A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic

David A. Bader*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Vipin Sachdeva
Electrical and Computer Engineering Department
University of New Mexico
Albuquerque, NM 87106

Abstract

The maximum flow problem is a combinatorial problem of significant importance in a wide variety of research and commercial applications. It has been extensively studied and implemented over the past 40 years. The push-relabel method has been shown to be superior to other methods, both in theoretical bounds and in experimental implementations. Our study discusses the implementation of the push-relabel network flow algorithm on present-day symmetric multiprocessors (SMP's) with large shared memories. The maximum flow problem is an irregular graph problem and requires frequent fine-grained locking of edges and vertices. Over a decade ago, Anderson and Setubal implemented Goldberg's push-relabel algorithm for shared memory parallel computers; however, modern systems differ significantly from those targeted by their implementation in that SMP's today have deep memory hierarchies and different performance costs for synchronization and fine-grained locking. Besides our new cache-aware implementation of Goldberg's parallel algorithm for modern shared-memory parallel computers, *our main new contribution is the first parallel implementation and analysis of the gap relabeling heuristic* that runs from 2.1 to 4.3 times faster for sparse graphs.

1 Introduction

A flow network is a directed graph $G = (V,E)$ with $|V| = n$ vertices and $|E| = m$ edges and with two distinguished vertices, the source vertex s and the sink vertex t. Each edge has a positive real-valued capacity function c, and there is a flow function f defined over every vertex pair. The flow function must satisfy three constraints:

- $f(u,v) \leq c(u,v)$ for all u,v in $V \times V$ (Capacity constraint)
- $f(u,v) = -f(v,u)$ for all u,v in $V \times V$ (Skew symmetry)
- $\sum_{v \in V} f(u,v) = 0$ for all u in $V - \{s,t\}$ (Flow conservation)

The flow of the network is the net flow entering the sink vertex t (which is equal to the net flow leaving the source vertex s). In mathematical terms, $|f| = \sum_{u \in V} f(u,t) = \sum_{v \in V} f(s,v)$. The maximum flow problem (MAX-FLOW) is to determine the maximum possible value for $|f|$ and the corresponding flow values for each vertex pair in the graph.

The maximum flow problem is not only an important theoretical graph algorithm, but has important practical applications in resource-allocation in networks and a variety of scheduling problems. Also, a surprising variety of linear programming problems in practice can be modeled as network flow problems. In such cases, special purpose network flow algorithms can solve such problems much faster than conventional linear programming methods. Also several of the graph problems such as bipartite matching, shortest path, and edge/vertex connectivity, can also be modeled as network flow problems [1, 23]. A large variety of sequential algorithms exist for MAX-FLOW. The sequential algorithms are typically grouped into two classes.

Augmenting Path Algorithms: maintain mass balance constraints at each vertex (other than s or t) and incrementally augment flow along paths from s to t;

Preflow Push-Push Algorithms: flood the network as a first step, and incrementally relieve flow from vertices with excesses by sending flow forward towards t or backward towards s based on the capacity of each edge.

*This work was supported in part by NSF Grants CAREER ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

Ford and Fulkerson [9] proposed the first maximum-flow algorithm, using the concept of augmenting paths (an augmenting path is a path from s to t that can be used to increase the flow from s to t because it is not being optimally used) and sending flows across these paths. Edmonds and Karp [8] improved upon the algorithm by sending flows across the shortest augmenting paths. They showed that using a breadth-first search in the labeling algorithm and selecting the shortest augmenting path always allows the algorithm to terminate in at most $O(nm^2)$. Dinic's algorithm [7] finds all the shortest augmenting paths in a single step, using "layered networks." Layers are determined by the present flow, and built on a breadth-first search using only useful arcs ($e = \langle u,v \rangle$ s.t. $f_e < c_e$ or $e = \langle v,u \rangle$ s.t. $f_e > 0$). (Note that throughout this paper for $e = \langle u,v \rangle$ we use the shorthand notation F_e to represent $F(u,v)$ for function F.) A phase consists of finding a layered network, then finding a maximum flow on the layered network and improving the original flow. The number of phases is at most $n-1$, and the algorithm runs in $O(n^2m)$. Karzanov [19] introduced the concept of preflows and the push operation and gave an $O(n^3)$ algorithm. Goldberg and Tarjan designed the push-relabel algorithm [13] with the time-bound of $O\left(nm \log \frac{n^2}{m}\right)$. In 1993, computational experiments confirmed that Goldberg's algorithm was the fastest algorithm in practice [2]. In a later paper by Goldberg and Cherkassky [5], several implementations of the push-relabel were studied and their results analyzed on a variety of graphs. We will discuss this algorithm in detail in Section 2, as our parallel implementation is based upon this sequential approach. Goldberg's survey paper [12] gives an excellent review of the algorithmic developments for the network-flow algorithm for the past forty years, including recent efforts.

