Power Transmission Control Using Distributed Max Flow

A. Armbruster [*†] M. Gosnell [*†] B. McMillin [†]
{aearmbru, mrghx4, ff}@umr.edu
Department of Computer Science

M. L. Crow [†]
crow@umr.edu
Electrical and Computer Engineering

Intelligent Systems Center
University of Missouri-Rolla, Rolla, MO 65409-0350

Abstract

Existing maximum flow algorithms use one processor for all calculations or one processor per vertex in a graph to calculate the maximum possible flow through a graph's vertices. This is not suitable for practical implementation. We extend the max flow work of Goldberg and Tarjan to a distributed algorithm to calculate maximum flow where the number of processors is less than the number of vertices in a graph. Our algorithm is applied to maximizing electrical flow within a power network where the power grid is modeled as a graph. Partitioning and mapping of graph vertices to processors is discussed in relation to the power grid problem where the number of controllers in the network is less than the number of vertices. Error detection measures are included to detect problems in a simulated power network. We show that our algorithm is successful in executing quickly enough to prevent catastrophic power outages.

Keywords: Fault Injection, FT Algorithms, FT Communication, maximum flow, power system.

[*]This research supported in part by NSF IGERT grant DGE-9972752

[†]This research is supported in part by the NSF MRI grant CNS-0420869 and in part by the UMR Intelligent Systems Center.

1 Introduction

A maximum flow (max flow) algorithm calculates the maximum flow possible between two given vertices through a connected graph. Although sequential and distributed algorithms exist for calculating max flow, they are not practical for all applications. Existing max flow algorithms use either one processor or one processor per vertex in a graph. In a real-world system, such as power flow control, many vertices will be computed by a single processor. We present a distributed algorithm to calculate the maximum flow where the number of processors is less than the number of vertices in the graph. Our algorithm is applied to electrical flow in a power network where the power grid is modeled as a graph. This mapping must lessen the time required to compute max flow over that of a single vertex per processor, thus partitioning and mapping of graph vertices to processors is discussed in relation to the power grid problem. Fault tolerance measures are included in our application to govern problems in the simulated power network.

Section 2 gives a background of maximum flow within a graph and its relationship to a power network. A distributed maximum flow algorithm is presented in Section 3 to perform maximum flow calculations where the number of processors is much smaller than the number of vertices in the graph. Error detection based on assertion checking is used to produce a fail-stop system. Section 4 presents timing and error detection results from applying the algorithm to a power network simulation. A conclusion and future work follow in Sections 5 and 6, respectively.

2 Background

2.1 Power Transmission as a Graph

The power transmission grid can be modeled as a directed graph with power flowing from generators (sources) to loads (sinks). Given a graph $G'(V', E')$ where the set of vertices, V', corresponds to the buses of the power network, the power flowing between vertices $v_i, v_j \in V'$ is represented by an edge $(v_i, v_j) \in E'$.

For each vertex $v \in V'$, power in must equal power out. Generators, however, can be modeled as outputting power without any input and loads can be modeled as power in with no power out. This can be modeled as a flow problem by adding to the graph a virtual source, s, that connects to all the generators, and a virtual sink, t which connects to all loads [1]. The virtual source can supply infinite power, but its outward arcs are limited by the generator capacities. The virtual sink can potentially consume an infinite amount of power flow, but is constrained by the inward arcs representing loads. The resultant graph, $G(V, E)$, is shown in Figure 1.

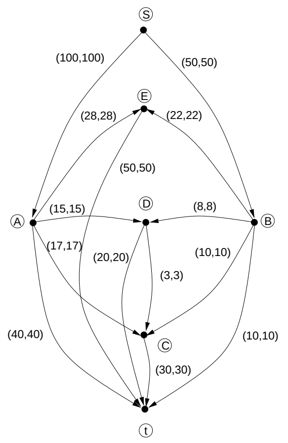

Figure 1: Power network shown as a directed flow graph with virtual vertices s and t. Edges are labeled with (flow, capacity). The capacity over all edges is fully utilized.

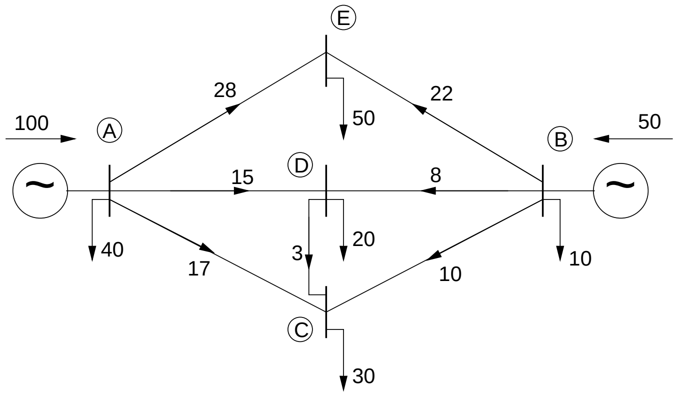

Figure 2: Example power system network with generators of 100 at A and 50 B and loads of 40, 50, 20, 30, and 10.

A power network $G'(V', E')$ corresponding to the graph from Figure 1 is shown in Figure 2. The power network example shows generators at locations A and B which were supplied by s in Figure 1. The arrows not connected to another power bus, or vertex, represent the edges to the virtual sink. It can be seen from Figure 2 that the power in each edge of G' corresponds to the maximum flow in each edge of G.

2.2 Controlling Power Flow

Cascading failures are the most severe form of failure that can occur in a power system. A cascading failure occurs when the loss of one line leads to the loss of another until the transmission grid can no longer sustain the power flow. Cascading failures have occurred in the United States in the 1960's, 1970's, and 2003. One reason for cascading failures is that present control of the transmission grid limits power service providers to selecting which lines are available, not how much power flows through them. When a power line is lost, remaining power redistributes throughout the grid according to the laws of physics. Since natural power flow is not determined by the capacity of the line, a line can be overloaded even though there is enough remaining capacity in the grid to transfer the power. To mitigate overloads, power companies either manually trip off a line when too much power is flowing through or redistribute generation. To use the other transmission lines that can handle the flow, power electronic devices need to be used to change properties of the lines so that power will choose to use the capacity of all lines. Flexible AC Transmission System (FACTS) devices can change power line properties and control the amount of flow on a line, preventing cascading failures due to line loss [2]. However, determining and coordinating FACTS devices is crucial to finding settings that will avoid cascading failures.

