Balanced PRAM Simulations via Moving Threads and Hashing

Ville Leppänen
(Department of Computer Science, University of Turku, Finland
Ville.Leppanen@cs.utu.fi)

Abstract: We present a novel approach to parallel computing, where (virtual) PRAM processors are represented as light-weight threads, and each physical processor is capable of managing several threads. Instead of moving read and write requests, and replies between processor&memory pairs (and caches), we move the light-weight threads. Consequently, the processor load balancing problem reduces to the problem of producing evenly distributed memory references. In PRAM computations, this can be achieved by properly hashing the shared memory into the processor&memory pairs.
We describe the idea of moving threads, and show that the moving threads framework provides a natural validation for Brent's theorem in work-optimal PRAM simulation situations on mesh of trees, coated mesh, and OCPC based distributed memory machines (DMMs). We prove that an EREW PRAM computation $\mathcal{C}$ requiring work W and time T, can be implemented work-optimally on those p-processor DMMs with high probability, if $W = \Omega(p \cdot T \cdot max(D, \log p))$, where D is the diameter of the underlying routing machinery. The computation is work-optimal regardless how (virtual) PRAM processors terminate or initiate new PRAM processors during the computation.
Our result is based on using only one randomly chosen hash function and on showing, how the threads (PRAM processors) can spawn new threads in required time on p-processor OCPC, 2-dimensional mesh of trees, 2-dimensional coated, and 3-dimensional coated mesh. A deterministic spawning algorithm is provided for all cases, although a randomized algorithm would be sufficient due to the randomized nature of time-processor optimal PRAM simulations.
Key Words: Balanced workload, moving threads, EREW, shared memory, simulation, work-optimal, randomized, OCPC, mesh of trees, coated mesh.
Category: C.1.2 C.2 C.5 F.1 F.2 G.3.

1 Introduction

This paper presents a new kind of framework for the PRAM (Parallel Random Access Machine) implementations. The idea is simply to represent each PRAM processor as a light-weight thread – just a small set of registers – and to implement references to non-local memory locations by moving the threads instead of read and write requests and replies. Besides simplifying the routing process in PRAM simulations, the approach also provides a natural way to keep the utilization of individual processors in balance.

1.1 Motivation

When PRAM models or PRAM algorithms are implemented, the cost of implementation comes essentially from three different sources: (1) The maximum load per processor, (2) the maximum memory congestion per memory module, and

(3) the diameter and capacity of routing machinery between the processors and the memory modules. In the moving threads framework, we eliminate the first by reducing it to the second. Since there exists a rich theory of hashing the shared memory into (local) memory modules of processors so that each memory reference pattern (produced by the PRAM processors) causes a roughly even load to each memory module, we consequently have a method to keep the workload of each real processor roughly in balance. In fact, we do not need to constantly observe, whether or not the workload of individual processors is in balance or not, since because of the framework, the workload stays in balance with high probability. Moreover, we do not even need to specify the allocation of PRAM processors (or threads) on the real processors, since the shared memory hashing strategy together with shared memory references that the PRAM processors make, does that automatically.

To our knowledge, balancing the workload of processors in context of the PRAM simulation has been done either by doing the balancing on the PRAM side or on the DMM side but not by providing the balance automatically through an implementation framework. Thus, this work represents a completely new approach to balancing processor allocation. We would like to claim that it is not wise to do explicit balancing on the PRAM side, since the advantage gained may be lost, when the PRAM is implemented on a DMM.

1.2 The problem and our solution

Purpose of a parallel algorithm description is to define a set of threads, which together, in cooperation, solve the computational problem in question. A parallel algorithm precisely describes the instruction stream of each thread. An efficient implementation of parallel algorithms requires that the need for communication and processing power is well in balance with the system resources. Moreover, if the topology of the parallel computer is very regular (e.g., as in the mesh), then the utilization of both resources should be as homogeneous as possible. However, during a parallel computation the number of threads (or processes or virtual processors) $nt(\tau)$ may vary dynamically as a function of the step counter τ. Existing threads can terminate or dynamically spawn new threads. New jobs with varying number of threads and execution time might enter to the system more or less unexpectedly. To keep the utilization of processing power in balance, each of the p real physical processors should have approximately $nt(\tau)/p$ threads on its responsibility at step τ. (We assume that each thread causes an equal mount of work per time unit to the executing physical processor, since we are mainly interested of PRAM algorithms working synchronously.)

Theorem Brent, [Brent 1974] *If a computation $\mathcal{C}$ can be performed in time T with W operations and sufficiently many processors which perform arithmetic operations in unit time, then $\mathcal{C}$ can be performed in time $T + (W - T)/p$ with p such processors.*

Brent's famous theorem tells that it is possible to balance the workload of real processors. Unfortunately, Brent's theorem does not tell, how threads can be allocated, if $nt(\tau)$ and the instruction streams of threads are not known beforehand. Moreover, Brent's theorem makes an assumption about the nature of cooperation: Arbitrary (but semantically well-defined) communication patterns between threads, consequently between processors, can be implemented with unit cost per thread. In the connection of PRAM models, the latter is naturally achieved.

