On the Cost–Effectiveness and Realization of the Theoretical PRAM Model

SFB Report 09/1991

Ferri Abolhassan, Jörg Keller, Wolfgang J. Paul

Revised and extended Version
of SFB Report 21/1990

Sonderforschungsbereich 124
Fachbereich 14
Universität des Saarlandes
W–6600 Saarbrücken, Germany

Abstract

Todays parallel computers provide good support for problems that can be easily embedded on the machines' topologies with regular and sparse communication patterns. But they show poor performance on problems that do not satisfy these conditions. A general purpose parallel computer should guarantee good performance on most parallelizable problems and should allow users to program without special knowledge about the underlying architecture. Access to memory cells should be fast for local and non local cells and should not depend on the access pattern. A theoretical model that reaches this goal is the PRAM. But it was thought to be very expensive in terms of constant factors.

Our goal is to show that the PRAM is a realistic approach for a general purpose architecture for any class of algorithms. To do that we sketch a measure of cost–effectiveness that allows to determine constant factors in costs and speed of machines. This measure is based on the price/performance ratio and can be computed by hand without building a machine. The smaller the measure M, the better the machine. Studying theoretical literature gives RANADE's Fluent Machine to be the best PRAM simulation. We evaluate the Fluent Machine with our measure and present changes to the machine that bring an improvement of the measure by a factor 5.

We show that for almost all PRAM algorithms that use concurrent access it is better to run them on a CRCW PRAM instead of an EREW PRAM. We sketch an architecture of a distributed memory machine (DMM) that reflects our view of existing parallel machines. To compare PRAMs and DMMs we construct worst case situations. This leads to upper and lower bounds $U = O(\log n)$ and $L = \Omega(1)$ for the term M(PRAM)/M(DMM) as one would expect. The astonishing fact is that the constant factor in U is quite small, the one in $1/L$ however is quite large. We conclude that for reasonable values of n a PRAM cannot be much worse than a DMM but vice versa. We show by examples that both bounds are tight.

In the last part of the article we describe a special feature of our PRAM — the multiprefix operation — and its realization. We show when it is useful to support multiprefix by hardware. Furthermore we sketch a possible prototype of our machine.

Keywords: Delayed `LOAD`, Hashing, Parallel Architecture, Pipelining, PRAM Emulation, Routing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Commercially available parallel machines can be classified as *distributed memory machines* or *shared memory machines*. Exchange of data between different processors is done in the first class of machines by explicit *message passing*. In the second class programs on different processors simply access variables in a *common address space*. Thus one gets a more comfortable programming model.

One is tempted to suspect big differences between the hardware architectures of the two classes, but this is actually not so. Processors of present shared memory machines tend to have local memories as well as large caches, and the exchange of cache lines between processors can be viewed as an automated way of message passing. As a consequence of this implementation one gets a very considerable gap between the access times for local and non local variables.

It would be highly desirable to make this gap disappear in such a way that local accesses remain their speed and such that remote accesses become as fast as local accesses. This almost looks like asking too much, but in fact this is a well studied problem in theoretical computer science, and there is a highly nontrivial body of potentially useful results. In a nutshell the results say that the desired machine, which in the theory world is called a **PRAM**, can be built, but building a PRAM with p processors costs $O(1)$ times more than building a distributed memory machine with p processors.

This paper is motivated by the simple question: what is the constant in $O(1)$?

1.1 Overview

If we are asking for cost we can mean cost in a theoretical model which permits to deal with constant factors or we can mean cost measured in Dollars for building real machines. We deal with the question in both ways. In both cases one has to start by surveying the relevant theoretical literature, identifying the constants which are hidden in the O–Notation, and determining those which can be determined without a detailed model. This is done in chapters 1.2 to 3.5.

It turns out that the crucial design decisions concern three things:

1. design of the node procesor.
2. choice of a network connecting processors with memory banks and a routing scheme for the network.
3. choice of a hashing scheme which tries to distribute memory accesses evenly over the memory banks.

In chapter 2 and chapter 3 we describe a combination of good solutions to 2 and 3 proposed in [11, 10, 30, 31, 32, 35, 41, 42, 46, 50, 51] (processor design is ignored in the theoretical literature).

If one does not insist in having proofs of correctness for all components of the machine but instead one is willing to rely on the outcome of simulations then one can shave some constant factors off the machine's cost. The corresponding simulation software and the results of the simulations are described in section 6.3.

After this point we have to consider detailed cost measures.

On the theoretical side we base our investigations on a formal framework proposed in [37]. There an architecture is viewed as characterized by 4 things:

1. a high level language
2. a machine language
3. hardware capable of interpreting the machine language
4. a compiler translating from the high level language to the low level language

A hardware model is proposed which generalizes the usual model of switching circuits but which allows for certain parameters of technology, e.g. the density of packing cells in a SRAM or the access time to main memory. The model for workload is an explicit benchmark program. Architectures are evaluated on the basis of their cost measured in gate equivalents and the time (measured in gate delays) to execute the benchmark program.

In chapter 4 we describe the hardware model from [37] in as much detailed as needed here. We add to the design from [42] a 32 bit RISC processor in the spirit of [39] and obtain a reference design $D0$. We sketch how to determine the cost c_{D0} of this design and state the result.

In chapter 6 we make a slight modification in the routing scheme and we change the processor design in such a way that each physical processor simulates say x virtual processors, where x is related to the latency of the network. The resulting design $D1$ is somewhat more complicated than $D0$ but the product of cost c_{D1} and time t_{D1} is by a factor of 5 better than for $D0$. In [42] there was of course no point to go after such improvements, because asymptotically $D1$ is not better than $D0$.

We study the question whether it is worth to realize combining [17] and multiprefix [17, 42] in hardware in chapters 8 and 10. The results are in favour of hardware support.

In chapter 9 we introduce a model of distributed memory machines that allows to give the interconnection of processors as a parameter. We restrict this model to use only neighbour–to–neighbour communications. We construct situations that are worst case for a PRAM or a DMM. First we do not use global memory at all. This happens for example in matrix multiplication. Second we are looking for a problem where the best known algorithm for a DMM is less cost–effective than a simulation of a PRAM on a DMM. This happens in computing the connected components of an undirected graph. These situations lead to lower and upper bounds.

In chapter 11 we sketch how a physical realization would look, if we had to freeze our designs today.

1.2 The PRAM Model

1.2.1 Definitions and Notations

Definition 1 *A* **processor** *in this chapter is a sequential register machine as described in [2].*

Definition 2 *The* n**–PRAM (parallel random access machine)** *is a parallel register machine with* n *processors* $P_0, \ldots, P_{n-1}$*, their local memories and a shared memory of size* m*. In each step each processor can work as a seperate register machine or can access a cell of the shared memory. The processors are working synchronously, one step takes unit time.*

The following example shows that such a model can shorten runtime:

Given are n variables $X_0, \ldots, X_{n-1}, X_i \in \{0, 1\}$, compute function OR_n with

$OR_n(X_0, \ldots, X_{n-1}) = 1$ iff. $\exists i \in \{0, \ldots, n-1\}$ with $X_i = 1$.

solution 1: Calculate on a tree network in which each inner node performs the operation OR_2 with data sent from its sons. The root gets the result. This solution has runtime $O(\log n)$.

solution 2: Use a $n+1$–PRAM, let processor P_i access variable X_i. Processor P_n gets the result as variable *Result*. The algorithm is as follows (we use the **pardo** construct as described in [16] to describe the algorithm):

for all processors P_i **pardo**
  if $X_i = 1$ **then** $Result := 1$
od.

This solution has constant runtime.

This simple example suggests that there are several PRAM models, depending on the access modes to shared memeory that are allowed. In the examples all processors $P_0, \ldots, P_{n-1}$ could simultaneously write to the cell that holds *Result*. Thus for this example we need a *concurrent write mode*. Thus Read or Write conflicts define the following models:

- **EREW:** *(exclusive read exclusive write)* It is not possible to read or write a memory cell simultaneously with several processors.
- **CREW:** *(concurrent read exclusive write)* It is only possible to read a cell simultaneously.
- **CRCW:** *(concurrent read concurrent write)* Processors can read and write a cell simultaneously. Concurrent write forces to define which one of the concurrent processors will win. Usually three possibilities are studied:
  - **arbitrary:** One processor wins, but it is not known in advance which one wins.
  - **common:** All processors must write identical data, thus it does not matter which one wins.
  - **priority:** The processor with the largest or lowest index wins.

The last model is the most powerful. Many algorithms however are designed for the EREW model. Overviews about algorithms for the different models can be found in [4, 16, 26]. We do not consider owner read and owner write accesses as defined by [12].

A CRCW PRAM can be simulated on a EREW PRAM with delay $O(\log n)$ [26]. The loss of speed however cannot be ignored even for small numbers of processors. If we have a PRAM with 256 processors, the slowdown factor is $8c$ where c is a constant factor that originates in the simulation overhead.

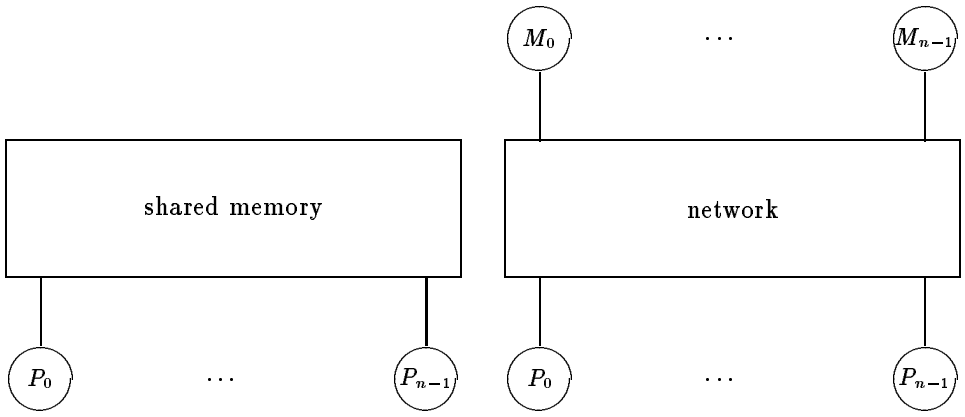


Figure 1.1: PRAM – Model and Realization

1.2.2 Simulation of a PRAM

The central part of a PRAM is a main memory that allows simultaneous access by several processors. Memories that allow simultaneous read access are called *multi port RAMs*. They are only commercialy available for small numbers of ports. Optoelectronic tries to transmit data by laser light. The laser beams can be received by several sensors or processors. This method is close to our CREW model but development is in a very early state and not ready for use.

A realization of a PRAM therefore will have to be a simulation.

We simulate a n–PRAM on a multi computer machine (MIMD) by distributing the main memory to memory modules $M_0, \ldots, M_{n-1}$. Each module M_i is close to processor P_i. The processors are connected by an interconnection network (see figure 1.1). Processor P_i communicates with processor P_j on this network if P_i wants to access a variable that is stored in M_j $(i \neq j)$. The communication consists of sending a packet from P_i to P_j, in which P_i specifies which variable it wants to access.

The simulation now consists of two steps:

1. Distribution of the used variables $V_0, \ldots, V_{r-1}, r \leq m$ (let $m = n^k$ be the total size of the PRAM memory) to memory modules $M_0, \ldots, M_{n-1}$ by a hash function,
2. Sending the request and answer packets through the interconnection network.

An index for the quality of the two steps is the time they need and the hardware costs they have. In the next chapters we will review known solutions for the above problems.

Chapter 2

Hashing

2.1 General Facts

One wants to map the used variables $V_0, \ldots, V_{r-1}$ onto the shared memory of size m. To do this one uses a hash function $g : \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$. The shared memory is distributed uniformly among the n memory modules $M_0, \ldots, M_{n-1}$ with size $\frac{m}{n}$ each. Therefore we think of the hash function g as a tuple (h, l) of functions $h : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$ and $l : \{0, \ldots, m-1\} \to \{0, \ldots, \frac{m}{n}-1\}$. The function h specifies the module, l specifies the location within the module. One gets h and l from g in the following way: $h(x) = g(x) \bmod n$, $l(x) = g(x) \quad div \quad n$. If one regards the binary representation of g(x) with length $\log m$, the lower $\log n$ bits give the module $h(x)$, the upper $\log m - \log n$ bits give the location $l(x)$ within the module. The binary representation of $g(x)$ looks as shown in figure 2.1.

As we will show in chapter 3 the function h influences the behaviour of the routing very strongly. Therefore one is interested in functions h that distribute variables among the memory modules $M_0, \ldots, M_{n-1}$ so that the routing of packets has as few conflicts as possible. If we have chosen a certain hash function there are always examples of data which are distributed poorly. Therefore one considers universal hash functions. This are families of hash functions that distribute data well with high probability. The meaning of 'good' in this sense is made precise in the following definition:

Definition 3 *A family $\mathcal{F}$ of hash functions $f : D \to R$ is called s_μ-wise independent, if $\forall y_0, \ldots, y_{s-1} \in R, x_0, \ldots, x_{s-1} \in D$ with $x_i \neq x_j$ for $0 \leq i < j \leq s-1$ holds*

$$|\{f \in \mathcal{F} : f(x_i) = y_i, i = 0, 1, \ldots, s-1\}| \leq \mu \frac{|\mathcal{F}|}{|R|^s}$$

This informaly means that there are only a few 'bad functions' in the following sense:

Suppose for $0 \leq i < j \leq s-1$ processor P_i wants to access address $x_i \in \{0, \ldots, m-1\}$ and all x_i are hashed by f to the same module M_y with $y \in \{0, \ldots, n-1\}$. Then access to module M_y will be sequential and access time will be $O(s)$. If f is chosen randomly with uniform distribution from an s_μ-wise independent family of hash functions, then for each y the probability that $f(x_0) = \ldots = f(x_{s-1}) = y$ is $\mu |R|^{1-s}$ with $|R| = n$.

The routing algorithms in chapter 3 demand $s = \log n$-wise independent hash functions.

The following resulting questions will be answered in the sections 2.2 to 2.5:

1. How can we find such s_μ-wise independent families $\mathcal{F}$ of hash functions?

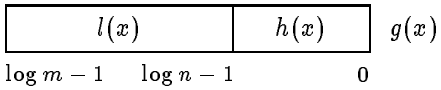


Figure 2.1: Binary Representation of Hash function $g(x)$

2. Which is the cheapest one in the sense of storage space and evaluation time?
3. Is the chosen function g bijective?
4. Does the routing need such a strong definition of hash functions?

2.2 Construction of Hash Functions

MEHLHORN and VISHKIN proved in [35] the existence of $\log n$–wise independent hash functions by constructing polynomials of degree $\log m = k \log n$.

Definition 4 *A function $pol(x)$ of the following family of functions*

$$\mathcal{H} = \left\{ pol : pol(x) = \left(\left(\sum_{i=0}^{d} a_i x^i \right) \bmod P \right) \bmod n, a_i \in Z_P \right\}$$

is called hash function of degree d. P is a prime not smaller than m, $d = k \log n$.

We choose a particular function *pol* of this family $\mathcal{H}$ by choosing the coefficients a_i randomly. This is done by a pseudo random number generator at the beginning of the run of each PRAM program. Because each processor must be able to evaluate the 'hash address', the coefficients must be stored in the local memory of each processor. Two important parameters of hash functions are the space needed to store the random words (in the case of the polynomial the coefficients) and the evaluation time. In the case of polynomials the required space has size $O((\log m)^2)$, the evaluation time is $O(\log m)$ (if one can multiply or perform division mod$P$ mod n in one unit of time).

If one wants to choose h out of $\mathcal{H}$ then g has to be constructed as

$g(x) = \left(\left(\sum_{i=0}^{d} a_i x^i \right) \bmod P \right) \bmod m.$

ALAN SIEGEL presents in [46] an s_μ–wise independent family of hash functions, which can be evaluated in time almost $O(1)$. Unfortunately there is a large trade off between space and time. For the construction of his hash functions SIEGEL uses a bipartite graph in which he stores the random words and which he calls *weak concentrator (w.C.)*. The degree d of this w.C. depends on $k = \log_n m$ ($d = 2 + k/\epsilon$ for any fixed $\epsilon < 1$). As shown in table 2.1 the evaluation time is $O(d^2)$ and therefore not really a constant. Even this 'constant' evaluation time is very large. We show this in table 2.1 where we compare polynomial hash functions with two versions of SIEGEL's families of hash functions with $s = \log n$ and $\mu = 1$. Not only the constant of the evaluation time in SIEGEL's family of function is very large, also the necessary w.C. is unreallistically big. Another disadvantage of this method (we call it Siegel1) is the fact, that there is no constructive way to find such a w.C.. SIEGEL gives only a proof of the existence. For this reason he gives a method (Siegel2) to construct a big w.C. from smaller ones — which more easily can be tested to be a w.C. — by building the cartesian product of the smaller w.C. with itself. By doing this the degree of the resulting w.C. is growing to $d^{k/\epsilon}$. The evaluation time of Siegel2 grows in an unacceptable way. As the table shows the trade off

	$\mathcal{H}$	Siegel1	Siegel2
evaluation time	$\log m$ Add $\log m$ Mult	d^2 Add d Mult $d = 2 + \frac{k}{\epsilon}$	$d'^2 = d^{\frac{2k}{\epsilon}}$ Add $d^{\frac{k}{\epsilon}}$ Mult $d' = d^{\frac{k}{\epsilon}}$
space (number of random words)	$\log m$	dn^{ϵ}	$dn^{\frac{\epsilon}{k}}$
example $m = 2^{25}, n = 32$, hence $k = 5$	 25 Add 25 Mult	$d \approx 8$ 64 Add 8 Mult	$d' \approx 105000$ 1.102510^{10} Add 105000 Mult
size of w.C.		2^{25} nodes	32 nodes

Table2.1: Comparison of Hash Functions

between space and time can not be avoided in SIEGEL's methods. The resulting constants are not for practical use.

2.3 Bijektivity and Modulo Division

The function g is not necessarily bijective, e.g. if chosen as described in section 2.2. If more than one variable is hashed into one single location one can use lists of variables. The disadvantage of this method is the increased access time and the increased memory size. In the worst case, which is unrealistic, one needs memory of size m^2 instead of m, because every variable could be hashed into the same cell. A good solution for such a scheme is given in [42]. RANADE needs modules of size $O(\frac{m}{n})$ but gives no exact constant. The access time is increased by a constant number of steps. A second possibility is to rehash as described by DIETZFELBINGER and MEYER AUF DER HEIDE in [10].

Another very important point for the choice of the hash function is division with rest. A division circuit takes too much time and is very expensive in comparison to adders and multipliers. Therefore one tries to avoid such circuits.

Let us consider the polynomials again:

$$pol(x) = \left(\sum_{i=0}^{d} a_i x^i \right) \bmod P, \quad \text{mit} \quad a_0, \ldots, a_d \in F.$$

The modulo computation should be executed in a very simple way; not by a separate division circuit but by simple shift or simple logic operation (e.g. exor). Therefore one needs special primes P, e.g. of the form $2^n - 1$. Good solutions for avoiding division circuits and tables of such primes are given in [24, 44].

2.4 A Practical Hash Function

For the realization we intend to use a linear hash function g of the type $g(x) = ax \bmod m$ with greatest common divisor $\gcd(a, m)=1$, $0 \leq a \leq m - 1$. This function gives good simulation results [42, 53] for many applications although is is not $\log n$–wise independent. DIETZFELBINGER showed

in [9] that for $m = 2^u$ for any integer u this function is in the sense of definition 3 with $s = 2$ as good as $g'(x) = ax + b \bmod P \bmod m$ where P is a prime. The advantages of the function $g(x)$ are the bijectivity, the small amount of space used for random words (only a and b) and the small evaluation time. Furthermore the division modulo m with $m = 2^u$ can be computed by taking the lower $\log m$ bits of ax ignoring the upper bits. Thus we do not need any divisions when evaluating the hash function.

2.5 Open Problems

The reason why we are interested in hash functions which satisfy definition 3 stems from the second randomized routing scheme described in chapter 3. If one uses there a network with diameter $O(\log n)$ and a $\log n$–wise independent hash function, the routing with high propability needs time $O(\log n)$. An interesting question in this context is 'Which properties of hash functions are **necessary** in order to have a routing time of $O(\log n)$ with high probability?' This question is an open problem in the theory of hashing. In order to deal with this problem empirically we have developed a simulator that works with several routing algorithms and hash functions. It is based on Alf Wachsmann's work [53].

Chapter 3

Networks and Routing Algorithms

3.1 General Facts

The decisions for choosing an interconnection network are made between:

- operation modes (synchronous/asynchronous)
- control strategies (centralized/distributed control)
- switching methologies (circuit/packet switching)
- network topologies

A good classification of networks with respect to these four points is given in [23]. There these points are described as follows:

Two types of communication can be identified: *synchronous* and *asynchronous*. One speaks of synchronous communication if all processors send out requests at the same time. One speaks of asynchronous communication if requests are sent out at arbitrary points and are handled dynamically.

A typical interconnection network consists of a number of switching elements and interconnection links. Later on switching elements will also be called network nodes. Interconnection functions are realized by properly setting control of the switching elements. The control–setting function can be managed by a centralized controller or by the individual switching element. The latter strategy is called *distributed control* and the first strategy corresponds to *centralized control*. If using centralized control a phase for precomputing the control–setting function is necessary. This is useful for communications which use very often the same control–setting function. Distributed control is useful for communications that often change the control–setting function.

The two major switching methodologies are *circuit switching* and *packet switching*. In circuit switching, a physical path is actually established between a source and a destination. In packet switching, data is put in a packet and routed trough the interconnection network without establishing a physical connection path. In general, circuit switching is much more suitable for bulk data transmission, and packet switching is more efficient for many short messages. Packet switched networks have been suggested mainly for MIMD machines.

A network can be depicted by a graph in which nodes represent switching elements and edges represent communication links. The topologies tend to be regular and can be grouped into two categories: static and dynamic. In a static topology, links between two processors are passive and

dedicated buses cannot be reconfigured for direct connections to other processors. On the other hand, links in the dynamic category can be reconfigured by setting the network's active switching elements.

3.2 Requirements for a PRAM Network

In order to choose the best suitable interconnection network for a PRAM we first have to tell something about the types of network access.

Suppose variables $V_0, \ldots, V_{r-1}, r \leq m$ are mapped to the n memory modules by a hash function $h : \{0, \ldots, m-1\} \rightarrow \{0, \ldots, n-1\}$. During a single PRAM step processor P_j could execute a command `STORE` V_i or `LOAD` V_i. In both cases it has to communicate with the memory module on which V_i is stored. This is $M_{h(i)}$. In the first case P_j sends a new value a together with the location $l(i)$ to $M_{h(i)}$ which replaces the old value of V_i by a. In the second case P_j sends a request to memory module $M_{h(i)}$ to send back the value of V_i. When the memory module $M_{h(i)}$ has received this request and has read location $l(i)$ it has to send back a packet with the value of V_i. This can happen on the same network with a new routing by exchanging source and destination. Another way is to use a seperate 'backward network' which uses the information of the 'first routing'.

We decide to take a synchronous network because our machine is defined to work synchronously (see definition 2). The control has to be distributed because normally in each PRAM step other variables are accessed and thus a new control–setting function (see 3.1) is needed. The consequence for our routing algorithm is to use *local routing* as defined in 3.4. Because accessing variables requires a large number of small packets, we will use packet switching. We still have to choose a topology with the following properties:

- support of simple packet routing,
- small diameter,
- slowly growing or even constant degree of the network nodes
- routing algorithms with little delay time, e.g. $O(\log n)$, available.

3.3 Choice of a Network

A topology that fulfils the properties above is the n–dimensional cube (*Hypercube*) defined as follows

Definition 5 *The* n–dimensional cube *network consists of* 2^n *nodes. A node* i *is connected to all nodes* j *with hammingdistance*$(bin(i), bin(j)) = 1$. $bin(i)$ *here is the binary representation of* i.

Its advantage is that other networks can be simulated on it. There exist many algorithms for routing on the hypercube ([7, 52]) and many theorems about networks have been proved for it ([51]). Its disadvantage for practical use is the degree $\log n$ of its nodes. A network node designed for a network realization of size n cannot be used for a realization of any other size. Because 'scalability' is an important aspect in constructing real connection networks, one tries to use one design for several network sizes. Therefore one prefers networks with constant degree such as the class of so called *Delta–Networks*. Wu and Feng showed in [55] the topological eqivalence of all networks in this class.

A typical representant of this class is the *butterfly network* (also called the FFT network) that is defined as follows:

Definition 6 *The* butterfly network *of degree 2 consists of* $n(1+\log n)$ *network nodes. Each node is assigned a unique number* $\langle col, row\rangle$ *where* $0 \leq col \leq \log n, 0 \leq row \leq n-1$. $\langle col, row\rangle$ *can be viewed as the concatenation of the binary representations of col and row. Node* $\langle col, row\rangle$, $col < \log n$ *is connected to node* $\langle col+1, row\rangle$ *and to node* $\langle col+1, row \oplus 2^{col}\rangle$, *where* $\oplus$ *denotates the bitwise exclusive or of the binary representations of the two arguments. The result is the number represented by the resulting bitstring. The butterfly network has diameter* $\log n$.

We use the notations $\mathbf{column}\langle col, row\rangle = col$, $\mathbf{row}\langle col, row\rangle = row$, $\mathbf{row\ leader}\langle row\rangle = \langle 0, row\rangle$.

3.4 Packet Routing

We now have to choose a routing algorithm that fits best the chosen network and is suitable for simulating a PRAM. We denote by pac_i the packet originating at node i $(0 \leq i \leq N-1)$ which is called the source. N denotes the number of network nodes. We denote by $d_i, 0 \leq d_i \leq N-1$ the destination of pac_i. We denote by N–packet routing the problem to transmit N packets from their sources to their destinations via an interconnection network.

The first decision is whether to use a deterministic or a randomized routing algorithm. In deterministic routing algorithms the path of packet pac_i from source i to destination d_i only depends on i and d_i. In randomized routing algorithms this path additionaly can depend on some random bits.

Investigating in deterministic routing LEIGHTON constructs in [28] a degree 3 N–node network capable of solving any N–packet routing in $O(\log N)$ steps. Although this result is optimal up to constant factors, the constant factors are quite large and the algorithm is of no practical use. UPFAL too presents in [49] an optimal deterministic algorithm, but there too the constant factor is large (about 25000, see [32]). Citing [30] the best small–constant–factor deterministic routing algorithm in an $O(\log^2 N/\log\log N)$–step algorithm on the butterfly.

In contrast to that there exist several randomized routing algorithms with expected runtime $O(\log N)$ where the constant factors are smaller than those of the deterministic algorithms. A detailed comparison of deterministic and randomized routing algorithms can be found in [32]. From now on we will only consider randomized algorithms. We first describe the kind of randomization in routing algorithms and then we give a short overview of the development and results of randomized routing algorithms in table 3.1.

There are two main kinds of randomization in routing algarithms. The first is called VALIANT*'s paradigm* [52]. His routing scheme simply consists of two phases: each packet pac_i is sent first to a random intermediate destination id_i and from there on to its final destination d_i (see figure 3.1, left part). The phases themselves are deterministic. The number of random bits for N–packet routing is $N \log N$. Each of the N packets needs $\log N$ bits for the binary representation of the intermediate destination.

Another way to randomize is by hashing the destinations d_i of the N packets pac_i with a hash function f randomly chosen from an s_μ–wise independent family $\mathcal{F}$ of hash functions with appropriate s and μ (see figure 3.1, right part). In this case the number of random bits is equal to the number of bits for random words. In the family $\mathcal{H}$ of polynomials $k^2 \log^2 N$ bits are needed. For SIEGEL's hash functions one needs $k(2+k/\epsilon)N^\epsilon \log N$ random bits. (Remember: $k = \log m/\log N$).

Randomized algorithms need a random number generator to produce the random bits. In practice one will use a pseudo number generator. KARLOFF and RAGHAVAN show in [25] how to construct a pseudo number generator that performs well for routing.

We now give a short summary of the development of randomized routing algorithms working with distributed control. All relevant facts are shown in table 3.1. The first three algorithms have the disadvantage of using queues of size $O(\log N)$. Queue here means that a number of packets can be

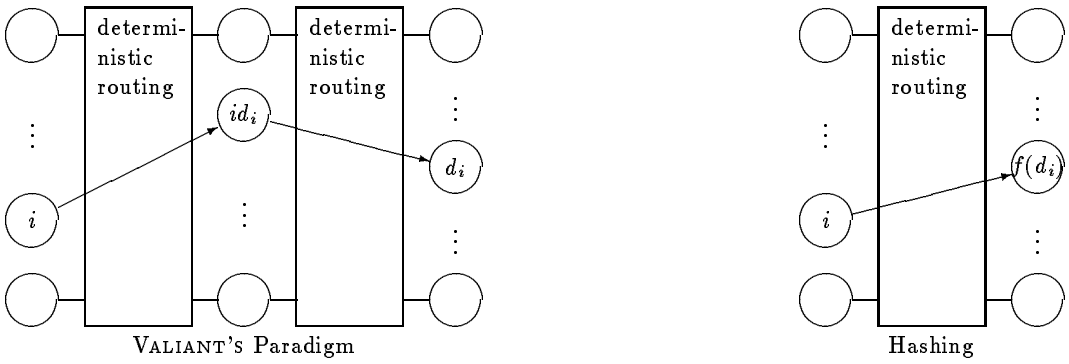


Figure 3.1: Methods of Randomization

stored in a FIFO buffer at the input of a network node. PIPPENGER's algorithm needs only queues of size $O(1)$. But if it fails to end in expected time $O(\log N)$ it runs in a deadlock.

The best algorithms are from RANADE and LEIGHTON, MAGGS and RAO, because they have constant size queues which is necessary for scalability. The arguments are exactly the same as for constant degree network nodes in section 3.3.

RANADE's algorithm uses the second method of randomization. The algorithm of LEIGHTON, MAGGS and RAO is a generalization of RANADE's algorithm with a new analysis. Table 3.1 contains some constants not specified in the original papers. They stem from an extended analysis in [32]. The runtime constants in table 3.1 are computed for a butterfly network. If one compares queue size and runtime of the two algorithms with identical probabilities, RANADE has larger queues but a little bit smaller runtime. RANADE proves the runtime for arbitrary requests, LEIGHTON, MAGGS and RAO only consider permutations. Simulations have shown much better behaviour of the algorithms than the theoretical results. RANADE reports in [42] a delay time of $11 \log N$ with buffer size 3. LEIGHTON, MAGGS and RAO have similar results in [31]. This is not surprising. Let us consider a dull algorithm and a worst case situation in the butterfly network like the following:

If a network node wants to transmit a packet and has to choose between several packets, it simply takes the packet with the smaller destination or the left when the destinations of the 2 incoming packets are identical. Each row leader want to send a packet to a node x with $\mathbf{column}(x) = \log N$. The first two packets arrive there after $\log N$ steps, in each following step 2 packets arrive. The routing takes time $\frac{N-2}{2} + \log N$.

If we compare this algorithm with that of RANADE with queue size $b = 2$, the dull algorithm is better for $N < 2^{15}$. This does not indicate that RANADE's algorithm is worse. It shows that the analysis is not tight enough for 'small' N.

Nevertheless we consider RANADE's algorithm most suitable for a butterfly network to simulate a PRAM. We will describe his algorithm in detail in chapter 3.5.

In the next chapters we will use a synchronous packet switching butterfly network with distributed control and the randomized packet routing algorithm of RANADE. The exact queue size and runtime will be figured out by simulations in chapter 6.3.

Algorithm	Year	Network	Runtime	Probability	Queue	Remark
VALIANT, BREBNER	81, [52]	Hypercube	$O(\log N)$	$1-\frac{1}{N}$	$O(\log N)$	any perm.
ALELIUNAS	82, [5]	Shuffle–exchange	$O(\log N)$	$1-\frac{1}{N}$	$O(\log N)$	any perm.
UPFAL	82, [48]	Butterfly	$O(\log N)$	$1-\frac{1}{N}$	$O(\log N)$	any perm.
PIPPENGER	84, [40]	Butterfly	$O(\log N)$	high Probability	$O(1)$	any perm., dead-locks possible
RANADE	87, [41]	Butterfly	$b \log N$	$1-(N \log N)^{-4 \log \frac{b-6}{96}}$	b	any Requests
LEIGHTON, MAGGS, RAO	88, [30]	constant degree networks	$2^{\frac{12}{b}+3} 5e^2 \log N$	$1-N^{-8}$	b	any perm.

Queue here means the maximum number of packets that can be stored in an input queue of a network node. In the column 'Probability' we list upper bounds on the probability that the algorithm succeeds in routing all packets in the given runtime.

Table3.1: Comparison of Routing Algorithms

3.5 Relationship between Routing and Hashing

In the previous section we mentioned that a $\log N$–wise independent family of hash functions allows to use the second method of randomization in routing algorithms. In this section we want to explain the relationship between routing algorithms and hash functions. We need the following definitions:

Definition 7 Module congestion c_m *is the maximum number of packets* pac_i *to be sent to one destination* d_i. Edge congestion c_e *is the maximum number of packets* pac_i *that use a certain edge in a routing phase.* Network diameter *dia is the length of the longest path in a network.*

LEIGHTON, MAGGS and RAO prove in [30] the following theorem 1.

Theorem 1 *On a bounded degree network with diameter dia and constant queue size N packets can be sent to their destinations in time* $O(c_e + dia + \log N)$ *with high probability.*

The underlying random experiment for this and the two next theorems is the following: Given a network as described in the theorem and N packet to be sent to their destinations, set the random bits, send the packets once and measure the number of steps.

Here we search for the probability of the following event: All packets have reached their destinations in the time given in the theorem.

A PRAM however only sets the random bits once by choosing the hash function and then does many routings of N packets. This problem cannot be solved by the known routing algorithms. One possibility would be to rehash as described in section 2.3.

To reach time $O(\log N)$ in theorem 1 *dia* and c_e have to be $O(\log N)$. HAGERUP proves in [19] the following theorem.

Theorem 2 *If the module congestion is* $O(\log N)$ *then with high probability the edge congestion is* $O(\log N)$ *as well.*

The exact analysis of this theorem (see [32]) shows for module congestion $c_m = x \log N, x$ any integer, edge congestion $c_e \leq 3x \log N + (\alpha + 1) \log N$ with probability at least $1 - N^{-\alpha}$.

MEHLHORN and VISHKIN show in [35] that the module congestion is $O(\log N)$ if one uses a $\log N$–wise independent family of hash functions to map the variables $V_0, \ldots, V_{r-1}$ to the destinations namely the modules $M_0, \ldots, M_{n-1}$.

RANADE has a similar result in [41] when analysing his routing algorithm as shown in the following theorem.

Theorem 3 *Using a butterfly network and the family $\mathcal{H}$ of poynomial hash functions N packets can be sent to their destinations in time $O(\log N)$ with high probability.*

As we mentioned in section 2.5 is it still an open problem if the condition of a $\log N$-wise independent hash function is necessary.

achapterThe Fluent Machine

3.6 Introduction

All routing algorithms considered in chapter 3 beside the one of RANADE only consider N–permutations as communications. In a PRAM however a communication does not have to be a permutation. Several variables accessed at the same time could be located on the same module. In a CRCW PRAM the situation is even more complicated. Several (possibly all) processors could access the same variable V_j at the same time. Let $P = \{p_0, \ldots, p_{n-1}\}$ be the set of processors in a PRAM and

$$S_j = \{p_i \in P | p_i \text{ reads } V_j \text{ in the current step}\},$$

$$PAC_j = \{pac_i | p_i \in S_j \text{ sends } pac_i \text{ into the network}\}.$$

We talk only of `READ` accesses because `WRITE` accesses can be treated in a similar way with the simplification that they do not need an answer back to the processor.

If all packets in PAC_j reach memory module $h(j)$, the module congestion $c_m = |PAC_j|$. In the worst case this could be n. Because the routing algorithms require module congestion $O(\log n)$ the number of packets in PAC_j that reach $h(j)$ has to be reduced in the following way.

The paths of the packets in PAC_j form a tree. However there is no need to send more than one packet along any branch of this tree. If a packet $pac_i \in PAC_j$ simply waits at each tree node until a packet $pac_l \in PAC_j$ appears along the other incoming edge (unless the node 'knows' that all packets coming in the future must originate from processors $p \notin S_j$), then the two packets can be merged and one forwarded along the tree. Later on this merging is called *combining*.

In order to decide whether two incoming packets pac_i, pac_l that access variables V_x, V_y have to be combined, a network node has to compare the destinations $g(x)$ and $g(y)$. If the hash function g is not bijective (see 2.3) the packets additionaly have to contain x and y which then have to be compared.

How can a network node know that no more packets $pac_i \in PAC_j$ will arrive in the future? RANADE's [41] main idea is to keep the packets leaving a network node sorted by their destinations. In the next section we will describe RANADE's routing algorithm in detail and there we will in some detail present RANADE's argument that the above idea is sufficient to guarantee that combining works.

3.7 The Routing Algorithm

RANADE uses a butterfly network. The nodes have indegree and outdegree 2, the inputs are called

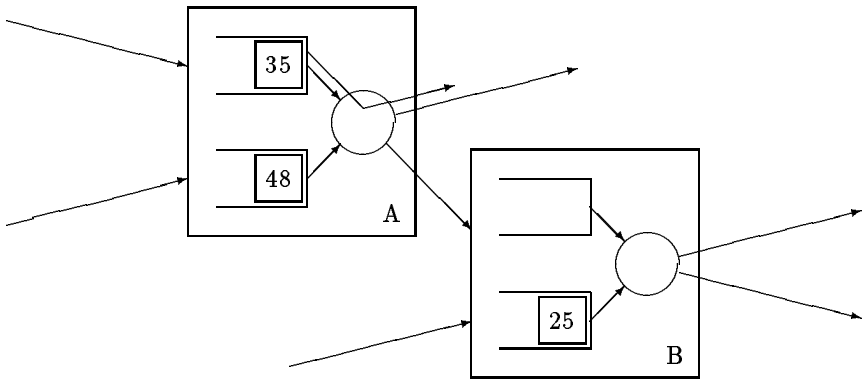


Figure 3.2: Function of Ghost Packets

in_0, in_1 the outputs are called out_0, out_1. The inputs contain FIFO queues of size b as buffers. Suppose a bijective hash function g is used. The case of a non bijective hash function is handled by secondary hashing (see section 2.3). If a processor p_i wants to access variable V_x it sends a packet $pac_i \in PAC_x$ into the network that consists of the variable's location $g(x)$, the mode $M \in \{\texttt{READ}, \texttt{WRITE}\}$ and a value d. If $M =$ `WRITE` d will be stored in $g(x)$, if $M =$ `READ` d will be replaced by the content of $g(x)$. As mentioned above the main idea of the algorithm is to keep the invariant that packets leave each network node in a sorted order by $g(x)$. If there are packets $pac_i \in PAC_x, pac_l \in PAC_y$ with $g(x) < g(y)$ in inputs in_0 and in_1 of a network node $\langle col, row\rangle$ pac_i is sent across one of the output edges, pac_l has to wait until the next decision which packet to take. This is a case of merging two sorted sequences to one. The merged sequence is sorted too (see [33]). If bit **column**$\langle col, row\rangle$ of $h(x)$ is zero pac_i is sent across out_0, otherwise it is sent across out_1. Remember figure 2.1 how $h(x)$ and $l(x)$ are represented in $g(x)$. The binary representation of $h(x)$ is only $\log n$ bits long but the butterfly network consists of $\log n + 1$ columns. Therefore in column $\log n$ all packets are sent across out_0. If the packets are put into the network in sorted order they will leave the network in sorted order too because each node maintains the order.

The decisions of the network nodes are sufficient to guarantee that each packet $pac_i \in PAC_x$ leaves the network at node $\langle \log n, h(x)\rangle$ and that combining works. This is formalized in Lemma 1.

Lemma 1 *Assume that in each row* row *of column zero a certain number ζ_{row} of packets sorted by their destinations are put into an n-butterfly network. The packets access γ variables $V_{i_0}, \ldots, V_{i_{\gamma-1}}$ and thus are organized in sets $PAC_{i_0}, \ldots, PAC_{i_{\gamma-1}}$ as defined above. Assume that all packets of a row belong to different sets.*

Claim:

1. *Each packet of PAC_{i_j} leaves the network at node $\langle \log n, h(i_j)\rangle$.*
2. *Packets, that leave a node $\langle col, row\rangle$ are sorted by their destinations.*
3. *Exactly one packet of each set PAC_{i_j} leaves the network in column* $\log n$.

The lemma can be proved by an easy induction over n.

The idea to keep packets sorted by their destinations has one deficiency. Let us consider figure 3.2. Network node B cannot transmit the packet with destination 25 because it must ensure that it will not receive a packet with a destination smaller than 25 in the future. When node A selects the packet with destination 35 for transmission across the upper output edge, it can forward this

information to node B by sending a packet with mode `GHOST` and destination 35 across the other output edge. When B receives the ghost packet, it knows that because of the sorted order all future packets that will arrive across that edge will have destinations larger than 35. Therefore one step later B can transmit the packet with destination 25.

If a node contains neither packets nor ghosts in his FIFO buffers, it sends an 'End of Stream' ghost (EOS) over both output edges. This is a ghost packet with a destination larger than m.

Figure 3.2 hints the fact that a ghost can only be effective if it reaches the head of an input queue immediately when it arrives. This only happens if this input queue is empty or if it contains one packet that is actually chosen to be transmitted. If the input queue contains more than one packet or one packet that has to wait the ghost can be destroyed immediately.

RANADE uses a seperate network to send back answers to packets with mode $M =$ `READ`. This *backward network* is identical to the first butterfly network except that inputs and outputs are exchanged. In the backward network no routing decisions are made but the packets are sent back in the same order in which they were sent in the first network. To do this there is a *direction queue* in each node of the first network in which the routing decision for `READ` packets are stored e.g. 'in_0 to out_1' or 'combine two packets, send result across out_0'. This queue has a width of 3 bits if we use compact coding. The length of the queue is $a \log n$ with $a \log n \leq c_e$. If we take the analysis of theorem 2 (see page 13) with $x = 1, \alpha = 10$ we get $a \leq 14$ with high probability. RANADE's simulations ([42]) have shown that $a = 11$ is sufficient.

Each node in the backward network is checking the top of the direction queue of the corresponding node in the first network and transmits the arriving packets in this order. By this the sorted order of the packets is maintained. The packets in the backward network have only to consist of the requested data. If no packet can be sent, a dummy packet is inserted that differs from a real backward packet only by an additional bit.

The routing over the backward network is not optimal, one can construct counter examples. But the realization is very simple, and this method supports a simple implementation of *parallel prefix* as described in chapter 10.

3.8 Description of the Fluent Machine

RANADE presents in [42] the Fluent Machine which is using the algorithm of section 3.7. The Fluent machine consists of $n \log n$ processors interconnected in the butterfly pattern (see Definition 6). There are $\log n$ columns with n network nodes. Each node contains one processor, one memory module and 2 network switches. The switches form 2 butterfly networks. For simpler descriptions we will talk of a logical network that consists of 6 butterfly networks put together as shown in figure 3.3. The machine operates in 6 phases:

1. Each processor sends a `READ` or a `WRITE` packet to its switch in the first network, if it is executing a `LOAD` or a `STORE` command. A switch $\langle col, row \rangle$ of the first network has two inputs, one from the processor and one from switch $\langle col - 1, row \rangle$, and one output to switch $\langle col + 1, row \rangle$. Each switch sends out one ghost packet with destination -1 which is smaller than any destination of the `READ` or `WRITE` packets. If the top of each input buffer contains a packet that is waiting to be transmitted the control of the switch selects always the packet with the smaller destination as described in section 3.7. Thus the ghosts are selected first to be transmitted. A `READ` or `WRITE` packet in column i can be selected only if all i ghosts that originate in the i columns before have passed. The $(i+1)^{st}$ packet to arrive in $\langle col, row \rangle$ is the one with the smallest destination among the packets in the columns $0, \ldots, i-1$ of row row. By an easy induction one can show that transmitting all $\log n$ packets of a row to the *row leader* of the second network in the manner described above sorts the packets by their destinations.

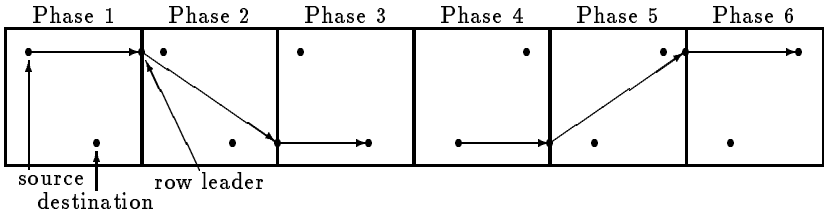


Figure 3.3: 6 Phase Routing of the Fluent Machine

2. The packets are routed through the second network by the algorithm of section 3.7.
3. The packets are shifted to the appropriate column and there to the memory module where the memory access happens. A switch in this phase has one input and two outputs, one to the memory module and one to the switch in the same row and next column.
4. The packets are sent from the memory module into the network and to the *row leader* of the 5^{th} network. The 4^{th} network is the backward network to the 3^{rd} network.
5. This is the backward network to the 2^{nd} network.
6. The packets are shifted to the nodes from where they were sent in phase 1. This is the backward network to the 1^{st} network.

Figure 3.3 shows an example for the 6 phase routing. The two physical networks serve for all phases. Network one serves for phases 1,3,5, network two serves for phases 2,4,6.

To examine the exact constants for runtime and costs in the Fluent Machine we have to introduce our model of valuating machines. Furthermore we have to model a processor for his machine because our valuation model requires to have one.

3.9 A RISC Processor

In the description of the Fluent Machine RANADE does not tell anything special about the intended processor. He only mentions that he intends to use RISC processors. The model of [37] requires to have a particular design of a machine in order to compute its costs and speed. We therefore now describe a processor design that fits for both the Fluent Machine and our proposed machine to be described in chapter 6. Furthermore we try to explain why the design of a PRAM processor almost necessarily leads to our or a similar design.

A natural explanation for RANADE to use a RISC instead of a CISC processor could be the required chip area. He intends to put 2 processors, a floating point unit, 128 Kbytes memory and the network switches belonging to the processors on a single chip. Chip area is significantly smaller for a RISC machine. The Berkeley RISC I microprocessor for example only consists of a 44,500–transistor integrated circuit [39].

But there are further design criteria for a PRAM processor that lead to RISC processors:

- All machine instructions should have similar execution times because of the synchronicity of PRAM processors (see Definition 2).
- All realised functions have to be computable directly, iterative methods are not possible because of the first item.

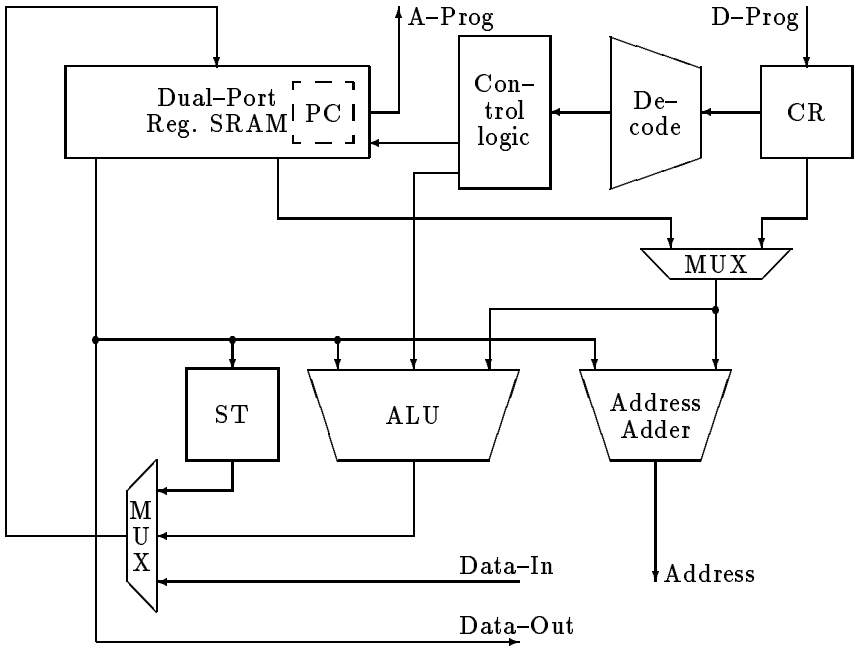


Figure 3.4: The Processor

For these reasons we decided to take a RISC architecture. The instruction set partly resembles the Berkeley RISC [39]. We take a `LOAD/STORE` architecture (memory access only takes place in `LOAD`, `STORE`, never in `JUMP`, `COMPUTE`) that separates the use of the two most time expensive parts of an execution, memory access and computation in the ALU. For the ALU we choose to realize addition, subtraction, multiplication, shifts and bitoriented functions. We choose a Harvard architecture in order to seperate data and instruction streams.

The processor has 32 registers each 32 bits wide. Registers 0 to 3 have special meanings: R_0 always is zero, R_1 is the program counter, R_2 and R_3 are local and global stack pointers. The processor also has a status register that involves beside the normal flags (zero, negative, overflow and carry), a counter flag, a modulo flag and a counter. The counter flag is set if and only if the counter is zero. The counter is decreased with each executed instruction; it can be preset. The modulo flag is set only in instructions with even numbers.

The modulo flag will be described in detail in chapter 10. It supports the special instructions `SYNC` and `MP` to simplify synchronization after parallel high level language constructs. The special instructions are described in chapter 10, too. In the next chapters we will ignore their existence. The counter flag and the counter serve for easy implementation of wait loops.

The only data type is a 32 bit word. Data and address paths are also 32 bits wide. This allows to use all registers as data and address registers. All problems that arise with multiple data widths disappear. An overview of the processor is shown in figure 3.4.

All instructions are coded in 32 bit words. The instruction format is uniform except for instruction LDHI. It is similar to that of PATTERSON and SEQUIN in [39]. The instruction set can be separated in `LOAD`, `STORE`, `COMPUTE`, `JUMP` and `STACK` instructions. Table 3.2 gives an overview.

Valid conditions in the `JMP` command in table 3.2 are *flagset* and *flagclear* where *flag* is an element of the following set $FLAGS = \{zero, negative, overflow, carry, modulo, counter\}$.

Remark: R_x means register x, $x = 0,\ldots,31$, $R_0 = 0$, $R_1 = PC$ =program counter, $R_2 = SP_0$ =global stack pointer, $R_3 = SP_1$ =local stack pointer, ST =status register, S_2 = 13 bit constant or register with number $S_2\langle 0:4\rangle$, Y = 19 bit constant. S(expression) means content of memory cell with address expression. Local and global memory access are distinguished by the most significant address bit.

`LOAD` instructions		
LD	R_x,S_2,R_d	R_d:= S$(R_x + S_2)$
LDHI	R_d, Y	$R_d\langle 13:31\rangle := Y$, $R_d\langle 0:12\rangle := 0$
GETPSW	R_d	$R_d := ST$
LDI	R_d	$R_d :=$ processor number
`MP` $\circ$	R_x,S_2,R_d	take part in `MP` $(R_x + S_2), \circ, R_d$, s. chapter 10
`STORE` instructions		
ST	R_x, S_2, R_m	$S(R_x + S_2) := R_m$
`SYNC` $\circ$	R_x, S_2, R_m	take part in `SYNC` $(R_x + S_2), \circ, R_d$, s. chapter 10
PUTPSW	R_m	$ST := R_m$
`COMPUTE` instructions		
ADD	R_x, S_2, R_d	$R_d := R_x + S_2$
ADC	R_x, S_2, R_d	$R_d := R_x + S_2$+carry
SUB	R_x, S_2, R_d	$R_d := R_x$-S_2
SBC	R_x, S_2, R_d	$R_d := R_x$-S_2-carry
AND	R_x, S_2, R_d	$R_d := R_x \wedge S_2$
OR	R_x, S_2, R_d	$R_d := R_x \vee S_2$
NAND	R_x, S_2, R_d	$R_d := R_x \overline{\wedge} S_2$
XOR	R_x, S_2, R_d	$R_d := R_x \oplus S_2$
ASL	R_x, S_2, R_d	$R_d := R_x$ shifted left arithm. by S_2 mod 32 bits
ASR	R_x, S_2, R_d	$R_d := R_x$ shifted right arithm. by S_2 mod 32 bits
LSL	R_x, S_2, R_d	$R_d := R_x$ shifted left logically by ...
LSR	R_x, S_2, R_d	$R_d := R_x$ shifted right logically by ...
WRL	R_x, S_2, R_d	$R_d := R_x$ rotated left by S_2 mod 32 bits without buffering in carry
WRR	R_x, S_2, R_d	$R_d := R_x$ rotated right by ...
XRL	R_x, S_2, R_d	$R_d := R_x$ rotated left by S_2 mod 32 bits with buffering in carry
XRR	R_x, S_2, R_d	$R_d := R_x$ rotated right by ...
MUL	R_x, S_2, R_d	$R_d := R_x \times S_2$
RM	R_x, R_d	$R_d := \lfloor \log \lvert R_x \rvert \rfloor$ if $R_x \neq 0$, -1 otherwise
`JUMP` and `STACK` instructions $(i \in \{0,1\})$		
JSRi	R_x, S_2	$S(SP_i + +) := PC$, $PC := R_x + S_2$
PSHi	R_x	$S(SP_i + +) := R_x$
POPi	R_x	$R_x := S(- - SP_i)$
JMP	Cond,R_x, S_2	if (Cond is satisfied) $PC := R_x + S_2$

Table3.2: Instruction Set

Other commands thought to be necessary can be defined as macro instructions. The most important ones are shown below.

- `NOP` (**N**o **O**peration). It can be replaced by `JMP NEVER,PC,1`.
- `RTS0` (**R**eturn from **S**ubroutine using SP_0). It can be replaced by `POP0 PC`.
- `RTS1` (**R**eturn from **S**ubroutine using SP_1). It can be replaced by `POP1 PC`.
- `CMP` R_x, S_2 (**Comp**are). It can be replaced by `SUB` R_x, S_2, R_0.
- `LDC` R_x, C (**L**oad **C**onstant). C here is a 32 bit constant. It can be replaced by commands `LDHI` $R_x, C >> 13$ and `ADD` $R_x, C\&1FF, R_x$.

Chapter 4

Valuation of Machines

4.1 Basic Facts

We valuate machines with a function TDC as defined below.

Definition 8 *Let D be a design of a machine with costs c_D. Let B be a program with runtime t_D on design D. B is called* **benchmark**. *We call $c_D t_D$ the* **time depending cost function** *TDC of design D with benchmark B.*

We use the model of [37] to compute c_D and t_D. A motivation for our valuation is the wellknown price/performance ratio, if we take performance as the reciprocal value of runtime at constant work B.

We determine c_D and t_D of a machine by specifying the whole machine by circuits and switching networks. Each type of gate has basic cost and delay given by functions *cost* and *delay*. Examples are shown in table 4.1. The cost of a circuit is the sum of the basic costs of its gates multiplied with *packing factors* which are examples of *technology parameters*. They represent the fact that structures such as logic, arithmetic, static RAM can be packed more or less densely. Typical parameters for different technologies can be derived from chip producers' statements about placement results. We will use particular parameters derived from [36] which are shown in table 4.2. The cost of a machine is the sum of the costs of all switching networks, main memory is not counted.

We take a carry–chain adder for 8–bit numbers as an example. It consists of 8 fulladders. A fulladder consists of two halfadders and an `OR` gate. A halfadder consists of an `AND` gate and an `EXOR` gate. We have 8 `OR` gates, 16 `AND` gates and 16 `EXOR` gates in total. The adder is an arithmetic unit and thus has a packing factor of 0.75. The cost of the adder is $0.75(8cost(\texttt{OR}) + 16cost(\texttt{AND}) + 16cost(\texttt{EXOR})) = 108$.

We compute the execution times of the machine instructions (ignoring delays on wires) by searching for the maximum delay of all paths in all circuits. The delay of a path is the sum of the gate delays on this path plus a short time to load a register at the end of the path. This is a lower bound for

	`INV`	`AND, OR`	`EXOR`	1 bit Reg.
cost	1	2	6	12
delay	1	1	3	5

Table4.1: Basic cost and delay functions

Structure	Parameter	Value
Logic	ρ	1
Arithmetic	ρ_A	0.75
small SRAM	ρ_S	0.45
large SRAM	ρ_L	0.31

Table4.2: Packing Factors

the cycle time. The execution time of a machine command is the cycle time multiplied with the number of cycles the command needs (if all cycles have the equal length).

In our example the longest path is the following one: in the first fulladder from input a_{in} or b_{in} to $carry_{out}$, in the 2^{nd} to the 7^{th} fulladder from $carry_{in}$ to $carry_{out}$, in the 8^{th} fulladder from $carry_{in}$ to sum_{out}. If the $carry_{in}$ of a fulladder goes to the 2^{nd} halfadder, our path meets an `EXOR`, an `AND` and an `OR` in the 1^{st} fulladder, an `AND` and an `OR` in the 2^{nd} to the 7^{th} fulladder and an `EXOR` in the 8^{th} fulladder. The total delay is $T_{total} = 7delay(\texttt{AND}) + 7delay(\texttt{OR}) + 2delay(\texttt{EXOR}) = 20$.

We formulate benchmarks in PASCAL with the **pardo** construct [16] as parallel extention. This is sufficient for an analysis. But implementation of this language would be difficult. A better solution is given by the language FORK [21].

We determine the runtime of a benchmark B by compiling it by hand and analyzing the machine code. Depending on the CPU architecture the result is something like the number of `LOAD`, `STORE` and `COMPUTE` commands. For each group we multiply its number of commands with its execution time, then we sum over the groups. The result is the runtime t_D in gate delays. If pipelining is allowed things become messier, but can still be handled.

Section A.1 gives a detailed example of a handcompiled program. Section 5.2 gives a detailed example of how to compute execution times.

4.2 Comparison of Machines

MÜLLER and PAUL use the TDC from definition 8 to compare machines.

Definition 9 *If two designs $D0$ and $D1$ have costs c_{D0} and c_{D1} and a benchmark B has runtime t_{D0} on $D0$ and t_{D1} on $D1$ then $D0$ is called better on B than $D1$ if and only if $TDC(D0, B) < TDC(D1, B)$.*

If one compares scalable parallel machines one really compares two families of machines, the members of which are only different in size. Their costs and the runtime of the benchmark are dependent of the number of processors. To compare the families we take corresponding representants of them. These could be members of the two families that have identical costs or that have equal processor numbers.

In the first case we choose a member $D0$ of family 0 with p processors and costs $x = c_{D0}(p)$. We calculate the size q of the corresponding member $D1$ of the second family by $q = c_{D1}^{-1}(x)$. The comparison of the $TDCs$ is now only a comparison of the runtimes. If q is not an integer, we take the next integer q' and calculate the correction factor $c_{D0}(p)/c_{D1}(q')$ with which t_{D0} is multiplied.

Chapter 5

Analysis of the Fluent Machine

In this chapter we compute costs and speed of the Fluent Machine. In what follows we will only work with a model that is cheaper than the one which is provably correct. We use simpler realizations that are suggested by simulations. One example is the hash function. RANADE proves the runtime of his routing scheme for a polynomial hash function, but in his simulations he uses a linear function. Similar things happen to buffer size and routing time. Table 5.1 shows the differences. The provably correct model is also called *certified version*, the simulation based model is also called *engineering version*.

5.1 Costs of the Fluent Machine

We compute the costs of the Fluent Machine with the method introduced in chapter 4. We will ignore control logic because it occupies only a fraction of at most 10 percent of the total costs. This would change if we would use CISC processors.

The RISC processor of section 3.9 mainly consists of an ALU and a register file. The ALU consists of a 32 bit WALLACE tree multiplier, a barrel shifter and a carry lookahead adder [54]. The register file of the Fluent Machine consists of 32 registers each 32 bits wide. Let the basic costs of the ALU be A and the basic costs of the register file be F. If we use the packing factors of table 4.2 we have processor costs $c_P = \rho_A A + \rho_S F$. Because we will use a linear hash function and a LOAD/STORE architecture, the multiplier can be used for compute commands and for hashing. Thus hashing does not require any special hardware.

RANADE's simulations [42] indicate that network nodes need buffers of length 2. A node consists of 2 buffers, 2 multiplexers, a comparator and a subtractor to compare addresses for each of the two networks and a direction queue of length $2 \log n$. Let the basic costs of a network node be N_A for its arithmetic part and N_S for its SRAM, then we have costs $c_N = \rho_A N_A + \rho_S N_S$ for a network node.

Part	certified	engineering
Hash function	Polynomial of degree $8 \log n$	linear function
Buffer size	120	2
Routing time	$120 \log n$	$11 \log n$

Table5.1: Differences between certified and engineering version

A	F	N_A	N_S
13572	12288	3104	9168

Table5.2: Actual Parameters

The machine consists of $n \log n$ processors and network nodes with total costs $c_{D0} = (c_P + c_N)\, n \log n$.

The exact numbers for A, F, N_A, N_S are shown in table 5.2. They can be computed in the following way:

The ALU mainly consists of a 32 bit wallace tree multiplier, a barrel shifter and a carry lookahead adder [54]. The multiplication is performed by adding 32 terms of length 1 to 32 bits. Each bit of each term is computed by an `AND` gate. The `AND` gates have basic costs $32 \cdot 16 \cdot 2 = 1024$.

4 terms of length i can be reduced to 2 terms of length $i+1$ with an i bit 4–2–Adder consisting of $2i$ full adders. Thus we get a first stage consisting of a 32 bit 4–2–adder for the longest terms, up to a 4 bit 4–2–adder for the shortest terms. In total the first stage contains $32 + 28 + \ldots + 4 = 144$ bit 4–2–adders. The second stage contains $28 + 20 + 12 + 4 = 64$, the third stage $24 + 8 = 32$ and the last stage 16 bit 4–2–adders. The wallace tree then consists of $144 + 64 + 32 + 16 = 256$ bit 4–2–adders containing 512 full adders. As we saw in section 4 a full adder has basic costs 18. The basic costs for the wallace tree then are 9216.

The carry lookahead adder finishing the multiplication consists of $2 \cdot 32$ components to compute generate and propagate signals. Each component consists of 4 `AND` and `OR` gates. Additionaly we need for each bit 2 `EXOR` gates to compute the sum bits and 1 `AND` and 1 `OR` gate to produce the generate and propagate signal for that bit. The carry lookahead adder has basic costs $64{\cdot}4{\cdot}2{+}32{\cdot}(2{\cdot}2{+}2{\cdot}6) = 1024$.

The barrel shifter consists of 5 stages of multiplexers. Because we allow rotations and buffering in carry each stage needs 33 multiplexers with 3 inputs. These are built of 2 multiplexers with 2 inputs. A multiplexer with two inputs consists of 2 `AND` gates, 1 `OR` gate and 1 inverter, having costs 7. The total basic costs of the barrel shifter now are $5 \cdot 33 \cdot 2 \cdot 7 = 2310$. The basic costs of the ALU then are $A = 1024 + 9216 + 1024 + 2310 = 13392$.

A register file with 32 registers 32 bit wide has basic costs $F = 32 \cdot 32 \cdot 12 = 12288$.

Packets in the network are 76 bits wide (32 bit address, 32 bit data, 12 bit control). Each network node then needs the following hardware: 8 registers with 76 bits each, $2 \log n$ 3 bit registers with routing informations and 4 multiplexers with 76 bits. Additionaly we need a comparator and an adder to test identical addresses and to select the smaller one. We can restrict $\log n$ to 26 because $n \log n$ processors have to be adressable with 31 bits and thus $n \leq 2^{26}$.

The registers have basic costs $N_S = (8 \cdot 76 + 2 \cdot 26 \cdot 3) \cdot 12 = 9168$. The multiplexers have basic costs $4 \cdot 76 \cdot 6 = 1824$. The adder has basic costs 1024 as computed above. The comparator consists of 1 `EXOR` gate and 1 `OR` gate for each of the 32 bits and thus has basic costs $32 \cdot (6 + 2) = 256$. The arithmetic of a network node then has total basic costs $N_A = 1824 + 1024 + 256 = 3104$.

Thus we get $c_{D0} = 22162.2 n \log n$.

5.2 Speed of the Fluent Machine

For a particular processor design in [1] we computed a minimal cycle time $\sigma_P = 50$ gate delays, which comes from access times to the register file. We here only compute the maximal delay path in network nodes.

In one network cycle the maximum delay path is the following: a packet has to be read out of the

input buffer, its address has to be compared with another, it has to be selected by a multiplexer and it has to be stored in the input buffer of the following node. Reading the input buffer takes 3 gate delays, comparing addresses with an i bit carry lookahead adder takes about $2\log i + 2$ gate delays, selecting with a multiplexer takes 2 gate delays, storing in a buffer takes about 3 gate delays. With 32 bit addresses we have $3 + 2\log 32 + 2 + 2 + 3 = 20$ gate delays. Because we did not count driver delays and setup and hold times we take a network cycle time $\sigma_N = 25$ gate delays.

In current VLSI technology with gate delays of $2ns$ we get cycle times of $100ns$ and $50ns$. RANADE reports in [42] simulation results such that one step of the Fluent machine takes $11\log n$ network cycles which is $\nu = 11\log n \cdot 25$ gate delays. The Fluent Machine then has a power of $n\log n/\nu$ Instructions per second. For $n = 128$ we get 232 MIPS.

Let B be a benchmark with sequential runtime T that is parallizable with efficiency ϵ. Then $t_{D0} = \nu T/(\epsilon n \log n)$. The TDC then is

$$\texttt{TDC}(D0, B) = c_{D0}t_{D0} = 6.09 \cdot 10^6 T \log n/\epsilon \quad \text{gate delays.}$$

Chapter 6

Improvements and new Model

6.1 A New Proposal

As chapters 3 and 3.5 show, it seems to be very hard to improve the speed of the Fluent Machine. In the design $D1$ that we will present in this chapter we try to realize more processors than the Fluent Machine has at essentially the same costs c_{D0}. By this our runtime t_{D1} will become smaller than t_{D0} and therefore we achieve a smaller TDC than the Fluent Machine has.

Definition 10 *A* round *is the time interval from the moment when the first of all* $n \log n$ *packets is injected into the network to the moment when the last packet is returned to its processor again with the answer of a* `READ` *access.*

In RANADE's algorithm the next round can only be started when the actual round is finshed completely. This means that overlapping of several rounds (*pipelining*) is not possible in the Fluent Machine. This is the first disadvantage that we want to eliminate. This could be reached by using 6 physical butterfly networks as shown in figure 3.3. But the networks for phases 1 and 6 can be realized by n sorting arrays of length $\log n$ as described in [29] and networks for phases 3 and 4 can be realized by driver trees respective `OR` trees. Both solutions have smaller costs than butterfly networks and are not slower. The sorting arrays only have one input and require that all $\log n$ processors of a row inject their packets sequentially into this input.

This leads to the following construction as shown in figure 6.1. The $\log n$ processors of a row inject their packets into the sorting array sequentially, the sorted packets are routed like in RANADE's phase 2, the packets are directed to the right modules via driver trees. Then the packets go all the way back to their processors.

The second disadvantage is that the processors spend most of the time waiting for returning packets. This cannot be avoided, but we can reduce the cost of the idle hardware. Our machine consists of n physical processors $Ph_0, \ldots, Ph_{n-1}$. Each physical processor Ph_i simulates X *virtual processors* $vP_{i,0}, \ldots, vP_{i,X-1}$. A closely related concept is *Bulk Synchronous Parallism* in [51]. For purposes of description we need the following definition.

Definition 11 *Given virtual processor* $vP_{i,j}, 0 \leq i < n, 0 \leq j < X$*, we say that it belongs to* layer *or* column i *and* row j.

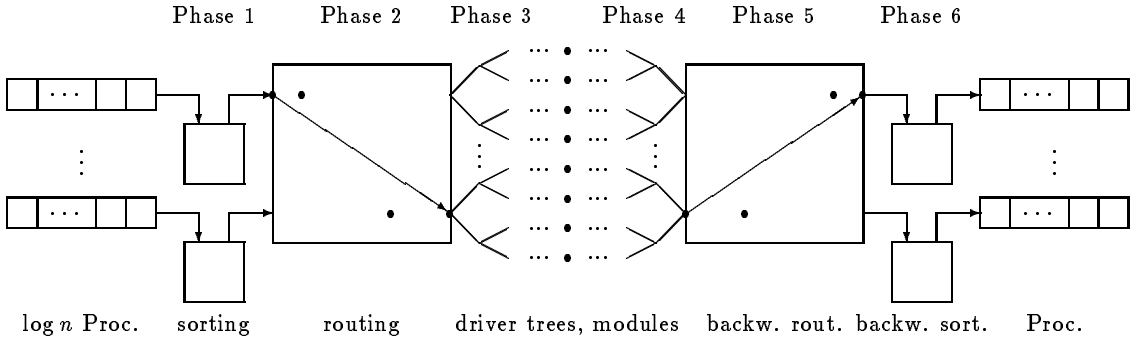


Figure 6.1: 6 Phase Routing in the New Machine

Time	1	2	3	4	5	6	x+3	x+4	x+5
Stage									
Fetch	I1	I2	I3						
Decode		I1	I2	I3					
Load arguments			I1	I2	I3				
Compute cycle 1				I1	I2	I3			
Compute cycle 2					I1	I2			
⋮									
Compute cycle x							I1	I2	I3
Store results								I1	I2

I = instruction

Figure 6.2: Pipelining in Vector Processors

6.2 Pipelining

The simulation of the virtual processors by the physical processor is done by the principle of *pipelining*. This principle is wellknown in vector computers and was also used in the first MIMD computer marketed commercially, the Denelcor HEP [22, 47]. In vector processors the execution of several instructions is overlapped by sharing the ALU. Figure 6.2 shows the pipelining used in our design. Here the ALU needs x cycles. A single instruction in this example needs $x + 4$ cycles. Execution of t instructions needs $t + x + 3$ cycles. Without pipelining they need $t(x + 4)$ cycles.

Instead of accelerating several instructions of a vector processor with a pipeline, we use pipelining for overlapped execution of one instruction for all X vP's that are simulated in one physical processor. For being able to simulate X vP's we increase the depth of our ALU artificially to $x = X - 4$. The virtual processors are represented in the physical processor only by their own register sets. Figure 6.3 shows how the RISC processor of chapter 3.9 changes when being pipelined.

Additional processor costs for pipelining several instructions are costs for latches that save the results of one pipeline stage to the next. Additional costs for pipelining X vP's in one physical processor are the costs for latches and for $X - 1$ additional register sets one for each vP.

We now want to determine X. Our goal was to increase the number of processors. The Fluent Machine uses $n \log n$ processors, we use Xn virtual processors. According to our goal we have to

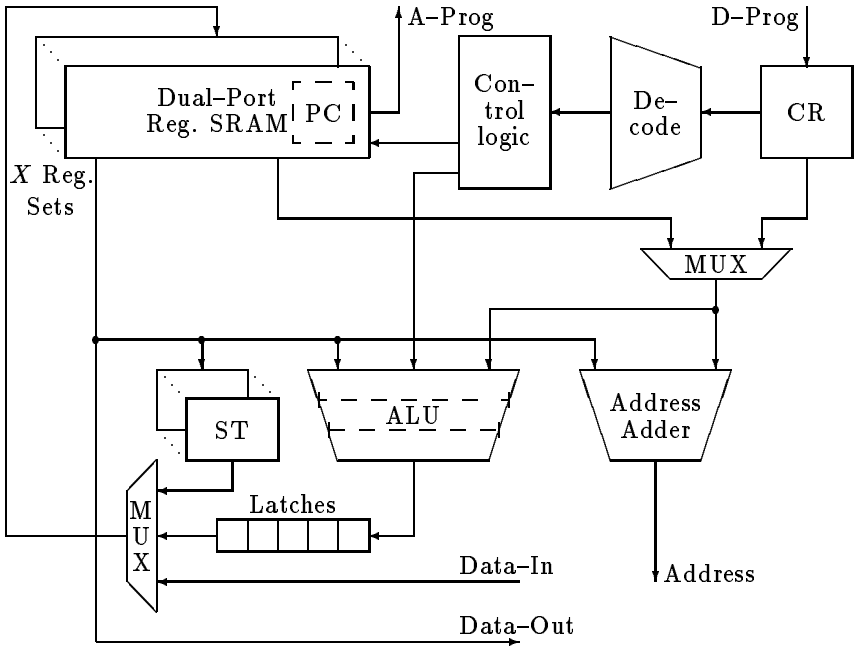


Figure 6.3: Physical Processor for the New Machine

choose $X = c \log n$ where $c > 1$. The exact c is determined in section 6.2.1 because it depends on network latency. This c increases the network conguestion. But network latency only grows slow with increasing c. Thus there exists an optimal c.

6.2.1 Routing Strategy

We now describe all phases of our routing algorithm.

Phase 1. Because we want to emulate a CRCW PRAM we have to sort the packets of all vP's of each physical processor (that is all vP's of a row) by their destinations. In contrast to the Fluent Machine we cannot sort by transmitting all packets to the row leader of phase 2. The reason for this is that the packets in our machine are produced one after another whereas in the Fluent Machine all packets of a row are put into the network at the same time.

We use a linear sorting array [29] of size X with each array element capable of combining packets with identical destinations as in the Fluent Machine's network nodes. Figure 6.5 shows an outline. The array elements look similar to the Fluent Machine's network switches with the difference that the input buffers have size 1 and that they have only one input and one output. Figure 6.4 shows an outline. An array element works as follows: if there are two packets in the registers A and B, then the one with the smaller destination stays in register B the other is transmitted across the output edge.

After the last of the X packets has entered the sorting array we move all packets to a second sorting array of equal size but inverse traffic direction. This second sorting array is necessary to pipeline phase 1 (see section 6.2.2). If there are array elements in the first array that contain two packets, these packets are inserted while outputting the packets. The claim that the packets leave the second array sorted can be proved easily.

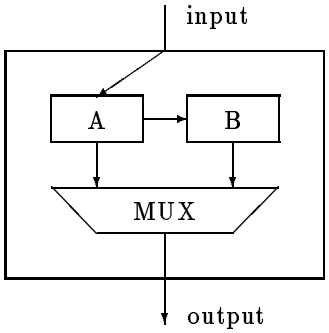


Figure 6.4: Element of the Sorting Array

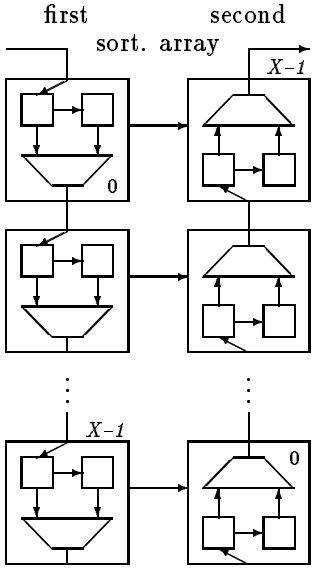


Figure 6.5: Sorting Array

Phase 2. This Phase is identical to Phase 2 of the Fluent Machine.

Phase 3. We replace RANADE's Phase 3 by the following construction: We put n memory banks at the end of the 2^{nd} network. Each bank contains $\log n$ modules that are addressed by a driver tree. The depth of the driver tree is $\alpha \log\log n$ with $\alpha \leq 2$. For $n \leq 2^{2^{10}}$ a packet can be transmitted across this tree in one network cycle, therefore pipelining in this tree is not useful for machines of reasonable size. If all packets that leave Phase 2 in row i access module $\langle i, 0\rangle$ Phase 3 needs the same amount of time in both machines. In all other cases our scheme is faster.

Phase 4. Phase 4 in our scheme consists of a *OR*–tree with invers direction to the one of phase 3.

Phase 5. Phase 5 is identical to Phase 5 in the Fluent Machine.

Phase 6. Phase 6 is the backward network to our sorting network and constructed in the same manner as the backward networks in the Fluent Machine are.

6.2.2 Pipelining the Network

In order to keep the processor pipeline filled, the network should be capable to work in pipelined mode as well. This can easily be shown by examining the phases 1 to 3.

Phase 1 can be pipelined because immediately after the last packet left a physical processor the packets are moved to the second sorting array. The first sorting array is empty again and can be filled again immediately.

Phase 2 can be pipelined too. CHANG and SIMON prove in [7] that for pipeline routing or continuous routing the routing time still is $O(\log n)$. Unfortunately they give no constants but our simulations in chapter 6.3 show small constants.

Phase 3 only takes one cycle to transmit a packet as shown above. Pipelining is not necessary in this phase.

Definition 12 *A* round *in machine $D1$ is the time interval from the moment when the first vP injects its packet into the network to the moment when the last vP injects its packet into the network.*

At the end of a round there are still packets of this round in the network but the processors have to proceed and thus must start the next round. The problem how to seperate the different rounds can easily be solved. After the last vP has injected its packet into the network, an *End of Round Packet (EOR)* is inserted. This is a packet with a destination larger than memory size m. Because the packets leave each node sorted by destinations, it has to wait in a network switch until another EOR enters this switch across its other input. It can be proved easily that this is sufficient to seperate rounds.

The only problem to be solved is that virtual processors that execute a `LOAD` instruction have to wait until the network repeats the answer to their `READ` request. We partially overcome this by introducing delayed `LOAD` instructions. We require an answer to a `READ` request being available not in the next instruction but in the next but one. Investigations show that insertion of additional 'dummy' instructions happens very rarely [39]. But if a program needs any dummy instructions, they can be easily inserted by the compiler. This reduces c by a factor of two without significantly slowing down the machine.

The sorting arrays should have length $c \log n$, too. Breaking a round in z parts is an alternative. This reduces the lengths to $(c \log n)/z$ but could slow down the machine. Simulations in chapter 6.3 will show that there is a maximum value of z that does not affect network speed. This additionaly decreases c.

6.3 Simulator and Simulation Results

It is necessary to simulate parts of the machine we intend to build. Not all decisions about what to build can be made by studying theory or proving theorems.

The most important facts about our machine that we have to figure out are

- the cheapest and fastest hash function that is sufficient for our routing scheme,
- the optimal number $c \log n$ of virtual processors in one physical processor,
- the size of buffers and queues in the network nodes,
- the maximum possible value for z.

It is clear that we will not simulate the machine as a whole to get the answers to the different questions we stated above. This just would cost too much time. Therefore we break our simulation into two parts:

- the *logical simulator* that simulates a program on the machine from the user's view. As an option it can produce a file with all requests that in the real machine would have been put into the network.
- the *network simulator* that simulates `LOAD` and `STORE` requests on the network. It produces statistics about average routing time at given parameters such as buffer size.

The simulation of the network enables us to determine good to optimal values for c and to give answers to our above questions. But these values depend on the requests that were used as input for the network simulation. In order to have 'real' requests we simulate benchmarks on the logical simulator. First we test whether we implemented them correctly, then we add the request option to get the requests. For testing a program the logical simulator has some debugging features such as single stepping and breakpoints.

Sometimes it is useful to generate requests with special patterns to test the network in situations that we hope to be fast handled by our network. For this use serves a *pattern generator program* that allows to generate many different request patterns or mixtures of them.

The advantages of breaking the simulation into two parts are that on the one hand we do not need too much time when simulating a program just to see if it runs correctly and on the other hand we can test the network with requests produced by PRAM programs or with generated patterns.

As a compiler for our PRAM will generate assembler files in text format we also wrote an assembler program that generates the input code for the logical simulator. Thus the compiler can be tested with the logical simulator already before the prototype will be finished.

The four programs work together as shown in figure 6.6.

Since the simulator has just recently been available we could only do few simulations. We are mostly interested in the number of network cycles needed for phase 2 to 5 of our scheme. This is necessary because the influence of pipelining on the constants in routing time is not known. Second we are interested whether sorting arrays of smaller size (e.g. $\frac{X}{2}$ or $\frac{X}{4}$) are sufficient.

We did two kinds of simulations:

1. We started the network simulator with a file of randomly chosen `READ` requests for 5 instructions of all vP's (i.e. $X * 5$ rounds for physical processors). We tested for $n = 32$ and $n = 256$ processors. We measured the expected number of cycles for a packet from its entry in phase 2 to its output of phase 5.

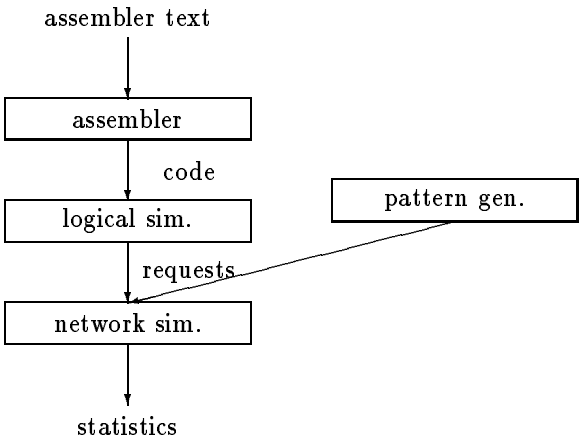


Figure 6.6: Simulation Software

n		32	256
simulation 1		28.4	44.6
simulation 2	X	21.5	
simulation 2	$\frac{X}{2}$	24	
simulation 2	$\frac{X}{4}$	26.2	45.5

Table6.1: First Simulation Results

2. We took a request file in which all layers of vP's execute 5 identical `READ` requests. If we take sorting arrays of size $X' < X$ this decreases the hardware costs of phase 1 and increases the module congestion c_m by a factor of X'/X. Packets with identical destinations out of layers i_0 and i_1 with $|i_0 - i_1| > X'$ will certainly not be combined. We tested for $n = 32$ and $n = 256$ processors and for sorting arrays of size X, $\frac{X}{2}$ and $\frac{X}{4}$. We measured the number of cycles as above.

We had the following results as shown in table 6.1. We show only the expected values, the variances are smaller than 3.

The results indicate that our routing scheme is fast ($\leq 5.8 \log n$ for phases 2 to 5). Sorting arrays of size $\frac{X}{4}$ still provide a routing for requests of the second kind as fast as for random requests.

6.4 Summary of the Improvements

We summarize our changes:

- Slowing down the execution time for one instruction to $c \log n$ processor cycles.
- Increasing the number of processors by a factor c.
- New organization with n physical processors each with $c \log n$ virtual processors.
- Simplifying Phases 3 and 4 by introducing a driver and an OR tree.

- Pipelining of virtual processors and network.
- Delayed `LOAD`.

In chapter 7 we will examine if our changes are real improvements, that means if the gain in speed from our increased number of processors keeps up with the slowing down of the machine.

Chapter 7

Analysis of the Improvements

7.1 Costs of the new Machine

To determine the costs of the new machine, we present all changes to the Fluent Machine and then give an overview of the costs. As processor of the new machine we take the processor of the Fluent Machine with virtual processors and pipeline. As in chapter 5 we use an engineering model. As the simulation results of chapter 6.3 indicate we take sorting arrays of size $\frac{X}{4}$.

The costs for the processor change to $c_{P'} = \rho_A A + \rho_L c \log nF$, because the processor now contains $c \log n$ register sets. A network node now consists of 2 buffers and 2 multiplexers 76 bits wide on the way from processors to memory, 2 buffers and 2 multiplexers 32 bits wide on the way back, a direction queue of length $2c \log n$ and a comparator and a subtractor to compare addresses. Sorting nodes look similar but only need buffers of length 1 and 1 multiplexer for each direction. Let the basic costs of a network node be N'_A for its arithmetic part and N'_S for its SRAM, the basic costs of a sorting node S'_A and S'_S. Then we have costs $c_{N'} = \rho_A N'_A + \rho_S N'_S$ for a network node and $c_{S'} = \rho_A S'_A + \rho_S S'_S$ for a sorting node.

The improved machine consists of n physical processors, of $2(cn \log n)/z$ sorting nodes and of $n \log n$ network nodes. It has total costs

$$c_{D1} = nc_{P'} + c_{S'} \frac{2c}{z} n \log n + c_{N'} n \log n.$$

The exact numbers for N'_A, N'_S, S'_A, S'_S are shown in table 7.1, their values can be obtained in the following way:

Packets on the way from processors to memory modules are 76 bits wide (32 bit address, 32 bit data, 12 bit control). On the way back 32 bits for transported data are sufficient. Each network node needs the following hardware: 4 registers with 76 bits each, 4 registers with 32 bits each, $2c \log n$ 3 bit registers with routing informations for the backward network, 2 multiplexers with 76 bits and 2 multiplexers with 32 bits. Additionaly we need a comparator and an adder to test identical addresses and to select the smaller one. As in chapter 5 we restrict $\log n$ to 26.

The registers have basic costs $N'_S = (4 \cdot 76 + 4 \cdot 32 + 2 \cdot 3 \cdot 26 \cdot 3) \cdot 12 = 10800$. The multiplexers have basic costs $(2 \cdot 76 + 2 \cdot 32) \cdot 6 = 1296$. The adder has basic costs 1024, the comparator has basic costs 256 as computed in section 5.1. The arithmetic of a network node then has total basic costs $N'_A = 1296 + 1024 + 256 = 2576$. The nodes for the sorting network only need buffers of length 1 and only 1 multiplexer. They have basic costs $S'_A = 1928$ and $S'_S = 8208$.

If we choose $c = 3$ as we will compute in section 7.2 and $z = 4$ as chapter 6.3 indicates, the total costs of the new machine are

N'_A	N'_S	S'_A	S'_S
2576	10800	1928	8208

Table7.1: Actual Parameters

$$c_{D1} = 10179n + 25929.24n \log n. \tag{7.1}$$

7.2 Speed of the new Machine

The cycle times for the improved machine are identical to those of the Fluent machine. One step of the improved machine takes $\nu' = c \log n 50$ gate delays. The improved machine then has a power of $\frac{cn \log n}{\nu'}$ Instructions per second.

c is chosen in such a way that a virtual processor normally does not have to wait for an answer to a `READ` request. Phase 1 needs at most $\frac{c}{4} \log n$ processor cycles for the first sorting array and $\frac{c}{4} \log n$ network cycles for the second. Phases 2 to 5 need at most $5.8 \log n$ network cycles. Phase 6 needs at most $\frac{c}{4} \log n$ network cycles for each sorting array. In total a packet needs at most $\frac{5c}{4} \log n + 5.8 \log n$ network cycles.

Because we use a delayed `LOAD` the network access can use $2c \log n - 5$ processor cycles. Execution of a instruction not using delayed `LOAD` (e.g. a `COMPUTE` instruction) needs $c \log n$ processor cycles which is the depth of the pipeline. Thus the total time for a delayed `LOAD` is $2c \log n$ processor cycles. We need 4 processor cycles at the beginning of the execution for instruction fetch and decoding and one cycle at the end for storing the loaded result. The rest of the $2c \log n$ processor cycles can be used for the network access itself. Now we get the following inequality:

$$50(2c \log n - 5) \geq 25 \left(\frac{5c}{4} \log n + 5.8 \log n \right). \tag{7.2}$$

This can be transformed to

$$c \geq 2.1 + \frac{3.64}{\log n}.$$

If we choose $n \geq 32$ and estimate $\frac{3.64}{\log n}$ by $3.64/5 = 0.73$ we get $c \geq 2.84$ and choose $c = 3$.

These execution times change for our prototype in chapter 11 because there we have strict pin and gate limitations.

Let B be a benchmark with sequential runtime T that is parallizable with efficiency ϵ. Then $t_{D1} = \nu' T/(\epsilon cn \log n)$. The `TDC` then is $\texttt{TDC}(D1, B) = c_{D1} t_{D1} = (0.51 + 1.3 \log n) \cdot 10^6 T/\epsilon$ gate delays.

7.3 Comparison

We now compute the quotient $\texttt{TDC}(D0, B)/\texttt{TDC}(D1, B)$. The corresponding members in our case have the same size n. The result is

$$\frac{\texttt{TDC}(D0, B)}{\texttt{TDC}(D1, B)} \approx 4.7$$

7.4 A Variation of Design D1

One can try to avoid the delayed `LOAD` in our design thus having a design $D2$. To do this we change inequality 7.2 to

$$50(c\log n - 5) \geq 25\left(\frac{5c}{4}\log n + 5.8\log n\right) \tag{7.3}$$

and get now $c \geq 10.4$, i.e. physical processors get more expensive. This leads to $\texttt{TDC}(D2, B) = (0.509 + 5.59\log n) \cdot 10^6 T/\epsilon$ and to a quotient

$$\frac{\texttt{TDC}(D1, B)}{\texttt{TDC}(D2, B)} = \frac{0.5 + 1.3\log n}{0.5 + 5.59\log n} \approx 0.232$$

This quotient is smaller than 0.5. This means that design $D1$ is better even if we take a program that only consists of `LOAD` instructions each depending on the preceeding instruction. One has to insert a dummy instruction (e.g. a `NOP`) after each `LOAD` instruction what slows down the machine by a factor of exactly 2 and makes the `TDC` twice as large as it was before, which is still smaller than 1. This argument shows that delayed `LOAD` is worthwile for any benchmark B.

Chapter 8

CRCW vs. EREW

8.1 Main Result

We investigate the question whether combining should be done by hardware *(hardwired combining)* or whether concurrent accesses should be simulated by software. We will prove the following theorem 4.

Theorem 4 *Let $D1$ be a CRCW PRAM as described in chapter 6 which supports combining by hardware. Let $D3$ be an EREW PRAM as described in section 8.2 on which each concurrent access is simulated by software as described in section 8.3. If a benchmark B that needs t_{D1} steps consists of αt_{D1} concurrent accesses with $0 \leq \alpha \leq 1$ then*

$$\texttt{TDC}(D1, B) < \texttt{TDC}(D3, B) \quad for \quad \alpha > \frac{0.408}{(\log n)^2}.$$

This means: if a benchmark that needs t_{D1} steps consists of more than $0.408(\log n)^{-2} t_{D1}$ concurrent accesses it is better to run it on a CRCW PRAM instead of simulating it on an EREW PRAM.

8.2 Design of an EREW PRAM

To determine $\texttt{TDC}(D3, B)$ it is necessary to sketch the design of an EREW PRAM $D3$. We get $D3$ from $D1$ by skipping all hardware that supports combining. These are the sorting networks in phases 1 and 6 of the routing and the comparators in the network switches which detect that combining is necessary. Additionally one can reduce the width of the direction queues in the switches to two bits because only four cases remain: 'in_i to out_j' where $i, j \in \{0, 1\}$. Removing the sorting networks reduces routing time and c can be decreased to $\tilde{c} = 1.5$. The costs for the new processors ar $c_{\tilde{P}} = \rho_A A + \rho_L \tilde{c} \log nF = 10179 + 5713.9 \log n$. The costs for network nodes decrease from N'_A to $\tilde{N}_A = 2320$ and from N'_S to $\tilde{N}_S = 7056$. The total costs for $D3$ are

$$c_{D3} = c_{\tilde{P}} n + c_{\tilde{N}} n \log n = 10179n + 10629.1n \log n. \tag{8.1}$$

The cycle time of $D3$ is exactly the same as of $D1$, one step of $D3$ takes $\tilde{c} \log n$ processor cycles.

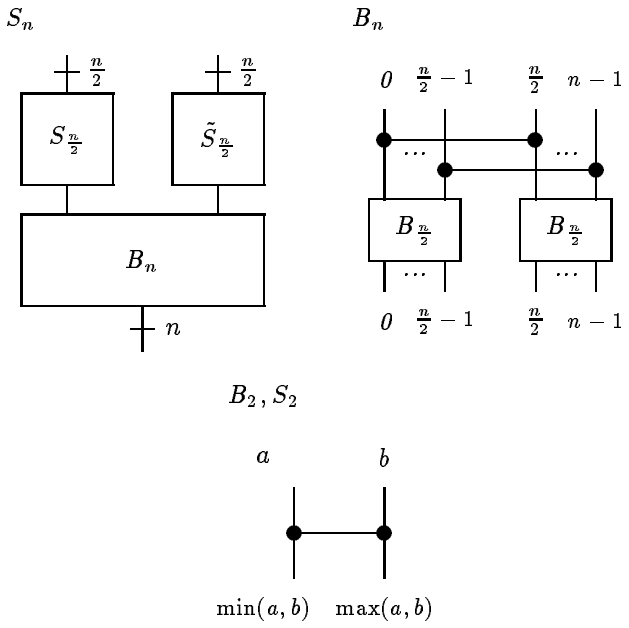
8.3 Simulation of CRCW on EREW

KARP and RAMACHANDRAN show in [26] how to simulate a CRCW PRAM on an EREW PRAM. They use the following method to simulate one step in which concurrent accesses can happen:

Suppose processor P_i wants to access variable V_j. Then it writes (i, j) to location i in global memory (we assume that locations 0 to $n-1$ are not accessed by the PRAM program). The contents of locations 0 to $n-1$ now get sorted by j. Duplicates which represent concurrent accesses are replaced by dummy accesses $(i, -j)$. P_i reads the content (i', j') of location i and accesses $V_{j'}$ if $j' \geq 0$. Then P_i writes the result of a `READ` access to location i'. The processors with eliminated duplicates now duplicate the results. At last P_i reads the result of its own access from location i and assigns it to variable V_j.

The most time consuming part of the simulation is the sort of the tuples (i, j). The sort can be parallized by using all n processors to sort the n tuples. Because a sequential sort by comparison of n elements needs time $\Omega(n \log n)$, an optimal parallel algorithm using all n processors should need parallel time $\Theta(\log n)$. Optimal sorting algorithms are described in [3, 8, 38], a randomized one is given in [43]. The constant factor in their runtime however is quite large. We will use BATCHER's *bitonic sort* [6], a parallel sorting algorithm with small constant that needs time $O(\log^2 n)$ to sort n elements using n processors. The bitonic sorting network can be defined recursively as in definition 13.

Definition 13 *B_2 and S_2 are identical circuits sorting two numbers. B_n is a circuit that merges two bitonic sequences each of length $\frac{n}{2}$ to one bitonic sequence of length n. The* bitonic sorting network *for n numbers is a circuit S_n. For one of these circuits S, $\tilde{S}$ denotes the circuit with reversed order of outputs.*



The bitonic sorter can be formulated as a program. The program needs n processors that simulate in step i the n comparators B_2 in depth i of the circuit. The algorithm looks as shown in figure 8.1.

We assume that the compiler for our benchmark can recognize all instructions in which concurrent access can occur and that only these instructions are simulated in the way described above. We further assume that the compiler knows the number of processors that are working at this time.

```
for pnum := 0 to n − 1 pardo
  for i := 1 to log n do
    for k := i − 1 to 0 do
      if bit k of pnum = 0 then
        if bit i of pnum = 0 then
          A[pnum] := min(A[pnum], A[pnum + 2^k])
        else
          A[pnum] := max(A[pnum], A[pnum + 2^k])
        fi
      else
        if bit i of pnum = 1 then
          A[pnum] := min(A[pnum − 2^k], A[pnum])
        else
          A[pnum] := max(A[pnum − 2^k], A[pnum])
        fi
      fi
    od
  od
od;
```

Figure 8.1: Bitonic Sort Algorithm

Now the compiler can generate code for the bitonic sort without using loops or subroutine calls. This makes it much faster. An assembler program would need $9.5\,(\log n')^2 + 10.5 \log n'$ instructions for the bitonic sort as described above. n' is the smallest power of two larger than the number of processors. In our design $D3$ $n' \geq \tilde{c} n \log n$. The complete simulation of one step then takes

$$t_{sim} = \frac{19}{2}\,(\log n')^2 + \frac{47}{2} \log n' + 46. \tag{8.2}$$

instructions. The complete assembler program can be found in [1]. Now we will prove theorem 4 using the results of the previous sections.

Proof: (indirect) Let B be a benchmark that needs T steps on $D1$, thus $t_{D1} = 50Tc \log n$ gate delays. On $D3$ it will need $T/\alpha t_{sim} + (1-\alpha) \cdot 1)$ steps and time

$$t_{D3} = 50T\,(\alpha t_{sim} + (1-\alpha)\,1)\,\tilde{c} \log n \quad \text{gate delays.} \tag{8.3}$$

$$\begin{array}{rcl} \texttt{TDC}(D3, B) & \leq & \texttt{TDC}(D1, B) \\ c_{D3} t_{D3} & \leq & c_{D1} t_{D1} \\ \overset{(8.3)}{\Rightarrow} \quad \alpha & \leq & \dfrac{\frac{c}{\tilde{c}}\frac{c_{D1}}{c_{D3}} - 1}{t_{sim} - 1} \end{array}$$

If we assume in favour of $D3$ that $n' = \tilde{c} n \log n$ then with equations 7.1, 8.1, 8.2 we get $\alpha \leq 0.408 (\log n)^{-2}$. ∎

For moderate n however, the exact value is even smaller.

8.4 Consequences

We mentioned in section 1.2.1 that PRAMs are classified in theory as EREW, CREW and CRCW PRAMs. Relations among these classes are given in [16, 26]. A further class of ERCW PRAMs is not considered there.

Definition 14 *A machine model A is said to be hierarchically weaker than B ($A \preceq B$) if each problem that can be solved on model A in time T and P processors can also be solved on model B in time $O(T)$ and $O(P)$ processors.*

Obviously $EREW \preceq CREW \preceq CRCW$.

Theorem 5 *If we change our CRCW design $D1$ to an EREW design $D3$, an ERCW design $D4$ and an CREW design $D5$ we get the relation*

$$\texttt{TDC}(D3, B) < \texttt{TDC}(D4, B) < \texttt{TDC}(D5, B) = \texttt{TDC}(D1, B).$$

Thus if a PRAM supports combining in the way we described in section 3.7 it is not worthwhile to consider CREW PRAMs but it might be useful to examine the role of ERCW PRAMs in the hierarchy.

Proof: (of theorem 5)

We get $D4$ from $D1$ by reducing the width of the direction queues with the same argument as in section 8.2. This shows $c_{D4} < c_{D1}$. We cannot skip the comparators because we still have to detect concurrent writes. This shows $c_{D3} < c_{D4}$. For $D5$ we cannot skip the comparators because we have to detect concurrent reads. We cannot reduce the width of the direction queues because of the same argument. This shows $c_{D5} = c_{D1}$. The sorting networks can only in $D3$ be skipped because of concurrent access in $D4$ and $D5$. Therefore $t_{D3} < t_{D4} = t_{D5} = t_{D1}$. ∎

Theorem 5 shows that $D5$ is identical to $D1$ and that for any PRAM program B $t_{D1} = t_{D5}$. Thus $\texttt{TDC}(D5, B) = \texttt{TDC}(D1, B)$ but $CRCW \not\preceq CREW$.

Chapter 9

PRAMs vs. Distributed Memory Machines

PRAMs always have been thought not to be competitive to Distributed Memory Machines (DMM) because some problems do not need the global memory. In order to compare our PRAM $D1$ with a DMM $D6$ one has to compute $R = \texttt{TDC}(D1, B)/\texttt{TDC}(D6, B)$. We are interested in how much more cost–effective DMMs can be than PRAMs and vice versa. Therefore we search for upper and lower bounds U and L with $L \leq R \leq U$ independently of B and of the particular DMM. In the following sections we will prove $U = O(\log n)$ and $L = \Omega(1)$ as one would expect and show that these bounds are tight.

Suprisingly the constant factor in U is smaller than 1 but the one in $1/L$ is significantly larger than 1. This means that for reasonable values of n a DMM cannot be much more cost–effective than a PRAM but vice versa.

9.1 An Upper Bound

Assume a benchmark that does not use the global memory but can be run on a distributed memory machine with simple hardwired communication. This is the worst case that can happen when comparing PRAMs and DMMs. We formulate an upper bound as theorem 6.

Theorem 6 *Assume we have a benchmark B as just described that has enough parallelism to be computed on a parallel machine with efficiency ϵ close to 1. We consider a DMM $D6$ with n processors and communication given by a graph of small degree with n nodes and our PRAM $D1$. Then we get*

$$R \leq U \leq 1.65 \log n + 0.65.$$

Proof: The distributed memory machine with n processors has costs $c_{D6} = nc_P = 15708.6n$. We only count processor costs c_P and ignore network costs although this is unfair towards the PRAM. Suppose that B needs T steps on a sequential machine. The DMM needs $T/\epsilon n$ steps. We assume in favour of the DMM that the benchmark B can be pipelined perfectly and thus one step only takes one cycle. Thus one has $t_{D6} = 50T/\epsilon n$.

The PRAM has costs c_{D1} as computed in equation 7.1 and needs $T/(\epsilon cn \log n)$ steps each taking $c \log n$ processor cycles. Thus $D1$ needs $T/\epsilon n$ cycles and therefore $t_{D1} = 50T/\epsilon n$ gate delays. We

then get

$$R = \frac{c_{D1}}{c_{D6}} \cdot \frac{t_{D1}}{t_{D6}} = \frac{25929.24n\log n + 10179n}{15708.6n} \cdot 1 = 1.65\log n + 0.65 = U.$$

■

For reasonable values of n, e.g. $n \leq 2^{16}$, the quotient is less than 27. If we would add floating point arithmetic to the ALU as usual in existing parallel machines, the parameter A increases to $A' \approx 100000$ [15] and the quotient decreases dramatically to $0.32\log n + 0.93$. For $n \leq 2^{16}$ the quotient is smaller than 6. If cost of memory is considered too, things change further in favour of the PRAM.

9.2 A Lower Bound

The worst case for a DMM is a benchmark where any known algorithm for a DMM is less cost–effective than the step–by–step simulation of a PRAM.

Theorem 7 *Let B be a benchmark that fulfils the above assumptions and that is parallelizable with efficiency ϵ. Then*

$$R \geq L \geq \frac{1}{320}.$$

Proof: Let the sequential runtime of B be T. B needs $T/\epsilon cn\log n$ steps on a PRAM $D1$ with $cn\log n$ processors. Because each step takes $c\log n$ processor cycles, $t_{D1} = 50T/\epsilon n$.

Let $D6$ be a DMM with $n\log n$ processors interconnected as a butterfly (a Cube–Connected Cycles network is also possible). $D6$ has costs $c_{D6} = c_P n\log n = 15708.6n\log n$ because we ignore network costs. In order to simulate one step of $D1$ on $D6$ we adapt RANADE's routing scheme in software. Because successing phases can overlap we use a link in forward manner for phases 1,3,5 and in backward manner for phases 2,4,6. Processors alternately execute one step of phase i and one of phase $i+1$. Because of this toggling the routing scheme needs at most twice as many routing steps as RANADE's scheme. The number of machine instructions to perform one routing step is 24:

# steps	comment
6	read address, data, mode of both inputs
1	compare the addresses
1	jump if equal (combining)
1	jump if less (left packet is to send)
1	compare address with routing mask
1	jump if equal (routing to left output)
1	test whether successing queue is full
1	jump if full
3	write address, data and mode
2	append direction queue if mode==read
1	mark input queue not full
1	test whether other successing queue full
1	jump if full
3	write address,data,ghost
$\sum$ 24	Total

If we assume that RANADE's scheme needs $11 \log n$ steps the new scheme needs $11 \log n \cdot 24 \cdot 2 = 528 \log n$ instructions. If we further assume that one instruction only takes one processor cycle, the total time to simulate one PRAM step is at most $S \log n$ processor cycles for $S = 528$. $D6$ simulates a PRAM with $n \log n$ processors. Therefore B needs $T/\epsilon n \log n$ steps on $D6$ and $t_{D6} = 50ST/\epsilon n$. We now can compute R:

$$R = \frac{c_{D1}}{c_{D6}} \cdot \frac{t_{D1}}{t_{D6}} \geq \frac{25929.24n \log n + 10179n}{15708.6n \log n} \cdot \frac{1}{S} = \frac{1}{320} = L$$

■

If we add floating point arithmetic L changes to $\frac{1}{1650}$.

9.3 Examples

In order to show that the bounds on R are tight we present two examples matching the bounds. The first example $B0$ is multiplication of two $s \times s$–matrices. We use design $D1$ with n physical processors and a distributed memory machine $D6$ with n processors interconnected as an $\sqrt{n} \times \sqrt{n}$ torus. Each processor of the torus then holds a $\frac{s}{\sqrt{n}} \times \frac{s}{\sqrt{n}}$–submatrix of both matrices. This example comes very close to the worst case described in section 9.1 and therefore R approximately matches the upper bound.

The second example $B1$ is computing the connected components of an undirected graph with v nodes and e edges. For the PRAM we use an algorithm of [45] in a form presented in [20]. Its runtime is $O(\log v)$ steps on a PRAM with $2e$ (virtual) processors. The formal explanation and the proofs for correctness and runtime can be found in [45]. On a PRAM with $n < v$ physical processors we have $t_{D1} = (300\frac{e}{n} + 108\frac{v}{n}) \log v$ as analyzed in appendix A and c_{D1} as computed in equation 7.1. For the distributed memory machine we could use an algorithm from [4] that runs on a hypercube. Its runtime is $O(\log^2 v)$ on v^3 processors. For a hypercube $D7$ with $n < v$ processors we would have $c_{D7} = 15708.6n$ as computed in section 9.1, $t_{D7} = 40\frac{v^3}{n}(\log v)^2$ as sketched now:

Let $G = (V, E)$ be an undirected graph with $v = |V|, V = \{0, \ldots, v-1\}$ and $E \subseteq V \times V$ with $e = |E|$. Represent E by the adjacency matrix A given by $a_{jk} = 1$ if $(j,k) \in E$, 0 otherwise. A is symmetric because G is undirected. The connected components algorithm from [4] first computes the connectivity matrix C from the given adjacency matrix. C is given by $c_{jk} = 1$ if there exists a path in G from j to k, 0 otherwise. Then it constructs a matrix D given by $d_{jk} = k$ if $c_{jk} = 1$, 0 otherwise. Last each vertex k is assigned to component l with $l = \min\{i | d_{ki} \neq 0\}$.

We assume that sending one word across a link of the hypercube only takes one step and that source and destination of this word are registers. The connectivity matrix is computed by $\log v$ times multiplying A with itself thus computing $C = A^v$. It turns out that multiplying 2 $v \times v$ matrices on a hypercube with n processors can be done in $(38\frac{v^3}{n} + 20) \log v + 5\frac{v^3}{n}$ steps. The computation of the connectivity matrix then needs approximately $(\log v)^2(38\frac{v^3}{n} + 20) + 5\frac{v^3}{n} \log v$ steps. The computation of matrix D takes approximately $7\frac{v^2}{n}$ steps, finding of minimums takes approximately $10\frac{v}{n} \log v$ steps. The total time t_{D7} then is approximately $40\frac{v^3}{n}(\log v)^2$.

Using the fact that $e \leq \frac{1}{2}v^2$ leads to $R \approx 5\frac{\log n}{v \log v}$. This would implicate $R < L$ and therefore a simulation of the PRAM algorithm is more cost–effective.

Chapter 10

Multiprefix

10.1 Definition

A special instruction of the Fluent Machine is *multiprefix*. Ranade defines it as follows:

Definition 15 *The instruction multiprefix has the form* `MP` $A, \circ, D$. *A is an address, $\circ$ an associative binary operator, D are data. If two processors are executing a multiprefix instruction at the same time and with the same address they have to use the same operator. Let $P_A = \{p_1, \ldots, p_k\}$ be at time T a set of processors with $p_1 < \ldots < p_k$ which are executing* `MP` $A, \circ, D_i$ *with data $D_1, \ldots, D_k$. Let a be the content of A at time T. After the execution of the multiprefix instruction p_i receives the value $a \circ D_1 \circ \ldots \circ D_{i-1}$, the new content of A is $a \circ D_1 \circ \ldots \circ D_k$.*

This definition looks very similar to parallel prefix. The most important difference is that multiprefix allows several disjunct groups $P_A, P_B, \ldots$ at the same time executing several parallel prefix computations.

In the Fluent Machine and in its improved version multiprefix can be supported by hardware in the following way:

The execution of this operation is done by the network nodes. Therefore the network and sorting nodes of phases 1,2,5,6 contain arithmetical units and buffers for storing the intermediate results. These buffers are called *multiprefix memory* (MP–memory). They have the same length as the direction queues, the width is that of a word. We expand the direction queue with the possibilty to execute multiprefix to the left/right output. The memory modules also contain arithmetical units because we divide one round in z parts and therefore at most z packets belonging to one `MP` instruction can arrive at the module containing A.

If two packets with data D_i, D_j from processors p_i, p_j with $i < j$ belong to the same `MP` Instruction, they will meet in a network (or a sorting) node as if they would perform a concurrent read. Combining these packets includes storing D_i in MP–memory, forwarding one packet with data $D_i \circ D_j$ and eliminating the other packet. If a packet with data D arrives at the memory, the content a of address A is read out, $D \circ a$ is stored at address A and A is sent back. If an answer of such an instruction arrives on its way back with data D, D is forwarded along left output and $D \circ D_i$ along right output.

It can be shown by an easy induction that this fulfills the above definition.

In addition to multiprefix we introduce an instruction called `SYNC`. The only difference between the two instructions is that in `SYNC` the processors do not get back any results. `SYNC` therefore only

needs operators in phases 1, 2 and at the memory modules. MP–memory and extended direction queues are not necessary.

10.2 Use of Multiprefix and SYNC

`SYNC` is used for synchronization after executing parallel high level language constructs. Before executing the construct a memory cell A is set to zero and all processors that take part execute `SYNC` $+, A, 1$. Now A contains the number of the processors that take part. If a processor has finished, he executes `SYNC` $+, A, -1$ and waits. The execution is finished completely if the content of A is zero. It is important that at no time one processor executes `SYNC` $+, A, -1$ and others try to read A at the same time. Some of them could read the old value larger than zero and others the new value zero. The synchronization has failed. We solve this problem by writing only in even instructions and reading in odd instructions. This restriction is only necessary for synchronization. To easily distinguish odd and even, the status register has a flag that changes its value with every instruction. The value of this flag is identical for all processors. Things are messier because of delayed LOAD. The wait loop of processors that have finished has minimum length 3. But if one takes loop length 4, processors can be synchronized in two steps.

`MP` is used in some PRAM algorithms as a subroutine. Furthermore it is used in the FORK compiler [21] for memory management. Suppose several processors p_i want to allocate storage of size D_i, a pointer to the beginning of free storage is held at address A. Each processor executes `MP` $+, A, D_i$. With the definition of multiprefix it can easily be shown that each processor gets as an answer a pointer to its storage and that A contains a pointer to the new beginning of free storage.

10.3 Cost–Effectiveness of Multiprefix

Theorem 8 *Let B be a benchmark with a fraction of β* `MP` *instructions. Let $D1$ be a PRAM without and $D8$ be a PRAM with hardware support of multiprefix for 4 operators: addition, bitwise AND, bitwise OR, maximum. Then*

$$\texttt{TDC}(D8, B) < \texttt{TDC}(D1, B) \quad for \quad \beta > \frac{0.25}{(\log n)^2}.$$

Proof: We compute the costs of the additional hardware to support multiprefix. We take 4 operators: addition, bitwise AND, bitwise OR, maximum.

One arithmetical unit to perform a multiprefix operation then consists of an adder for the addition, gates for bitwise AND and OR, an subtractor and a multiplexer for maximum, and 3 multiplexers to select the chosen operation. an adder has basic costs 1024 as computed in section 5.1, the gates have basic costs $2 \cdot 32 \cdot 2 = 128$, the subtractor consists of an adder and an inverter and thus has basic costs $1024 + 32 \cdot 1 = 1056$, the four multiplexers have basic costs $4 \cdot 32 \cdot 6 = 768$. The arithmetical unit then has basic costs $MP_A = 1024 + 128 + 1056 + 768 = 2976$.

The extended direction queue and the MP–memory has basic costs $MP_S = (32 + 1) \cdot 12 \cdot 2 \cdot c \cdot \log n$. If we restrict $\log n$ to 26 as in section 5.1 $MP_S = 61776$.

Each of the sorting and network nodes contains 2 arithmetical units (one for each direction) and a MP–memory. Each memory module contains 1 arithmetical unit. The additional costs for multiprefix are

$$c_{MP} = \left(n \log n + \frac{2c}{z} n \log n \right) (2\rho_A MP_A + \rho_L MP_S) + n \log n \rho_A MP_A = 61268.4 n \log n.$$

A design $D8$ that supports multiprefix in hardware has costs $c_{D8} = c_{D1} + c_{MP}$.

The time to simulate one `MP` instruction on design $D1$ is similar to the time to simulate a concurrent read/write step on an EREW PRAM. Let B be a benchmark that has a sequential runtime T and that is parallizable with efficiency ϵ. Let β be the fraction of `MP` instructions. On design $D1$ B has runtime $t_{D1} = (\beta t_{sim} + (1-\beta)1)T/(\epsilon cn \log n)$. The runtime on $D8$ is $t_{D8} = T/(\epsilon cn \log n)$.

We compute the range for β for which $\texttt{TDC}(D8, B)/\texttt{TDC}(D1, B) < 1$.

In analogy to section 8.3 we compute

$$\frac{\texttt{TDC}(D8, B)}{\texttt{TDC}(D1, B)} < 1 \Rightarrow \beta > \frac{\frac{c_{D8}}{c_{D1}} - 1}{t_{sim} - 1}$$

It follows $\beta > 0.25(\log n)^{-2}$. ∎

Chapter 11

A possible Prototype

In this section we sketch the hardware of a possible prototype. We do this step by step starting with the processor. We will not give all details here because this would take too much space, but we give an overview of the different parts.

11.1 Processor Chip

In this section we describe a realization of the RISC processor described in chapter 3.9. We plan to design the physical processor in Sea of Gates VLSI technology [27]. The registers except status register, program counter and stack pointer will be put in a separate static RAM chip. This increases the number of pins but decreases the number of gate equivalents on the chip and simplifies a processor design suitable for several sizes n. We plan to clock the processor with a frequency of 10 MHz. The execution of a virtual processor starts each cycle. Thus the pipeline cycle time is 100 ns.

The processor has 32 bits wide address and data busses to memory, a 32 bits wide data bus to the registers and a 10 bits wide address bus to the registers. The busses are used in the ways described in table 11.1.

11.2 CPU Board

We plan to place the processor on a board together with 4 memories which are shown with their sizes in table 11.2. All memories are 32 bits wide.

The prototype is planned with $n = 128$ and $c = 3$. Now $c \log n = 21$. The memories are 1280K $\times$ 32 bit each, they will be built out of dynamical RAM chips 1M $\times$ 4 bit. The register RAM will be

Cycle	address bus	data bus	address bus register	data bus register
0	program addr.	data in	addr. argument 1	result out
1	data addr.	data out	addr. Argument 2	argument 1 in
2	—	instr. in	adr. result	argument 2 in

Table11.1: Use of Address and Data Busses

Memory	size
Global segment	1M
Register RAM	$c \log n \times 32$ reg.
Local memory	$c \log n \times 64$K
Program memory	$c \log n \times 64$K

Table11.2: Memories on a CPU Board

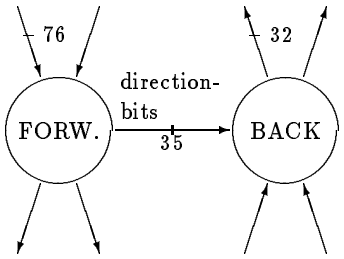


Figure 11.1: Pins of a Network Node

built with fast static RAM chips 2K $\times$ 8 bit.

The CPU board will contain the processor chip, 44 memory chips, PALs and drivers for memory control, several chips for evaluating the hash function and DMA ports to read and write memories under host control as described in section 11.4.

11.3 Network Chip

Before designing the network chip we have to determine the size of the packets. It is the sum of the size of the destination, the data and the mode and control bits. Destination and data are 32 bits wide, mode and control bits are together 12 bits. The packet size in the network back to the processor is 32 bits because here only data has to be transmitted. The input and output pins of the network node are shown in figure 11.1.

We do not want to design several different chips, therefore all inputs and outputs of the node have to be pins. A network node has indegree 2 and outdegree 2, the total number of pins is $4(76 + 32) = 432$. These are too many pins for a normal chip. Furthermore the number g of gates used is small compared to the number of pins. Several methods are possible to solve these problems:

- Send the packet in two parts. Model this feature in the simulations to test its influence on the routing time.
- Send the parts of a packet to several identical chips that select their functions by mode bits and that work together via special control signals *(bit slice)*.
- Together with bit slicing we could instead of one network node implement a 2×2 butterfly network in a chip. This uses $4g$ gates but the number of pins is only doubled. The relation is improved by a factor of 2.

We decided to take method three. The structure of the network chip is shown in figure 11.2. It reflects RANADE'S routing schmeme. We probably will choose 3 for length of the input queues and

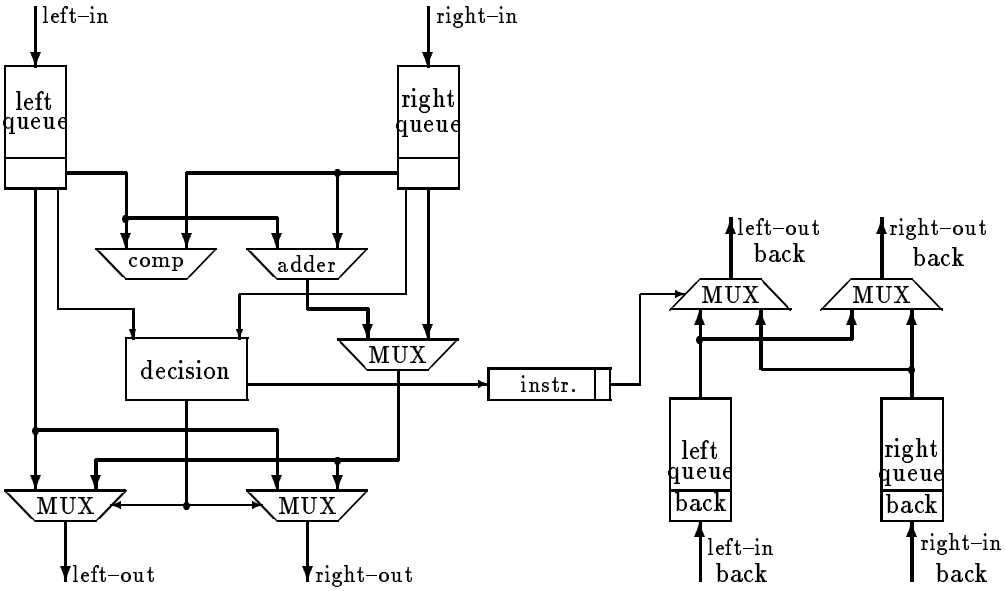


Figure 11.2: The Network Node

20 for length of the instruction queue.

11.4 Connection to the Host

As host we will choose a SUN–4 workstation with a SBus–VMEBus adapter, the connection between host and host adapter will be realized as a standard VMEBus. The host adapter will have its own microprocessor, probably a Motorola 68020, to control the input and output activities. A processor of the machine communicates with the host adapter via a special region of its local memory, that can be read and written by both adapter and processor.

In this memory part processors can store requests to the host adapter, i.e. 'Output of x words, start address A' or 'Input x words from standard input to address A'. Such a request leads to an interrupt in the host adapter to guarantee a fast reaction.

The host adapter can store status messages for the processor, i.e. 'last requested input is stored, start address A' or 'requested file could not be opened for writing'. The PRAM processors have to read the status messages explicitly, because they have no interrupt.

The host adapter communicates with the host that administrates the terminal and the disk storage. A model for this communication scheme is the operating system HCP/SCP (Host Control Program/Source Control Program) [13, 14] for the SPARK vector computer 2.0.

We send data to or from a memory segment over DMA ports and a special bus to the host adapter where the data are buffered until the communication with the host takes place. While a memory access via DMA takes place the parallel machine stops in order to avoid conflicts.

11.5 The Language

Hagerup, Schmitt and Seidl in [21] defined a PRAM language called *FORK*. The syntax and

semantic of the language is described in the paper mentioned above. Based on this language definition we started to build a compiler for our prototype.

Chapter 12

Summary

Our goal was to construct a general purpose parallel computer. We studied theoretical literature to find the best simulation of a CRCW PRAM. We found RANADE's Fluent Machine to be most suitable. We have used the framework from [37] which allows to treat computer architecture as a formal optimization problem and to deal quantitatively with hardware/software tradeoffs. In this framework we have improved the price/performance ratio of RANADE's Fluent Machine by a factor of 6. We have determined when combining should be done in hardware (namely always for practical purpose). We have compared the cost–effectiveness of PRAM's and DMM's. The results are surprisingly favourable for PRAM's. In reality things are somewhat worse, e.g. because of connectors and wires. Nevertheless we are applying for funds to build a prototype of design $D1$ with $n = 128$ physical processors. Some design decisions are already given here.

Next goal we have is to compare our PRAM design with existing (means commercial available) parallel machines such as ALLIANT FX/8 or INTEL Hypercube.

Appendix A

The Benchmark

A.1 The Algorithm

To compute the connected components for undirected graphs. We use an algorithm of [45] in a form presented in [18]. Another variant of this algorithm can be found in [20]. The formal explanation and the proofs for correctness and runtime can be found in [18] and [45]. The problem to be solved can be made precise in the following way:

Given an undirected graph $G = (V, E)$ with $v = |V|$ nodes and $m = |E|$ vertices we want to compute all elements $F[i], 0 \leq i < v$ of an array F. F represents the following function $F : V \rightarrow V$ such that for all $u, w \in V$, $F[u] = F[w]$ if and only if u and w belong to the same connected component of G. The algorithm needs time $O(\log v)$ and $\max(v, 2m)$ processors to solve this problem on a CRCW PRAM [18]. The sequential case needs time $O(v + m)$ [34] which obviously is also up to a constant factor a lower bound on sequential runtime.

Definition 16 *A parallel algorithm is said to be optimally parallized if*

parallel runtime $\times$ number of processors used $= O$(lower bound on sequential runtime).

Using definition 16 the parallel algorithm is not optimal by a factor of $\log v$. But constants are small and data structures are simple. Therefore we use this one instead of one out of [20] that is asymptotically closer to optimality but has large constants.

The graph is represented by two functions $head, tail : E \rightarrow V$ that for each vertex in E give the nodes that are connected by it. The algorithm is shown in figure A.1.

The algorithm constructs a pointer graph in array F where all nodes of G that belong to the same connected component are members of a tree in F. Step (3) reduces the height of the trees to finally 1. Procedure *starcheck* tests whether a given node of a graph belongs to a *star*. A star is a tree with maximum height of 1.

We analysed the hand compiled program for a graph with v nodes, m edges on a PRAM with P processors. The runtime is

$$2\lceil \log v \rceil \left(120 + 98\frac{v}{P} + 70\frac{m}{P} \right) + 6\frac{v}{P} + 19$$

instructions.

```
    for u ∈ V pardo F[u] := u; od;
    for t := 1 to 2 log v do
      for u ∈ V pardo change[u] := 0; od;
      starcheck;
      for alle (u, w) mit {u, w} ∈ E pardo
(1)     if star[u] and F[w] < F[u] then
          F[F[u]] := F[w];
          change[F[u]] := 1;
          change[F[w]] := 1;
        fi;
      od;
      starcheck;
      for alle (u, w) mit {u, w} ∈ E pardo
(2)     if star[u] and not change[F[u]]
                    and F[w] ≠ F[u] then
          F[F[u]] := F[w];
        fi;
(3)     F[u] := F[F[u]]
      od
    od.

proc starcheck ;
begin
  for i ∈ V pardo
    star[i] := true;
    if F[F[i]] ≠ F[i] then
      star[F[F[i]]] := false;
    fi;
    star[i] := star[F[F[i]]]
  od
end;
```

Figure A.1: Connected Components on a CRCW PRAM

A.2 Handcompiled Machine Program

We now present the hand compiled program of our benchmark. We used two assembler directives not described before. `DC` defines a constant in memory, `DS` reserves a certain amount of space in memory.

```
; ------------------------------------------------------------------------
; begin of program                          Connected Components on a CRCW PRAM
; ------------------------------------------------------------------------


.makroanf ladediv(a,b)
        push0 r31
        ldc(r31,divplace)
        ld r31,a,b
        pop0 r31
        nop()
.makroend

.include '../simneu/include.ass'        ; Contains all standard macro definitions
.include '../simneu/div.ass'            ; Contains division macro div(a,b,q,r)
                                        ; with runtime independent
                                        ; of operands a and b
.include '../simneu/stdio.ass'

.define STACKBASE0 #$80000000           ; Stack in local Memory
.define STACKBASE1 #$80000064

.define m       #2560                   ; Definition of max. graph size (edges)
.define v       #2560                   ; we assume v to be multiple
                                        ; of pnum (nodes)

.define one     #1                      ; increment and decrement constant

.define s       R27
.define f       R28                     ; Definition of array pointers
.define tf      R19
.define h       R29
.define t       R30
.define c       R31

.define i       R4                      ; number of virtual processor
.define u       R6                      ; node u
.define w       R7                      ; node w
.define fu      R9                      ; content of F[u]
.define fw      R10                     ; content of F[w]
.define ffu     R21                     ; content of F[F[u]]
.define staru   R8                      ; content of star[u]
.define chngefu R11                     ; content of change[F[u]]
.define truereg R12                     ; register for true
.define flsereg R13                     ; register for false
.define loopcnt R5                      ; counter for sequential loop
```

```
.define null    R0                      ; number zero
.define trash   R23                     ; intermediate register for computations
.define round   R24                     ; counter for number of rounds
.define pnumr   R25                     ; register that contains pnum
.define vptr    R26                     ; pointer to v
.define vr      R22                     ; register for v
.define loops   R20                     ; trash

.define fi      R9              ; content of F[i] in starcheck
.define ffi     R10             ; content of F[F[i]] in starcheck
.define sffi    R11             ; content of star[F[F[i]]] in starcheck

; Reserve space in global memory for arrays

.GDATA                          ; global data
.ORIGIN #100                    ; start address

lab_n:          .DC v           ; Value of n
                .DC m           ; Value of m
lab_s:          .DS v           ; Space for array star
lab_f:          .DS v           ; Space for array F
lab_tf:         .DS v           ;Space for temporary array F
lab_h:          .DS ( m << #1 )           ; Space for array head
lab_t:          .DS ( m << #1 )          ; Space for array tail
lab_c:          .DS v           ; Space for array change
divplace:       .ds #2          ; Space for v/pnum and 2m/pnum

.CODE                           ; Beginn of program
.ORIGIN #0                      ; Start address

; Initialization of constants
        ldc     (truereg,true)  ; load truereg with true
        ldc     (flsereg,false) ; load falsereg with false

; Initialization of stackpointers
        ldc     (SP0,STACKBASE0)
        ldc     (SP1,STACKBASE1)

        ldc     (r31,sync_lab)  ; compute pnum
        st      r31,#0,r0
        ldc     (r30,#1)
        sync    plus,r31,#0,r30
        ld      r31,#0,pnumr

; Initialization of array pointers

        ldc     (vptr,lab_n)
        ld      vptr,#0,vr
        ldc     (s,lab_s)
        ldc     (f,lab_f)
        ldc     (tf,lab_tf)
        ldc     (h,lab_h)
```

```
        ldc     (t,lab_t)
        ldc     (c,lab_c)

        push0   r31
        ldc     (r31,divplace)
        div     (vr,pnumr,round,null)   ; compute v/pnum
        st      r31,#0,round
        LD      vptr,one,trash          ; load m to trash
        nop()                           ; delayed
        LSL     trash,one,trash         ; multiply with 2 by shifting
        div(trash,pnumr,round,null)
        st r31,#1,round
        pop0 r31
        nop()

; 1st parallel construct, number of rounds round=v/pnum

        LDI     i               ; load number of virtual processor
        ladediv(#0,round)       ; load v/pnum to round
in0:    SUB     round,one,round ; compute outer loop
        JMP     negset,out0     ; end of outer loop

                                ; i contains number of program processor which is
                                ; number of virt.proc.+act.round*pnum

                                ; execute F[u]:=u and change[u]:=0

        ST      f,i,i           ; store progr.proc.number to location f+i,
                                ; thus F[u]:=u
        ST      c,i,null        ; store null=0 to location c+i, thus change[u]:=0

        ADD     i,pnumr,i       ; add pnum to i to compute progr.proc.numb. for
                                ; next round
       JMP     in0

out0:                           ; End of 1st parallel construct no sync necessary

; FOR loop, compute $2+log_{3/2} v as $\lceil 2*log v\rceil$

        LD      vptr,null,loopcnt       ; load loopcnt with v
        nop()                           , delayed
        RM      loopcnt,loopcnt         ; compute number of highest bit set
        ADD     loopcnt,one,loopcnt     ; round up
        LSL     loopcnt,one,loopcnt     ; multiply with 2 by shifting

loop:   SUB     loopcnt,one,loopcnt     ; program loop
        JMP     negset,endloop          ; jump to end of program

        JSR0    starcheck               ; call procedure starcheck
; reset array change
        LDI     i
        ladediv(#0,round)
```

```
beginx: sub     round,#1,round
        JMP     negset,endx
        ST      c,i,flsereg
        ADD     i,pnumr,i
        JMP     beginx
endx:

; compute number of rounds r=2m/pnum, consider m a multiple of pnum

; init temporary array for F
        LDI     i
        ladediv(#0,round)
begin1: SUB     round,one,round
        JMP     negset,end1
        LD      f,i,trash
        nop()                           ; delayed load nop
        ST      tf,i,trash              ; copy
        ADD     i,pnumr,i
        JMP     begin1
end1:

        LDI     i                       ; load number of virtual processor
        ladediv(#1,round)
in1:    SUB     round,one,round         ; compute outer loop
        JMP     negset,out1             ; end of outer loop

                                ; i contains number of program processor which is
                                ; number of virt.proc.+act.round*pnum

; 2nd parallel construct, compute (u,w)

        LD      h,i,u           ; load head[i] to u
        LD      t,i,w           ; load tail[i] to w

; IF statement (1)

cont1:                          ; evaluate IF condition
        LD      f,u,fu          ; load F[u] to fu
        LD      f,w,fw          ; load F[w] to fw
        LD      s,u,staru       ; load star[u] to staru
        cmp     (fw,fu)         ; test whether F[w] > F[u]
        JMP     negclear,fail1  ; if F[w] >= F[u] stop continuing
        cmp     (staru,false)   ; test whether star[u] is false
        JMP     zeroset,fail2   ; if star[u] = false stop continuing

; IF condition is true execute THEN part

        ST      tf,fu,fw         ; store F[v] to F[F[u]] to temporary array
        ST      c,fu,truereg    ; set change[F[u]] to true
        ST      c,fw,truereg    ; set change[F[w]] to true
        JMP     cont2           ; jump to end of round
```

```
fail1:  nop()                    ; compensate test for star[u]= false with 2 NOP
        nop()
fail2:  nop()                    ; compensate time for then part with 6 NOP
        nop()
        nop()
        nop()

cont2:  ADD i,pnumr,i           ; add pnum to i to compute progr.proc.numb. for
                                ; next round
        JMP in1

out1:                           ; End of 2nd parallel construct no sync necessary

; restore array F
        LDI     i
        ladediv(#0,round)
begin2: SUB     round,one,round
        JMP     negset,end2
        LD      tf,i,trash
        nop()                           ; delayed load nop
        ST      f,i,trash               ; copy
        ADD     i,pnumr,i
        JMP     begin2
end2:

        JSR0    starcheck               ; call procedure starcheck

; temporary array F is still actual

; compute number of rounds r=2m/pnum, consider m a multiple of pnum

        LDI     i                       ; load number of virtual processor
        ladediv(#1,round)
        ADD     round,null,loops        ; save round to loops
in2:    SUB     round,one,round         ; compute outer loop
        JMP     negset,out2             ; end of outer loop

                                ; i contains number of program processor which is
                                ; number of virt.proc.+act.round*pnum

; 3rd parallel construct, compute (u,w)

        LD      h,i,u           ; load head[i] to u
        LD      t,i,w           ; load tail[i] to w

; IF statement (2)

cont3:                          ; evaluate IF condition

        LD      f,u,fu          ; load F[u] to fu
        LD      f,w,fw          ; load F[w] to fw
        LD      s,u,staru       ; load star[u] to staru
```

```
        LD      c,fu,chngefu    ; load change[F[u]] to chngefu
        cmp     (fw,fu)         ; test whether F[w] = F[u]
        JMP     zeroset,fail3   ; if F[w] = F[u] stop continuing
        cmp     (staru,false)   ; test whether star[u] is false
        JMP     zeroset,fail4   ; if star[u] = false stop continuing
        cmp     (chngefu,true) ; test whether change[F[u]] is true
        JMP     zeroset,fail5   ; if change[F[u]] = true stop continuing

; IF condition is true execute THEN part

        ST      tf,fu,fw         ; Store F[w] to F[F[u]] to temporary array
        JMP     cont4           ; jump to end of round
fail3:  nop()                     ; compensate test for star[u]= false with 2 NOP
        nop()
fail4:  nop()                     ; compensate test for change[F[u]] = false
        nop()
fail5:  nop()                     ; compensate time for then part with 2 NOP
        nop()

cont4:
        ADD     i,pnumr,i       ; add pnum to i to compute progr.proc.numb. for
        JMP     in2             ; next round

out2:                           ; End of 3rd parallel construct no sync necessary
; restor array F
        LDI     i
        ladediv(#0,round)
begin7: SUB     round,one,round
        JMP     negset,end7
        LD      tf,i,trash
        nop()                             ; delayed load nop
        ST      f,i,trash               ; copy
        ADD     i,pnumr,i
        JMP     begin7
end7:

; temporary array is still up to date

        LDI     i
        ladediv(#0,round)
begin5: SUB     round,one,round
        JMP     negset,end5
        LD       f,i,fu          ; load F[u] to fu
        nop()
        LD      f,fu,ffu        ; load F[F[u]]n
        nop()                   ; delayed load nop
        ST       tf,i,ffu        ; Store F[F[u]] to F[u] in temporary array
        ADD     i,pnumr,i
        JMP     begin5
end5:

; restore array F
```

```
        LDI     i
        ladediv(#0,round)
begin6: SUB     round,one,round
        JMP     negset,end6
        LD      tf,i,trash
        nop()                           ; delayed load nop
        ST      f,i,trash               ; copy
        ADD     i,pnumr,i
        JMP     begin6
end6:
        JMP     loop            ; jump back to FOR loop

endloop:JMP     endloop         ; end of program

starcheck:                      ; code for procedure starcheck

        LDI     i
        ladediv(#0,loops)
        ADD     loops,null,round
begin3: SUB     round,one,round
        JMP     negset,end3
        ST      s,i,truereg             ; star[i]:=true
        ADD     i,pnumr,i
        JMP     begin3
end3:

        LDI     i               ; load number of virtual processor
        ADD     loops,null,round ; number of rounds
in3:
        LD      f,i,fi          ; load F[i] to fi
                                ; belongs to evaluation of condition in IF
        SUB     round,one,round ; compute outer loop
        JMP     negset,out3     ; end of outer loop

                                ; i contains number of program processor which is
                                ; number of virt.proc.+act.round*pnum

        LD      f,fi,ffi        ; load F[F[i]] to ffi
                                ; belongs to evaluation of condition in IF

        nop()                   ; delayed load nop

; IF statement evaluate condition

        cmp     (fi,ffi)        ; test whether fi <> ffi
        LD      s,ffi,sffi      ; load star[F[F[i]]] to sffi
                                ; belongs to next statement
        JMP     zeroset,fail6   ; if fi=ffi stop continuing

; condition is true execute THEN part
```

```
        ST      s,ffi,flsereg   ; store false to star[F[F[i]]]
        JMP     cont5           ; jump to next statement

fail6:  nop()                     ; compensate time for then part by 2 NOP
        nop()
cont5:
        ADD     i,pnumr,i       ; add pnum to i to compute progr.proc.numb. for
                                ; next round
        JMP     in3

out3:                           ; End of parallel construct no sync necessary

        LDI     i
        ADD     loops,null,round        ;in round einladen
begin4: SUB     round,one,round
        JMP     negset,end4
        LD      f,i,fi
        nop()                           ; delayed load nop
        LD      f,fi,ffi
        nop()
        LD      s,ffi,sffi
        nop()
        ST      s,i,sffi                ; star[i]:=star[F[F[i]]]
        ADD     i,pnumr,i
        JMP     begin4
end4:
        rts0 ()                   ; return from subroutine
        nop()                     ; nop necessary since PC has to be
                                  ; reloaded correctly
```

; ---

; end of program

; ---

Bibliography

[1] Ferri Abolhassan and Jörg Keller. Detailed analysis of PRAM machines. Manuscript, July 1990.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[3] M. Ajtai and E. Komlós, J. andSzemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Annual Symposium on Theory of Computing*, pages 1–9, New York, 1983. ACM.

[4] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice–Hall, 1989.

[5] R. Alelunias. Randomized parallel communication. In *Proceedings of the ACM SIGACT–SIGTOPS Symposium on Principles of Distributed Computing*, pages 60–72, August 1982.

[6] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference, Vol. 32*, pages 307–314, Reston, Va., 1968. AFIPS Press.

[7] Y. Chang and J. Simon. Continuous routing and batch routing on the hypercube. In *5th ACM Symposium on Principles of Distributed Computing*, pages 272–281, 1986.

[8] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[9] Martin Dietzfelbinger. Hashing modulo powers of two. Personal Communication, October 1990.

[10] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. How to distribute a dictionary in a complete network. Reihe Informatik Bericht Nr. 68, Universität–GH Paderborn, April 1990.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. Reihe Informatik Bericht Nr. 67, Universität–GH Paderborn, April 1990.

[12] P.W. Dymond and W.L. Ruzzo. Parallel RAMs with owned global memory and deterministic context–free language recognition. In *ICALP 1986, Automata, Languages and Programming*, pages 96–104, 1986.

[13] Arno Formella. minhcp spark 2.0 minimal host control program. Universität des Saarlandes, 1988.

[14] Arno Formella. Scp v1.3 spark 2.0 control program. Universität des Saarlandes, 1988.

[15] Arno Formella. *Effizienz numerischer Rechnerarchitekturen.* PhD thesis, Universität des Saarlandes, 1991. to appear.

[16] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[17] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *Proceedings of the International conference on Parallel Processing*, 1981.

[18] Torben Hagerup. Parallele Algorithmen. Lecture script, Universität des Saarlandes, 1989.

[19] Torben Hagerup. Randomized simulation of prams on processor networks. Lecture script, Universität des Saarlandes, 1989.

[20] Torben Hagerup. Optimal planar algorithms on planar graphs. *Information and Computation*, 84:71–96, 1990.

[21] Torben Hagerup, Arno Schmitt, and Helmut Seidl. FORK: A high–level–language for PRAMs. In *Proceedings of PARLE 91*, 1991.

[22] R.W. Hockney and C.R. Jesshope. *Parallel Computer 2.* Adam Hilger, Bristol and Philadelphia, 1988.

[23] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw–Hill, 1984.

[24] Toshiya Itoh and Shigeo Tsujii. Structure of parallel multipliers for a class of fields GF(2^m). *Information and Computation*, 83:21–40, 1989.

[25] H.J. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. Research report, IBM Research Division, 1988.

[26] Richard M. Karp and Viaya L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, 1990.

[27] Reiner Kolla, Paul Molitor, and Hans Georg Osthoff. *Einführung in den VLSI–Entwurf.* Teubner, 1989.

[28] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

[29] F.T. Leighton and Charles E. Leiserson. Theory of parallel and VLSI computation. Lecture notes, research seminar series, Massachusetts Institute of Technology, May 1990.

[30] F.T. Leighton, Bruce Maggs, and Satech Rao. Universal packet routing algorithms. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[31] F.T. Leighton, Bruce Maggs, and Satech Rao. Personal communication. The results of their simulations are not yet published, 1990.

[32] Werner Massonne. Schnelle Routing–Algorithmen auf Butterfly Netzwerken. Master's thesis, Universität des Saarlandes, 1990.

[33] Kurt Mehlhorn. *Data Structures and Algorithms*, volume 1, Sorting and Searching. Springer, 1984.

[34] Kurt Mehlhorn. *Data Structures and Algorithms*, volume 2, Graph Algorithms and NP–Completeness. Springer, 1984.

[35] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[36] Motorola, Inc. ASIC Division, Chandler, Arizona. *Motorola High Density CMOS Array Design Manual*, July 1989.

[37] Silvia M. Müller and Wolfgang J. Paul. Towards a formal theory of computer architecture. In *Proceedings of PARCELLA 90, Advances in Parallel Computing*. North–Holland, 1990.

[38] M.S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5:75–92, 1990.

[39] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, 1982.

[40] Nick Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, pages 127–136, 1984.

[41] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[42] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnson. The Fluent Abstract Machine. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, 1988.

[43] J.H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *Journal of the Association of Computer Machinery*, 34(1):60–76, January 1987.

[44] P. Ribenboim. *The Book of Prime Number Records*. Springer, 2nd edition, 1989.

[45] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[46] Alan Siegel. On universal classes of fast high performance hash functions, their time–space tradeoff, and their applications. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 20–25, 1989.

[47] B.J. Smith. A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Comference on Parallel Processing*, pages 6–8. IEEE, 1978.

[48] Eli Upfal. Efficient schemes for parallel communication. In *Proceedings of the ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 55–59, August 1982.

[49] Eli Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st ACM Annual Symposium on Theory of Computing*, pages 241–250, May 1989.

[50] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[51] Leslie G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.

[52] Leslie G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th ACM Annual Symposium on Theory of Computing*, pages 263–277, May 1981.

[53] Alf Wachsmann. Eine theoretische und experimentelle Untersuchung von Emulationsalgorithmen eines gemeinsamen Speichers auf einem Butterfly–Netzwerk. Master's thesis, Universität Dortmund, 1990.

[54] Ingo Wegener. *The Complexity of Boolean Functions*. Teubner, 1987.

[55] C.-L. Wu and T.-Y. Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, C–29(8):694–702, August 1980.