A natural approach to calculate flow and determine FACTS settings is to model the power flow as a maximum flow problem. The approach of modeling power flow as max flow was first reported in [3]. Using techniques from Section 2.1, one can model the power network as a graph with vertex set V. Each FACTS device, $p_i \in P$ contains power electronics and an embedded computer and computes settings for a subset of vertices V_{p_i} using the state of the power system as a weighted graph of line flows. The state can be obtained through the techniques described in [4]. A distributed max flow algorithm can then be used to determine settings to force the flow on the lines v_k, v_l where the FACTS devices are located. In the worst case, if the power network can no longer satisfy the load, max flow can also shed load. Experimentally [3], only a small number of FACTS devices, which contain the necessary control hardware and an embedded computer, are needed to control a large power network.

3 Max Flow

3.1 Sequential Max Flow

The sequential maximum flow (max flow) algorithm was originally developed by Fulkerson and Danzig in 1955 [5] and was further formalized by Ford and Fulkerson in 1956 [6]. While this work provides a method to find the max flow through a graph, it does not guarantee that the solution will be found in finite time. Other solutions to the max flow problem continued to be developed. The new algorithms fall into groupings that include blocking, scaling and push/relabel. The blocking method, pioneered by [7], includes works by [8, 9]. Instead of trying to find just one path at a time, blocks are formed to allow several paths from the source to the sink to be determined at once. Scaling methods are used in [10, 11]. The push and relabel method, pioneered by Goldberg [12] was further refined with Tarjan [13]. The push and relabel algorithm builds a ladder with vertices

```
**Goldberg and Tarjan Max Flow Algorithm:**
**MaxFlow(V, E, s, t, c)**
 1. $\forall (v,w) \in (V-s) \times (V-s), f(v,w) = 0; f(w,v) = 0$
 2. $\forall v \in V, f(s,v) = c(x,v); f(v,s) = c(s,v)$
 3. $d(s) = |V|$
 4. $\forall v \in V - \{s\}, d(v) = 0; e(v) = f(s,v)$
 5. while $\exists$ a Relabel or Push operation for which the precondition is true, execute the operation.

**Push(v,w)**
Precondition: $e(v) > 0$ and $v \notin \{s,t\}$ and $r_f(v,w) > 0$ and $d(v) = d(w) + 1$
 1. Send $\Delta = min(e(v), r_f(v,w))$ units of flow from v to w as follows:
      $f(v,w) = f(v,w) + \Delta; f(w,v) = f(w,v) - \Delta$
      $e(v) = e(v) - \Delta; e(w) = e(w) + \Delta$

**Relabel(v)**
Precondition: $e(v) > 0$ and $v \notin \{s,t\}$ and $\forall w \in V r_f(v,w) > 0 \rightarrow d(v) = d(w) + 1$
 1. $d(v) = min\{d(w) + 1 | (v,w) \in A_f\}$
```

Figure 3: Basic Push-Relabel Algorithm. V is the set of vertices. E is the set of edges. c is a function that maps arcs to the capacity of the edges. A_f is the set of residual edges that are not at capacity. $f(v,w)$ is the flow on the edge (v,w). $r_f(v,w) = c(v,w) - f(v,w)$. $e(v)$ is the excess flow at the vertex. $d(v)$ is the distance of the vertex from the sink.

on various rungs. The flow can only pass from one vertex to a vertex a level below it [13]. Goldberg and Tarjan's push-relabel max flow algorithm can be found in Figure 3.

3.2 Distributed Max Flow

In addition to the many sequential max flow algorithms, several parallel or distributed max flow algorithms have been developed [13, 14, 15]. Three solutions were presented by Goldberg and Tarjan: a parallel solution using a PRAM model, a synchronous distributed model, and an asynchronous distributed algorithm that requires one processor per vertex [13]. The asynchronous variant works by sending messages listed in Definition 3.1, which constitute the message sets in Definition 3.2. When flow is pushed from one graph vertex to another, a message *PFm* is sent. The receiving vertex first checks the distance in the message to verify that the message can be accepted. If the distance sent is one more than the current distance, an accept message *AFm* is returned; otherwise, a reject message *RFm* is sent as the reply. A distance message, *Distm*, is sent to every neighboring vertex every time the distance at a vertex is updated. Although there are several distributed max flow algorithms, for example the Goldberg-Tarjan asynchronous algorithm, none of them address the problem of executing where a single processor handles multiple vertices.

Definition 3.1 Max Flow Message Types

Cm: The message currently being processed.

PFm: A message attempting to push flow to a neighboring vertex.

AFm: A message replying to a *PFm* indicating the requested flow is accepted.

RFm: A message replying to a *PFm* indicating the requested flow cannot be accepted.

Distm: A distance message indicating an update to a nodes distance.

MFm: A max flow message which is a *PFm*, *AFm*, *RFm*, or *Distm.*

Tm: A token message used to get a snapshot to determine if the algorithm is finished.

Dm: A done message indicating the algorithm is finished.

Fm: A fault message indicating a fault was detected.

Ctrlm: A control message which is a *Tm*, *Dm*, or *Fm.*

Definition 3.2 Message Communication Sets

$\boldsymbol{MFS}_{v_i,v_j}$*:* A set of *MFm*'s sent from v_i to v_j where $v_i, v_j \in V$.

$\boldsymbol{MFS}_{v_i,*}$*:* A set of *MFm*'s sent from $v_i \in V$ to all other $v \in V - v_i$.

$\boldsymbol{MFS}_{*,v_j}$*:* A set of *MFm*'s received in $v_j \in V$ to all other $v \in V - v_j$.

$\boldsymbol{MFS}_{i,j}= \bigcup \mathrm{MFS}_{v_m,v_n} \mid \forall v_m \in V_{p_i}, v_n \in V_{p_j}$.

$\boldsymbol{MFS}_{i,*}= \bigcup \mathrm{MFS}_{v_m,v_n} \mid \forall v_m \in V_{p_i}, v_n \in V - V_{p_i}$.

$\boldsymbol{MFS}_{*,j}= \bigcup \mathrm{MFS}_{v_m,v_n} \mid \forall v_n \in V_{p_j}, v_m \in V - V_{p_j}$.

$\boldsymbol{MFS}= \bigcup \mathrm{MFS}_{v_i,*} \mid \forall v_i \in V$.

The algorithm of this paper was adapted from Goldberg and Tarjan's push-relabel maximum flow algorithm mapping many vertices to one processor. The code for the blocked style max flow is presented in Figures 4 and 5. The algorithm uses the same accept and reject message passing system using messages from Definition 3.1, but instead of instantly sending the messages, they are queued at the processor. If message *Cm* is being delivered to a vertex also located on the same processor, *Cm* is processed immediately and the messages generated are added to the end of its FIFO queue. If message *Cm* needs to be delivered to another processor, *Cm* is put in the other process' queue and progress continues with the next message. Inherently, due to communication delays, it takes considerably longer to process a message that must be sent to another processor. This difference in time for sending messages to the same or different processes means partitioning and mapping of vertices to processes is an important factor to this algorithm.