Several researchers have given theoretic parallel algorithms for MAX-FLOW using the PRAM model [10, 15]. Goldberg and Tarjan [13] proposed an implementation of their push-relabel algorithm which takes $O(n^2 \log n)$ on an EREW PRAM with $O(n)$ processors. Details of Goldberg's parallel implementation using parallel prefix-sums is given in [11]. The MAX-FLOW problem restricted to planar directed graphs can be solved in $O(\log^3 n)$ time using $O(n^4)$ processors or in $O(\log^2 n)$ time using $O(n^6)$ processors on a CREW PRAM [16]. A more recent result for the MAX-FLOW problem on graphs with integer capacities is given by Sibeyn [21]. His solution finds the maximum flow in $O((\log C + \log^4 n) \log n / \log(m/n))$ using $O(n^2)$ processors on a CREW PRAM where C is the average edge capacity. Shiloach and Vishkin [20] give a parallel MAX-FLOW algorithm which runs in $O(n^2 \log n)$ using $O(n)$ processors on CRCW PRAM. There exists a randomized parallel algorithm to construct a maximum flow in a directed graph whose edge weights are given in unary, such that the number of processors is bounded by a polynomial in the number of vertices, and its time complexity is $O(\log^k n \log C)$ for some constant k, where C is the largest capacity of any edge [18]. While several researchers have proposed PRAM algorithms for the maximal flow problem, practical parallel implementations of any of these algorithms are rare. Anderson and Setubal [3] gave the first practical parallel implementation of the push-relabel algorithm for a uniform shared-memory address space. Their parallel implementation used only the *global relabeling* heuristic (described in Section 2) and demonstrated good speedups on the Sequent Symmetry over a sequential implementation for the families of graphs that were tested.

Our target architecture is a symmetric multiprocessor (SMP). Most of the new high-performance computers are clusters of SMPs having from 2 to over 100 processors per node. In SMPs, processors operate in a true, hardware-based, shared-memory environment. SMP computers bring us much closer to PRAM, yet it is by no means the PRAM used in theoretical work—synchronization cannot be taken for granted, memory bandwidth is limited, and good performance requires a high degree of locality. Designing and implementing parallel algorithms for SMPs requires special considerations that are crucial to a fast and efficient implementation. For example, memory bandwidth often limits the scalability and locality must be exploited to make good use of cache.

Our major innovations discussed in this paper are

- **a cache-aware optimization of Anderson and Setubal's approach, and**
- **the first design, implementation, and analysis, of a new shared-memory parallel algorithm for the gap relabeling heuristic that has been shown to improve performance.**

The organization of the rest of this paper is as follows. In Section 2 we review Goldberg and Tarjan's sequential push-relabel method for MAX-FLOW, including the global and gap relabeling heuristics. Section 3 describes Anderson and Setubal's parallel implementation of push-relabel that uses only the global relabeling heuristic. Our new high-performance and cache-aware parallel implementation using both global and gap relabeling is presented in Section 4. In Section 5 we perform experimental studies and analyze the performance using our parallel gap relabeling heuristic.

