THE MAXIMUM FLOW PROBLEM A REAL TIME APPROACH

by

NAYA NAGY

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2001

Copyright © Naya Nagy, 2001

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-55921-1

Canada

Abstract

Given the large variety of its applications, the maximum flow problem has been studied ever since its definition in 1974. The problem aims to find the maximum flow that can exist between a source and a sink in a graph. The graph is weighted with the capacities of the edges.

In this work, a dynamic version of the problem is defined and studied. The graph receives corrections to its structure or capacities and consequently the value of the maximum flow is modified. Corrections arrive in real-time. The real-time paradigm imposes time constraints (deadlines) on the input and output of the computation. Parallel and sequential solutions are developed and a comparative analysis of these solutions is presented.

The generally accepted advantage of using parallel computers is to reduce the running time of computations. In real-time computation, a parallel solution can make the difference between success and failure of the computation. A parallel machine of an appropriate power is able to cope with the deadlines in due time, thus rendering the computation successful, while a sequential implementation is slower and may fail to meet the deadlines. Details are given in the analysis of the algorithms designed for the sequential random access machine (RAM) and the parallel reconfigurable multiple bus model (RMBM).

The real-time maximum flow problem is applied to the solution of a real-time process scheduling. The scheduling is an extension of Stone's static two processor allocation problem. The initial static problem assigns processes to two processors to minimize an objective function. The real-time version described in this thesis is a natural extension given the changing characteristics of the processes in real-time. The model proposed allows processes to be created and destroyed, to change the amount of communication between them, and so on. The two processor allocation problem is then solved taking in consideration these variations in real-time. Parallel and sequential algorithms are designed and analyzed. 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 polynomial in the number of processors used by the parallel computer.

Acknowledgments

Thanks to all persons who contributed to this work. Prof. Dr. Selim Akl has patiently supervised the whole process. I would like to express my respect and gratefulness for his guidance, encouragement and right advice at the right moment. In addition, thanks to all my professors, members of the staff and fellow students for their direct or indirect support in elaborating this thesis.

Thanks also go to Flori Teoh and Sabine McConnell for professional and personal friendship, to my parents for babysitting our son, and to my husband Marius Nagy for encouragement and help.

Contents

1	Introduction	1
2	Maximum Flows in Networks : A Review	5
2.1	Algorithms	12
2.1.1	The Ford-Fulkerson Method	14
2.1.2	The Preflow-Push Method	19
2.2	Parallel Algorithms for the Maximum Flow Problem	26
2.2.1	Shiloach and Vishkin's Algorithm	26
2.2.2	The Atomic Method	28
2.3	Parallel Algorithms versus Sequential Algorithms	30
2.3.1	Synergy	32
3	Real-Time Computation	34
3.1	The Input	35
3.2	The Output	35
3.3	Previous Work	36
3.3.1	Formal Definitions of Real-Time Computation	37

3.3.2	Parallel Algorithms versus Sequential Algorithms in Real-Time Computation	37
4	Flow Networks in Real-Time	39
4.1	The Real-Time Paradigm	39
4.2	The Real-Time Maximum Flow Problem	40
5	Unitary Edge Corrections	45
5.1	Sequential Solution	49
5.2	Parallel Solution	51
5.2.1	Model of Computation	51
5.2.2	Basic Techniques	54
5.2.3	The Configuration of the RMBM	54
5.2.4	The Algorithm	58
5.2.5	Analysis of the Parallel Algorithm	66
5.3	Comparative Analysis	67
6	Edge and Vertex Corrections in Real-Time	69
6.1	Edge Deletion and Addition in Real-Time	69
6.1.1	Sequential Solution	70
6.1.2	Parallel Solution	73
6.2	Vertex Deletion and Addition	77
6.3	Conclusion	80
7	An Application: Process Scheduling	81
7.1	The Module Allocation Problem	82

7.1.1	Communication Cost	82
7.1.2	Process Execution Cost	83
7.1.3	Objective Function	84
7.1.4	Stone's Solution	85
7.1.5	The Minimum Cut Problem	88
7.2	Two Processor Allocation Problem in Real-Time	89
7.2.1	Sequential Solution	91
7.2.2	Parallel Solution	94
7.2.3	Comparative Analysis of the Sequential and Parallel Algorithms	94
8	Conclusion	98
	Bibliography	102
	Vita	110

List of Tables

2.1	History of sequential algorithms	25
2.2	Parallel algorithms	30
7.1	Process execution cost	84

List of Figures

1.1	A map of roads connecting two towns	2
2.1	A flow network. The numbers on the edges represent the capacity . .	7
2.2	A flow through the network. The numbers on the edges represent the capacity/flow	7
2.3	The residual graph	9
2.4	The augmenting path $p = s, v_1, v_3, v_2, t$	10
2.5	A cut (S, T)	11
2.6	The Ford-Fulkerson method	15
2.7	Paths in a planar graph	17
2.8	The Push procedure	20
2.9	The Lift procedure	21
2.10	The general PREFLOW-PUSH procedure	22
4.1	Flow network. maximum flow = 7.	41
4.2	Unitary edge correction. maximum flow = 8.	41
4.3	Edge addition. maximum flow = 10.	41
4.4	Edge deletion. maximum flow = 4.	41
4.5	Edge deletion. maximum flow = 7.	43

4.6	Vertex addition. maximum flow = 14.	43
4.7	Vertex deletion. maximum flow = 5	43
5.1	Residual network with unitary augmenting path	46
5.2	Residual network with rerouting path	46
5.3	Reduced flow network with decrementing path	47
5.4	General procedure for unitary correction	48
5.5	The RMBM	52
5.6	One processor of the RMBM	53
5.7	Flow network with capacity/flow	55
5.8	Residual network	55
5.9	Configuration of the RMBM to represent the residual graph in fig. 5.8	55
5.10	Detail of the RMBM's configuration in fig. 5.9	58
5.11	Unitary correction on RMBM (continued on next page)	60
5.12	Unitary correction on RMBM (continued on next page)	61
5.13	Unitary correction on RMBM	62
5.14	Find the shortest path in a graph, using the RMBM (continued on next page)	64
5.15	Find the shortest path in a graph, using the RMBM	65
6.1	Sequential edge correction (to be continued)	71
6.2	Sequential edge correction	72
6.3	Edge correction: capacity increases (continued on next page)	74
6.4	Edge correction: capacity decreases (continued on next page)	75
6.5	Edge correction: capacity decreases	76
6.6	Delete Vertex	78

6.7	Add Vertex	79
7.1	Communication process model	83
7.2	Communication process graph	85
7.3	Flow network for process scheduling	86
7.4	Flow network with cuts	87
7.5	Initial minimum cut	93
7.6	Minimum cut after increasing $c(D, E)$ and sequential cut	93
7.7	Minimum cut and sequential cut in worst case	93
7.8	Determine Minimum Cut	95

Chapter 1

Introduction

A certain commodity has to be transported from a place where it is manufactured (the source) to a place where it is consumed (the sink). Predefined paths forming a network link the source to the sink and make the transportation possible. Paths have a given capacity, limiting the quantity they can carry. The commodity generated by the source travels through the network and is finally consumed by the sink. The question immediately follows: what is the maximum amount of commodity that can be pushed by the source into the network in one time unit so that the sink can receive and consequently consume the same amount also in one time unit? Given the network's characteristics, the maximum flow problem aims to determine this maximum amount and how it distributes in the network while flowing towards the sink.

For example, consider a network of roads (fig. 1.1) connecting town s (the source) to town t (the sink). A merchandise produced in s has to be transported via these roads to the storage place in t . The roads' traffic capacity is limited by the number of lanes. Each road (u, v) , connecting u to v , allows a given quantity $c(u, v)$ of

Figure 1.1: A map of roads connecting two towns

merchandise to be carried in trucks. Here, $c(u, v)$ represents the quantity that passes one spot of the road in unit time. Finding the maximum merchandise quantity that can leave s in unit time, without jamming the traffic, leads to a maximum flow problem. A similar problem can be defined in terms of fluid flowing through a network of pipes.

The maximum flow problem has been constantly studied for the last forty years. The first solution was given together with the definition of the problem by Ford and Fulkerson [29]. Numerous refinements and more efficient implementations have been proposed to the same method. A completely different approach to solving the maximum-flow problem was taken by Goldberg and Tarjan [36]. They developed the preflow-push method. The present most time efficient sequential algorithms are based on the preflow-push method. The preflow-push method was also used to develop parallel algorithms [52, 37, 60].

The real-time paradigm has recently gained interest. Different paradigms have been proposed: the data-accumulating paradigm [14], correcting algorithms [18], on-line algorithms [45] and the real-time producer-consumer paradigm [40]. The advantage of using parallel computers over sequential ones in real-time problems is reflected not only in a faster computation. Solutions given by parallel algorithms are better in terms of being more accurate, closer to optimal, more secure, better compressed, etc.

[8], depending on the problem.

This thesis aims to address the maximum flow problem in a real-time setting. The definition of the problem in real-time is application oriented. It treats the case when the initial data receive corrections during the computation and the effect of the corrections on the solution (the maximum flow) has to be computed.

Algorithms to solve the real-time maximum flow problem are proposed for both a sequential model (the random access machine – RAM) and a parallel model (the re-configurable multiple bus machine – RMBM). The two solutions are then compared in terms of optimality of the solution. The comparison between the two solutions renders the parallel version more suited to meet the requirements of real-time computation. The sequential algorithm is not able to meet the deadlines imposed by the real-time paradigm.

The algorithms are applied to solve a real-time process scheduling. The scheduling is an extension of the static process scheduling problem defined and solved by Stone [55]. Several processes have to be scheduled to run on exactly two processors such that an objective function is minimized. The number of processes and their characteristics vary in real-time. The parallel solution is always optimal, while the sequential solution can be substantially worse.

The contributions of this thesis are:

1. Definition of the maximum-flow problem in the real-time correcting paradigm.

The flow network receives corrections in real time, which have to be incorporated in the solution (the maximum flow).

2. Sequential and parallel solutions for the various corrections defined. The

parallel solution is the first parallel algorithm (to the best of our knowledge) based on the Ford-Fulkerson method.

3. The definition of a dynamic (real-time) processing scheduling method based on Stone's [55] static processing scheduling method.
4. Parallel and sequential solutions to the real time process scheduling solved by the aid of the real-time correcting maximum flow problem.

The remainder of this thesis is organized in 7 chapters. Chapter 2 describes the history of the solutions to the maximum flow problem both sequential and parallel. Chapter 3 presents the previous research in real-time computation. Chapter 4 defines the maximum flow problem in the real-time correcting paradigm. Various possible corrections occurring in real-time are described here. Chapter 5 solves the unitary correction problem, which is the simplest correction that can occur. This solution is further used to develop the more general edge and vertex corrections in chapter 6. Chapter 7 gives an application of the real-time maximum flow problem. The application is a real-time process scheduling problem, based on Stone's static two processor allocation problem. Solutions to the real-time process scheduling problem are given (both sequential and parallel). Chapter 8 comprises the conclusions of this thesis.

Chapter 2

Maximum Flows in Networks : A Review

The underlying formalism for flow problems comprises flow networks and related notions from graph and flow network theory.

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

(i) $G = (V, E)$, where $E \subseteq V \times V$, is a simple ¹, directed graph, with a set V of vertices and a set E of edges.

(ii) $c : V \times V \rightarrow \mathbb{R}^+ \cup \{0\}$ is the weight associated with the edges. $c(u, v)$ expresses the capacity of the link connecting u to v . If there is no edge from u to v then $c(u, v) = 0$.

(iii) s and t are two vertices in V .

Intuitively, the capacity reflects the maximum quantity of a commodity that is able to cross a section of the edge in unit time. From now on, we will avoid mentioning

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

that capacity measures a density rather than a quantity and will call it the maximum quantity of commodity. The same applies to other quantities to be defined later: flow, residual capacity, etc.

The graph contains two vertices with special meaning: the source s and the sink (or target) t . The source generates the commodity that will flow through the graph to be eventually consumed by the sink. Through all other vertices, the commodity can only pass, it cannot be stored, consumed, or generated.

If the network is in use, then the source generates a quantity q of commodity, which then dissipates in the network and is finally collected by the sink. Edges in the network can propagate the commodity up to their capacity. The movement of the commodity through the network is characterized by the flow.

Definition 2 *The commodity flow in the network is a real-valued function $f : V \times V \rightarrow \mathbb{R}$. $f(u, v)$ measures the amount of commodity that flows from u to v . A flow satisfies the following three properties:*

1. Capacity constraint - for all $u, v \in V$, $f(u, v) \leq c(u, v)$. This means the flow through an edge cannot exceed the capacity of that edge.
2. Skew symmetry - for all $u, v \in V$, $f(u, v) = -f(v, u)$. A real (positive) commodity flow from u to v is considered a negative flow from v to u .
3. Flow conservation - for all $u \in V - \{s, t\}$: $\sum_{v \in V} f(u, v) = 0$. This means that a vertex cannot store any amount of commodity.

Note that the flow is defined for any two vertices $u, v \in V$ with no regard as to whether the edges (u, v) or (v, u) exist.

Fig. 2.1 shows a flow network. The numbers on each edge represent the capacity of the edge. A valid flow for this network is given in fig. 2.2, where the two numbers,

Figure 2.1: A flow network. The numbers on the edges represent the capacity

Figure 2.2: A flow through the network. The numbers on the edges represent the capacity/flow

separated by "/", represent the capacity and the value of the flow. Only positive flows are shown in the figure.

By the skew symmetry property, every positive flow in a particular direction implies a negative flow in the opposite direction such that they are equal in the absolute value. Thus, we know $f(v_1, v_2) = 5$, $f(v_1, v_3) = 3$ and applying the skew symmetry it follows $f(v_2, v_1) = -5$, $f(v_3, v_1) = -3$. For vertices that are not connected, the flow is null: $f(s, v_2) = 0$.

The flow given in fig. 2.2 also obeys the flow conservation property. For example. for v_3 , we have: $\sum_{v \in V} f(v_3, v) = f(v_3, v_1) + f(v_3, v_2) + f(v_3, t) = (-3) + 2 + 1 = 0$.

Flow f *saturates* edge (u, v) , if $c(u, v) = f(u, v)$. In this case, we have a maximum flow through edge (u, v) , it reaches the capacity limit. In fig. 2.2, (v_3, t) is a saturated edge, and all the others are non-saturated. If the edge (u, v) is non-saturated by f (i.e., $f(u, v) < c(u, v)$), the residual capacity $r(u, v)$ measures the free, unused capacity of the edge.

Definition 3 [33] *For any flow f and for each pair of nodes $(u, v) \in V \times V$, the residual capacity $r(u, v)$ is*

$$r(u, v) = c(u, v) - f(u, v).$$

Using the residual capacities in a network N , a new graph, the residual graph can be formed. It contains the same nodes. Edges in the residual graph connect nodes with positive residual capacity.

Definition 4 [33] *For any flow f in network $N = (G, f, s, t)$, the residual network of N with respect to f is defined as the network $N_f = (G_f = (V_f, E_f), r, s, t)$, where*

Figure 2.3: The residual graph

$V_f = V$ and $(u, v) \in E_f$ if and only if $r(u, v) > 0$. With each edge we associate the positive weight $r(u, v)$.

Thus for the network in fig. 2.2, the residual graph is given in fig. 2.3.

The value of a flow The maximum flow problem searches for the maximum flow in a flow network. The *value of the flow* f is the sum of the flows exiting the source [24]:

$$|f| = \sum_{v \in V} f(s, v).$$

Alternatively, the value of flow f is also equal to the sum of the flows entering the sink [24]

$$|f| = \sum_{v \in V} f(v, t).$$

Theorem 1 *The Integrality Theorem [29]. If the capacity function c has integer values, the maximum flow f is also an integer value.*

In fig. 2.2, the value of the flow is $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t) = 8$.

