [**10**]. Our current implementation, however, improves the original one in several areas, including better ordering and pipelining of operations of messages. Also, the original implementation was on CM-1 using the *LISP interpreter where as the current implementation is on CM-2 using the C* compiler.

This paper contains six sections including the introduction. In Section 2 we outline the push-relabel algorithm and discuss parallel implementation of push and various relabel operations including Derigs and Meier's gap-relabeling and parallel breadth–first search. In Section 3 we describe the mapping of networks to the Connection Machine processors. This mapping is designed to take advantage of the parallel prefix and suffix operations provided in the Connection Machine. In Section 4 we describe in more detail parallel push and relabel operations in the context of this mapping. In Section 5 we report on the running time of the program on various sample inputs. We present our conclusions in Section 6.

2. The Parallel Push-Relabel Algorithm

In this section we review some of the basic concepts of the push-relabel method and its parallel implementation. We assume that the reader is familiar with [**13**]. (See also [**12**].) Because of high-grain parallelism of the CM-2 architecture, the goal of our implementation is to get a tight inner loop of the algorithm rather then to achieve high processor utilization by careful processor scheduling (see [**11, 23**]).

A *flow network* is a directed graph $G = (V, E, s, t, u)$, where V and E are node set and arc set, respectively; s and t are the source and the sink, respectively; and u is a non-negative capacity function on the arcs. We define $n = |V|$ and $m = |E|$, and assume that for each arc (v, w), the arc (w, v) is also present. A flow is a function on the arcs that satisfies capacity constraints on all arcs and conservation constraints on all nodes except the source and the sink. The conservation constraint at a node v indicates that the *excess* $e_f(v)$, defined as the difference between the incoming and the outgoing flows, is equal to zero. A *preflow* [**17**] satisfies the capacity constraints and the relaxed version of conservation constraints that requires the excesses to be nonnegative.

An arc is *residual* if the flow on it can be increased without violating the capacity constraints, and *saturated* otherwise. The residual capacity $u_f(v, w)$ of an arc (v, w) is the amount by which the arc flow can be increased. The residual graph is induced by the residual arcs.

The *distance labeling* $d : V \to N$ satisfies the following conditions: $d(s) = n$, $d(t) = 0$, and for every residual arc (v, w), $d(v) \leq d(w) + 1$. It is easy to see that if $d(v) < n$, then $d(v)$ is a lower bound on the actual distance from v to t in the residual graph, and if $d(v) > n$, then $d(v) - n$ is a lower bound on the actual distance of v to s. A residual arc (v, w) is *admissible* if $d(v) > d(w)$.

A push-relabel algorithm maintains a preflow and a distance labeling. We say that a node v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. The preflow is modified using push operation, which pushes excess flow from an active node to an adjacent one that has a smaller distance label. When an active node cannot push its excess because its label is at most that of its neighbors, then the distance labeling is modified using a relabel operation, which increases the distance label of the node to the largest value allowed by the labeling constraints. Distance labels also define a *$s - t$-cut set* with respect to the current preflow. The set of nodes is partitioned into two sets by the distance labels: those nodes whose labels are larger than or equal to n, and those whose labels are less than n. When the algorithm terminates the final labeling determines the minimum cut set.

Our parallel implementation works as follows. The preflow f is initialized to $f(s, w) = u(s, w)$ for all nodes u adjacent to s, and $f(v, w) = 0$ for all $v \neq s$. The initial distance labeling is computed using breadth-first search (BFS) on the residual graph induced by the initial preflow. The preflow and distance labeling are updated in *pulses* using the push and relabel operations, respectively, until no active nodes remain. Each pulse consists of one or more parallel push operations followed by one or more parallel relabel operations.

The *parallel push* operation works as follows. For active nodes, the parallel push operation distributes their excess among admissible outgoing arcs as described in Figure 1; in this figure, "push flow from v to w" means increase the flow from v to

OPERATION ***Parallel-Push***
For all active nodes v in parallel **do**
    push flow from v **until** $e_f(v) = 0$ **or** $\forall w$ such that $d(w) < d(v)$, $u_f(v, w) = 0$.

FIGURE 1. The parallel push operation.

OPERATION ***Relabel***
For **all** nodes $v \notin \{s, t\}$ in parallel **do**
$d'(v) \leftarrow \min\{d(w) + 1 : u_f(v, w) > 0\}$;
If $d(v) \neq d'(v)$ **then**
broadcast $d(v)$ to all neighbors of v;
end;

FIGURE 2. The parallel relabel operation.

OPERATION ***gap-relabel***
(1) Find the smallest g where $0 < g < n$ and no node has label g.
(2) For all nodes v in parallel do:
if $g < d(v) < n$ set $d(v) \leftarrow n$.
(3) For all v which have a new label broadcast the new labels to all neighbors of v.

FIGURE 3. The gap-relabel operation.

w by the minimum of $e_f(v)$ and $u_f(w)$ and update the excesses of v and w.

For parallel relabeling we use several alternatives in our implementation. The simple parallel relabel, described in Figure 2, sets the distance label of every node except s and t to one plus the minimum distance label of its residual neighbors.

Another relabeling procedure is the *gap–relabel* operation of Derigs and Meier [**7**] based on the following observation. Let g be an integer and $0 < g < n$. Suppose at certain stage of the algorithm some nodes are labeled 0, some labeled 1, and so on, through $g - 1$, but no node is labeled g—although some other nodes may be labeled by integers between g and n. Derigs and Meier observe that the sink is not reachable from any of the nodes whose labels are strictly between g and n. Therefore, the labels of such nodes may just be increased to n. This procedure may be implemented in parallel; see Figure 3.

Note that the most accurate labeling is obtained by applying BFS backwards from the sink and forward from the source. This can be implemented by applying parallel relabel operation repeatedly until distance labels stop changing. Doing this during every pulse is too expensive. The implementation of [**10**] performs a breadth-first search once after a sequence of pulses so that the breadth-first search time can be amortized over the pulse time. We experimented with this approach as well as with using the gap-relabel operation instead.

No theoretical result suggests that gap-relabel and BFS operations improve the worst-case time bounds on the algorithm. In practice, both periodic BFS and gap-relabel operations reduce the number of pulses drastically. The gap-relabel operation, however, is much less expensive (usually as fast as simple parallel relabel in our implementation) and as a result gives better running times most of the time in spite of the fact that the labeling after such an operation is not exact.

Our parallel implementation can be viewed as running the parallel push, relabel, and gap-relabel (or BFS relabel) processes “simultaneously”. However, due to the SIMD nature of the machine, we have to time-multiplex the operations. We can adjust relative speed of the operations by running several push operations followed by several relabel operations followed by a gap-relabel or a BFS relabel operation. (Note that it does not make sense to run two gap-relabel operations without running a simple relabel operation in between.) We can also pipeline some of the operations as described in section 4.2.

A *pulse* is a sequence of push and relabeling operations (possibly pipelined) that is repeated over and over. A general rule is that the closer the labels are to their actual distance to the sink, the fewer pushes will be performed. Therefore, the more accurate labeling may reduce the total number of pulses. However, maintaining a very accurate labeling is too expensive. Our strategy is to update labels as accurately as possible without increasing the time spent in each pulse by too much. In Section 4 we describe our implementation in more detail.

3. CM Architecture and Tools

In this section we review some aspects of the CM architecture relevant to our program. Then, we describe a set of useful operations available on the Connection Machine for accumulating in parallel partial sums, partial minimums, etc.

3.1. Connection Machine Architecture. The following is a brief outline of the CM architecture. For more details see the book by Hillis [**16**] and documents from the Thinking Machine Corporation, for instance [**6**].

The Connection Machine is a distributed memory parallel computer [**19**]. It consists of thousands of processors connected by a routing network. Each processor has local memory. The local memory of a processor can be accessed by other processors via the routing network. [1]

The CM is a *single instruction, multiple data* (SIMD) machine: the program is stored in a host computer which executes a sequential program containing parallel instructions. When a parallel instruction appears in the program, the host broadcasts it to all processors. Each processor, depending on its memory contents, either executes the instructions or remains idle. The operation of the machine is totally synchronous: the next instruction does not start until all processors have completed the execution of the current instruction.

Each processor can access its own memory, or it can access the memory of other processors. However, accessing the local memory is much faster than accessing other processors' memory. In general, the time required by a processor to access memory of a processor that is "close" (in the routing network) is less than the time required to access the memory of a processor that is far. If during the execution of an instruction several processors access memories of other processors, the longest

[1] Actually 16 processors are located on a processor chip, and every pair of processor chips (32 processors) share a memory chip. The entire machine may be thought of as a hypercube whose nodes are these pairs of processor chips. For simplicity in our discussion we assume that each processor has its own memory.

memory access time determines the execution time of the instruction. A *routing cycle* is the amount of time it takes to execute such an instruction. It is important to realize that the routing cycle time varies depending on the interprocessor communication pattern and the size of data being accessed.

The Connection Machine software also provides the notion of *virtual processors.* The user may request any number of processors he or she wishes. If this number is larger than the number of physical processors available, each physical processor simulates v virtual processors, where v is the VP ratio, that is $v = \lfloor \frac{\#\ of\ virtual\ processors}{\#\ of\ physical\ processors} \rfloor$.