Currently, there exists a quite rich theory of dynamically balancing the workload of each PRAM processor; see e.g., [Gil 1991, Gil, Matias 1991, Gil et al. 1991, Goodrich 1991, Hagerup 1992, Matias, Vishkin 1991]. For example, consider a parallel algorithm $\mathcal{A}$ that takes time $T(q)$ using q PRAM processors, and for which $nt(\tau) \geq nt(\tau + 1)$ for all $\tau \in \{1, \ldots, T(q)\}$. In [Matias, Vishkin 1991], it is proved that there is a work-optimal simulation of $\mathcal{A}$ on a PRAM with p ($\leq q$) processors in time $O(T(p) + \log^* q \log\log^* q)$ so that the processor-time product asymptotically equals (with high probability) to the number of operations required by $\mathcal{A}$ with q processors.

However, balancing a computation on the PRAM side does not necessarily guarantee efficient utilization of the processor and communication resources, since it only attempts to achieve high processor utilization. The communication resources are needed, since the only method to actually implement PRAM models seems to be simulation on distributed memory machines (DMMs), and therefore implementation of non-local memory references requires communication. Even if the computation is balanced on the PRAM side, the memory references caused by the program at each step might be severely in imbalance on the DMM side. This can result in bad processor&memory pair utilization, and consequently bad processor utilization. Luckily, there exist good shared memory hashing strategies [Carter et al. 1979, Dietzfelbinger et al. 1991, Dietzfelbinger et al. 1994], which guarantee even distribution for all reference patterns with high probability. These are used in many work-optimal PRAM simulations [Goldberg et al. 1994, Leppanen 1996, Leppanen, Penttonen 1995, Luccio et al. 1988, Ranade 1991]. By implementing a PRAM processor utilization strategy on top of a work-optimal PRAM simulation, one can expect asymptotically good overall utilization of resources.

Our improvement to the situation is an observation that we do not need to explicitly strive for keeping the processor side on balance, if we reduce the problem of thread balancing to the problem of producing evenly distributed memory reference patterns. We can achieve this by applying hashing to the shared memory and moving light-weight threads instead of the read and write requests and the replies. Consequently, we have a practical method that automatically keeps the utilization of system resources in balance on the DMM side. The only requirement of good utilization is the same as for work-optimal PRAM implementations in general: On the PRAM side we must, on average, have parallel slackness [Valiant 1990] at least proportional to the diameter of the routing machinery and the expected memory module congestion. Notice that no matter how evenly the threads are distributed among the processors, we are on the mercy of the maximum memory module congestion, if all or most of the processors make a non-local memory reference during each step.

We continue by giving an overview of the moving threads based computation on distributed memory machines in Section 2. In Section 3, we discuss the implementation of an elementary operation, namely spawning new threads. Then in Section 4, we show that EREW PRAM algorithms can be simulated work-optimally on certain distributed memory machines, although the termination of old threads and the spawning of new ones is not known in advance. We draw conclusions, and propose some topics for further research in Section 5.

2 Preliminaries

First, we give an overview of the moving threads based computation, and then in Section 2.2, we define the distributed memory machines used in this paper.

2.1 Overview of moving threads framework

In moving threads based computation [see Fig. 1], a thread is executed on some processor $\mathcal{P}_i$, and transferred via an interconnection network to another processor $\mathcal{P}_j$, when it makes a reference to a memory location stored at the local memory of $\mathcal{P}_j$. Besides the usual load, store, arithmetic, and logical operations, we assume that each thread can do a halt and a spawn operation in unit time. The halt operation causes the thread to terminate – i.e., it it simply destroyed. The spawn(x, id, loc) has three arguments. It creates x (> 0) new threads, whose identities (PIDs) are $id, id + 1, \ldots, id + x - 1$. The next instruction of those new threads is at location loc of the program. (For simplicity, we assume that each processor has a local copy of the program(s).) In addition to the virtual processor identifier, PID, each thread consists of a program counter (PC), a next instruction register (INSTR), an accumulator register (ACC), and a small set of other registers. We do not introduce synchronization primitives to the threads but rather pass to the physical processors and the routing machinery the responsibility to maintain certain kind of synchrony.

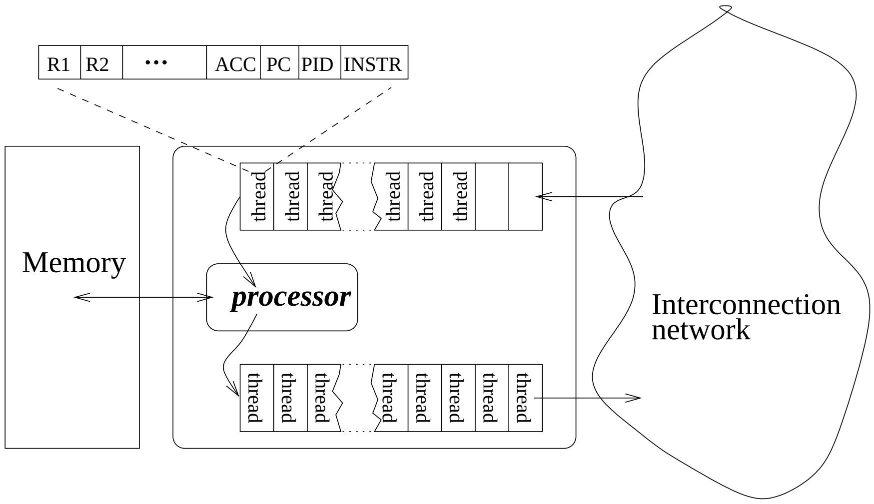

Figure 1: Function of a processor in moving threads based computation.