The solution to the maximum flow problem is the flow with the maximum value.

Figure 2.4: The augmenting path $p = s, v_1, v_3, v_2, t$

The flow in fig. 2.2 is not maximal. To test whether a flow is maximal, a very useful theorem, the Max-flow Min-cut theorem, has been formulated by Ford and Fulkerson [29].

The Max-Flow Min-Cut Theorem A sequence $p = x_1, x_2, x_3, \dots, x_n$ of distinct nodes in V forms a *path* if the pairs (x_1, x_2) (x_2, x_3) ... (x_{n-1}, x_n) are edges in N . Thus starting at x_1 we can reach x_n following path p .

In particular, we are interested in paths starting in s and ending in t . If a path p from s to t exists, such that all edges on p are non-saturated by flow f , f can be increased. Such a path is called an augmenting path.

Path $p = s, v_1, v_3, v_2, t$, in fig. 2.1, contains only non-saturated edges. Equivalently, p also exists in the residual network N_f (see fig. 2.4).

Definition 5 Given a flow network $N = (G, c, s, t)$ and a flow f . an augmenting path p is a simple path from s to t in the residual network N_f .

The contribution of an augmenting path in increasing the value of f is

$$r_f(p) = \min_{(u,v) \in p} r(u, v).$$

Figure 2.5: A cut (S, T)

The Max-flow Min-cut theorem shows that to test a flow for being maximal is equivalent to searching for augmenting paths.

Another way of studying flows is with cuts.

Definition 6 [24] A *cut* (S, T) of the flow network $N = (G, c, s, t)$ is a partition of V into two sets S and $T = V - S$ such that $s \in S$ and $t \in T$.

An edge (u, v) belongs to a cut, if and only if the endpoints of the edge belong to different partitions of V : $u \in S$ and $v \in T$.

The capacities of the cut edges sum up to the *capacity* of the cut. In fig. 2.5 $c(S, T) = c(v_2, t) + c(v_1, v_3) = 12 + 8 = 20$. Note that edge (v_3, v_2) does not contribute to $c(S, T)$ because of the wrong direction.

Analogously, the flows through the cut edges sum up to the *net flow* across a cut (S, T) . In fig. 2.5 $f(S, T) = f(v_2, t) + f(v_2, v_3) + f(v_1, v_3) = 7 - 2 + 3 = 8$. Note that $f(v_2, v_3) = -f(v_3, v_2)$.

We notice that $f(S, T) \leq c(S, T)$, because the inequality holds for every edge in the sum. Also the value of the flow is given by the flow across any cut: $|f| = f(S, T)$. This can be seen by the fact that all the flow has to pass from S to T .

Theorem 2 *The Max-flow Min-cut theorem [24] If f is a flow in a flow network $N = (G, c, s, t)$ then the following conditions are equivalent:*

1. f is a maximum flow in N .
2. The residual network N_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of N .

A proof to the theorem, originally due to Ford and Fulkerson [29] can be found in [24].

2.1 Algorithms

Two main directions have been developed to solve the maximum flow problem: the Ford-Fulkerson algorithm and the preflow-push algorithm. They differ in the process of reaching the maximum flow.

The first method is due to Ford and Fulkerson [28] who also defined the maximum flow problem in 1956. It is called the Ford-Fulkerson algorithm or the labeling method. The 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 . Throughout the computation, the flow in the network is a valid flow and moreover, it is an improving approximation of the optimal, maximal flow. Various improvements and refinements have been proposed by subsequent authors: see table 2.1.

The second, the preflow-push method, uses preflows. Preflows are not valid flows, in that nodes are allowed to stock and stop the flow. The preflow becomes a valid flow only at the end of the computation, when the flow is also maximal. The algorithm

is based on Karzanov's preflows [43] and was first designed by Goldberg and Tarjan [36].

The conceptual difference between the two approaches is that Ford-Fulkerson's method relies on knowledge about the whole network. The algorithm 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 [52, 37, 60].

The complexity of sequential algorithms is given by their running times (T). To measure the running time of an algorithm, the traditionally defined *time unit* will be used [6]. A time unit is the length of time required by a processor to access a memory location for reading or writing, or perform a constant-time operation (+, −, AND, OR, etc.). The running time generally depends on the size of the problem. In this case, it is a function of the graph underlying the flow network, of the number of vertices and edges:

$$T = T(|V|, |E|).$$

The complexity of a parallel model is given by both the running time and the characteristics of the parallel model. The size of the parallel model is usually given by the number of processors. The number of processors depends again on the size of the problem, in our case on the number of vertices and edges of the underlying graph:

$$p = p(|V|, |E|).$$

The complexity of the parallel algorithm is defined by the product of the running

time and the number of processors:

$$C = C(|V|, |E|) = T(|V|, |E|) \times p(|V|, |E|).$$

The functions describing the running time of the algorithms or the number of processors used will be analyzed using the following notations:

Definition 7 [38] *Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals. Then $f(n)$ is $\Omega(g(n))$ if there exist a constant $\alpha > 0$ and a constant $n_0 > 0$, such that $0 \leq g(n) \leq \alpha f(n)$ for all $n \geq n_0$.*

Definition 8 [38] *Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals. Then $f(n)$ is $\Theta(g(n))$ if there exist the constants $\alpha_1 > 0$, $\alpha_2 > 0$ and a constant $n_0 > 0$, such that $0 \leq \alpha_1 g(n) \leq f(n) \leq \alpha_2 g(n)$ for all $n \geq n_0$.*

Definition 9 [38] *Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals. Then $f(n)$ is $O(g(n))$ if there exist a constant $\alpha > 0$ and a constant $n_0 > 0$, such that $0 \leq f(n) \leq \alpha g(n)$ for all $n \geq n_0$.*

Definition 10 [38] *Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals. Then $f(n)$ is $o(g(n))$ if, for any constant $\alpha > 0$, there exists a constant $n_0 > 0$, such that $0 \leq f(n) < \alpha g(n)$ for all $n \geq n_0$.*

2.1.1 The Ford-Fulkerson Method

The Ford-Fulkerson method is conceptually simple. It is given in fig. 2.6. The algorithm starts by initializing the flow through all edges to zero. This is the initial approximation of the final maximum flow. The algorithm then searches iteratively

Ford-Fulkerson ($N = (G = (V, E), c, s, t)$)

Notation:

$f(u, v)$ = the flow from u to v , where $u, v \in V$.

$r_f(u, v)$ = the residual capacity of the edge (u, v) .

```

1  for each edge  $(u, v) \in E$  do
2       $f(u, v) \leftarrow 0, f(v, u) \leftarrow 0$ ;
   endfor
3  while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $N_f$  do
4       $r(p) \leftarrow \min \{r(u, v) : (u, v) \text{ is in } p\}$ ;
5      for each edge  $(u, v)$  in  $p$  do
6           $f(u, v) \leftarrow f(u, v) + r(p)$ ;
7           $f(v, u) \leftarrow -f(u, v)$ ;
      endfor
   endfor

```

Figure 2.6: The Ford-Fulkerson method

for augmenting paths (line 3). For each augmenting path found, the flow in the network will be increased. The value of the flow increase is given by the maximum flow supported by the augmenting path. This is the minimum residual value along the path (line 4). The updated flow (the new approximation) is obtained in lines 5 and 6.

The running time of the Ford-Fulkerson Algorithm depends on the implementation of line 3. In the most general case, where the capacities are real numbers, the algorithm is not guaranteed to terminate. The algorithm guarantees a steady increase of the flow, but the amount by which the flow is increased at each step can be infinitesimal. Consequently, the algorithm might even not converge to the solution [29].

If the capacities are integers, a straightforward implementation runs in $O(|E| \times$

$|f^*|)$ time where f^* is the maximum flow.

The capacities may nevertheless have arbitrary values. Of special interest are commensurable capacities. Two capacities are commensurable if their ratio can be expressed as a ratio of integers [62]. For example, $c_1 = 2\sqrt{2}$ and $c_2 = 3\sqrt{2}$ are commensurable capacities, because $c_1/c_2 = 2/3$. On the other hand $c_3 = 2\sqrt{2}$ and $c_4 = \sqrt{6}$ are incommensurable, because $c_3/c_4 = 2/\sqrt{3}$. A set of capacities in which every possible two members are commensurable is called a commensurable set. If the set of capacities of the flow network is commensurable, the maximum flow problem can be solved as if the capacities were integers. For example, if the capacities are $c_1 = 2\sqrt{2}$, $c_2 = 3\sqrt{2}$, $c_3 = 5\sqrt{2}$, and $c_4 = 3\sqrt{2}$ then the maximum flow is first determined for $c'_1 = 2$, $c'_2 = 3$, $c'_3 = 5$, and $c'_4 = 3$. The final maximum flow is obtained by multiplying the flow through each edge by $\sqrt{2}$.

Edmonds and Karp [27] propose a simple refinement of line 3. From among all existing augmenting paths, pick the one of shortest length (with the least number of edges). This can be achieved by breadth first traversal of the search space. To reach the maximum flow with this method, we need to do at most $O(|V| \times |E|)$ augmentations. Each shortest augmentation path needs the traversal of virtually all edges ($O(|E|)$) to be found, such that the overall running time of the Edmonds-Karp algorithm is $O(|V| \times |E|^2)$. The method applies to the most general real valued capacities, possibly incommensurable.

Dinic [26] uses the breadth first traversal of the network, and defines layers. A node v belongs to layer i , $0 \leq i < N$, if the shortest path from s to v has length i (it contains i edges). Thus in one iteration, the algorithm finds all shortest paths of the same length. The algorithm runs in $O(|V|^2 \times |E|)$ time.

Figure 2.7: Paths in a planar graph

Scaling [30] can be applied to the labeling method, provided the capacities are integers. The idea is to take advantage of the finite number of bits that express the capacity value. If we assume that each capacity value is in the range from 0 to M , for some positive integer M , then the number of bits is $\log M$. Scaling iterates $\log M$ times, starting with the most significant bit, and successively finds flows closer to the optimum. Using scaling the algorithm's execution time is $O(|V| \times |E| \times \log M)$ [27, 30]. For incommensurable capacities, the scaling method renders approximations that converge to the maximum flow, and is therefore also applicable.

Different simplified maximum flow algorithms can be obtained for networks with specialized underlying graphs (i.e., less general graphs, such as planar graphs for example).

Algorithms for Planar Graphs A planar graph is a graph that can be drawn in the plane such that no two edges intersect. Generally, a planar graph may have several representations in the plane. We consider that the representation of the planar graph is given.

Berge [11] and Itai and Shiloach [39] develop algorithms for (s, t) -planar graphs.

An (s, t) -planar graph is a planar graph where the source s and the sink t are on the same face of the graph (see fig. 2.7). s and t belong to the same face. if they can be connected by an edge without violating the planarity. A partial order can be defined on the augmenting paths. In fig. 2.7, path p_1 is *above* p_3 , because p_1 is physically drawn above p_3 . Paths that intersect, like p_1 and p_2 , are in no order. There exists an unique uppermost path p_0 , that is above all other paths.

Berge [11] defines a data structure to conveniently find the uppermost augmenting path from s to t . Once a path is found the critical edge (a saturated edge) is deleted, the flow is updated and then the procedure restarts. The algorithm's running time is $O(|V|^2)$.

Itai and Shiloach [39] use an efficient priority queue to find the bottleneck edge and reach an $O(|E| \times \log |E|)$ running time, which is more efficient than the previous result, only under the assumption $|E| = O(|V|)$. This method can be applied to find a flow of a given value D (less than or equal to the maximum possible) in a general planar network (not necessarily an (s, t) -planar network). D is pushed through the shortest path (of length l) between s and t resulting in a pseudo-flow. The pseudo-flow satisfies the conservation property of the flows, yet pseudo-flows are allowed to exceed the capacity of edges (capacity constraint). Berge's modified algorithm is then applied for at most l times, for each over-flowing edge. The algorithm runs in $O(l \times |V| \times \log |V|)$ time.

2.1.2 The Preflow-Push Method

The Preflow-Push method is also called the Push-Relabel method. Contrary to the Ford-Fulkerson method, the preflow-push method keeps no overview of the flow network. Actions are performed locally on vertices or their adjacent edges, again based on local knowledge about the vertex and its neighbors. There are only two operations (**Push** and **Lift**) that a vertex can possibly undertake. If several vertices satisfy the conditions to apply a **Push** or **Lift**, the order in which these operations are performed is immaterial for the correctness of the result.

Goldberg and Tarjan [36], who first proposed a complete preflow-push algorithm, chose not to work with flows. Instead, their method employs preflows, introduced by Karzanov [43]. Preflow is a relaxed concept of flow. Recall that a flow through an edge was defined as the quantity of commodity that passes through that edge. Vertices were allowed to redirect the flow from incoming edges to outgoing edges, such that the flow that leaves a node is equal to the one that enters the node (conservation of flow). A preflow allows a node to stock whatever quantity of flow, such that the incoming flow may exceed the outgoing flow.

Definition 11 *The preflow of a commodity in the network $pf : V \times V \rightarrow \mathbb{R}$ is a relaxed version of flow, that satisfies the capacity constraint and the skew symmetry (properties 1 and 2 of Definition 2). The flow conservation (property 3 of Definition 2) becomes:*

3. Preflow excess - for all $u \in V - \{s, t\} : e(u) = \sum_{v \in V} pf(v, u) \geq 0$. This means that a vertex can store flow, but it cannot be in flow deficit (the outflow cannot exceed the inflow).

Here, $e(u)$ is the excess flow into u . If $e(u) > 0$ then u is an overflowing vertex.

Push (u, v) in $N_f = (G_f = (V, E_f), s, t, c)$
 Applicability: u is overflowing, $r_f(u, v) > 0$ and $h(u) = h(v) + 1$
 Action: Send $\delta = \min(e(u), r_f(u, v))$ units of flow from u to v as follows:
 $f(u, v) \leftarrow f(u, v) + \delta$; $f(v, u) \leftarrow f(v, u) - \delta$;
 $e(u) \leftarrow e(u) - \delta$; $e(v) \leftarrow e(v) + \delta$.

Figure 2.8: The Push procedure

The excess of a vertex is eliminated by pushing operations (fig. 2.8). The **Push** operation reduces/eliminates the excess of a vertex by increasing the outgoing flow along residual edges. $r_f(u, v)$ is the residual capacity of edge (u, v) . A push happens preferably towards the sink or else back to the source. Excess flow is pushed along nonsaturated edges (edges with strictly positive residual capacity). A push can therefore be *saturating*, if the residual capacity becomes zero after the push; or it can be *non-saturating*, otherwise.

To avoid pushing flow back and forth along the same edge, it is necessary to evaluate which neighbors might be nearer to the sink. Therefore a *height* label (attribute) is added to each node of the graph. Formally, the height is $h : V \rightarrow \mathbf{R}$. Throughout the computation, the heights of the source and the sink are fixed and unalterable. The source's height is equal to the number of vertices in the graph: $h(s) = |V|$ and $h(t) = 0$. Intuitively, the flow that emerges from s has to flow down to the sink. Note that the longest path possible from s to t is less than or equal to $|V|$. Thus an additional condition to push excess flow from vertex u to vertex v is that v is lower than u and the difference is exactly 1.

