Fast Rehashing in PRAM Emulations[*]

Jörg Keller
Fachbereich 14 Informatik
Universität des Saarlandes
Postfach 151150, 66041 Saarbrücken, Germany

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 $m = 2^u$ on a PRAM emulation with p processors in time $O(m/p + \log m + L)$, where L denotes the network latency. For the common case that m is polynomial in p and $L = O(\log p)$ the runtime is $O(m/p + \log p)$. The algorithm requires $O(\log m + L)$ words of local storage per processor. We show that an obvious simplification of the algorithm will significantly increase runtime with high probability.

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 (denoted as high module congestion), 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 CWI, Amsterdam, The Netherlands. A preliminary version appeared in Proc. 5th IEEE Symp. on Parallel and Distributed Processing, pp. 626-632, IEEE CS Press 1993.

Rehashing in a machine with p processors and a shared memory of size m 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 application is interrupted, the contents of the shared memory are copied to the 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 also will stop the application in order to rehash as it is unclear how to interleave both tasks. 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]. Much effort has been put in emulating PRAMs on processor networks [10, 14, 15]. We restrict to randomized solutions; we omit the deterministic solutions because they use special expander graphs for which no constructions are known today. 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 obtain unit memory access time when emulating a PRAM, multiple threads are run per processor to mask the network latency L [2, 5]. 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 steps of the machine, each thread has executed another instruction. We will call the $N = Lp$ threads of the emulation *virtual processors*.

Hashing 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 simulations of his emulation algorithm linear functions $h(x) = ax \bmod m$, where a must be relatively prime to m, "perform well in practice" [16, p. 77]. We will restrict to the case $m = 2^u$. The most significant $\log p$ bits of the u-bit binary representation of $h(x)$ specify the memory module, the remaining 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, 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.

Rehashing then consists of choosing a new hash function $h'(x) = a'x \bmod m$ and redistributing the address space according to the new hash function. As the hash functions h and h' both are bijective, the redistribution is a permutation of the contents of the memory cells. It can also be expressed as a permutation π of the addresses. 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 ith 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 $O(m/p + \log m + L)$ 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 discuss 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

To express the rehashing problem as a permutation of addresses, we consider an arbitrary address x. Before rehashing, x is mapped to cell $h(x)$, after rehashing it will be mapped to cell $y = h'(x)$. Before rehashing, address $x' = h^{-1}(y)$ is mapped to cell y. Hence, the redistribution can be expressed as permuting addresses according to $\pi(x) = x'$.

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

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

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

We want to permute the addresses without using secondary storage. This can be accomplished if we permute cycles of π sequentially and employ parallelism by permuting several cycles with different processors. Then each processor only needs to buffer one item locally in addition to the information about the cycle structure of π.

The above idea leads to the following high level description of our rehashing strategy:

- Split permutation π into its cycles $C_1, C_2, \ldots$
- Find an element of each cycle.
- Distribute the cycles among the processors such that work is evenly balanced.
- Have each processor permute its assigned cycles sequentially.

To follow our idea, we need to explore the cycle 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 $U(m) = \{0, \ldots, m-1\}$ we define $j(x) = \max\{k \mid 2^k \text{ divides } x\}$. Then every x in $U(m)$ 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 $U(m)$ into sets

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

We apply π to an address x in $U_k(m)$. Then $\pi(x) = bx \bmod m = b2^k x' \bmod m$. As b and x' are units, $\tilde{x} = bx' \bmod m/2^k = bx' - rm/2^k$ for some r also is a unit and $2^k\tilde{x} \bmod m = 2^k(bx' - rm/2^k) \bmod m = \pi(x)$. 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 finding the cycles of π in $U_0(m)$. We can use that method and the bijections ϕ_k to find all cycles of π in $U_k(m)$ for $k = 1, \ldots, u-1$. As the $U_k(m)$ partition $U(m)$ we have then found all cycles of π and hence we have fulfilled the the first task in our rehashing strategy.

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[1] $x, bx, b^2x \ldots, b^{l-1}x, x$. Then l is the order of b in $U_0(m)$. We can conclude that all cycles in $U_0(m)$ have the same length. This length must be a power of two because the order of $U_0(m)$ is a power of two. The number of cycles, which must also be a power of two, is denoted by $\sigma = |U_0(m)|/l$.

