Technical Report No. 2001-444

The Maximum Flow Problem: A Real-Time Approach*

Naya Nagy and Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada
Email: {nagy,akl}@cs.queensu.ca

Abstract

The dynamic version of the maximum flow problem allows the graph underlying the flow network to change over time. The graph receives corrections to its structure or capacities and consequently the value of the maximum flow is modified. These corrections arrive in real time. In this paper, parallel and sequential solutions to the real-time maximum flow problem are developed on the Reconfigurable Multiple Bus Machine (RMBM) model and on the Random Access Machine (RAM) model, respectively. The parallel solution successfully meets the deadlines imposed in real time, while the sequential one fails to do so.

The two solutions are then applied to a real-time process scheduler, an extension of Stone's static two-processor allocation problem. The scheduler allows processes to be created and destroyed, the amount of communication between two processes to change with time, and so on. The parallel algorithm is always able to compute the optimal schedule, while the solution obtained sequentially is only an approximation. The improvement provided by the parallel approach over the sequential one is superlinear in the number of processors used by the parallel model.

Key words and phrases: maximum flow, parallelism, real-time computation, module allocation.

1 Introduction

Due to its many applications, the maximum flow problem has been widely studied for the last forty years. This paper proposes a real-time version of the problem. In it, the flow network on which a maximum flow is to be computed receives corrections to its structure *during* the computation. These corrections affect the maximum flow. Because the problem is to be solved in real time, a new maximum flow has to be computed before a given *deadline*. Sequential and parallel approaches to solving the real-time maximum flow problem are proposed and compared. In most situations where parallel computers are employed, their primary purpose is to perform calculations *faster* than their sequential counterparts. Here it is shown that, in addition, the presence of deadlines allows a parallel computer to obtain a *better flow* as well, meaning that the parallel solution is *closer to optimal* than any solution arrived at sequentially.

The real-time paradigm can be used to express dynamic variants of static computations. One example is the module allocation problem [27] which relies on advance knowledge about a system of processes. Such a system may be modeled more realistically by allowing its characteristics to change with time in an unpredictable way. A real-time computational setting is therefore a more suitable environment for treating

*This research was supported by the Natural Sciences and Engineering Research Council of Canada.

the dynamic version of the module allocation problem. Again, the solution computed in parallel is of higher quality than possible sequentially. In this case, the improvement is shown to be *superlinear* in the number of processors used to obtain the parallel solution.

The remainder of this paper is organized as follows. Section 2 introduces some terminology from the theory of flows in networks. The sequential and parallel models of computation used in the design of the proposed algorithms are characterized in Section 3. A definition of the maximum flow problem in a real-time setting, with corrections arriving in real time, is provided in Section 4. Section 5 presents sequential and parallel algorithms for different correction types. The correcting algorithms are applied in Section 6 to a real-time process scheduler. Finally, Section 7 offers some suggestions for future research.

2 Background

This paper is concerned with integer flow networks. We begin by defining these networks on which a maximum flow is to be computed.

Definition 1 *A* flow network *is a quadruple $N = (G, c, s, t)$, where:*

(i) $G = (V, E)$ is a simple [1]*, directed graph, with a set V of vertices and a set E of edges.*

(ii) $c : V \times V \to \boldsymbol{Z}^+ \bigcup \{0\}$ is a function associating each edge (u, v) with a capacity *$c(u, v)$. If there is no edge from u to v then $c(u, v) = 0$.*

(iii) s (the source) and t (the sink) are two distinguished vertices in V.

The source generates a certain commodity that travels through the network and is finally consumed by the sink. The capacity expresses the maximum quantity of that commodity that is able to traverse an edge.

Definition 2 *The* flow *in the network $N = (G, c, s, t)$ is an integer-valued function $f : V \times V \to \boldsymbol{Z}$, such that $f(u, v)$ measures the amount of commodity that flows from u to v. A flow satisfies the following three properties:*

1. Capacity constraint*: for all $u, v \in V$, $f(u, v) \leq c(u, v)$.*

2. Skew symmetry*: for all $u, v \in V$, $f(u, v) = -f(v, u)$.*

3. Flow conservation*: for all $u \in V - \{s, t\} : \Sigma_{v \in V} f(u, v) = 0$.*

Flow f *saturates* edge (u, v), if $c(u, v) = f(u, v)$.

The *residual capacity,* $r(u, v) = c(u, v) - f(u, v)$, with $(u, v) \in V \times V$, forms the residual network $N_f = (G_f = (V_f, E_f), r, s, t)$, where $V_f = V$ and $(u, v) \in E_f$ if and only if $r(u, v) > 0$. A path from s to t in the residual network N_f is called an augmenting path.

For the purpose of this paper, we define the reduced flow network $FN_f = (FG_f = (FV_f, FE_f), f, s, t)$, where $FV_f = V$ and $(u, v) \in FE_f$ if and only if $f(u, v) > 0$.

The value of the flow is the flow emerging from the source, that is, $\sum_{v \in V} f(s, v)$. To solve the maximum flow problem means to find a flow of maximal value.

A *cut* (S, T) of the flow network is a partition of V into two sets S and $T = V - S$ such that $s \in S$ and $t \in T$. If the sum of the capacities of the cut edges (starting in S and ending in T) is minimum, the cut is a minimum cut.

The *Max-flow Min-cut theorem*, a fundamental result in flow network theory, states that a flow f is maximal if and only if the residual network N_f contains no augmenting paths. The value of the maximum flow is equal to the capacity of the minimum cut.

2.1 Algorithms

In what follows, we use the standard definition of a time unit to express running times. Specifically, a time unit is the length of time required by a processor either to access a memory location for reading or writing, or to perform a constant-time operation (such as $+$, $-$, AND, OR, and so on).

[1] For any two nodes $u, v \in V$, there exists only one edge of the form (u, v) starting from u and ending at v. Between u and v there exist at most two edges: (u, v) and (v, u).

Existing maximum-flow algorithms fall into two categories: the Ford-Fulkerson method [16] and the preflow-push method [21].

The Ford-Fulkerson algorithm starts with a null flow (or an arbitrary initial flow) and iteratively increases the value of the flow until it reaches the maximum. Every increase in the flow is due to an augmenting path from s to t. The contribution of an augmenting path p in increasing the value of the flow is $r_f(p) = min_{(u,v)\in p} r(u,v)$.

Based on this method, Edmonds and Karp [15] develop the shortest augmenting path algorithm whose running time is $O(|V| \times |E|^2)$, where $|V|$ is the number of vertices and $|E|$ the number of edges. Improvements have followed. Galil and Naamad [18] obtain an $O(|V| \times |E| \times \log^2 |V|)$ time algorithm, which is the first algorithm within a polylogarithmic factor from the lower bound $\Omega(|V| \times |E|)$ [20].

The preflow-push method is more time efficient. The first complete method was designed by Goldberg and Tarjan [21] and uses Karzanov's preflows [23]. Goldberg and Tarjan [21] achieve $O(|V|\,|E| \log(\frac{|V|^2}{|E|}))$ running time and King, Rao, and Tarjan [24] obtain a running time of $O(|V|\,|E|\,(\log_{\frac{|E|}{|V| \log |V|}} |V|))$.

Goldberg and Rao's algorithm [20] achieves a running time smaller than the $\Omega(|V| \times |E|)$ lower bound at the expense of limiting the capacity of the edges to a maximum value U. The execution time is $O(min(|V|^{\frac{2}{3}}, |E|^{\frac{1}{2}}) \times |E| \log(\frac{|V|^2}{|E|}) \log U)$.

For a comprehensive history of sequential maximum-flow algorithms see [19].

The conceptual difference between the two approaches is that the Ford-Fulkerson method has to have an overview of the current state of the network. Decisions are taken after the whole network is inspected. Goldberg and Tarjan's algorithm makes decisions locally. A node performs a change on the preflow, based on the knowledge it has about itself and its neighbors. A global vision of the network is not necessary. Therefore, the preflow-push method naturally lends itself to parallel implementations [22, 26, 30].

By contrast, the parallel algorithm developed in this paper is based on the Ford-Fulkerson method. This is due to the capabilities of the parallel model used, which can depict the structure of a graph in such a way that its global properties are easily extracted.

3 Models of Computation

The two models of computation to be used in this paper are now described. The sequential model is the Random Access Machine (RAM) and the parallel model is the Reconfigurable Multiple Bus Machine (RMBM). For the sake of comparison, the two models are considered to have processors of equal power. Furthermore, these processors are assumed to be the fastest possible.

3.1 The Random Access Machine

The RAM model [4] consists of one processor connected to a random access memory. The processor has some local registers to store intermediate results. The RAM runs a program by serially executing one instruction after the other. An instruction may fetch a datum from memory, perform an operation on it, and write it back to memory.