We should mention that the Connection Machine processors are fairly simple *bit processors* and can operate only on one bit of information at a time. In contrast even the simplest personal computers operate on at least 8 bits of data at a time. Also, the Connection Machine clock operates at 10MHz (in contrast, even the slowest PC's have a clock rate of at least 16MHz). Therefore, in order for a program to run efficiently on the Connection Machine it must have very high grain parallelism and should keep a large number of processors busy at any given moment.

3.2. Parallel Prefix and Suffix Operations. Parallel prefix operations have been recognized as fundamental parallel operations, and their implementation and use in parallel algorithms has been widely studied; see *e.g.* [**18, 22**]. Given an associative binary operation "$*$" and a sequence of s numbers $a_1, a_2, \cdots, a_s$, the parallel prefix "$*$" operation maps this sequence into the sequence $a_1, a_1 * a_2, \cdots, a_1 * a_2 * \cdots * a_s$. Similarly the parallel suffix operation maps this sequence into $a_1 * a_2 * \cdots * a_s, \cdots, a_{s-1} * a_s, a_s$. In our implementation of the parallel push-relabel method, we use prefix and suffix operations with "$*$" replaced by addition, min, and copy.

An extended form of parallel prefix and suffix operations are *segmented parallel prefix* operation. In the segmented form a list of sequences $S_1, S_2, \cdots, S_l$ is mapped into $S'_1, S'_2, \cdots, S'_l$, where each sequence S'_i is obtained from the corresponding sequence S_i using the parallel prefix or suffix "$*$" operation. The lengths of sequences S_i may be different.

A sequence of numbers is stored in the Connection Machine by placing the entries of the sequence in contiguous processors. For simple operations such as addition, copying, and minimum, the time required to perform parallel prefix and suffix operations is of the same order of magnitude as the time required for one routing cycle on the data of the same size. Although the routing cycle time and the time to perform a parallel prefix operation vary depending on the communication pattern and on the type of the parallel prefix operation, we shall refer to these times as simply the routing cycle in our description of algorithms. The main point to remember is that a routing cycle is much larger than a local memory access.

4. Data Structures and Implementation Details

4.1. Parallel Implementation of the Pulse Procedure. We use a mapping of the input network to the machine as first described by Blelloch in [**4**]. This mapping allows us to use locality of data through the parallel prefix (suffix) operations. Each node v is assigned a processor P_v which we call a *node processor.* Each arc (v,w) is also assigned a processor P_{vw} called an *arc processor.* Recall that for each arc (v,w) we assume that the arc (w,v) is also present in the network and therefore, in our implementation, a separate arc processor P_{wv} is assigned to this arc. We call processors P_{vw} and P_{wv} *pair processors.* In the machine, each processor P_v is followed by all the processors P_{vw} corresponding to the arcs incident on it; the positions of arc processors associated with a node are arbitrary within themselves, and so is the positions of the node processors as long as the associated arc processors follow them. Each arc processor P_{vw} stores the processor address of its pair processor P_{wv}.

The part of the program that reads in the input, allocates processors, and initializes the system is relatively simple. The main part consists of application of the *pulse* procedure until no active nodes remain. The implementation of this procedure is summarized in Figure 4. This implementation include the simple *relabel* operations as part of it because all of the versions use this operation predominantly, although occasionally other relabel operations such as BFS or *gap-relabel* is used.

Steps 1–3 and 7 implement the parallel push procedure. In Step 1, the value of excess at each node processor P_v is sent to the arc processors immediately following it. Step 2 distributes the node excess to the outgoing arcs. First, each arc processor P_{vw} determines how much excess may be sent through its arc. This is equal to the residual capacity if $d(w) < d(v)$ (that is if w is estimated to be closer to sink) and zero otherwise. Then a *seg-suffix-add* is performed on these values. Now, each arc processor P_{vw} has information about the excess to be pushed from v, the amount it can push, and the amount that can be pushed through the arcs that follow (v,w) on the arc list of v. This information is enough to compute the amount $\sigma(v,w)$ to be pushed through the arc (v,w). After an execution of the *seg-suffix-add* operation, each node processor P_v contains the information about how much excess can be pushed from the node v at this pulse. The processor sets the value of its variable $e_f(v)$ to the amount that will remain after the pushing. Finally in step 3, all arc processors P_{vw} for which the amount $\sigma(v,w) > 0$, increase$f(v,w)$ by $\sigma(v,w)$, decrease $u_f(v,w)$ by the same amount, and send the value of $\sigma(v,w)$ to their pair processor P_{wv}. Each processor P_{wv} that receives such a message decreases $f(w,v)$ by $\sigma(v,w)$ and increases $u_f(w,v)$ by the same amount. Step 7 computes the new excesses on each node by performing a *seg-suffix-add* on the amount of new flow pushed through each arc, and this amount is added to $e_f(v)$.

Steps 4–6 implement the simple *relabel* operation. In Step 4, each processor P_{vw} sets its variable *head-label* to either $d(w)+1$ or $2n$, depending on whether the arc

Procedure *pulse*
(1) **For all** $v \in V - \{s,t\}$ copy $e_f(v)$ to all P_{vw} using *seg-prefix-copy* operation.
(2) { distribute excess }
For all P_{vw} use *seg-suffix-add* to compute the amount that can be pushed to lower labeled nodes through arcs that follow (v,w) on the incident list of v.
For all P_{vw} compute the amount $\sigma(v,w)$ to be pushed from v to w.
For all $v \in V$ compute the amount of excess that remains at v after the pushing.
(3) { Push flow }
For all P_{vw} **do if** $\sigma(v,w > 0$ **do begin**
$f(v,w) \leftarrow f(v,w) + \sigma(v,w)$; $u_f(v,w) \leftarrow u_f(v,w) - \sigma(v,w)$;
send a message containing $\sigma(v,w)$ to processor P_{wv}.
end.
For all P_{wv} that received $\sigma(v,w)$ **do begin**
$f(v,w) \leftarrow f(v,w) - \sigma(v,w)$; $u_f(v,w) \leftarrow u_f(v,w) + \sigma(v.w)$;
If *simple relabel* is the relabeling operation chosen **then begin**
(4) { Compute new distance labels }
For all P_{vw} **do**
if $u_f(v,w) > 0$ **then** *head-label*$(v,w) \leftarrow d(v) + 1$
else *head-label*$(v,w) \leftarrow 2n$.
For all $v \in V - \{s,t\}$ compute *new-d*(v) using *seg-suffix-min.*
(5) **For all** $v \in V - \{s,t\}$ copy *new-d*(v) to all P_{vw} using *seg-prefix-copy*
(6) { Broadcast new labels }
For all P_{vw} such that $v \notin \{s,t\}$ **do**
If $d(v) \neq new - d(v)$ **then**
send a message containing the value of $d(v)$ to P_{wv}
and set $d(v)$ to *new-d*(v) that was broadcast.
end.
(7) { Update excess }
For all $w \in V$ **do begin**
Use *seg-suffix-add* to compute the amount of flow new-$e_f(w)$ pushed into w,
$e_f(w) \leftarrow e_f(w) + new\text{-}e_f(w)$
end.

FIGURE 4. Implementation of the pulse procedure.

OPERATION ***Parallel BFS***
$d'(s) \leftarrow n$, and $d'(t) \leftarrow 0$;
For all nodes $v \in V - \{s,t\}$ $d'(v) \leftarrow 2n$;
Repeat
Run steps **(3)**, **(4)** and **(5)** of *Pulse* procedure.
Until *new-d*$(v) = d(v)$ for all nodes v.

FIGURE 5. Implementation of parallel breadth-first search operation.

OPERATION ***Parallel gap-relabel***
(1) For each node whose label $d(v)$ satisfies $d(v) < n$ in parallel **do**:
Broadcast a flag to the processor numbered $d(v)$;
(2) Find the smallest g where processor g did not receive any message in step 1).
(3) For all nodes v in parallel do:
if $g < d(v) < n$ set $d(v) \leftarrow n$.
(4) **For all** v with new labels copy *new-d*(v) to all P_{vw} using *seg-prefix-copy*.
(5) **For all** p_{vw} which received new labels send a message to P_{wv} containing *new-d*(v).

FIGURE 6. Implementation of the parallel gap-relabel operation.

(v,w) is residual or not. This process involves only local memory access. Next, a *seg-suffix-min* operation is performed on the *head-label* variable, and as a result each node processor P_v contains new value of $d(v)$. In Step 5 all node processors except P_s and P_t copy this value to their corresponding arc processors using *seg-prefix-copy*. In Step 6 each arc processor P_{vw} checks if the new $d(v)$ is different from the old one and if so, sends a message to its pair processor P_{wv} updating $d(v)$ in the pair processor.

Each step 1 through 7 contains either a segmented parallel prefix (suffix), or a communication primitive, and the running time of each step is dominated by the primitive. Therefore, the overall running time of the pulse procedure is roughly seven routing cycles of the machine. Also observe that general communications are done along paths that are fixed through entire program: each processor P_{vw} has to communicate to processor P_{wv} in steps 3 and 6 and these are the only general communication operations. Therefore, the communication path has to be computed only once for each arc processor and the same information is used throughout the program.

