Fast Rehashing in PRAM Emulations*

Jörg Keller
CWI
Postbus 94079
1090 GB Amsterdam, The Netherlands

Abstract

In PRAM emulations, universal hashing is a well-known method for distributing the address space among memory modules. However, if the memory access patterns of an application often result in high module congestion, it is necessary to rehash by choosing another hash function and redistributing data on the fly. For the case of linear hash functions $h(x) = ax \bmod m$ we present an algorithm to rehash an address space of size m on a p processor PRAM emulation in time $O(m/p + \log p)$. The algorithm requires $O(\log m)$ words of local storage per processor.

1 Introduction

Parallel machines give their users more and more the view of a global shared memory. This simplifies parallel program design because it frees the programmer from partitioning data and from programming communications in message–passing networks. As massively parallel machines with a physical shared memory are unrealistic, the shared address space is mapped onto distributed memory modules by a *hash function* and accessed via a packet-switching network, both invisible for the user. A hash function distributes almost every memory access pattern evenly among the memory modules. If a particular application, however, requests one memory module much more frequently than the others, it is necessary to choose a new hash function and redistribute data on the fly. This is called *rehashing*. Rehashing has often been neglected in theoretical investigations. However, if it can be done fast, it is an important technique to obtain the expected performance without restarting the application.

*This research is partially supported by DFG through SFB 124, TP D4, and by NWO through NFI Project ALADDIN under Contract No. NF 62–376. Part of this work was done while the author was working at Universität des Saarlandes, Computer Science Dept., Saarbrücken, Germany.

Rehashing is very simple if there is additional storage of size at least m. Either a shadow memory or disk space of size m/p per processor is sufficient. The contents of the shared memory can be copied to this additional storage, and then written back in permuted order. This works in time $O(m/p)$ but is either expensive in case of shadow memory or slow in case of disks. We are interested in rehashing without using secondary storage. We investigate the rehashing problem in the setting of PRAM emulations.

The *PRAM (parallel random access machine)* [8] is a widely used theoretical machine model for processors working synchronously on a shared memory, with unit memory access time. Many numerical and combinatorical parallel algorithms have been designed for the PRAM [4, 9, 11]. However, massively parallel computers normally consist of $p = 2^t$ processors and memory modules connected by a packet switching network, because a physical shared memory would become a bottleneck. Much effort has been put in emulating PRAMs on processor networks [10, 14, 15]. All these solutions are randomized; we omit the deterministic solutions because they use expander graphs and are therefore nonconstructive. A second approach for shared memory emulations uses caches to avoid using the network. An example is the DASH multiprocessor [12]. We do not consider that approach here.

To emulate a PRAM, the shared address space is mapped to the memory modules. Processors that want to access a memory cell send a request across the network to the appropriate module. Multiple threads are run per processor to mask the network latency [2, 5]. The mapping has to guarantee that the number of requests arriving at each memory module (denoted as module congestion) is small for almost all memory access patterns. Otherwise the performance of the emulation gets very poor. This is done by using classes of universal hash functions [6]. Each function of the class provides low module congestion for almost every access pattern. Before running an application, one function of the class is picked randomly. Hence, the probability of an application using patterns that induce high module congestion is very small.

The emulations mentioned above use polynomials of degree $O(\log p)$. But already Ranade mentions that in his simulations linear functions $h(x) = ax \bmod m$ are sufficient [16]. The size of the shared memory is denoted by $m = 2^u$, a must be relatively prime to m. The most significant $\log p$ bits of the u-bit binary representation of $h(x)$ specify the memory module, the lower $u - \log(p)$ bits specify the location on that module. Our own detailed simulations support Ranade's assessment of the usefulness of linear hash functions [7]. In contrast to polynomials, the linear functions bijectively map addresses to memory cells, which avoids secondary hashing at the modules and the waste of memory caused by it [15]. They also have a shorter evaluation time. We will therefore consider linear hash functions.

