Concurrent Algorithms for the Max-Flow Problem

William J. Dally

5191:DF:85

July 23, 1985

Abstract

This paper presents three novel concurrent algorithms for solving the max-flow problem. All three algorithms achieve concurrency by propagating multiple augmenting paths simultaneously. The algorithms differ in their approach to resolving conflicts for edge capacity between paths. To avoid the bottleneck of a single source or sink vertex, the concept of a distributed vertex is introduced. Experimental results show that the concurrent algorithms require about the same number of operations as the best known sequential algorithms, and achieve speedups of $O(\sqrt{|V|})$.

1 Introduction

The problem of determining the maximum flow in a network subject to capacity constraints, the max-flow problem, is a form of linear programming problem that is often encountered in solving communication and transportation problems. These problems usually involve lare networks and are very computation intensive. If a concurrent algorithm were available, the solution of max-flow problems would benefit considerably from the use of concurrent computers.

Sequential algorithms for constructing maximal flows are widely known. Unfortunately, these algorithms depend on sequentially labeling the vertices of a graph and thus can not easily be extended to run concurrently. This paper presents three new concurrent algorithms for the max-flow problem: the concurrent augmenting paths (CAP) algorithm, the concurrent vertex flow (CVF) algorithm and the concurrent augmenting digraph (CAD) algorithm.

All three of these algorithms achieve concurrency by investigating multiple augmenting paths simultaneously. They differ in their approach to resolving conflicts over edge capacity between paths. The CAP algorithm propagates all paths regardless of conflicts and then uses a reservation mechanism to allocate flow to specific paths. The CVF algorithm on the other hand avoids conflicts by finding how much flow can reach a vertex and only propagating that amount of flow further in the network. The CAD algorithm is a modification of the CAP algorithm that avoids duplication of paths in cases of reconvergent fanout.

Section 2 describes the the max-flow problem and introduces some of the notation that will be used in the remainder of this paper. Two sequential algorithms for the max-flow problem: Ford and Fulkerson's algorithm [1,2] and Dinic's algorithm [3]. are discussed in Section 3. Section 4 presents three novel concurrent algorithms: the CAP algorithm, the CVF algorithm and the CAD algorithm. The performance of these algorithms, in terms of number of operations and in terms of concurrency is investigated in Section 5. The paper concludes with a discussion of experimental measurements of the performance of these algorithms in Section 6.

2 The Problem

Consider a directed graph G(V,E) with two distinguished vertices, the source, s, and the sink, t. Each edge $e \in E$ has a *capacity*, $c(e)$. A *flow* function $f : E \mapsto R$ assigns a real number $f(e)$ to each edge e subject to the constraints:

1. $0 \leq f(e) \leq c(e)$,
2. Except for s and t, the flow out of a vertex equals the flow into a vertex: vertices conserve flow.

$$\forall v \in V - \{s,t\}\ , \sum_{e \in \text{in}(v)} f(e) = \sum_{e \in \text{out}(v)} f(e). \tag{1}$$

where in(v) is the set of edges into vertex v, and out(v) is the set of edges out of vertex v.

The network flow, $F(G,f)$, is the sum of the flows out of s. It is easy to show that F is also the sum of the flows into t. [1]

$$F = \sum_{e \in \text{out}(s)} f(e) = \sum_{e \in \text{in}(t)} f(e). \tag{2}$$

The max-flow problem is to find a legal flow function, f, that maximizes the network flow F.

3 Sequential Max-Flow

3.1 Ford and Fulkerson's Algorithm

An algorithm due to Ford and Fulkerson [1] incrementally constructs a maximal flow, $f_{\max}$, by finding *augmenting paths* from s to t.

Definition 1 An edge e is useful(v,u) if either

1. $e = (v,u)$ and $f(e) < c(e)$, or
2. $e = (u,v)$ and $f(e) > 0$.

An edge that is useful(v,u) can be used to increase the flow between v and u either by increasing the flow in the forward direction or decreasing the flow in the reverse direction.

Definition 2 An *augmenting path* is a sequence of edges $e_1, \ldots, e_n$ where

1. e_1 is useful(s, v_1),
2. e_i is useful(v_{i-1}, i) $\forall i \ni 1 < i < n$,
3. e_n is useful(v_{n-1}, t).

Thus, an augmenting path is a sequence of edges from s to t along which the flow can be increased by increasing the flow on the forward edges and decreasing the flow on the reverse edges. The construction of augmenting paths and the augmentation of flow along these paths is repeated until no further improvement is possible.

[1] It is assumed that there is no flow into the source or out of the sink.

In the Ford and Fulkerson algorithm, labeling is used to construct the augmenting path as shown in Figure 1. The outer loop repeatedly constructs augmenting paths from s to t and then follows these paths back to s augmenting the flow along the way. An augmenting path is constructed using a wavefront algorithm. The wavefront, set A, is initialized to contain only s. To propagate the path toward t, a vertex, v is removed from A and for each edge, e_j, that is useful(v, u) for some vertex, $u \in$ B, the vertex u is labeled and moved from set B to set A. The label has two components, $a_u = \min(a_j, a_v)$, the available capacity on the path from s to u, and the edge, $e_u = e$. Once an augmenting path from s to t has been discovered, the path is traversed by following the labels from t back to s. During this traversal, the flow on each edge of the path is adjusted by the amount a_t, the available capacity from s to t.

The general case of the Ford and Fulkerson algorithm is not guaranteed to converge unless the edge capacities and initial flows are integral valued. An example of nonconvergence is given in [1]. However, if the search for an augmenting path is performed *breadth first* (this corresponds to set A being treated as a FIFO queue) then the complexity of the algorithm is $O(|V|^3|E|)$ [2]. Dinic's algorithm, discussed below, improves significantly on this bound.

3.2 Dinic's Algorithm

Dinic's algorithm works by repeatedly layering the network considering only *useful edges* and then constructing a max-flow on the layered network. When no further layering is possible, a max-flow for the original network has been constructed [3].

The vertices in the network are partitioned into *layers* as shown in Figure 2. Initially, the source, s, is assigned to layer 0, set `A[0]`, and all other vertices are assigned to set B. During each iteration of the loop, the i^{th} layer, `A[i]`, is constructed from the $(i-1)^{\text{st}}$ layer by finding all vertices, u, in B connected to a vertex, v, in `A[i-1]` by useful edges. These vertices form layer i, and are moved from set B to set `A[i]`. When the sink, t, is added to a layer, the algorithm terminates with the t as the only vertex in the last layer. The layering algorithm requires time $O(|E|)$ since each edge is examined exactly twice by the `for` loop: once in the forward direction and once in the reverse direction [3].

After each partitioning of the network into layers, the flow is augmented subject to the constraints:

1. only edges between adjacent layers, `A[i-1]` and `A[i]` are considered;
2. forward edges (from `A[i-1]` to `A[i]`) can only have their flow increased, and backward edges (from `A[i]` to `A[i-1]` can only have their flow decreased.

A modified version of the Ford and Fulkerson algorithm using *depth-first search* and adapted to satisfy the two constraints is used to construct the max-flow for each layering.