Conforming to the definition of time unit given above, a memory access or a constant-time operation lasts one time unit.

3.2 The Reconfigurable Multiple Bus Machine

The RMBM model [29] consists of m independent processors that communicate via n independent (electronic) buses (Fig. 1). Its topology (structure) can be changed during the execution of an algorithm. The buses can be fused to form larger buses, or a bus can be segmented to form two or more buses. Each processor can connect to any bus by means of a read or write port. However one processor can read or write from/to at most one bus in a given communication step. In addition, a processor can change the communication configuration by segmenting a bus or fusing two or more buses together.

Buses are used for communication among processors. When a processor reads or writes to a bus, reading can be exclusive (ER) or concurrent (CR), and similarly writing can be exclusive (EW) or concurrent (CW).

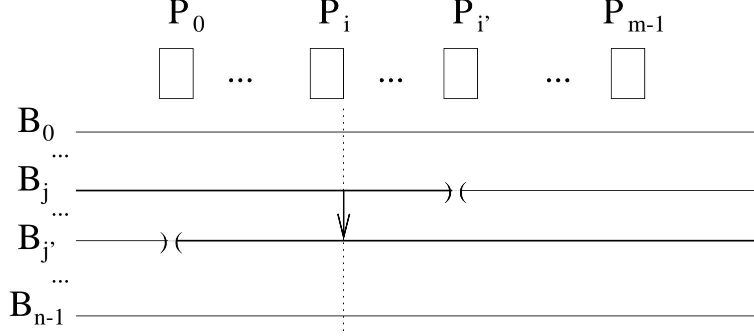

Figure 1: The RMBM

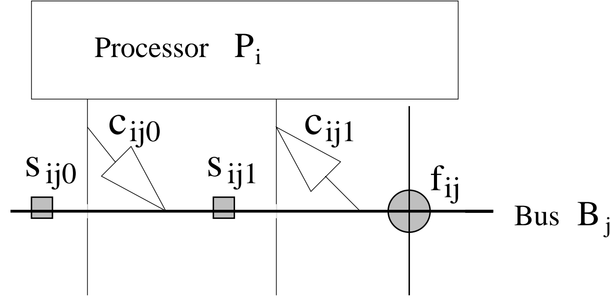

Figure 2: One processor of the RMBM

Processors are allowed to concurrently write if they are writing the same datum (common concurrent write). In our algorithms we consider a CRCW RMBM.

Thus, an RMBM with m processors $P_0, P_1, ..., P_{m-1}$ and n buses $B_0, B_1, ..., B_{n-1}$ has mn sets of switches $Q_{i,j} = \{c_{i,j,0}, c_{i,j,1}, s_{i,j,0}, s_{i,j,1}, f_{i,j}\}$, where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$. Each processor has a fuse line associated to it, which lies perpendicular to the buses. Each processor has a write port (port 0) and a read port (port 1). The switches in the set $Q_{i,j}$ (Fig. 2) lie physically on bus B_j and are controlled by processor P_i. Their meaning is:

c_{ij0} - controls writing to bus B_j.

c_{ij1} - controls reading from bus B_j.

s_{ij0},s_{ij1} - segment the bus but will not be used in the algorithm.

f_{ij} - connects the fuse line to bus B_j.

If f_{ij} and $f_{ij'}$ are both set (Fig. 1), the buses B_j and $B_{j'}$ are fused.

The RMBM can configure the buses into several "fused bus segments" by means of the segment and fuse switches.

The directional RMBM, an extension proposed by Trahan [29], and used here, fuses the bus lines directionally: if $j_1 < j_2$, data can flow either (i) only from bus j_1 to bus j_2, or (ii) only from bus j_2 to bus j_1, or (iii) in both directions.

The processors of the RMBM are considered to work synchronously. All processors receive a copy of the algorithm and execute each of its steps simultaneously. Thus, communication between two processors is possible by synchronizing the writing and reading of the bus. A communication step takes one time unit. For our purposes, the RMBM model will prove valuable as it will be able to reconfigure itself dynamically to capture the structure of the graph under consideration.

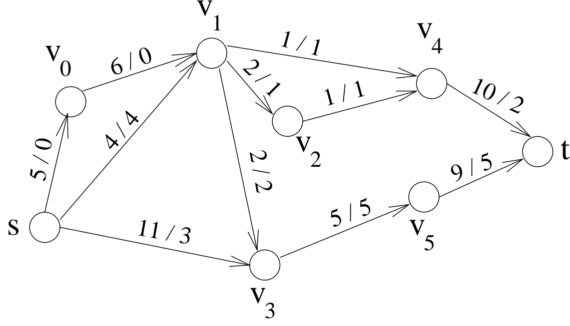

Figure 3: Flow network (maximum flow = 7)

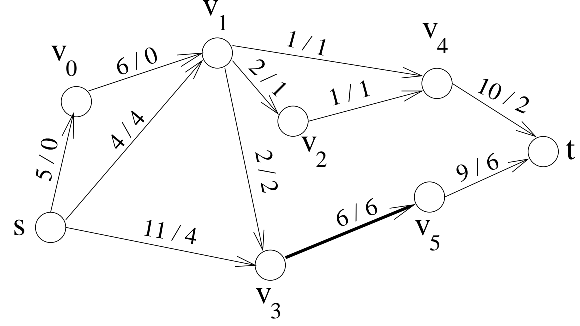

Figure 4: Unitary edge correction (maximum flow = 8)

4 The Real-Time Paradigm

In the real-time paradigm, input and output data are subject to time constraints. The input is not all available at the beginning and arrives during the computation. Output has to be produced before a deadline. Depending on the input data arrival law and the output time constraints, different sub-paradigms have been defined and studied for different problems [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 25].

The real-time maximum flow problem, as defined in this paper, is a variant of the data-accumulating paradigm [8] and the data-correcting paradigm [12]. The initial data undergo small adjustments during the computation.

The initial flow network $N = (G = (V, E), c, s, t)$, together with the already computed maximum flow f, is given at the outset. Here, $G = (V, E)$ is an arbitrary directed graph that may contain cycles, s and t are the source and sink, respectively, and c, the capacity function, takes only nonnegative integer values. Thus, the maximum flow in the network will also be an integer.

The network undergoes corrections during the computation. These corrections form a (possibly endless) stream and arrive at a constant rate. The time elapsed between two consecutive corrections is τ time units, where τ is a positive number which may depend on the size of the network, but remains fixed throughout the computation. Each correction possibly determines a variation in the network's maximum flow f. The paradigm imposes a deadline for each maximum flow to be output before it is modified. This deadline is firm, in the sense that any data produced after the deadline is useless. The problem therefore is to compute the new maximum flow before the next correction arrives, that is, in τ time units. This implies that the output is also a (possibly endless) stream, that has to be generated at the rate of one new flow every τ time units. The following four types of corrections will be studied:

Type 1. Unitary edge correction. At the beginning of each interval of τ time units, an edge capacity gets incremented or decremented by 1. This simple correction can affect the value of the maximum flow by one unit. Consider the flow network given in Fig. 3. The two numbers on each edge, separated by a '/', represent the capacity and the flow, respectively. Fig. 4 shows the result of incrementing the capacity of (v_3, v_5). Note that the flow through an arbitrary number of edges is affected; in this example, these edges are (s, v_3) and (v_5, t).

Type 2. Arbitrary edge correction. The capacity of an edge is changed by an arbitrary integer value. This is a generalization of Type 1 corrections. Edge addition and deletion also fall in this category. An edge addition means a capacity increase from 0 to some positive value, while an edge deletion means a capacity decrease to 0.

Fig. 5 shows the flow network in Fig. 3 after edge (v_3, v_4) with capacity 3 has been added. The maximum flow is increased by 3 units.

Fig. 6 shows the effect of deleting edge (s, v_3) from the same flow network in Fig. 3. The maximum flow is also decreased to 4.