Unfortunately, if an application uses a memory access pattern that leads to high module congestion, it tends to use this pattern several times. Then it is better to rehash the address space: choose a new hash function $h'(x) = a'x \bmod m$ and redistribute the address space according to the new hash function. If h and h' both are bijective, then the redistribution is a permutation of the contents of the memory cells. It can also be expressed as a permutation π of the addresses while still using h. This allows to formulate the rehashing algorithm as a PRAM program to permute an array of items according to π.

The permutation problem on PRAMs was investigated by Aggarwal, Chandra and Snir [3]. However, their permutation must be fixed. If we consider the hash functions themselves as permutations of $\{0, \ldots, m-1\}$, then we could think of choosing a start hash function h_0 and a fixed permutation π and generate other hash functions $h_i = \pi \circ h_{i-1} = \pi^i \circ h_0$ when rehashing for the i-th time. As however the group of units in $\mathbf{Z}/m\mathbf{Z}$ is not cyclic if m is a power of two [17, p. 124], the choice of new hash functions would be restricted. This argument even holds for arbitrary permutations, as the symmetric group $\mathbf{S}_n$ is not cyclic for $n > 2$. Hence we must deal with a permutation π that is not fixed.

We present an algorithm to permute m data items on a PRAM emulation with p processors and memory modules in time $O(m/p + \log p)$ if the permutation is a linear function. The algorithm does not require any global storage and can therefore be used to rehash the address space of the PRAM emulation.

In section 2 we provide facts and notations to be used later on. In section 3 we present the rehashing algorithm and analyze its runtime and space complexity. In section 4 we show how to decide when to invoke the rehashing algorithm. In section 5 we show that an obvious simplification of the rehashing algorithm will probably be slow due to long cycles.

2 Linear permutations

2.1 Form of permutation π

We want to express the rehashing problem as a permutation of addresses while still using the hash function h. If we do this, we can redistribute the address space by executing the PRAM program to permute the addresses, and then switch the hash function to h'. Consider an arbitrary address x. Before rehashing, x is mapped to cell $h(x)$, after rehashing it will be mapped to cell $x' = h'(x)$. Before rehashing, address $y = h^{-1}(x')$ is mapped to cell x'. Hence, the redistribution can be expressed as permuting addresses according to $\pi(x) = y$.

In $\mathbf{Z}/m\mathbf{Z}$, the numbers relatively prime to m form a multiplicative group, the group of units [17, p. 119]. It follows that a and a' can be inverted and that h and h' are bijective. Then

$$\pi(x) = h^{-1}(h'(x)) = a^{-1}a'x \bmod m\,. \qquad (1)$$

As a and a' are units, $b = a^{-1}a' \bmod m$ also is a unit and $\pi(x) = bx \bmod m$ is bijective. We investigate $m = 2^u$. The group of units here is the set of odd numbers between 1 and $m - 1$.

2.2 Structure of permutation π

We want to permute the addresses without using secondary storage. This can be done by splitting permutation π into its cycles $C_1, C_2, \ldots$, distributing the cycles among the processors, and then having each processor permute its assigned cycles sequentially. A processor needs only local space to buffer one item if it permutes a cycle sequentially.

To follow this idea, we need to explore the structure of π. For each cycle, we need to know an entry element and its length. The length is necessary to schedule the cycles among the processors, as the time to permute a cycle is proportional to its length. Fortunately, the structure of linear permutations is very regular.

For x in $\{0, \ldots, m-1\}$ we define $j(x) = \max\{k | x \text{ can be divided by } 2^k\}$. Then every x in $\{0, \ldots, m-1\}$ has a unique representation $x = 2^{j(x)}x'$

where $0 \leq j < u$ and $x' < m/2^{j(x)}$ is odd. We can now partition the set $U(m) = \{0, \ldots, m-1\}$ into sets

$$\begin{aligned} U_k(m) &= \{x \in U(m) | j(x) = k\} \\ &= \{x \in U(m) | x = 2^k x' \text{ and } x' \text{ odd}\} \quad . \end{aligned}$$