Because of the layering constraints, the *max-flow* constructed during each partition/augment iteration is not the max-flow for the network. In fact, the flow constructed may not even be the best that can be achieved within the constraints. For example, consider Figure 3. If

```
--------------------------------------------------------------------------
max_flow(G)
{
  for each edge, e, f(e) = 0 ;
  /* construct and augment paths until no more improvement */
  do {
    A = {s}; B = V-A ; a(s) = infinity ;
    /* propagate augmenting path from s to t */
    while (A != {} and t is not in A) {
      pick a vertex v in A ;
      for each edge e = (v,u) so f(e) < c(e) and u is not in A
                 or e = (u,v) so f(e) > 0 and u is not in A {
        if (e does not exist) do nothing ;
        else if (e = (v,u))
          label(u) = e ; dir(u) = forward ; a(u) = min(a(v), c(e)-f(e)) ;
        else if (e = (u,v))
          label(u) = e ; dir(u) = backward ; a(u) = min(a(v), f(e)) ;
        A = A U {u} ;
      }
      A = A - {v} ;
    }
    /* follow path back from t to s augmenting flow */
    if(A != {}) {
      u = t ;
      do {
        if(dir(u) = forward) { (v,u) = e = label(u) ; f(e) += a(t) ;}
        else { /* backward */  (u,v) = e = label(u) ; f(e) -= a(t) ;}
        u = v ;
      } while (u != s) ;
    }
  } until A = {} ;
}
--------------------------------------------------------------------------
```

Figure 1: Ford and Fulkerson's Sequential Max-Flow Algorithm

```
--------------------------------------------------------------------------
layer_network(G)
{
  A[0] = {s} ; B = V-{s} ; i = 0 ;
  do {
    i = i+1 ;
    for every edge e, that is useful(v,u) for a vertex, v, in A[i-1]
    and a vertex u in B {
      A[i] = A[i] U {u} ;
    }
    B = B - A[i] ;
  } while((t is in B) and (A[i] is non empty)) ;
  if(A[i] is non empty) A[i] = {t} ;
}
--------------------------------------------------------------------------
```

Figure 2: Algorithm For Partitioning the Network into Layers

the first path considered is s, a, d, t, the algorithm will terminate with one unit of flow even though a flow of two is possible. Once the path s, a, d, t is constructed, every path from s to t subject to the constraints is blocked by a saturated edge.

The complexity of Dinic's algorithm can be bounded by calculating the number of partition/augment iterations that must be performed and the amount of time required to perform each iteration. In [3] it is shown that each iteration, the number of layers in the partition increases. Since the number of layers is bounded by $|V|$, at most $|V|$ iterations are required. Constructing the *max-flow* each iteration requires $O(|V| \times |E|)$ time. Thus, the complexity of Dinic's algorithm is $O(|V|^2|E|)$, a factor of $O(|V|)$ better than Ford and Fulkerson's algorithm. A variant on Dinic's algorithm due to Karzanov [4] requires only $O(|V|^3)$ time.

4 Concurrent Max-Flow

This section presents three concurrent max-flow algorithms: the concurrent augmenting paths algorithm (CAP) the concurrent vertex flow (CVF) algorithm, and the concurrent augmenting digraph (CAD) algorithm. All three algorithms iteratively partition the network into layers and increase the flow subject to the partitioning. The algorithms differ in their approach to increasing the flow in a layered network. The CAP algorithm increases flow by finding augmenting paths while the CVF algorithm attempts to *push* flow from vertex to vertex. The CAD algorithm is a modification of the CAP algorithm which improves performance by merging the path tree into a path digraph.

In a network flow problem, the source and sink can become bottlenecks since every augmenting path must pass through both of these vertices. In this section the concept of a distributed vertex is introduced to allow simultaneous operations on the source and sink

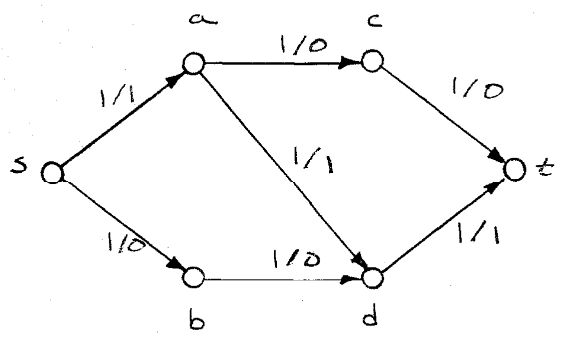

Figure 3: Suboptimal Layered Flow

vertices.

4.1 The Concurrent Augmenting Paths Algorithm (CAP)

The CAP algorithm iteratively partitions the network into layers and increases the flow subject to the layering. The flow is increased by finding augmenting paths as described in Section 3.1. Potential augmenting paths are constructed while the network is being partitioned. To prevent multiple paths from claiming the same edge capacity, each path is constructed in three phases.

1. The network is partitioned into layers by labeling each vertex, v, with a layer number. At the same time, all potential augmenting paths from s to t in the layered network are found by constructing a path tree rooted at s

   Construction of the path tree begins by finding all useful edges, out of s. The vertices at the opposite end of these edges are assigned to layer 1 and allocate path records for the first level of internal nodes in the path tree. The layer 1 vertices find all the edges to layer 2 vertices in a similar manner. Note that a layer $i > 1$ vertex may store several path records representing several internal nodes of the path tree, one for each unique path from the source to that vertex. The process continues until vertex t is reached. Unlike Dinic's algorithm since the construction is performed in parallel, it is impossible to terminate all paths at the same layer and paths continue to propagate until they reach vertex t or are blocked. The purpose of layering in the CAP algorithms is to assign a direction to each edge. Thus it is not important if the layering is not as strict as in the case of the sequential layering described in Section 3.2.

   For each path, p, propagated to vertex t, the maximum flow possible along p, C_p is recorded. The capacity of the edges used by the paths discovered in phase 1 is not locked, however, and several paths may use the same capacity. Conflicts over edge capacity are resolved in phases 2 and 3.

2. Paths discovered in phase 1 reserve edge capacity during phase 2 by following links in the path tree backwards from t to s. As a path, p, is propagated from t to s, it attempts to reserve capacity, C_{pR}, at each edge. Initially, at vertex t, $C_{pR} = C_p$. At each edge e_j along the path from t to s, C_{pR} is set to $\min(C_{pR}, a_j)$. If C_{pR} is reduced to zero, propagation is terminated and the path is followed back to t to cancel all reservations.

3. Paths that were able to reserve a positive capacity along their entire lengths during phase 2 confirm their reservations during phase 3. These paths also cancel reservations for any excess capacity that may have been reserved it their C_{pR} was decreased. When a phase 2 path, p, reaches s, with final capacity C_{pF}, the confirmation is performed by propagating p back from s to t the flow in each edge of p is adjusted by the amount C_{pF}, and if the initial reservation, C_{pR} was greater than C_{pF}, the reservation for the excess capacity is canceled.

The phases are propagated like wavefronts. During the first iteration, phase 1 propagates from s to t and upon reaching t, *reflects* back from t to s as phase 2. When phase 2 reaches s, it is reflected back to t as phase 3. When an iteration terminates without completing phase 1, a max-flow has been constructed.

The vertices, V, are partitioned among the processing nodes of a concurrent computer. Forward and backward adjacency lists are maintained for each vertex. A label, l_i, and tag, t_i, are associated with each vertex, v_i, to indicate the layer that v_i is assigned to. The label, l_i, contains the layer number, and the tag, t_i indicates the iteration for which this layer number is valid. Each edge, e_i, in an adjacency list has a capacity, c_i, a flow, f_i, and a reserved capacity, r_i. The source vertex of edge e_i is called s_i and its destination vertex is called d_i, these relationships are expressed using the notation $e_i = (s_i, d_i)$. Note that there are two pointers to each edge, e_i, one in the forward adjacency list of s_i, and one in the reverse adjacency list of d_i.

Each vertex contains a list of path records used to record the progress of path finding in the graph. During phase 1, path records are used to construct the path tree from s to t. During phase 2, they are used to record the individual paths back through the path tree from t to s. Each path record contains the following fields:

1. `phase` The phase for which the record is valid.

2. `flow` During phase 1, the minimum of available flow for all edges e_i from s to the current edge. During phase 2, C_p, the minimum amount of flow reserved for all edges from t back to the current edge.

3. `edge` The *preceeding* edge: during phase 1, the edge from layer $i-1$ to the current layer, i, during phase 2, the edge from layer $i+1$ to the current layer, i.

4. `vertex` The *preceeding* vertex (on the opposite side of edge).

5. `prev id` A pointer to the *preceeding* path record (in vertex `vertex`).

phase	flow	edge	prev vertex	prev id	next id	layer	tag

Figure 4: A Path Message

6. `count` A reference count of child path records.