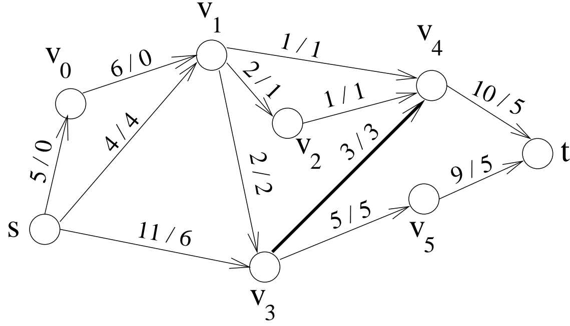

Figure 5: Edge addition (maximum flow = 10)

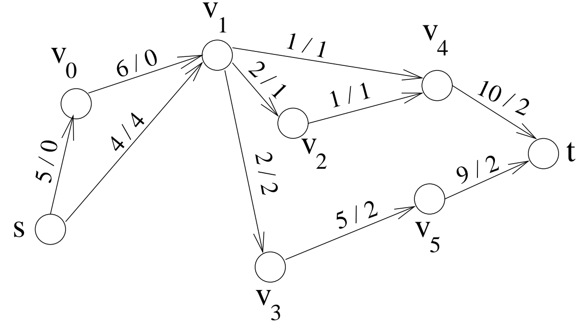

Figure 6: Edge deletion (maximum flow = 4)

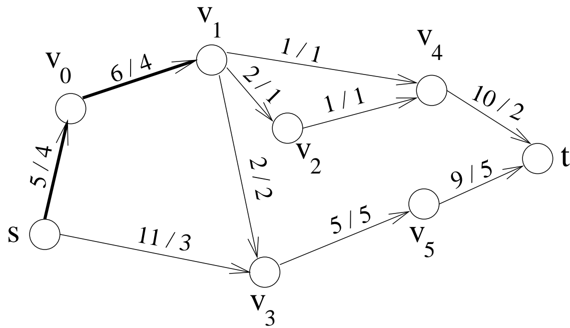

Figure 7: Edge deletion (maximum flow = 7)

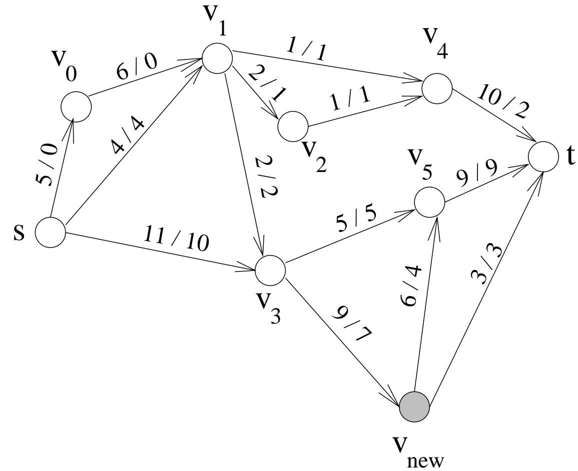

Figure 8: Vertex addition (maximum flow = 14)

Nevertheless, an edge deletion need not necessarily induce a flow decrease. In Fig. 7, edge (s, v_1) has been deleted. The flow through this edge had been 4. This value has been redirected through edges (s, v_0) and (v_0, v_1) such that the overall maximum flow remains unchanged.

Type 3. Vertex addition. A new vertex arrives together with a number of edges to connect to existing vertices. In Fig. 8, v_{new} has arrived with three edges: an incoming edge (v_3, v_{new}), and two outgoing edges (v_{new}, v_5) and (v_{new}, t). Using the additional capacity, the network supports a larger flow.

Type 4. Vertex deletion. The reverse problem to Type 3 arises when a vertex is deleted from the network together with its incoming and outgoing edges. Fig. 9 shows the flow network after vertex v_1 has been deleted.

The significance of the corrections defined above depends on the application and the specific meaning given to vertices, edges and capacities. Thus, in Section 6 we apply the real-time maximum flow problem to a process scheduler. Vertices in the graph represent processes or processors. Therefore, adding a vertex means creating a new process, and deleting a vertex means destroying a process. The edges of the graph with their capacities represent the amount of interaction between the two processes (vertices) that the edge connects. Adding an edge means that two processes start to communicate. Deleting an edge means the processes cease to interact. Any variation in the capacity of an edge shows that the amount of interaction between two processes changes.

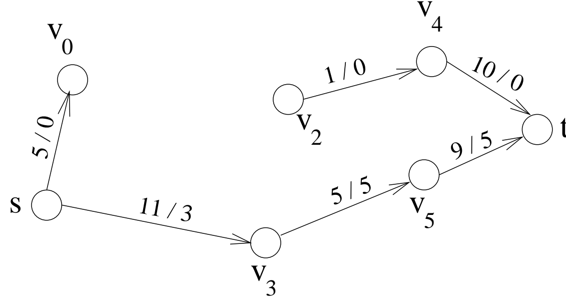

Figure 9: Vertex deletion (maximum flow = 5)

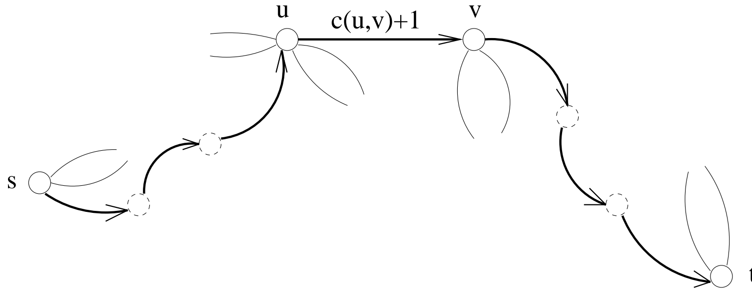

Figure 10: Residual network with unitary augmenting path

5 Correcting Algorithms

As it turns out, solving the unitary edge correction, the simplest correction possible, gives a method for addressing all other more complex correction types.

5.1 Unitary Edge Corrections

A flow network $N = (G = (V, E), c, s, t)$ with nonnegative integer capacities, $c : V \times V \rightarrow \boldsymbol{Z}^{+} \bigcup \{0\}$, is given. The maximum flow, $f : V \times V \rightarrow \boldsymbol{Z}$, is already computed and also given at the outset.

Conforming to the real-time setting, after each τ time units, an arbitrary edge (u, v) changes its capacity by one unit.

If (u, v) is nonsaturated, that is, $f(u, v) < c(u, v)$, then updating its capacity (by incrementing or decrementing it) does not affect the maximum flow.

If (u, v) is saturated, that is, $f(u, v) = c(u, v)$, then incrementing its capacity or decrementing it might cause a unitary increase/decrease of the maximum flow. Therefore, only saturated edge corrections need to be considered.

According to the Max-flow Min-cut theorem, there exist no paths from s to t in the residual network induced by the maximum flow. When the capacity $c(u, v)$ of saturated edge (u, v) is incremented, it generates a new unitary directed edge (u, v) in the residual network. This new edge can be the missing link to form a path from s to t (Fig. 10). Further, to update the maximum flow, the flow along the newly formed augmenting path has to be incremented.

If $c(u, v)$ is decremented, the algorithm attempts to redirect the now excessive unitary flow in (u, v) through other edges (Fig. 11). It searches for a path from u to v in the residual network [2]. If such a path exists, the unitary flow is successfully redirected through the network by incrementing the flow along the path and the value of the maximum flow is unaffected.

If there is no path from u to v in the residual network, the algorithm has to decrease the flow in the network by one. The algorithm searches for a path in the reduced flow graph FG_f, from s to u and from v

[2]Such a path is not allowed to use an edge of the form (t, s). Because such an edge is of no use in defining the flow capacity of a network, we consider that such an edge does not exist in the original network.

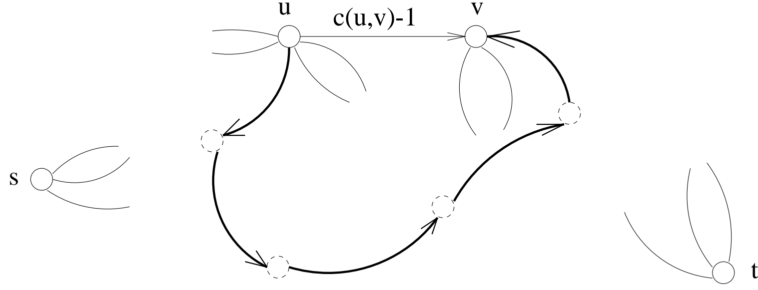

Figure 11: Residual network with rerouting path

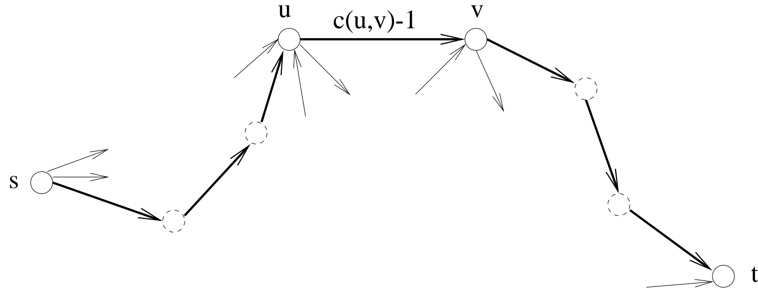

Figure 12: Reduced flow network with decrementing path

to t (Fig. 12). These paths are guaranteed to exist. Further, decrementing along the paths updates the flow.

This general procedure is independent of the model on which the algorithm is implemented. Note that the only time consuming part of the above method is finding and updating the flow on a path in either the residual network (N_f), or in the reduced flow network (FN_f).