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. To fulfill the second task in our rehashing strategy, we first 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 $i \neq k$. The cycles $C(c_i)$, $i = 0, \ldots, \sigma - 1$, span $U_0(m)$. Lemma 1 makes sure that there is such a set where the entry elements of the cycles have a very regular form. To find entry elements to the cycles in $U_k(m)$ we use the bijections ϕ_k as we did to find the cycles.

Lemma 1 *If $b \neq -1$, then the elements $c_{2k} = 5^k$ and $c_{2k+1} = (-1)5^k$, where $0 \leq k < \sigma/2$, are all on different cycles. If $b = -1$, then the elements $c_k = 5^k$, 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] We omit writing $bx \bmod m, b^2x \bmod m, \ldots$ here to simplify notation.

$(-1)5^k \neq b^g 5^v$. 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 $(-1)5^{k-v} \neq (-1)^{g\beta} 5^{g\beta'}$. 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 $\sigma/2$. □

3 Algorithm

We will now describe the permutation algorithm for a PRAM with N processors. We assume N to be a power of two. The algorithm works in phases, in each phase the cycles of one $U_j(m)$ are permuted, as long as $|U_j(m)| \geq N$. All $U_j(m)$ with $|U_j(m)| < N$ are handled together in a final phase. We ensure with a preprocessing phase that each processor can find the entry elements of its assigned cycles in constant time. We will distinguish l and σ in different phases by an index j.

To permute the cycles of $U_j(m)$ in phase j, we have each processor permute σ_j/N cycles sequentially if $\sigma_j \geq N$. As all of these cycles have the same length, the work is balanced. Processor y is assigned to cycles $C(c_y), C(c_{y+N}), \ldots$.

If there are fewer than N cycles, then $v = N/\sigma_j$ processors work together to permute one cycle. Processor y is assigned to cycle $C(c_k)$ with $k = y \bmod \sigma_j$. We split each cycle in pieces of size v, each piece is moved in one round. If the entry element of this cycle is c_k, then the first piece consists of the elements $c_k, bc_k, b^2c_k, \ldots, b^{v-1}c_k$, the second piece consists of $b^v c_k, \ldots, b^{2v-1}c_k$ and so on. The number of pieces of one cycle is $l_j/v = |U_j(m)|/N$.

To move the wth piece of cycle $C(c_k)$, the processors $k, k+\sigma_j, \ldots, k+(v-1)\sigma_j$ load the contents of $b^{(w-1)v}c_k, \ldots, b^{wv-1}c_k$ and store them to $b^{(w-1)v+1}c_k, \ldots, b^{wv}c_k$.

Permuting a cycle this way is somewhat tricky, because the processor that picked the last element of one piece may store it only if another processor has picked the first element of the next piece. Hence, movements of pieces have to be overlapped: before the wth piece is stored, the $(w+1)$st piece is loaded.

An alternative would be to split each cycle $C(c_k)$ into v pieces of length l_j/v. The wth piece consists of $b^{(w-1)l_j/v}c_k, \ldots, b^{wl_j/v-1}c_k$. Each processor assigned to this cycle would then move one piece sequentially. However, the computation of $b^{(w-1)l_j/v}$ would lead to difficulties in the preprocessing phase. Therefore the first method is preferred.

Note that while working on $U_j(m)$, each element $x \in U_j(m)$ represents address $2^j x$ because we have to apply ϕ_j^{-1}. Therefore, when permuting a cycle, the entry element has to be

```
for($j = 0; j < f; j++$){ /* phase $j$, work on $U_j(m)$ */
  if($\sigma_j \geq N$){ /* if at least $N$ cycles */
    $x = 2^j\cdot$ FirstEntry($j, y$); /* entry element of first cycle */
    for($k = 0; k < \sigma_j/N; k++$){
      PermuteCycle($x, l_j$); /* permute $C(x)$ */
      $x =$NextEntry($x$); /* entry element of next cycle */
    }
  }else{ /* less than $N$ cycles */
    $x = 2^j\cdot$ FirstPiece($j, y$); /* element of first piece */
    for($k = 0; k < |U_j(m)|/N; k++$){
      MovePiece($x$); /* move pieces overlapped */
      $x =$NextPiece($x$); /* element of next piece */
    }
  }
}

$x = y \cdot 2^f$; /* final phase */
$tmp = A[x]$;
$A[(b \cdot x) \bmod m] = tmp$;
```

Figure 1: Rehashing Algorithm for virtual processor y

multiplied with 2^j before processors start to move elements.