If a vertex is overflowing, it has to push flow to a lower vertex. If all neighbor vertices are at the same level or higher, the vertex has to be lifted above one of its

Lift (u in $N_f = (G_f = (V, E_f), s, t, c)$)
 Applicability: u is overflowing and $\forall v \in V, r_f(u, v) > 0 \Rightarrow h(u) \leq h(v)$
 Action: $h(u) \leftarrow \min\{h(v) + 1 \mid (u, v) \in E_f\}$

Figure 2.9: The Lift procedure

neighbors. Procedure **Lift** (fig. 2.9) lifts a vertex with exactly one unit above its lowest neighbor. Thus, pushes will be preferred towards neighbors of lowest heights.

The general procedure (fig. 2.10) starts with an initialization of height and preflow. All vertices except s are labeled to the lowest possible height: $h(u) = 0, \forall u \in V - \{s, t\}$. The initial preflow is built by saturating all outgoing edges from the source, as if the source would push as much flow as it can towards its neighbors. Therefore, the first overflowing vertices are the source's neighbors.

The procedure applies **Push** and **Lift** as long as there are overflowing vertices. If the height of a vertex is less than $|V|$, the vertex pushes flow towards the sink and if the height is more than $|V|$, the flow is pushed back towards the source.

The correctness of the algorithm can be proved in two steps. First the algorithm terminates, that is, no matter what sequence of pushes and lifts is employed the algorithm reaches a stage where there are no more overflowing vertices. Second, when the algorithm stops, the preflow has become a flow and moreover the maximum flow possible. For a proof see [24, 36].

Analysis of the preflow-push algorithm The number of pushes and lifts remains unspecified by the algorithm and so does the order in which they are to be applied. Therefore, to show that the algorithm terminates and evaluate its complexity, we have to find an upper bound on pushes and lifts.

```

Procedure Preflow-Push  $N = (G = (V, E), s, t, c)$ ;
<< initialization >>
  << initialize heights and excesses >>
     $h(s) \leftarrow |V|$ ;
     $\forall v \in V - \{s\}$  do begin
       $h(v) \leftarrow 0$ ;
       $e(v) \leftarrow f(s, v)$ ;
    end;
  << initialize preflow >>
     $\forall (v, w) \in (V - \{s\}) \times (V - \{s\})$  do begin
       $f(v, w) \leftarrow 0$ ;  $f(w, v) \leftarrow 0$ ;
    end
  << saturate the edges emerging from  $s$  >>
     $\forall v \in V$  do begin
       $f(s, v) \leftarrow c(s, v)$ ;
       $f(v, s) \leftarrow -c(s, v)$ ;
    end;
<< loop >>
  while  $\exists$  a basic operation that applies do
    select a basic operation and apply it;
  return ( $f$ );
end Procedure.

```

Figure 2.10: The general PREFLOW-PUSH procedure

A lift operation increases the height of a vertex. Thus, its height is ever increasing. A vertex is lifted only when it is overflowing and so reaches the highest point when there is a simple path from s to it with continuously increasing heights.

Lemma 1 [24] *During the execution of Preflow-Push on $N_f = (G_f = (V, E_f), s, t, c)$ the following bounds hold:*

- *the number of lift operations is at most $2|V| - 1$ per vertex and at most $(2|V| - 1)(|V| - 2) < 2|V|^2$ overall.*
- *the number of saturating pushes is at most $2|V||E|$.*
- *the number of nonsaturating pushes is at most $4|V|^2(|V| + |E|)$.*

For a detailed proof see [24].

Note that a push and lift implies a constant number of basic operations: additions, subtraction etc. Thus, the execution time of Preflow-Push on $N_f = (G_f = (V, E_f), s, t, c)$ is $O(|V|^2 |E|)$.

Heuristics for the Relabeling of Vertices. A vertex that is overflowing but all its neighbor vertices have higher levels, has to be lifted. The algorithm imposes no particular order on the possible operations in order to yield a correct run. Nevertheless, relabeling alone can provoke the flow to move back and forth for a couple of times before it is routed to another section of the network. This happens because the height of a vertex is only an estimate (a lower bound) of the distance of the vertex to the sink (if $h(v) < |V|$) or to the source (if $h(v) \geq |V|$). Therefore, a process called “global relabeling” was proposed by Goldberg and Tarjan [36]. Global relabeling aims to assign heights to the vertices equal to the exact distance to the sink/source. The

procedure starts from the sink and traverses the residual network breadth first, labeling the reached vertices with their distance to the sink. Then, it performs a second breadth first search from the source to label the remaining vertices (unreachable from the sink). Global Relabeling is time costly: $O(|E|)$ sequentially. Therefore it has to be used at appropriate intervals.

A parallel implementation of global relabeling can be found in Anderson and Setubal [10]. The implementation uses global relabeling waves, a method that permits concurrent execution of the global relabeling and the usual push and lift (relabel).

Derigs and Meier [25] use a “gap heuristic”. If there exists a value $l < |V|$ such that for all vertices $v \in V$, $h(v) \neq l$, then all $w \in V$ with $h(w) > l$ can no longer reach the sink. In this case $h(w)$ can be set to $|V|$, such that no attempts will further be made to push flow towards the sink. Instead excessive flow will be pushed towards the source.

In table 2.1 a brief history of sequential algorithms for the maximum flow problem is given. The table is a compiled version of the tables given in [36, 35, 34]. $|V|$ and $|E|$ have the usual meaning: number of vertices and number of edges. U is the maximum capacity an edge may have, and ϵ is a real number $0 < \epsilon < 1$. Algorithms 1-11 use methods finding augmenting paths. Algorithms 12-17 are based on Karzanov’s preflows.

No.	Author	Complexity	Ref.
1	Edmonds and Karp (1969)	$O(V E ^2)$	[27]
2	Dinic (1970)	$O(V ^2 E)$	[26]
3	Karzanov (1974)	$O(V ^3)$	[43]
4	Cherkasky (1977)	$O(V ^2 E ^{\frac{1}{2}})$	[22]
5	Mallotra, Pramodh Kumar and Maheshwari (1978)	$O(V ^3)$	[47]
6	Galil (1978)	$O(V ^{\frac{5}{3}} E ^{\frac{2}{3}})$	[31]
7	Galil and Naamad; Shiloach (1978)	$O(V E (\log V)^2)$	[32, 51]
8	Sleator and Tarjan (1983)	$O(V E \log V)$	[53, 54]
9	Shiloach and Vishkin (1982)	$O(V ^3)$	[52]
10	Gabow (1983)	$O(V E \log U)$	[30]
11	Tarjan (1984)	$O(V ^3)$	[56]
12	Goldberg (1985)	$O(V ^3)$	[36]
13	Goldberg and Tarjan (1986)	$O(V E \log(\frac{ V ^2}{ E }))$	[36]
14	Ahuja and Orlin (1987)	$O(V E + V ^2 \log U)$	[1]
15	King, Rao, and Tarjan (1994)	$O(V E (\log_{\frac{ E }{ V \log V }} V))$	[44]
16	Phillips and Westbrook (1993)	$O(V E (\log_{\frac{ E }{ V }} V + \log^{2+\epsilon} V))$	[50]
17	Cheriyān, Hagerup and Mehlhorn	$O(E V + V ^2 \log^2 V)$	[21]
18	Goldberg and Rao	$O(\min(V ^{\frac{2}{3}}, E ^{\frac{1}{2}}) \times E \log(\frac{ V ^2}{ E } \log U))$	[35]

Table 2.1: History of sequential algorithms

2.2 Parallel Algorithms for the Maximum Flow Problem

Parallel models contain several processors, as opposed to sequential models which have only one. The preflow-push method has been the preferable start for parallel algorithms. This is due to vertices performing pushes and lifts based on local information. Thus, processors assigned to vertices of the graph can work only with a limited information and need not be aware of the underlying graph's structure.

2.2.1 Shiloach and Vishkin's Algorithm

Shiloach and Vishkin [52] designed the first parallel solution to the maximum flow problem, restricted to a particular network: the *layered network*. Vishkin [60] extended the method to apply to any directed, acyclic network.

Definition 12 [52]

A flow network $N = (G, c, s, t)$ is called a *layered network* if G has the following properties:

1. Each vertex v has a layer number $l(v)$.
2. $\forall v \in V : l(s) = 0$ and $0 \leq l(v) \leq l(t)$.
3. If $u \rightarrow v \in E$ then $l(v) - l(u) = 1$.

The set $L_j = \{v : l(v) = j\}$ is the j th layer of G .

Karzanov [43] first used layered networks to design a sequential algorithm for finding the maximum flow. Shiloach and Vishkin [52] adapted the same ideas to a parallel model, the Parallel Random Access Machine (PRAM).

The PRAM consists of p processors that have access to a common memory. Processors are allowed to access the same memory location simultaneously for reading. They may also simultaneously access the memory location for writing, provided they attempt to write the same value. To measure the running time of the algorithm, elementary operations are considered to take one time unit.

Shiloach and Vishkin's algorithm proceeds in *pulses* (iterations). It starts from the source where it saturates all outgoing edges. Subsequently it works through the network towards the sink, pushing flow through subsequent edges and rerouting excess flow.

At the beginning of each pulse, vertices can be divided into two categories. Some vertices are *balanced*, for which the incoming flow is equal to the outgoing flow (the vertex obeys the flow conservation rule). The other vertices are *unbalanced*, which means the incoming flow exceeds the outgoing flow (the vertex stores flow).

Vertices that have all outgoing edges saturated can no longer influence the algorithm and are therefore called *blocked*. Vertices once blocked remain in this state until the end of the algorithm.

Definition 13 *A vertex v becomes blocked as soon as all its emanating edges are either saturated or lead only to vertices that were blocked in previous pulses. The sink t is never blocked.*

Only unbalanced vertices are active during a pulse. They can perform two possible actions. First they attempt to push the overflow through the unsaturated outgoing edges, towards the sink. The rest of the stored flow is then returned through the incoming edges towards the source. The algorithm does not define height for vertices, as the standard preflow-push algorithm does. Instead, every vertex keeps a strict

order of the incoming and separately of the outgoing edges. Excess flow is pushed into outgoing edges according to the order in which the edges are kept. Flow is returned according to a LIFO (last in - first out) rule. Therefore, incoming edges with the flow they bring in are kept on a stack.

The important characteristic of these pulses to be parallelisable is that unbalanced vertices cannot belong to two consecutive layers. Moreover, during an even (odd) pulse, all unbalanced vertices lie in odd (even) layers. This has the immediate consequence that unbalanced vertices do not interfere, if they push or return their exceeding flow in parallel. The method is correct, given the correctness of the preflow-push method.

The number of pulses needed is limited from above by $2|V|$. The running time of the algorithm is $O(|V| \times \log |V|)$, using $|V|$ processors.

The method described above fails if a vertex attempts to push flow through the same edge in two subsequent pulses, such that the second push collides with a flow return. While this is not possible in layered networks (a vertex is active every second pulse) it can happen in a general network. Vishkin [60] redesigned the algorithm's data structures, such that they can deal with interfering pushes and returns. The algorithm needs at most $3|V|$ pulses, but behaves asymptotically the same: $O(|V| \times \log |V|)$ running time with $|V|$ processors.

2.2.2 The Atomic Method

Goldberg and Tarjan [37] describe a parallel method for the maximum flow problem using *atoms* of flow. The method again uses preflows, pushed from the source towards the sink. Excess that cannot be pushed towards the sink is returned towards the

source.

The quantity pushed at some step through an edge is an atom. Moreover, an atom is traceable from the source to its present vertex location v . Thus, an atom a has a path $trace(a)$ associated with it, that keeps the history of its movements (pushes and returns). The simple path $path(a)$ from s to v contains only “useful” pushes. An atom a at v can perform one of the following actions:

- a can be pushed entirely to a neighbor vertex w , $(v, w) \in E$.
- a can be returned to u , where it came from, $(u, v) \in E$.
- a can be split into two atoms a and a' , such that the latter a is pushed to w . $(v, w) \in E$, and a' remains at v for further processing.

During initialization all source outgoing edges become saturated and an atom is created for each edge.

The intuition behind the algorithm is that each atom does a depth first search from s in an attempt to reach t . The graph being searched changes dynamically as edges become saturated and vertices become blocked.

The parallel implementation of the method, similar to Shiloach and Vishkin's algorithm, proceeds in pulses. In one pulse, all active atoms change their location with at least one step. Using an $|E|$ -processor PRAM, the algorithm runs in $O(|V| \log |V|)$ time.

The short history of parallel maximum flow algorithms is given in table 2.2.

No.	Author	Running Time	Nr. of Procs	Ref.
1	Shiloach and Vishkin (1982) (layered network)	$O(V \log V)$	$ V $	[52]
2	Goldberg and Tarjan (1989)	$O(V \log V)$	$ E $	[37]
3	Vishkin (1992)	$O(V \log V)$	$ V $	[60]

Table 2.2: Parallel algorithms

2.3 Parallel Algorithms versus Sequential Algorithms

Sequential and parallel algorithms differ by the model they use. Sequential models have only one processor, while parallel models have several processors. The main reason for parallel algorithms is to speed up computation. The execution time of algorithms is a function of the size of the input to the given problem. In the case of the maximum flow problem, the size is given by the underlying graph. Therefore, the parameters we will use to define the execution time are the number of vertices $|V|$ in the graph and the number of edges $|E|$. We will use the notation

$$T_1 = T_1(|V|, |E|).$$

for the running time of the sequential algorithm and

$$T_p = T_p(|V|, |E|).$$

for the running time of the parallel algorithm using p processors.

The speedup evaluates the gain in time offered by the parallel algorithm. It is the ratio of T_1 , the worst case running time of the fastest known sequential algorithm, to T_p , the worst case running time of the parallel algorithm employing p processors:

$$S(1, p) = \frac{T_1}{T_p}.$$

For the majority of computational problems, the best obtainable speedup is equal to the number of processors. Because of its wide application, this property is known as a theorem.

Theorem 3 [2] Folklore Speedup Theorem: *For a given computational problem, the speedup provided by a parallel algorithm using p processors, over the fastest possible sequential algorithm for a problem, is at most equal to p ; that is, $S(1, p) \leq p$.*

Between two parallel algorithms that have the same speedup, naturally the one employing fewer processors is preferred. Recall that the complexity of a sequential algorithm is given by its running time T_1 , while the complexity of the parallel algorithm is defined as the product of the execution time and the number of processors $C = T_p \times p$.

The efficiency of a parallel algorithm is the ratio of the sequential complexity over the parallel complexity:

$$E(1, p) = \frac{T_1}{T_p \times p}.$$

Again according to the Speedup theorem the efficiency has to be less than one. If the efficiency is equal to one, the parallel algorithm is optimal. According to the Speedup theorem, no "better" parallel algorithm can be developed.

Traditionally it is also accepted, that if a parallel algorithm reduces the number of processors, the running time will increase proportionally to the number of processors eliminated. In other words, the performance decreases gracefully with the number of processors. This property was formulated as a theorem by Brent.

Theorem 4 [46, 12] *If a computation C can be performed in time t with q operations and sufficiently many processors that perform arithmetic operations in unit time, then C can be performed in time $t + (q - t)/p$ with p such processors.*

2.3.1 Synergy

Although the above principles should apply in general, problems/algorithms that violate one or both of these principles exist. Such algorithms are said to exhibit synergy. Synergy happens mainly because there are some implicit assumptions underlying those principles; these assumptions might not hold in some cases. We state them [46], perhaps not exhaustively, in the following:

1. The overhead incurred by context switching is negligible. A sequential machine that simulates a parallel algorithm on p processors needs to simulate each step performed in parallel on the p processors, by at most p sequential steps. Nevertheless, in each step, the context has to be changed from one of the p processors to another. This can involve considerable overhead.
2. The data are static and do not change their values dynamically during the computation. Changing values imply that the computation has to adapt to the new situation, possibly redoing some computation. A sequential computer might not be able to catch up.
3. The computation is serializable. If a parallel algorithm is simulated by a sequential one, it is assumed that the computation can be performed sequentially.