5.1.1 Sequential solution

The implementation of the method given above is straightforward. Searching for a path in any of the networks (residual or reduced flow) needs to inspect all edges in the worst case. This takes $\Theta(|E_f|) = \Theta(|FE_f|) = \Theta(|E|)$ time in the worst case. The path contains at most all vertices. Therefore, the time required in the worst case to update the flow along the path is $\Theta(|V|)$.

This means that T^u_{seq}, the overall worst-case running time of the sequential implementation, is $\Theta(|V|+|E|)$. This is still better than the lower bound of $\Omega(|V| \times |E|)$, when computing the maximum flow from scratch.

The sequential solution is able to meet the deadline provided that $\tau \geq T^u_{seq}$. If $\tau < T^u_{seq}$, the sequential model either performs a limited number of corrections or simply approximates the maximum flow by the maximum flow given at the outset.

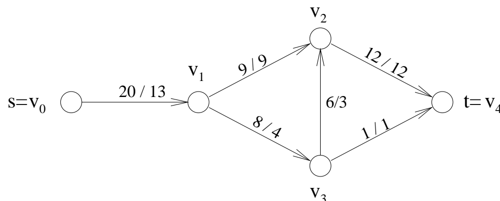

Figure 13: Flow network with capacity/flow

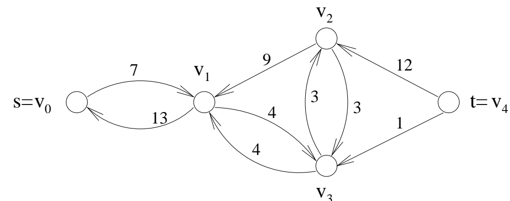

Figure 14: Residual network

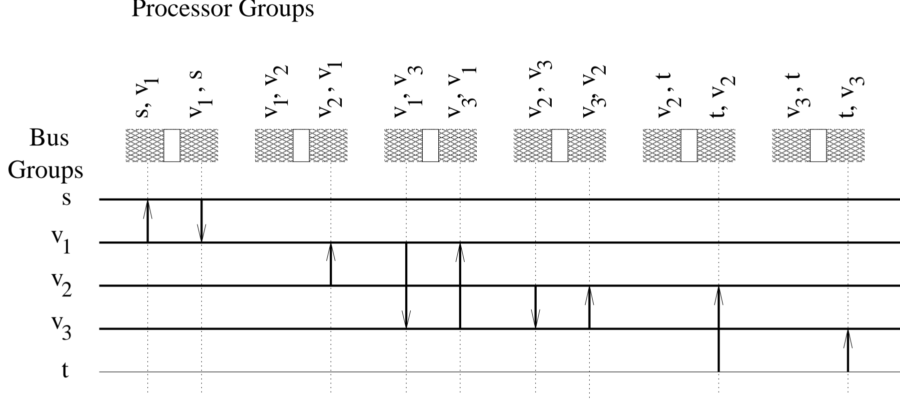

Figure 15: Configuration of the RMBM to represent the residual graph in Fig. 14

5.1.2 Parallel Solution

For the RMBM, processors represent edges and buses represent vertices. More precisely, an edge is monitored by a group of processors ($\boldsymbol{P}$) and a vertex by a group of buses ($\boldsymbol{B}$). The processors in the group associated with edge (v_i, v_j) are responsible for fusing the buses in the groups associated with the endpoints of that edge, directionally, from $\boldsymbol{B}_{v_i}$ to $\boldsymbol{B}_{v_j}$.

An example flow network is given in Fig. 13 with the induced residual network illustrated in Fig. 14 (numbers on the edges represent the residual capacities). Fig. 15 shows the configuration of the RMBM that embeds the residual network. Note that the residual network may have two edges of different directions, between two given vertices (for example, (v_1, v_3) and (v_3, v_1)). A signal written on the bus group associated with the source vertex ($\boldsymbol{B}_s$) will be seen on all buses drawn with thick lines. As evident from Fig. 15, bus group $\boldsymbol{B}_t$ is not reachable by a signal sent from $\boldsymbol{B}_s$. This is equivalent to the fact that there is no path from s to t in the residual graph (Fig. 14).

Thus, for a given graph, the RMBM can be configured to have the following property. For any two vertices v_i and v_j, there exists a group of fused directed buses in the RMBM that connect $\boldsymbol{B}_{v_i}$ to $\boldsymbol{B}_{v_j}$, if and only if there exists a directed path from v_i to v_j in the graph.

Further, in order to determine one path, the method given in [29] for list ranking, is applicable to the selection of the shortest path from among the paths connecting v_i to v_j. Each intermediate vertex v_k, including v_j, measures the lengths of the paths from v_i to itself. It identifies the path of minimum length and the predecessor vertex on this minimum path. The lengths of the paths to v_k are measured on the buses $\boldsymbol{B}_{v_k}$. The bus group $\boldsymbol{B}_{v_k}$ contains $|V|$ buses (equal to the longest simple path), denoted $B_{v_k,\,l}$, with $0 \leq l < |V|$. Any processor group $\boldsymbol{P}_{u,v}$, that represents a valid edge for the present graph instance, fuses the buses incrementally: $B_{u,\,l} \rightarrow B_{v,\,l+1}$ where $0 \leq l < |V| - 1$. To be able to fuse $|V| - 1$ lines, the processor group must contain $|V| - 1$ processors. Thus, a signal that travels along a path will be flipped to an immediately upper bus, each time it reaches a new vertex. Fig. 16 refines the upper left corner of Fig. 15 to show the contents of two bus groups ($\boldsymbol{B}_s$ and $\boldsymbol{B}_{v_1}$) and two processor groups ($\boldsymbol{P}_{s,\,v_1}$ and $\boldsymbol{P}_{v_1,\,s}$). Note that a signal written on a bus gets only incremented, never decremented. In particular, the length of a path containing a loop will be larger than the same path without the loop. Therefore, when selecting the shortest path, loops will be avoided.

To find a path from v_i to v_j, groups of the form $\boldsymbol{P}_{v_i,\,x}$ (with $(v_i, x) \in E$) send a signal on the lowest bus of vertex v_i, namely, $B_{v_i,\,0}$. This signal gets incremented whenever it reaches another vertex and may arrive to any reachable vertex v_k on different levels, according to the lengths of the different paths. That is, the signal can be read from buses $B_{v_k,l}$, $0 \leq l < |V|$, if there exists a path of length l from v_i to v_k.

An arbitrary group $\boldsymbol{P}_{u,\,v}$ will inspect both bus groups it is allowed to access ($\boldsymbol{B}_u$ and $\boldsymbol{B}_v$). Thus, all processors $P_{u,\,v,\,l}$ ($1 \leq l < |V|$) read buses $B_{u,\,l}$ and $B_{v,\,l}$. If there is any signal on $B_{v,\,l}$, then v is reachable from v_i.

Additionally, the group determines the length of the shortest path and whether (u, v) belongs to a shortest

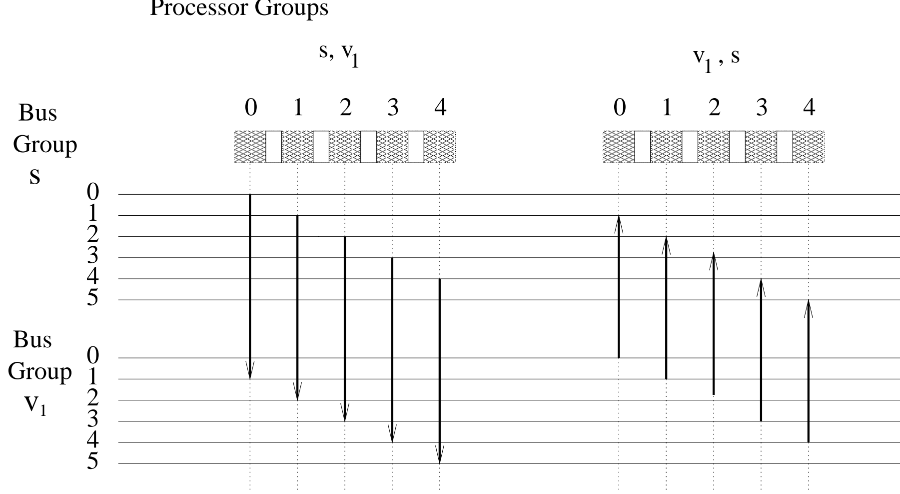

Figure 16: Detail of the RMBM's configuration in Fig. 15

path from v_i to v. Therefore, the group determines the minimum index l_1 in $\boldsymbol{B}_u$ on which the signal was present, and then the minimum l_2 in $\boldsymbol{B}_v$. This can be achieved in constant time by neighbor localization [28]. Here, l_1 represents the length of the shortest path from v_i to u, and l_2 the length of the shortest path from v_i to v. If the edge (u, v) is part of a shortest path from v_i to v, then the two lengths differ by one: $l_{v_i,v} = l_{v_i,u} + 1$. Once (u, v) is established to be part of a shortest path from v_i to v, the group's leader $P_{u,\,v,\,0}$ and other processors of the form $P_{u',\,v,\,0}$ are the subject of another neighbor localization whose purpose is to elect exactly one shortest path.