Now we consider the final phase. $|U_j(m)| = m/2^{j+1}$ is less than N for $j \geq \log(m/N) = f$ and $\sum_{j\geq f} |U_j(m)| = N - 1$. When applying ϕ_j^{-1}, we see that those $U_j(m)$ represent addresses $i2^f$, $1 \leq i < N$. To permute the cycles in these $U_j(m)$, processor i loads the content of $i2^f$ and stores it to $bi2^f \bmod m$.

The program in C notation for processor y is shown in Figure 1. Array `A` represents the shared memory, all other variables are local.

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. The preprocessing phase works only on processors' local memories. Therefore, we will not run multiple threads during the preprocessing phase.

Each physical processor z first computes eight tables. Each table is stored once per physical processor. Tables t_0, t_1 and t_2 are identical for all processors, the contents of the other tables depend on z. Let $(z_{\log p-1}, \ldots, z_0)$ be the binary representation of z. The first six tables are necessary for $b \neq -1$, the last two tables are needed for the case $b = -1$.

Table t_0 consists of some powers of b, i.e. $t_0[i] = b^{2^i} \bmod 2^u$ for $0 \leq i < \log m$. Table t_0 can be computed by repeated squaring, i.e. $t_0[0] = b$, $t_0[i+1] = t_0[i] \cdot t_0[i] \bmod 2^u$ for $i \geq 1$.

Table t_1 does the same for powers of 5, i.e. $t_1[i] = 5^{2^i} \bmod 2^u$ for $0 \leq i < \log m$. Table t_2 contains the first L powers of b, i.e. $t_5[i] = b^i \bmod 2^u$ for $0 \leq i < L$. Table t_2 can be computed by repeatedly multiplying with b, i.e. $t_2[0] = 1$, $t_2[i+1] = t_2[i] \cdot b \bmod 2^u$ for $i \geq 1$.

Table t_3 has the form $t_3[i] = 5^{\lfloor (z \bmod 2^{i+1})/2 \rfloor} \bmod 2^u$ for $0 \leq i < \log p$. As $\lfloor (z \bmod 2^{i+1})/2 \rfloor = \lfloor z/2 \rfloor \bmod 2^i = \sum_{s=0}^{i-1} z_{s+1} \cdot 2^s$ it follows that

$$t_3[i] = \prod_{s=0}^{i-1} \left(5^{2^s}\right)^{z_{s+1}} \bmod 2^u \ .$$

Hence, t_3 can be computed as $t_3[0] = 1$, $t_3[i+1] = t_3[i] \cdot (t_1[i])^{z_{i+1}} \bmod 2^u$ for $i \geq 1$.

Table t_4 has the form $t_4[i] = 5^{\lfloor (z+ip)/2 \rfloor} \bmod 2^u$ for $0 \leq i < L$. As $\lfloor (z+ip)/2 \rfloor = \lfloor z/2 \rfloor + i \cdot (p/2)$, table t_4 can be computed as $t_4[0] = t_3[\log p - 1]$, $t_4[i+1] = t_4[i] \cdot t_1[\log p - 1] \bmod 2^u$ for $i \geq 1$.

Table t_5 has the form $t_5[i] = b^{\lfloor z/2^i \rfloor}$ for $1 \leq i \leq \log p$. As $\lfloor z/2^i \rfloor = 2 \cdot \lfloor z/2^{i+1} \rfloor + z_i$, table t_5 can be computed as $t_5[\log p] = 1$, $t_5[i] = t_5[i+1] \cdot t_5[i+1] \cdot b^{z_i} \bmod 2^u$ for $i < \log p$.