To implement the parallel BFS operation we simply take steps 4–6 of pulse and run them over and over until the labels do not change. The number of times the simple *relabel* is iterated in a BFS operation is at most the larger of maximum distance of a node to the sink (if sink is reachable) and maximum distance of a node to the source (if sink is not reachable) in the residual graph.

The gap-relabel procedure is also easy to implement on the Connection Machine; see Figure 6. Clearly this procedure is not much more costly than a simple *relabel* operation (roughly four routing cycles vs. three). However, it may increase the labels of many nodes by a substantial amount.

We have experimented with several variants using gap-relabel and BFS operations. In all of the variants each pulse uses push and simple relabel operations, but at certain times instead of simple relabel a BFS or a gap-relabel operation is used.

When using BFS we follow the following rule. We measure the amount of computational work done in the last call to BFS. Then we accumulate the amount of work done by the simple relabel since the last call to BFS. We also fix a parameter k. If k times the amount of work since the last call to BFS exceeds the amount of work in the last BFS then we use BFS, otherwise we use simple relabeling. The amount of work itself can be measured in several ways. One way is to simply look at the CPU time used. Another way is to count the number of "expensive" operations, in this case the number of routing cycles. Each simple relabel contributes three routing cycles (one parallel suffix copy, one parallel prefix min, and one general communication step), and the work accumulated by the simple relabeling procedure is simply three times the number of pulses since last call to BFS. The amount of work in each BFS varies as the residual graph changes with each new preflow. We have used this technique for both push–relabel and push–push–relabel methods. The latter is a variation of a the push-relabel method where we only relabel every other pulse.[2] (One may think of this method as choosing the relabeling operation that does nothing, and alternate using this operation with simple relabeling.) We also tested this technique with the pipelined variants of push–relabel and push–push-relabel techniques (to be discussed in the next section.)

Another approach is to use push and relabel operations but after each relabel to apply a gap-relabel operation as well. Our experiments show that one does not need to apply gap-relabel every time. We fix a parameter k and call gap-relabel after every k pulses. Again, we tested gap-relabel with both push–relabel and push–push–relabel variants of pulse and their pipelined versions. All of the timings are reported in Section 4.

These algorithms are expected to work well if at each pulse a large proportion of nodes have positive excess and can distribute their excess to the lower labeled nodes. In the worst case a network made up of a simple long path will have only one node active at each pulse; see Figure 7(a). On the other hand the best case happens in the network shown in Figure 7(b) where only one pulse will find the the maximum flow (in both cases we assume that all arc capacities are equal, say to one.) It is easy to see that the number of pulses is at least as large as the the length of the shortest path from source to sink. In general, our parallel implementation works best if the diameter of the network is relatively small in comparison to the number of nodes. But in networks with very long source-sink distance, there is little parallelism to exploit, and the performance is expected to be worse than serial implementations. (Recall that CM processors are very slow even compared to the slowest PC's, and if only a few of them are computing at a time, not much performance can be expected.)

[2] In general a pulse can have x push operations followed by y relabel operations.

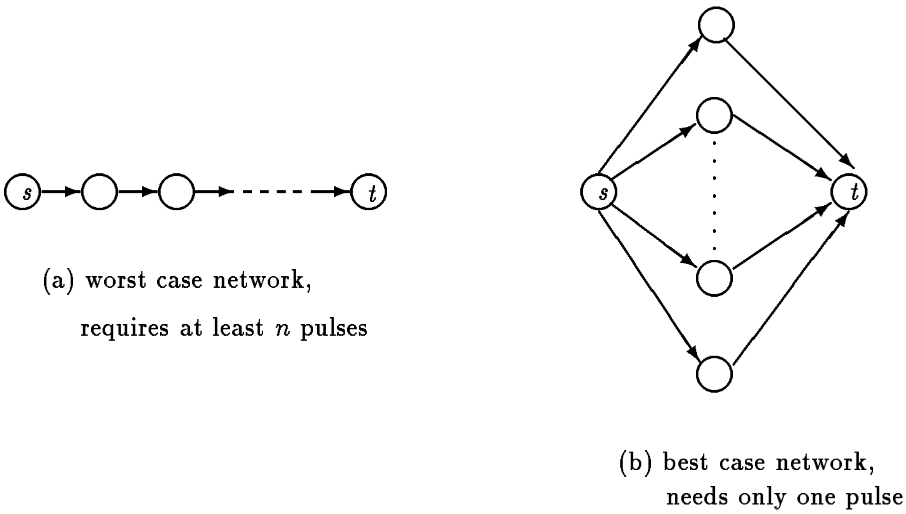

FIGURE 7. Best and worst case networks for the parallel push-relabel algorithms.

4.2. Pipelining Independent Operations. In the Connection Machine, if two segmented parallel prefix or suffix operations work on exactly the same sequences and perform the same binary operations, it is possible to *pipeline* them so that the pipelined operation, performed on the two sequences at once, is faster than two operations, each performed on one of the sequences at a time. For instance, steps 1 and 5 of the pulse could be pipelined, and so could steps 2 and 7. Steps 3 and 6 also do the same kind of operation, except that they involve message routing instructions. In theory we should be able to get better performance by pipelining the message routing steps, but on the Connection Machine we have not seen significant improvement. Pipelining the segmented prefix and suffix operations, however, results in about 10 to 20 percent improvement. (For example, instead of two *seg-suffix-copy* operations on 32 bit parallel integer variables, we may concatenate the two integers into a 64 bit long integer and apply *seg-suffix-copy* on this longer parallel variable. Then, the time required to do the latter operation is generally 10 to 20 percent faster than twice the time required to do a *seg-suffix-copy* operation on a 32 bit parallel variable.)

There are seven routing cycles in the pulse procedure (including cycles in simple relabeling). Some steps need to be performed after others (for instance, Step 6 must follow Step 5, and Step 5 must follow Step 4), whereas others are independent of each other (for example, steps 1 and 2 do not depend on each other and either may follow the other.) Figure 8 shows the dependency relationship among these steps. The arrows in the downward direction (say between steps 1 and 3) indicate dependency within a pulse; those in the upward direction (*e.g.* between steps 6 and 2) indicate dependency from one pulse to the next, (therefore, step two of the current pulse must start only after step 6 of the previous pulse is completed.)

The problem with pipelining steps 1 and 5, 2 and 7, and 3 and 6 is that they are sequentially related in the dependency graph. In order to curb this problem we take advantage of the robustness of the method. The dependency of Step 4 on Step 3 exists so that in the relabel step we have the updated residual capacities. Also, the dependency of Step 2 on Step 6 (of the previous pulse) exists so that the push operation is done based on new labels. These dependencies may be loosened somewhat so that steps 1 and 5, steps 2 and 7, and steps 3 and 6 can be pipelined. The disadvantage is that now the labeling becomes less accurate, and this translates into more iterations of the pulse procedure. The advantage is that now each pulse has only four routing cycles, albeit three of which are pipelined (and thus operate on longer data), see Figure 9.

The question is whether the time saved in each pulse more than compensates the time lost due to increased number of pulses. We report on the experiments in the next section. Notice that the SIMD nature of the Connection Machine does not allow us to exploit the independence of other operations such as, for example, steps 2 an 4 (one is a *seg-suffix-min* operation and the other a *seg-suffix-add*.)

We also used pipelining on the push-push-relabel implementation. (Recall that in this implementation we call the relabel operation only every other pulse.) The dependency graph for this version of the algorithm is unfolded in Figure 10. Notice that every stage in Figure 10 is equivalent to two pulses, only one of which has a relabel stage. Thus in this form the total number of routing cycles for two pulses is seven, whereas in Figure 9 there are eight routing cycles per two pulses.

5. Experimental Results

In this section we report on the running times of our program on several classes of medium and large networks. These experiments were conducted on two similar Connection Machines, one located at the Thinking Machine Corporation in Cambridge, Massachusetts, and the other one at the Army High-Performance Computing Research Center (AHPCRC) at the University of Minnesota in Minneapolis. Both machines have 32K processors. The timings reported here are based on test runs on the machine in AHPCRC. The program was coded using the new C* programming language, an extension of standard C for data parallel programming.