2 The Push-Relabel Algorithm

In this section, we detail the push-relabel algorithm by Goldberg and Tarjan [13]. The motivation behind the push-relabel algorithm is to push a large amount of flow from s to any internal vertex v in a single operation rather than augmenting the flow from the source in a time-consuming

operation in some cases. This initial flow might be passed from the internal vertex to the sink, if there exists sufficient capacity, or might be passed back to the source if it is in excess of the capacity of the network from v to the sink. This introduces the concept of *preflow* that relaxes the constraints discussed previously in which the net flow to any internal vertex, i.e. the difference between the incoming and the outgoing flows, is allowed to be non-negative during the running of the algorithm as opposed to be strictly zero. When the constraints are again satisfied for all the vertices of a graph, preflow becomes the maximum-flow of the graph.

All of the vertices $v \in V$ for which net flow is non-zero are *active vertices*. *Admissible edges* are edges $\langle u,v\rangle$ for which flow can be further increased without violating the maximum capacity, i.e. for which $c(u,v) - f(u,v) = u_f(v,w) > 0$. In Alg. 1 we first define a *push* and *relabel* operation after which we detail the algorithm.

2.1 Heuristics of Push-Relabel

A number of computational studies have focused on the push-relabel algorithm [6, 2]. The push-relabel algorithm is slow in practice, and relies upon two major heuristics (Global Relabel and Gap Relabel) to improve its performance. The following definitions are needed. The *residual capacity* of an edge $\langle u,v\rangle$ is $r(u,v) = c(u,v) - f(u,v)$. The edges with $r(u,v) > 0$ are residual edges E_f which induce the *residual graph* $G_f = (V,E_f)$. An edge with $r(u,v) = 0$ is *saturated*.

Global Relabeling heuristic: The distance labels ($d(v)$ for $v \in V$) in the push-relabel represent a lower bound on the distances from any vertex to the sink. These labels help the algorithm to push flow towards the sink, as the push operation is always carried from a vertex with a higher label connected to another with a lower label. Global relabeling updates the distance labels on the vertices as the shortest distance from the vertex v to the sink t along the residual graph $G_f = (V,E_f)$. This can be performed by a breadth-first search to the sink, the cost of which is $O(n+m)$. Such a relabeling is performed periodically after a number of push-relabel steps to amortize the expensive computational cost of the heuristic.

Gap Relabeling heuristic: updates the labels of the vertices which are unreachable from the sink to the label of the source which is $|V| = n$. Such a situation arises if there are no vertices with labels σ but vertices with distance labels $d(v)$ such that $\sigma < d(v) < n$. The distance labels of such vertices can be updated then to n. Such an update makes it possible to remove these vertices from consideration for pushing flow to the sink at once.

push(v,w)
Requirement: v is active and $\langle v,w\rangle$ is admissible.
Action: send $\delta = (0, \min(e_f(v), u_f(v,w))$ units of flow from v to w.

relabel(v)
Requirement: v is active **and** *push*(v,w) does not apply for any w.
Action: replace $d(v)$ by $\min_{\langle v,w\rangle \in E_f} d(w) + 1$

Data : (1) A directed graph $G = (V,E)$ of $|V| = n$ and $|E| = m$ with two distinguished
vertices source s and sink t
(2) Each vertex $v \in V$ has an adjacency list $\lambda(v)$
which has all outgoing edges outgoing from v
(3) Each edge $e = \langle u,v\rangle \in E$ has a capacity of $c(u,v)$
which is the maximum flow which can be passed through the edge

Result : The maximum flow $f(s,t)$ which can be routed through the graph i.e. from the source s to the sink t.

begin
(1) Set the source label $d(s) = n$, the sink label to $d(t) = 0$, and the labels on the remaining vertices to $d(v) = 0$ for all $v \in V - \{s,t\}$.
(2) Saturate all edges in the adjacency list of the source s i.e. $e \in \lambda(s)$ placing
excess flow on all the vertices connected to the source i.e. all w such that $\langle v,w\rangle \in \lambda(s)$.
(3) Calculate the residual edges i.e. all $e \in E$ such that $c_e - f_e > 0$.
while (active vertices) **do**
(3.1) Perform the *Relabel* operation on the active vertices.
(3.2) Perform the *Push* operation on the admissible edges.