We apply π to an address x in $U_k(m)$. $\pi(x) = bx \bmod m = b2^k x' \bmod m$. As b and x' are units, $\tilde{x} = bx' \bmod m/2^k$ also is a unit and $2^k \tilde{x} \bmod m = 2^k(bx' - rm/2^k) \bmod m$ (for some r) $= \pi(x)$. Hence $\pi(x)$ is an element of $U_k(m)$, too. We conclude that each cycle of π is contained completely in one of the $U_k(m)$. Furthermore $\phi_k(x) = x/2^k$ is a bijection from $U_k(m)$ to $U_0(m/2^k)$, $\pi_k(x') = bx' \bmod m/2^k$ is a permutation on $U_0(m/2^k)$ and for $x \in U_k(m)$ we have $\pi(x) = \phi_k^{-1}(\pi_k(\phi_k(x)))$. We therefore restrict our attention to the problem of permuting odd numbers ($U_0(m/2^k)$) and then apply this method by using ϕ_k^{-1} to permute $U_k(m)$. Note that $U_0(m)$ is the set of units and hence a multiplicative group. Consider the cycles of π when applied on $U_0(m)$. A cycle starting with an element x has the form $x, bx, b^2x \ldots, b^{l-1}x, x$. Then l is the order of b in $U_0(m)$. We can conclude that all cycles have the same length, which must be a power of two because the order of $U_0(m)$ is a power of 2. The number of cycles $\sigma = |U_0(m)|/l$ then also is a power of two.

We call x the *entry element* of the cycle and denote the cycle with entry element x by $C(x)$. Note that each element of a cycle can be chosen to be the entry element. We try to find a set of entry elements c_i, $i = 0, \ldots, \sigma - 1$, such that $C(c_i) \neq C(c_k)$ for $i \neq k$ and that all cycles together span $U_0(m)$. The following Lemma makes sure that there is such a set where the entry elements of the cycles have a very regular form.

Lemma 1 *If $b \neq -1$, then the elements 5^k and $(-1)5^k$, where $0 \leq k < \sigma/2$, are all on different cycles. If $b = -1$, then the elements 5^k, where $0 \leq k < \sigma$, are all on different cycles.*

Proof: $U_0(m)$ is generated by -1 and 5 [17, p. 124]. Each x in $U_0(m)$ thus has a unique representation $x = (-1)^\alpha 5^{\alpha'} \bmod m$, where $\alpha \in \{0, 1\}$ and $\alpha' \in \{0, \ldots, m/4 - 1\}$. Let $b = (-1)^\beta 5^{\beta'}$. If $b = 1$ or $b = -1$, then the result is straightforward.

Let us now consider that $b \notin \{-1, 1\}$ and therefore that $\beta' \neq 0$. We have to show that for every $k, v \in \{0, \ldots, \sigma/2 - 1\}$ and any $g \in \{0, \ldots, l-1\}$, $5^k \neq b^g 5^v$ if $k \neq v$ and $(-1)5^k \neq b^g 5^v$. The first inequality is equivalent to $5^{k-v} \neq b^g$. With $b = (-1)^\beta 5^{\beta'}$, we obtain $5^{k-v} \neq (-1)^{g\beta} 5^{g\beta'}$. As $0 < |k - v| < \sigma/2$, we have the desired property if β' is a multiple of $\sigma/2$.

The second inequality is equivalent to $(-1)5^{k-v} = (-1)^{g\beta} 5^{g\beta'}$. In order to meet $(-1) = (-1)^{g\beta}$, g has to be odd, especially not equal to zero. But if β' is a multiple of $\sigma/2$, then $5^{g\beta'}$ can never equal 5^{k-v} because $0 < |k - v| < \sigma/2$.