4. There are no time limits imposed on the algorithm. Time is not mentioned in the definition of the problem, neither in connection with the incoming data, nor with the output. Time constraints may result in a poor behavior of the sequential algorithm.

In general, problems that violate some of these assumptions lead to synergetic behavior. Algorithms for such problems can behave much better in a parallel design. In this case, the efficiency is superunitary and the speedup theorem does not hold. On the other hand, an algorithm designed to run on p processors, when run on a machine with q processors, $q < p$, can exhibit a large decrease in the performance. Although, Brent's theorem permits a moderate decrease of performance (proportional with $\frac{q}{p}$), the performance can drop below this threshold, violating the principle.

Point 2, the data needs to be static - unchangeable in time, and point 4, the problem does not define time limits, eliminate the time when discussing an algorithm in traditional terms. If a problem has dynamically changing data, or there are some time constraints on the input/output data, the problem belongs to the real-time computation paradigm. We can see from here that synergy is likely to be an issue in a real-time problem.

Chapter 3

Real-Time Computation

An algorithm takes some input data, performs a computation, and produces output data. The reading in of the initial data and the delivery of the output data is the connection of the algorithm to the external world. Conventional computations are performed on static input data. The data are all available at the beginning and do not change during the computation. There are also no stringent time conditions concerning the output. Usually, results are *useful* no matter *when* they are produced, provided that they are available after a reasonable delay, depending on the application.

In general, a real time problem is one in which time appears as a parameter. A real time problem has time constraints in its definition. It is reasonable to impose time constraints on the parts where the algorithm interacts with the external world. Thus, time constraints can be defined on the input data and/or on the output data. Depending on the input data arrival law and the output time constraints, different sub-paradigms have been defined and studied for different problems [3, 4, 5, 6, 7, 8, 14, 15, 16, 17, 18, 19, 20, 49].

3.1 The Input

In the real-time paradigm input data can arrive on a timely basis. The input is not all available at the beginning of the computation. Input data continue to arrive during the computation. Some initial data are available at the outset. The *arrival-rate* can be *constant* or it can follow a *data arrival law*, which is a function $f(n, t)$, where n is the amount of data that is available beforehand and t denotes the time [13].

The simplest input arrival law is with data arriving at constant rate. The computation is logically divided into time intervals, each τ time units long. During the computation, a fixed size of input data arrive at the beginning of each time interval.

3.2 The Output

The time when the output has to be produced is also specified. Output produced after the *deadline* is either useless or it is of less practical value. Output can be seen as one single event that has to happen before a specified moment in time. In the more general case, output is a stream of data (possibly infinite) where each data item has its own time-constraint. For example, output has to be generated at a given rate, with a possible delay for the first data item.

The deadline can either be firm or soft [15]. In both cases the usefulness of the data is maximum if produced before the deadline. An output with firm deadline that is produced after the deadline is completely useless. By contrast, an output with soft deadline that is produced after its deadline has a nonzero usefulness and the function that describes it is decreasing. Formally, the usefulness function can have the following forms:

$$\text{firm:} \quad u(t) = \begin{cases} \max & t \leq \text{deadline} \\ 0 & t > \text{deadline} \end{cases}$$

and

$$\text{soft:} \quad u(t) = \begin{cases} \max & t \leq \text{deadline} \\ \max \times \frac{1}{1+(t-\text{deadline})} & t > \text{deadline} \end{cases}$$

3.3 Previous Work

The real-time paradigm used in [6, 7, 8] implies a constant rate input data arrival law. The input stream may or may not be bounded. τ time units elapse between two consecutive data arrivals. The algorithm has to incorporate the newly arrived data into the solution, before the next data arrive. That is, the solution has to be computed in less than τ time units. Problems addressed in this setting are optimization problems (the minimum spanning tree), encryption problems and numerical problems (approximation of a surface integral, equation solving using the bisection method).

Bruda and Akl [14] defined the data-accumulating paradigm. Input is again an endless stream. The arrival laws studied are either polynomial or described by a non-continuous function. For a given problem, the algorithm has to incorporate the newly arrived data into the solution. The algorithm terminates, when all the data received so far is processed. The paradigm is applied to sorting, with a comparison between sequential and parallel algorithms.

Correcting algorithms [18] are a variant of data-accumulating algorithms in which the initial data undergoes small adjustments during the computation. Such problems are also called dynamic, with dynamic data.

3.3.1 Formal Definitions of Real-Time Computation

On Line Algorithms

To characterize the arrival of input on a timely basis Kosaraju [13, 45] has defined the linear-time algorithm which needs $O(n)$ steps to read in the input, where n is the size of the input. In this paradigm the input is allowed to arrive with different constant size time intervals between data items. One data item has to be read in $O(1)$. This condition is not strong enough for many real-time applications. The arrival rate in a practical problem is usually better defined, like one item in 20s. In such a case the constraint $O(1)$ is too vague.

The Real Time Producer/Consumer Paradigm

The producer/consumer paradigm [13, 40] considers two computational entities: the producer and the consumer. They communicate via a stream data flow containing messages. The producer generates the messages and the consumer consumes them in the same order (FIFO). Traditionally, a buffer might be necessary to account for the different rate at which the two parts produce/consume the messages. In real-time, the rate is part of the producer/consumer definition. Thus the buffer can be eliminated.

3.3.2 Parallel Algorithms versus Sequential Algorithms in Real-Time Computation

The usual expectations from parallel computers are that they should perform “better”, which means they should perform faster. Speedup and efficiency are common

measures for parallel algorithms. They all address the time gain obtained by parallelizing the computation.

In real-time computation, time gain is not the only advantage for using a parallel computer. Indeed, the ‘quality’ of a solution may be improved, as the following examples indicate:

1. More accurate solutions can be obtained in numerical problems [8].
2. Closer to optimal solutions can be obtained for optimization problems [3, 6, 49].
3. More secure solutions can be obtained for cryptographic problems [5, 7].
4. Better statistical solutions can be obtained using larger sample sizes [4].
5. Better compression rates can be obtained in source coding applications [4].
6. Better error correction capability can be obtained in data communications [4].
7. Image filtering can use larger filters.
8. Image processing can deal with larger image files, which means either images representing larger scenes or more accurate images of the same scene size.

Chapter 4

Flow Networks in Real-Time

4.1 The Real-Time Paradigm

In the following we will study the maximum flow problem in a real-time paradigm. The paradigm we use is similar to the one described in [3, 14]. The initial input set undergoes a number of corrections during the computation. In our definition, together with the input data, the algorithm also receives an initial solution to the given computational problem. Corrections arrive at constant rate. The time interval between two consecutive corrections is equal to τ time units. τ is fixed in advance and is a parameter of the problem. Corrections may cause a variation in the solution. The algorithm has to compute a new solution before the next correction arrives. Thus the algorithm has to be able to update the solution in τ time units.

4.2 The Real-Time Maximum Flow Problem

The maximum flow problem in the paradigm described above has to allow the structure and attributes of the flow network to be changing in time.

The initial variant of the flow network $N = (G = (V, E), c, s, t)$ together with the already computed maximum flow f is given at the outset. $G = (V, E)$ is an arbitrary oriented graph that may contain cycles. s and t are the source and sink respectively. c , the capacity function, takes only integer values. Thus, by theorem 1 the maximum flow in the network will also be an integer.

The network undergoes small changes during the computation. The changes can be viewed as updates or corrections to the original flow network. Corrections arrive at a constant rate. The time elapsed between two consecutive corrections is τ time units. Each correction possibly determines a variation in the network's maximum flow f . The problem is to compute the new maximum flow before the next correction arrives, that is in τ time units.

One obvious approach to the problem is to compute the new maximum flow using only the updated flow network, which is equivalent to starting from scratch. In this case, we don't make any use of previous computations. Therefore, a better approach would design an algorithm that incorporates the correction in the previous solution, avoiding duplicate/unnecessary computations.

Corrections Let the network be $N = (G = (V, E), c, s, t)$. The following categories of corrections will be studied:

Figure 4.1: Flow network. maximum flow = 7.

Figure 4.2: Unitary edge correction. maximum flow = 8.

Figure 4.3: Edge addition. maximum flow = 10.

Figure 4.4: Edge deletion. maximum flow = 4.

1. At the beginning of the time interval τ an edge capacity gets incremented or decremented by one. This is the simplest possible correction. Nevertheless it can affect the value of the maximum flow, also by one unit. Consider the flow network given in fig. 4.1, with the maximum flow equal to 7. The two numbers on each edge, separated by "/", represent the capacity and the flow respectively. The capacity of edge (v_3, v_5) gets incremented by one unit. We can see in fig. 4.2 that the maximum flow value increases to 8, and also that an arbitrary number of edges are affected: the flows of edges (s, v_3) and (v_5, t) are both increased by one.

2. At the beginning of the time interval τ the capacity of an edge is changed by an arbitrary integer value. Category 1. is a particular case of this category. Edge addition and deletion also fall in this category. An edge addition means a capacity increase from zero to some positive value.

Fig. 4.3 shows the flow network in fig. 4.1 after it has performed an edge addition. The edge (v_3, v_4) with the capacity 3 has been added. Consequently the maximum flow has also increased by 3 units and has the value of 10. Together with the edge (v_3, v_4) , the flow through edges (s, v_3) and (v_4, t) has been increased with 3 units.

Fig. 4.4 shows the effect of an edge deletion on the same flow network in fig. 4.1. Edge (s, v_3) with capacity 11 carrying a flow of 3 units has been removed. The capacity being no longer available produces a flow decrease. The new maximum flow becomes 4. Edges (v_3, v_5) and (v_5, t) also decrease the flow they carry.

Nevertheless, an edge deletion need not necessarily induce a flow decrease. In fig. 4.5, 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 is still 7.

3. A new vertex arrives at the beginning of τ . The vertex comes with a number of edges to connect to vertices already in the network. The new maximum flow can use the capacity offered by the new vertex together with its incoming and outgoing edges. In fig. 4.6 a new vertex has arrived v_{new} in addition to the already existing flow network in fig. 4.1. v_{new} comes

Figure 4.5: Edge deletion. maximum flow = 7.

Figure 4.6: Vertex addition. maximum flow = 14.

Figure 4.7: Vertex deletion. maximum flow = 5

with three edges (v_3, v_{new}) (incoming), (v_{new}, v_5) (outgoing) and (v_{new}, t) (outgoing). The network supports a larger flow through itself of 14 units.

4. The reverse problem to category 3, arises when a vertex is deleted from the network together with its incoming and outgoing edges. Fig. 4.7 shows a reduced flow network after vertex v_1 has been deleted. It reduces the value of the maximum flow to 5, yet almost the whole graph is affected: edges (s, v_3) , (v_2, v_4) , and (v_4, t) .

The significance of the corrections defined above depends on the application and the specific meaning given to vertices, edges and capacities.

Consider the intuitive examples given in chapter 1. In the map of roads example, vertices are towns and edges are roads. Deleting an edge in the graph means that a road has become inaccessible. A new edge represents the construction of a new road. The change in capacity of an edge implies some change in the traffic conditions of the road. Addition and deletion of a vertex express whether a town is allowed to be transited or not.

Likewise in the fluid pipes example. Edges represent pipes and vertices are their joints. Deletion and addition of an edge represent the crash of a pipe and the introduction of a new pipe respectively. The change in capacity of an edge simulates some change in the functionality of a pipe. Addition and deletion of a vertex have less obvious meaning in this case.

In chapter 7 we will define and analyze an application of the real-time maximum flow problem. The application is 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.

Solving the problem for the corrections defined in the general case, leads to immediate solutions for the specific instances of the real-time maximum flow problem.

Chapter 5

Unitary Edge Corrections

The simplest correction possible is when the capacity of an edge changes by one unit. Nevertheless, solving this simple correction type gives us the method to solve all other more complex correction types we have defined in the previous chapter. This chapter is dedicated solely to solving the unitary edge correction: the capacity of an edge changes by one unit; it is either incremented or decremented.

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

Conforming to the real-time setting defined in chapter 4, after each τ time units, an arbitrary edge changes its capacity by one unit. The capacity correction may affect the value of the maximum flow also by one unit.

Suppose the current updated edge is (u, v) , with $u, v \in V$.

If (u, v) is non-saturated, $f(u, v) < c(u, v)$, then its capacity update (incrementation or decrementation) does not affect the maximum flow.

If (u, v) is saturated, $f(u, v) = c(u, v)$, then a capacity incrementation/decrementation

Figure 5.1: Residual network with unitary augmenting path

Figure 5.2: Residual network with rerouting path

might cause a unitary increase/decrease of the maximum flow. Therefore, only saturated edge corrections need to be considered. The value by which f is possibly modified is also unitary.

According to the Max-flow Min-cut theorem (cf. theorem 2) there exist no paths from s to t in the residual network induced by the maximum flow. If (u, v) is saturated, the residual graph contains no directed edge (u, v) . When the capacity $c(u, v)$ 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 . Thus, if there exists a path from s to u and from v to t (cf. fig. 5.1), the incrementation of $c(u, v)$ generates a path from s to t . Further, to update the maximum flow, the flow along the new formed augmenting path has to be incremented.

If $c(u, v)$ is decremented, the algorithm attempts to redirect the now excessive

Figure 5.3: Reduced flow network with decrementing path

unitary flow in (u, v) through other edges (cf. fig. 5.2). It searches for a path from u to v in the residual network¹. 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. For this purpose, we define a new notion, the "reduced flow network". The reduced flow network is the counterpart of the residual network (definition 4).

Definition 14 [33] *For any flow f in network $N = (G = (V, E), c, s, t)$, the reduced flow network of N with respect to f is defined as the 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 $f(u, v) > 0$. With each edge we associate the positive weight $f(u, v)$.*

To decrease the flow, the algorithm searches for a path in the reduced flow graph FG_f , from s to u and from v to t (cf. fig. 5.3). These paths are guaranteed to exist. Further, decrementing along the paths updates the flow.

¹Such 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.

Unitary Correction;**Initial State:**

$N = (G = (V, E), c, s, t);$ << up to date version >>

$f;$ << maximum flow >>

Real Time Correction:

(u, v) = the edge whose capacity changes

cor = the value by which the capacity changes (+/- 1)

Notation:

N_f = current residual network

FN_f = current reduced flow network

```

1       $r(u, v) = c(u, v) - f(u, v);$       << residual capacity >>
2       $c(u, v) = c(u, v) + cor;$       << update capacity >>
3      if  $r(u, v) = 0$  then      <<  $(u, v) \notin N_f$  >>
4          if  $cor = +1$  then
5              in  $N_f$  search path  $p$  from  $s$  to  $t$ 
6              if  $\exists p$  then
7                  increment  $f$  along  $p$ ;
8              endif
9          endif
10         if  $cor = -1$  then
11              $f(u, v) = f(u, v) - 1;$ 
12              $f(v, u) = f(v, u) + 1;$ 
13             in  $N_f$  search path  $p$  from  $u$  to  $v$ ;
14             if  $\exists p$  then
15                 increment  $f$  along  $p$ ;
16             else
17                 in  $FN_f$  search path  $p_1$  from  $s$  to  $u$  and
18                     path  $p_2$  from  $v$  to  $t$ ;
19                 decrement  $f$  along  $p_1$  and  $p_2$ ;
20             endif
21         endif
22     endif
23 endif

```

Figure 5.4: General procedure for unitary correction

The ideas described above are summed up in fig. 5.4 as a general procedure to solve unitary corrections.