In this paper, we concentrate on the simulation of PRAM models. However, the moving threads framework applies for other approaches to general purpose parallelism as well. For example, if we aim to implement the sequential memory consistency model [Mosberger 1993], then we are done. To implement stronger atomic consistency models requires that we have a method to keep two consecutive (PRAM) steps separated. One possibility would be to use ghost packets [Ranade 1991] in the processors and routing machinery nodes in the form of synchronization wave technique [Leppanen 1996]. Observe that in the moving threads framework, we only need one routing phase (no replies) per simulated step. On the other hand, the packets (threads) are now larger.

Certain similarities between the moving threads framework and other recently proposed approaches to parallelism can be found. In the *active messages* communication mechanism [Saavedra-Barrera et al. 1990, von Eicken et al. 1992], each message contains a reference to a user-level handler that is executed on the message arrival (to the destination processor) with the message as an argument. In the moving threads framework, there is only one fixed handler. The active messages approach is designed for current message passing multiprocessors, and it aims to improve the efficiency of message handling and to minimize communication overhead. Another approach to balance the workload of processors is *work stealing* [Blumofe, Leiserson 1994, Blumofe, Papadopoulos 1998], where processors needing work steal threads from other processors. Stealing requires ownership – in the moving threads approach processors do not own threads. In the MuSE (Multithreaded Scheduling Environment) runtime environment [Leberecht 1996] the idea of moving threads (tokens of dataflow graphs) also appears. Another recent approach similar to the moving threads is the theoretical model of computation *spatial machine* [Feldman, Shapiro 1992]. The spatial machine is essentially a 3-dimensional grid of cells, where processors can move to adjacent cells and change the contents of cells.

The moving threads approach is new and the feasibility of it is open. Understandably, current parallel machines are not directly suitable for the moving threads approach. The feasibility and implementation issues are discussed in [Leppanen 1996].

2.2 Definition of DMMs

Next, we give a definition for OCPC [Goldberg et al. 1993] (an optical variant of the complete graph [Leighton 1992]), 2-dimensional mesh of trees [Leighton 1992, Leppanen 1995], and d-dimensional coated meshes [Leppanen, Penttonen 1995].

2.2.1 OCPC

The processors of a p-processor completely connected Optical Communication Parallel Computer, $OCPC(p)$, can send a packet to any other processor, but a prerequisite of successful receiving is that only one processor tries to send to the receiving processor at a given step. If several messages are sent to a processor at a given step, the processor will receive none of them. The underlying topology of OCPC is the complete graph. Each processor has a local memory module.

2.2.2 Mesh of trees

A 2-dimensional p-processor *Mesh of Trees* $MT(p)$ [see Fig. 2] is based on a $p \times p$ mesh of nodes (without grid edges). To the i'th row (and to the j'th column) is attached a complete binary tree, *row tree* RT_i (respectively *column tree* CT_j), whose leaves are the nodes on the i'th row (respectively on the j'th column). The edges of complete binary trees are bidirectional. The MT contains no other edges. The degree of MT is 3, the diameter is $4\log p$, and the number of nodes is $3p^2 - 2p$. We assume (as in [Luccio et al. 1988]) that the roots of row tree RT_i and column tree CT_i are merged for all i.

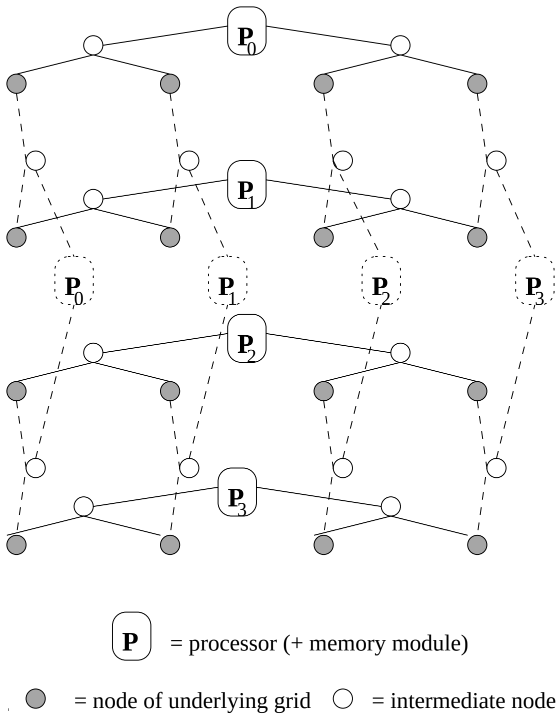

Figure 2: A 2-dimensional 4-sided mesh of trees.

2.2.3 Coated mesh

A d-dimensional p-processor *coated mesh* $CM(p, d)$ [Leppanen, Penttonen 1995] consists of a regular d-dimensional $n = \sqrt[d-1]{p/2d}$-sided mesh-connected routing machinery, which is coated from all sides with n^{d-1} processors [see Fig. 3]. The degree of routing machinery nodes is $2d$, and the diameter is $d(n-1)$. All connections between nodes and processor&memory pairs are bidirectional.

3 Arbitrary spawning on various DMMs

Next, we prove that an OCPC and a 2-dimensional mesh of trees as well as 2-dimensional and 3-dimensional coated meshes can deterministically implement arbitrary spawning of new threads.