Grouping vertices in the same process saves time only if communication can be overlapped with computation ($\exists p_i \in P \; \exists v_k, v_j \in V_{p_i} \; \exists m \in MFS_{v_k,v_j}$ and $\forall p_j, p_k \in P$ such that p_j does not

```
Blocked Max Flow as executed on $p_i \in P$
1. Assign an initial flow ($f(v_i, v_j) = 0\ \forall v_i, v_j \in E | v_i \in V_{p_i}$)
2. Assign an initial distance ($d(v_i) = 1 \forall v_i \in V_{p_i}$)
3. Assign an initial distance to the sink ($d(t) = 0$)
4. Assign an initial distance to all sources($d(v_i) = |V|$ for all source vertices in $V$)
5. $\forall v \in V \mid v\ is\ a\ source$, push flow from the source.
6. while (not completed)
   (a) If $q_i = \emptyset$, wait until $q_i \neq \emptyset$.
   (b) Let $Cm = First(q_i)$
   (c) If $Cm$ is a $Ctrlm$, ControlProcessing($Cm$)
   (d) Else
       i. If $intended\_vertex(Cm) \notin V_{p_i}$, QMR($Cm$, $p_j$) $\mid intended\_vertex(Cm) \in V_{p_j}$.
       ii. If $intended\_vertex(Cm) \in V_{p_i}$ and $Cm$ is $PFm$ or $RFm$, set all $Tm$ in Token List to
           busy.
       iii. If $intended\_vertex(Cm) \in V_{p_i}$ and $Cm$ is not $Ctrlm$, VertexProcessing($Cm$).
   (e) $\sigma_i = \sigma_i \bigcup Cm$

ControlProcessing($Cm$)
1. If $Cm$ is a $Tm$, check to see if a matching $Tm$ has already been received. Token Mes-
   sage's $Tm_p$ and $Tm_q$ match if the $initiating\_process(Tm_p) = initiating\_process(Tm_q)$ and
   $token\_number(Tm_p) = token\_number(Tm_q)$. Let $Tm_c$ be the $Tm \in$ Token List matching $Cm$
   if one exists.
   (a) If $Cm \in$ Token List decrease the count on $Tm_c$ by one.
   (b) If $Cm \notin$ Token List, create $Tm_c$ setting $count(Tm_c) = |N_{p_i}| - 1$,
       $initiating\_process(Tm_c) = initiating\_process(Cm)$, $sender(Tm_c) = sender(Cm)$
       and $token\_number(Tm_c) = token\_number(Cm)$. Let Token List = Token List $\bigcup Tm_c$.
       $\forall p \in N_{p_i} - sender(Cm)$, QMR($Tm_c$, $p$).
   (c) If $Tm_c$ is not busy and $Cm$ is busy, set the $Tm_c$ as busy.
   (d) If $count(Tm_c) = 0$ and $p_i \neq initiating\_process(Tm_c)$, QMR($Tm_c$, $sender(Tm_c)$).
   (e) If $count(Tm_c) = 0$ and $p_i = initiating\_process(Tm_c)$ and $Tm_c$ is not busy, $\forall p \in N_{p_i}$
       QMR($Dm$, $p$). Set the process as completed.
2. If $Cm$ is a $Dm$, $\forall p \in N_{p_i}$ QMR($Dm$, $p$). Set the process as completed.
3. If $Cm$ is a $Fm$, $\forall p \in N_{p_i}$ QMR($Fm$, $p$). Set the process as completed and set the execution
   as faulty.
```

Figure 4: Blocked Distributed Push-Relabel Algorithm for Max Flow. Each $p_i \in P$ runs a copy of this algorithm.

cause p_k to wait for communication with p_j while p_j handles local messages). Definition 3.6 and Theorem 3.1 show that the new algorithm requires no more messages between processes than the algorithm in [13]. A reasonable mapping of vertices to processes increases the number of local messages $m \in MFS_{v_i,v_j}$, where $\exists p_i \in P\ \exists v_i, v_j \in V_{p_i}$. Theorem 3.2 places a bound on the number of messages that p_k could be waiting to complete processing before being allowed to communicate

```
**VertexProcessing(Message)**
1. Let $Cm$= Message. Let IV=$intended\_vertex(Cm)$.
2. If $Cm$ is a $PFm$ and IV is not a source or sink and $d(Cm) = d(IV) + 1$, let $e(IV) = e(IV) + flow(Cm)$, let $w = sender(Cm)$, let $f(IV, w) = f(IV, w) - flow(Cm)$, and create $AFm_r$ setting $intended\_vertex(AFm_r) = sender(Cm)$ and $sender(RFm_r) = IV$. Execute AddLocalMessage($AFm_r$). If $message\_count$(IV) = 0, execute Pulse(IV).
3. If $Cm$ is a $PFm$ and IV is not a source or sink and $d(Cm) \neq d(IV) + 1$, create $RFm_r$ setting $intended\_vertex(RFm_r) = sender(Cm)$, $sender(RFm_r) = IV$, and $flow(RFm_r) = flow(Cm)$. Execute AddLocalMessage($RFm_r$).
4. If $Cm$ is a $PFm$ and IV is a source or sink, create $AFm_r$ setting $intended\_vertex(AFm_r) = sender(Cm)$. Execute AddLocalMessage($AFm_r$). Create $Tm_{init}$ setting $initiating\_process(Tm_{init}) = p_i$ and $token\_number(Tm_{init}) = |\{t|t \in$ and $initiating\_process(t) = p_i\}| + 1$. Let Token List = Token List $\bigcup Tm_{init}$. $\forall p \in N_{p_i}$, execute QMR($Tm_{init}$, $p$).
5. If $Cm$ is an $AFm$, let $message\_count$(IV) = $message\_count$(IV) − 1. If $message\_count$(IV) = 0 and $e$(IV) > 0, Pulse(IV).
6. If $Cm$ is a $RFm$, let $w = sender(Cm)$, $message\_count$(IV) = $message\_count$(IV) − 1, $e(IV) = e(IV) + flow(Cm)$ and $f(IV, w) = f(IV, w) - flow(Cm)$. If $message\_count$(IV) = 0, Pulse(IV).
7. If $Cm$ is a $Distm$, update IV's knowledge of $sender(Cm)$'s distance.

**Pulse(IV)**
1. Let continuePulsing = true
2. while (continuePulsing is true)
    (a) Until $e(IV) = 0$, $\forall w$ such that $d(w) = d(IV) - 1, r_f(IV, w) > 0$, create a $PFm$, $PFm_n$, setting $intended\_vertex(PFm_n) = w$, $d(PFm) = d(IV)$, $sender(PFm_n) = IV$ and $flow(PFm_n) = min(e(IV), r_f(IV, w))$. Let $message\_count$(IV) = $message\_count$(IV) + 1, $e(IV) = e(IV) - flow(PFm_n)$ and $f(IV, w) = f(IV, w) + flow(PFm_n)$. Execute AddLocalMessage($PFm_n$).
    (b) If $message\_count$(IV) = 0 and $e$(IV) > 0, set $d$(IV) = $min\{d(w)|r_f(IV, w) > 0\}$. Create $Distm$, $Distm_k$, and let $d(Distm_k) = d$(IV). $\forall v \in \delta_{IV}$, execute AddLocalMessage($Distm_k$).
    (c) If $message\_count$(IV) > 0, let continuePulsing = false.

**QMR($m$, $p_j$)**: Once $q_j = \emptyset$, on $p_j$, AddLocalMessage($m$).

**AddLocalMessage($m$)**: Prior to first execution, $queueCounter_i = 0$.
1. $queueCounter_i = queueCounter_i + 1$
2. Let $messageId(m) = queueCounter_i$.
3. $\rho_i = \rho_i \bigcup m$
```