This general procedure is independent of the model on which the algorithm is implemented. Note that the only time consuming part of the Unitary Correction procedure is finding and updating the flow on a path in either the residual network (N_f), lines 5, 7, 10, 12, or in the reduced flow network (FN_f), lines 14, 15.

To compare the sequential approach to a parallel solution, the above procedure will be refined to conform to a sequential model, the Random Access Machine (RAM), and to a parallel model, the Reconfigurable Multiple Bus Machine (RMBM). For the sake of comparison, the two models are considered to have processors of equal power, that is the fastest possible processors. To measure the running time of the algorithm, the traditionally defined *time unit* will be used [6]. A time unit is the length of time required by a processor to access a memory location for reading or writing, or perform a constant-time operation (+, −, AND, OR, etc.) see 2.1.

5.1 Sequential Solution

The Sequential Model employed is the conventional Random Access Machine (RAM) [6]. It 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. It may also use the local registers, in which case the memory access is simplified or eliminated.

Conforming to the definition of time unit given above, a memory access or a

constant-time operation costs one time unit.

Implementation and Analysis. The only non-constant steps in the general procedure are 5, 7, 10, 12, 14, and 15. All the other steps have straightforward implementations.

Steps 5, 10, and 14 search a path between two vertices in a graph. The graph is either underlying the residual network or the reduced flow network. In the worst case, a sequential algorithm that searches a path has to inspect all edges [24]. For both graphs the number of edges has the same order with the original graph in the flow network: $O(|E_f|) = O(|FE_f|) = O(|E|)$. Therefore, steps 5, 10, 14 take $\Theta(|E|)$ time in the worst case.

Steps 7, 12, and 15 traverse a path. The length of the path is $O(|V|)$ which also represents the time complexity for these steps.

The overall complexity of the sequential implementation is $\Theta(|V| + |E|)$ in the worst case.

In the real-time paradigm, the correction has to be incorporated into the solution in the interval τ that is available between two consecutive corrections. The sequential solution is meaningful only if $\tau = \Theta(|V| + |E|)$, i.e. the sequential computation is able to terminate before the next correction arrives. If $\tau = o(|V| + |E|)$, the sequential solution is incapable of correcting the value of the maximum flow, and has either to perform a limited number of corrections or simply approximate the maximum flow by the initial maximum flow given at the outset.

Recall that, the sequential algorithms that compute the maximum flow from

scratch (table 2.1) are lower bounded by $\Omega(|V||E|)$ [35]. There exists only one algorithm [35] that has a lower complexity $O(\min(|V|^{\frac{2}{3}}, |E|^{\frac{1}{2}}) \times |E| \log(\frac{|V|^2}{|E|} \log U))$. Nevertheless, this solution depends on U , the maximum capacity value of an edge. We can see from here that our unitary correction algorithm, $\Theta(|V| + |E|)$ in the worst case, is definitely preferable over any attempt to recompute the maximum flow from scratch.

5.2 Parallel Solution

The RMBM, the parallel model used, is reconfigurable. Reconfigurable models are parallel models that have the ability to change their topology during the execution of a program. That is, the structure of the model is changeable. The RMBM is a model with electronic buses. Due to its reconfigurability, buses may have different forms at different moments. Two buses can be fused to form one fused, larger bus, or buses can be segmented to form two or more segments of buses.

Reconfigurable models seem to be appropriate to solve a large number of graph problems [9, 61]. The structure of the model can be configured to capture the structure of the graph. Even dynamically changing graphs can be represented by dynamically reconfiguring the model. The maximum flow problem is a graph problem. In the real-time paradigm, the graph is also dynamically changing in time.

5.2.1 Model of Computation

The model we use for the algorithms is the Reconfigurable Multiple Bus Machine (RMBM) proposed by Trahan et. al [58]. The model consists of m independent

Figure 5.5: The RMBM

processors that communicate via n independent buses (fig. 5.5). 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. A processor might read or write to a bus. The simplest form of a bus permits processors to access it only exclusively. Exclusive read (ER) means that only one processor at a time is allowed to read from the bus (during a communication step). Likewise exclusive write (EW) allows only one processor to gain access to the bus for writing. More advanced buses may allow multiple processors to read concurrently (CR) a certain datum available on the bus. Concurrent write (CW) implies that multiple processors may simultaneously write to the same bus. Conflicts when several processors attempt to concurrently write to a bus are usually solved by allowing the processors to write only if they write the same datum (common concurrent write). Different RMBMs can be defined by the accessibility of their buses: exclusive read - exclusive write (EREW), concurrent read - exclusive write (CREW), etc.

Figure 5.6: One processor of the RMBM

Thus, an RMBM with m processors (indexed $0, 1, \dots, m-1$), n buses (indexed $0, 1, \dots, n-1$) 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 $0 \leq i < m$ and $0 \leq j < n$. 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 5.6) lie physically on bus B_j and are controlled by processor P_i . Their meaning is:

c_{ij0} - controls writing to bus j .

c_{ij1} - controls reading from bus j .

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

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

If f_{ij} and $f_{ij'}$ are both set (fig. 5.5), the buses j and j' are fused.

The RMBM can configure the buses into several "fused bus segments" by means of the segment and fuse switches. In our algorithms we consider a CREW RMBM.

The directional RMBM, an extension proposed by Trahan [58], 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 code of the program. Any line of the program is executed synchronously by all processors. Thus, communication between two processors is possible by synchronizing the writing and reading of the bus.

5.2.2 Basic Techniques

Suppose each of m processors holds a value. A problem that often arises is that a processor has to identify the nearest processor to its left or right that holds the same value. This can be solved by neighbor localization [59]. In the case of the RMBM the following simplified lemma [57] holds:

Lemma 2 *Given m processors with an arbitrary subset of them marked, the minimum/maximum index of the marked processors can be found on an EREW RMBM with m processors and 1 bus in $O(1)$ time.*

To prove the lemma, each marked processor cuts the bus and broadcasts a signal to the left. The marked processor that receives no signal has the maximum index. The minimum is found analogously only that the processor P_{i-1} writes the signal for processor P_i on the bus.

5.2.3 The Configuration of the RMBM

The structure of the RMBM has to be able to meet the challenge of efficiently searching and updating a path in a graph: lines 5, 7, 10, 12, 14, 15 in the general procedure (fig. 5.4). The idea of the parallel procedure is to configure the RMBM such that it represents the graph's structure. Information about the graph, like the existence of a path, can be retrieved by inspecting the configuration of the RMBM.

Figure 5.7: Flow network with capacity/flow

Figure 5.8: Residual network

Figure 5.9: Configuration of the RMBM to represent the residual graph in fig. 5.8

The RMBM has to work on the residual graph and/or the reduced flow graph. Note that these two graphs depend on the specific instance of the problem: each correction may induce a modification in both the residual graph and the reduced flow graph. A change in the set of edges appears, for example, if a residual capacity becomes zero; this is equivalent to deleting the edge from the residual graph. In the same way edges may appear in the residual graph if the residual capacity increases from zero to a positive value (actually only the value one is possible). The reduced flow graph behaves likewise.

For the RMBM, processors represent edges of the graph, and buses represent vertices. Actually, for each edge, there is a whole group of processors ($\mathcal{P}$) that monitor the behavior of the edge, and likewise a whole group of buses ($\mathcal{B}$) is responsible for one vertex.

The reconfigurability of the RMBM is used to include or exclude edges to/from the graph. An edge, that belongs to the present instance of the graph, joins the two vertices on its endpoints. Because the graph is oriented, the joining is directional from the tail to the head of the pointed edge. In terms of the RMBM, the buses representing the two endpoint vertices are fused directionally from tail to head.

The processors in an edge group (v_i, v_j) are only allowed to address buses in the groups of the endpoints ($\mathcal{B}_{v_i}$ and $\mathcal{B}_{v_j}$).

An example graph is given in fig. 5.7. The numbers on each edge represent the capacity and the flow respectively. The residual network induced is given in fig. 5.8. Fig. 5.9 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 (ex. (v_1, v_3) and (v_3, v_1)). The RMBM has only one processor group for both edges (P_{v_1, v_3} and P_{v_3, v_1}). A signal written on the source bus group ($\mathcal{B}_s$) will be seen on all buses drawn with a thick line. Bus group $\mathcal{B}_t$ is not reachable by a signal sent from $\mathcal{B}_s$. This is equivalent to the fact that there is no path between s and t in the residual graph (fig. 5.8).

Thus, for a given (residual / reduced flow) graph, the RMBM can be configured to have the following property. For any two vertices v_i and v_j , if and only if there exists a directed path between them in the graph, then there exists a group of fused directed buses in the RMBM that connects $\mathcal{B}_{v_i}$ to $\mathcal{B}_{v_j}$.

With the method given above, it is possible to determine whether there exists a path in a graph between two arbitrary vertices. Further, the path has to be determined. That is, if more than one path exists, one path has to be selected. For convenience, as explained later, the shortest path will be selected. Once a path is

determined, the edges' flow along the path has to be updated (lines 7, 12, 15 in the general procedure). For the residual graph in fig. 5.8, the configuration of the RMBM is given in fig. 5.9.

The method given in [58] for list ranking, will be adapted to select the shortest path from among the paths connecting v_i to v_j . Each intermediary vertex v_k , including v_j , measures the lengths of the paths from v_i to itself. v_k determines the minimum length and the predecessor on this minimum path.

The length of the paths to v_k will be measured on the buses $\mathcal{B}_{v_k}$. The length of a simple path (without loops) between two arbitrary vertices is at most $|V|$. Thus, the bus group $\mathcal{B}_{v_k}$ contains $|V|$ buses, denoted $B_{v_k, l}$ with $0 \leq l < |V|$. A signal sent on a bus (vertex) gets incremented each time it passes through an edge and arrives at a new vertex (bus). A processor group $\mathcal{P}_{v_i, v_j}$, that represents a valid edge for the present graph instance, fuses the buses incrementally: $B_{v_i, l} \rightarrow B_{v_j, l+1}$ 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. 5.10 refines the upper left corner of fig. 5.9 to show the contents of two bus groups ($\mathcal{B}_s$ and $\mathcal{B}_{v_1}$) and two processor groups ($\mathcal{P}_{s, v_1}$ and $\mathcal{P}_{v_1, s}$). Note that, a signal written on a bus gets only incremented, never decremented. In particular, a path that loops (s, v_1, s) will have a length two units longer than the simple path. Therefore, when selecting the shortest path, loops will be avoided.

Figure 5.10: Detail of the RMBM's configuration in fig. 5.9

The Size of the RMBM

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 in the original flow network. The two groups represent the directed edge and its opposite. 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 $|V|^2$. The size of the RMBM is $\Theta(|V|^3 \times |E|)$. The RMBM thus sized is able to configure any residual or reduced flow network derived from the original flow network.

5.2.4 The Algorithm

Initially, The RMBM has full knowledge about the flow network. All processors in a group $\mathcal{P}_{v_i, v_j}$ know the capacity of the edge $c(v_i, v_j)$ and the maximum flow through

it $f(v_i, v_j)$. Processor $P_{v_i, v_j, 0}$ is designated the leader of the group.

Except for the path searches, the general procedure has a sequential nature. The most appropriate processor on the RMBM to execute it is the leader of the processor group representing the edge that receives the update. The processor group for the updated edge is the one that contains the edge's information (capacity, flow). The RMBM version (fig. 5.13) decides what paths have to be searched for and calls a procedure *FindPath*, that will be executed by all processors on the RMBM.

In the following, we will design a procedure (fig. 5.14 and 5.15) to find and update a path between two arbitrary vertices ss and tt in a graph embedded in an RMBM. The graph can be either the residual graph or the reduced flow graph. In the first case, the graph's edges are defined by the residual capacity $r(v_i, v_j) = c(v_i, v_j) - f(v_i, v_j)$, while in the second case, the edges are defined by the flow $f(v_i, v_j)$. The procedure behaves the same for both.

The RMBM is already configured to represent the graph. A signal will be sent from ss with the hope that it will reach tt . Processors in groups representing edges emerging from ss have access to the bus group representing the vertex ss . Thus, leaders of groups of the form $P_{ss, x}$ (with $(ss, x) \in E$) send a signal on the lowest bus of vertex ss : $B_{ss, 0}$. This signal gets incremented whenever it reaches another vertex and may arrive to tt on different levels, according to the lengths of the different paths. That is, the signal can be read from buses $B_{tt, l}$, $0 \leq l < |V|$, if there exist a path of length l between ss and tt . The same holds for all intermediary vertices v_k .

Group P_{v_i, v_j} will inspect both bus groups it is allowed to access (B_{v_i} and B_{v_j}). Thus, all processors $P_{v_i, v_j, l}$ ($\forall v_i, v_j \in V$ and $1 \leq l < |V|$) read buses $B_{v_i, l}$ and $B_{v_j, l}$. If there is any signal on $B_{v_j, l}$, then v_j is reachable from ss .

Unitary Correction on RMBM:**The RMBM's structure :****Processors :** $2|E|$ groups $\mathcal{P}_{v_i, v_j}$, where $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$.Each $\mathcal{P}_{v_i, v_j}$ contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$. $P_{v_i, v_j, 0}$ is leader of group $\mathcal{P}_{v_i, v_j}$.**Buses :** $|V|$ groups of buses $\mathcal{B}_{v_i}$, where $v_i \in V$.Each $\mathcal{B}_{v_i}$ contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.**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 $(+/- 1)$ **Notation:** N_f = residual network induced by f FN_f = reduced flow network induced by f

```

1   $P_{u, v, 0}$  do
2     $r(u, v) = c(u, v) - f(u, v);$       << residual capacity >>
3     $c(u, v) = c(u, v) + cor;$       << update capacity >>
4    if  $r(u, v) = 0$  and  $cor = +1$  then << capacity increases >>
5       $r(u, v) = r(u, v) + cor ;$ 

      << Embed the graph incrementally >>
5    for all  $P_{v_i, v_j, l}$  do in parallel
6      <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
7      if  $r(v_i, v_j) > 0$  then
8        fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1} ;$ 
9      endif
10   endfor

11    $FindPath(s, t) ;$ 

12   for all marked  $\mathcal{P}_{v_i, v_j}$  do in parallel
13     update  $\mathcal{P}_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) + cor ;$ 
14     update  $\mathcal{P}_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) - cor ;$ 
15   endfor
16   endif

```

Figure 5.11: Unitary correction on RMBM (continued on next page)

Unitary Correction on RMBM;

```

17   if  $r(u, v) = 0$  and  $cor = -1$  then << capacity decrease >>

      << Reduce the flow through the edge >>
18   update  $P_{u,v} : f(u, v) = f(u, v) + cor$  ;
19   update  $P_{v,u} : f(v, u) = f(v, u) - cor$  ;

      << Embed the residual graph incrementally >>
20   for all  $P_{v_i, v_j, l}$  do in parallel
21       <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
22       if  $r(v_i, v_j) > 0$  then
23           fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
24       endif
25   endfor

26        $FindPath(u, v)$  ;

27   for all marked  $P_{v_i, v_j}$  do in parallel
28       update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) + cor$  ;
29       update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) - cor$  ;
30   endfor

```

Figure 5.12: Unitary correction on RMBM (continued on next page)

```

Unitary Correction on RMBM;

31      if (PATH not found) then

          << Embed the reduced flow graph incrementally >>
32      for all  $P_{v_i, v_j, l}$  do in parallel
33          <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
34          if  $f(v_i, v_j) < 0$  then
35              fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
36          endif
37      endfor

          FindPath( $s, u$ ) ;
38      for all marked  $P_{v_i, v_j}$  do in parallel
39          update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) - cor$  ;
40          update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) + cor$  ;
41      endfor

          << Embed the reduced flow graph incrementally >>
42      for all  $P_{v_i, v_j, l}$  do in parallel
43          <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
44          if  $f(v_i, v_j) < 0$  then
45              fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
46          endif
47      endfor

          FindPath( $v, t$ ) ;
48      for all marked  $P_{v_i, v_j}$  do in parallel
49          update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) - cor$  ;
50          update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) + cor$  ;
51      endfor

52      endif
53
54

```

Figure 5.13: Unitary correction on RMBM

Additionally, the group determines the length of the shortest path and whether (v_i, v_j) belongs to a shortest path from ss to v_j . Therefore, the group determines the minimum index l_1 in $\mathcal{B}_{v_i}$ on which the signal was present and then the minimum l_2 in $\mathcal{B}_{v_j}$. This can be achieved in constant time by lemma 2. l_1 represents the length of the shortest path from ss to v_i , and l_2 the length of the shortest path from ss to v_j . If the edge (v_i, v_j) is part of a shortest path from ss to v_j than the two lengths differ by one: $l_{ss, v_j} = l_{ss, v_i} + 1$. In the case when (v_i, v_j) does not belong to a shortest path: $l_1 \geq l_2$, v_j is reached earlier (l_2) on another path, than through v_i , which would be $l_1 + 1$. Once (v_i, v_j) is established to be part of a shortest path from ss to v_j , the group's leader $P_{v_i, v_j, 0}$ enters a race with other processors of the form $P_{v'_i, v_j, 0}$ to elect exactly one shortest path. This can be done by determining another minimum index (lemma 2).