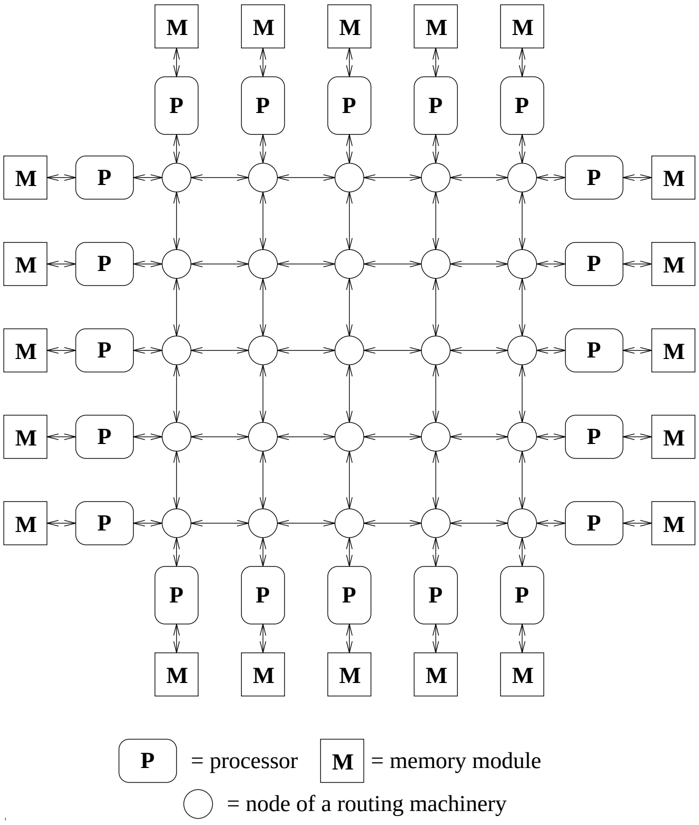

Figure 3: A 2-dimensional coated mesh with 20 processors.

In the following, we consider only implementing the spawn operations. For simplicity, we implicitly make the naive assumption that the threads issuing other than a spawn operation are executed (trivially) first and the spawn operations are then solved collectively. We believe that in practice this kind of separation can and should be avoided [Leppanen 1996].

3.1 Deterministic spawning algorithm

Consider a situation, where each of the N_{old} existing threads can issue an arbitrary spawn operation. Assume that the total number of new threads created is N_{new}, and originally each of the p real processors has $O(N_{old}/p+\log p)$ threads in its custody. We do the spawning deterministically with the following algorithm.

Algorithm 1 *(Deterministic spawning of new threads.)*
Let processor $\mathcal{P}_i$ have a_i threads $T_i^0, \ldots, T_i^{a_i-1}$ in its custody. Assume that T_i^j issues spawn operation $\texttt{spawn}(b_{i,j}, id_{i,j}, loc_{i,j})$ *at the current step. If T_i^j creates no threads, set $b_{i,j} = 0$.*

1. *$\mathcal{P}_i$ calculates $S_i = \sum_{j=0}^{a_i} b_{i,j}$.*
2. *Calculate prefix sums over the S_i-values, and broadcast $N_{new} = \sum_{i=0}^{p-1} S_i$ to all processors.*
3. *Spread the new spawn operations.*
4. *$\mathcal{P}_i$ creates its (at most $\lceil N_{new}/p \rceil$) new threads.*

In each case, the processors first calculate, how many new threads their current threads spawn. This is done deterministically in time $max\{a_i | 0 \le i < p\} = O(N_{old}/p + \log p)$. Then, the processors calculate prefix sums over the number of new threads to be spawn, and broadcast the total number to each processor. Using that information the task of spawning can be evenly distributed. Finally, the new tasks are created in time $O(N_{new}/p)$.

On a p-processor OCPC and mesh of trees, the prefix sums and broadcasting can be done in $O(\log p)$ routing steps, although on the mesh of trees some of the nodes are required to do certain elementary operations besides redirecting incoming packets. On a 2-dimensional p-processor coated mesh, the same can be done in $O(p)$ routing steps, and on a 3-dimensional coated mesh in $O(\sqrt{p})$ routing steps [Leppanen, Penttonen 1995]. All these results are based on embedding a binary tree (with processors as leaves) to the underlying graph.

All that remains to be shown, is how to efficiently inform each real processor, what threads it should create – we call this *spreading of new threads*. In the following (Lemmas 3 – 6), we prove that it can be done deterministically in $O(N_{old}/p + N_{new}/p + D + \log p)$ routing steps, where D is the graph-theoretic diameter of the underlying routing machinery. (The additive term "$\log P$" might seem odd. It is needed, since the OCPC has constant diameter and achieving time-processor optimal EREW simulation (by using only one hash function) on it requires parallel slackness $\Omega(\log p)$.) Consequently, we have Theorem 2. Observe that if $\frac{N_{old}+N_{new}}{p} = \Omega(D + \log p)$, then implementing the spawn operations requires constant work per thread.

Theorem 2. *Distributed memory machines $OCPC(p)$, $MT(p)$, $CM(p,2)$, and $CM(p,3)$ having $O(N_{old}/p + \log p)$ threads per processor can execute arbitrary spawning of N_{new} new threads in $O(N_{old}/p + N_{new}/p + D + \log p)$ routing steps so that each processor will receive $\lfloor N_{new}/p \rfloor$ or $\lceil N_{new}/p \rceil$ of the new threads. D is the graph-theoretic diameter of the underlying routing machinery.*

3.2 Spreading new threads

By the prefix computation, we assign to each $\texttt{spawn}(b_{i,j}, id_{i,j}, loc_{i,j})$ operation an interval $x_{i,j}, \ldots, x_{i,j} + b_{i,j} - 1$. Using the values of N_{new} and p, we get an interval $\mathcal{P}_k, \ldots, \mathcal{P}_l$ $(k \le l)$ of processors that collectively receive the $b_{i,j}$ new threads. Notice that the above information can be used to define exactly how many new threads each of the $l - k + 1$ destination processors receive. In the following, when we discuss spreading and splitting spawn operations, we assume that proper information is always attached to the spawn operations. We call *spreading problem*, the above problem, where the spawn operations are split and distributed so that each of the processors receives instructions to create $\lfloor N_{new}/p \rfloor$ or $\lceil N_{new}/p \rceil$ new threads.