We finish the proof by showing that $\beta' \neq 0$ is a multiple of $\sigma/2$. Consider b^l, which equals $1 \bmod m$ because l is the order of b. With the above representation we obtain $(-1)^{l\beta} 5^{l\beta'} \equiv 1 \bmod m$. It follows that $l\beta' \equiv 0 \bmod m/4$. This is equivalent to $\beta' \equiv 0 \bmod m/(4l)$, because l is a power of two. As $l = |U_0(m)|/\sigma = (m/2)/\sigma$, we obtain $\beta' \equiv 0 \bmod \sigma/2$. Therefore β' must be a multiple of $\sigma/2$. $\square$

2.3 Working with multi-threaded processors

Assume that the time to access a shared memory cell via the network is L. In order to hide this latency from the user, each processor runs L threads. Each thread has its own register set. The threads are executed in a round-robin manner with one instruction per turn. The processors are pipelined with pipeline depth L. Hence every L cycles, each thread has executed another instruction. We will call the $N = Lp$ threads of the emulation *virtual processors*. We assume N to be a power of two.

Consider a problem with sequential time complexity T, which is also called *work*. If it can be completely parallelized on N virtual processors, then it needs T/N steps on a p-processor PRAM emulation, each taking L cycles. Thus the runtime will be T/p. We will proceed in the same way with the rehashing problem.

3 Algorithm

We will now describe the permutation algorithm for a PRAM with N processors. The algorithm works in rounds, in each round one $U_j(m)$ is permuted, as long as $|U_j(m)| \geq N$. All $U_j(m)$ with $|U_j(m)| < N$ are handled together in a final round. We will distinguish l and σ in different $U_j(m)$ by an index j.

To permute one $U_j(m)$, we have each processor permute σ_j/N cycles sequentially if $\sigma_j \geq N$. If there are fewer than N cycles, then N/σ_j processors work together to permute one cycle. We split each cycle in pieces of size N/σ_j, each piece is permuted in one step. Permuting a cycle piece after piece is somewhat tricky, because the virtual processor that picked the last element of the piece may store it only if another processor has picked the first element of the next piece.

Now, consider the final round. $|U_j(m)| = m/2^{j+1}$ is less than N for $j \geq \log(m/N) = f$ and $\sum_{j \geq f} |U_j(m)| = N-1$. We split the cycles in these $U_j(m)$ completely and obtain $N-1$ pieces consisting of single cells, that can be permuted in a single step.

We ensure with a preprocessing phase that each processor can find the entry elements of its assigned cycles and pieces in constant time.

3.1 Preprocessing phase

The preprocessing phase has to provide the processors with σ_j and l_j for all j, and with the entry elements of their assigned cycles and pieces of cycles. We assume that multiplication and shifts of integers and $\lfloor \log_2(x) \rfloor$ for positive integers x can be computed in one instruction. The preprocessing phase works only on processors' local memories. Therefore, we will not run multiple threads during the precomputation phase. We assume that physical processor x will run virtual processors $x, x+p, \ldots, x+(L-1)p$ during the rehashing phase.

The computation of l_j and σ_j has to be done once per physical processor and is identical for all processors. We compute a table of b^{2^i} for $0 \leq i < u$ by successively computing $b^{2^{i+1}} = b^{2^i} \cdot b^{2^i}$. We obtain the l_j by checking whether $b^{2^i} \bmod m/2^j$ equals 1. As the l_j are decreasing with increasing j, we have to traverse the table only once. The σ_j are obtained as $|U_j(m)|/l_j$.

To compute entry elements, we build up a table of values 5^{2^i} similarly to the table of b^{2^i}. Each physical processor x computes 5^x as $\prod_{x_i=1} 5^{2^i}$, if $(x_{\log p-1}, \ldots, x_0)$ is the binary representation of x. With the help of this value and the table, the entry elements for each virtual processor run on this physical processor can be computed in constant time per entry element, for an appropriate assignment of cycles to processors.

For the final phase, we split each cycle completely and assign each processor one element to move. This can be done in constant time.

3.2 Analysis

The preprocessing phase takes time $O(\log m + L)$. If we only consider bounded-degree networks, then $L = \Omega(\log p)$. Moreover, there are emulations with $L = \Theta(\log p)$ [2, 15]. For m polynomial in p, $\log m = \Theta(\log p)$ and hence the time for the preprocessing phase is $\Theta(\log p)$. The space needed for each physical processor also is $\Theta(\log p)$.