For each class of networks we have run four basic algorithms: push-relabel without pipelining (PR0), push-relabel with pipelining (PR1), push-push-relabel without pipelining (PPR0) and push-push-relabel with pipelining (PPR1). In all cases the simple relabel is used in the pulse procedure, except at certain pulses gap-relabel or BFS operation is used instead, as discussed in the previous section. Gap-relabel is used once in every k pulses for some parameter k. We experimented with values of k at 6, 8, 10, 12, and 14. The optimal value of k depends on the structure of the input network, and it is somewhat of an art to find it without excessive computa-

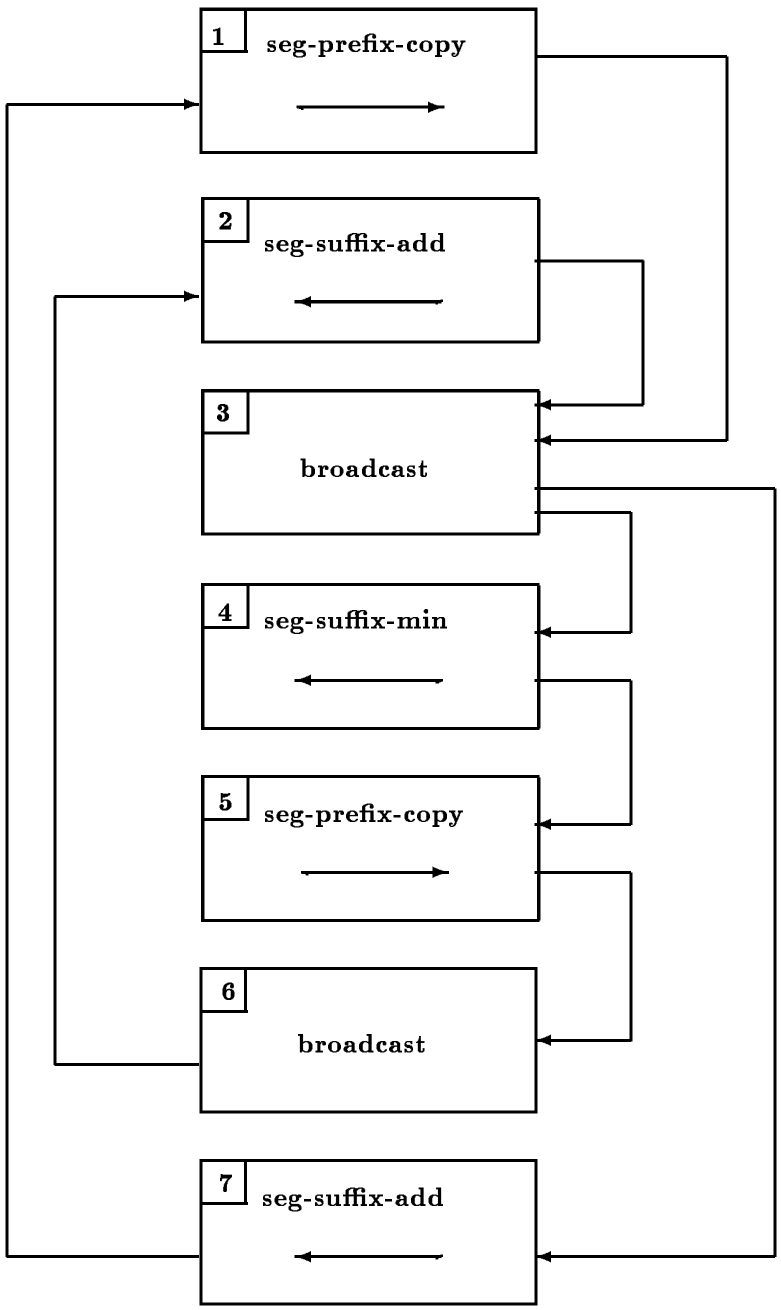

FIGURE 8. The dependency graph in the pulse procedure.

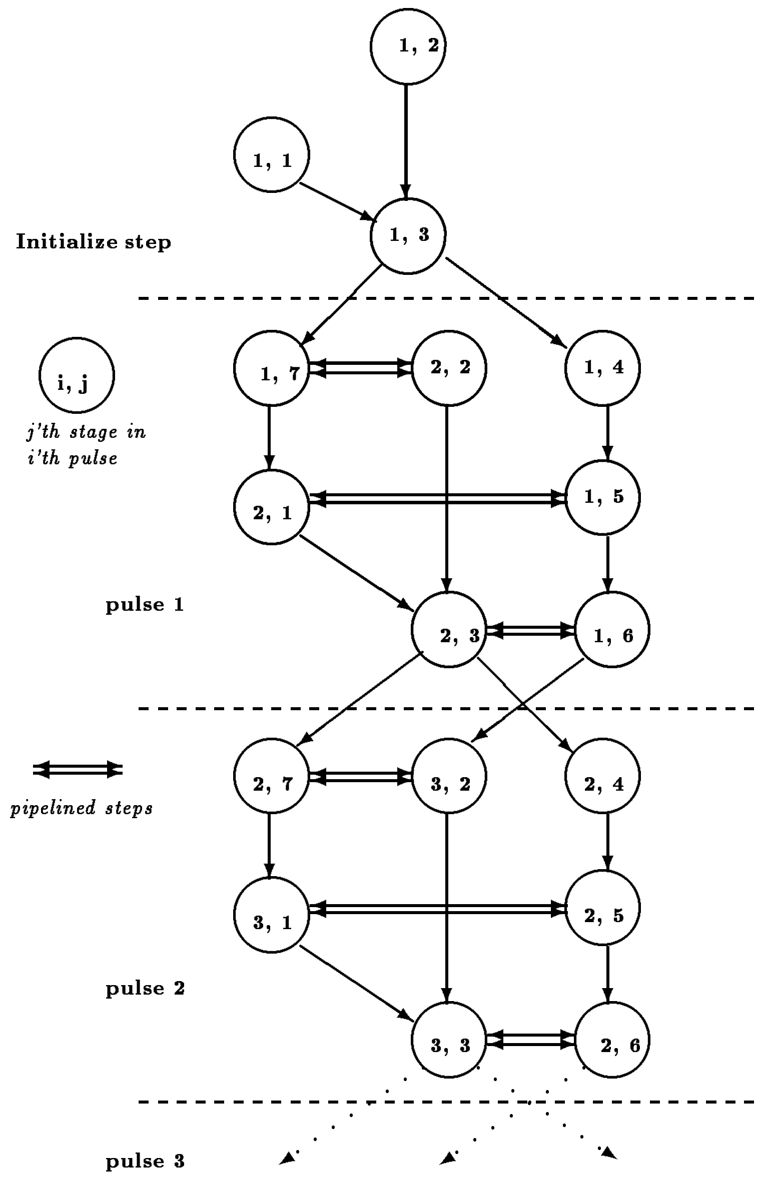

FIGURE 9. Unfolded dependency graph of the pulse procedure with pipelined steps.

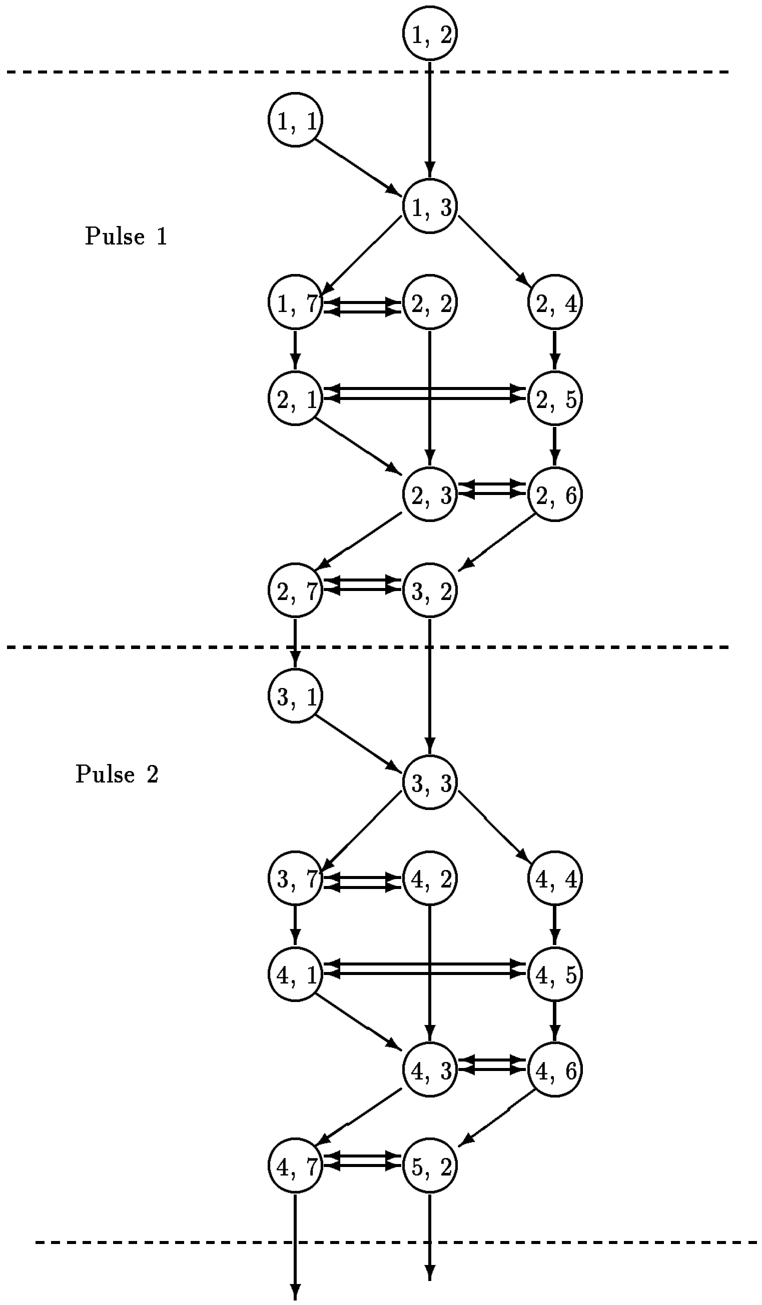

FIGURE 10. Unfolded dependency graph of push–push–relabel procedure with pipelined steps.

tion. Our experiments show that in the range $6 \leq k \leq 14$ the running times are reasonably good over all input classes.