3.2.1 OCPC

First we consider the OCPC, where each processor can receive at most one packet per step (from an arbitrary source). Notice that some processor can spawn all the N_{new} new threads, and therefore we must delegate the work of spreading the new threads. The spawn operations we send can contain instructions to create several threads, and those threads can be targeted to several processors. However, our packets are always of fixed size.

Lemma 3. *An OCPC(p) solves the spreading problem in time* $O(\frac{N_{old}}{p} + \frac{N_{new}}{p} + \log p)$.

Proof. After the prefix sum calculation, the processors can decide whom to send the spawn operations. The prefix sums also give us a collision free sending schedule. In general, the processor $\mathcal{P}_i$ needs to spread its spawn operations to processors $\mathcal{P}_k, \ldots, \mathcal{P}_l$ $(0 \le k \le l \le p-1)$. Each of the processors eventually receives at most $\lceil N_{new}/p \rceil$ spawn operations. Clearly, $\mathcal{P}_i$ can deliver all its spawn operations (or their subsets) destined to $\mathcal{P}_k$ in time $O(N_{old}/p + \log p) + N_{new}/p$, if no collisions take place. Collisions can be avoided by sending a spawn operation that creates threads $s_1, \ldots, s_2$ (of the at most $\lceil N_{new}/p \rceil$ threads) to $\mathcal{P}_k$ at time $T + s_x$, where T is the global beginning time of sending and $s_1 \le s_x \le s_2$. Similarly, $\mathcal{P}_i$ can deliver the spawn operations to $\mathcal{P}_l$ (on steps $T + \lceil N_{new}/p \rceil, \ldots, T + 2\lceil N_{new}/p \rceil - 1$). On steps $T + 2\lceil N_{new}/p \rceil, \ldots, T + 2\lceil N_{new}/p \rceil + O(N_{old}/p + \log p)$, the processor $\mathcal{P}_i$ simply forwards rest of the spawn operations to the processor in the middle of the destination area of each spawn operation. Those spawn operations that are destined to several processors are then split to two spawn operations (using information about N_{new}, p, and the destination area), which are sent to the middle of their destination area. This pipelined recursive decomposition process ends in time $O(\log p)$, and can clearly be made without collisions in packet receiving. □

3.2.2 Mesh of trees

The mesh of trees case is easier. Again, after the prefix sums stage each processor knows, where their spawn operations or continuous subsets of those should be delivered.

Lemma 4. *An MT(p) solves the spreading problem in time* $O(\frac{N_{old}}{p} + \frac{N_{new}}{p} + \log p)$.

Proof. The path from a processor to another down along a row tree and up along a column tree is unique. Therefore, it is straightforward for the processors and the row tree nodes to split arriving spawn requests (if the proper information is included) and forward them to their left and right subtrees. Once, a spawn request has arrived to a leaf of a row tree, it is simply forward up along the column tree to the destination processor. All requests arrive to their destinations in time $O(N_{old}/p + \log p + N_{new}/p)$, since none of the requests is delayed in the row trees (if the queues at the column tree edges are sufficiently large), and any request is delayed at most $\lfloor N_{new}/p \rfloor$ times while traversing up along a column tree. □

3.2.3 Coated mesh

To solve the spreading problem on coated meshes, we need to solve a certain partial h-h routing problem in a d-dimensional mesh based routing machinery. In the *partial h-h routing problem* each processor initially and finally has at most h packets. Lemma 5 states Kunde's partial routing results [Kunde 1993] in asymptotic form.

Lemma 5. *[Kunde 1993] A d-dimensional n-sided mesh can solve partial h-h routing problem deterministically in time $O(hn)$.*

Lemma 6. *Coated meshes $CM(p,2)$ and $CM(p,3)$ solve the spreading problem in time $O(\frac{N_{old}}{p}+\frac{N_{new}}{p}+p)$ and $O(\frac{N_{old}}{p}+\frac{N_{new}}{p}+\sqrt{p})$, respectively.*

Proof. First, consider the 2-dimensional case. By the prefix calculation, we can assign a destination processor $\mathcal{P}_{\delta(i,j)}$ for spawn operation $\texttt{spawn}(b_{i,j}, id_{i,j}, loc_{i,j})$. The processor $\mathcal{P}_{\delta(i,j)}$ is in the middle of (an array of consecutively numbered) processors for which the spawn operation creates the new threads. We solve the spreading problem by first injecting $\texttt{spawn}(b_{i,j}, id_{i,j}, loc_{i,j})$ to the closer half of the pile of routing machinery nodes that the processor $\mathcal{P}_i$ is connected. Then, we route each spawn operation via a certain node on the closer half of the pile of routing machinery nodes, attached to the destination processor, to the destination processor. Finally, we fix the result by splitting large spawn operations and by forwarding appropriate parts to neighboring processors.