The algorithm operates by passing messages between vertices. Three types of messages are used, one for each phase. All of the messages have the same format. As shown in Figure 4, they contain the fields:

1. `phase` The phase of the path's development, 1, 2 , 3 or reject.
2. `flow` The maximum flow that can be added along the path to the current edge. For a phase 1 message, the minimum excess capacity for an edge on the path between s and the current edge. For a phase 2 message, C_{pR}, the minimum reserved capacity for an edge on the path between the current edge and t.
3. `prev vertex` The previous vertex along the path, in layer $i-1$ for a phase 1 message, in layer $i+1$ for a phase 2 message.
4. `prev id` The id of the active path record in the current vertex.
5. `next id` The id of the relevant path record in the destination vertex.
6. `layer` During phase 1, the layer number, i, of the source vertex. The destination vertex, if unlabeled, is assigned to layer $i+1$.
7. `tag` The vertex labels are tagged with the iteration number so stale layer numbers do not affect later iterations.

In the algorithms shown below, messages are sent by calling procedure **send** which takes as its arguments the destination vertex and the fields of the message. The receipt of a message invokes a procedure in the destination vertex corresponding to the phase of the message.

Each iteration of a max-flow problem is started by sending a phase 1 message to the source vertex. On receipt of this message, the source vertex scans its adjacency lists for useful edges as shown in Figure 5. Phase one messages with the layer field set to one are transmitted across all useful edges.

When a phase 1 message arrives at a vertex, v_i, the processor containing the vertex executes the code shown in Figure 6. First the vertex label and tag are checked. If the label is stale (tag $\neq$ message tag) or if the label is greater than or equal to the layer field of the message, the vertice's label and tag are updated, and construction of the path tree continues. Otherwise, the message is rejected by sending a reject message to the sender.

To continue the path tree, a new path record is allocated to record the link back to the previous vertex on the path. The adjacency lists of the current vertex, v_i, are then scanned.

```
--------------------------------------------------------------------------
phase1_source()
{
  self.tag = self.tag+1 ;
  count = 0 ;
  for each edge e_j incident on the source {
    if(e_j is a forward edge) {
      (self,v_nxt) = e_j ;
      a_j = c_j - r_j - f_j ;
    }
    else { /* e_j is a reverse edge */
      (v_nxt,self) = e_j ;
      a_j = f_j - r_j ;
    }
    if(a_j > 0) {
      send(v_nxt,phase1,a_j,e_j,self,NIL,NIL,1,tag) ;
      count++ ;
    }
  }
}
--------------------------------------------------------------------------
```

Figure 5: Phase 1 Initiation

For each forward edge, e_f, the available capacity, a_f is calculated as $a_f = c_f - r_f - f_f$ and for each reverse edge, e_r, $a_r = f_r - r_r$. For each edge, e_j, for which $a_j > 0$, a phase 1 message is sent to the vertex at the opposite end of the edge. The flow field the outgoing messages corresponding to edge e_j is set to $\min(a_j, f_m)$ where f_m is the flow specified in the message.

The reject message decrements the reference count of the parent path record. When the reference count reaches zero, the record is freed, and a reject message is sent to the parent vertex to decrement its reference count.

When a phase 1 message arrives at the sink, a phase 2 message is reflected back along the path to reserve the required flow as shown in Figure 7. A count of phase 1 messages reaching the sink is kept to detect when an iteration is completed.

As shown in Figure 8, as each vertex along the path receives a phase 2 message, it determines the maximum amount of capacity that can be reserved for the corresponding path by taking the minimum of the available capacity for the edge, a_j, and the requested flow, `flow`: $f = \min(\texttt{flow}, a_j)$. A reservation for f units of edge e_j's capacity is then made by adding f to r_j. If a phase 2 message cannot reserve any capacity for the path, propagation of phase 2 messages stops, and a phase 3 message with zero flow is transmitted to cancel all reservations on the path from the current vertex to the sink.

When a phase 2 message reaches the source it increases the flow on the appropriate edge and reflects a phase 3 message back toward the sink. Phase 3 messages are propagated back along the path to the sink to increase the flow by the minimum reservation along the path, and to release any excess reserved capacity. Figure 9 shows the code to perform this reflection.

Following on the heels of the phase 2 messages that are propagating back to the source, reject messages are propagated back from the leaves of the path tree to the root to free path records after they have been passed by all phase 2 messages. When a reject message arrives at a vertex it decrements the appropriate path record's reference count. When a reference count goes to zero, the path record is freed and a reject message is sent to the next vertex down the path tree. Reject messages arriving at the source decrement the source's count of pending phase 1 messages.

As the phase 3 message propagates back along the path to the sink, each vertex along the path executes the code shown in Figure 10. The flow along the current path edge is increased, any excess capacity is released, the phase 2 path record for the path is freed, and the message is forwarded to the next vertex on the path.

When a phase 3 message reaches the sink, the flow along the path has been increased by a uniform amount. The path computation is complete so the count of pending path computations set during phase 1 is decremented as shown in Figure 11. When the counts in both the source and sink reach zero, the iteration is complete and the next iteration is started. When an iteration decrements both source and sink reference counts to zero without increasing the flow, the max-flow algorithm is finished.

There are some fundamental differences between the CAP algorithm and Dinic's algorithm. While both algorithms partition the network into layers, the CAP algorithm assigns layers only to assign direction to the edges and to partially order the edges (eliminating cycles).

```
-------------------------------------------------------------------------
/* all vertices except the source and sink */
phase1(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  /* check layer and tag */
  n = 0 ;
  if((self.tag != tag) || (self.label >= layer)) {
    self.tag = tag ;
    self.label = layer ;

    /* continue path tree */
    path_id = new_path_record(phase1,flow,edge,prev_vertex,prev_path_id,0) ;
    for each edge e_j incident on the current vertex {
      if(e_j is a forward edge) {
        (self,v_nxt) = e_j ;
        a_j = min(flow,c_j-r_j-f_j) ;
      }
      else { /* e_j is a reverse edge */
        (v_nxt,self) = e_j ;
        a_j = min(flow,f_j-r_j) ;
      }
      if(a_j > 0) {
        send(v_nxt,phase1,a_j,e_j,self,path_id,NIL,layer+1,tag) ;
        n++ ;
      }
    }
    if((path_id.count = n) == 0) free_path_record(path_id) ;
  }
  /* if label < layer, or count = 0, reject message */
  if(n == 0) {
    send(prev_vertex,reject,0,NIL,NIL,prev_path_id,NIL,0,NIL) ;
  }
}
-------------------------------------------------------------------------
```

Figure 6: Phase 1 Message Propagation

```
--------------------------------------------------------------------------
phase1_sink(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  if(self.tag != tag) {
    /* set count to zero for new iteration */
    self.tag = tag ;
    self.count = 0 ; self.flow = 0 ;
  }
  self.count = self.count + 1 ;
  send(prev_vertex,phase2,flow,edge,self,prev_path_id,NIL,0,0) ;
}
--------------------------------------------------------------------------
```

Figure 7: Phase 1 Message Reflection

```
--------------------------------------------------------------------------
/*  for all vertices except the source and sink */
phase2(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  p = &(path_record[next_path_id]) ;
  f = min(flow,a_j) ; /* j refers to p->edge */
  if(f!=0) {
    r_j = r_j + f ;
    a_j = a_j - f ;
    path_id = new_path_record(phase2,f,edge,prev_vertex,prev_path_id,1) ;
    send(p->prev_vertex,phase2,f,p->edge,self,path_id,p->prev_path_id,0,0) ;
  }
  else {
    send(prev_vertex,phase3,0,edge,self,prev_path_id,NIL,0,0) ;
  }
}
--------------------------------------------------------------------------
```

Figure 8: Phase 2 Message Propagation

```
-------------------------------------------------------------------------
/* for the source - also receives reject messages */
phase2_source(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  /* let e_j correspond to edge */
  /* adjust flow */
  if(e_j is a forward edge) f_j = f_j + flow ;
  else /* reverse edge */   f_j = f_j - flow ;
  send(prev_vertex,phase3,flow,edge,self,NIL,prev_path_id,0,0) ;
}
-------------------------------------------------------------------------
```