For BFS, as mentioned earlier, we compare the amount of "work" (in our experiments the number of routing cycles) done by the last BFS, and if it exceeds k times the accumulated "work" done by simple relabels then we use BFS. We experimented with values of k ranging over $\{1.5, 1.7, 1.9.2.1, 2.3, 2.5\}$. Although there seems to be a large variation in the running time of the algorithms with varying k this range seems to work well overall.

The programs were run using either 8K, 16K, or 32K processors as appropriate. We experimented with a number of networks recommended by DIMACS. The timing and number of pulses for each run are reported in the sequel. In reporting the timings, we used the Connection Machines' CM-timer library. These timings report only the cpu time used by the Connection Machine and exclude computations done by the front end. In our implementation, the front end has minimal computational role. Aside from broadcasting the parallel instructions and simple computations (such as comparing two numbers to decide whether to do simple relabel or gap/BFS relabel) virtually all computing is done by the Connection Machine. Before we report on our computational results we should also mention that for the push–push–relabel heuristic with pipelining, the number of pulses should be multiplied by two. This is because every pulse in this variant corresponds to two pulses in other algorithms.

The first group are the fully dense acyclic networks constructed on a complete graph with an acyclic orientation. The characteristics of these networks are summarized in Table 11. For all variants of push–relabel algorithm, the fully dense acyclic networks seem to be particularly easy. The number of pulses for all of the variants are very small, and even the largest one, with over two million arcs is solved in a matter of seconds.

The second and third group consist of networks created by the "genrmf" generator. This generator takes as input two parameters a and b, and creates a network made of b parallel grids of size $a \times a$ (for a total of $a^2 b$ nodes in the network altogether). Nodes in each grid are connected to nodes to their right, left, up and down. In addition, ach node of a grid is connected to exactly one node of the next grid determined by a random permutation. We experimented with two classes of such networks: wide and long. Each network has approximately 2^x nodes. For wide networks we set $a \approx 2^{2x/5}$ and $b \approx 2^{x/5}$. For long networks we set $a \approx 2^{x/4}$ and $b \approx 2^{x/2}$. We experimented with values of x set at 10, 12, 14, and 16. The characteristics of these networks are summarized in Tables 12 and 13. For this class of networks the timings and pulse counts are shown in Figures 18, 19, 20, and 21. It seems that the gap-labeling heuristic outperforms BFS-labeling for RMF-Long class. However, between push–relabel and push–push–relabel variants there is no clear winner. Also, notice the sharp fluctuation for the BFS labeling as k varies. For instance, in the pipelined version of push-push-relabel with BFS labeling the running time for $k = 1.7$ is almost twice slower than for $k = 1.9$.

The timing results for RMF–Wide networks are summarized in figures 22, 23, 24 and 25. Among all networks recommended by DIMACS, this class by far requires the largest number of pulses. As it can be seen from these tables the number of pulses for each member of this class was almost in the same order of magnitude as the number of nodes. The large number of pulses for this class is reflected in the unusually large cpu time.

Finally, we experimented with networks known as "basic line mesh" class. These networks are made of 2^{x-2} rows and only 4 columns. Nodes in each column are connected to d random nodes in the next column. The parameter d is set to $\sqrt{n}$. The characteristics of these networks are summarized in Table 14. In figures 26, 27, 28, and 29 the timings and pulse counts for this class of networks are reported. It seems that for this group the gap-relabel heuristic is more efficient than the BFS labeling. Also the results in Table 28 indicate that the parallel speed up is linear. In addition, it seems that for this class push–push–relabel algorithm outperforms push–relabel, even though generally the number of pulses in the former exceeds that of the latter variation. (Remember that push–push–relabel applies relabeling only every other pulse.) For the BFS labeling heuristic there is virtually no variation in the number of pulses and in the running time of the program as the parameter k changes, except for BLM1. This insensitivity is rather atypical as the other classes often show sharp fluctuations with even the smallest change in k. Our version of push–relabel with pipelining failed to halt for this class of graphs when gap–relabeling were used. Even for BFS relabeling, when the program took unusually long time and had a much larger than expected pulse count. This may be due to an error in the code which manifests itself only for this class of problems.

The success of parallel implementation of the push–relabel algorithm rests largely on the number of pulses required to find the solution. The number of pulses in turn depends on the amount of parallel computation taking place in each pulse. If in each pulse a large number of nodes push their excess or a large number of nodes are relabeled, then we expect that fewer pulses are required to find the maximum flow. In fact, BFS and gap–relabeling operations are meant to increase the average amount of relabeling in each pulse. For the push operation no such heuristics are known to us. Each node may participate in a push only if it has positive excess. Therefore, it is reasonable to assume that if on average a large proportion of nodes have positive excess, then more computation is performed in each pulse and therefore, fewer pulses may be required. For each run of our program we calculated the number of nodes and arcs with positive excess in each pulse. The results are shown in Figure 30. It appears that only a very small fraction of nodes have positive excess on average. However, the number of physical processors we used is at most 32K and the size of networks most of the time far exceeds this number. Therefore, on average, a much larger proportion of physical processors participate in computation of each pulse.

A comparison between RMF-Long and RMF-Wide networks with approximately equal size is instructive. The number of active nodes and arcs in RMF-Long is almost one order of magnitude larger than that of RMF-Wide class; the converse is true for their running time.

6. Conclusions

We described several parallel implementations of the push-relabel method for the maximum flow problem and evaluated several heuristics that improve the practical performance of the method. The most encouraging aspect of our results is that the number of pulses, which is essentially the main bottleneck for parallel implementation of push–relabel algorithm, is usually only a fraction of the number of nodes in the networks. The small number of pulses is a direct consequence of occasional use of more accurate relabeling procedures such as BFS and gap–relabel. This shows that our approach will result in much better timings as more powerful processors are employed in future generations of Connection Machine or other parallel machines. In fact, since the speed up from 8K to 16K to 32K is almost linear, we expect the 64k Connection Machine would be almost twice as fast as the 32K machine on large problems.

Out of sequential implementations for maximum flow codes for the DIMACS Challenge, test examples of Nguyen and Venkateswaran [**21**] are most similar to ours, and their sequential implementation seems quite good. The sequential implementation is done on SUN SPARC-2 workstation. For the biggest problem sizes for which the data is available, the 32K parallel implementation and sequential implementation compare as follows. On the acyclic dense problems with 1024 nodes, the parallel implementation is about 40 to 50 times faster. On the RMF-long problems with 65536 nodes, the parallel implementation is about 3 to 4 times slower. On the RMF-wide problems with 63504 nodes, the parallel implementation is about 4 to 7 times slower. On Basic Line Mesh problems with 4098 nodes, the parallel implementation is about 3 to 5 times slower.

For the problem families we tried, one can conclude that the parallel implementation is competitive with the sequential one. On one hand, this is disappointing since CM-2 is much more expensive than SPARC-2. On the other hand, this is encouraging since the SPARC chip technology is superior to that of CM-2.

CM-2 is a SIMD machine. In a MIMD environment we expect to take advantage of independence of some parallel instructions more effectively. In particular the pipelining operation should result in much larger savings in time, (though not necessarily in the number of pulses.) Also, in the presence of wider communication bandwidth between processors the pipelining operations should become a lot faster. (In the CM-2 the bandwidth between two processors is only 1 bit.)

Another interesting research direction is towards obtaining an efficient implementation of the maximum flow method on parallel machines with powerful processors.

On a small parallel machine, this problem was addressed in [**2**].

Acknowledgment

We would like to thank the Thinking Machine Corporation and the Army High Performance Computing Research Center at the University of Minnesota for making their Connection Machines available to us. The first author would also like to thank J. B. Rosen for his support during the course of this work, and Gene Golub and the Stanford University Computer Science department for their hospitality.

Graph	nodes	arcs	size
AC1	512	130816	262144
AC2	1024	523776	1048576
AC3	2048	2096128	4194304

FIGURE 11. Complete acyclic networks.

Graph	a	b	nodes	arcs	size
rmf-L1	6	32	1152	4956	7224
rmf-L2	8	64	4096	18368	26496
rmf-L3	11	128	15488	71687	102542
rmf-L4	16	256	65536	311040	441856

FIGURE 12. Rmf-Long graphs: $a \approx 2^y$, $b \approx 2^{2y}$.

Graph	a	b	nodes	arcs	size
rmf-W1	16	4	1024	4608	6400
rmf-W2	28	5	3920	18256	25312
rmf-W3	49	7	16807	80262	111475
rmf-W4	84	9	63504	307440	427392

FIGURE 13. Rmf-Wide graphs: $a \approx 2^{2y}$, $b \approx 2^y$.