Let $n = p/4$. The processor $\mathcal{P}_i$ can inject its $a_i = O(N_{old}/p + \log p)$ spawn operations evenly to the closer half of the pile of routing machinery nodes in time $O(a_i + n/2)$. Each routing machinery node clearly has at most

$$h_1 = 2 \times \lceil \frac{max\{a_k | 0 \le k < p\}}{n/2} \rceil = O(1 + N_{old}/p^2)$$

spawn operations. If an operation $\texttt{spawn}(b_{i,j}, id_{i,j}, loc_{i,j})$, producing the new threads $s_{i,j,\delta(i,j)}, \ldots, e_{i,j,\delta(i,j)}$ for $\mathcal{P}_{\delta(i,j)}$, is routed via the $(s_{i,j,\delta(i,j)} \; mod \; n/2)$'th node of an $n/2$ size destination area, then each routing machinery node is an intermediate target of at most

$$h_2 = 2 \times \lceil \frac{N_{new}/p}{n/2} \rceil = O(1 + N_{new}/p^2)$$

spawn operation. I.e., we have a partial $max(h_1,h_2)$-$max(h_1,h_2)$ routing problem that by Lemma 5 can be solved in time $O(N_{old}/p + N_{new}/p + p)$. Clearly, all the spawn operations can be delivered from their intermediate target to the target processor in time $O(N_{new}/p + n/2)$.

Observe that because of the prefix computation, each processor has at most two spawn operations that it has to split, and at most one to deliver both for the next higher and lower numbered neighboring processors. The processors and nodes on the surface of the 2-dimensional routing machinery form a ring, where all the large spawn operations can clearly be split and spread in time $O(p)$.

The 3-dimensional case is proved similarly. After the injection phase, we have a partial h-h routing problem, where $h = O(1 + N_{old}/p^{\frac{3}{2}} + N_{new}/p^{\frac{3}{2}})$. By Lemma

5, it is solved in time $O(N_{old}/p + N_{new}/p + \sqrt{p})$. The injection and removal phases are done in time $O(N_{old}/p + \sqrt{p})$ and $O(N_{new}/p + \sqrt{p})$, respectively. Finally, it is straightforward to show that splitting and spreading all the large spawn operations can be done in time $O(\sqrt{p})$. □

4 Automatically balanced EREW simulations

Simply using a greedy routing on a DAG based routing machinery with a synchronization technique is not enough to keep the work load of processors in balance with high probability in all cases. If the executed EREW program is such that after referring to a shared memory location x, each thread (simulating an EREW processor) makes only local arithmetic operations with register values, then all the threads end up to the real processor on whose custody is the location x. Sending each thread that makes no shared memory reference at the current step to a randomly chosen processor obviously solves the problem. Typically, routing results using congestion analysis techniques assume that there are no dependencies between the sources and destinations when the paths are chosen. Unfortunately, in the moving threads approach the simulated EREW program can cause such dependences.

To prove our main result, Theorem 9, we choose to "worsen" the situation a little. We simply use Valiant's idea [Valiant 1982] to choose an intermediate target for each thread and to split the original routing problem to two separate routing problems. Proving Theorem 9 requires that we have a result concerning simulation of one EREW step on various DMMs, when the distribution of simulated EREW processors is not known to be even. Results, where the simulated EREW processors are evenly distributed, are proved in e.g. [Leppanen 1995, Leppanen 1996, Leppanen, Penttonen 1995, Valiant 1990]. Those results assume that the simulated processors are evenly distributed among the simulating processors. However, it is straightforward to extend those results to uneven simulated processor distributions, and express them in the form of Lemma 7. The results are based on distributing the shared memory into the local memories of the DMM by randomly choosing a hash function from the family

$$\mathcal{H}^{\zeta}_{m,p} = \left\{ h \middle| h(x) = (\sum_{0 \le i < \zeta} a_i x^i \; mod \; q) \; mod \; p; 0 < a_i < q, p \le q = O(m) \right\},$$

where shared memory locations $\{0, \ldots, m-1\}$ are the domain of h, $\{0, \ldots p-1\}$ is the range, q is a prime, and $\zeta = \Omega(\log p)$.

Lemma 7. *[Leppanen 1995, Leppanen 1996, Leppanen, Penttonen 1995, Valiant 1990] Assume that the EREW processors are randomly distributed to the simulating processors. For each constant $\beta > 0$, there exist constants γ and κ such that one step of an N-processor EREW can be simulated in time $\gamma\lambda_{max} + \kappa D'$ on $OCPC(p)$, $MT(p)$, $CM(p,2)$, and $CM(p,3)$ with probability at least $1 - p^{-\beta}$, where λ_{max} is the maximum number of PRAM processors some DMM processor is simulating, and*

$$D' = \begin{cases} \log p \; for \; OCPC(p); \\ \log p \; for \; MT(p); \\ \sqrt{p} \quad for \; CM(p,3); \; and \\ p \quad\;\; for \; CM(p,2). \end{cases}$$

The constant β in Lemma 7 affects the simulation cost (through γ and κ), but can be chosen arbitrarily. To prove Theorem 9, we also need Lemma 8, which says that when N keys are hashed with $h \in \mathcal{H}_{m,p}^{\Omega(\log p)}$, none of the memory modules receives more than $O(N/p + \log p)$ keys with high probability.