The last two tables are obtained by changing tables t_3 and t_4 to $t_3'[i] = 5^{z \bmod 2^{i+1}}$ and $t_4'[i] = 5^{z+ip}$. The computation of those tables is easily derived from the computation of t_3 and t_4.

The computation of l_j and σ_j is done once per physical processor and is identical for all processors. 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 table t_0 only once. The σ_j are obtained as $|U_j(m)|/l_j$.

We show that the eight tables are sufficient to compute the entry elements for all cycles and pieces of cycles if physical processor z simulates virtual processors $z, z+p, \ldots, z+(L-1)p$.

Lemma 2 *The procedures FirstEntry, NextEntry, FirstPiece and NextPiece can be computed with constant numbers of operations in each phase j.*

Proof: We start with $b \neq -1 \bmod 2^{u-j}$. If $\sigma_j \geq N$, virtual processor y is assigned to cycles $C(c_y), C(c_{y+N}), \ldots$. Hence, FirstEntry$(j, y)$ has to compute c_y, NextEntry(x) has to compute $c_{y+(s+1)N}$ when given $x = c_{y+sN}$.

If y is even then $c_y = 5^{y/2} \bmod 2^{u-j}$, if y is odd then $c_y = (-1)5^{(y-1)/2} \bmod 2^{u-j}$. As virtual processor y is simulated on physical processor z with $y = z + ip$, it follows that $c_y = (-1)^{y \bmod 2} 5^{\lfloor (z+ip)/2 \rfloor} \bmod 2^{u-j}$. Hence, procedure FirstEntry(j, y) consists of computing $(-1)^{y \bmod 2} \cdot t_4[\lfloor y/p \rfloor] \bmod 2^{u-j}$.

From Lemma 1 it follows directly that $c_{y+(s+1)N} = c_{y+sN} \cdot 5^{N/2}$. As N is a power of two, procedure NextEntry(x) simply consists of computing $x \cdot t_1[\log N - 1] \bmod 2^u$.

If $\sigma_j \leq N/2$, virtual processor y is assigned to cycle $C(c_k)$, where $k = y \bmod \sigma_j$, and will move the wth element of each piece for $w = \lfloor y/\sigma_j \rfloor$. Procedure FirstPiece$(j, y)$ has to compute the wth element of the first piece of this cycle, NextPiece(x) has to compute the wth element of the next piece when given x, the wth element of one piece.

Procedure FirstPiece works differently for $\sigma_j \leq p$ and $2p \leq \sigma_j \leq N/2$. If $\sigma_j \leq p$, then $k = (z + ip) \bmod \sigma_j = z \bmod \sigma_j$, as both σ_j and p are powers of two. The entry element c_k has the form $(-1)^{z \bmod 2} 5^{\lfloor (z \bmod \sigma_j)/2 \rfloor}$. Hence, it can be computed as $(-1)^{z \bmod 2} \cdot t_3[\log \sigma_j - 1] \bmod 2^{u-j}$.

To find the wth element of the first piece, the entry element has to be multiplied with b^w. As $y = z + ip$, $w = \lfloor y/\sigma_j \rfloor = \lfloor z/\sigma_j \rfloor + i(p/\sigma_j)$. If $i = 0$, b^w can be computed as $t_5[\log \sigma_j] \bmod 2^{u-j}$. Otherwise, it can be computed by multiplying b^{w-p/σ_j} (computed by virtual processor $z + (i-1)p$ on the same physical processor) and b^{p/σ_j}, which can be obtained from $t_0[\log p - \log \sigma_j]$.

If $2p \leq \sigma_j \leq N/2$, then $k = z + (ip \bmod \sigma_j) = z + p \cdot (i \bmod (\sigma_j/p))$. If $i \bmod (\sigma_j/p) = 0$, then $c_k = (-1)^{z \bmod 2} 5^{\lfloor z/2 \rfloor}$, which can be computed as $(-1)^{z \bmod 2} \cdot t_3[\log p - 1] \bmod 2^{u-j}$. Otherwise, c_k can be computed by multiplying $c_{k'}$, where $k' = z + p \cdot ((i-1) \bmod (\sigma_j/p))$ (computed by virtual processor $z + (i-1)p$ on the same physical processor) and $5^{p/2}$, which can be obtained from $t_1[\log p - 1]$.