Graph	nodes	arcs	size
BLM1	1026	8071	17168
BLM2	4098	65035	134168
BLM3	16386	522259	1060904
BLM4	65538	4186035	8437608

FIGURE 14. Moderately sparse Basic Line Mesh networks

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
AC1	16K	gap	6	21	0.83	28	1.29	25	0.82	16	1.03
			8	22	0.86	28	1.30	25	0.81	16	1.03
			10	22	0.86	28	1.31	25	0.82	16	1.02
			12	21	0.84	28	1.33	27	0.88	16	1.02
			14	22	0.88	32	1.51	31	0.99	16	1.03
		bfs	1.5	21	1.17	19	1.22	22	1.25	14	1.24
			1.7	16	0.88	19	1.24	25	1.22	14	1.12
			1.9	21	1.05	26	1.52	22	1.15	15	1.18
			2.1	21	1.05	25	1.39	22	1.03	15	1.18
			2.3	21	1.07	26	1.44	22	1.03	16	1.24
			2.5	21	1.07	26	1.44	25	1.10	17	1.30
	32K	gap	6	21	0.46	28	0.71	25	0.46	16	0.58
			8	22	0.48	28	0.72	25	0.46	16	0.57
			10	22	0.48	28	0.72	25	0.46	16	0.57
			12	21	0.47	28	0.74	27	0.49	16	0.57
			14	22	0.49	32	0.83	31	0.56	16	0.58
		bfs	1.5	21	0.67	19	0.69	22	0.73	14	0.71
			1.7	16	0.50	19	0.70	25	0.71	14	0.64
			1.9	21	0.60	26	0.86	22	0.67	15	0.67
			2.1	21	0.60	25	0.78	22	0.59	15	0.67
			2.3	21	0.61	26	0.81	22	0.59	16	0.71
			2.5	21	0.61	26	0.81	25	0.63	17	0.74

FIGURE 15. computational results for AC1 network, $|V| = 512$, $|E| = 130816$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
AC2	16K	gap	6	7	1.44	10	2.13	7	1.26	6	1.85
			8	7	1.45	10	2.13	7	1.26	6	1.81
			10	7	1.45	10	2.13	7	1.26	6	1.81
			12	7	1.45	10	2.13	7	1.26	6	1.81
			14	7	1.45	10	2.13	7	1.26	6	1.81
		bfs	1.5	7	1.97	8	2.34	7	1.78	5	2.14
			1.7	7	1.97	8	2.34	7	1.78	5	2.14
			1.9	7	1.97	8	2.34	7	1.78	5	2.14
			2.1	7	1.95	8	2.34	7	1.78	5	2.14
			2.3	7	1.97	8	2.34	7	1.78	5	2.14
			2.5	7	1.97	8	2.34	7	1.78	5	2.14
	32K	gap	6	7	0.72	10	1.09	7	0.63	6	0.95
			8	7	0.73	10	1.09	7	0.63	6	0.93
			10	7	0.73	10	1.09	7	0.63	6	0.93
			12	7	0.73	10	1.09	7	0.63	6	0.93
			14	7	0.73	10	1.09	7	0.63	6	0.93
		bfs	1.5	7	0.99	8	1.18	7	0.89	5	1.08
			1.7	7	0.99	8	1.18	7	0.89	5	1.08
			1.9	7	0.99	8	1.18	7	0.89	5	1.08
			2.1	7	0.99	8	1.18	7	0.89	5	1.08
			2.3	7	0.99	8	1.18	7	0.89	5	1.08
			2.5	7	0.99	8	1.18	7	0.89	5	1.08

FIGURE 16. computational results for AC2 network, $|V| = 1024$, $|E| = 523776$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
AC3	16K	gap	6	8	7.33	11	10.15	9	6.94	7	9.29
			8	8	7.33	11	10.15	9	6.94	7	9.13
			10	8	7.36	11	10.15	9	6.94	7	9.13
			12	8	7.36	11	10.15	9	6.94	7	9.13
			14	8	7.36	11	10.15	9	6.94	7	9.13
		bfs	1.5	8	10.60	10	12.75	9	13.64	6	11.59
			1.7	8	10.60	10	12.75	9	13.64	6	11.59
			1.9	8	10.60	10	12.75	9	10.17	6	11.59
			2.1	8	10.60	10	12.75	9	10.17	6	11.59
			2.3	8	10.60	10	12.75	9	10.17	6	11.59
			2.5	8	10.60	10	12.75	9	10.17	6	11.59
	32K	gap	6	8	3.84	11	5.27	9	3.64	7	4.75
			8	8	3.84	11	5.27	9	3.64	7	4.75
			10	8	3.86	11	5.27	9	3.64	7	4.75
			12	8	3.86	11	5.27	9	3.64	7	4.75
			14	8	3.86	11	5.27	9	3.64	7	4.75
		bfs	1.5	8	5.62	10	6.70	9	7.30	6	6.12
			1.7	8	5.62	10	6.70	9	7.30	6	6.12
			1.9	8	5.62	10	6.70	9	5.40	6	6.12
			2.1	8	5.62	10	6.70	9	5.40	6	6.12
			2.3	8	5.62	10	6.70	9	5.40	6	6.12
			2.5	8	5.62	10	6.70	9	5.40	6	6.12

FIGURE 17. computational results for AC3 network, $|V| = 2048$, $|E| = 2096128$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
rmfl1	8K	gap	6	137	1.01	158	1.09	167	1.00	93	1.06
			8	142	0.95	158	1.18	167	0.90	92	1.05
$a = 6$			10	146	1.06	158	1.37	167	0.89	101	1.02
$b = 32$			12	140	0.93	158	1.08	169	0.99	101	1.02
			14	141	0.93	164	1.23	167	0.88	92	0.93
		bfs	1.5	146	1.16	221	1.89	160	1.30	142	1.69
			1.7	159	1.24	241	1.91	170	1.46	155	1.91
			1.9	173	1.43	262	2.05	186	1.46	169	1.94
			2.1	187	1.41	282	2.18	199	1.51	183	2.16
			2.3	200	1.49	303	2.32	152	1.12	196	2.18
			2.5	214	1.58	324	2.55	158	1.15	210	2.40

FIGURE 18. computational results for RMF-Long1 network, $|V| = 1152$, $|E| = 4956$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
rmfl2	16K	gap	6	484	4.36	552	5.10	628	4.63	344	4.67
			8	482	4.31	534	4.99	621	4.51	333	4.47
$a = 8$			10	507	4.49	524	4.77	625	4.49	343	4.53
$b = 64$			12	499	4.51	552	4.97	643	4.59	354	4.91
			14	540	4.73	560	5.03	638	4.53	382	4.94
		bfs	1.5	492	5.12	840	8.73	485	5.35	512	7.48
			1.7	569	5.81	889	9.08	609	6.35	586	8.40
			1.9	637	6.43	543	5.58	684	6.91	337	4.85
			2.1	722	7.17	555	5.52	548	5.27	357	5.10
			2.3	792	7.80	595	5.87	599	5.67	384	5.42
			2.5	438	4.26	635	6.23	669	6.15	410	5.73
	32K	gap	6	484	3.16	552	3.79	628	3.34	344	3.51
			8	482	3.14	534	3.64	621	3.26	333	3.36
			10	507	3.29	524	3.56	625	3.26	343	3.43
			12	499	3.24	552	3.72	643	3.33	354	3.51
			14	540	3.48	560	3.76	638	3.29	382	3.76
		bfs	1.5	492	3.84	840	6.60	485	4.09	512	5.78
			1.7	569	4.34	889	6.84	609	4.84	586	6.49
			1.9	637	4.81	543	4.23	684	5.26	337	3.74
			2.1	722	5.36	555	4.16	548	3.99	357	3.92
			2.3	792	5.82	595	4.42	599	4.28	384	4.17
			2.5	438	3.18	635	4.68	669	4.63	410	4.41

FIGURE 19. computational results for RMF-Long2 network, $|V| = 4096$, $|E| = 18368$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
rmfl3	16K	gap	6	811	21.57	864	23.62	959	19.38	533	21.63
			8	788	20.28	864	22.85	959	18.58	533	20.61
$a = 11$			10	833	20.47	864	22.38	959	18.11	533	20.08
$b = 128$			12	787	18.83	864	22.05	959	17.80	533	19.77
			14	789	18.55	864	21.86	959	17.57	533	19.53
		bfs	1.5	782	21.44	798	23.68	873	22.86	825	34.21
			1.7	923	24.45	798	24.04	870	22.02	552	23.14
			1.9	1052	27.26	836	25.15	864	20.68	565	23.72
			2.1	800	19.73	914	27.07	904	21.67	617	25.48
			2.3	809	20.06	992	28.99	910	21.98	669	27.14
			2.5	823	20.46	1071	30.92	889	19.66	721	28.80
	32K	gap	6	811	12.94	864	13.88	959	11.70	533	12.70
			8	788	12.00	864	13.44	959	11.23	533	12.18
			10	833	12.25	864	13.21	959	10.96	533	11.87
			12	787	11.40	864	13.05	959	10.77	533	11.67
			14	789	11.26	864	12.90	959	10.65	533	11.52
		bfs	1.5	782	13.07	798	14.10	873	14.03	825	20.27
			1.7	923	14.91	798	14.30	870	13.51	552	13.73
			1.9	1052	16.61	836	14.95	864	12.68	565	14.07
			2.1	800	12.07	914	16.08	904	13.28	617	15.11
			2.3	809	12.27	992	17.20	910	13.46	669	16.09
			2.5	823	12.50	1071	18.33	889	12.02	721	17.07