Figure 9: Phase 2 Message Reflection

```
-------------------------------------------------------------------------
/* for all vertices except the source and sink */
phase3(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  p = &(path_record[next_path_id]) ;
  /* let e_j correspond to edge */
  /* adjust flow */
  if(e_j is a forward edge) f_j = f_j + flow ;
  else /* reverse edge */   f_j = f_j - flow ;
  /* release excess reservation */
  r_j = r_j - p->flow ;
  a_j = a_j + (p->flow - flow) ;
  send(p->prev_vertex,phase3,flow,p->edge,self,NIL,p->prev_path_id) ;
  free_path_record(p) ;
}
-------------------------------------------------------------------------
```

Figure 10: Phase 3 Message Propagation

```
-------------------------------------------------------------------------
/* for the sink */
phase3_sink(phase,flow,edge,prev_vertex,prev_path_id,next_path_id,layer,tag)
{
  self.count = self.count - 1 ;
  self.flow = self.flow + flow ;
}
-------------------------------------------------------------------------
```

Figure 11: Phase 3 Message Termination

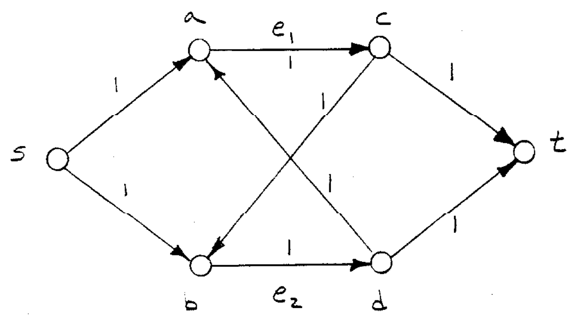

Figure 12: An Example of Livelock

Thus the CAP algorithm produces a looser layering where edges may span several layers. Dinic's algorithm requires that edges span only a single layer and that only shortest paths to the sink in the layered network be considered. Generating a strictly layered partition concurrently would require tighter synchronization and thus would result in lower performance.

The assignment of direction and order to edges solves two problems that plagued early versions of the CAP: livelock and cycle detection. Livelock (unlike deadlock) occurs when no process is blocked, but yet no progress is made toward a solution. If augmenting paths are propagated in parallel on an unordered network using the three phase CAP algorithm, livelock can occur as shown in Figure 12. Paths $\alpha = s, a, c, b, d, t$ and $\beta = s, b, d, a, c, t$ share edges $e_1 = (a, c)$ and $e_2 = (b, d)$. If the propagation of the two paths are roughly synchronized, path α will reserve all of the capacity of e_2 and path β will reserve all of the capacity of e_1. Then each path will discover that no available capacity remains on the other edge and both paths abort. The process can repeat indefinitely with no progress made toward a solution.

Assigning a partial order to edges eliminates the possibility of livelock. The partial order implies that two edges e_1 and e_2 must be in the same order on all paths that include both. Thus, it is impossible for two paths to reserve capacity needed by both before either observes the other's reservation.

In the early versions of the CAP algorithm it was necessary to include a list of *visited* vertices in each path message to avoid propagating messages around cycles. For example, in Figure 12, a message that has traversed path s, a, c, b, d must record this path so it does not propagate back to a and enter a cycle. The partial order assigned to edges by the layering eliminates the need to propagate these lengthy *visited* lists.

Unfortunately, one cycle of the CAP algorithm (phases 1, 2 and 3) is not in general sufficient to guarantee that a maximal flow has been constructed in the layered network. Consider, for

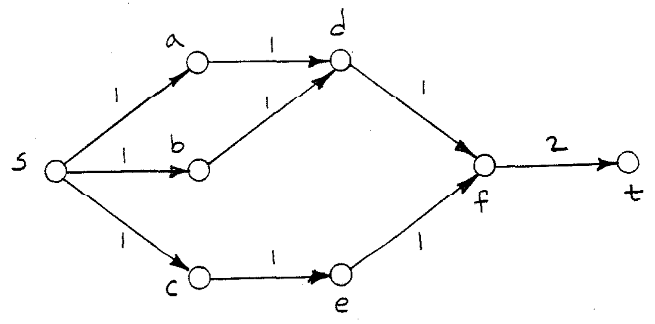

Figure 13: Sub-Optimal Flow Constructed by the CAP Algorithm

example, Figure 13. Three paths are propagated from s to t during phase 1: $\alpha = s, a, d, f, t$, $\beta = s, b, d, f, t$, and $\gamma = s, c, e, f, t$. Each path has a potential flow of one unit. During phase 2, suppose α and β propagate from t to e before γ. Then γ will abort since all the capacity on (f, t) has been reserved. One of α or β will abort at the next stage. Thus, the iteration will complete with only a single unit of flow and a path without a saturated edge, γ. Experiments have shown that for most practical graphs this does not seriously limit performance; however, the fact that a layered max-flow is not constructed each iteration makes it very difficult to prove a tight upper bound on the time complexity of the algorithm.

4.2 The Concurrent Vertex Flow (CVF) Algorithm

Like the CAP algorithm, the CVF algorithm operates by iteratively partitioning the network into layers and then increasing the flow in the layered network. Unlike the CAP algorithm, however, the CVF algorithm does not use augmenting paths. More importantly the CVF algorithm is guaranteed to construct a *max-flow* in a layered network in a single iteration. *Max-flow* here means a flow for which every path from s to t in the layered network has some edge saturated.

The CVF algorithm simultaneously partitions the network into layers and constructs a *max-flow* in the layered network. The flow is constructed by pushing flow requests from s to t and propagating flow acknowledgements and flow rejections back from t to s.

A flow request message, shown in Figure 14 has the following fields

1. `flow` The amount of flow requested by the sender.
2. `edge` The edge across which the request is propagated.
3. `prev vertex` The requesting vertex.

`REQUEST`	`flow`	`edge`	`prev vertex`	`layer`	`tag`

Figure 14: A Request Message

4. `layer` The layer number to be assigned to the destination vertex.

5. `tag` The tag corresponding to this iteration.

On receiving a request message, a vertex, v, compares its tag and label to the tag and layer fields of the message and updates its fields if the tag is stale or the label larger than the layer. If the tag and layer pass inspection, v then scans its adjacency lists for useful edges. The requested flow is arbitrarily partitioned over the useful edges and request messages are sent to the vertices at the opposite end of the useful edges. If more flow was requested of v than available capacity on all useful edges, v rejects the excess flow with a reject message. If all of the requested flow is used up, and more useful edges remain, the next available useful edge `self.next_ edge` and the flow already requested across this edge `self.edge_ flow` is recorded. More flow may be requested across this edge if some other flow request is rejected. The code for processing a request message is shown in Figures 15 and 16.

When a vertex cannot satisfy all or part of a flow request, it rejects the flow it cannot handle with a reject message containing the fields shown in Figure 17.

1. `flow` The amount of flow rejected.

2. `edge` The edge across which the request was propagated.

When a vertex, v, receives a reject message, it first checks if it has any useful edges that have not yet received request messages. If useful edges remain it attempts to request additional flow from those edges. When all the useful edges are exhausted, the vertex sends a reject message for the remaining flow to a vertex that requested flow from it. This code is shown in Figure 18.

Flow acknowledgement begins when a request message reaches the sink, t. The sink acknowledges all flow requests by sending an acknowledge message with the fields shown in Figure 19.

1. `flow` The amount of flow acknowledged.

2. `edge` The edge across which the request was propagated.