Edges (u, v) that have failed one of the above tests are withdrawn from the graph. This means that group $\boldsymbol{P}_{u,\,v}$ stops fusing any buses from $\boldsymbol{B}_u$ to $\boldsymbol{B}_v$.

Following the steps above, there exists exactly one path that connects v_i to any reachable v_k. In particular, there exists only one path from v_i to v_j. To mark all vertices/edges on the path, v_j sends a signal backwards towards v_i. Therefore all processors simply invert their fusing lines: $P_{u,\,v,\,l}$ directionally fuses $B_{v,l+1}$ with $B_{u,l}$. All groups $\boldsymbol{P}_{u,v}$ that are still active and receive a signal on both $\boldsymbol{B}_u$ and $\boldsymbol{B}_v$ are edges on the shortest path from v_i to v_j and have to update their flow according to the required correction.

The steps performed by the RMBM for the unitary correction are summarized in what follows:

Initial State:
  $P_{v_i,v_j,0}$ contains $c(v_i, v_j)$ and $f(v_i, v_j)$.
Real-Time Correction:
  (u, v) = the edge whose capacity changes.
  cor = the value by which the capacity changes ($+1$ or -1).

if $r(u, v) = 0$ **and** $cor = +1$ **then** << capacity increases >>
  Update parameters for (u, v): capacity, flow, residual capacity;
  Embed the residual graph incrementally;
  Find a path from s to t and **if** found increment the flow;
if $r(u, v) = 0$ **and** $cor = -1$ **then** << capacity decreases >>
  Update parameters for (u, v): capacity, flow, residual capacity;
  Embed the residual graph incrementally;
  Find a path from u to v and **if** found increment the flow;
  if (path not found) **then**
    Embed the reduced flow graph incrementally;
    Find a path from s to u and reduce the flow;
    Find a path from v to t and reduce the flow.

When the algorithm searches for a path, the start vertex writes the same signal on all outgoing edges. The signals propagate to reachable vertices and can collide, if there exist two paths of equal lengths to an arbitrary vertex. Therefore, the RMBM has to allow common concurrent writes on its (fused) buses. Likewise, all edges concurrently test whether the signal is propagating through them, and in doing so they perform a concurrent read from the buses.

All steps in the RMBM's unitary correction procedure take constant time and the overall execution time T^u_{par} is therefore also a constant. It is interesting to observe that the size or the intricacy of the network's underlying graph is immaterial to the parallel algorithm's execution time. The size and complexity of the graph is mirrored only by the size of the parallel model.

The parallel algorithm is able to meet the deadline, even if τ represents a constant number of time units, provided that $\tau \geq T^u_{par}$.

The size of the RMBM is generally defined as the number of switches, the product of the number of processors and the number of buses. There exist two groups of processors for each edge (u, v) in the original flow network: $\boldsymbol{P}_{u,v}$ and $\boldsymbol{P}_{v,u}$. The number of groups is therefore $\Theta(|E|)$. Each group contains $|V| - 1 = \Theta(|V|)$ processors, which yields a total of $\Theta(|V| \times |E|)$ processors. The number of bus groups is equal to the number of vertices $|V|$ and again each group contains $|V|$ buses. The total number of buses is $|V|^2$. It follows that the size of the RMBM is $\Theta(|V|^3 \times |E|)$. An RMBM of this size is able to configure any residual or reduced flow network derived from the original flow network.

5.1.3 Comparative Analysis

There is a large range of values that τ can take, in which the parallel algorithm is able to recompute the maximum flow, while the sequential algorithm fails to do so; specifically, $T^u_{par} \leq \tau < T^u_{seq}$, where T^u_{par} is $\Theta(1)$ and T^u_{seq} is $\Theta(|V| + |E|)$. Note that the execution time of the parallel algorithm does not depend on the size of the flow network (that is, $|V|$ and $|E|$). Therefore, even if the network is large, the parallel algorithm successfully recomputes the maximum flow, even for small values for τ. The larger the network is, the larger τ has also to be defined in order for the sequential solution to succeed. Clearly, the parallel algorithm offers the guarantee of success, while its sequential counterpart is inapplicable even for networks of reasonable sizes.

Although one unitary correction has a small impact on the value of the maximum flow, let us recall that corrections arrive each τ time units for an indefinite time. Therefore, if the computation is to run for a considerable time, the value of the maximum flow will also be considerably altered compared to its initial value. For example, suppose that the initial value of the maximum flow is $\Theta(|V|)$. Further, assume that $|V|^2$ edge corrections are received. If each correction produces a unitary increase in the flow, the value of the maximum flow at the end of the computation will be $\Theta(|V|^2)$. It follows that a computation taking $\tau \times |V|^2$ time units produces an order of magnitude increase in the maximum flow. Therefore, depending on the application, even small corrections may have to be taken into account in real-time computations.

5.2 Edge Deletion and Addition

The method used to handle unitary edge corrections leads to a general approach that applies to the rest of the correction types.

The deletion/addition of an edge from/to a graph is a most natural correction to a flow network. Deleting an edge means reducing its capacity to zero. Consequently, if the edge also carried a flow, that flow becomes an excess flow to be redirected or annulled. Adding an edge to a flow network means increasing its capacity from zero to a positive constant. The increased capacity may increase the maximum flow that the network allows.

Both addition and deletion of an edge are special cases of changing the capacity of an edge by an arbitrary value. For definiteness, we assume in what follows that the maximum capacity of an edge is U, such that $U = \lceil |V|^{\alpha} \rceil$, where $0 < \alpha < 1$. Thus, $\lceil |V|^{\alpha} \rceil$ is also an upper bound on the absolute value of a correction.

Sequential Solution Consider adding or deleting or simply updating an edge with capacity $a \leq \lceil |V|^{\alpha} \rceil$. We apply Gabow's [17] scaling method, as follows: in each step, the algorithm searches for a path of a certain capacity. The starting value is $\lfloor \frac{a}{2} \rfloor$. Subsequently, the capacity of the path to be sought decreases by one

half. Thus, the next path capacity to seek is $\lfloor \frac{a}{4} \rfloor$, then $\lfloor \frac{a}{8} \rfloor$, $\lfloor \frac{a}{16} \rfloor$, and so on. For each capacity value, the algorithm has to seek as many paths as it can find or until the whole capacity a has been processed.

In the worst case, augmenting paths will all have capacity 1. Therefore, the number of unitary edge corrections that have to be applied is equal to the capacity correction a. As a consequence, T^e_{seq}, the worst-case execution time for the addition/deletion of an edge, is $\Theta(a \times (|V| + |E|)) = \Theta(|V|^\alpha \times (|V| + |E|))$. This time is still better than that required to compute the maximum flow from scratch, namely, $\Omega(|V| \times |E|)$.

In the real-time setting, if the time interval τ available to perform the edge correction is less than T^e_{seq}, then the sequential algorithm will not be able to recompute the maximum flow before the deadline.

Parallel solution As with the sequential solution, the parallel algorithm for the RMBM is a simple iterative scaling of the parallel unitary update. In the worst case, when all iterations of the program perform only unitary augmentations, the program iterates U times. Therefore, T^e_{par}, the worst-case running time of the algorithm, is $\Theta(|V|^\alpha)$. When the time interval τ is greater than or equal to T^e_{par}, the parallel algorithm is capable of recomputing the maximum flow for the general edge correction and meeting the required deadline.

5.3 Vertex Corrections

These corrections amount to either deleting an existing vertex or adding a new vertex to the graph under consideration. We make the assumption that the degree of the vertex to be added or deleted is constant, that is, the number of both incoming and outgoing edges is constant. This assumption is suitable for the application described in Section 6. We also assume as in Section 5.2 that the maximum capacity of an edge is U, where $U = \lceil |V|^\alpha \rceil$ and $0 < \alpha < 1$. The algorithm that deletes a vertex does so by deleting iteratively all its incoming and outgoing edges. Similarly, the algorithm that adds a vertex performs edge additions iteratively for all incoming and outgoing edges. Because the degree of the vertex to be deleted/added is constant, the time required to delete/add a vertex is of the same order as the time taken by an edge deletion/addition. In both cases, T^v_{seq} the worst-case sequential running time is $\Theta(|V|^\alpha \times (|V| + |E|))$, while T^v_{par}, the worst-case parallel running time is $\Theta(|V|^\alpha)$. As before, when $T^v_{par} \le \tau < T^v_{seq}$, the parallel algorithms for deletion or addition of a vertex can meet the deadlines, while their sequential counterparts are unable to do so.