end

Algorithm 1: Goldberg's Push-Relabel Algorithm for Maximum Flow

Goldberg and Cherkassky [5] implemented the push-relabel algorithm, and studied the running times based on operation orderings and distance update heuristics on a variety of graph families. They concluded that both the global relabeling as well as gap relabeling heuristics give the best performance. They also affirmed that the processing of vertices should be carried out preferably in highest-label order, as compared to first-in, first-out (FIFO) order. Goldberg [11] showed that the worst-case running time of FIFO order is $O(n^3)$, compared with $O(n^2\sqrt{m})$ for highest-label order. Also, the implementation of highest-label dramatically reduces the work necessary for finding gaps; hence even if the gaps are not found in some cases, the overhead is sizably small and can still achieve close to optimal performance [5].

3 Parallel Implementation of Push-Relabel

In this section, we focus on the parallel implementation by Anderson and Setubal [3]. We chose their implementation as, to our knowledge, it is the only practical push-relabel algorithm that has demonstrated a good speedup on shared-memory architectures. To achieve this performance, Anderson and Setubal optimized the concurrent global relabeling implementation. They realized in a shared-memory machine with a low number of processors, synchronous implementation of global or gap relabeling heuristics will offset any advantage in incorporating such a step in the parallel implementation. Goldberg's valid relabeling requires that $d(v) \leq d(w)+1$ for all edges $\langle v,w\rangle \in E_f$. Due to multiple processors working on possibly overlapping data, invalid relabelings might occur which could push the flow towards the source s causing incorrect results. Hence for simultaneous periodic global relabeling, they introduced the concept of *waves*. Each vertex of the graph, in addition to its label $d(v)$, is now assigned a wave number $wave(v)$. The wave number denotes the number of times the vertex has been globally relabeled. Alg. 2 details the augmented definitions of *push* and the *global relabel* operation required for concurrent global relabeling [3]. *CurrentWave* and *CurrentLevel* are the current wave number and the current level in the BFS tree, respectively.

Global relabeling is performed periodically, i.e. after $2n$ discharge operations are carried out by all the processors in total. Each processor has two local queues: an in-queue and an out-queue. A processor works on its in-queue in a FIFO order, until it runs out of work, in which case it gets vertices from the shared queue. Newly active vertices which are created during the discharge operation are placed in the out-queue of a processor until it gets full; after which the processor places all the activated vertices in the out-queue of the shared queue. The number of vertices transferred between the shared-queue and the in- or out-queues is varied during the program execution for dynamic granularity control through heuristics. Processors use locks for any access of the shared queue (i.e., for transferring vertices in or out of the shared queue).

Push$_i(v,w)$
Requirement: Processor i holds the locks for both v and w, $\langle v,w\rangle \in E_f, d(v) = d(w)+1$, and $wave(v) = wave(w)$.
Action: Push as much flow to w as $\langle v,w\rangle$ affords, and update v's and w's excesses.

Global Relabel$_i(v)$
Requirement: Processor i holds the locks for v, $wave(v) < CurrentWave$.
Action: if $d(v) < CurrentLevel$ **then**
1.1 $d(v) \leftarrow CurrentLevel$;
1.2 $wave(v) \leftarrow CurrentWave$;