Edges (v_i, v_j) that have failed one of the above tests are withdrawn from the graph. This means, the group P_{v_i, v_j} stops fusing any buses from $\mathcal{B}_{v_i}$ to $\mathcal{B}_{v_j}$.

After the steps above, there exists exactly one path that connects ss to any reachable v_j . In particular, there exist only one path from ss to tt . To mark all vertices/edges on the path, tt will send a signal backwards towards ss . Therefore all processors simply invert their fusing lines: $P_{v_i, v_j, l}$ directionally fuses $\mathcal{B}_{v_j, l+1}$ with $\mathcal{B}_{v_i, l}$. All still active groups P_{v_i, v_j} that read a signal on both $\mathcal{B}_{v_i}$ and $\mathcal{B}_{v_j}$ are edges on the shortest path between ss and tt and have to update their flow according to the required correction.

Find Path on RMBM;

The RMBM's structure :

Processors :

$2|E|$ groups P_{v_i, v_j} , where $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$.

Each P_{v_i, v_j} contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$.

$P_{v_i, v_j, 0}$ is leader of group P_{v_i, v_j} .

Buses :

$|V|$ groups of buses B_{v_i} , where $v_i \in V$.

Each B_{v_i} contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.

Initial State:

$P_{v_i, v_j, 0}$ contains $c(v_i, v_j)$ and $f(v_i, v_j)$.

Input:

ss start vertex of the path

tt end vertex of the path

```

    << Determine reachable vertices from ss >>
1   for all  $P_{v_i, v_j, l}$  do in parallel <<  $(v_i, v_j) \text{ or } (v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
2        $P_{ss, x, 0}$  do <<  $(ss, x) \in E$  >>
3           write '*' on  $B_{ss, 0}$  ;
4           write '*' on  $B_{ss, 0}$  ;
5        $P_{v_i, v_j, l}$  do <<  $(v_i, v_j) \text{ or } (v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
6           read  $B_{v_i, l}$  into star1;
7           read  $B_{v_j, l}$  into star2;
8   endfor

```

Figure 5.14: Find the shortest path in a graph, using the RMBM (continued on next page)

```

    << Determine edges that are on shortest paths >>
  9  for all  $P_{v_i, v_j}$  do in parallel <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E$  >>
 10     $l_1 = \min_{(0 \leq l < |V|-1)} l$ , such that  $B_{v_i, l} = ' \star '$ ; << lemma 2 >>
 11     $l_2 = \min_{(0 \leq l < |V|-1)} l$ , such that  $B_{v_j, l} = ' \star '$ ; << lemma 2 >>
 12    if  $l_2 = l_1 + 1$  then
 13      set  $P_{v_i, v_j}$  active
 14  endfor

    << Select one shortest path >>
 15  for all (active)  $P_{v_i, v_j}$  do in parallel
 16    on  $B_{v_j}$ :  $v_{min} = \min v_i$ ;
 17  endfor

    << mark the edges on the shortest path from  $ss$  to  $tt$  >>
 18  for all (active)  $P_{v_{min}, v_j, l}$  do in parallel
    << Reverse fusing lines >>
 19    fuse  $B_{v_j, l+1} \rightarrow B_{v_i, l}$ ;
 20  endfor
 21  for all (active)  $P_{v_{min}, v_j}$  do in parallel
 22    all  $P_{x, tt, 0}$  do
 23      write ' $\star$ ' on  $B_{tt, l_2}$ 
 24    all  $P_{v_{min}, v_j, 0}$  do
 25      read  $B_{v_j, l_2}$ 
 26  endfor
 27  for all (active)  $P_{v_{min}, v_j}$  do in parallel
 28    if  $B_{v_j, L_2} = ' \star '$  then
 29      mark  $P_{v_{min}, v_j}$ 
 30    if
 31  endfor

```

Figure 5.15: Find the shortest path in a graph, using the RMBM

5.2.5 Analysis of the Parallel Algorithm

A unitary update implies the execution of the main procedure (fig. 5.13). The main procedure calls procedure *FindPath* either once or twice, depending on the value of the *correction(cor)*. Apart from these procedure calls, all steps in the main procedure take constant time. *FindPath* also consists only of steps that take constant time. Therefore, the whole unitary update takes constant time.

It is interesting to observe that the execution time of the unitary correction algorithm does not depend on the flow network on which it works. The size, or the intricacy of the network's underlying graph is immaterial for the parallel algorithm. The size and complexity of the graph is mirrored by the size of the parallel machine, the RMBM.

The size of the RMBM, as evaluated in section 5.2.3, is $\Theta(|V|^3 \times |E|)$, with $\Theta(|V| \times |E|)$ processors and $\Theta(|V|^2)$ buses. Thus, the complexity of the algorithm, defined by the product of the time and size of the machine (chapter 2), is

$$C(G = (V, E)) = \Theta(|V|^3 \times |E|).$$

Conforming to the real-time paradigm, the parallel algorithm has to perform one correction before the next correction arrives. It has τ time units available to accomplish the computation. Note that the execution time of the parallel algorithm is constant. Therefore, even if τ represents only a constant number of time units and is not a function of either $|V|$ or $|E|$, the parallel algorithm is able to incorporate the correction in the maximum flow.

5.3 Comparative Analysis

Given the same unitary flow correcting problem, the two solutions, sequential and parallel, have different capabilities to solve the task. In particular, given the real-time paradigm and the presence of a deadline, the solutions may meet or fail the deadline. To meet the deadline, means to solve the problem entirely. To fail the deadline, can either be considered a complete failure of the computation, or the initial precomputed maximum flow may be considered an approximation of the current solution. For the purpose of this analysis, not meeting the deadline is considered complete failure.

The sequential algorithm runs in $\Theta(|V| + |E|)$ in the worst case and the parallel algorithm runs in $\Theta(1)$. The time interval available to do the computation is τ . τ may vary from one particular practical situation to another. Nevertheless, there is a large range of values τ can take, in which the parallel algorithm is able to recompute the maximum flow, while the sequential algorithm is useless. The range is given by $\Omega(1)$ and $o(|V| + |E|)$. Note that, the execution time of the parallel algorithm does not depend on the size of the flow network ($|V|$ 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 for networks of reasonable sizes.

Recall that the efficiency of a parallel algorithm is defined (chapter 2) as the ratio of the sequential complexity over the parallel complexity:

$$E(1, p) = \frac{T_1}{T_p \times p}.$$

where T_1 and T_p are the execution times of the sequential algorithm and the parallel algorithm, respectively, and p is the number of processors or generally the size of the model.

In our algorithm the efficiency is:

$$E(1, p) = \frac{\Theta(|V| + |E|)}{\Theta(|V|^3 \times |E|)}.$$

The efficiency being less than one, the parallel algorithm is not optimal.

In this chapter we studied unitary edge corrections. The impact of such a correction is small, it may change the value of the flow by one. Nevertheless, the method described here will be used to perform the more general corrections treated in the next chapter: edge capacity corrections by arbitrary values, edge and vertex additions and deletions.

Although one unitary correction produces 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 $\Theta(|V|)$. Increasing unitary edge corrections are produced each τ time unit for $|V|^2$ times. If each correction produces a unitary increase of the flow, it follows that the value of the maximum flow at the end of the computation will be $\Theta(|V|^2)$. Therefore, a computation taking $\tau \times |V|^2$ time produces an increase of the order of magnitude of the maximum flow. That is, depending on the application (the computation's running time), even small corrections may be necessary to be incorporated into the solution.

Chapter 6

Edge and Vertex Corrections in Real-Time

The previous chapter solved the simplest correction, the unitary correction of an edge capacity. In this chapter we will address all other correction types, as defined in chapter 4. They refer to general edge corrections and vertex corrections. The results and algorithms developed in the previous chapter will be used throughout the current chapter. The method used to solve the unitary edge correction gives the general idea applicable to the rest of the correction types.

6.1 Edge Deletion and Addition in Real-Time

The deletion or addition of an edge 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 was also carrying a flow, that flow becomes an excess flow to be redirected or annulled.

Adding an edge to a flow network means to increase its capacity from zero to a positive constant. The increased capacity may increase the maximum flow the network allows.

Both addition and deletion of an edge are particular cases of changing the capacity of an edge by an arbitrary value. We consider that the maximum capacity an edge can have is $|V|^\alpha$, where $0 < \alpha < 1$. Thus, the absolute correction value is upper bounded by the same number.

The paradigm in which we consider the edge corrections to appear, is the same described earlier for the real-time paradigm. Corrections arrive as a stream. The time interval between two consecutive corrections is τ . Each correction has to be incorporated into the solution, the maximum flow, before the next correction arrives. Therefore, the algorithm has τ time units available to complete the task.

6.1.1 Sequential Solution

Consider adding or deleting or simply updating an edge with capacity $a \leq |V|^\alpha$. We will apply Gabow's [30] scaling method, as follows: In each step the algorithm searches for a path of a certain capacity. The starting value is $\lfloor \frac{a}{2} \rfloor$. In each subsequent step the path's capacity to be sought decreases to half. The next path capacity to seek is $\lfloor \frac{a}{4} \rfloor$, then $\lfloor \frac{a}{8} \rfloor$, $\lfloor \frac{a}{16} \rfloor$, etc. 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.

The sequential unitary update can be modified to search for paths of arbitrary value. The algorithm is given in fig. 6.1 and 6.2. Fig. 6.1 treats the case where the capacity of an edge is increased, or an edge is added, while fig. 6.2 shows the decrease on an edge's capacity, or an edge deletion.

Sequential Edge Correction;**Initial State:**

$N = (G = (V, E), c, s, t);$ << up to date version >>

$f;$ << maximum flow >>

Real Time Correction:

(u, v) = the edge whose capacity changes

cor = the value by which the capacity changes $|cor| < |V|^\alpha, 0 < \alpha < 1$

Notation:

$N_f(frac)$ = residual network induced by f ,

$(v_1, v_2) \in N_f(frac)$ only if $r(v_1, v_2) > frac$.

$FN_f(frac)$ = reduced flow network induced by f ,

$(v_1, v_2) \in FN_f(frac)$ only if $f(v_1, v_2) > frac$.

```

1   $r(u, v) = c(u, v) - f(u, v);$       << residual capacity >>
2   $c(u, v) = c(u, v) + cor;$       << update capacity >>
3  if  $r(u, v) = 0$  and  $cor > 0$  then <<CAPACITY INCREASES>>
4   $r(u, v) = r(u, v) + cor$  ;
5   $frac = \frac{cor}{2}$  ; <<  $frac$  is the quantity to be updated in one step >>
6  while  $frac \geq 1$ 
7  repeat
8  in  $N_f(frac)$  search path  $p$  from  $s$  to  $t$ 
9  if  $\exists p$  then
10 increment  $f$  along  $p$ ;
11 endif
12 until no more paths or all corection value has been used
13  $frac = frac/2$  ;
14 endwhile
15 endif

```

Figure 6.1: Sequential edge correction (to be continued)

Sequential Edge Correction;

```

16  if  $r(u, v) = 0$  and  $cor < 0$  then <<CAPACITY DECREASES>>
17     $r(u, v) = r(u, v) + cor$  ;
18     $f(u, v) = f(u, v) + cor$ ;
19     $f(v, u) = f(v, u) - cor$ ;

    << Redirect the flow >>
20     $frac = \frac{cor}{2}$  ; <<  $frac$  is the quantity to be updated in one step >>
21    while  $frac \geq 1$ 
22      repeat
23        in  $N_f$  search path  $p$  from  $u$  to  $v$ ;
24        if  $\exists p$  then
25          increment  $f$  along  $p$ ;
26        endif
27      until no more paths or all corection value has been used
28       $frac = frac/2$  ;
29    endwhile

    << Reduce the flow >>
30     $frac = \frac{cor}{2}$  ;
31    while  $frac \geq 1$ 
32      repeat
33        in  $FN_f$  search path  $p_1$  from  $s$  to  $u$  and
34          path  $p_2$  from  $v$  to  $t$ ;
35        if  $\exists (p_1 \text{ and } p_2)$  then
36          decrement  $f$  along  $p_1$  and  $p_2$ ;
37        endif
38      until no more paths or all corection value has been used
39       $frac = frac/2$  ;
40    endwhile
41  endif
42 endif

```

Figure 6.2: Sequential edge correction

In the worst case, augmenting paths will all have capacity one, which is the smallest possible. Therefore, the number of unitary edge correction that have to be applied is equal to the capacity correction a . Recall that the sequential unitary correction algorithm takes $\Theta(|V|+|E|)$ in the worst case. Therefore, 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 result is still better than computing the maximum flow from scratch, i.e. $O(|V| \times |E|)$.

In the real-time setting, if the time interval τ available to perform the edge correction is less than $\Theta(|V|^\alpha \times (|V|+|E|))$, that is $\tau = o(|V|^\alpha \times (|V|+|E|))$, then the sequential algorithm will not be able to recompute the maximum flow before the deadline.

6.1.2 Parallel Solution

The parallel solution for the RMBM is also a variant of the unitary update given in chapter 5. The same paths are to be sought in this case, as in the algorithm given there for the RMBM. The difference is that we use again the scaling method. Therefore, the paths that we seek are of certain capacities. In the unitary correction algorithm, paths of capacity one were found. In the edge correction case, in one iteration of the algorithm the capacity of the path to be sought is predetermined and only paths of that or larger capacities are valid solutions for the iteration.

The algorithm is given in figs. 6.3, 6.4, 6.5. It has two main parts:

1. Fig. 6.3 treats the case where the capacity increases, or there is an edge addition. Augmenting paths are searched for in the residual graph from the source to the sink. Each existing augmenting path produces an increase of the maximum flow, equal to the value of *frac* for that iteration.

Edge Correction on RMBM;**The RMBM's structure :**

Processors : $2|E| + 2$ groups $\mathbb{P}_{v_i, v_j}$, where $(v_i, v_j) \in E$
or $(v_j, v_i) \in E$ and two additional empty groups.

Each $\mathbb{P}_{v_i, v_j}$ contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$.

$P_{v_i, v_j, 0}$ is leader of group $\mathbb{P}_{v_i, v_j}$.

Buses : $|V|$ groups of buses $\mathbb{B}_{v_i}$, where $v_i \in V$.

