Towards Better Algorithms for Parallel Backtracking

IB 6/95

Peter Sanders
Department of Computer Science
University of Karlsruhe, 76128 Karlsruhe, Germany
E-mail: sanders@ira.uka.de

January 14, 1995

Abstract

Many algorithms in operations research and artificial intelligence are based on depth first search in implicitly defined trees. For parallelizing these algorithms, a load balancing scheme is needed which is able to evenly distribute parts of an irregularly shaped tree over the processors. It should work with minimal interprocessor communication and without prior knowledge of the tree's shape.

Previously known load balancing algorithms either require sending a message for each tree node or they only work efficiently for large search trees. This paper introduces new randomized dynamic load balancing algorithms for *tree structured computations*, a generalization of backtrack search. These algorithms only need to communicate when necessary and have an asymptotically optimal scalability for many important cases. They work work on hypercubes, butterflies, meshes and many other architectures.

1 Introduction

Load balancing is one of the central issues in parallel computing. Since for many applications it is almost impossible to predict how much computation a given subproblem involves, we need dynamic load balancing strategies which are able to keep the processors (PEs) busy without incurring an undue overhead.

We discuss this in the following context (for a more detailed explication of our model refer to Section 2.1): We consider n PEs which interact by exchanging messages through a network of diameter d. The problems to be load balanced are *tree shaped computations*: Initially, there is only one large root problem. Subproblems can be generated by splitting existing problems into two independent subproblems; nothing is known about the relative size of the two parts. The only thing the load balancer knows about a subproblem is whether it is exhausted or not. The performance analysis is based on the total sequential execution time T_{seq} and the height h of the binary tree defined by splitting the root problem into atomic pieces.

One application domain for which this model is useful is parallel depth first tree search (Backtracking). Search trees are often very irregular and the size of a subtree is hard to predict, but it is easy to split the search space into two parts[1] Also, interactions between the subtrees often follow the tree structure (e.g. reporting results) or they are hard to exploit by a load balancer anyway (e.g. broadcasting of the current best solution or accessing distributed hash tables). Note that depth first tree traversal is a central aspect of many AI and OR applications and of parallel functional and logical programming languages.

[1]Note that the underlying search tree need not be a binary tree. In [16], a variety of heuristics for splitting the stacks representing a tree are presented. None of them is restricted to binary trees.

We now go on by introducing *Receiver induced tree splitting*, a simple and successful scheme for parallelizing tree shaped computations in Section 1.1 which is compared to other approaches found in the literature in Section 1.2. On overview of the remainder of the paper concludes this introduction.

1.1 Receiver induced tree splitting

The basic principle is that a PE works only on a single subproblem at a time and only activate the load balancer when this subproblem is exhausted. The load balancer supplies new subproblems by requesting other PEs to split their subproblem. Idle PEs receiving a request either reject the request or redirect it to another PE. Figure 1 shows pseudocode for such a generic tree splitting algorithm.

```
put the root problem on PE 0
DOPAR on all PEs
    WHILE not finished DO
        IF subproblem is empty THEN
            get new work from load balancer
        WHILE subproblem is not empty DO
            IF there is a load request THEN
                split subproblem
                send one part to the initiator of the request
            do some work on subproblem
```

Figure 1: Receiver induced tree splitting.

This approach has proved useful under a variety of circumstances [3, 16, 17, 6, 2, 13, 7, 5, 20, 18, 19]. A major advantage of receiver induced tree splitting is that load balancing only takes place when necessary. Also, in the beginning, the size (execution time) of transmitted subproblem will be fairly large; subsequent productive work done on the migrated subprolem will make up for the expense of communication. For sufficiently large problem sizes, most receiver induced tree splitting schemes can achieve efficiencies arbitrarily close to 1, i.e., the parallel execution time T_{par} can be written as $(1+\epsilon)\frac{T_{\text{seq}}}{n}+(\text{ lower order terms})$ for arbitrary $\epsilon > 0$. However, in practice it is crucial how the problem size has to be scaled with the number of processors in order to achieve a desired efficiency. In this respect there are large differences between different load balancing strategies.

For example, in [17] it is shown that sending requests to neighboring processors is quite inefficient except for the combination of a low diameter interconnection networks (e.g. hypercubes) and a work splitting function which produces subproblems of nearly equal size. The basic problem of these *neighborhood polling* schemes is that highly loaded PEs will quickly be surrounded by a cluster of busy PEs and are therefore unable to transmit work; subproblem transmissions at the border of these clusters only involve small subproblems which are not worth the effort of communicating them.

In [6, 7] a variety of other partner selection schemes is analyzed. There is a tradeoff between schemes based on local information which may produce many vain requests to idle processors, and global selection schemes which incur additional message traffic and often suffer from contention at centralized schedulers. *Random polling*, i.e., selecting communication partners uniformly at random is identified as a promising scheme. Good speedups are reported for up to 1024 processors. In

[19] it is proved that random polling works in time $(1+\epsilon)\frac{T_{seq}}{n} + O(dh)$ with high probability for crossbars, butterflies, meshes and many other architectures. This is asymptotically optimal for networks with constant diameter because the sequential component for following the maximum depth branch implies a lower bound of

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + h\right); \tag{1}$$

[19] also looks at the asymptotic influence of message lengths and atomic grain sizes of subproblems which we (like most other authors) assume to be constant throughout this paper.

On SIMD computers load balancing is done in separate load balancing phases initiated by some triggering condition [13, 5]. The best schemes use the ability of many SIMD computers to quickly compute prefix-sums: Communication partners can be matched by enumerating the busy and idle PEs respectively. Good speedups have been observed on up to 32K PEs. In [6], a similar idea is used to design a deterministic asynchronous load balancing scheme which is asymptotically as efficient as random polling. However, it does not perform as well in practice.

1.2 Other related work

Another important class of algorithms which are applicable to tree shaped computations are *dynamic tree embedding* algorithms [10, 15, 4]. Using our terminology, these algorithms are based on splitting the tree into a maximum number of atomic subproblems. The tree generated by this process is on-line embedded into the interconnection network.

Building on results from [10], it is shown in [15] how randomized dynamic tree embedding algorithms can be used to perform backtracking on butterflies and hypercubes in time $O(\frac{T_{seq}}{n} + h)$ with high probability. These algorithms achieve constant efficiency for problems of size $\Omega(nh)$ meeting the lower bound from Equation (1). However, if communicating an atomic subproblem is expensive compared to solving it, the efficiency of tree embedding algorithms is limited by a quite small constant factor and this figure does not improve for larger subproblems where algorithms like random polling can achieve very high efficiencies.

The situation is even worse if tree embedding is to be used on meshes because this is not possible with constant dilation. In [4], it is demonstrated how trees with $O(n)$ leaves can be deterministically embedded into an r-dimensional mesh in time $O\left(\sqrt[r]{nh}\right)$. Also a lower bound of $\Omega\left(\sqrt[r]{\frac{nh}{\log n}}\right)$ is proven. It is not clear however, how these results can be expanded for larger problem sizes.

On the other side of the spectrum, load balancing can be done with very little communication by broadcasting the root problem to all PEs and locally splitting it into individual pieces based on the PE number. Applied in a straightforward way, this technique leads to poor load balancing [1], but using it as an initialization for dynamic load balancers can yield a significant improvement. In [21], it is shown that for certain search trees with $h \in O(\log T_{\text{seq}})$ the combination of a randomized initialization scheme and a variant of random polling achieves execution times in $(1+\epsilon)T_{\text{seq}}/n + O(n^{1/r})$ on the average. This is asymptotically optimal because the diameter d of the network imposes a lower bound of

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + d\right) \tag{2}$$

on the parallel execution time. (Some subproblem must be transmitted every PE.) By randomly chopping the tree into much more pieces than PEs it is even possible to

devise an efficient static load balancing scheme for tree shaped computations which uses a single broadcast of the root problem as the only nonlocal operation. (Plus collecting results.)

1.3 Overview

The goal of this paper is to present receiver induced tree splitting algorithms which are as scalable as dynamic tree embedding schemes but retain the advantage of low communication overhead. Section 2 introduces the notation used and presents some basic lemmata.

Section 3 presents the main idea. The PEs perform receiver induced tree splitting; communication is done with neighbors in a hypercube. By synchronously iterating through the dimensions of the hypercube, it can be guaranteed that the load remains evenly distributed as long as "fresh" dimensions of the hypercube are available. When all dimensions are exhausted, the subproblems are randomly permuted and the cycle can start again. Additional subsections explain how random permutations can be determined efficiently, how the necessary synchronizations can be achieved using local interactions and how to port the algorithm to constant degree networks like butterflies. Execution times are in $(1+\epsilon)T_{\text{seq}}/n+O(h)$ with high probability.

In Section 4, the algorithm is adapted to r-dimensional meshes and fat trees. Execution times are in $(1+\epsilon)T_{\text{seq}}/n+O(h)n^{1/r}/\log n$ and $(1+\epsilon)T_{\text{seq}}/n+O(h)\sqrt{\log n}$ respectively with high probability. For these networks it is also possible to replace the synchronized phases by a simpler load balancing algorithm based on a local variant of random polling.

Finally, Section 5 evaluates the results by comparing them to the known lower bounds. A discussion of possible future work points out how the algorithms might be further developed.

2 Notation and Basic Results

2.1 Machine and Application Model

Let the n PEs be numbered 0 through $n-1$. A message packet can be communicated to a neighboring PE in unit time. We assume the packet switching model of communication, i.e., sending a packet to a PE k hops away takes time k. The network diameter is denoted by d.

Initially, a data structure describing the entire problem (the root problem) is located on PE 0. Let T_{seq} denote the root problem's sequential execution time or *size*. We do not want to look at very small problems; we assume that $T_{\text{seq}} \in \Omega(n)$. The *splitting function* is able to split a subproblem of size T into two subproblems of size T_1 and T_2 in unit time. An important assumption is that $T_1+T_2 = T$ regardless when and where the subproblems are processed[2]. A subproblem generated by h subsequent splits of the root problem is guaranteed to be reduced to a constant atomic size T_{atomic} or smaller. An immediate consequence of the above definitions is that

$$h \in \Omega(\log n).$$

[2]This excludes many important aspects of parallel search, for example, the influence of heuristics like branch-and-bound or $\alpha\beta$ or the speedup anomalies observed when search is stopped as soon as a solution is found. But the assumption is valid for many other applications and the algorithms analyzed here do not explicitly use it. An algorithm which is able to effectively do load balancing will often also work well if the problem has an additional speculative computation aspect.

Splitting an atomic subproblem yields the same subproblem plus an empty subproblem. We do not discuss termination detection and reporting results because they are not a bottleneck if implemented properly. Finally, we assume that a description of a subproblem fits into a single network packet.

2.2 Randomized Algorithms

The analysis of the randomized algorithms described here is based on the notion of behavior *with high probability*. Among the various variants of this notions we have adopted the one from [14].

Definition 1 *A random variable X is in $O(f(n))$* with high probability *— or $X \in \tilde{O}(f(n))$ for short — iff*

$$\exists c > 0, n_0 > 0 : \forall \beta \geq 1, n \geq n_0 : \mathbf{P}[X > c\beta f(n)] \leq n^{-\beta},$$

i.e., the probability that X exceeds the bound f by more than a constant factor a is polynomially small and a grows only linearly with the desired exponent. In this paper, the variable used to express high probability is always n — the number of PEs.

It is quite easy to derive high probability bounds for the maximum of random variables from known bounds for the individual variables:

Lemma 1 *Let $X_1 \in \tilde{O}(f_1(n)), \ldots, X_m \in \tilde{O}(f_m(n))$ be random variables where m is at most polynomial in n. Then [18]*

$$\max_{i=1}^{m} X_i \in \tilde{O}\left(\max_{i=1}^{m} f_i(n)\right)$$

This lemma is particularly important for parallel computing because it allows us to conclude from the behavior of an algorithm on one PE to the behavior on the "worst" of n PEs.

Finally, we need the following *Chernoff bounds* which are a keystone of many probabilistic proofs.

Lemma 2 (Chernoff bounds) *Let the random variable X represent the number of heads after n independent flips of a loaded coin where the probability for a head is p. Then [14, 9]:*

$$\mathbf{P}[X \leq (1-\epsilon)np] \leq e^{-\epsilon^2 np/3} \text{ for } 0 < \epsilon < 1 \tag{3}$$

$$\mathbf{P}[X \geq \alpha np] \leq e^{\left(1-\frac{1}{\alpha}-\ln\alpha\right)\alpha np} \text{ for } \alpha > 1 \tag{4}$$

3 Hypercube poll-and-shuffle

3.1 The basic algorithm

We are now looking at a $\log n$ dimensional hypercube network.[3]

In order to understand the algorithm, it is also useful to assume the existence of a globally synchronized clock for a moment. We now partition time into *phases* of constant length T_{phase}. Idle processors are only allowed to send requests after a phase. After phase number i, requests go to the neighbor along dimension i. When we have reached phase $\log n$, we are out of fresh dimensions for communication. Therefore, we randomly permute the subproblems and start a new *cycle* by resetting the phase counter to 0. Figure 2 shows this partitioning of the time line for $n = 2^4$

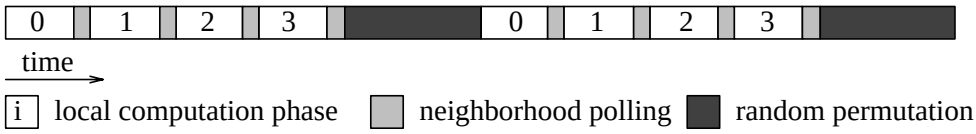

Figure 2: Two cycles of hypercube poll-and-shuffle for $n = 2^4$.

and 2 cycles. Using this schedule we can guarantee that after most phases with low PE utilization subproblems have a certain likelihood of receiving a request.

Lemma 3 *For any $\gamma \in (0,1)$, for any subproblem S, and for any phase with a number less than $\log n - \log \frac{2}{\gamma}$, if at any point during this phase at least γn PEs are idle, then after this phase S receives a request with a probability of at least $\gamma/2$.*

Proof: Since the number of busy PEs can only decrease during a phase, at least γn PEs will issue a request after the phase under consideration. Let $i < \log n - \log \frac{2}{\gamma}$ denote the number of the phase under consideration. During the current cycle, S can only have interacted with the $2^i < \frac{\gamma}{2}n$ PEs reachable over links $\{0, \ldots, i-1\}$. Therefore, there are at least $\frac{\gamma}{2}n$ idle PEs with which S did not interact in the current cycle. Due to the random permutation applied at the before a cycle[4], each of the subproblems which S did not interact with is equally likely to be S's neighbor along dimension i. Therefore, S will receive a request with probability[5] at least $\frac{\gamma}{2}$. ■

So, at the end of each cycle there is a constant number of phases about which we cannot say very much. The other phases either do productive work or they reduce the size of the remaining subproblems. Furthermore, if we make the phases sufficiently long, the time for doing productive work and issuing requests will dominate the time for routing the random permutations.

Lemma 4 *For any constant $\gamma > 0$, $\tilde{O}(h)$ phases with at least γn idle PEs are sufficient such that* every *subproblem receives at least h requests.*[6]

Proof: We first show that $\tilde{O}(h)$ phases are sufficient such that a particular subproblem S receives at least h requests. Let the random variable K_S denote the number of phases necessary such that S receives at least h requests. We need to find a c such that for all $\beta \geq 1$ and sufficiently large n

$$P := \mathbf{P}\left[K_S > \frac{c\beta h}{\gamma}\right] \leq n^{-\beta}$$

or

$$\mathbf{P}\left[\text{after } \tfrac{c\beta h}{\gamma} \text{ phases: } (\#\text{ of requests for } S) < h\right] \leq n^{-\beta}$$

[3] In this paper, log always means the base 2 logarithm.

[4] The initial cycle is a special case. Either we have the root problem at PE 0. In this case, moving it anywhere else would make no sense. Or we use a specialized initialization scheme along the lines of [21] which takes care of randomization.

[5] The probability space under consideration are sequences of permutations over PE numbers. Permutations are chosen uniformly at random and independently of the earlier permutations.

[6] We use the term "a subproblem receives a request" as a shorthand for "The PE where a subproblem is located receives a request." The proposition "A subproblem S receives k requests" means that S was generated by splitting (and possibly transmitting) a subproblem which received $k-1$ requests.

Since the phases are independent and subproblem S receives a request with probability at least γ, Lemma 2 is applicable. By writing h as $\left(1-\left(1-\frac{1}{c\beta}\right)\right)(c\beta\frac{h}{\gamma})\gamma$ we get

$$P \leq \exp - \left[\left(1-\frac{1}{c\beta}\right)^2 \frac{c\beta h}{3}\right]$$

Since $h \in \Omega(\log n)$, there is a constant $k > 0$ such that $h \geq k \ln n$ for sufficiently large n. Using $\beta \geq 1$ we can further estimate:

$$\begin{aligned} P &\leq \exp - \left[\left(1-\frac{1}{c}\right)^2 \frac{c\beta k \ln n}{3}\right] \\ &= n^{-\beta\left(1-\frac{1}{c}\right)^2 \frac{ck}{3}} \\ &\leq n^{-\beta} \text{ if } c \geq 1 + \frac{3+\sqrt{12k+9}}{2k} \end{aligned}$$

Now, using Lemma 1, we can conclude that *all* PEs obey this bound. ∎

Theorem 1 *Let T_{par} denote the execution time of the hypercube poll-and-shuffle algorithm. For every $\epsilon > 0$ there is a choice of the phase length T_{phase} such that*

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \tilde{O}(h) .$$

Proof: Let $\gamma \in (0,1)$ be a constant we are free to choose. In order to determine an appropriate value for T_{phase}, we consider it an additional variable. We first bound the number of phases with at most γn and with more than γn idle PEs respectively. There can be at most

$$\frac{T_{\text{seq}}}{T_{\text{phase}} n(1-\gamma)}$$

phases with high PE utilization since in this number of phases $n(1-\gamma)$ active PEs are able process the entire problem. Using the results of Lemma 4 we see that $\tilde{O}(h)$ phases with number $< \log n - \log \frac{2}{\gamma}$ are sufficient to reduce all subproblems to atomic size. If we choose $T_{\text{phase}} > T_{\text{atomic}}$ all work will be completed in the next phase. Putting this together we see that

$$\frac{\frac{T_{\text{seq}}}{T_{\text{phase}} n(1-\gamma)} + \tilde{O}(h)}{\log n - \log \frac{2}{\gamma}}$$

cycles[7] are sufficient to process the entire problem. A complete cycle takes time $(T_{\text{phase}} + O(1)) \log n + \log n + o(\log n) \in (T_{\text{phase}} + O(1)) \log n$: There are $\log n$ phases, after every phase we need time for a request a split and a reply; a random permutation can be completed in time $\log n + o(\log n)$ with high probability using an appropriate routing algorithm [9, Theorem 3.27]. A bound for the execution time is:

$$T_{\text{par}} \in \frac{\frac{T_{\text{seq}}}{T_{\text{phase}} n(1-\gamma)} + \tilde{O}(h)}{\log n - \log \frac{2}{\gamma}} (T_{\text{phase}} + O(1)) \log n$$

This can be rewritten as

$$\frac{\log n}{\log n - \log \frac{2}{\gamma}} \frac{T_{\text{phase}} + O(1)}{T_{\text{phase}}(1-\gamma)} \left(\frac{T_{\text{seq}}}{n} + \tilde{O}(h)\right)$$

[7] In order to make the proof more concise, we use the $\tilde{O}$ notation quite freely, hoping that a suspicious reader is willing to take the effort to mentally fill in the details of "for sufficiently large n", "for some constant c", "with high probability", ...

We choose $\gamma = \frac{\epsilon}{2}$. The factor $\frac{\log n}{\log n - \log \frac{2}{\gamma}}$ gets arbitrarily close to 1 for sufficiently large n. For sufficiently large T_{phase} the factor $\frac{T_{\text{phase}}+O(1)}{T_{\text{phase}}(1-\gamma)}$ is smaller than $(1+\epsilon)$. Therefore,

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \tilde{O}(h) .$$

■

So, hypercube poll-and-shuffle is asymptotically optimal — it meets the lower bound of Equation (1).

3.2 Random permutations

Choosing a permutation uniformly at random is not as easy as it sounds. $\Omega(n \log n)$ random bits are necessary to define a random permutation. Although this can be done in time $O(\log n)$ if we assume an independent source of random bits in every PE, we still need to coordinate the information in such a way that every PE knows where to send its information.

One possibility works as follows: First, every PE chooses a PE number uniformly at random and sends its subproblem to this PE (idle PEs send empty subproblems). From the analysis of randomized routing algorithms (e.g. [9]) we know that the maximum number of subproblems destined for the same PE is in $\tilde{O}(\log n)$. Now, every PE sequentially permutes the locally present subproblems in time $\tilde{O}(\log n)$. We then enumerate the subproblems using a parallel prefix sum of the number of subproblems in each PE (time $O(\log n)$). Finally, every subproblem is sent to the PE defined by its number (time $\tilde{O}(\log n)$).

In practice, it might be better to replace this quite expensive procedure by some kind of pseudorandom permutations. For example, it is common practice in computational group theory [12] to precompute a small set of random permutations which have the property of generating the entire group (in this case the symmetric group S_n of all permutations over PE numbers). Then, a pseudorandom permutation is constructed by combining a small randomly selected sample of these precomputed permutations.

3.3 Synchronization of phases

The assumption of a global clock for defining phases is convenient for the analysis but not really necessary. We only need to make sure that requests are always exchanged between PEs in the same phase. This can be achieved by local synchronization. Figure 3 shows pseudocode for a simple poll-and-shuffle algorithm with explicit local synchronization. In practice, one would additionally make sure that PEs waiting for synchronization can work on their local subproblem (e.g. using multithreading).

3.4 Constant degree networks

The next key observation is that the hypercube dimensions are used one after the other. Using the quite general results from [8] on routing and [9, Section 3.3.3] on emulating *normal* hypercube algorithms we can conclude:

Corollary 1 *(Asynchronous) hypercube poll-and-shuffle can be adapted to butterflies, perfect-shuffle, and all networks which can efficiently emulate normal hypercube algorithms.*

```
WHILE not finished DO
    FOR i:=0 TO log n − 1 DO
        IF subproblem is not empty THEN
            work on subproblem for time T_phase or until exhausted
        send message "phase i finished" along dimension i
        wait for message "phase i finished" along dimension i
        IF subproblem is empty THEN
            send a request along dimension i
            reject incoming requests
            receive new subproblem or a reject message
        ELSE
            IF a request arrives THEN
                split subproblem
                send one part to the initiator of the request
    participate in randomly permuting the subproblems
```

Figure 3: Asynchronous hypercube poll-and-shuffle.

4 Meshes and fat trees

Our starting point is the idea to adapt the hypercube poll-and-shuffle algorithm to other networks by embedding subhypercubes. Section 4.1 explains this for meshes and Section 4.2 for fat trees. Finally, Section 4.3 lines out how simpler algorithms which do not need synchronizations between phases can also be used on these networks.

4.1 Meshes

Consider an r-dimensional mesh (n a power of 2, $d = 2^{\lceil n/r \rceil}$). A hypercube can be embedded in such a way that every j-dimensional subcube[8] is embedded into a submesh of diameter[9] $2^{j/r}$ (e.g. [7, Figure 6.11]). Using this embedding, a simple calculation shows that the communication necessary for $\log n$ phases of poll-and-shuffle can be performed in time $O(n^{1/r})$. Routing can also be performed in time $O(n^{1/r})$ [9].

The only complication we have to deal with is that the proof of Theorem 1 only works for phases of equal length. In fact, if we used a phase length proportional to the communication expense it would be conceivable that the short phases have good PE utilization and the long phases have low PE utilization, resulting in a poor overall efficiency. The solution is quite simple: We omit the last $r \log\log n$ phases of each cycle and set $T_{\text{phase}} := c\frac{n^{1/r}}{\log n}$, that is, a constant times the communication expense of the most expensive remaining phase. (The embedding of a $\log n - r\log\log n$-dimensional subcube has diameter $\left(2^{\log n - r\log\log n}\right)^{1/r} = \frac{n^{1/r}}{\log n}$.):

Theorem 2 *Let T_{par} denote the execution time of the hypercube poll-and-shuffle algorithm simulated on an r-dimensional mesh with the last $r\log\log n$ phases of*

[8] Given a PE, let the i-dimensional subcube it belongs to be defined as the 2^i PEs reachable over the i lowest numbered links.

[9] Strictly speaking, the figure is $2^{\lceil j/r \rceil}$, but for clearness we omit roundings which do not have an asymptotic effect.

each cycle omitted. For every $\epsilon > 0$ there is a choice of the constant c such that

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \tilde{O}\left(h\frac{n^{1/r}}{\log n}\right).$$

Proof: Analogous to the proof of Theorem 1. $r \log\log n$ takes the role of $\log \frac{2}{\gamma}$ and we have to substitute the appropriate execution times for polling and random permutations. ∎

4.2 Fat trees

We can use a similar approach as for meshes in order to derive a fairly good load balancing algorithm for fat trees [11]. We partition the network into sub fat trees of height $\sqrt{\log n}$ (with $2^{\sqrt{\log n}}$ PEs each). Setting T_{phase} to $c\sqrt{\log n}$, we can perform $\sqrt{\log n}$ poll-phases in time $O(\log n)$. Since routing is also possible in logarithmic time, we get:

Theorem 3 *Let T_{par} denote the execution time of the hypercube poll-and-shuffle algorithm simulated on a fat tree performing only $\sqrt{\log n}$ phases per cycle. For every $\epsilon > 0$ there is a choice of the constant c such that*

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \tilde{O}\left(h\sqrt{\log n}\right).$$

Proof: (Outline) Similar to proof of Theorem 2. This time we need a factor $O(\sqrt{\log n})$ more cycles than for the hypercube case. But a cycle takes no more time than in the hypercube case. ∎

4.3 Local random polling

Instead of taking a detour over a hypercube algorithm we can also use algorithms which exploit the full communication capacity of meshes and fat trees. Phases are sufficiently long that we can do communication anywhere within the subnetworks during every phase. We could even (locally) apply the deterministic load balancing schemes described in [6, 13, 5].

One attractive possibility is to replace the phases of each cycle by the random polling algorithm described in the introduction. Requests are sent to randomly selected PEs within a partition of diameter $\frac{n^{1/r}}{\log n}$ on the mesh and $\sqrt{\log n}$ on the fat tree. There is no synchronization between phases but busy PEs service requests in intervals no shorter than T_{phase}. Random permutations are initiated as often as before.

Theorem 4 *On meshes and fat trees, the algorithm defined by replacing the phases of a cycle by a local random polling scheme are asymptotically as efficient as poll-and-shuffle.*

Proof: (Outline) During local random polling in a time span of iT_{phase} at most 2^i subproblems can be derived from one subproblem; all other subproblems have unrelated positions. This observation can be used to derive a lemma similar to lemma 3. When we are not too late in a cycle, idle PEs will be well distributed over the partitions. And if there are γn idle PEs, every subproblem receives a request with probability at least $\gamma/2$ during a time span of T_{phase}. The remainder of the proof is analogous to the earlier results. ∎

5 Conclusion

The load balancing algorithms for tree shaped computations presented in this paper are a promising family of algorithms. For low diameter networks they achieve efficiencies arbitrarily close to 1 for a per PE load in $O(h)$ which is asymptotically optimal since the sequential component of the problem is of the same order. Therefore, the new algorithms are at the same time asymptotically as scalable as tree embedding techniques and have the same communication economy as earlier tree splitting based algorithms which require larger problem sizes for good efficiency.

For meshes, the algorithms have a better scalability (by a factor $\log n$) than the best previously known algorithms. In the important case of logarithmic depth trees ($h \in O(\log n)$), the required per PE load of $O(d)$ is asymptotically optimal. The new algorithms for fat trees are by a factor $\sqrt{\log n}$ better than the best previously known ones.

5.1 Future work

An interesting observation is that as the algorithms get simpler and more practical the analysis gets more involved. In fact, there is a number of additional attractive modifications whose effectiveness we have not yet been able to verify:

- Perhaps, synchronizations between phases can be completely omitted even for the hypercube case.
- Shuffling subproblems can be simplified by using permutations which are easy to determine and easy to route. For example, neighboring PEs can agree on a random bit and exchange their subproblems if it is a 1-bit. Iterating this for all dimensions might be a sufficiently random permutation. This random exchange can even be interleaved with the computation.
- The algorithms based on local random polling could mix local and global requests in such a way that the average cost for a request is not substantially changed. (For example allowing a global request with probability $O(1/\log n)$ on a mesh.) Random permutations are then only an additional provision for the case that something goes wrong. Simulation experiments indicate that this approach works well even if random permutations are completely disabled.

References

[1] O. I. El-Dessouki and W. H. Huen. Distributed enumeration on between computers. *IEEE Transactions on Computers*, C-29(9):818–825, September 1980.

[2] R. Feldmann. *Game Tree Search on Massively Parallel Systems*. PhD thesis, Universität Paderborn, August 1993.

[3] R. Finkel and U. Manber. DIB— A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, Apr. 1987.

[4] C. Kaklamanis and G. Persiano. Branch-and-bound and backtrack search on mesh-connected arrays of processors. In *ACM Symposium on Parallel Architectures and Algorithms*, 1992.

[5] G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1057–1072, 1994.

[6] V. Kumar and G. Y. Ananth. Scalable load balancing techniques for parallel computers. Technical Report TR 91-55, University of Minnesota, 1991.

[7] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[8] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17:157–205, 1994.

[9] T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, 1992.

[10] T. Leighton, M. Newman, A. G. Ranade, and E. Schwabe. Dynamic tree embeddings in butterflies and hypercubes. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 224–234, 1989.

[11] C. E. Leiserson. Fat trees: Universal networks for hardware efficient supercomputing. In *International Conference on Parallel Processing*, pages 393–402, 1985.

[12] T. Minkwitz. Personal communication. Department of Informatics, University of Karlsruhe, 1995.

[13] C. Powley, C. Ferguson, and R. E. Korf. Depth-fist heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.

[14] S. Rajasekaran. Randomized algorithms for packet routing on the mesh. In L. Kronsjö and D. Shumsheruddin, editors, *Advances in Parallel Algorithms*, pages 277–301. Blackwell, 1992.

[15] A. Ranade. Optimal speedup for backtrack search on a butterfly network. *Mathematical Systems Theory*, pages 85–101, 1994.

[16] V. N. Rao and V. Kumar. Parallel depth first search. Part I. *International Journal of Parallel Programming*, 16(6):470–499, 1987.

[17] V. N. Rao and V. Kumar. Parallel depth first search. Part II. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[18] P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.

[19] P. Sanders. A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures Algorithms and Networks*, pages 382–389, Kanazawa, Japan, 1994. IEEE.

[20] P. Sanders. Massively parallel search for transition-tables of polyautomata. In *Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, pages 99–108, Potsdam, 1994.

[21] P. Sanders. Randomized static load balancing for tree shaped computations. In *Workshop on Parallel Processing*, TR Universität Clausthal, page (to appear), Lessach, 1994.