We conclude this section by noting that, for the version of the real-time network flow problem studied, the use of a parallel approach represents the difference between success and failure of the computation.

6 An Application: Process Scheduling

In what follows, the real-time maximum flow solutions will be applied to a real-time (dynamic) extension of Stone's [27] (static) module allocation problem. In a distributed environment, n processes have to be scheduled to run on m processors. A *process* is a program entity defined by an executable code and private data. An objective function is defined to evaluate the communication cost and the process execution cost:

$$Cost = Communication + Execution.$$

The problem is to assign processes to processors such that the objective function (the cost) is minimized. There are no precedence constraints among processes. In Stone's static case, the completion time of processes is unspecified. In the static case, processes can be considered to run indefinitely.

The module allocation problem with an arbitrary number of processes and processors is NP-complete [27]. Stone [27] solves the allocation problem for two processors in polynomial time.

The communication cost between two processes depends on their physical location. If two processes run on the same processor, the communication cost is insignificant. But if they run on different processors, the cost is given by the amount of interaction. Communication is modeled by an undirected graph. Vertices represent processes. An edge between two processes means that there exists an interaction between the two processes. The *weight* of the edge represents the amount of interaction. Fig. 17 shows a process communication model with 7 processes: p_A, p_B, p_C, p_D, p_E, p_F, and p_G.

A process needs different resources (such as memory, input/output devices, and so on). It is often assumed in process scheduling applications that these resources differ to some extent from one processor to the other.

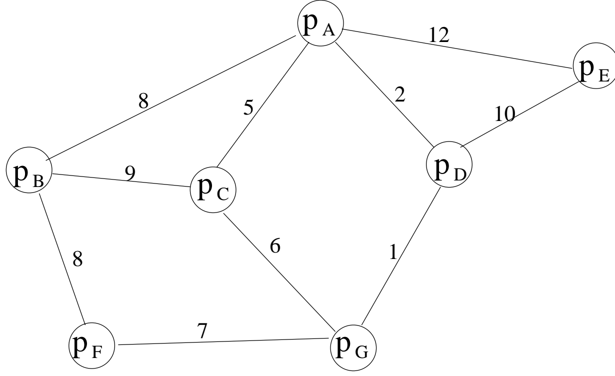

Figure 17: Process communication model

Process	Cost on P_1	Cost on P_2
p_A	5	5
p_B	5	∞
p_C	2	7
p_D	8	5
p_E	∞	3
p_F	4	4
p_G	3	6

Figure 18: Process execution cost

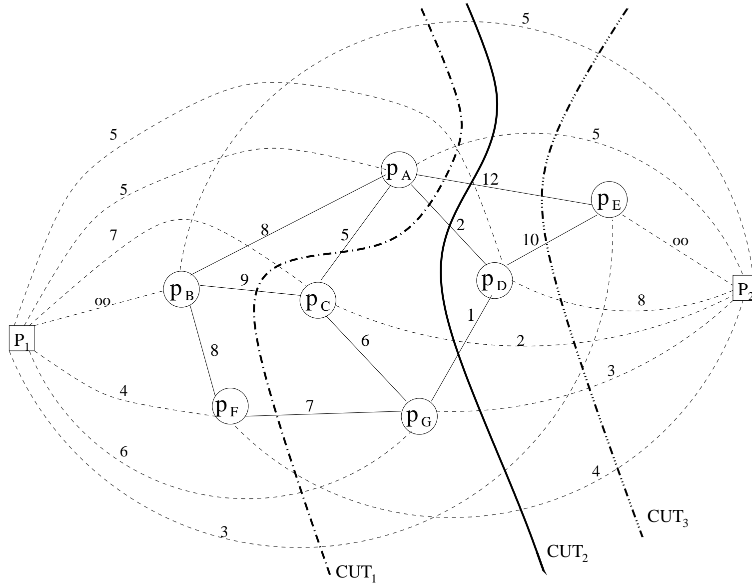

Figure 19: Flow network with cuts

As a result, a given processor may be more or less appropriate than another to run a particular process. Therefore, the cost of executing a process is a function of the processor on which it runs. The execution cost is initially given in the form of a table (Fig. 18) with entries for two processors, P_1 and P_2. If some process needs a resource which is not offered by a particular processor, that execution cost is infinity (for example, see the cost of executing p_E on P_1 in Fig. 18).

Again, the objective function to be minimized is the sum of the communication cost and the execution cost.

6.1 Stone's Solution

Stone builds a flow network based on both the communication process graph and the process execution table. Two vertices, acting as source and sink, are added to the communication process graph. They represent processors P_1 and P_2, respectively. Edges are added for each cost in the process execution table. The edge connecting processor P_1 to some process carries a weight equal to the execution cost of that process on processor P_2, and vice-versa. This is illustrated in Fig. 19 for the example in Figs. 17 and 18.

It should be noted here that the flow networks discussed thus far were directed, while the graph in this application is undirected. This undirected graph can be transformed into a directed graph by replacing any

undirected edge with a pair of directed edges of different directions and both having the capacity of the original undirected edge. Thus, whenever we discuss an algorithm on an undirected graph, we actually refer to its directed graph transformation.

In Stone's flow network, any cut defines a valid assignment of processes to processors. Moreover, the value of the cut is equal to the cost of the assignment it defines. For example in Fig. 19, three possible cuts are presented. The values of the cuts are: $CUT_1 = 74$, $CUT_2 = 42$, and $CUT_3 = 52$. The minimum cut is CUT_2. It assigns processes p_A, p_B, p_C, p_F, and p_G to processor P_1, and processes p_D and p_E to processor P_2. Therefore the two-processor module allocation problem reduces to determining the minimum cut in a flow network.

6.2 Two-Processor Allocation Problem in Real-Time

The static module allocation problem does not take into account that characteristics of the system can change in time. In a real-time setting, processes are not all created at the beginning of computation and their lifetime is not indefinite. Moreover, the amount of interaction between processes may vary in time. Therefore, a precomputed process allocation schedule may be optimal at the point of its computation but can become completely inadequate in some further stage of the computation.

Suppose that a complex computation goes on an indefinite amount of time. While it runs, processes are created and discarded in an unpredictable way. The amount of interaction between two processes may also vary in time and is bounded from above by a value depending on the number of processes.

As in the static case, the processes are to be allocated to two processors. We will consider that a newly created process does not immediately communicate with other processes, but is at first defined only by its execution cost on the two processors. Similarly, a process stops interacting with other processes before being discarded.

Formally, the processes together with the two processors form a flow network (as defined in Section 2). In this network, the cost of edges representing inapplicable executions is set arbitrarily to infinity; these edges, though present, are of no concern in the subsequent discussion. We assume that all other (communication and execution) costs are nonnegative integers smaller than or equal to $\lceil |V|^\alpha \rceil$, where $0 < \alpha < 1$. At the beginning of the computation the maximum flow and the minimum cut are given. The underlying graph undergoes corrections during the computation. Corrections occur one at a time, such that the interval separating each pair of consecutive corrections is τ time units. The following corrections will be considered:

1. The capacity of an edge changes by an amount a, where a can be positive or negative, and $|a| \leq \lceil |V|^\alpha \rceil$, which means the amount of interaction between two processes changes, or the execution cost of a process on a processor changes.

Particular cases of this situation arise when the edge does not exist and its null capacity is increased to some positive value. This means that two processes start to communicate. Similarly, deleting an edge, that is, reducing its capacity to zero, means that two processes cease to communicate.

2. A new vertex is added to the graph. The new vertex comes with exactly two edges, one connecting to the source (P_1) and the other connecting to the sink (P_2). This means that a process is created and its execution cost on the two processors is defined.

3. A vertex connected only to the source and the sink is deleted. This is equivalent to the termination of a process.

When a correction arrives, the best schedule for executing the processes on the two processors has to be determined in τ time units, before another correction arrives. The best schedule (as defined in Section 6.1) is given by the minimum cut of the corresponding flow network. Therefore, the problem is to determine the new minimum cut in τ time units.