Figure 5: Blocked Distributed Push-Relabel Algorithm for Max Flow Continued. Each $p_i \in P$ runs a copy of this algorithm.

with p_j. Experimental times for the power flow problem are reported in Section 4.2.

Definition 3.3 *Queue Definitions*

Let q_i be the message queue on process $p_i \in P$.

Let ρ be the set of messages added to q_i. At any given time $\rho_i = \bigcup \text{MFS}_{v_i,*} | v_i \in V_{p_i}$.

Let σ_i be the set of messages removed from q_i, thus, $q_i = \rho_i - \sigma_i$.

Let $First(q_i) = m | messageId(m) = min(messageId(l)\ \forall\, l \in q_i)$.

Definition 3.4 *Time Definitions*

Let τ be the time to process a message locally.

Let κ be the time from removing a message from q_i, through adding the message to q_j, and processing the message locally on p_j.

Definition 3.5 *Graph Definitions*

Let P be the set of processors.

Let V' be the set of vertices.

Let $V = V' \cup \{s, t\}$.

Let E' be the set of edges. E' consists of pairs (v_i, v_j) where $v_i, v_j \in V$.

Let E be the set of edges such that if $(v_i, v_j) \in E'$, then $(v_j, v_i) \in E$. If $(v_j, v_i) \notin E', c(v_j, v_i) = 0$.

Let V_{p_i} be the set of vertices processed on $p_i \in P$.

Let E_{p_i} be the set of edges entirely on process $p_i \in P$. E_{p_i} consists of pairs (v_i, v_j) where $v_i, v_j \in V_{p_i}$.

Let δ_{v_i} be the set of edges connected to $v_i \in V$. δ_{v_i} consists of pairs (v_i, v_j).

Let N_{p_i} be the set of neighbors of $p_i \in P$. $N_{p_i} = \{p_j | p_j, p_i \in P, \forall v \in V_{p_i} \forall w \in V_{p_j} \exists (v, w) \in E$ or $(w, v) \in E\} - \{p_i\}$

Definition 3.6 *Let Γbe the fraction of messages that are sent between all processes in P.*

$$\Gamma = \frac{\sum_{k=1}^{n} |\text{MFS}_{k,*}|}{\sum_{i=1}^{m} |\text{MFS}_{v_i,*}|}$$

Theorem 3.1 $\Gamma \leq 1$ *if* $|P| < |V|$.

Proof: Due to the pigeon hole principle, when $|P| < |V|$ $\exists p_k \in P$ such that $|V_{p_k}| > 1$. If $\exists p_k \in P$ and $\exists v_i, v_j \in V_{p_k}$ such that $MFS_{v_i,v_j} \neq \emptyset$, then $\Gamma < 1$ by definitions 3.6 and 3.2. If $\forall p_k \in P\ \forall v_i, v_j \in V_{p_k} MFS_{v_i,v_j} = \emptyset$, $\Gamma = 1$ by definitions 3.6 and 3.2. $\square$

Theorem 3.2 *The number of messages added to ρ_i on process $p_i \in P$ without an external communication ranges between* 1 *and* $((2|V|-1) \times |V_{p_i}|) + ((2|V|-1) \times |E_{p_i}|) + ((2|V|-1) \times 2(|V_{p_i}||E_{p_i}|)) + ((2|V|-1)(|V_{p_i}|-1))$.

Proof: External communication occurs in p_i during step 6a in Blocked Max Flow (Figure 4) only when $q_i = \emptyset$ or during QMR. The longest sequence of messages without external communication is the longest sequence of messages, $s = \langle m_1, m_2, \cdots, m_n \rangle$, such that $\forall m_k \in s, intended\_vertex(m_k) \in V_{p_i}$ and $q_i \neq \emptyset$ between consecutive executions of 6 in Blocked Max Flow. $|s|$ is maximal when $\exists v_k \in V_{p_i} \forall v \in V - v_k \; d(v) = 0$ when $m_1 = First(q_i)$ and $\forall m_{n+1}, m_{n+2}, \cdots \; d(v)$ does not change. That sequence is equivalent to computing max flow on the smaller graph, $G(V_{p_i} - v_k, E_{p_i})$. From Lemmas 3.8, 3.9, and 3.10 in [13], the number of relabelings per vertex is at most $2|V| - 1$, the number of saturating push operations in the sequence is at most $(2|V| - 1)|E_{p_i}|$, and the number of nonsaturation push operations is at most $(2|V| - 1) \times 2(|V_{p_i}||E_{p_i}|) + (2|V| - 1)(|V_{p_i}| - 1)$. Thus, sequence uses at most $(2|V|-1) \times |V_{p_i}| + (2|V|-1) \times |E_{p_i}| + (2|V|-1) \times 2(|V_{p_i}||E_{p_i}|) + (2|V|-1)(|V_{p_i}|-1)$ messages. $\Box$

Theorem 3.3 *With a reasonable mapping of vertices to processors, the algorithm runs in* $\frac{|\text{MFS}| \times \tau}{|P|} + \Gamma \times |\text{MFS}| \times (\kappa - \tau)$.

Proof: A reasonable mapping will try to balance the processing time on each processor. Since processing is based upon messages, $|MFS| \times \tau$ time is needed to process each message, which becomes $\frac{|MFS| \times \tau}{|P|}$, when processing is balanced. With any distributed algorithm, time will be needed for communication. In this case, the additional time is $\Gamma \times |MFS| \times (\kappa - \tau)$. $\Box$

It remains to be shown that the algorithm of this paper correctly computes the maximum flow. The approach is to show that it preserves the correctness of the original Goldberg and Tarjan algorithm.