Each $\mathbb{B}_{v_i}$ contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.

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, where $c(u, v) + cor \leq U$

Notation: $N_f(frac)$ = residual network induced by f ,

$(v_1, v_2) \in N_f(frac)$ only if $r(v_1, v_2) > frac$.

```

1   $P_{u, v, 0}$  do
2     $r(u, v) = c(u, v) - f(u, v);$       << residual capacity >>
3     $c(u, v) = c(u, v) + cor;$           << update capacity >>
4    if  $r(u, v) = 0$  and  $cor > 0$  then <<CAPACITY INCREASES>>
       $r(u, v) = r(u, v) + cor;$ 
5     $frac = \frac{cor}{2};$  <<  $frac$  is the quantity to be updated in one step >>
6    while  $frac \geq 1$ 
7      repeat
8        << Embed the graph incrementally >>
9        for all  $P_{v_i, v_j, l}$  do in parallel
10           <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E; 0 \leq l < |V| - 1$  >>
11           if  $r(v_i, v_j) \geq frac$  then
12             fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1};$ 
13           endif
14         endfor
15       FindPath( $s, t$ );
16       for all marked  $\mathbb{P}_{v_i, v_j}$  do in parallel
17         update  $\mathbb{P}_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) + frac;$ 
18         update  $\mathbb{P}_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) - frac;$ 
19       endfor
20     until no more paths
21      $frac = \lfloor frac/2 \rfloor;$ 
22   endwhile
23   endif

```

Figure 6.3: Edge correction: capacity increases (continued on next page)

Edge Correction on RMBM;**Notation:**

$FN_f(frac)$ = reduced flow network induced by f ,
 $(v_1, v_2) \in FN_f(frac)$ only if $f(v_1, v_2) > frac$.

```

17   if  $r(u, v) = 0$  and  $cor < \text{then} << \text{CAPACITY DECREASES} >>$ 

      << Reduce the flow through the edge >>
23   update  $P_{u,v} : f(u, v) = f(u, v) + cor$  ;
24   update  $P_{v,u} : f(v, u) = f(v, u) - cor$  ;

25    $frac = \frac{cor}{2}$  ;
26   while  $frac \geq 1$ 
27     repeat

      << Embed the residual graph incrementally >>
28     for all  $P_{v_i, v_j, l}$  do in parallel
29       <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E ; 0 \leq l < |V| - 1$  >>
30       if  $r(v_i, v_j) \geq frac$  then
31         fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
32       endif
33     endfor

34     FindPath( $u, v$ ) ;

35     for all marked  $P_{v_i, v_j}$  do in parallel
36       update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) + frac$  ;
37       update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) - frac$  ;
38     endfor
39     if (PATH found) then
40        $cor = cor - frac$  ; <<  $frac$  is succesfully redirected >>
41     until no more paths
42      $frac = \lfloor frac/2 \rfloor$  ;
43   endwhile

```

Figure 6.4: Edge correction: capacity decreases (continued on next page)

Edge Correction on RMBM;

```

44      if  $cor < 0$  then
45           $frac = \frac{cor}{2}$  ;
46      while  $frac \geq 1$ 
47          repeat

              << Embed the reduced flow graph incrementally >>
48          for all  $P_{v_i, v_j, l}$  do in parallel
49              <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E$  ;  $0 \leq l < |V| - 1$  >>
50              if  $f(v_i, v_j) < 0$  then
51                  fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
52              endif
53          endfor
54               $FindPath(s, u)$  ;
55          for all marked  $P_{v_i, v_j}$  do in parallel
56              update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) - cor$  ;
57              update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) + cor$  ;
58          endfor

              << Embed the reduced flow graph incrementally >>
59          for all  $P_{v_i, v_j, l}$  do in parallel
60              <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E$  ;  $0 \leq l < |V| - 1$  >>
61              if  $f(v_i, v_j) < 0$  then
62                  fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
63              endif
64          endfor
65               $FindPath(v, t)$  ;
66          for all marked  $P_{v_i, v_j}$  do in parallel
67              update  $P_{v_i, v_j} : f(v_i, v_j) = f(v_i, v_j) - cor$  ;
68              update  $P_{v_j, v_i} : f(v_j, v_i) = f(v_j, v_i) + cor$  ;
69          endfor

70      until no more paths
71       $frac = \lfloor frac/2 \rfloor$  ;
72  endwhile
73  endif

```

Figure 6.5: Edge correction: capacity decreases

2. Figs. 6.4, 6.5 treat the case when the capacity decreases, or an edge is deleted. Thus, the algorithm first tries to redirect the flow that is reduced and if not possible, reduces the maximum flow in the network.

In the worst case, all successful iterations of the program, path updates, are unitary. As the correction has the maximum value $|V|^\alpha$, $0 < \alpha < 1$, the algorithm iterates at most $|V|^\alpha$ times. All steps in one iteration, including the path search, take constant time. Therefore the running time of the algorithm is $\Theta(|V|^\alpha)$ in the worst case. As the time interval τ was also defined to be $|V|^\alpha$, the parallel algorithm is capable of recomputing the maximum flow for the general edge correction.

6.2 Vertex Deletion and Addition

Corrections pertaining to a vertex amount to either deleting an existent vertex from the graph or adding a new vertex to the existing graph. When a vertex is deleted, the edges that connect the vertex to the graph (both incoming and outgoing) also have to be deleted. Likewise a new vertex arrives together with a number of edges connecting to the graph. We make the assumption that the degree of the vertex to be added or deleted is constant. That means the number of both incoming and outgoing edges is constant, that is, it does not depend on the size of the graph. In the next chapter, we will see that this assumption is true for the application we consider. The algorithm that deletes a vertex (fig. 6.6) deletes sequentially all its incoming and outgoing edges. Because the degree of the vertex is constant, deleting a vertex increases the time taken for an edge deletion by a constant. Therefore, the running time for a vertex deletion is still $\Theta(|V|^\alpha)$ in the worst case. The algorithm that adds a vertex (fig. 6.7) assigns processors for all outgoing edges and then performs edge additions sequentially for

Vertex Deletion;

The RMBM's structure :

Processors :

$2|E|$ groups P_{v_i, v_j} , where $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$.

Each P_{v_i, v_j} contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$.

$P_{v_i, v_j, 0}$ is leader of group P_{v_i, v_j} .

Buses :

$|V|$ groups of buses B_{v_i} , where $v_i \in V$.

Each B_{v_i} contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.

Initial State:

$P_{v_i, v_j, 0}$ contains $c(v_i, v_j)$ and $f(v_i, v_j)$.

Real Time Correction:

u = the vertex to be deleted

```

1  while  $\exists P_{x, u}$  or  $P_{u, x} \ll (x, u) \in E, (u, x) \in E \gg$ 
2    for all  $P_{x, u} \ll (x, u) \in E \gg$ 
3      determine the group with the smallest index  $P_{x_m, u}$ 
3      such that  $(x_m, u) = \min(x, u)$ ;
         $\ll$ either the group physically the leftmost  $\gg$ 
         $\ll$ or such that  $(x + u)$  is minimum  $\gg$ 
4      DeleteEdge  $(x_m, u)$  ;
    endfor
  endwhile
```

Figure 6.6: Delete Vertex

Vertex Addition;**The RMBM's structure :****Processors :**

$2|E| + 2(\alpha + \beta)$ groups P_{v_i, v_j} , where $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$. (α and β are defined below)

Each P_{v_i, v_j} contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$.

$P_{v_i, v_j, 0}$ is leader of group P_{v_i, v_j} .

Buses :

$|V| + 1$ groups of buses B_{v_i} , where $v_i \in V$.

Each B_{v_i} contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.

Initial State:

$P_{v_i, v_j, 0}$ contains $c(v_i, v_j)$ and $f(v_i, v_j)$.

Real Time Correction:

u = the vertex to be added

```

    << Assign groups to all outgoing edge >>
1  if  $(u, x) \in E$  then
2      assign processors for  $P_{x, u}$  and  $P_{u, x}$  with
3           $c(u, x); c(x, u) = 0$  ;
4           $f(u, x) = f(x, u) = 0$  ;
5  endif

    << Update flow for each incoming edge >>
6  while  $\exists P_{x, u}$  or  $P_{u, x}$  <<  $(x, u) \in E, (u, x) \in E$  >>
7      for all  $P_{x, u}$  <<  $(x, u) \in E$  >>
8          determine the group with the smallest index  $P_{x_m, u}$ 
9          such that  $(x_m, u) = \min(x, u)$ ;
10         <<either the group physically the leftmost >>
11         <<or such that  $(x + u)$  is minimum >>
12         AddEdge  $(x_m, u)$  ;
13     endfor
14 endwhile

```

Figure 6.7: Add Vertex

all incoming edges. Again, given the constant degree of added vertices, the execution time is given by the execution time of an edge addition: $\Theta(|V|^\alpha)$ in the worst case.

6.3 Conclusion

This chapter has treated the more general corrections in real time: arbitrary edge capacity corrections and vertex addition/deletion.

The following assumptions have been made. The time interval between two consecutive corrections is $\tau = \Theta(|V|^\alpha)$, $0 < \alpha < 1$. Similarly, the maximum capacity an edge may have is $\Theta(|V|^\alpha)$. A vertex that is added or deleted has constant degree.

Under these assumptions, the parallel algorithms designed are able to meet the requirements imposed by the real-time paradigm, since they meet the deadline within $\Theta(|V|^\alpha)$ time units. By contrast, the sequential solutions need more time, namely, $\Theta(|V|^\alpha \times (|V| + |E|))$. Recall from chapter 3, that a firm deadline means that the output is completely useless if produced after the deadline. Thus, the sequential algorithms are not capable of solving the problem in real-time, with firm deadlines. In this case, the use of a parallel machine makes the difference between success and failure.

Chapter 7

An Application: Process Scheduling

This chapter aims to illustrate the usefulness of real-time algorithms for the maximum flow problem on a practical example. The application is in the field of process scheduling in a distributed environment. Explicitly, we will develop a real-time (dynamic) extension to Stone's [55] module allocation problem, a static process scheduling method.

A *process* is a program entity defined by an executable code and private data. Among process scheduling methods, we distinguish static process scheduling and dynamic process scheduling [23].

Static process scheduling relies on previous knowledge about the number of processes, the communication among them, and the resources they use. The interaction between processes can be modeled in several ways. One possibility is to define a precedence relation among processes. Thus, the scheduling procedure views the precedence relation as a constraint applied to the schedule. On the other hand, processes can be

viewed to run concurrently and involve communication among them. Communication between two processes has a certain cost. A good quality schedule keeps the overall cost as low as possible. Stone's method [55] uses the second communication model.

Dynamic process scheduling allows a process to physically migrate from one processor to another.

7.1 The Module Allocation Problem

The *Module Allocation Problem* was first defined by Stone [55]. The definition we will give here is the problem's version given in [23]. Consider a distributed application that consists of n processes that runs on a distributed system with m processors (machines). An objective function is defined to evaluate the communication cost and the process execution cost. The problem is to assign processes to processors such that the objective function is minimum. There are no precedence constraints among processes. Still they need to communicate. The completion time of processes is unspecified. In the static case, originally formulated, processes can be considered to run indefinitely.

7.1.1 Communication Cost

Communication between two processes incurs a cost. 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 G with the set of vertices V and the set of edges E . Vertices

Figure 7.1: Communication process model

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. 7.1 shows a communication process model with 7 processes: A , B , C , D , E , F , and G . The weights are drawn on each edge. The amount of interaction between processes A and B is 8.

7.1.2 Process Execution Cost

A process needs different resources (memory, input/output devices, etc.). While processors are differently equipped, they might be more or less appropriate for a process to run. Therefore, process execution cost is a function of the processor on which it runs.

The execution cost is initially given in the form of a table. As our further study will be restricted to two processors, the table will contain entries for two processors, denoted by P_1 and P_2 . For the seven processes defined in fig. 7.1, let the execution

Process	Cost on P_1	Cost on P_2
A	5	5
B	5	∞
C	2	7
D	8	5
E	∞	3
F	4	4
G	3	6

Table 7.1: Process execution cost

costs be defined by the table 7.1. If some process needs a resource which is not offered by a particular machine, the process will simply not be able to run on that machine. In our example, process B cannot run on processor P_2 , thus the cost is infinity. The same reasoning holds for process E running on processor P_1 . Therefore the two processes B and E will necessarily run on P_1 and P_2 respectively.

7.1.3 Objective Function

The objective function that the module allocation problem seeks to minimize, is simply the sum of the communication cost and the execution cost. With $e_j(P_i)$ denoting the execution cost of process j on processor P_i and $c_{i,j}(P_i, P_j)$ denoting the communication cost between processes i and j when running on processors P_i and P_j respectively, the objective function (cost) becomes:

$$Cost(G, P) = \sum_{j \in V(G)} e_j(P_i) + \sum_{(i,j) \in E(G)} c_{i,j}(P_i, P_j)$$

The module allocation problem with the number of processes and processors arbitrary is NP-complete [55]. Stone [55] solves the allocation problem for two processors in polynomial time.

Figure 7.2: Communication process graph

7.1.4 Stone's Solution

The problem is to map the processes of an application to exactly two processors, such that the objective function described above becomes minimal.

To start, let us consider only the communication cost. Any cut (as defined in chapter 2) in the communication process graph (fig. 7.1) defines a possible assignment of the processes to two processors. Moreover the cut set edges describe exactly the communication cost between the two processors. Therefore, a minimum cut in the graph (fig. 7.2), will give the optimum assignment to minimize the communication cost. In fig. 7.2 the minimum communication cost is 14.

Note that, the graphs we have discussed so far were directed, while the graph in this application is undirected. The undirected graph can be transformed into a directed graph to suit the solutions given in the previous chapters. Any undirected edge is transformed into a pair of two 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.

Figure 7.3: Flow network for process scheduling

Figure 7.4: Flow network with cuts

Stone builds a flow network based on both the communication process graph and the process execution table. Two vertices with the role of source and sink are added to the communication process graph. The source represents processor P_1 and the sink represents processor P_2 . Further, edges are added for each cost in the process execution table. The edge connecting processor P_1 to some process X ($X \in \{A, B, C, D, E, F, G\}$) carries a weight equal to the execution cost of process X on machine P_2 , and vice-versa (see fig. 7.3). Note that the indices are different for the processor's vertex and weight.

In this 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. 7.4, 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 A, B, C, F, G to processor P_1 and processes D, E to processor P_2 . Note that the additional edges connecting processes to P_1 or P_2 that are part of the cut represent exactly the execution cost. Thus, edge (B, P_2) is weighted with the execution cost of B on P_1 , which is according to the assignment given by CUT_2 .

Therefore the binary module allocation problem reduces to determining the minimum cut in a flow network.

7.1.5 The Minimum Cut Problem

The minimum cut problem is formulated either as the $s - t$ minimum cut problem or as the general minimum cut problem in a graph. The $s - t$ minimum cut problem requires that the source s and the sink t belong to different partitions as defined by the cut. The general minimum cut problem makes no distinction among the vertices of the graph, there are no constraints as to what vertices shouldn't belong to the same partition.

Recall that the Max Flow - Min Cut theorem (see chapter 2) states that the value of the maximum flow in a network is equal to the minimum cut of the network. Therefore, determining the maximum flow induces a solution to the $s - t$ minimum cut problem.

Other approaches, avoiding the computation of flows, were subsequently developed and proved to be more time efficient. Nagamochi and Ibaraki [48] give a procedure to identify an edge in the graph which does not belong to the minimum cut set (of

edges). The procedure takes $O(|E| + |V| \log |V|)$ time. Thus the algorithm that does find the minimum cut eliminates iteratively all the other edges; it runs in $O(|E| |V| + |V|^2 \log |V|)$.