Lemma 8. *[Dietzfelbinger et al. 1994] If a randomly chosen $h \in \mathcal{H}_{m,p}^{\zeta}$ is used for hashing a set S of unique memory locations into p modules, then for $|S| = n$ and for all i $(0 \leq i < p)$:*

$$Pr(b_i \geq u) \leq \left(\frac{e \cdot n}{u \cdot p}\right)^{min(u,\zeta)},$$

where $b_i = |\{x \in S \mid h(x) = i\}|$.

Next, consider an arbitrary PRAM computation $\mathcal{C}$ that with some number of PRAM processors requires time T and total work W. By high probability, we mean probability of the form $1 - p^{-\xi}$, where constant $\xi > 1$.

Theorem 9. *An arbitrary EREW PRAM computation $\mathcal{C}$ solving problem $\mathcal{A}$ in time T with work W using arbitrary number of EREW PRAM processors, can be simulated work-optimally on $OCPC(p)$, $MT(p)$, $CM(p,2)$, and $CM(p,3)$ with high probability, if $T = O(p^{\alpha})$ for some constant $\alpha > 0$ and $W = \Omega(p \cdot T \cdot D')$, where*

$$D' = \begin{cases} \log p \ for\ OCPC(p); \\ \log p \ for\ MT(p); \\ \sqrt{p} \quad for\ CM(p,3);\ and \\ p \quad\ for\ CM(p,2). \end{cases}$$

No matter how the PRAM processors are allocated and released during the computation.

Proof. Denote the number of active EREW processors at step τ with N_τ, and represent each EREW processor as a light-weight thread. Since we are simulating an EREW PRAM, we assume that the unique PIDs of active PRAM processors are between 0 and $m-1$. We use a randomly chosen $h \in \mathcal{H}_{2m,p}^{\eta \log p}$ to distribute the shared memory locations to the local memory modules of the DMM. If we move threads that make no memory reference at step τ to processor $\mathcal{P}_{h(m+x)}$, where x is the PID of the thread, then by Lemma 8 each memory module receives no more than $O(N_\tau/p + \log p)$ threads with high probability. By properly choosing η and the constant factors in $O(N_\tau/p + \log p)$, the probability of failing can be made smaller than $p^{-\alpha-\sigma}$ for any constant $\sigma > 1$.

Assume that routing the threads to their new target is done in two phases: (1) First route all the threads to a randomly chosen processor, and (2) then route the threads to their target. By Lemma 7, the first phase can be done in time $O(N_\tau/p + D')$ and the second phase in time $O(N_{\tau+1}/p + D')$, with high probability. By applying Theorem 2, we can conclude that at step τ the new threads can be spawned deterministically in time $O(N_\tau/p + N_{\tau+1}/p + D')$. The total work done by the simulation is thus

$$\begin{aligned} W' &= p \times \sum_{\tau=1}^{T} O(N_\tau/p + D') + O(N_{\tau+1}/p + D') + O(N_\tau/p + N_{\tau+1}/p + D') \\ &= p \times \sum_{\tau=1}^{T} O(N_\tau/p + D') = O(W) + O(p \cdot T \cdot D'). \end{aligned}$$

If $W = \Omega(p \cdot T \cdot D')$, the total work W' done by the simulating processors is $O(W)$ – i.e., the execution of computation $\mathcal{C}$ is work-optimal. □

Theorem 9 basically says that if the computation $\mathcal{C}$ on average has parallel slackness proportional to the maximum of the diameter and expected module congestion, then the simulations on those DMMs utilize a constant fraction of the peak performance throughout the computation $\mathcal{C}$. In asymptotic sense, this means that the Brent's theorem is valid for certain time-processor optimal EREW PRAM simulations.

5 Discussion

The moving threads approach is based on two assumptions: Fast exchange of messages between neighboring nodes, and fast hardware-supported multithreading (context switching). Neither is inherently problematic, but both are (somewhat) insufficiently supported by current parallel computers. However, the rapid development of optical communication and the huge potential bandwidth of it makes the moving threads approach interesting despite of the fast & high bandwidth communication assumptions.

Our main result in this paper is that the moving threads framework can be used to achieve a work-optimal implementation for EREW algorithms on certain distributed memory machines, if the EREW algorithms contain enough parallel slackness on average. The required amount of parallel slackness is the same as is previously proved for time-processor optimal simulations based on fixed number of EREW processors and one hash function. What is significant in our result is that nothing is assumed about how the number of PRAM processors – participating the computation – varies throughout the computation. Notice that the parallel slackness can come from several simultaneously running PRAM algorithms.

We dealt with the OCPC, mesh of trees, and coated meshes because of the deterministic spawning results shown for those architectures. We expect that such a result – or a corresponding randomized spawning result – can be shown for several other architectures, e.g., the P-direct butterfly, fat mesh, coated fat mesh, hypercube and mesh of optical buses. For the P-direct butterfly this might be done by solving the spreading problem as a sequence of $max\{a_k | 0 \leq k < P\}$ monotone routing problems; see [Leighton 1992, Section 3.4.3].

We would like to believe that in practice it is not necessary to move those threads that make no shared memory reference during the current step. We also believe that using the two-phase routing strategy in moving the threads to their new target is not necessary – as a consequence of the randomized hashing technique the source and destinations of memory references (and therefore the threads) should be random enough. Likewise, our deterministic spawning method is too expensive in practice, and a heuristic method is likely to be more practical. Moreover, we believe that one should route the spawn operations with the threads, and use some heuristic method to split the spawn operations evenly while they are being routed greedily towards heuristically chosen targets. It would be interesting to study the actual routing cost of continuous moving threads based PRAM simulation on an architecture, where a directed acyclic graph is embedded into the routing machinery, and consecutive PRAM steps are kept separate by the synchronization wave technique [Leppanen 1996, Ranade 1991].