Theorem 3.4 *If all messages generated by the vertices are delivered, the flow at termination is a maximum flow.*

Proof: In [13], the original algorithm was proven to be correct. Since our algorithm generates the same messages, it is also correct as long as the messages are delivered. $\Box$

Theorem 3.5 $\forall p_i \in P, \forall m \in \rho_i, \exists p_j \in P$ *such that* $m \in \sigma_j$, $\forall i, j$.

Proof: A message may be delivered to a vertex on the same process, $i = j$, or to a different process, $i \neq j$. Assume $i = j$. Given steps 6a and 6b and that *messageId*'s form a monotonically increasing sequence, $\forall m \in \rho_i, m = First(q_i)$ only once. Thus, each execution of step 6a and 6b, increases $messageId(First(q_i))$. Given that the general algorithm terminates [13], the set of *messageId*'s is finite. Thus, $\forall p_i \in P, \forall m \in \rho_i, \exists p_j \in P$ such that $m \in \sigma_j$ when

$i = j$. Given the proof for $i = j$, when $i \neq j$, it must be shown that $\forall p_i \in P, \forall m \in \rho_i$ where $intended\_vertex(m) \notin V_{p_i}, \exists p_j \in P$ such that $m \in \rho_j$ and $intended\_vertex(m) \in V_{p_j}$. When $intended\_vertex(m) \notin V_{p_i}$, step 6(d)i of Blocked Max Flow is executed. In that step, QMR is executed. QMR does not terminate until $q_j = \emptyset$. From Theorem 3.2, there is a bound on q_j, which along with the proof for $i = j$ gives that eventually, $q_j = \emptyset$, and QMR will terminate. Thus, $\forall p_i \in P, \forall m \in \rho_i, \exists p_j \in P$ such that $m \in \sigma_j$, $\forall i, j$. $\square$

3.3 Organization of Vertices to Processors

As mentioned in Section 2.2, a power system will have $|P|$ processors, see Definition 3.5, each $p \in P$ located on a FACTS device which execute all max flow calculations over $|V|$ vertices using message passing. Mapping vertices to processors to minimize Γ is the well-known partitioning and mapping problem. For our partitioning, vertices were weighted commensurate to the weight of the corresponding arc to the sink[1] and partitioned using a multilevel Kernighan-Lin method [16] [2]. Results of the mapping are reported in Section 4.

3.4 Error Detection in Distributed Max Flow

The power network is susceptible to errors that can include hardware malfunction or failure, software malfunction or corruption, malicious attacks, and unknown or unseen failures. Fault tolerance requires detecting the error, reconfiguring around it, and recovering. The focus of this paper is to construct a fail-stop system for errors that would lead to an incorrect result. Future work can address reconfiguration and recovery.

This paper constructs a fail-stop system through executable assertion checking. This is in contrast to various other forms of error detection such as masking redundancy through hardware or software [18] that require extensive hardware replication or software diversity. Assertion checking is advantageous in that it can not only detect errors in data corrupted by faulty hardware, but it can also detect errors in data received from external inputs. Our error detection is implemented by checking constraints, or assertions, on the state of the system.

When using constraint checking, there is a greater dependency on knowing the correct, or expected, behavior of the system. The constraints that must be maintained are given as Con-

[1]The assumption is that higher arc weight may imply more flow messages to that vertex.

[2]Chaco, a graph partitioning tool distributed by Sandia National Laboratories [17], was used to obtain mappings of vertices to processors. It is beyond the scope of this paper to explore the optimal mapping.

straints C1, C2, and C3. These constraints apply to both the max flow algorithm and the power system.

Constraint C1 *Flow Balance* $\forall v \in V - \{s\}\ \forall z, y \in V \sum f(w, v) = e(v) + \sum f(v, z)$

Constraint C2 *Flow Feasibility* $\forall v, w \in V - \{s\}\ f(v, w) \leq c(v, w)$

Constraint C3 *Reverse Flow* $\forall v, w \in V - \{s\}\ f(v, w) = -f(w, v)$

Theorem 3.6 $\forall v, z \in V - \{s, t\}$ *$\exists p_i$ such that $v \in V_{p_i}$ and $\exists p_j$ such that $z \in V_{p_j}$, if $\neg \exists m \in q_i \cup q_j$ where m is a* PFm *or* AFm *such that $intended\_vertex(m) = z$, and $sender(m) = v$, or* RFm *such that $intended\_vertex(m) = v$, and $sender(m) = z$, Constraints C1 and C3 are invariant over the algorithms execution.*

Proof: At the start of the algorithm, $\forall v \in V, e(v) = 0$ and $\forall v, w \in V, f(v, w) = 0$ and $c(v, w) \geq 0$, so the constraints are satisfied. In order to invalidate one of the constraints, a change must be made to f or e. f and e are modified in steps 2 and 6 of VertexProcessing as well as step 2a of Pulse. Consider step 2 of VertexProcessing. During that step, $e(v)$ becomes $e(v) + flow(Cm)$, and $f(v, w)$ becomes $f(v, w) - flow(Cm)$. Both of those expressions are on the right hand side of C1, thus the addition and subtraction of flow results in a net zero (0) change, and Constraint C1 is maintained. In step 6 of VertexProcessing, $e(v)$ and $f(v, w)$ are changed in the same manner as step 2; therefore, step 6 also maintains Constraint C1. In step 2a of Pulse, a set amount flow is subtracted from $e(v)$ and added to $f(v, w)$; therefore, step 2a of Pulse maintains Constraint C1 as well.

In step 2a of Pulse, a message $m = PFm_n$ from v to w is added to q_i and $f(v, w)$ becomes $f(v, w) + flow(m)$. At the end of the step $f(v, w) \neq f(w, v)$, but $m \in q_i$, which still satisfies the theorem. Thus, step 2a of Pulse does not invalidate the theorem. The same m is $\in q_i \cup q_j$ through processing of $VertexProcessing(m)$ for w. After finishing $VertexProcessing(m)$ for w, $m \notin q_i \cup q_j \cup \{Cm_i\} \cup \{Cm_j\}$. Steps 1, 4, 5, and 7 of VertexProcessing do not affect the theorem. In step 2, $f(w, v)$ becomes $f(w, v) - flow(m)$. Before step 2a of Pulse for v and step 2 of VertexProcessing for w, $f(v, w) = -f(w, v)$, and after both steps, $f(v, w) + flow(m) = -\left(f(w, v) - flow(m)\right)$. Thus, Constraint C3 and the theorem are satisfied for step 2 of VertexProcessing. In step 3, message $m_r = RFm_r$ is created satisfying the theorem since $m_r \in q_i \cup q_j$ with $intended\_vertex(m_r) = v$, and $sender(m_r) = z$. The same m_r is $\in q_i \cup q_j$ through processing of $VertexProcessing(m_r)$ for

v. In step 6 of VertexProcessing, m_r is processed and $f(v,w)$ becomes $f(v,w) - flow(m_r)$. Since $flow(m_r) = flow(m)$, the value of $f(v,w)$ is returned to the value prior to executing step 2a of Pulse, which means that Constraint C3 is satisfied. □