To find the wth element of the first piece, we have to compute b^w. As $\sigma_j \geq 2p$, $w < v \leq L/2$. Hence, b^w can be obtained from $t_2[w]$.

Each piece of each cycle in phase j has length $v = N/\sigma_j$. Hence, the result of NextPiece(x) must be $b^v x$. As v is a power of two, one only needs to compute $t_0[\log N - \log \sigma_j] \cdot x \bmod 2^u$.

If $b = -1 \bmod 2^{u-j}$, then FirstEntry has to compute 5^y which can be obtained from $t'_4[\lfloor y/p \rfloor] \bmod 2^{u-j}$. NextEntry has to compute $x \cdot 5^N \bmod 2^u$, 5^N can be obtained from $t_1[\log N]$. In procedure FirstPiece, only the computation of c_k is changed. If $\sigma_j \leq p$, then c_k can be obtained from $t'_3[\log \sigma_j - 1]$. If $2p \leq \sigma_j \leq N/2$, then c_k can be obtained from $t'_3[\log p - 1]$ if $i \bmod (\sigma_j/p) = 0$ and from $c_{k'}$ and $5^p = t_1[\log p]$ otherwise. The computation of the wth element in procedure FirstPiece and procedure NextPiece remain unchanged. □

3.2 Example

We illustrate the rehashing algorithm by an example. We will assume $m = 2^6 = 64$, $p = 2^2 = 4$ and $L = 2$. Hence $N = Lp = 8$ and $f = 3$. If $h(x) = 19x \bmod 64$ and the new hash function is $h'(x) = 13x \bmod 64$, the permutation is $\pi(x) = 31x \bmod 64$, as $19^{-1} = 27$ and $27 \cdot 13 = 31$. The cycle structure of π is shown in Figure 2. The numbers in the left column show the phase in which the cycles are permuted, the numbers in the top row show the number of the processor that permutes the entry element of the cycles in that column.

We start by computing the eight tables and the l_j and σ_j. The results are shown in table 1.

In phase $j = 0$, each processor is assigned two cycles, each of length two. The procedure FirstEntry computes the following values for processors 0 to 7: $c_0 = 1$, $c_1 = 63$, $c_2 = 5$, $c_3 = 59$, $c_4 = 25$, $c_5 = 39$, $c_6 = 61$, $c_7 = 3$. The procedure NextEntry computes $c_8 = c_0 \cdot 49 = 49$, $c_9 = 15$, $c_{10} = 53$, $c_{11} = 11$, $c_{12} = 9$, $c_{13} = 55$, $c_{14} = 45$, $c_{15} = 19$.