Randomized algorithms for the minimum cut can be found in [41, 42]. The minimum cut is found with high probability. The best algorithm [41] runs in $O(|E| \log^3 |V|)$.

7.2 Two Processor Allocation Problem in Real-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. These conditions describe a new problem, namely process allocation in a real-time paradigm.

Consider that a complex computation goes on an indefinite amount of time. During its running, 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.

The cost (of both execution or communication) is upper bounded by $|V|^\alpha$, where $0 < \alpha < 1$. At the beginning of the computation the maximum flow and the minimum cut are given. The underlying graph undergoes changes (corrections) during the computation. Corrections occur one at a time. The interval between two consecutive corrections is fixed and equal to $|V|^\alpha$. The following corrections will be considered:

1. The capacity of an edge changes by an amount a , where a can be positive or negative, but the absolute value $|a| \leq |V|^\alpha$, 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 two processes start to communicate. Similarly, deleting an edge, i.e. reducing its capacity to zero, means 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 (processor 1) and the other connecting to the sink (processor 2). This means 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 the processes on the two processors has to be determined in $|V|^\alpha$ time units, before another correction arrives. The best schedule (as defined in the previous section) is given by the minimum cut of the $s - t$ flow network. Therefore, the problem is to determine the new minimum cut in $|V|^\alpha$ time units.

7.2.1 Sequential Solution

The $s-t$ graphs 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 at least $|V| - 2$ edges. A sequential algorithm that computes the minimum cut from scratch has to be able to inspect the edges of the cut. The time interval between two consecutive corrections is less than the time to inspect the cut: $\tau = |V|^\alpha < |V| - 2$. It follows, that the sequential machine is not capable of computing a new minimum cut.

Nevertheless, the minimum cut for the initial graph is given at the outset. In consequence, the sequential algorithm has the only choice to approximate the minimum cut throughout the computation by the minimum cut at start. This approximation may grow worse with each correction. Consider the flow network given in fig. 7.5. All process execution costs are one. The minimum cut is $MinCut = |V| - 2 + 1 = |V| - 1$. The next correction (see fig. 7.6) increases the cost of edge (D, E) : $c(D, E) \leftarrow |V|^\alpha$. The new minimum cut is

$$MinCut = |V|. \quad (7.1)$$

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

$$SeqCut(1 \text{ step}) = |V| - 1 + |V|^\alpha. \quad (7.2)$$

If the subsequent corrections (see fig. 7.7) are $c(C, E) \leftarrow |V|^\alpha$, $c(B, E) \leftarrow |V|^\alpha$, $c(D, F) \leftarrow |V|^\alpha$, $c(D, G) \leftarrow |V|^\alpha$, and $c(D, H) \leftarrow |V|^\alpha$, the minimum cut remains the same $MinCut = |V|$, yet the sequential computer outputs:

$$SeqCut(6 \text{ steps}) = |V| - 1 + 6 \times |V|^\alpha. \quad (7.3)$$

The accuracy of the sequential cut can continuously decrease for $|V|^2$ steps, the maximum number of edges that can be added to the initial cut:

$$SeqCut(|V|^2 \text{ steps}) = |V| - 1 + |V|^2 \times |V|^\alpha = \Theta(|V|^{2+\alpha}). \quad (7.4)$$

In general, the initial cut could receive $\Theta(|V|^2)$ new edges of the maximum capacity.

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:

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

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 7.1 and 7.2, is:

$$accuracy_{sequ}(1 \text{ step}) = \frac{|V|}{|V| - 1 + |V|^\alpha} = \Theta(1).$$

Thus, after one step, the sequential solution is still asymptotically optimal. Nevertheless, for $|V|^2$ worst case steps, from equations 7.1 and 7.4, the accuracy becomes:

$$accuracy_{sequ}(|V|^2 \text{ steps}) = \frac{\Theta(|V|)}{\Theta(|V|^{2+\alpha})} = \frac{1}{\Theta(|V|^{1+\alpha})}.$$

Thus, the sequential algorithm is not optimal any more.

Figure 7.5: Initial minimum cut

Figure 7.6: Minimum cut after increasing $c(D, E)$ and sequential cut

Figure 7.7: Minimum cut and sequential cut in worst case

7.2.2 Parallel Solution

Parallel algorithms for edge update and insertion/deletion of a vertex were already designed in the previous chapter. Recall that

- an edge update takes $\Theta(|V|^\alpha)$ time in the worst case and
- a vertex addition or deletion takes $\Theta(|V|^\alpha)$ time.

Once the maximum flow is computed, the minimum cut can be determined using the following algorithm. Reconfigure the RMBM to reflect the residual graph: saturated edges do not belong to the graph. 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. The algorithm is given in fig. 7.8. Note that the algorithm determines the minimum cut but does not compute its value.

The most time consuming part of the algorithm is the edge update. The overall running time is $\Theta(|V|^\alpha)$. Therefore, the parallel algorithm is capable of meeting the deadline, and always outputs the optimum process schedule. The RMBM contains $\Theta(|V| \times |E|)$ processors and $\Theta(|V|^2)$ buses.

Because the parallel algorithm always outputs the optimal solution, its accuracy is 1.

7.2.3 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

Minimum Cut;

The RMBM's structure :

Processors :

$2|E|$ groups $\mathcal{P}_{v_i, v_j}$, where $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$.

Each $\mathcal{P}_{v_i, v_j}$ contains $|V| - 1$ processors $P_{v_i, v_j, l}$, where $0 \leq l < |V| - 1$.

$P_{v_i, v_j, 0}$ is leader of group $\mathcal{P}_{v_i, v_j}$.

Buses :

$|V|$ groups of buses $\mathcal{B}_{v_i}$, where $v_i \in V$.

Each $\mathcal{B}_{v_i}$ contains $|V|$ buses $B_{v_i, l}$, where $0 \leq l < |V|$.

Initial State:

$P_{v_i, v_j, 0}$ contains $c(v_i, v_j)$ and $f(v_i, v_j)$.

```

    << Embed the graph >>
1   for all  $P_{v_i, v_j, 0}$  do in parallel
2       fuse  $B_{v_i, l} \rightarrow B_{v_j, l+1}$  ;
    endfor
3   for all  $P_{v_i, v_j, l}$  do in parallel
4       all  $P_{s, v, 0}$  do    <<  $(s, v)$  or  $(v, s) \in E$  >>
5           write ' $\star$ ' on  $B_{s, 0}$ 
6           write ' $\star$ ' on  $B_{s, 0}$ 
7       all  $P_{v_i, v_j, l}$  do in parallel    <<  $(v_i, v_j)$  or  $(v_j, v_i) \in E, 0 \leq l < |V| - 1$  >>
8           read  $B_{v_i, 0}$ 
9           read  $B_{v_j, 0}$ 
    endfor
10  for all  $P_{v_i, v_j, l}$  do in parallel
11      if  $B_{v_i, 0} = \star$  and  $B_{v_j, 0} \neq \star$  then
12          curr_cut( $v_i$ ) = ' $s'$ ' ;
14          curr_cut( $v_j$ ) = ' $t'$ ' ;
      endif
11      if  $B_{v_i, 0} \neq \star$  and  $B_{v_j, 0} = \star$  then
12          curr_cut( $v_i$ ) = ' $t'$ ' ;
14          curr_cut( $v_j$ ) = ' $s'$ ' ;
      endif

```

Figure 7.8: Determine Minimum Cut

the sequential and the parallel algorithm:

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

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 with this term in each step. The accuracy ratio becomes:

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

This means the sequential computer does not asymptotically perform worse than the parallel computer in one step.

Nevertheless, the sequential computer can continually decrease its performance. As seen in the example of the worst case (subsection 2.1), the sequential computer can steadily decrease its accuracy for $\Theta(|V|^2)$ steps.

Therefore the accuracy ratio after these steps becomes:

$$\text{accuracy ratio}(|V|^2 \text{ steps}) = \frac{1}{\frac{\Theta(|V|)}{\Theta(|V|^{2+\alpha})}} = \Theta(|V|^{1+\alpha}).$$

Thus, the parallel computer performs asymptotically better than the sequential computer 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. The cumulative error is the sum of the errors committed while incorporating all corrections. In the sequential worst case:

$$\text{cumulative error}_{\text{sequ}} = |V|^\alpha + 2 \times |V|^\alpha + \dots + |V|^2 \times |V|^\alpha = \Theta(|V|^{1+\alpha}).$$

The parallel algorithm commits no error:

$$\textit{cumulative error}_{\textit{par}} = 0.$$

The number of processors in the RMBM is $\Theta(V^3)$. Therefore, the improvement achieved by the parallel computer is superlinear in the number of processors.

Chapter 8

Conclusion

The maximum flow problem has been widely studied for approximately 30 years and two main directions to solve it have been developed: the Ford-Fulkerson method and the preflow-push method. Algorithms of lower complexity have been developed if planar graphs underly the flow network. Parallel algorithms have also been proposed. They rely on the preflow-push method.

Real-time computation includes the time as a parameter to the definition of a computational problem. The problem imposes time constraints on the input and/or output of the computation. Different real-time paradigms have been recently proposed.

This thesis 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 CREW 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 the 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 [55] (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 (execution cost, and communication cost). These assumptions are restrictive in a practical environment. We gave a definition in real-time of the two processor allocation-problem that takes into consideration the change of the characteristics of a computation in time. Communication among processes varies in time, or two processes might start to communicate at some moment and cease to communicate later on. On the other hand, 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. The real-time setting, in which the problem was studied, gives a formal definition for the changes (viewed as corrections) of the processes' characteristics.

Sequential and parallel solutions have been proposed to the real-time two processor allocation problem. The sequential solution was designed on the RAM model and the parallel one on the CREW RMBM model. Both solutions have to meet the

requirements of the real-time paradigm, which imposes deadlines on the computation. As a result, the parallel algorithm is consistently computing the optimal solution before the deadline. The sequential algorithm is able to give only an approximation of the optimal scheduling. The accuracy ratio between the parallel and the sequential solution is $\Theta(|V|^{1+\alpha})$, in the worst case.

In the problem and application studied in this thesis, the algorithm is passive. The stream of corrections describes the evolution of a system (in our case, 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: 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 on 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, etc.). Tuning the system means modifying the flow in the system through subsequent corrections.

Bibliography

- [1] R. K. Ahuja and J. B. Orlin. A fast and simple algorithm for the maximum flow problem. Technical Report No. 1905-87, Sloan School of Management. Massachusetts Institute of Technology, Cambridge, Massachusetts, 1987.
- [2] S. G. Akl. *Parallel Computation, Models and Methods*. Prentice-Hall International, Engelwood Cliffs, NJ, 1993.
- [3] 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.
- [4] 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.
- [5] S. G. Akl and S. D. Bruda. Improving a solution’s quality through parallel processing. *The Journal of Supercomputing*. to appear.
- [6] S. G. Akl and S. D. Bruda. Parallel real-time optimization: Beyond speedup. *Parallel Processing Letters*, 9:499–509, 1999.

- [7] 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.
- [8] 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.
- [9] H. M. Alnuweiri. A fast reconfigurable network for graph connectivity and transitive closure. *Parallel Processing Letters*, 4(1-2):105–115, June 1994.
- [10] R. Anderson and J. C. Setubal. A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of Parallel and Distributed Computing*, 29:17–26, 1995.
- [11] C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks*. John Wiley & Sons, New York, 1962.
- [12] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.
- [13] S. D. Bruda. *Searching for Superunitary Behavior: Nondeterministic and Real-Time Computations*. depth paper conforming to the Departmental requirements for the PhD programme, Department of Computing and Information Science. Queen's University, Kingston, Ontario, 1999.

- [14] 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.
- [15] S. D. Bruda and S. G. Akl. The characterization of data-accumulating algorithms. *Theory of Computing Systems*, 33:85–96, Jan. 2000.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.

- [20] S. D. Bruda and S. G. Akl. A case study in real-time parallel computation: Correcting algorithms. *to appear in Journal of Parallel and Distributed Computing*. 2001.
- [21] J. Cheriyan and T. Hagerup. A randomized maximum-flow algorithm. *SIAM Journal of Computing*, 24(2):203–226, April 1995. A preliminary version appeared in FOCS 1989.
- [22] R. V. Cherkasky. An algorithm for constructing maximal flows in networks with complexity of $o(v^2\sqrt{E})$ operations. *Math.Methods solution Econ. Probl.*, 7:112–125, 1977. (in Russian).
- [23] R. Chow and T. Johnson. *Distributed Operating Systems & Algorithms*. Edison-Wesley, 1997.
- [24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.
- [25] U. Derigs and W. Meier. Implementing Goldberg’s max-flow-algorithm - a computational investigation. *ZOR - Methods and Models of Oper. Res.*, 33:383–403. 1989.
- [26] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.
- [27] 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.
- [28] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
- [29] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press. Princeton, NJ, 1973.
- [30] H. N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, Oct. 1985.
- [31] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximum flow problem. *Acta Informatica*, 14:221–242, 1980.
- [32] 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.
- [33] S. Ghosh, A. Gupta, and S. V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. *Distributed Computing*, 10(4):167–180, 1997.
- [34] A. V. Goldberg. Recent developments in maximum flow algorithms. Technical Report No. 98-045, NEC Research Institute, Inc., Apr. 1998.
- [35] 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.
- [36] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35, 4:921–940, 1988.

- [37] 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.
- [38] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 2001.
- [39] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8(2):135–150, 1979.
- [40] K. Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, pages 796–804, 1993.
- [41] D. R. Karger. Minimum cuts in near-linear time. In *Proceedings of the 28th annual ACM Symposium on Theory of Computing*, pages 56–63, 1996.
- [42] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, July 1996.
- [43] 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.
- [44] V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.
- [45] S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proceedings of STOC 94*, pages 310–316, Montreal, Quebec, Canada, May 1994.

- [46] L. F. Lindon. *Synergy in Parallel Computation*. PhD thesis, Department of Computing and Information Science, Queen's University, Kingston, Canada. 1996.
- [47] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7:277-278. 1978.
- [48] H. Nagamochi and T. Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal of Discrete Mathematics*, 5(1):54-66, 1992.
- [49] 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.
- [50] S. Phillips and J. Westbrook. On line load balancing and network flow. pages 402-411, 1992.
- [51] Y. Shiloach. An $O(nI \log^2 I)$ maximum-flow algorithm. Technical Report STAN-CS-78-802, Computer Science Department, Stanford University, Stanford, California, 1978.
- [52] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel MAX-FLOW algorithm. *Journal of Algorithms*, 3(2):128-146, June 1982.
- [53] D. D. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, Stanford, California, 1980.
- [54] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362-391, 1983.

- [55] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, 1977.
- [56] R. E. Tarjan. A simple version of karzanov’s blocking flow algorithm. *Operations Research Letters*, 2:265–268, 1984.
- [57] J. L. Trahan, R. Vaidyanathan, and Subbaraman. Constant time graph algorithms on the reconfigurable multiple bus machine. *JPDC: Journal of Parallel and Distributed Computing*, 46, 1997.
- [58] 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, 10 Apr. 1996.
- [59] R. Vaidyanathan. Sorting on PRAMs with reconfigurable buses. *Information Processing Letters*, 42(4):203–208, June 1992.
- [60] U. Vishkin. A parallel blocking flow algorithm for acyclic networks. *Journal of Algorithms*, 13:489–501, 1992.
- [61] B.-F. Wang and G.-H. Chen. Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):500–507. Oct. 1990.
- [62] E. W. Weisstein. *CRC concise encyclopedia of mathematics*. Boca Raton, Fla. : CRC Press, 1999.