The rehashing phase is completely parallelized. The total work $T = \Theta(m)$ is distributed evenly and hence the runtime is $O(m/p)$ due to subsection 2.3. The rehashing phase needs $O(L) = O(\log p)$ space per physical processor.

The total runtime is $O(m/p + \log p)$. For $m \geq p \log p$, this is $O(m/p)$, which is optimal.

4 Detection

When using the algorithm for rehashing in a PRAM emulation, we encounter the problem of automatically detecting the necessity to rehash. A complete solution to this problem would consist of predicting the address trace of the remaining program part, computing the distributions with and without rehashing and computing from this the runtimes T_b and T_a, respectively. If the time to rehash the address space is T_r, then rehashing is useful if $T_b + T_r < T_a$.

However, this prediction is often impossible because of future input or it would take too much time to compute T_b and T_a, even if we perform it only every x cycles to predict the next x cycles.

To avoid prediction, we take advantage of the regular structure of programs. A lot of applications spend most of their time in loops. Hence, future performance can be guessed by observing current performance. A simple approach consists of counting the fraction of stalled cycles in the last x cycles. If this fraction gets larger than a certain user-defined threshold $1/t$, then rehashing is initiated. This detection can be done by maintaining two counters $\mathrm{CO}_{\mathrm{ST}}$ and $\mathrm{CO}_{\mathrm{TO}}$ for the number of stalled and the number of total cycles, and a register for storing t. In the beginning, both counters are set to zero. If $\mathrm{CO}_{\mathrm{TO}}$ reaches x, we want to check whether

$$\frac{\mathrm{CO}_{\mathrm{ST}}}{\mathrm{CO}_{\mathrm{TO}}} > \frac{1}{t} .$$

To do this, we multiply $\mathrm{CO}_{\mathrm{ST}}$ with t and subtract $\mathrm{CO}_{\mathrm{TO}}$ from it. If the result is positive, we initiate rehashing. Afterwards, the counters are set to zero again.

This allows the user to define a threshold in a wide range, and detection can be made without floating point operations or divisions. The value of t might depend on the application and on the particular implementation of the rehashing algorithm.

5 Simplification of the algorithm

One might think about simplifying the algorithm for rounds where there are less than N cycles. Instead of having several processors permuting one cycle, one could use only σ_j processors. The runtime of this round then will increase from $\sigma_j l_j/N$ to l_j. If this does not happen to often and l_j is not too large, the loss in runtime would be quite small. However, Theorem 2 shows that the probability of a small loss of performance is quite small.

Theorem 2 *Let T_0 and T_1 be the runtimes of the original and the simplified algorithm for a randomly chosen b. Then*

$$\mathrm{Prob}(T_1/T_0 \leq \delta) \leq 4\delta/N \tag{2}$$

for any real number δ with $1 \leq \delta \leq N/8$.

After choosing an element b, the quotient T_1/T_0 can be computed in time $O(\log m)$. One might think to increase $\mathrm{Prob}(T_1/T_0 \leq \delta)$ by repeatedly choosing b until $T_1/T_0 \leq \delta$ or until a time bound, e.g. m/p, is reached. However, this would affect the random choice of a new hash function and should not be done.

The proof of Theorem 2 relies on the distribution of orders of elements in $U_0(m)$. This distribution is given in the following Lemma 3.

Lemma 3 *If we randomly choose an element b of $U_0(m)$, then its order can be 2^j, where $0 \leq j \leq u-2$. Furthermore,*

$$\mathrm{Prob}(\mathrm{ord}(b) = 2^j) = \begin{cases} 1/2^{u-j-1} & \text{if } j \neq 1 \\ 3/2^{u-1} & \text{if } j = 1. \end{cases} \tag{3}$$

Proof: As the order of $U_0(m)$ is 2^{u-1}, the order of an element b has to be a power of two because it has to divide the group's order. As $U_0(m)$ is not cyclic [17, p. 124], the order of b can be at most 2^{u-2}.