As shown in Figure 21, when a vertex receives an acknowledge message, it increases the flow along the specified edge by the acknowledged amount. Pending requests are then dequeued and acknowledged either totally or partially until the acknowledged flow is used up.

```
---------------------------------------------------------------------------
/* all vertices except the sink */
request(flow,prev_vertex,e_j,layer,tag)
{
  /* check layer and tag */
  if((self.tag != tag) || (self.label >= layer)) {
    if(self.tag != tag) {
      self.tag = tag ;
      self.next_edge = self.first_edge ;
      self.edge_flow = 0 ;
    }
    self.label = layer ;

    /* continue request */
    remaining_flow = flow ;
    try_flow(remaining_flow,self.next_edge,self.edge_flow) ;
  }
  /* remember pending request */
  if((flow-remaining_flow) > 0) {
    enqueue_request(flow-remaining_flow,prev_vertex,e_j) ;
  }
  /* reject excess request */
  if(remaining_flow > 0) {
    send(prev_vertex,reject,remaining_flow,e_j) ;
  }
  /* if label < layer, reject message */
  else {
    send(prev_vertex,reject,flow,e_j) ;
  }
}
---------------------------------------------------------------------------
```

Figure 15: CVF Request Propagation

```
--------------------------------------------------------------------------
try_flow(VAR flow,VAR next_edge, VAR edge_flow)
{
  for each edge e_j incident on the current vertex starting with next_edge {
    if(e_j is a forward edge) {
      (self,v_nxt) = e_j ;
      a_j = c_j-f_j-edge_flow ;
    }
    else { /* e_j is a reverse edge */
      (v_nxt,self) = e_j ;
      a_j = f_j-edge_flow ;
    }
    rflow = min(a_j,flow) ;
    if(a_j > 0) {
      send(v_nxt.request,rflow,e_j,layer+1,tag) ;
      flow = flow - rflow ;
    }
    if(flow == 0) {
      next_edge = e_j ;
      edge_flow += rflow ;
      break ;
    }
    else edge_flow = 0 ;
  }
  /* on normal termination, no continuation */
  if(flow >= 0) next_edge = NIL ;
}
--------------------------------------------------------------------------
```

Figure 16: Subroutine Try-Flow

REJECT	flow	edge

Figure 17: A Reject Message

```
--------------------------------------------------------------------------
reject(flow,edge)
{
  /* try remaining edges */
  remaining_flow = flow ;
  if(self.next_edge != NIL) {
    try_flow(remaining_flow,self.next_edge) ;
  }
  /* if there's still some flow, reject it */
  while (remaining_flow > 0) {
    dequeue_request(req_flow,prev_vertex,e_j) ;
    reject_flow = min(req_flow,remaining_flow) ;
    send(prev_vertex,reject,reject_flow,e_j) ;
    if(req_flow > remaining_flow) {
      enqueue_request(req_flow-remaining_flow,prev_vertex,e_j) ;
    }
    remaining_flow = remaining_flow - reject_flow ;
  }
}
--------------------------------------------------------------------------
```

Figure 18: Rejecting Flow

ACK	flow	edge

Figure 19: An Acknowledge Message

```
--------------------------------------------------------------------------
/* executed when the sink receives a request */
request_sink(flow,prev_vertex,e_j,layer,tag)
{
  send(prev_vertex,ack,flow,e_j) ;
}
--------------------------------------------------------------------------
```

Figure 20: Initiating Acknowledgement

```
/* for all vertices except s and t */
ack(flow,e_j)
{
  if(e_j is a forward edge) f_j = f_j + flow ;
  else                      f_j = f_j - flow ;
  remaining_flow = flow ;
  while(remaining_flow > 0) {
    dequeue_request(req_flow,prev_vertex,e_k) ;
    ack_flow = min(req_flow,remaining_flow) ;
    send(prev_vertex,ack,ack_flow,e_k) ;
    if(req_flow > remaining_flow) {
      enqueue_request(req_flow-remaining_flow,prev_vertex,e_j) ;
    }
    remaining_flow = remaining_flow - ack_flow ;
  }
}
```

Figure 21: Propagating Acknowledgement

When either an acknowledge or a reject message reaches the source, the source decrements a pending flow counter. The iteration is complete when the pending flow is reduced to zero. If no acknowledge messages are received during an iteration, a max-flow has been constructed and no further iterations are required.

The major difference between the CAP algorithm and the CVF algorithm is the method of allocating flow to paths. In the CAP algorithm, all potential paths are discovered concurrently without considering possible conflicts. The CVF algorithm on the other hand never generates any conflicts. It only propagates paths that have guaranteed available capacity on their initial segments.

The CVF algorithm, by avoiding conflicts, always constructs a max-flow on the layered network. It does not suffer from the problem of reconvergent fan-in, illustrated in Figure 13, that causes the CAP algorithm to allocate capacity of an edge to two paths that share an initial segment.

The problem with avoiding conflicts is that it limits concurrency. As shown in Figure 22, the CVF algorithm can be reduced to performing sequential depth-first search for the pathological case of a tree with equal weight edges where only one leaf of the tree is effectively connected to the sink. The CAP algorithm, on the other hand, solves the max-flow problem on this network in one iteration. Thus, while the CAP algorithm is limited by fan-in, the CVF algorithm is limited by fan-out.

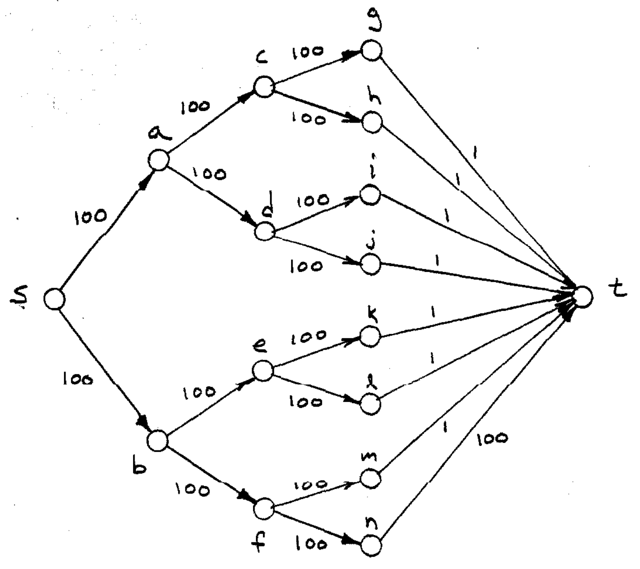

Figure 22: Sequential Execution by CVF

4.3 The Concurrent Augmenting Digraph (CAD) Algorithm

The concurrent augmenting digraph (CAD) algorithm overcomes the problems of the both the CAP algorithm with fan-in and the CVF algorithm with fan-out. The CAD algorithm, like the CAP algorithm, finds all potential augmenting paths regardless of conflicts. Thus, the CAD algorithm has no trouble with fan-out networks such as the one shown in Figure 22. To avoid the problem of reconvergent fan-in, illustrated in Figure 13, the CAD algorithm builds a path *digraph* instead of the path *tree* constructed by the CAP algorithm. In the CAD algorithm each vertex waits until it has received path messages from all of its incoming edges, merges these paths and transmits a single message across each of its outgoing edges.

The CAD algorithm is more efficient than the CAP algorithm both because it succeeds in generating some paths that the CAP blocks, for example Figure 13, and because it reduces the number of messages transmitted. The only disadvantage of the CAD algorithm is that it reduces concurrency. For a vertex, v, to know which of its edges are incoming either the labeling of $v's$ neighbors must be completely finished before constructing the augmenting digraph, or the first phase of the algorithm must be globally synchronized with all layer 1 vertices finishing before any layer 2 vertices begin.