Theorem 3.7 *Constraint C2 is invariant over the algorithm's execution.*

Proof: At the start of the algorithm, $\forall v \in V, e(v) = 0$ and $\forall v, w \in V, f(v,w) = 0$ and $c(v,w) \geq 0$, so the constraint is satisfied. In order to invalidate the constraint, a change must be made to f or c. c is never modified in the algorithm. f is modified in steps 2 and 6 of VertexProcessing as well as step 2a of Pulse. In step 2a of Pulse, a message, m, is created, and $flow(m) = min(e(v), r_f(v,w))$. Let $r_f(v,w) = c(v,w) - f(v,w)$.

$$\begin{aligned} f(v,w) &= f(v,w) + flow(m) \\ f(v,w) &= f(v,w) + min(e(v), r_f(v,w)) \\ f(v,w) &= f(v,w) + min(e(v), c(v,w) - flow(v,w)) \\ f(v,w) &= min(e(v) + f(v,w), c(v,w)) \end{aligned}$$

Thus, in step 2a of Pulse, $f(v,w)$ becomes no more than $c(v,w)$, and $flow(m) \leq c(v,w)$. In steps 2 of VertexProcessing, $f(w,v)$ becomes $f(w,v) - flow(m)$. Since $f(w,v) \leq c(w,v)$ initially, step 2 does not violate Constraint C2. In step 6 of VertexProcessing, $f(v,w)$ becomes $f(v,w) - flow(m)$. Since $f(v,w) \leq c(v,w)$ initially, step 2 does not violate Constraint C2. □

It is not enough to check Constraints C1-C3 in a single process; errors that affect the computation could also affect the constraint check. Thus, it is desirable to check the constraints in each processor such that non-faulty processors can check potentially faulty processors. This requires distributing the state of the system across processors that preserves the correctness of the constraint evaluation. The key to this approach is to define the constraints as invariants over all interleavings of processes in the algorithm. The astute reader will note that correctness of this approach is inspired by the formal proof system of [19]. In [20], we define a run-time system for checking program invariants based on collecting and evaluating the distributed state of the system in a partial order induced by Lamport clocks [21]. This system is implemented as CCSP [22, 23] (C in CSP) system for transmitting state variables and is used here.

Theorem 3.8 *If Constraints C1, C2, and C3 are proven to be invariant over a program's execution, then it is invariant over the state variables collected by CCSP.*

Proof: In [20], it was proven that when a contraint is invariant over a program's execution, CCSP's distribution of state variables will maintain the invariant. □

In max flow, the distributed state variables are the capacities of the arcs, the flow of the arcs, and the excess flow at the vertices. The state variable changes are passed at the same time as other communication. This allows constraints to be checked on a consistent cut of the system. By allowing the processes to not only exchange state variables, but also the message data during a rendezvous communication, extraneous communication is avoided and run times are improved.

Theorem 3.9 *Constraints C1, C2, and C3 are accurately checked under all possible interleavings of process execution.*

Proof: Following the proof of Theorems 3.6 and 3.7, Constraints C1, C2, and C3 are invariant over all process interleavings. By Theorem 3.8, any interleaving that delays a state update must be consistent with a possible interleaving of the program. □

Not all processes need to check all disseminated state information. The following Corollary preserves the logical correctness of this approach through invariance of the constraints. The amount of disseminated state information can be reduced to minimize the impact of the state variable changes; the updates are only sent to processes a set number of hops away from the process that changed the value of the variable. As long these regions of state dissemination overlap sufficiently [24], a good error coverage can be obtained. The constraints can still be checked according to Theorem 3.9.

Corollary 3.1 *Not every process needs to receive state information about every process in order to correctly check Constraints C1, C2 and C3.*

Proof: Immediate from Theorem 3.9 and [24].□

4 Performance and Fault Injection

4.1 Performance of Distributed Max Flow with Error Detection

Five Ultra Sparc machines were used to simulate FACTS devices cooperatively determining max flow. Each machine had a 440MHz UltraSPARC-IIi processor (110MHz bus), 256 MB of memory,

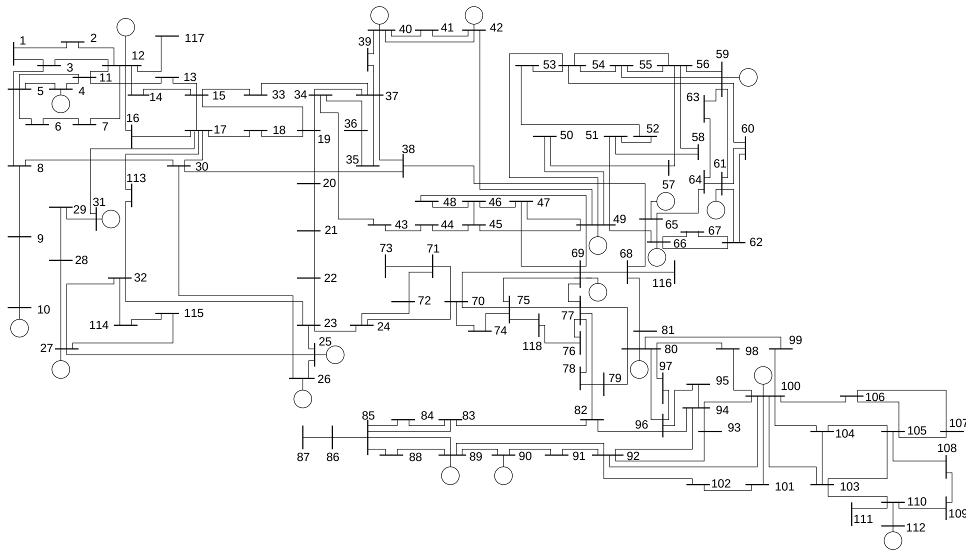
Figure 6: Model Problem: IEEE 118 bus test system, part of the IEEE power systems test suite [25].

and was connected through a full duplex 100 Mb/s Ethernet switch on a dedicated VLAN. Although machine usage was monitored before execution, these machines were available in an unregulated multi-user environment.