FIGURE 20. computational results for RMF-Long3 network, $|V| = 15488$, $|E| = 71687$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
rmfl4	16K	gap	6	1946	196.11	1882	194.45	2335	174.35	1292	196.57
$a = 16$			8	1853	173.42	1882	186.74	2335	164.91	1292	185.90
$b = 256$			10	1870	167.61	1882	182.21	2335	159.35	1292	179.55
			12	1918	166.44	1882	179.05	2335	155.60	1292	175.24
			14	1876	159.21	1882	176.99	2335	152.95	1292	172.20
		bfs	1.5	1890	174.94	2675	280.78	1974	180.69	1735	249.33
			1.7	2309	206.34	2693	292.01	1992	173.47	1747	260.77
			1.9	2616	230.53	1903	194.17	2034	177.42	1229	175.43
			2.1	2446	212.68	1909	196.82	2083	172.94	1254	179.76
			2.3	2351	202.20	2003	206.25	2082	168.46	1338	190.17
			2.5	2614	222.02	2153	219.81	2164	170.94	1441	202.02
	32K	gap	6	1946	120.64	1882	108.31	2335	102.01	1292	113.43
			8	1853	103.66	1882	102.25	2335	94.50	1292	105.17
			10	1870	98.27	1882	98.68	2335	90.13	1292	99.97
			12	1918	96.14	1882	96.15	2335	87.18	1292	96.60
			14	1876	91.20	1882	94.53	2335	85.02	1292	94.17
		bfs	1.5	1890	92.84	2675	144.66	1947	95.02	1735	129.39
			1.7	2309	109.18	2693	150.62	1992	91.27	1747	135.34
			1.9	2616	122.02	1903	99.96	2034	93.34	1229	90.99
			2.1	2446	112.48	1909	101.39	2083	91.05	1254	93.24
			2.3	2351	107.00	2003	106.28	2082	88.73	1338	98.66
			2.5	2614	117.52	2153	113.28	2164	90.04	1441	104.81

FIGURE 21. computational results for RMF-Long4 network, $|V| = 65536$, $|E| = 311040$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
rmfw1	8K	gap	6	862	5.13	812	5.17	1134	5.53	624	5.92
$a = 16$			8	822	4.89	812	5.63	1127	5.43	631	6.09
$b = 4$			10	795	4.80	806	5.13	1127	5.45	634	5.90
			12	806	4.83	812	5.08	1153	5.51	624	5.88
			14	807	4.81	852	5.63	1109	5.46	624	5.85
		bfs	1.5	739	5.47	871	6.52	892	6.81	641	7.11
			1.7	825	6.16	825	5.98	836	6.13	652	7.02
			1.9	729	5.28	942	6.75	926	6.49	653	6.84
			2.1	727	5.13	922	6.62	1042	7.20	589	6.12
			2.3	781	5.55	921	6.56	920	6.07	616	6.41
			2.5	699	4.97	933	6.62	1094	7.11	696	7.08

FIGURE 22. computational results for RMF-Wide1 network, $|V| = 1024$, $|E| = 4608$.

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
rmfw2	8K	gap	6	2613	33.45	2821	38.84	3305	33.72	1770	35.46
$a = 28$			8	2354	29.60	2821	38.39	3318	33.08	1778	35.23
$b = 5$			10	2258	28.21	2821	37.78	3305	32.69	1777	34.89
			12	2401	29.78	2821	37.58	3305	32.23	1770	34.16
			14	2358	29.28	2855	37.92	3318	32.20	1766	33.71
		bfs	1.5	2386	35.37	2557	39.66	2801	42.48	1795	40.04
			1.7	2268	32.39	2428	36.85	2807	40.56	1985	42.91
			1.9	2309	32.45	2708	40.50	2965	41.20	1922	40.29
			2.1	2326	32.32	2559	37.36	2792	37.46	2046	42.53
			2.3	2672	36.54	3048	44.62	2712	35.49	1743	35.63
			2.5	2340	31.47	2872	41.24	3127	39.90	1661	34.98
	16K	gap	6	2613	22.17	2821	24.44	3305	23.07	1770	22.76
			8	2354	19.72	2821	24.13	3318	22.69	1778	22.39
			10	2258	18.80	2821	23.97	3305	22.44	1777	22.12
			12	2401	19.90	2821	23.77	3305	22.23	1770	21.86
			14	2358	19.50	2855	23.93	3318	22.20	1766	21.76
		bfs	1.5	2386	24.29	2557	25.62	2801	29.55	1795	26.05
			1.7	2268	22.51	2428	23.78	2807	28.17	1985	28.14
			1.9	2309	22.49	2708	25.97	2965	28.69	1922	26.43
			2.1	2326	22.32	2559	23.89	2792	26.20	2046	27.71
			2.3	2672	25.30	3048	28.47	2712	24.69	1743	23.37
			2.5	2340	21.79	2872	26.42	3127	27.90	1661	22.17

FIGURE 23. computational results for RMF-Wide2 network, $|V| = 3920$, $|E| = 18256$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
rmfw3	16K	gap	6	10441	254.17	9551	236.69	14240	265.30	7292	270.70
$a = 49$			8	10280	236.60	9551	229.20	14240	253.65	7292	259.06
$b = 7$			10	10253	228.20	9551	224.71	14240	246.69	7292	252.11
			12	9112	197.76	9551	221.74	14240	242.00	7292	247.44
			14	10651	226.98	9551	219.59	14240	238.67	7292	244.14
	32K	gap	6	10441	150.05	9551	138.90	14240	157.45	7292	157.61
			8	10280	140.96	9551	134.92	14240	151.39	7292	151.46
			10	10253	136.65	9551	132.54	14240	147.76	7292	147.76
			12	9112	119.25	9551	130.95	14240	145.35	7292	145.35
			14	10651	137.12	9551	129.80	14240	143.61	7292	143.59
		bfs	1.5	8361	123.91	9101	141.40	9104	140.75	6550	144.52
			1.7	8527	122.80	10615	160.84	11202	164.36	6920	149.53
			1.9	9216	129.86	10402	155.58	11026	156.19	7670	162.15
			2.1	9164	127.51	9864	144.64	10307	141.42	6650	140.61
			2.3	8885	122.37	10582	154.42	11347	150.77	7009	145.71
			2.5	8446	115.46	10140	147.59	11344	147.34	7096	146.29

FIGURE 24. computational results for RMF-Wide3 network, $|V| = 16807$, $|E| = 80262$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
rmfw4	32K	gap	6	22608	1103.01	22926	1103.61	+	+	+	+
$a = 84$			8	23863	1086.00	22926	1057.43	+	+	+	+
$b = 9$			10	21753	938.90	22926	1029.70	+	+	+	+
			12	21771	908.77	22926	1011.25	+	+	+	+
			14	22835	933.58	22926	998.03	+	+	+	+
		bfs	1.5	21044	914.96	23099	1088.31	+	+	+	+
			1.7	21142	901.94	21599	1009.38	+	+	+	+
			1.9	20751	870.86	22478	1027.95	+	+	+	+
			2.1	19885	820.40	20746	944.18	+	+	+	+
			2.3	20004	811.68	21837	977.08	+	+	+	+
			2.5	20230	813.74	23194	1029.25	+	+	+	+

FIGURE 25. computational results for RMF-Wide4 network, $|V| = 63504$, $|E| = 307440$. (No experiments were conducted where a + appears.)

				PR0		PR1		PPR0		PPR1	
GR	n-PRC	LBL	k	pls	tm	pls	tm	pls	tm	pls	tm
blm1	8K	gap	6	155	2.08	*	*	190	2.07	104	2.22
			8	155	2.09	*	*	196	2.22	104	2.42
			10	154	2.07	*	*	190	2.15	105	2.31
			12	152	2.09	*	*	195	2.09	104	2.18
			14	146	2.07	*	*	207	2.31	107	2.24
		bfs	1.5	167	2.74	223	4.22	147	2.50	143	3.68
			1.7	182	2.93	245	4.46	126	1.99	158	3.97
			1.9	196	3.12	267	4.79	132	2.05	172	4.24
			2.1	211	3.21	289	5.03	140	2.13	187	4.72
			2.3	226	3.61	311	5.27	148	2.20	202	4.91
			2.5	241	3.80	333	5.70	158	2.40	217	5.10

FIGURE 26. computational results for Basic Line Mesh network BLM1, $|V| = 1026$, $|E| = 8071$. (The program exceeded its imposed time limit wherethere is a *.)