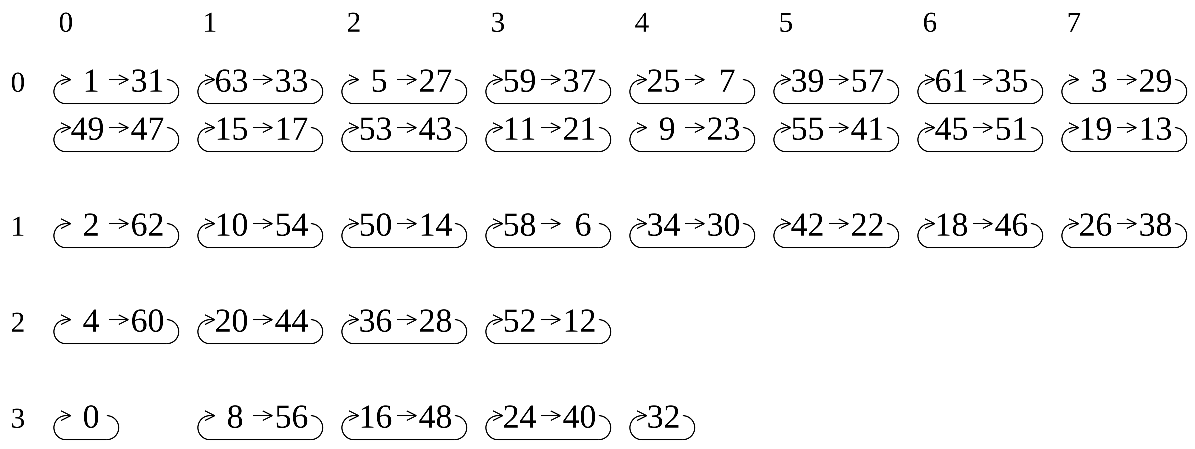

Figure 2: Cycle Structure of the example permutation

	0	1	2	3	4	5
t_0	31	1	1	1	1	1
t_1	5	25	49	33	1	1
t_2	0	5				
t_3, $z = 0, 1$	1	1				
t_3, $z = 2, 3$	1	5				
t_4, $z = 0, 1$	1	25				
t_4, $z = 2, 3$	5	61				
t_5, $z = 0, 1$		1	1			
t_5, $z = 2, 3$		31	1			
t'_3, $z = 0$	1	1				
t'_3, $z = 1$	5	5				
t'_3, $z = 2$	1	25				
t'_3, $z = 3$	5	61				
t'_4, $z = 0$	1	49				
t'_4, $z = 1$	5	53				
t'_4, $z = 2$	25	9				
t'_4, $z = 3$	61	45				
l_j	2	2	2	2	2	1
σ_j	16	8	4	2	1	1

Table 1: Precomputed tables for example permutation

In phase $j = 1$, each processor is assigned one cycle of length two. As $31 = -1 \bmod 32$, FirstEntry computes values different from those in phase 0: $c_0 = 2$, $c_1 = 10$, $c_2 = 50$, $c_3 = 58$, $c_4 = 34$, $c_5 = 42$, $c_6 = 18$, $c_7 = 26$.

In phase $j = 2$, two processors are assigned to one cycle of length two, hence there is only one piece and NextPiece is not needed. FirstPiece computes 4, 20, 36, 52, 60, 44, 28, 12.

In the final phase, the processors load the values 0, 8, 16, 24, 32, 40, 48, 56 and move them to their destination.

3.3 Analysis

We will now analyze the runtime and the memory requirements of the rehashing algorithm. The results are summarized in Theorem 3.

Theorem 3 *Rehashing of linear hash functions on a PRAM emulation with p processors, network latency L and a shared memory of size $m = 2^u$ can be done in time $O(m/p + \log m + L)$. Each processor needs local storage of size $O(\log m + L)$.*

If we only consider polynomial time algorithms, we can assume that m is polynomial in p. Furthermore, there are PRAM emulations with $L = O(\log p)$ [2, 15]. With these assumptions the runtime is $O(m/p + \log p)$, the storage requirements are $O(\log p)$.

Proof: We assume that multiplication, shifts of integers, $\lfloor \log_2(x) \rfloor$ for positive integers x and $x \bmod 2^{u-j}$ can be computed in one instruction.

All operations during the preprocessing phase work on local memories. Computing tables t_0 and t_1 takes time $O(\log m)$ and $O(\log m)$ space per physical processor. Computing tables t_3, t'_3 and t_5 takes time and space $O(\log p) = O(\log m)$, because $m \geq p$. Computing tables t_2, t_4 and t'_4 takes time $O(L)$ and $O(L)$ space per physical processor. Computation of l_j and σ_j for all j requires one traversal of table t_0 and takes time and space $O(\log m)$.

Hence, the runtime of the preprocessing phase is $O(\log m + L)$, and so are the space requirements. Also the preprocessing phase guarantees that all entry elements can be found in constant time per element during the rehashing phases, as shown in Lemma 2.

The rehashing phases (including the final phase) are completely parallelized as all cycles of one phase have equal length. The total work is $\Theta(m)$ and hence the runtime of the rehashing phases is $O(m/N)$ rounds of the emulation. Each of these rounds takes L steps. Thus for an arbitrary network with p pipelined processors and latency L the runtime will be $m/p + L$ steps. The rehashing phases need $O(L)$ space per physical processor as each virtual processor only needs a constant amount of local storage during rehashing.

The total runtime is $O(m/p + \log m + L)$. The storage requirements are $O(\log m + L)$ per physical processor. $\square$

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 steps to predict the next x steps.