Each execution of the max flow algorithm was configured to use a vertex-to-processor mapping. Multiple vertex-to-processor mappings obtained from the multilevel Kernighan-Lin method were tested with multiple executions of the proposed distributed max flow algorithm over the standard IEEE 118-bus power flow test system [25] shown in Figure 6. The 118-bus system, when converted to a graph, contains 118 vertices, 179 edges, 19 sources, 99 sinks, and is a standard system for power flow experiments. The maximum results of execution times using multilevel Kernighan-Lin via bisection and quadrasection partitioning methods are shown in Figure 7. It can easily be seen from Figure 7 that vertex-to-processor configurations exist which are substantially worse than others. Figure 7 displays the Γ from the execution of the associated run time. The Γ was normalized to be in the range between the lowest and highest run time. The solid line represents the projected time to run the distributed, asynchronous Goldberg/Tarjan algorithm with one vertex per processor [13]. The time is projected by multiplying the average time to send a message times the maximum number of messages that a vertex generates. This projection assumes a perfect interleaving of messages. Running a set of 100 runs with one of the better mappings yielded statistics shown in Table 1. One outlier, which might be attributed to an unregulated multi-user testing environment, was removed.

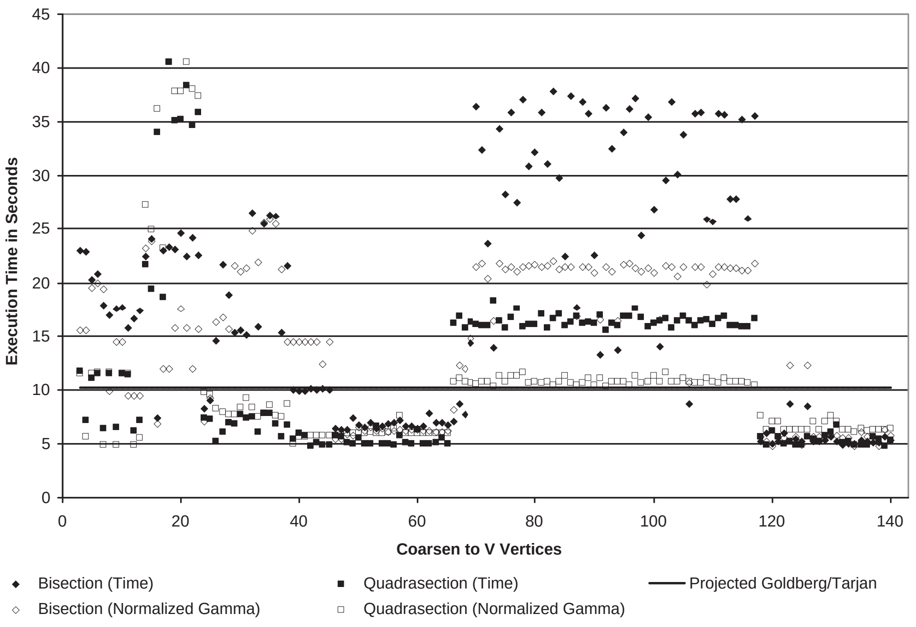

Figure 7: Run times and Γ's (gamma's) of the blocked distributed max flow algorithm under various vertex-to-processor mappings. The mappings were obtained by varying coarsening and partitioning parameters of the multilevel Kernighan-Lin method. The line is the projected time to execute Goldberg/Tarjan's distributed, asynchronous max flow algorithm based upon average message times and maximum number of messages generated by a vertex.

Table 1: Run time results for a Kernighan-Lin vertex-to-processor mapping coarsening to 126 vertices under bisection partitioning and the number of messages exchanged over all processors are in the table below. Times are for all processors to realize the algorithm terminated.

Maximum Run Time	7.37 seconds
Minimum Run Time	4.87 seconds
Average Run Time	5.72 seconds
Std Dev Run Time	0.73 seconds
Avg. External Messages	667.24
Avg. Total Messages	25547.11

The metric used for evaluating the effectiveness of our algorithm was the completion of the max flow algorithm in a time that would allow for re-configuration before events that would lead to a cascading failure. In the cascading failure that led to the 2003 blackout, the first event that started to change the pattern of flow in the power grid happened four hours before the rapid sequence of changes that resulted in widespread power loss [26]. Both the average and maximum execution times of less than 8 seconds were well within the timeline associated with the cascading failures this algorithm was designed to help prevent.

4.2 Performance of Error Injection

A variety of errors from Definition 4.1 were injected into the system to determine if our distributed max flow algorithm from Section 3.2 would detect them. These errors are not comprehensive, but are representative of errors resulting from algorithm corruption, transmission errors, and some cyber attacks in the form of message or program modification. This section evaluates testing and detection of errors based on Constraints C1 through C3. The errors were injected individually with results summarized in Table 2. *Vertex Errors* and *Edge Errors* were tested for each of the vertices and each of the edges connected to a vertex, respectively. Constraint C1 was explicitly tested by injecting *Vertex Errors*. Constraint C2 was explicitly tested by injecting *Edge Errors*. The remaining errors in Table 2 tested Constraint C2 and Constraint C3 as well as the ability to detect errors for both excessive and minimal failures within the system. Bounds on the number of messages until detecting and disseminating the errors are given in Theorems 4.1 and 4.2.

Theorem 4.1 *An error that violates one or more of Constraints C1 through C3 is detected in $O(|V||V_p||E_p|)$ messages.*

Proof: An error is detect at communication boundaries. Thus, detection is not guaranteed until a process communicates, which happens within $O(|V||V_p||E_p|)$ messages according to Theorem 3.2. $\square$

Theorem 4.2 *A detected error is disseminated in $O(|V||V_p||E_p|)$ messages.*

Proof: An error could be detected by any process, possibly quite early in the computation, but to disseminate the error, all processes must be ready to receive that an error occurred. This is not guaranteed until a process communicates, which happens within $O(|V||V_p||E_p|)$ messages according to Theorem 3.2. $\square$

Definition 4.1 Error Types

Edge Error: A given edge's flow was increased by 10%.

Vertex Error: A given vertex's calculated excess flow was doubled.

Lose All Flow Messages: All *PFm* messages would not be transmitted.

Randomly Lose Flow Messages: Each *PFm* message would not be transmitted with probability $P = 0.1\%$.

Alter All Flow Messages: All *PFm* messages were modified by one unit of flow.

Randomly Alter Flow Messages: Each *PFm* message was modified by one unit of flow with probability $P = 0.1\%$.

Invert All Accept/Reject Messages: All *AFm* messages were changed to *RFm* and all *RFm* messages were changed to *AFm*.

Randomly Invert Accept/Reject Messages: Each *AFm* or *RFm* message was reversed with probability $P = 0.1\%$.

Table 2: Error detection capabilities of the distributed max flow implementation.