6.3 Sequential Solution

The flow networks that describe the two-processor allocation problem have a particular form. All vertices (except s and t) are connected to both the source and the sink. Therefore, all cuts in the graph will contain a minimum of $|V| - 2$ and a maximum of $((|V|/2) \times (|V|/2)) - 1$ edges. A sequential algorithm that computes a

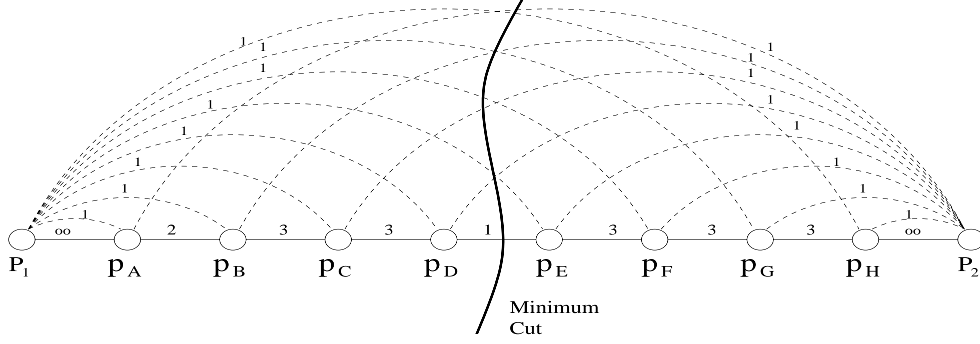

Figure 20: Initial minimum cut

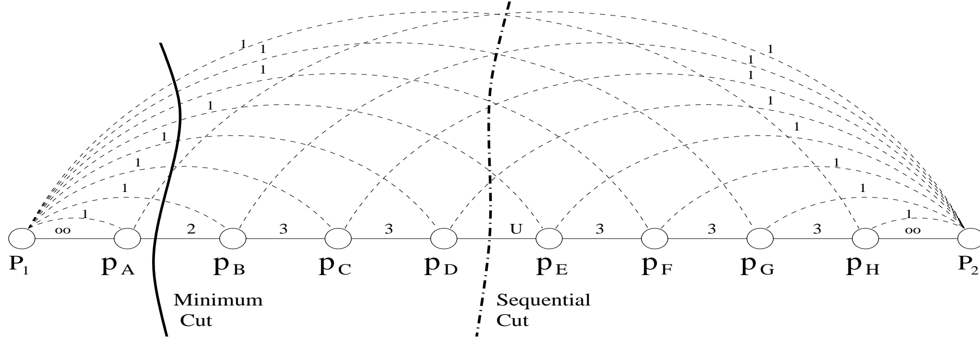

Figure 21: Minimum cut and sequential cut after one step

new minimum cut needs enough time to examine each edge of the cut at least once. This places a lower bound of $W = (|V|^2/4) - 1$ on the worst-case running time of any sequential algorithm. If the time interval between two consecutive corrections is smaller than this lower bound (in other words, if $\tau < W$), the sequential model is not capable of computing a new minimum cut before the deadline.

Nevertheless, the minimum cut for the initial graph is given at the outset. Consequently, the sequential algorithm has but one choice, and that is to use the initial minimum cut as an approximation of the minimum cut throughout the computation. This approximation may grow worse with each correction. Consider the flow network given in Fig. 20. All process execution costs are 1. The minimum cut is $MinCut = |V| - 2 + 1 = |V| - 1$. The next correction (see Fig. 21) increases the cost of edge (p_D, p_E), such that $c(p_D, p_E) = U$, where $U = \lceil |V|^\alpha \rceil$. The new minimum cut is

$$MinCut = |V|. \tag{1}$$

But the sequential model will keep to the former cut which is now

$$SeqCut(1\ step) = |V| - 1 + U. \tag{2}$$

If the subsequent corrections (see Fig. 22) are additions of edges of maximum capacity, whereby $c(p_C, p_E) = U$, $c(p_B, p_E) = U$, $c(p_D, p_F) = U$, $c(p_D, p_G) = U$, and $c(p_D, p_H) = U$, the minimum cut remains the same (that is, $MinCut = |V|$), yet the sequential model outputs:

$$SeqCut(6\ steps) = |V| - 1 + 6 \times U. \tag{3}$$

The value of the sequential cut can continuously increase for at most W steps (W being the maximum number of edges that can be added to the initial cut), such that

$$SeqCut(W\ steps) = |V| - 1 + W \times U = \Theta(|V|^{2+\alpha}). \tag{4}$$

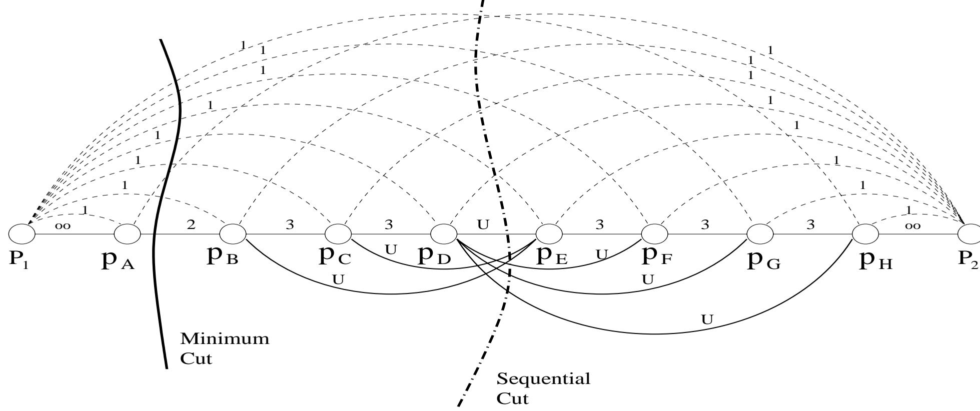

Figure 22: Minimum cut and sequential cut in worst case

To evaluate the 'goodness' of the solution, we define the accuracy of the solution as the ratio between the optimal solution and the computed solution:

$$accuracy\ of\ the\ solution = \frac{value\ of\ the\ minimum\ cut}{value\ of\ the\ computed\ cut}.$$

Therefore, if the solution is optimal (minimal), the accuracy is equal to 1. If the solution is only an approximation of the optimum, the accuracy is less than 1.

Our example presents the worst case for the sequential algorithm. The accuracy of the sequential algorithm after one step, from equations (1) and (2), is:

$$acurracy_{seq}(1\ step) = \frac{|V|}{|V| - 1 + U} = \Theta(1).$$

Thus, after one step, the sequential solution is still asymptotically optimal. Nevertheless, for W worst-case steps, from equations (1) and (4), the accuracy becomes:

$$acurracy_{seq}(W\ steps) = \frac{\Theta(|V|)}{\Theta(|V|^{2+\alpha})} = \frac{1}{\Theta(|V|^{1+\alpha})}.$$

Thus, the sequential algorithm is no longer optimal.

6.4 Parallel Solution

Parallel algorithms for edge update and insertion/deletion of a vertex were presented in Sections 5.2 and 5.3, respectively. Recall that their worst-case execution time is $\Theta(|V|^{\alpha})$.

Once the maximum flow is computed, the minimum cut can be determined using the following algorithm. The RMBM is reconfigured to reflect the residual network (to which saturated edges do not belong). A signal sent on the source's bus will reach all buses representing reachable vertices. Processors, representing edges, read both the bus of their starting vertex and end vertex. If one of the vertices is reachable from the source and the other one is not, then the edge belongs to the minimum cut. All processes (vertices) reachable from processor P_1 (the source) are scheduled on P_1 and all the other processes are scheduled on P_2. Note that the algorithm determines the minimum cut but does not compute its value. Finding the minimum cut from the maximum flow takes constant time.

The overall worst-case execution time T^c_{par} for processing a correction is $\Theta(|V|^{\alpha})$. If the time interval τ between two consecutive corrections happens to be larger than or equal to T^c_{par}, the parallel algorithm always outputs the optimal solution, and therefore its accuracy is 1. The range of values of τ for which the parallel algorithm succeeds in meeting the deadline with an optimal solution while its sequential counterpart fails to do so is $T^c_{par} \leq \tau < W$ (where T^c_{par} and W are $\Theta(|V|^{\alpha})$ and $\Theta(|V|^2)$, respectively). For example, one such value of τ would be $|V| \log |V|$.

6.5 Comparative Analysis of the Sequential and Parallel Algorithms

To compare the two approaches, sequential and parallel, we will compare the accuracies of the two solutions. The accuracy ratio measures the relative performance of the sequential and the parallel algorithm:

$$accuracy\ ratio = \frac{accuracy\ of\ the\ parallel\ solution}{accuracy\ of\ the\ sequential\ solution}.$$

The minimum cut is in the range from $\Theta(|V|)$ to $\Theta(|V|^{2+\alpha})$.

In the worst case, the sequential algorithm can add at each correction step a value of $\Theta(|V|^{\alpha})$ to the previous approximation of the minimum cut. That is, the solution can deteriorate by this amount in each step. The accuracy ratio becomes:

$$accuracy\ ratio(1\ step) = \frac{1}{\frac{|V|}{|V|-1+U}} = \Theta(1).$$

This means that the sequential model does not asymptotically perform worse than the parallel one in a single step.

However, the sequential model can continually decrease its performance. As seen, in the worst case, the accuracy of the sequential solution can steadily decrease for W steps, where W is $\Theta(|V|^2)$. Therefore the accuracy ratio after this number of steps becomes:

$$accuracy\ ratio(W\ steps) = \frac{1}{\frac{1}{\Theta(|V|^{1+\alpha})}} = \Theta(|V|^{1+\alpha}).$$

Thus, the parallel model performs asymptotically better that the sequential model after a certain number of steps. The improvement is polynomial in the number of processors.

Another measure to evaluate the performance of the algorithms is the *cumulative error*, defined as the sum of the errors made while incorporating all corrections [3]. This measure is particularly useful if a penalty is to be paid at the end of each time interval, this penalty being a function of the error in the solution computed during the present interval. For computational convenience, if an algorithm makes no error during an entire computation, its cumulative error is set to 1 (rather than 0), by definition. For the sequential algorithm of Section 6.3, in the worst case,

$$cumulative\ error_{seq} = \lceil |V|^{\alpha} \rceil + 2 \times \lceil |V|^{\alpha} \rceil + ... + W \times \lceil |V|^{\alpha} \rceil = \Theta(|V|^{4+\alpha}).$$

The parallel algorithm of Section 6.4, on the other hand, makes no error, and

$$cumulative\ error_{par} = 1.$$

The number of processors in the RMBM is $\Theta(|V|^3)$. Therefore, the improvement achieved by the parallel model, measured as the ratio of the sequential and parallel cumulative errors, is *superlinear* in the number of processors used to obtain the parallel solution.

7 Conclusion

This paper studies the maximum flow problem in a real-time setting. The flow network undergoes a number of corrections. Corrections arrive as a stream of data in real time. Each correction has to be incorporated in the solution before its deadline. Allowed corrections form a set of changes in the network: addition/deletion of an edge, addition/deletion of a vertex, increase/decrease of an edge's capacity.

Sequential and parallel algorithms are designed for each correction defined. The sequential algorithms run on a RAM, while the parallel algorithms run on a CRCW RMBM. In view of the real-time constraints, the parallel algorithms are able to meet the deadline for all possible corrections. The sequential algorithms need more time and consequently fail to meet the deadline. Choosing a parallel solution instead of a sequential one makes the difference between success and failure of the computation.

The maximum flow problem has practical applications in real time. We have defined and studied a dynamic two-processor allocation problem, derived from Stone's [27] (static) two-processor allocation problem. The problem defines a well known process scheduling method and reduces to solving a maximum flow problem. It schedules processes on two processors, minimizing an objective function.

Stone's schedule is performed with previous knowledge about the processes (specifically, the execution and communication costs). These assumptions are restrictive in a practical environment. We gave a definition of the two-processor allocation problem in a real-time setting that takes into consideration changes in the characteristics of a computation with time. Thus, communication costs among processes may vary in time, or two processes might start to communicate at some moment and cease to communicate later on. Furthermore, the actions a process takes during its lifetime affect the cost of its execution on the processors. New processes are created and destroyed during the computation. Our real-time formulation allows all of these changes to be viewed as corrections to the initial conditions.

The parallel algorithm consistently computes the optimal solution before the given deadline. The sequential algorithm is able to give only an approximation of the optimal scheduling. The accuracy ratio between the parallel and sequential solutions is $\Theta(|V|^{1+\alpha})$, in the worst case. The cumulative error ratio is $\Theta(|V|^{4+\alpha})$, indicating that the improvement due to parallelism is superlinear in the number of processors used to obtain the optimal schedule.

One feature of the algorithms used in this paper is that they are passive. The stream of corrections describes the evolution of a system (for example, several processes running on two processors) that is independent of the real-time computation. The real-time algorithm adapts to corrections that happen without its interference. Future research may focus on systems where the stream of corrections is affected by previous real-time computations.

The simplest such system is where the real-time computation determines the next correction. Possible questions in such a system are:

1. What edge deletion affects the maximum flow most?

2. What corrections are necessary to increase/decrease the maximum flow by some value a?

3. What vertex can be deleted from the graph without affecting the maximum flow?

A possible application of active correction algorithms is in biological systems metabolism. The metabolism of a set of nutrients can be modeled by a network. The initial nutrients represent a set of sources, while the final products are sinks. Note that the network comes in its more general form, allowing for multiple sources and sinks. Intermediate products represent vertices in the graph. The network is affected by different physical and chemical parameters: temperature, pressure, the presence or absence of substances (catalysts). Human interaction with these parameters is possible (in real time or in real-time modeling) to tune the network towards certain characteristics, pertaining especially to the sinks (quantity of products, relative quantity of products, and so on). Tuning the system means modifying the flow in the system through subsequent corrections.

References

[1] S. G. Akl. Nonlinearity, maximization and parallel real-time computation. In *Proceedings of the 12th Conference on Parallel and Distributed Computing and Systems*, pages 31–36. Las Vegas, Nevada, Nov. 2000.

[2] S. G. Akl. Parallel real-time computation: Sometimes quantity means quality. In *Proceedings of the International Symposium on Parallel Architectures*, pages 2–11, Dallas, Texas, Dec. 2000.

[3] S. G. Akl. Superlinear performance in real-time parallel computation. Technical Report No. 2001-443, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 2001. 17 pages.

[4] S. G. Akl and S. D. Bruda. Parallel real-time optimization: Beyond speedup. *Parallel Processing Letters*, 9:499–509, 1999.

[5] S. G. Akl and S. D. Bruda. Parallel real-time cryptography: Beyond speedup II. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1283–1289. Las Vegas, Nevada, June 2000.

[6] S. G. Akl and S. D. Bruda. Parallel real-time numerical computation: Beyond speedup III. *International Journal of Computers and their Applications, Special Issue on High Performance Computing Systems*, 7:31–38, 2000.

[7] S. G. Akl and S. D. Bruda. Improving a solution's quality through parallel processing. *The Journal of Supercomputing*, 2001. To appear.

[8] S. D. Bruda and S. G. Akl. On the data-accumulating paradigm. In *Proceedings of the Fourth International Conference on Computer Science and Informatics*, pages 150–153, Research Triangle Park, North Carolina, Oct. 1998.

[9] S. D. Bruda and S. G. Akl. The characterization of data-accumulating algorithms. *Theory of Computing Systems*, 33:85–96, 2000.

[10] S. D. Bruda and S. G. Akl. On the necessity of formal models for real-time parallel computations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1291 – 1297, Las Vegas, Nevada, June 2000.

[11] S. D. Bruda and S. G. Akl. Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems. Technical Report No. 2000-438, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Sept. 2000. 19 pages.

[12] S. D. Bruda and S. G. Akl. Real-time computation: A formal definition and its applications. Technical Report No. 2000-435, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 2000.

[13] S. D. Bruda and S. G. Akl. Towards a meaningful formal definition of real-time computations. In *Proceedings of the Fifteenth International Conference on Computers and Their Applications*, pages 274 – 279, New Orleans, Louisiana, Mar. 2000.

[14] S. D. Bruda and S. G. Akl. A case study in real-time parallel computation: Correcting algorithms. *Journal of Parallel and Distributed Computing*, 2001. To appear.

[15] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In R. K. Guy, H. Hanani, N. Sauer, and J. Schonheim, editors, *Proceedings of the Calgary International Conference on Combinatorial Structures and their Applications*, pages 93–96. Gordon and Breach, New York, London, Paris, 1970.

[16] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[17] H. N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

[18] Z. Galil and A. Naamad. An $O(EV \log^2 V)$ algorithm for the maximum flow problem. *Journal of Computer and System Sciences*, 21:203–217, 1980.

[19] A. V. Goldberg. Recent developments in maximum flow algorithms. Technical Report No. 98-045, NEC Research Institute, Inc., Apr. 1998.

[20] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 2–11, 1997.

[21] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow-problem. *Journal of the ACM*, 35(4):921–940, 1988.

[22] A. V. Goldberg and R. E. Tarjan. A parallel algorithm for finding a blocking flow in an acyclic network. *Parallel Processing Letters*, 31:265–271, 1989.

[23] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *DOKLADY: Russian Academy of Sciences Doklady. Mathematics (formerly Soviet Mathematics–Doklady)*, 15, 1974.

[24] V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[25] M. Nagy and S. G. Akl. Real-time minimum vertex cover for two-terminal series-parallel graphs. Technical Report No. 2000-441, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 2000. 18 pages.

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

[27] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, 1977.

[28] J. L. Trahan, R. Vaidyanathan, and C. P. Subbaraman. Constant time graph algorithms on the reconfigurable multiple bus machine. *Journal of Parallel and Distributed Computing*, 46(1):1–14, 1997.

[29] J. L. Trahan, R. Vaidyanathan, and R. K. Thiruchelvan. On the power of segmenting and fusing buses. *Journal of Parallel and Distributed Computing*, 34(1):82–94, 1996.

[30] U. Vishkin. A parallel blocking flow algorithm for acyclic networks. *Journal of Algorithms*, 13:489–501, 1992.