Algorithm 2: Anderson-Setubal definitions for *Push* and Global Relabel

4 Our New High-Performance Implementation

Anderson and Setubal conducted their studies on the Sequent Symmetry, a shared-memory parallel machine circa 1987, no longer in production, and based on 16 MHz Intel 80386 processors. Superscalar processors capable of running two orders of magnitude faster are now widely pervasive in present day SMP's. The rate of improvement in microprocessor speed has been exponential and has exceeded the rate of improvement in DRAM speed. Hence, algorithm designers are faced with an increasing processor-memory performance gap, often referred to as the memory wall, a primary obstacle for attaining improved performance of computer systems. Cache-aware algorithm design is emerging as a possible technique for addressing this issue. Our initial port of Setubal's implementation for modern shared-memory computers scaled linearly in relative speedup with the number of processors on one family of graphs (acyclic dense graphs, described later), and nearly linearly on other families of graphs. However, the performance lacked absolute speedup compared with an optimized sequential implementation such as Goldberg's *hipr* (available from `http://www.avglab.com/andrew/soft.html`). For instance, our parallel code, running on eight processors, barely achieved the performance of the sequential implementation. Profiling the execution revealed a high rate of cache misses due to irregular memory access patterns, hindering performance.

4.1 Cache-Aware Implementation

In the push-relabel method, each directed edge $e = \langle v, w \rangle \in E$ is converted into two edges in opposite directions, $e_1 = \langle v, w \rangle$ and $e_2 = \langle w, v \rangle$. Edge e_1 appears in the adjacency list of v and has a capacity of the original edge e; edge e_2 appears in the adjacency list of w and has a capacity of 0, denoting that there cannot be flow along edge e_2. We refer to e_1 as the mate edge of e_2 and vice-versa in later sections. The antisymmetry constraint by Sleator [22] then specifies that the flow in e_1 should always be the opposite of the flow in e_2. Thus, during the execution of the code, any increase in the flow of e_1 must be met by a decrease in the flow of e_2. Such an access is also required for the global relabeling step since it has to read the mate edge's flow for a valid global relabeling. In this case, the mate edge's flow is just read and not updated contrary to the *push* operation.

For each edge we save its maximum flow and current flow information *and* its mate's information. This reduces the number of memory accesses when the mate edge's information is just read and not updated. For updates though, an effective solution is the contiguous allocation of memory used for the mated pair of edges. This ensures spatial locality so that a cache line or pair of adjacent lines holds the mated edge pair's portion of the data structure during the updating. When this *cache-aware* code was now tested for the families of graphs, it was found to give an excellent relative speedup for each family of graphs. However, the absolute speedups, compared to the optimized sequential implementation by Goldberg using push-relabel method with highest-label order vertices processing and gap and global relabeling heuristics, were not consistently improved. For dense graphs with 1,000 or more vertices, our cache-aware parallel implementation demonstrated good absolute speedups relative to Goldberg's code. However, the absolute speedup was poor on random level graphs. We discuss these issues and improvements to our parallel implementation in the next section.

4.2 Highest-Label Ordering of Vertices

Our cache-aware implementation, while improving the performance on dense graphs, lacked absolute speedup improvements on other families of graphs. The parallel code performed an order of magnitude more push and relabel operations than the sequential code. Due to the inherent cost of locking used in every push-relabel operation in the parallel code, this led to a significant performance degradation of the parallel code. There are two noteworthy differences between the sequential code (Goldberg's *hipr*) and our cache-aware parallel implementation.

- The sequential code processes the vertices in highest-label order (vertices with highest label are processed first) compared to parallel code which was processing the vertices in approximate FIFO order.
- The sequential code uses both the gap and global relabeling heuristics compared to the parallel code which lacked the gap relabeling heuristic.

Goldberg asserted that with FIFO order processing of vertices, the gap relabeling heuristic did not give further improvements. However, with highest-label processing order, the gap relabeling gives significant improvements. Thus for optimized performance, we needed to design and implement the following two modifications together:

- The processing of vertices must occur in highest-label rather than FIFO order.
- Gap relabeling must occur asynchronously; i.e., carried out concurrently with the push/relabel operations performed on the active vertices.

Next we detail our new approach for highest label processing in the parallel implementation; and defer the design and implementation of concurrent gap relabeling to the next subsection.