Error Type	Errors Detected By			Unreported Errors	Coverage of Errors Detection	Average Time (sec)
	Program	Timeout	Connection Termination			
Edge Error (over all edges)	117 (100%)	0 (0%)	0 (0%)	0 (0%)	100%	3.437
Vertex Error (over all vertices)	115 (98.3%)	0 (0%)	2 (1.7%)	0 (0%)	98.3%	1.181
Lose All Flow Messages	0 (0%)	100 (100%)	0 (0%)	0 (0%)	100%	NA
Randomly Lose Flow Messages	0 (0%)	131 (97.0%)	0 (0%)	4 (3.0%)	97.0%	NA
Alter All Flow Messages	50 (100%)	0 (0%)	0 (0%)	0 (0%)	100%	0.454
Randomly Alter Flow Messages	50 (100%)	0 (0%)	0 (0%)	0 (0%)	100%	0.452
Invert All Accept/ Reject Messages	100 (100%)	0 (0%)	0 (0%)	0 (0%)	100%	11.803
Randomly Invert Accept/Reject	50 (100%)	0 (0%)	0 (0%)	0 (0%)	100%	6.852

Nearly all injected errors were detected, and the error was properly disseminated to the other processes. The two *Vertex Error* instances detected by connection termination ended in local infinite loops which exhausted resources before the specified timeout of 200 seconds. The *Edge Error and Vertex Error* instance not listed above actually has zero (0) flow through the vertex, so there was no error. The associated bus is at the end of a path and communicates with only one other vertex in the corresponding graph. In 150 trials of injecting the *Randomly Lose Flow Messages* error, 15 runs contained no errors, 131 runs detected the injected errors, and 4 runs failed to detect the injected errors. Three of the four undetected errors randomly lost only one message. The results of the undetected errors violated Constraint C3 within the corresponding graph. For Constraint C3 to be violated, the flows must not match and the number of *PFm*'s sent from vertex v to vertex w must agree with the number of *AFm*'s and *RFm*'s sent from w to v. The fact that the flows did not match was known, but the counts were not the same since the sent count was larger due to the lost *PFm*. Adding a flag to the constraint signifying the end of the algorithm would allow for Constraint C3 to be checked at least at the end of the program.

5 Conclusion

This paper presented a distributed max flow algorithm with fewer processors than vertices in a graph. Our algorithm was augmented with an error detection fail-stop through constraint checking to control the active power flow through a simulated power system using FACTS devices. Each FACTS device processed multiple vertices in the graph, utilizing a multilevel Kernighan-Lin heuristic for mapping vertices to processors. The IEEE 118 bus system was used as a test bed to show the effectiveness of the graph vertex to processor allocation and the error detection system. It was shown that with an appropriate mapping of vertices to processors, max flow completed quickly enough to prevent cascading failures. Most injected errors were detected with the given constraints, and with additional constraints on local resource usage and variable state upon determining that the algorithm had finished, test executions could detect all errors.

6 Future Work

Work continues in reducing the time required to run the fault-tolerant algorithm as well as improving error detection capabilities. In particular, the findings in [27] have not been implemented, and may prove beneficial. The fail-stop nature is only a starting point; reconfiguration and recovery techniques for the system need to be developed. It has been observed that excess flow is passed back and forth between a set of vertices until the flow is returned to the source. The gap relabel heuristic presented in [27] could reduce the number of passes between vertices and possibly messages between processors. A few errors were not detected due to local infinite loops. Those errors are not detected due in part to the mapping of vertices to processors. Alternative mappings that make it impossible to enter the infinite loop will be investigated.

A real-time simulation environment is being constructed which will more thoroughly test hardware and software interactions and better predict what may happen in real power system scenarios. This simulation will open up new methods of failure that can't currently be tested, and open new methods of attacking the system to better test our fault tolerance.

References

[1] C. Berge, *Graphs and Hypergraphs.* North-Holland Publishing Company, 1973.

[2] N. Li, Y. Xu, and H. Chen, "FACTS-based power flow control in interconnected power systems," *IEEE Transactions on Power Systems*, vol. 15, no. 1, February 2000.

[3] A. Armbruster, B. McMillin, and M. L. Crow, "Controlling power flow using FACTS devices and the max flow algorithm," in *Proceedings of the International Conference on Power Systems and Control*, December 2002, Abuja, Nigeria.

[4] D. E. Bakken, *Encyclopedia of Distributed Computing.* Kluwer Academic Press, 2001, ch. Middleware.

[5] D. Fulkerson and G. Dantzig, "Calculations of maximum flow in networks," *Naval Logic Quart.*, vol. 2, pp. 277–283, 955.

[6] L. Ford and D. Fulkerson, "Max flow through a network," *Can. J. Mathematics*, pp. 399–404, 1956.

[7] E. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.*, 1970.

[8] A. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Dokl.*, pp. 434–437, 1974.

[9] R. Tarjan, "A simple version of Karzanov's blocking flow algorithm," *Operations Research Letters*, vol. 2, pp. 265–268, 1984.

[10] J. Edmonds and R. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, pp. 248–264, 1972.

[11] H. Gabow, "Scaling algorithms for network problems," *Journal of Computer Systems Science*, vol. 31, pp. 148–168, 1985.

[12] A. Goldberg, "Efficient graph algorithms for sequential and parallel computers," Ph.D. dissertation, Massachusetts Institute of Technology, 1987.

[13] A. Goldberg and R. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM*, vol. 35, pp. 921–940, 1988.

[14] T.-Y. Cheung, "Graph traversal techniques and the maximum flow problem in distributed computation," *IEEE Transactions on Software Engineering*, pp. 504–512, 1983.

[15] B. Awerbuch, "Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization," *Networks*, vol. 15, no. 4, pp. 425–437, 1985.

[16] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1995, p. 28.

[17] ——, "The Chaco user's guide: Version 2.0," Sandia National Laboratories, Tech. Rep. 2692, 1994.

[18] R. Chown and T. Johnson, *Distributed Operating Systems & Algorithms*. Addison Wesley, 1998.

[19] G. M. Levin and D. Gries, "A proof technique for communicating sequential processes," *Acta Informatica*, vol. 15, pp. 281–302, 1981.

[20] H. Lutfiyya, M. Schollmeyer, and B. McMillin, *Fault-Tolerant Distributed Sort Generated from a Verification Proof Outline*. Springer-Verlag, 1992, pp. 71–96.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, pp. 558–565, 1978.

[22] B. McMillin, H. Lutfiyya, E. Arrowsmith, and C. Serban, "CCSP - a formal system for distributed program debugging," University of Missouri – Rolla, Tech. Rep. 30, 1994.

[23] B. McMillin and E. Arrowsmith, "CCSP - a formal system for distributed program debugging," *Programming and Computer Software*, vol. 21, no. 1, pp. 45–50, 1995.

[24] M. Schollmeyer and B. McMillin, "A general method for maximizing the error-detecting ability of distributed algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 2, pp. 164–172, February 1997.

[25] R. Christie, "Power systems test case archive: 118 bus power flow test case," May 1993.

[26] U. S. DOE, "Initial blackout timeline of the August 14, 2003 outage."

[27] B. V. Cherkassky and A. V. Goldberg, "On implementing the push-relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, September 1997.