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 steps within the last x steps. By stalled steps we mean steps where the active virtual processor cannot execute an instruction because it waits for an answer to a read request with latency larger than L.

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 total number of machine steps, 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.

The counter $\mathrm{CO}_{\mathrm{TO}}$ is normally present in the system as a timer, the counter $\mathrm{CO}_{\mathrm{ST}}$ 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 $\mathrm{CO}_{\mathrm{ST}} = \mathrm{CO}_{\mathrm{TO}} - R$.

5 Simplification of the algorithm

One might think about simplifying the algorithm for phases 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 phase 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 4 shows that the probability of a small loss of performance is quite small.

Theorem 4 *Let T_0 and T_1 be the runtimes of the original and the simplified algorithm for a randomly chosen b. Then*

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

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

After choosing an element b, the quotient T_1/T_0 can be computed in time $O(\log m)$. One might think to increase $\mathrm{Prob}(T_1/T_0 \le \delta)$ by repeatedly choosing b until $T_1/T_0 \le \delta$ 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 4 relies on the distribution of orders of elements in $U_0(m)$. This distribution is given in the following Lemma 5.

Lemma 5 *If we randomly choose an element b of $U_0(m)$, then its order can be 2^j, where $0 \le j \le u-2$. Furthermore,*

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

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 group $U_0(m)$ is generated by -1 and 5, their orders are 2 and 2^{u-2} respectively. Hence, each $x \in U_0(m)$ has a unique representation $x = (-1)^\alpha 5^\beta \bmod m$, where $\alpha \in \{0,1\}$ and $\beta \in \{0,\ldots,2^{u-2}-1\}$. We define $\psi(x) = (\alpha, \beta)$.

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 $b_2 \ne 0$, 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 $\varphi(2^j)$ (the Euler function) [17, p. 119]. If $b_2 \ne 0$ and hence $\mathrm{ord}(b_2) \ge 2$, 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 $2\varphi(2^j)$ if $j \ge 2$, $2\varphi(2) + 1$ 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 $\varphi(P^r) = (P-1)P^{r-1}$ for a prime P and an integer r [17, p. 120], Equation (3) follows. □

Proof of Theorem 4: 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

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

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

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

We now show that $l_{j+1} \geq l_j/2$. As l_{j+1} is the order of b in $U_{j+1}(m)$, $b^{l_{j+1}} \bmod 2^{u-(j+1)} = 1$ and hence $b^{l_{j+1}} = 1 + i \cdot 2^{u-(j+1)}$ for some i. Then

$$(b^{l_{j+1}})^2 \bmod 2^{u-j} = (1 + 2i \cdot 2^{u-(j+1)} + i^2 \cdot (2^{u-(j+1)})^2) \bmod 2^{u-j} = 1\,.$$

It follows that the order l_j of b in $U_j(m)$ is at most $2l_{j+1}$.

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 $\mathrm{Prob}(B/T_0 > \delta)$. $B/T_0 > \delta$ 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 $\mathrm{ord}(b) = l_0 = 2^x$, we get

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

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

6 Conclusions

Under reasonable assumptions for memory size and network latency, 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. The counters needed for the detection of rehashing are present at system level or can be implemented in software. 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 rounds and base their future behavior 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 rounds of the PRAM application and c rounds 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 Elwert 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.* **36** (1993) 756–762.

[2] F. Abolhassan, J. Keller and W. J. Paul, On the cost–effectiveness of PRAMs, in: *Proc. 3rd Symp. on Parallel and Distributed Processing* (IEEE CS Press, Los Alamitos, CA, 1991) 2–9.

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

[4] S. G. Akl, *The Design and Analysis of Parallel Algorithms* (Prentice–Hall, Englewood Cliffs, NJ, 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* (ACM, New York, 1990) 1–6.

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

[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*, Lecture Notes in Computer Science, Vol. 694 (Springer, Berlin, 1993) 1–11.

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

[9] J. JáJá, *An Introduction to Parallel Algorithms* (Addison Wesley, Reading, MA, 1992).

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

[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* (Elsevier, Amsterdam, 1990) 869–941.

[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** (1992) 63–79.

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

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

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

[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* (MIT Press, Cambridge, MA, 1988) 71–93.

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