The CAD algorithm proceeds in four phases as follows:

1. Labeling is performed during phase 1. The source sends a label message with a layer field of 1 over all of its useful edges. When a vertex, v, receives a label message it compares the layer and tag fields of the message to its current label and tag updating them appropriately. If the layer is less than the current layer or the tag is different than the current tag, the edge on which the message arrived is labeled incoming. Finally, the vertex transmits a layer message over all of its useful outgoing edges.
2. The augmenting digraph is constructed during phase 2. The source sends a digraph

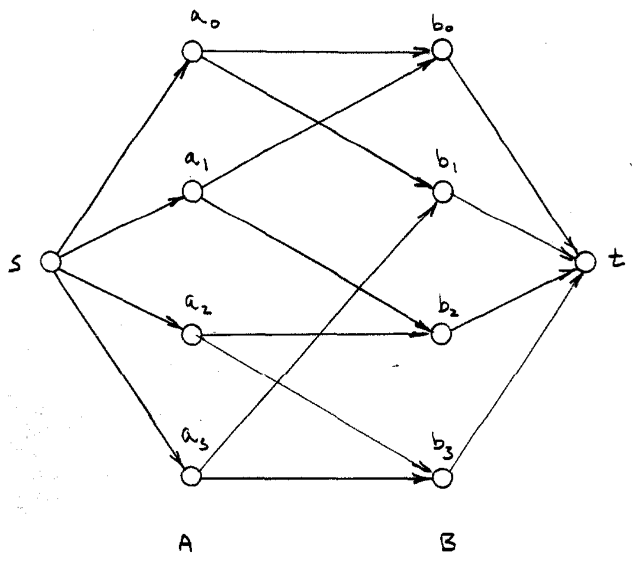

Figure 23: A Bipartite Flow-Graph

message over all its useful edges. When a vertex, v, has received digraph messages over all of its incoming edges, it combines their flows to determine its maximum potential flow and sends digraph messages out over all of its outgoing edges.

3. Reservations for available edge capacity are made during phase 3. When the sink has received digraph messages over all its incoming edges, it starts phase 3 by sending reservation messages back over these edges. When a vertex, v, has received reservation messages over all of its outgoing edges, it arbitrarily divides the reserved capacity over the incoming edges and sends reservation messages back over these edges.
4. Finally, successful portions of the digraph confirm their reservations and increase the flow during phase 4.

The code for the CAD algorithm is similar to the code for CAP and will not be repeated here.

4.4 Distributed Vertices

In all three of the concurrent max-flow algorithms, the source and the sink are bottlenecks that serialize the algorithm. At most one path can be processed by the source or sink per unit time and each path must pass through both the source and sink twice. The problem is especially acute in the case of a flow-graph for solving a bipartite matching problem where the fanout of the source and fanin of the sink is $\frac{|V|}{2} - 1$ as shown in Figure 23.

The source and sink bottlenecks can be removed by distributing these vertices. The only operations performed at the source and sink are:

1. keeping reference counts of pending path messages,

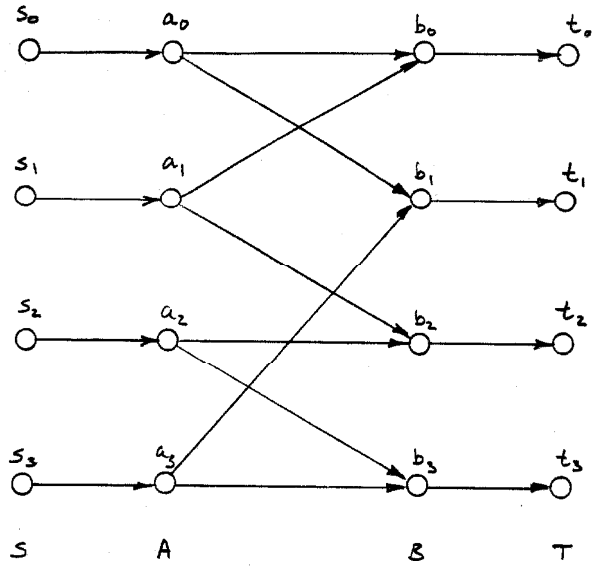

Figure 24: Distributed Vertices

2. adjusting flow on edges adjacent to the source.

Messages to or from the source or sink on a particular edge affect no other edges. Thus, we can split the source and sink into multiple vertices: one for each edge incident on the original source or sink as shown in Figure 24. Adjusting edge flow is performed by the individual source vertices. To determine reference counts, each of the split source and sink vertices keeps its own reference count and a global wire-or bus is used to determine when all reference counts reach zero.

5 Performance

Because of the effects of reconvergent fanout, it is difficult to develop a good bound on the performance of the CAP algorithm. An edge from layer l to layer $l+1$ can be scanned at most $\frac{|E|^l}{l}$ times since there can be no more than $\frac{|E|^l}{l}$ unique paths of length l. Thus, each iteration of the CAP Algorithm requires no more than $O(\frac{|E|^{L+1}}{L})$ operations when the network is partitioned into L layers. Note that these traversals take place in parallel. Since an edge can only be a bottleneck once for a given L [2] at most $|E|$ iterations are performed for each value of L. The depth of the layering, L can be at most $|V|-1$, the number of operations required by the CAP algorithm is bounded by $O(|E|^{|V+2|})$. In Section 6 experimental performance measurements are presented that show that this bound is very pessimistic.

A tighter bound can be placed on the complexity of the CVF algorithm since it constructs a max-flow in the layered network each iteration. As in Karzanov's algorithm, at most $O(|V|^2)$ operations are required each iteration. Since a layered max-flow is constructed each iteration, at most $O(|V|)$ iterations are required. Thus at most $O(|V|^3)$ operations are required.

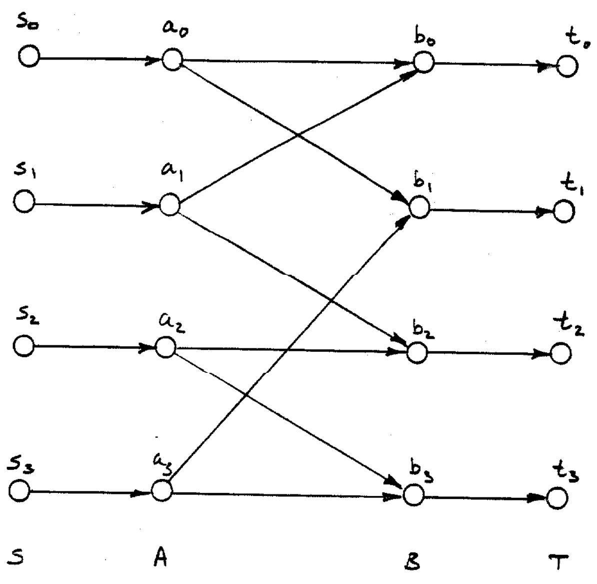

Figure 25: Bipartite Graph with Distributed Sources and Sinks, $|V| = 16$

The CAD algorithm scans each edge at most a constant number of times per iteration and thus requires $O(|E|)$ operations per iteration. Unfortunately, like the CAP algorithm, the CAD algorithm is not guaranteed to construct a layered max-flow each iteration and thus may require as many as $O(|E| \times |V|)$ iterations. This gives an upper bound of at most $O(|E|^2|V|)$ operations for the CAP algorithm.

6 Experimental Results

To characterize the performance of the CAP and CVF algorithms, these algorithms have been programmed and run on a multiprocessor simulator. The simulator provides a binary n-cube message passing environment where one unit of time is required for a message to traverse one communications channel in the cube.

The CAD algorithm was not programmed because of its similarity to the CAP algorithm. The performance of the CAD algorithm is expected to be slightly better than the CAP algorithm.

Randomly generated unit weight bipartite graphs with distributed sources and sinks were used as test cases for the simulations. An example of this class of graph is shown in Figure 25. There are four equal size sets of vertices, sets S, A, B and T. Each source, $s_i \in S$ is connected by a unit weight edge to the corresponding vertex $a_i \in A$. Each vertex $a_i \in A$ has out degree $d = 2$ and is randomly connected by unit weight edges to two vertices $b_j, b_k \in B$. Finally, each vertex $b_i \in B$ is connected by a unit weight edge to the corresponding sink vertex, $t_i \in T$. Solving the max-flow problem on a bipartite graph is equivalent to solving the bipartite matching problem represented by the graph.