The group $U_0(m)$, which is the group of units in $\mathbf{Z}/2^u\mathbf{Z}$, is isomorphic to the product $U' \times U'' = (\{0,1\}, + \bmod 2) \times (\{0,\ldots,2^{u-2}-1\}, + \bmod 2^{u-2})$ by an isomorphism ψ [17, p. 124]. The order of an element b in $U_0(m)$ with $\psi(b) = (b_1, b_2)$ is determined by the order of b_2 in U'' if $b_2 \neq 0$, and by the order of b_1 in U' otherwise. U'' is cyclic and therefore the number of elements in U'' with order 2^j equals $\phi(2^j)$ (the Euler function) [17, p. 119]. If $b_2 \neq 0$ and hence $\mathrm{ord}(b_2) \geq 2$, there are two elements $\psi^{-1}(0, b_2)$ and $\psi^{-1}(1, b_2)$ in $U_0(m)$ with order $\mathrm{ord}(b_2)$. If $b_2 = 0$ and hence $\mathrm{ord}(b_2) = 1$, there are are two elements $\psi^{-1}(0,0)$ and $\psi^{-1}(1,0)$ in $U_0(m)$ with orders 1 and 2, respectively. It follows that the number of elements in $U_0(m)$ with order 2^j is $2\phi(2^j)$ if $j \geq 2$, $2\phi(2)+1$ if $j = 1$, and 1 if $j = 0$.

For a randomly chosen element b in $U_0(m)$ we can now define $\mathrm{Prob}(\mathrm{ord}(b) = 2^j)$ as the quotient of the number of elements in $U_0(m)$ with order 2^j and the order of $U_0(m)$. With $\phi(P^r) = (P-1)P^{r-1}$ for a prime P and an integer r [17, p. 120], Equation (3) follows. □

Proof of Theorem 2: We will prove the Theorem by computing T_0, a lower bound B on T_1, and $\mathrm{Prob}(B/T_0 > \delta)$. Then we obtain

$$\begin{aligned} \mathrm{Prob}(T_1/T_0 \leq \delta) &= 1 - \mathrm{Prob}(T_1/T_0 > \delta) \\ &\leq 1 - \mathrm{Prob}(B/T_0 > \delta). \end{aligned} \tag{4}$$

We measure the runtime in number of movements per processor. In the original algorithm, this is $|U_0(m)|/N$ for all stages but the last one, where it is 1. Hence

$$T_0 = 1 + \sum_{j=0}^{u-\log N-1} |U_j(m)|/N = 2^u/N\,.$$

In the simplified algorithm, the runtime increases to l_j in stages where $\sigma_j < N$. Hence

$$T_1 = 1 + \sum_{j=0}^{u-\log N-1} \max(|U_j(m)|/N, l_j)\,. \tag{5}$$

From $l_{j+1} \geq l_j/2$, it follows that $l_j \geq l_0/2^j$. We will assume that $l_0 = 2^x$. We also know that $|U_j(m)| = 2^{u-j-1}$. We bound T_1 from below by putting these facts into Equation (5).

$$T_1 \geq 1 + \sum_{j=0}^{u-\log N-1} \max(2^{u-j-1-\log N}, 2^{x-j})\,.$$

If $x \leq u-1-\log N$, then the maximum always takes the left term's value, and it follows that $T_1 \geq T_0$. If $x \geq u - \log N$, then the maximum always takes the right term's value, and

$$T_1 \geq 1 + 2^{x+1} - 2^{x-u+\log N+1}\,. \tag{6}$$

If $u \geq \log N + 1$, then $2^{x-u+\log N+1} \leq 2^x$ and we can simplify Equation (6) to $T_1 \geq 2^x$.