The prior implementation [3] uses a shared queue and a queue local to each processor for active vertices. Each local queue is further divided into a local in-queue and a local out-queue. The processor discharges or relabels vertices from its local in-queue and places the new active vertices into the local out-queue. When the local out-queue is full, it is emptied into the global shared queue. This structure is primarily maintained for load-balancing and work-stealing. Transfer of vertices between the local and the global queues is carried out in batches, for instance of size b each. This parameter b is varied during the course of the run for improved results: Anderson and Setubal gave different rules for increasing or decreasing the parameter b to prevent too much oscillation. We retain the queue structure and the load-balancing rules for transferring vertices. To implement highest-label ordering, we modify the structure of the local in-queue and the global queue, while retaining the concept of transfer of vertices between the shared queue and the local in- and out-queues. We divide the local in-queue into *buckets*, each of which holds vertices with the same label. The number of buckets is thus equal to the number of possible labels of vertices (0 to $n-1$). When a vertex is moved into the in-queue of a processor, it is placed in the appropriate bucket which holds all the vertices of the same label. The global queue is similarly divided into *buckets*, and any transfer between a local and global queue is thus *emptying of buckets* with the bucket of the highest label emptied first. Thus, when a processor attempts to transfer vertices from either the global queue or the local out-queue into the in-queue, the active vertices are copied starting from the highest label of the non-empty bucket. The highest label of the local in-queue and the global queue is suitably altered in case of such transfers: the highest label of the in-queue mostly increases while the

highest label of the global queue decreases as the buckets with the highest labels are *emptied*.

To optimize this implementation, several parameters are added to each local queue: number of vertices of each label or vertices present in a bucket b_i, total number of vertices each processor holds in its local in-queue or in all its buckets, and the highest label held by any processor in its local in-queue. We added this last parameter as we discovered that frequently the highest label held by any processor was much less than the maximum label n which could label any vertex in the graph. An issue of synchronization remains in that a processor running the global relabeling heuristic may update the labels of the vertices that are held in another processor's local in-queue. This occurs because there is a separate queue for the global relabeling, with processors gaining control of the queue at different intervals, and leads to vertices being held in a bucket with an updated label. We solved this issue by adding a flag to each vertex. When a processor changes the labels of a vertex in the global relabeling step, it sets the flag of the vertex denoting that the vertex has been *worked upon*. A processor then checks the flag of a vertex before it transfers the vertices from the local in-queue to the out-queue or the global queue: if the flag is set, it then moves the vertex into the correct bucket while transferring to the global queue. The transfer of vertices starting with the *highest-label bucket* ensures that the processing of the vertices is approximately highest label.

4.3 Concurrent Gap Relabeling

For improved performance, we use the gap relabeling heuristic in conjunction with the highest-label processing described in the previous section. For gap relabeling, we require additional bookkeeping such as the counts of the number of vertices with each particular label. Thus, when a processor changes the label of a vertex, it also updates the counts of the previous and new labels. This leads to a slight overhead: for updating the label, the processor was locking the vertex, but now also has to lock the previous label and the new label as well. We maintain a shared data structure comprised of a Boolean flag and a label which is initialized to n. For the gap relabeling heuristic, if the count of vertices of any label reaches zero due to relabeling, local or global, it is identified as a gap G_l. Therefore, vertices with labels greater than the label of the gap discovered previously are updated to n and are identified as *gap-active*. Once the gap is discovered, the processor then proceeds with the locking of the data structure, sets the Boolean flag, and updates the shared label to the label of the gap discovered G_l. We now introduce the updated *relabel* operation: the flag and the label are first read before relabeling, and if the flag is not set, the processors continue with the normal relabel operation. If however the flag is set, the processor checks the label of the vertex which it is to relabel, and if the label is greater than the label of the gap, the new label is n. Other processors may also discover other gaps; however, these gaps will only help in faster running of the implementation if a newly discovered gap has a lower label than the previous gap. Hence, the gap label is updated only if the newly discovered gap is lower than the previous gap label. The gap relabeling heuristic presented here is thus performed concurrently, without explicit synchronization. In Alg. 3 we give the algorithms for the updated and the newly introduced operations for gap relabeling.