Dinic's sequential max-flow algorithm was run to give a performance baseline with which the concurrent algorithms could be compared. A plot of number of operations as a function

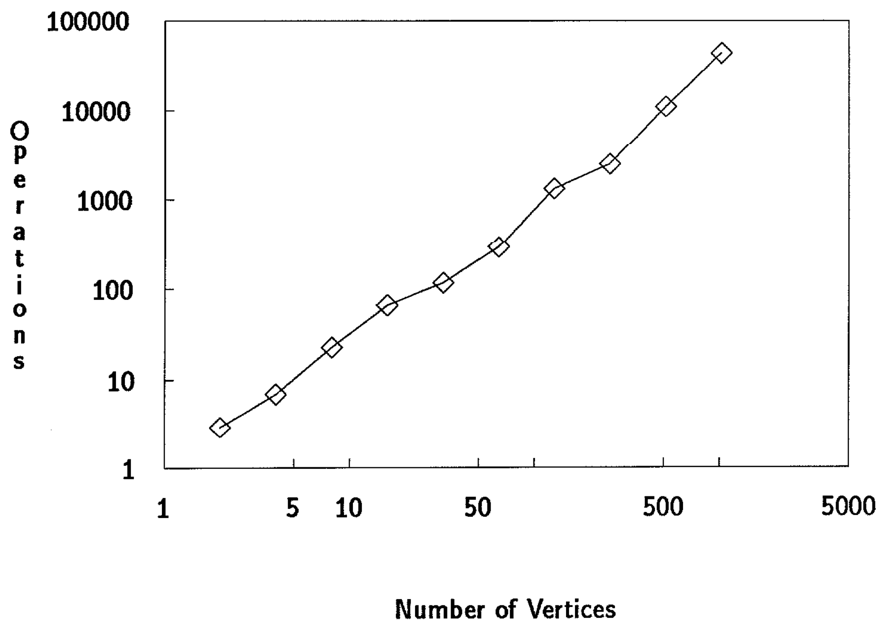

Figure 26: Operations vs. Graph Size for Dinic's Algorithm

of graph size for Dinics algorithm is shown in Figure 26.

Both the CAP and CVF algorithms require fewer operations than Dinic's algorithm to construct a max-flow in the test graphs. The number of operations as a function of graph size for the CAP algorithm is shown in Figure 27. The corresponding plot for the CVF algorithm is shown in Figure 28. For both algorithms, the number of operations grows slightly faster than linearly with the size of the graph, with the CAP algorithm requiring almost exactly twice as many operations per vertex as the CVF algorithm.

Two differences between the CAP and CVF algorithms account for this factor of two in performance:

1. The CAP algorithm is a three-phase algorithm (request, reserve and confirm) while the CVF algorithm requires only two phases (request and acknowledge).
2. The CAP algorithm propagates paths that have conflicts over edge capacity while the CVF algorithm only propagates paths that have guaranteed capacity back to the source. Thus, the CAP algorithm performs a fair amount of unnecessary work propagating paths that later fail to reserve the required resources.

While the CAP algorithm requires more operations than the CVF algorithm, it actually has slightly better performance because it has greater concurrency. The speedup of the CAP and CVF algorithms as a function of the number of processors is shown in Figures 29 and

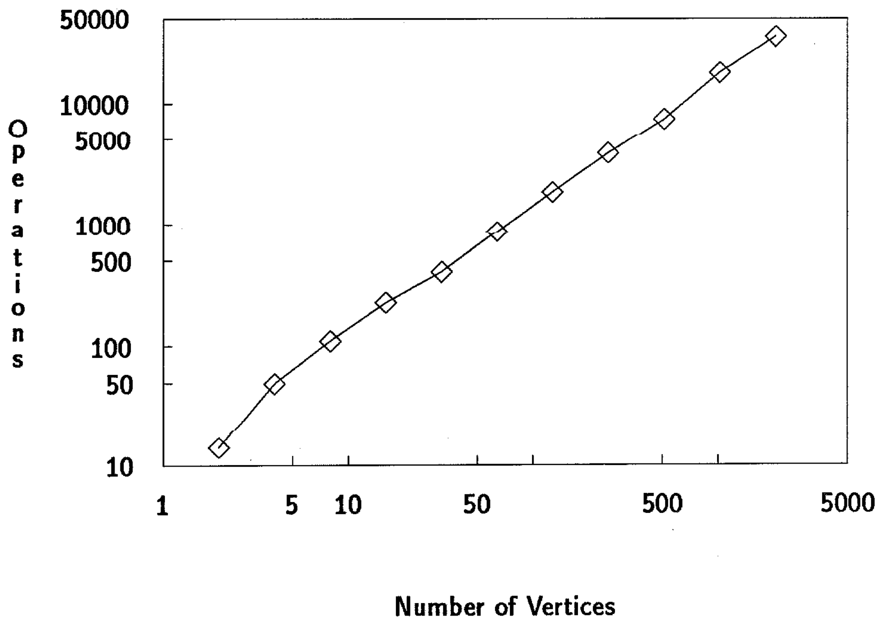

Figure 27: Operations vs. Graph Size for the CAP Algorithm

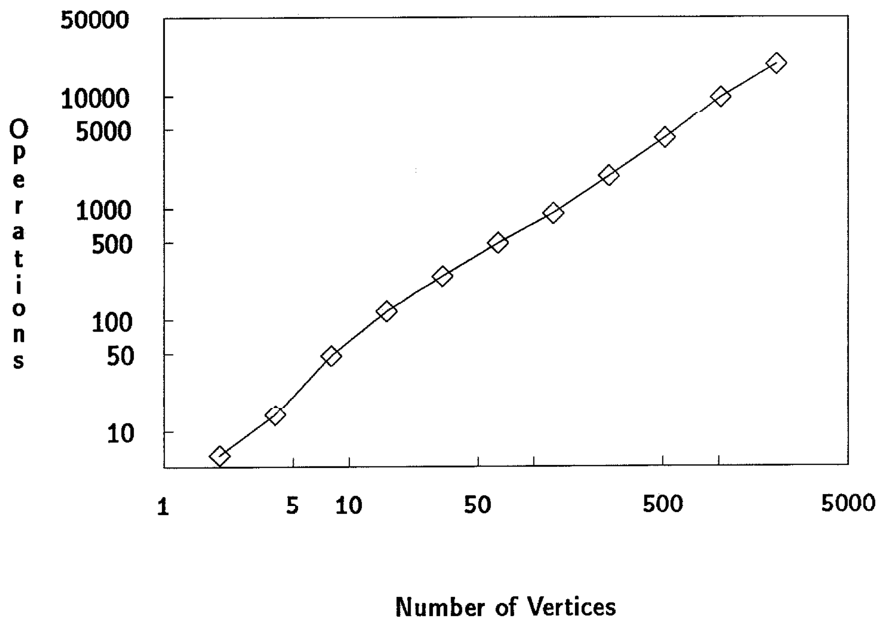

Figure 28: Operations vs. Graph Size for the CVF Algorithm

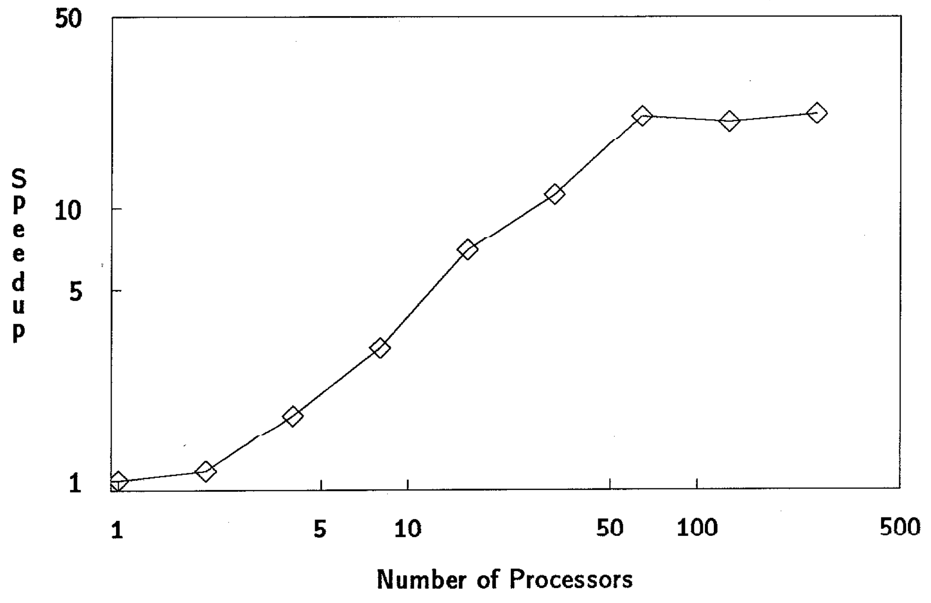

Figure 29: Speedup vs. Number of Processors for the CAP Algorithm, $|V| = 256$

30. For both algorithms the speedup grows slightly slower than linearly with the number of processors until an upper bound is reached. The CAP algorithm gives more than twice the speedup of the CVF algorithm. This improvement in concurrency more than offsets the greater number of operations performed by the CAP algorithm.