GR	n-PRC	LBL	k	PR0 pls	PR0 tm	PR1 pls	PR1 tm	PPR0 pls	PPR0 tm	PPR1 pls	PPR1 tm
blm2	8K	gap	6	90	9.70	*	*	97	8.92	53	9.53
			8	87	9.37	*	*	97	8.83	53	9.37
			10	88	9.49	*	*	97	8.78	53	9.54
			12	91	9.68	*	*	97	8.58	53	9.29
			14	91	9.54	*	*	97	8.64	53	9.54
		bfs	1.5	92	12.29	313	37.96	97	11.43	53	12.14
			1.7	92	12.19	354	41.15	97	11.31	53	12.17
			1.9	92	12.40	395	44.79	97	11.42	53	12.06
			2.1	92	12.30	437	47.82	97	11.34	53	12.04
			2.3	92	12.31	478	50.88	97	11.15	53	12.06
			2.5	92	12.60	519	54.12	97	11.43	53	12.10
	16K	gap	6	90	4.72	*	*	97	4.31	53	4.58
			8	87	4.56	*	*	97	4.26	53	4.54
			10	88	4.60	*	*	97	4.21	53	4.51
			12	91	4.69	*	*	97	4.21	53	4.49
			14	91	4.67	*	*	97	4.18	53	4.47
		bfs	1.5	92	5.92	313	17.91	97	5.36	53	5.64
			1.7	92	5.92	354	19.50	97	5.36	53	5.64
			1.9	92	5.92	395	21.09	97	5.36	53	5.64
			2.1	92	5.92	437	22.72	97	5.37	53	5.64
			2.3	92	5.92	478	24.31	97	5.37	53	5.64
			2.5	92	5.92	519	25.89	97	5.37	53	5.64
	32K	gap	6	90	2.63	*	*	97	2.37	53	2.52
			8	87	2.53	*	*	97	2.34	53	2.48
			10	88	2.54	*	*	97	2.30	53	2.47
			12	91	2.59	*	*	97	2.30	53	2.45
			14	91	2.58	*	*	97	2.28	53	2.43
		bfs	1.5	92	3.24	*	*	97	2.91	53	3.08
			1.7	92	3.24	*	*	97	2.91	53	3.08
			1.9	92	3.24	*	*	97	2.91	53	3.08
			2.1	92	3.24	*	*	97	2.91	53	3.08
			2.3	92	3.24	*	*	97	2.91	53	3.08
			2.5	92	3.24	*	*	97	2.91	53	3.08

FIGURE 27. computational results for Basic Line Mesh network BLM2, $|V| = 4098$, $|E| = 65035$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
blm3	16K	gap	6	168	69.38	*	*	172	61.46	92	64.72
			8	166	68.64	*	*	172	60.86	92	64.29
			10	168	69.06	*	*	172	60.55	92	63.82
			12	164	67.80	*	*	172	60.40	92	63.52
			14	166	68.52	*	*	172	60.25	92	63.37
		bfs	1.5	165	89.01	601	272.47	172	80.46	91	83.25
			1.7	165	89.01	680	295.52	172	80.46	91	83.25
			1.9	165	89.01	760	318.87	172	80.46	91	83.25
			2.1	165	89.01	840	342.22	172	80.46	91	83.25
			2.3	165	89.01	920	365.57	172	80.46	91	83.25
			2.5	165	89.01	999	388.63	172	80.46	91	83.25
	32K	gap	6	168	36.57	*	*	172	31.96	92	33.54
			8	166	35.98	*	*	172	31.53	92	33.11
			10	168	36.08	*	*	172	31.32	92	32.90
			12	164	35.34	*	*	172	31.22	92	32.69
			14	166	35.67	*	*	172	31.11	92	32.58
		bfs	1.5	165	45.93	*	*	172	41.32	91	42.63
			1.7	165	45.93	*	*	172	41.32	91	42.63
			1.9	165	45.93	*	*	172	41.32	91	42.63
			2.1	165	45.93	*	*	172	41.32	91	42.63
			2.3	165	45.93	*	*	172	41.32	91	42.63
			2.5	165	45.93	*	*	172	41.32	91	42.63

FIGURE 28. computational results for Basic Line Mesh network BLM3, $|V| = 16386$, $|E| = 522259$.

GR	n-PRC	LBL	k	PR0		PR1		PPR0		PPR1	
				pls	tm	pls	tm	pls	tm	pls	tm
blm4	32K	gap	6	301	515.07	*	*	312	462.82	166	482.97
			8	303	516.02	*	*	312	458.43	166	478.58
			10	299	510.52	*	*	312	455.92	166	476.07
			12	300	510.66	*	*	312	454.66	166	474.18
			14	300	510.34	*	*	312	453.40	166	472.93
		bfs	1.5	298	675.46	*	*	312	616.96	158	622.07
			1.7	298	675.46	*	*	312	616.96	158	622.07
			1.9	298	675.46	*	*	312	616.96	158	622.07
			2.1	298	675.46	*	*	312	616.96	158	622.07
			2.3	298	675.46	*	*	312	616.96	158	622.07
			2.5	298	675.46	*	*	312	616.96	158	622.07

FIGURE 29. computational results for Basic Line Mesh network BLM4, $|V| = 65538$, $|E| = 4186035$.

Network	size	PR0		PR1		PP R0		PPR1	
	(× 1K)	gap	BFS	gap	BFS	gap	BFS	gap	BFS
AC1	262.0	3.0	3.6	3.8	3.6	4.0	3.0	3.5	3.9
AC2	1048.0	4.8	4.8	4.7	4.4	4.8	4.4	5.1	4.7
AC3	4194.0	5.6	5.6	5.2	5.3	5.5	5.5	5.9	6.1
rmfl1	7.2	0.4	0.4	0.3	0.2	0.4	0.4	0.3	0.2
rmfl2	26.0	0.8	0.7	0.7	0.6	0.8	0.8	0.7	0.7
rmfl3	102.0	2.5	3.0	2.0	3.0	2.4	2.5	2.3	2.1
rmfl4	441.0	7.5	6.7	6.2	6.7	7.2	7.3	6.3	6.74
rmfw1	6.4	0.1	10.3	0.1	0.1	0.1	0.1	0.1	0.1
rmfw2	25.0	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.2
rmfw3	111.0	0.4	0.4	0.4	0.43	0.3	0.4	0.3	0.4
rmfw4	427.0	1.1	1.2	0.9	1.0	+	+	+	+
BLM1	17.0	0.7	0.5	*	0.5	0.6	0.7	0.6	0.5
BLM2	134.0	3.8	3.3	*	0.9	4.1	4.1	4.2	4.1
BLM3	1060.0	18.0	14.6	*	3.8	20.1	20.1	20.3	20.4
BLM4	84.0	78.0	59.5	*	+	85.7	85.7	88.1	92.4

FIGURE 30. The average number of active nodes and arcs in units of 1K.

REFERENCES

1. R. K. Ahuja and J. B. Orlin. Personal communication. 1987.
2. R. J. Anderson and J. C. Setubal. On the Parallel Implementation of Goldberg's Maximum Flow Algorithm. In *4th Annual ACM Symp. on Parallel Algo. and Arch.*, pages 168–177. ACM, 1992.
3. R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Prespective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching*. AMS and ACM, to appear.
4. G. Blelloch. Parallel Prefix vs. Concurrent Memory Access. Technical report, Thinking Machines, Inc., 1986.
5. B. V. Cherkassky. Personal communication. 1991.
6. Thinking Machine Corporation. *Connection Machine Model CM2 Technical Summary*. 1990.
7. U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.
8. E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
9. A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.
10. A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).
11. A. V. Goldberg. Processor-Efficient Implementation of a Maximum Flow Algorithm. *Information Processing Let.*, 38:179–185, 1991.
12. A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.

13. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.
14. L. M. Goldschlager, R. A. Shaw, and J. Staples. The Maximum Flow Problem is Log Space Complete for P. *Theoretical Computer Sci.*, 21:105–111, 1982.
15. M. D. Grigoriadis. Personal communication. 1988.
16. W. D. Hillis. *The Connection Machine.* MIT Press, 1985.
17. A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.
18. R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *J. Assoc. Comput. Mach.*, 27:831–838, 1980.
19. C. E. Leiserson and B. M. Maggs. Communication-Efficient Parallel Graph Algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.
20. X. Li and S. A. Zenios. Data-level Parallel Solution of Min-Cost Network Flow Problems Using ϵ-Relaxations. Technical Report 91-05-04, Decision Sciences Department, The Wharton School, University of Pennsylvania, Philadelphia, PA 19104, 1991.
21. Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Dimacs Implementation Challenge Workshop: Algorithms for Network Flows and Matching.* AMS and ACM, to appear.
22. J. T. Schwartz. Ultracomputers. *ACM Trans. Prog. Lang. and Syst.*, 2:484–521, 1980.
23. Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.

Andrew V. Goldberg, Computer Science Department, Stanford University, Stanford, CA 94305.
E-mail address: goldberg@cs.stanford.edu

Farid Alizadeh, International Computer Science Institute, Berkeley, CA 94706
E-mail address: alizadeh@icsi.berkeley.edu