gap_active(l_1)
Requirement: Count$[l_1]$ is 0 **and** *gapFlag* is not set.
Action: Set the *gapFlag*, and update *gapLabel* to l_1.

gap_update(l_2)
Requirement: Count$[l_2]$ is 0, *gapFlag* is set, **and** *gapLabel* $l_2 < l_1$
Action: update *gapLabel* to l_2.

relabel_nogap(v)
Requirement: v is active, *gapFlag* is not set, **and** *push*(v,w) does not apply for any w.
Action: replace $d(v)$ by $\min_{\langle v,w\rangle \in E_f} d(w)+1$

relabel_gap(v)
Requirement: v is active, *gapFlag* is set, **and** *push*(v,w) does not apply for any w and $d(v) >$ *gapLabel*.
Action: replace $d(v)$ by n

Algorithm 3: Updated and newly introduced operations for gap relabeling.

5 Experimental Results

We tested our shared-memory implementation on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers [4].

We use three families of graphs (taken from the 1$^{\text{st}}$ DIMACS Implementation Challenge [17]) for the experimental results:

Random Level Graphs: These graphs are rectangular grids of vertices, where every vertex in a row has three edges to randomly chosen vertices in the following row. The source and the sink are external to the grid,

the source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink.

RMF graphs: These graphs, described by Goldfarb and Grigoriadis [14], are comprised of l_1 square grids of vertices (frames) having $l_1 \times l_2$ vertices, and connected to each other in sequence. They can be generated by the RMFGEN generator by Goldfarb. The source vertex is in a corner of the first frame, and the sink vertex is in a corner of the last frame. Each vertex is connected to its grid neighbors within the frame and to one vertex randomly chosen from the next frame.

Acyclic Dense Graphs: These are complete directed acyclic dense graphs: each vertex is connected to every other vertex, the source and the sink included.

In Fig. 1 we plot the running times with increasing number of processors for instances of the three separate families. The graphs draw a comparison between the FIFO implementation with no gap relabeling, the FIFO implementation with gap relabeling, and our new highest-label processing with concurrent gap relabeling heuristic. In our experiments with FIFO-processing order, using gap relabeling has negligible effect on the performance, as expected. For acyclic dense graphs, the execution time difference between the FIFO implementations and the highest-label implementation with gap relabeling is negligible, and we expect this for the following reason. Since each vertex is connected to all other vertices, very few gaps (if any) are discovered, and the gap relabeling heuristic is not very effective in this case. We do observe a decrease in speedup with increasing number of processors, a problem due to smaller input sizes of graphs. On the other hand, we found significant improvement for random level graphs and the RMF graphs with the gap relabeling heuristic used in conjunction with the highest-label processing. In these cases of sparse graphs, the improvements ranged from 2.1 to 4.3 times faster than the FIFO implementations.

Acknowledgments

We wish to thank João Setubal for his parallel implementation of Goldberg's push-relabel maximum flow algorithm. Emeline Picart, while visiting University of New Mexico, ported Setubal's code from the Sequent Symmetric to modern Symmetric Multiprocessors. The porting was a non-trivial task as Emeline had to fix newly introduced race conditions caused by the significant differences between the shared memory models.

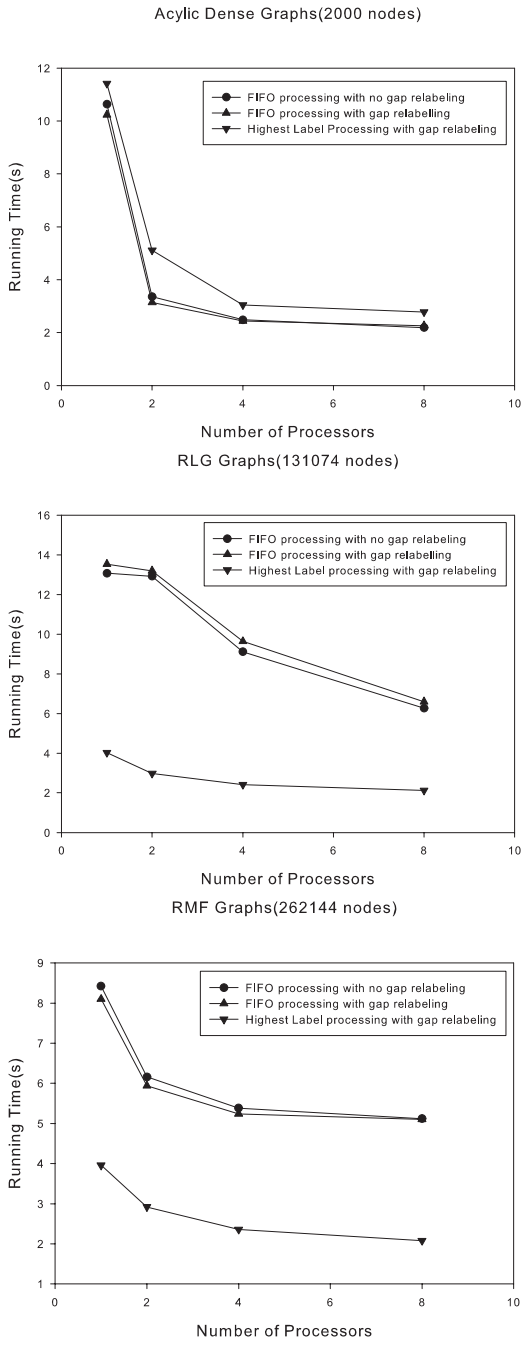

Figure 1: Performance of the Parallel Maximum Flow Implementations for Acyclic Dense Graphs (top), Random Level Graphs (middle), and RMF Graphs (bottom). We compare the performance of our cache-aware optimized implementations of FIFO processing with and without gap relabeling to our new optimized version with highest-label processing and the concurrent gap relabeling heuristic.

References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, editors. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] R.J. Anderson and J. C. Setubal. Goldberg's algorithm for the maximum flow in perspective: A computational study. In *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18, 1993.

[3] R.J. Anderson and J.C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. 4th Ann. Symp. Parallel Algorithms and Architectures (SPAA-92)*, pages 168–177, San Diego, CA, July 1992.

[4] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[5] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[6] B.V. Cherkassky, A.V. Goldberg, P. Martin, J.C. Setubal, and J. Stolfi. Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms. *ACM J. Experimental Algorithmics*, 3(8), 1998. `www.jea.acm.org/1998/CherkasskyAugment/`.

[7] E.A. Dinic. Algorithm for solution of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[8] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[9] L.R. Ford, Jr. and D.R. Fulkerson, editors. *Flows in Networks*. Princeton Univ. Press, NJ, 1962.

[10] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Ann. Symp. of Theory of Computing (STOC)*, pages 114–118, San Diego, CA, May 1978. ACM.

[11] A.V. Goldberg. *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, MIT, Cambridge, MA, January 1987.

[12] A.V. Goldberg. Recent developments in maximum flow algorithms. In *6th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 1–10, Stockholm, Sweden, July 1998.

[13] A.V. Goldberg and R.E. Tarjan. A new approach to the maximal flow problem. *Journal of the ACM*, 35:921–940, 1988.

[14] D. Goldfarb and M.D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. *Annals of Oper. Res.*, 13:83–123, 1988.

[15] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[16] D.B. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM*, 34(4):950–967, 1987.

[17] D.S. Johnson and C.C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1993.

[18] R.M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6(1):35–48, 1986.

[19] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.

[20] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel MAX-FLOW algorithm. *J. Algs.*, 3(2):128–146, 1982.

[21] J. Sibeyn. Better trade-offs for parallel list ranking. In *Proc. 9th Ann. Symp. Parallel Algorithms and Architectures (SPAA-97)*, pages 221–230, Newport, RI, June 1997. ACM.

[22] D. D.K. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, 1980.

[23] K. Steiglitz and C. H. Papadimitriou. *Combinatorial Optimization : Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.