Figures 31 and 32 show the peak speedup of the CAP and CVF algorithms as a function of graph size. As indicated by the slope of these plots, the concurrency of these algorithms is proportional to $\sqrt{|V|}$. This square-root concurrency is due to the *wavefront* nature of the max-flow algorithms. The algorithms operate by propagating *wavefronts* of activity between the sources and sinks. Only vertices along the wavefront are active. For a large number of graphs, including the test cases used for the simulations shown here, the average length of the wavefront tends to be $\sqrt{|V|}$.

7 Conclusion

This paper has developed three new concurrent algorithms for the max-flow problem: the CAP, CVF and CAD algorithms. These algorithms are designed for message passing concurrent computers and achieve concurrency by investigating multiple paths concurrently.

The performance analysis of Section 5 shows that the CVF algorithm has a worst-case complexity equal to the best known sequential algorithm. However, sequential measures of performance are not always relevant to concurrent algorithms. The CAP algorithm

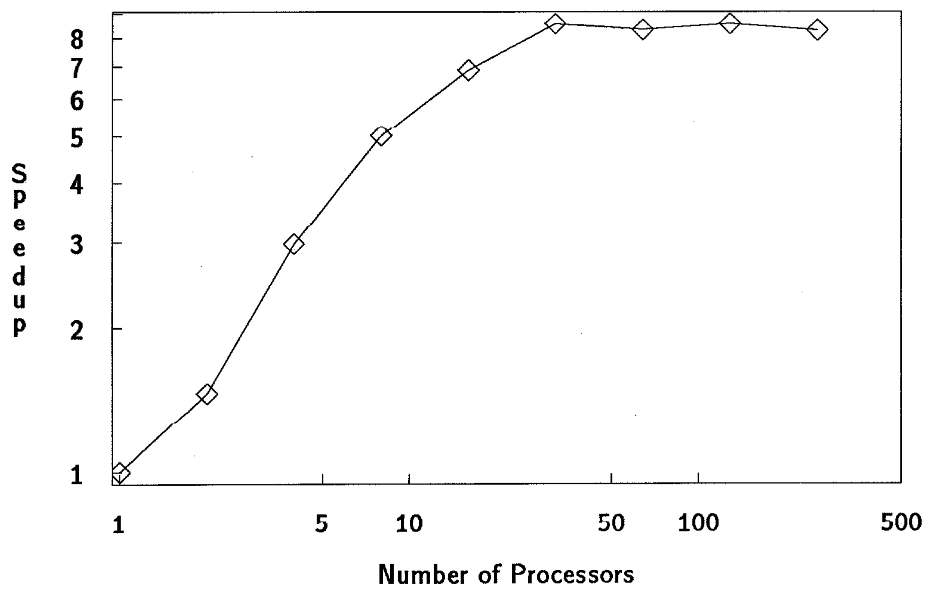

Figure 30: Speedup vs. Number of Processors for the CVF Algorithm, $|V| = 256$

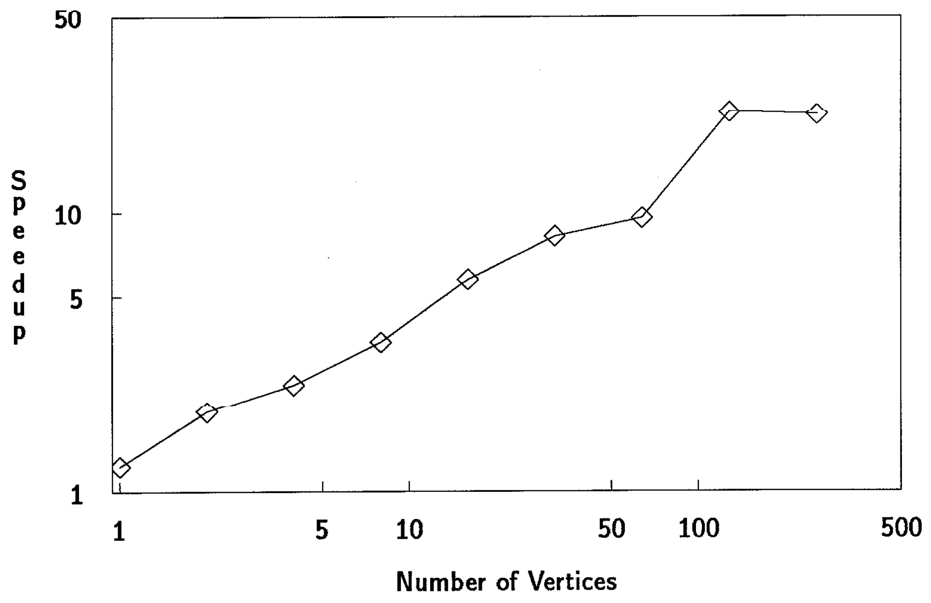

Figure 31: Speedup vs. Number of Vertices for the CAP Algorithm

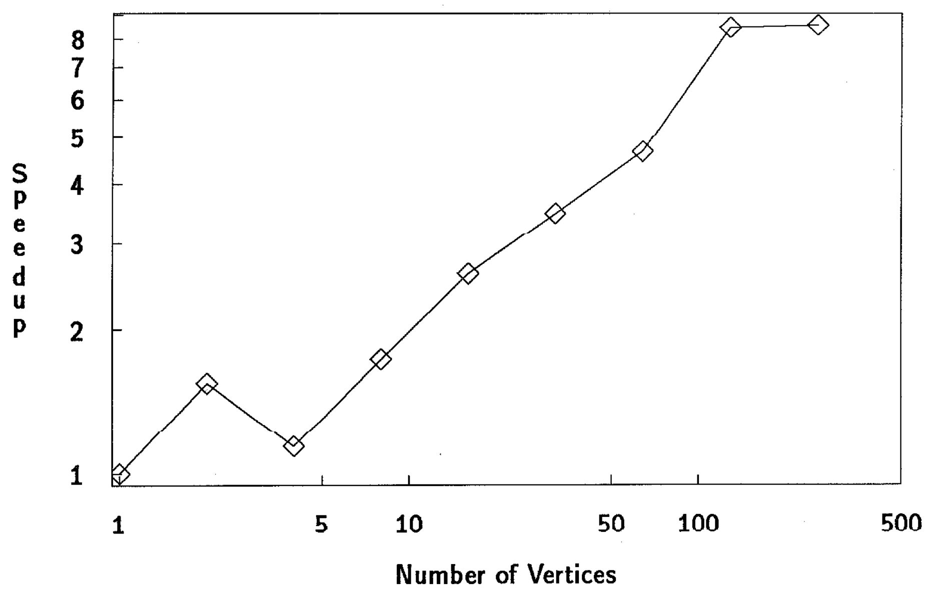

Figure 32: Speedup vs. Number of Vertices for the CVF Algorithm

which has an abysmal worst-case performance and requires more operations than the CVF algorithm is in practice faster than the CVF algorithm because of its greater concurrency.

References

[1] Ford, L.R., Jr. and Fulkerson, D.R., *Flows in Networks*, Princeton University Press, 1962.

[2] Edmonds, J. and Karp, R.M., "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *JACM*, Vol. 19, 1972, pp. 248-264.

[3] Even, Shimon, *Graph Algorithms*, Computer Science Press, 1979, pp. 97-103.

[4] Hu, T.C., *Combinatorial Algorithms,* Addison-Wesley, 1982, pp. 53-56, 59-60.