With this we have a lower bound B on T_1 with

$$B = \begin{cases} 2^x & \text{if } x \geq u - \log N \\ T_0 & \text{if } x \leq u - 1 - \log N. \end{cases}$$

We use B to compute $\text{Prob}(B/T_0 > \delta)$. $B/T_0 > \delta$ can only happen if $x \geq u - \log N$, because $B = T_0$ otherwise. As $B/T_0 = 2^x/2^{u-\log N}$, the condition $B/T_0 > \delta$ is equivalent to $x > \log\delta + u - \log N = \kappa$. With $\text{ord}(b) = l_0 = 2^x$, we get

$$\begin{aligned}\text{Prob}(B/T_0 > \delta) &= \text{Prob}(x > \kappa) \\ &= \sum_{j=\kappa+1}^{u-2} \text{Prob}(\text{ord}(b) = 2^j) \\ &= \begin{cases} 1 - 4\delta/N & \text{if } \delta \leq N/8 \\ 0 & \text{otherwise}\,. \end{cases} \qquad (7)\end{aligned}$$

By combining Equations (4) and (7), we prove the claimed Equation (2) of the Theorem. □

6 Conclusions

PRAM emulations that use linear hash functions can be rehashed in optimal time. The algorithm does not require secondary storage devices like hard disks. The computations only require multiplication and shifts of integers at instruction level. Only for the detection of rehashing two counters are needed. The counter CO_{TO} is normally present in the system as a timer, the counter CO_{ST} can be realized in software. One can modify the compiler to increase a register R by the number of executed instructions at the end of each basic block. This gives $\text{CO}_{\text{ST}} = \text{CO}_{\text{TO}} - R$. Therefore the rehashing algorithm can be implemented without any hardware changes.

The practical usefulness of rehashing has not yet been tested, because there is no working prototype of a PRAM emulation. However, Lipton and Naughton [13] construct programs that use timers to measure emulation times of PRAM steps and base their future behaviour on these times. These programs are called "clocked adversaries" and they lead provably to bad distributions of requests and hence to long runtimes. This hints that rehashing will be needed in practice.

The concept of rehashing will be implemented in the SB-PRAM [1], the prototype of the PRAM emulation described in [2].

It is still an open problem whether on-line rehashing is possible. By on-line rehashing, we understand that c steps of the PRAM application and c steps of the rehashing procedure can be executed alternately for the time span of rehashing. Currently, the PRAM application has to be stopped while rehashing the address space.

Acknowledgements

I am very thankful to Dany Breslauer for suggestions about the choice of entry elements and to Martin Dietzfelbinger for many stimulating discussions. I also want to thank Stefan Ellwert and Volker Müller for providing some help in algebraic notation.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul and D. Scheerer, On the physical design of PRAMs. *Comput. J.*, to appear.

[2] F. Abolhassan, J. Keller, and W. J. Paul. On the cost–effectiveness of PRAMs. In *Proc. 3rd Symp. on Parallel and Distributed Processing*, pp. 2–9. IEEE, 1991.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proc. 1st Symp. on Parallel Algorithms and Architectures*, pp. 11–21. ACM, 1989.

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

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. 1990 Internat. Conf. on Supercomputing*, pp. 1–6. ACM, 1990.

[6] J. Carter and M. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18:143–154, 1979.

[7] C. Engelmann and J. Keller. Simulation-based comparison of hash functions for emulated shared memory. In *Proc. PARLE '93, Parallel Architectures and Languages Europe*, pp. 1–11. Springer LNCS, 1993.

[8] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Symp. on Theory of Computing*, pp. 114–118. ACM, 1978.

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

[10] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *J. Assoc. Comput. Mach.*, 35:876–892, 1988.

[11] R. M. Karp and V. L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Vol. A*, pp. 869–941. Elsevier, 1990.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *Comput.*, 25(3):63–79, 1992.

[13] R. J. Lipton and J. F. Naughton. Clocked adversaries for hashing. *Algorithmica*, 9:239–252, 1993.

[14] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, 21:339–374, 1984.

[15] A. G. Ranade. How to emulate shared memory. *J. Comput. System Sci.*, 42:307–326, 1991.

[16] A. G. Ranade, S. N. Bhatt, and S. L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conf. on Advanced Research in VLSI*, pp. 71–93, 1988.

[17] H.-J. Reiffen, G. Scheja, and U. Vetter. *Algebra.* B.I.–Wissensch.v., 2nd edition, 1984.