Finally, one should notice that balancing the workload of processors is a problem that has been studied not only in the context of the PRAM approach to parallelism. It might be advantageous to apply the moving threads framework (together with the shared memory hashing techniques) to those approaches, too.

References

[Blumofe, Leiserson 1994] R.D. Blumofe and C.E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356 – 368, 1994.

[Blumofe, Papadopoulos 1998] R.D. Blumofe and D. Papadopoulos. The Performance of Work Stealing in Multiprogrammed Environments, 1998. Manuscript, submitted for publication.

[Brent 1974] R.P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[Carter et al. 1979] J.L. Carter and M.N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[Dietzfelbinger et al. 1991] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial Hash Functions Are Reliable. In *Proceedings, 18th International Colloquium on Automata Languages and Programming, LNCS 623*, pages 235 – 246, 1991.

[Dietzfelbinger et al. 1994] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738 – 761, August 1994.

[Feldman, Shapiro 1992] Y. Feldman and E. Shapiro. Spatial Machines: A More Realistic Approach to Parallel Computation. *Communications of the ACM*, 35(10):60 – 73, 1992.

[Gil 1991] J. Gil. Fast Load Balancing on a PRAM. In *Proceedings, 3rd IEEE Symposium on Parallel and Distributed Processing, ACM Special Interest Group on Computer Architecture, and IEEE Computer Society*, pages 10 – 17, 1991.

[Gil, Matias 1991] J. Gil and Y. Matias. Fast Hashing on a PRAM – Designing by Expectation. In *Proceedings, 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271 – 280, 1991.

[Gil et al. 1991] J. Gil, Y. Matias, and U. Vishkin. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *Proceedings, 32th Annual Symposium on Foundations of Computer Science*, pages 698 – 710, 1991.

[Goldberg et al. 1993] L.A. Goldberg, M. Jerrum, T. Leighton, and S. Rao. A Doubly Logarithmic Communication Algorithm for the Completely Connected Optical Communication Parallel Computer. In *SPAA'93, 5th Annual Symposium on Parallel Algorithms and Architectures, Velen, Germany*, pages 300 – 309, June 1993.

[Goldberg et al. 1994] L.A. Goldberg, Y. Matias, and S. Rao. An Optical Simulation of Shared Memory. In *SPAA'94, 6th Annual Symposium on Parallel Algorithms and Architectures, Cape May, New Jersey*, pages 257 – 267, June 1994.

[Goodrich 1991] M.T. Goodrich. Using Approximation Algorithms to Design Parallel Algorithms that May Ignore Processor Allocation. In *Proceedings, 32th Annual Symposium on Foundations of Computer Science*, pages 711 – 722, 1991.

[Hagerup 1992] T. Hagerup. The Log-Star Revolution. In *Proceedings, 9th Symposium on Theoretical Aspects of Computer Science, STACS'92, LNCS 577*, pages 259 – 278, 1992.

[Kunde 1993] M. Kunde. Block Gossiping on Grids and Tori: Deterministic Sorting and Routing Match the Bisection Bound. In T. Lengauer, editor, *Proc. of Algorithms ESA'93, LNCS 726*, pages 272 – 283. Springer-Verlag, 1993.

[Leberecht 1996] M. Leberecht. A Concept for a Multithreaded Scheduling Environment. In *Proceedings of the 4th PASA Workshop (GI/ITG), Germany*, pages 161 – 175, 1997.

[Leighton 1992] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufman, San Mateo, CA, 1992.

[Leppanen 1995] V. Leppänen. On Implementing EREW Work-Optimally on Mesh of Trees. *Journal of Universal Computer Science*, 1(1):23 – 34, January 1995.

[Leppanen 1996] V. Leppänen. *Studies on the Realization of PRAM.* PhD thesis, TUCS, Department of Computer Science, University of Turku, November 1996. TUCS Dissertation, No 3.

[Leppanen, Penttonen 1995] V. Leppänen and M. Penttonen. Work-Optimal Simulation of PRAM Models on Meshes. *Nordic Journal on Computing*, 2(1):51 – 69, 1995.

[Luccio et al. 1988] F. Luccio, A. Pietracaprina, and G. Pucci. A Probabilistic Simulation of PRAMs on a Bounded Degree Networks. *Information Processing Letters*, 28(3):141–147, 1988.

[Matias, Vishkin 1991] Y. Matias and U. Vishkin. Converting High Probability into Nearly-Constant Time – with Applications to Parallel Hashing. In *Proceedings, 22nd Annual ACM Symposium on Theory of Computing*, pages 307 – 316, 1991.

[Mosberger 1993] D. Mosberger. Memory Consistency Models. *Operating Systems Review*, 17(1):18 – 26, January 1993.

[Ranade 1991] A.G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[Saavedra-Barrera et al. 1990] R. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *SPAA'90, 2nd Annual Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece*, pages 169 – 178, Crete, Greece, July 1990. (Also available as Technical Report UCB/CSD 90/569, CS Div., University of California at Berkeley).

[Valiant 1982] L.G. Valiant. A Scheme for Fast Parallel Communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[Valiant 1990] L.G. Valiant. General Purpose Parallel Architectures. In *Algorithms and Complexity, Handbook of Theoretical Computer Science*, volume A, pages 943–971, 1990.

[von Eicken et al. 1992] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. (Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley).