On the Cost–Effectiveness of PRAMs[*]

Ferri Abolhassan[1] Jörg Keller[2] Wolfgang J. Paul[3]

[1]SAP Retail Systems, Neue Bahnhofstraße 21, 66386 St. Ingbert, Germany

Phone/Fax: +49-6894-981-0/199, Email: Ferri.Abolhassan@sap-ag.de

[2]FernUniversität-GH, FB Informatik, 58084 Hagen, Germany

Phone/Fax: +49-2331-987-376/308, Email: Joerg.Keller@FernUni-Hagen.de

[3]Universität des Saarlandes, FB Informatik

Postfach 151150, 66041 Saarbrücken, Germany

Phone/Fax: +49-681-302-2436/4290, Email: wjp@cs.uni-sb.de

Abstract. We introduce a formalism which allows to treat computer architecture as a formal optimization problem. We apply this to the design of shared memory parallel machines. While present parallel computers of this type only support the programming model of a shared memory but often process simultaneous access by several processors to the shared memory sequentially, theoretical computer science offers solutions for this problem that are provably fast and asymptotically optimal. But the constants in these constructions seemed to be too large to let them be competitive. We modify these constructions under engineering aspects and improve the price/performance ratio by roughly a factor of 6. The resulting machine has surprisingly good price/performance ratio even if compared with distributed memory machines. For almost all access patterns of all processors into the shared memory, access is as fast as the access of only a single processor.

[*]This research was done while all authors were working at Universität des Saarlandes. This research was partly supported by the German Science Foundation (DFG) under SFB 124, TP D4. A preliminary Version of this article appeared in the *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, Dec. 1991.

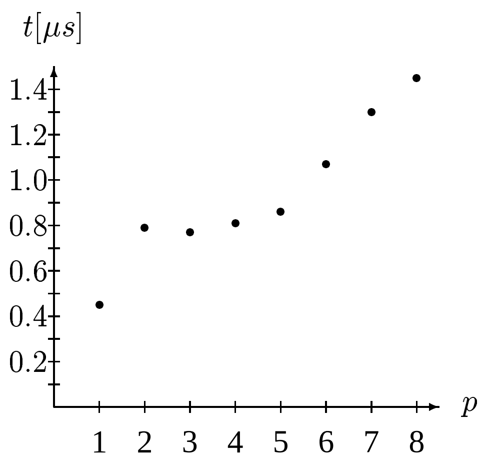


Figure 1: Concurrent Write on ALLIANT FX/2816

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[1] 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 large variation of the memory access time depending on the access patterns of the processors. In fact a single concurrent write of all say p processors of a parallel machine to the same memory location might very well be slower than p accesses of a single processor to its local memory. As an example figure 1 shows the time of a concurrent write by $p = 1, \ldots, 8$ processors to the same memory location in an ALLIANT FX/2816. Thus present shared memory machines support only the programming model but not the timing behaviour of a true shared memory.

[1]Notable exceptions are Tera MTA and Cray T3E [5, 34].

Parallel machines which support both the programming model and the timing behaviour of true shared memory are called PRAMs in the theoretical literature. The problem of simulating PRAMs by more technically feasible models has been extensively studied [4, 8, 12, 21, 25, 26, 32, 37, 38, 39]. The construction from [32], called the Fluent Machine, is considered a promising candidate because of its combined simplicity and efficiency.

We will describe the design of a reengineered version of the Fluent Machine. We will review a formalism from [28] which permits to compare cost–effectiveness of architectures. It will turn out that the reengineered version of the Fluent Machine is more than 5 times more cost–effective than the original machine and that it is surprisingly cost–effective when compared to distributed memory machines.

In section 1.1 we define the formalism to compare machines. Section 1.2 describes the theoretical PRAM model and principles of emulations on more realistic machines. Chapter 2 contains the description of the Fluent machine and the reengineered version. In chapter 3 we analyze both machines and compare them in the formalism given in section 1.1. In chapter 4 we show that it is worthwhile to support concurrent accesses by hardware. In chapter 5 we compare PRAMs and distributed memory machines.

1.1 Comparison of Machines

Definition 1 *Let D be a design of a machine with cost 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.*

A motivation for the `TDC` is the well–known 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 gates has basic cost and delay given by functions *cost* and *delay*. The values are normalized relative to the cost (resp. delay) of an inverter. Examples are shown in table 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 and static RAM can be packed

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

Table 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

Table 2: Packing Factors

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 [27] which are shown in table 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 $\rho_A \cdot (8\mathit{cost}(\texttt{OR}) + 16\mathit{cost}(\texttt{AND}) + 16\mathit{cost}(\texttt{EXOR})) = 108$.

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 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 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 $T_{total} = 7\mathit{delay}(\texttt{AND}) + 7\mathit{delay}(\texttt{OR}) + 2\mathit{delay}(\texttt{EXOR}) = 20$.

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 [18].

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.

Definition 2 *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* $\texttt{TDC}(D0, B) < \texttt{TDC}(D1, B)$.

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 depend on the number of processors. To compare the families we take corresponding ”representatives” of them. These will be members of the two families that have equal processor numbers. By this, both will require the same degree of parallelism in the benchmark.

1.2 The PRAM Model and Emulation

The PRAM model was introduced by FORTUNE and WYLLIE [15], we will briefly sketch the features important for our work.

Definition 3 *An 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 which is polynomial in n. In each step each processor can work as a separate register machine or can access a cell of the shared memory. The processors work synchronously.

We consider the following kinds of PRAMs:

EREW: *(exclusive read exclusive write)* a memory cell cannot be accessed simultaneously by several processors.

CREW: *(concurrent read exclusive write)* It is only possible to read a cell simultaneously.

CRCW: *(concurrent read concurrent write)* Processors can read or write a cell simultaneously (nothing is specified about simultaneous reads and writes). 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. Overviews about algorithms for the different models can be found in [3, 16, 22].

One simulates an n–PRAM on a multi–computer machine (MIMD) by distributing the shared memory uniformly among memory modules $M_0, \ldots, M_{n-1}$ each of size m/n. Processors and memory modules are connected by an interconnection network. If processor P_i wants to access a memory cell that is stored in module M_j, P_i sends a packet to M_j specifying the required memory cell. In case of a `LOAD` instruction M_j sends the content of that cell back to P_i.

In order to map the address space onto the memory modules one uses a hash function $g : \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$. One would rather expect a pair (h, l) of functions where $h : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$ specifies the module and $l : \{0, \ldots, m-1\} \to \{0, \ldots, (m/n)-1\}$ specifies the location within the module. One gets h and l from g by

$h(x) = g(x) \bmod n$, $l(x) = g(x)$ div n. Binary representations for $h(x)$ und $l(x)$ can be easily obtained from the binary representation of $g(x)$ by taking the $\log n$ least significant bits and the $\log(m/n)$ most significant bits respectively.

The communication between processors and memory modules can be handled by packet routing on the chosen interconnection network.

The time to simulate one step of the PRAM depends on the memory congestion c_m (the maximum number of packets that one memory module receives) and the network latency (for which the diameter of the interconnection network is a lower bound). If we restrict to constant degree networks this diameter is at least $\log n$. This implies that it is sufficient to demand $c_m = O(\log n)$.

Hash functions that distribute provably well are examined in [12, 21, 26]. Provably well here means that for each n–tuple of distinct addresses (the cells accessed by the processors in this step) the module congestion is $c_m = O(\log n)$ with very high probability. An example are randomly chosen polynomials of degree $O(\log n)$. Simulations [13, 32] indicate that for practical use particular linear hash functions g of the type $g(x) = ax \bmod m$ where m is a power of two, greatest common divisor gcd(a, m)=1, $a \in \{0, \ldots, m-1\}$ randomly chosen, are good enough. The advantages of the function $g(x)$ are its bijectivity and the short evaluation time. In this case, the definition of h and l has to be changed to $h(x) = g(x)$ div m/n and $l(x) = g(x) \bmod m/n$ [11].

Constant degree networks with diameter $\log n$ are for example butterfly networks. Routing algorithms for these networks that handle $\log n$–relations (at most $\log n$ packets go to the same module) in time $O(\log n)$ are presented in [25, 31]. The latter algorithm also handles concurrent access to the same cell by combining packets.

The simulation so far causes a slowdown of $O(\log n)$, because one step of the PRAM takes constant time but one step of the simulation takes time $O(\log n)$. We overcome this by increasing the number of processors and memory modules of the simulating machine to $n' = n \log n$. The time for one step now is $O(\log n') = O(\log(n \log n))$ which is still $O(\log n)$. But the number of necessary steps has reduced by a factor of $O(\log n)$ if we assume that the problem to be solved has enough parallelism to keep $n \log n$ processors running. This reduces the slowdown to $O(1)$.

We base our work on RANADE's Fluent Machine as described in section 2.1 that uses the routing algorithm mentioned above and polynomials for hashing.

2 The Machine D1

We first give a short summary of the Fluent Machine which is precicely described in [31, 32]. Then we present some improvements that lead to our design $D1$.

2.1 The Fluent Machine

The Fluent Abstract Machine simulates a CRCW priority PRAM with $n \log n$ processors. The processors are interconnected by an $n \log n$ butterfly network as given in Definition 4.

Definition 4 *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.*

Each network node contains a processor, a memory module of the shared memory and the routing switch. If a processor $\langle col, row\rangle$ wants to access a variable V_x it generates a packet of the form (destination,type,data) where destination is the tuple $(h(x), l(x))$ and type is `READ` or `WRITE`. This packet is injected into the network and sent to node $h(x) = \langle row', col'\rangle$ and back (if its type is `READ`) with the following six phase deterministic packet routing algorithm.

1. The packet is sent to node $\langle \log n, row\rangle$. On the way to column $\log n$ all packets injected into a row are sorted by their destinations.

2. The message is routed along the unique path from $\langle \log n, row\rangle$ to $\langle 0, row'\rangle$. The routing algorithm used is given in [31].

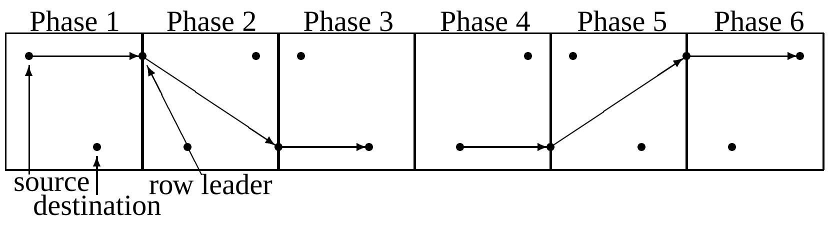


Figure 2: 6 phase routing of the Fluent Machine

3. The packet is directed to node $\langle col', row' \rangle$ and there the memory access takes place.

4. – 6. The packet is sent the same way back to $\langle col, row \rangle$.

Figure 2 shows the phases performed on a network consisting of 6 butterflies. RANADE realizes these six phases with two butterfly networks where column i of the first network corresponds to column $\log n - i$ of the second one. Phases 1,3,5 use the first network, phases 2,4,6 use the second network. Thus the Fluent Machine consists of $n \log n$ nodes each containing one processor, one memory module and 2 network switches.
The reason for sorting in phase 1 is given in section 2.2.

2.2 Combining

In a CRCW PRAM several (possibly all) processors could access the same cell with address x_j at the same time. Let

$$S_j = \{P_i | P_i \text{ reads } x_j \text{ in current step}\},$$

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

We talk only of `READ` accesses because `WRITE` accesses can be treated in a similar way with the simplification that they do not return an answer to the processor.
If all packets in PAC_j reach memory module $h(x_j)$, the module congestion c_m equals $|PAC_j|$. In the worst case this could be n. Because the routing algorithms require module congestion $O(\log n)$ (see last section) the number of packets in PAC_j that reach $h(x_j)$ 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 future packets of the current step must originate from processors $P \notin S_j$), then the two packets can be merged and one forwarded along the tree. This merging is called *combining*.
In order to decide whether two incoming packets $pac_1 \in S_i, pac_2 \in S_j$ have to be combined, a network node has to compare the destinations $g(x_i)$ and $g(x_j)$.
How can a network node know that no more packets will arrive in the future? RANADE gives in [31] the following solution: sort the packets during phase 1 by their destinations and then maintain for each node the sorted order of the packets that leave the node.

2.3 Improvements

Definition 5 *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 finished 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 2. But the networks for phases 1 and 6 can be realized by n sorting arrays of length $\log n$ as described in [1, 24] 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 3. 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 by

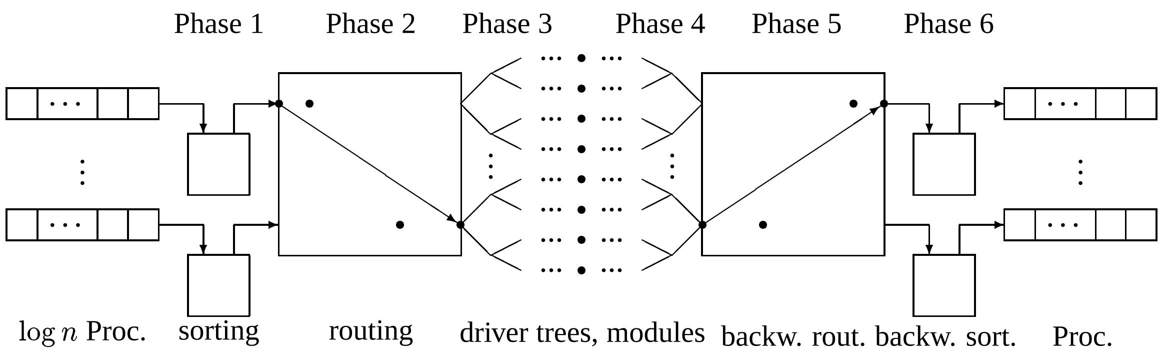


Figure 3: 6 phase Routing in the New Machine

replacing the $\log n$ processors of a row by only one physical processor (pP) which simulates the original $\log n$ processors as virtual processors (vP). Another advantage of this concept is that we can increase the total number of PRAM processors by simulating $X = c \log n$ (with $c > 1$) vP's in a single pP. The simulation of the virtual processors by the physical processor is done by the principle of *pipelining*. This principle is well known from vector computers and was also used in the first MIMD computer marketed commercially, the Denelcor HEP [20, 36]. A closely related concept is *Bulk Synchronous Parallism* in [39].

In vector processors the execution of several instructions is overlapped by sharing the ALU. Figure 4 shows how pipelining is 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. 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 simply by their own register sets. We save the costs of $X - 1$ ALU's.

The depth δ of this pipeline serves to hide network latency. This latency is proved to be $c \log n$ for some c with high probability [31]. Thus, if $\delta = c \log n$, then normally no vP has to wait for a returned packet. This c increases the number of vP's and the network conguestion. But network latency only grows slowly with increasing c. Thus there exists an optimal c. The exact value and its influence on the length of the sorting arrays is discussed

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

Figure 4: Pipelining in the Processor

in section 2.4. VALIANT calls this *parallel slackness* [38].

Definition 6 *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 on the one hand still packets of this round in the network, on the other hand the processors have to proceed (and thus must start the next round) to return these packets. CHANG and SIMON prove in [9] that this works and that the latency still is $O(\log n)$. The remaining problem how to separate 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 separate rounds.

2.4 Delayed LOAD and Sorting

One problem to be solved is that virtual processors that execute a LOAD instruction have to wait until the network returns the answer to their READ packets. Simulations indicate, that for $c = 6$ this works most of the time (see [1]). But this is quite large in comparison to $\log n$. We partially overcome this by using delayed LOAD instructions as in [30]. We require an answer to a READ packet to be available not in the next instruction but in the next but one. Investigations show that insertion of additional 'dummy' instructions happens very rarely [19]. But if a program needs any dummy instructions, they can be easily inserted by the compiler. This reduces c to 3 without significantly slowing down the machine.

The sorting arrays should have length $c \log n$ too. But breaking a round in z parts is an alternative. This reduces the lengths to $(c/z) \cdot \log n$ but could slow down the machine's speed. Simulations show [1] that $z = 4$ is the maximum value that does not slow down speed if we double the sorting networks. The doubling garantuess that always one sorting array can be filled while the other sends packets into the butterfly network. Therefore we choose this value.

In order to examine the exact constants for runtime and costs in machine $D1$ by the method sketched in section 1.1 we have to model the processor for this machine. In [32] nothing special about it is mentioned except that the use of RISC processors is proposed.

2.5 A Processor

We use a processor similar to the Berkeley RISC processor [30]. Instead of register windows we have the register sets of the virtual processors. The processor has a LOAD–STORE architecture, i.e. COMPUTE instructions only work on registers and immediate constants and memory access only happens on LOAD and STORE instructions. The COMPUTE instructions involve adding, multiplying, shifts and bit oriented operations. All instructions need the same amount of time. In order to get a pipeline of depth $c \log n$, the ALU depth is increased artificially.

Because of the LOAD–STORE architecture the same multiplier can be used for multiplications in COMPUTE instructions and for hashing global addresses with a linear hash function

in `LOAD` and `STORE` instructions. This means that hashing does not require much special hardware.
A more detailed description of the processor can be found in [23].

3 Cost and Speed

3.1 Cost of the machine

We compute the costs of the improved Fluent Machine with the method introduced in section 1.1. We will ignore control logic because it usually 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 2 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 [40]. The register file of the Fluent Machine consists of 16 registers each 32 bits wide, the one in the improved machine consists of $c \log n \times 16$ registers each 32 bits wide. Let the basic costs of the ALU be A and the basic costs of the Fluent Machine's register file be F. If we use the packing factors of table 2 we have costs $c_P = \rho_A A + \rho_L c \log n F$ for the processor of our design $D1$ and $c_{\tilde{P}} = \rho_A A + \rho_S F$ for the Fluent Machine's processor.
Simulations [1] indicate that network nodes need buffers of length 2. A node consists of 2 buffers and 2 multiplexers on the way from processors to memory, 2 buffers and 2 multiplexers on the way back, a direction queue of length $2c \log n$ and a comparator and a subtractor to compare addresses. Sorting nodes 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 $2 \cdot (c/4) \cdot n \log n$ sorting nodes and of $n \log n$ network nodes. It has total costs

$$c_{D1} = nc_P + c_S \frac{2c}{4} n \log n + c_N n \log n.$$

A	F	N_A	N_S	S_A	S_S
13572	6144	2576	6696	1928	4104

Table 3: Actual Parameters

The exact numbers for A, F, N_A, N_S, S_A, S_S are shown in table 3, the computation can be found in appendix A. The result is

$$c_{D1} = 10179n + 15598n \log n. \tag{1}$$

The Fluent Machine's network nodes have slightly larger basic costs $\tilde{N}_A = 3104, \tilde{N}_S = 8808$ because RANADE's routing algorithm needs full routing information in forward and backward network. $c_{\tilde{N}}$ is computed in analogy to c_N. The costs of the Fluent Machine then are

$$c_{D0} = c_{\tilde{P}} \cdot n \log n + c_{\tilde{N}} \cdot n \log n = 19235.4n \log n.$$

For $n = 128$ the Fluent Machine is 1.128 times more expensive than our improved machine.

3.2 Speed of the Machine

In section A.3 we compute the maximal delay path in network nodes. We get a minimal cycle time of $\sigma_N = 30$ gate delays for the network and sorting nodes. For a particular processor design in [23] we computed a minimal cycle time $\sigma_P = 60$ gate delays, which comes from access times to the register file. In current VLSI technology with gate delays of $2ns$ we get cycle times of $120ns$ and $60ns$.

One step of the improved machine takes $c \log n$ processor cycles which is $\nu = c \log n \times 120ns$. RANADE reports in [32] simulation results such that one step of the Fluent machine takes $11 \log n$ network cycles which is $\nu' = 11 \log n \times 60ns$.

The improved machine then has a power of $(cn \log n)/\nu$ Instructions per second. For $n = 128$ we get 1066 MIPS. The corresponding value for the Fluent Machine is 193 MIPS. Thus the improved machine 5.5 times faster and 6.2 times more cost–effective than the Fluent Machine.

In order to have the same number of virtual processors, we also investigate a modified Fluent Machine $\tilde{D0}$ with $N = (kn) \cdot \log(kn)$ processors. We choose k such that $N = cn \log n$. Then $c_{\tilde{D0}} = N \cdot (c_{\tilde{P}} + c_{\tilde{N}} = 19235.4N$. One step of $\tilde{D0}$ takes $\tilde{\nu} = 11 \log(kn) \times 60ns$. A benchmark B with sequential runtime T that can be parallelized with efficiency ϵ will need $T/(\epsilon N)$ steps on both machines. Then, for $n = 128$, machine $D1$ is only $t_{\tilde{D0}}/t_{D1} = \tilde{\nu}/\nu = 2.18$ times faster but $c_{\tilde{D0}}/c_{D1} \cdot \tilde{\nu}/\nu = 7.38$ times more cost-efficient than $\tilde{D0}$.

4 CRCW vs. EREW

4.1 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 1.

Theorem 1 *Let $D1$ be a CRCW PRAM as described in section 2.3 which supports combining by hardware. Let $D2$ be an EREW PRAM as described in section 4.2 on which each concurrent access is simulated by software as described in section 4.3. If a benchmark B that needs t_{D1} steps consists of αt_{D1} concurrent accesses with $0 \leq \alpha \leq 1$ then*

$$\mathtt{TDC}(D1, B) < \mathtt{TDC}(D2, B) \quad \textit{for} \quad \alpha > 0.117 \frac{1}{(\log n)^2}.$$

This means: if a benchmark that needs t_{D1} steps consists of more than $0.117(\log n)^{-2} t_{D1}$ concurrent accesses it is better to run it on a CRCW PRAM instead of simulating it on an EREW PRAM.

4.2 Design of an EREW PRAM

To determine $\mathtt{TDC}(D2, B)$ it is necessary to sketch the design of an EREW PRAM $D2$. We get $D2$ 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 $c' = 1.5$. The costs for the new processors are $c_{P'} = \rho_A A + \rho_L c' \log n F = 10179 + 2857 \log n$. The costs for network and sorting nodes decrease from N_A to $N'_A = 2320$ and N_S to $N'_S = 6192$ as shown in appendix A. The total costs for $D2$ are

$$c_{D2} = c_{P'} n + c_{N'} n \log n = 10179n + 7389.4n \log n. \tag{2}$$

The cycle time of $D2$ is exactly the same as of $D1$, one step of $D2$ takes $c' logn$ processor cycles.

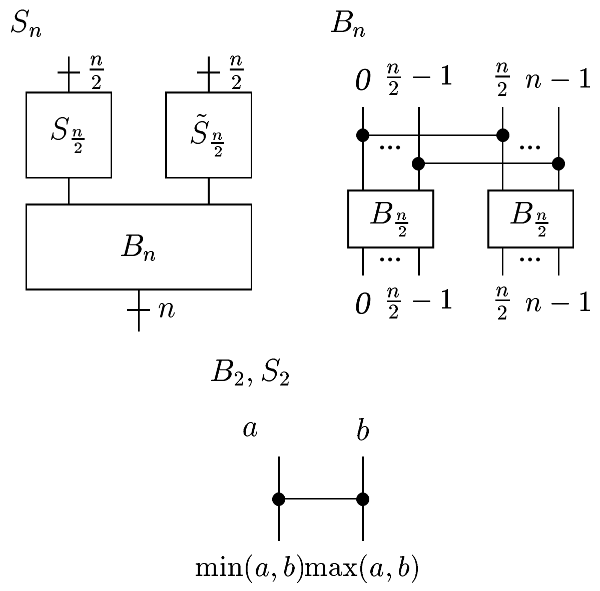
4.3 Simulation of CRCW on EREW

KARP and RAMACHANDRAN show in [22] 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 $n-1$ get sorted now 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 duplicate now 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 [2, 10, 29], a randomized one is given in [33]. The constant factor in their runtime however is quite large. We will use BATCHER's *bitonic sort* [7], 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 7.

Definition 7 *B_2 and S_2 are identical circuits sorting two numbers. B_n is a circuit that merges two bitonic sequences each of length $n/2$ 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 5.

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. 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 $D2$ $n' \geq c' n \log n$. The complete simulation of one step then takes

$$t_{sim} = \frac{19}{2}\,(\log n')^2 + \frac{47}{2} \log n' + 46. \tag{3}$$

```
**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 5: Bitonic Sort Algorithm

instructions. The complete analysis of the assembler program can be found in [23]. Now we will prove theorem 1 using the results of the previous subsections.

Proof: (indirect) Let B be a benchmark that needs time t_{D1} on $D1$. On $D2$ it will need time

$$t_{D2} = t_{D1}\left(\alpha t_{sim} + (1-\alpha)\,1\right). \tag{4}$$

$$\begin{array}{rcl} \mathtt{TDC}(D2,B) & \leq & \mathtt{TDC}(D1,B) \\ c_{D2}t_{D2} & \leq & c_{D1}t_{D1} \\ \overset{(4)}{\Rightarrow} \quad \alpha & \leq & \dfrac{\frac{c_{D1}}{c_{D2}} - 1}{t_{sim} - 1} \end{array}$$

If we assume in favour of $D2$ that $n' = c'n\log n$ then with equations 1, 2, 3 we get $\alpha \leq 0.117(\log n)^{-2}$. ■

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

4.4 Consequences

We mentioned in section 1 that PRAMs are classified in theory as EREW, CREW and CRCW PRAMs. Relations among these classes are given in [16, 22]. A further class of ERCW PRAMs is not considered there.

Definition 8 *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 2 *If we change our CRCW design $D1$ to an EREW design $D2$, an ERCW design $D3$ and an CREW design $D4$ we get the relation*

$$c_{D2} < c_{D3} < c_{D4} = c_{D1}.$$

Thus if a PRAM supports combining in the way we described in section 2.1 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 2)

We get $D3$ from $D1$ by reducing the width of the direction queues with the same argument as in subsection 4.2. This shows $c_{D3} < c_{D1}$. We cannot skip the comparators because we still have to detect concurrent writes. This shows $c_{D2} < c_{D3}$. For $D4$ 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 $c_{D4} = c_{D1}$. ∎

Theorem 2 shows that $D4$ is identical to $D1$ and that for any PRAM program B $t_{D1} = t_{D4}$. Thus $D4$ has the same TDC as $D1$ but $D1 \not\preceq D4$.

5 PRAMs vs. Distributed Memory Machines

PRAMs have always been thought to be uncompetitive to Distributed Memory Machines (DMM) because some problems do not need the global memory. In order to compare our PRAM $D1$ with a DMM $D5$ one has to compute

$$R = \mathtt{TDC}(D1, B)/\mathtt{TDC}(D5, B) \ .$$

We are interested in how much more cost–effective DMMs can be than PRAMs and vice versa. Therefore we search for bounds U and L with $L \leq R \leq U$ independently of B and of the particular DMM. It will turn out that for reasonable values of n a DMM cannot be much more cost–effective than a PRAM but vice versa a PRAM can be much more cost–effective than a DMM.

5.1 Simulation of DMMs by PRAMs

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 3.

Theorem 3 *Assume we have a benchmark B as has just been described that has enough parallelism to be computed on a distributed memory machine with efficiency ϵ close to 1. We consider a DMM $D5$ with $N = cn \log n$ processors and communication given by a graph of small degree with N nodes and our PRAM $D1$. Then we get*

$$R \leq U \leq 1.21 \log n + 0.79.$$

Proof: The distributed memory machine with N processors has costs $c_{D5} = N \cdot c_{\tilde{P}} = 12944N$. We only count processor costs $c_{\tilde{P}}$ and ignore network costs although this is unfair towards the PRAM. Suppose that B needs T steps on a sequential machine. Then both the DMM and the PRAM need $T/(\epsilon N)$ steps. We assume in favour of the DMM that the benchmark B can be pipelined perfectly and thus one step takes only one cycle. Thus one has $t_{D5} = 60 \cdot T/(\epsilon N)$.

The PRAM has costs c_{D1} as computed in equation 1 and needs $T/(\epsilon N) = T/(\epsilon cn \log n)$ steps each taking $c \log n$ processor cycles. Thus $D1$ needs $T/(\epsilon n)$ cycles and therefore $t_{D1} = 60 \cdot T/(\epsilon n)$. We then get

$$\begin{aligned} R = \frac{c_{D1}}{c_{D5}} \cdot \frac{t_{D1}}{t_{D5}} &= \frac{15598n \log n + 10179n}{12944N} \cdot \frac{N}{n} \\ &= 1.21 \log n + 0.79 = U. \end{aligned}$$

■

For reasonable values of n, e.g. $n \leq 2^{16}$, the quotient is less than 20. If we would add floating point arithmetic to the ALU as usual in existing parallel machines, the parameter A increases to $A' \approx 100000$ [14] and the quotient decreases dramatically to $0.2 \log n + 0.97$. For $n \leq 2^{16}$ the quotient is smaller than 4.2. If cost of memory is considered too, things change further in favour of the PRAM.

5.2 Simulation of PRAMs by DMMs

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

$$R \geq L \geq (1/438).$$

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, $t_{D1} = 60T/(\epsilon n)$.

Let $D5$ be a DMM with $N = cn \log n$ processors interconnected as a hypercube. $D5$ has costs $c_{D5} = c_{\tilde{P}} N = 12944N$ because we ignore network costs. In order to simulate one step of $D1$ on $D5$ 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 $11 \log n \times 24 \times 2 = 528 \log n$ 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 $S = 528$. $D5$ simulates a PRAM with $N = cn \log n$ processors. Therefore B needs $T/(\epsilon N)$ steps on $D5$ and $t_{D5} = 60 \cdot S \cdot T/(\epsilon cn)$. We now can compute R:

$$R = \frac{c_{D1}}{c_{D5}} \cdot \frac{t_{D1}}{t_{D5}} \geq \frac{15598n \log n + 10179n}{12944N} \cdot \frac{c}{S} = \frac{1}{438} = L$$

■

If we add floating point arithmetic L changes to $1/2640$.

While it is true that most distributed memory machines incorporate some sophisticated routing hardware that would make routing faster, we use the machine from section 5.1 in order to have a common framework for both bounds. Incorporation of routing hardware makes S small but costs increase. The value of L might well get closer to 1, but the corresponding value of R would become smaller too! Note that L will always remain strictly less than one, because otherwise $D5$ would simply be considered a better PRAM emulation than $D1$ and replace it.

5.3 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 $D5$ with N processors interconnected as a $\sqrt{N} \times \sqrt{N}$ torus. Each processor of the torus then holds a $(s/\sqrt{N}) \times (s/\sqrt{N})$–submatrix of both matrices. This example comes very close to the worst case described in section 5.1 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 [35] in a form presented in [17]. 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 [35]. On a PRAM with $n < v$ physical processors we have $t_{D1} = (300 \cdot (e/n) + 108 \cdot (v/n)) \cdot \log v$ as analyzed in [1] and c_{D1} as computed in equation 1. For the distributed memory machine we could use an algorithm from [3] that runs on a hypercube. Its runtime is $O(\log^2 v)$ on v^3 processors. For a hypercube $D5$ with $N < v$ processors we would have $c_{D5} = 12944N$ as computed in section 5.1, $t_{D5} = 40 \cdot (v^3/N) \cdot (\log v)^2$ as sketched in appendix B. Using the fact that $e \leq 0.5 \cdot v^2$ leads to $R \approx 5 \cdot (\log N/(v \log v))$. This would imply $R < L$ and therefore a simulation of the PRAM algorithm is more cost–effective.

6 Conclusions

We have used the framework from [28] which allows to treat computer architecture as a formal optimization problem and to deal quantitatively with hardware/software trade-offs. 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 hard-

ware (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, a prototype with 4 physical processors is running [6], the construction of a prototype with $n = 64$ processors is underway.
In our analyses, we assumed that the benchmarks can be parallelized with efficiency ϵ, yet we did not require ϵ to be a constant. If the parallelism available in a problem is restricted, as e.g. in a vector reduction, this might give a hint about the size of the machine to use.

Acknowledgements

We thank A. Bingert for support while programming the Alliant FX.

References

[1] ABOLHASSAN, F., KELLER, J., AND PAUL, W. J. On physical realizations of the theoretical PRAM model. FB 14 Informatik, SFB–Report 21/1990, Universität des Saarlandes, December 1990.

[2] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. 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.

[3] AKL, S. G. *The Design and Analysis of Parallel Algorithms*. Prentice–Hall, 1989.

[4] ALT, H., HAGERUP, T., MEHLHORN, K., AND PREPARATA, F. P. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, October 1987.

[5] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. The Tera Computer System. In *Proceedings International Conference on Supercomputing*, pages 1–6, 1990.

[6] BACH, P., BRAUN, M., FORMELLA, A., FRIEDRICH, J., GRÜN, T., LICHTENAU, C. Building the 4 Processor SB-PRAM Prototype. In *Proceedings of the Hawaii 30th International Symposium on Computer and System Sciences*, vol. 5, pages 14–23, 1997.

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

[8] BILARDI, G., AND HERLEY, K. T. Deterministic simulations of PRAMs on bounded degree networks. In *Proceedings of the 26th Annual Allerton Conference on Communication, Control and Computation*, September 1988.

[9] CHANG, Y., AND SIMON, J. Continuous routing and batch routing on the hypercube. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 272–281, 1986.

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

[11] DIETZFELBINGER, M., HAGERUP, T., KATAJAINEN, J., AND PENTTONEN, M. A reliable randomized algorithm for the closest-pair problem. Research Report No. 513, Universität Dortmund, FB Informatik, 1993.

[12] DIETZFELBINGER, M., AND MEYER AUF DER HEIDE, F. A new universal class of hash functions and dynamic hashing in real time. Reihe Informatik Bericht Nr. 67, Universität–GH Paderborn, April 1990.

[13] ENGELMANN, C., AND KELLER, J. Simulation-based Comparison of Hash Functions for Emulated Shared Memory. In *Proceedings PARLE '93*, pages 1–11, 1993.

[14] FORMELLA, A. *Leistung und Güte numerischer Vektorrechnerarchitekturen*. PhD thesis, Universität des Saarlandes, FB Informatik, 1992.

[15] FORTUNE, S., AND WYLLIE, J. Parallelism in random access machines. In *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[16] GIBBONS, S., AND RYTTER, W. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[17] HAGERUP, T. Optimal parallel algorithms on planar graphs. *Information & Computation*, 84:71–96, 1990.

[18] HAGERUP, T., SCHMITT, A., AND SEIDL, H. FORK: A high–level–language for PRAMs. In *Proceedings of the Parallel Architectures and Languages Europe 91*, 1991.

[19] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[20] HOCKNEY, R. W., AND JESSHOPE, C. R. *Parallel Computers 2*. Adam Hilger, Bristol and Philadelphia, 1988.

[21] KARLIN, A. R., AND UPFAL, E. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, October 1988.

[22] KARP, R. M., AND RAMACHANDRAN, V. L. A survey of parallel algorithms for shared–memory machines. In VAN LEEUWEN, J., (Ed.), *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, 1990.

[23] KELLER, J. *Zur Realisierbarkeit des PRAM Modells*. PhD thesis, Universität des Saarlandes, FB Informatik, 1992.

[24] LEIGHTON, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Francisco, 1992.

[25] LEIGHTON, F. T., MAGGS, B., AND RAO, S. Universal packet routing algorithms. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[26] MEHLHORN, K., AND VISHKIN, U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[27] MOTOROLA, INC. ASIC DIVISION, Chandler, Arizona. *Motorola High Density CMOS Array Design Manual*, July 1989.

[28] MÜLLER, S. M., AND PAUL, W. J. Towards a formal theory of computer architecture. In *Proceedings of PARCELLA 90, Advances in Parallel Computing*. North–Holland, 1990.

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

[30] PATTERSON, D. A., AND SEQUIN, C. H. A VLSI RISC. *IEEE Computer*, 15(9):8–21, 1982.

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

[32] RANADE, A. G., BHATT, S. N., AND JOHNSON, S. L. The Fluent Abstract Machine. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, 1988.

[33] REIF, J. H., AND VALIANT, L. G. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[34] SCOTT, S. L. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[35] SHILOACH, Y., AND VISHKIN, U. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

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

[37] UPFAL, E. Efficient schemes for parallel communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 55–59, August 1982.

[38] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[39] VALIANT, L. G. General purpose parallel architectures. In VAN LEEUWEN, J., (Ed.), *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.

[40] WEGENER, I. *The Complexity of Boolean Functions*. Teubner, 1987.

Appendices

A Parameter Values

A.1 Processor Costs

The ALU mainly consists of a 32 bit wallace tree multiplier, a barrel shifter and a carry lookahead adder (see [40]). 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 $32 \times 16 \times 2 = 1024$.

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 1.1 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 2×32 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 $64 \times 4 \times 2 + 32 \times (2 \times 2 + 2 \times 6) = 1024$.
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 $5 \times 33 \times 2 \times 7 = 2310$. The basic costs of the ALU then are $A = 1024 + 9216 + 1024 + 2310 = 13392$.
A register file with 16 registers 32 bit wide has basic costs $F = 16 \times 32 \times 12 = 6144$.

A.2 Costs of the network

Our simulations [1] show that network nodes only need buffers of length 2. Packets on the way from processors to memory modules are 76 bits wide (32 bit address, 32 bit data, 12 bit control). In the backward network 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. Additionally we need a comparator and an adder to test identical addresses and to select the smaller one.
The registers have basic costs $N_S = (4 \times 76 + 4 \times 32 + 2 \times 3 \times 7 \times 3) \times 12 = 6696$. The multiplexers have basic costs $(2 \times 76 + 2 \times 32) \times 6 = 1296$. 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 $32 \times (6 + 2) = 256$. 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 $S_S = 4104$.
The network nodes for the Fluent Machine have width 76 for the forward and the backward network. Thus they have basic costs $\tilde{N}_A = 3104$ and $\tilde{N}_S = 8808$.
If we reduce the network nodes of design $D1$ to design an EREW PRAM, we spare the costs for the comparator and reduce the width of the instruction queue by one bit. We then

have basic costs $N'_A = 2320$ and $N'_S = 6192$.

A.3 Network Speed

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, it has to pass a multiplexer that changes the original mode to GHOST and it has to be stored in the input buffer of the following node. Reading the input buffer takes 5 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 5 gate delays. With 32 bit addresses we have $5 + 2 \log 32 + 2 + 2 + 2 + 5 = 26$ gate delays. Because we did not count driver delays and setup and hold times we take a network cycle time $\sigma_N = 30$ gate delays.

B Analysis of Benchmark B1

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 [3] 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. Finally 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 takes only 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 $(38 \cdot (v^3/N) + 20) \log v + 5 \cdot (v^3/N)$ steps. The computation of the connectivity matrix then needs approximately $(\log v)^2 \cdot (38 \cdot (v^3/N) + 20) + 5 \cdot (v^3/N) \cdot \log v$ steps. The computation of matrix D takes approximately $7 \cdot (v^2/n)$ steps, finding of minimums takes approximately $10 \cdot (v/N) \cdot \log v$

steps. The total time t_{D5} then is approximately $40 \cdot (v^3/N) \cdot (\log v)^2$.