Better Algorithms for Parallel Backtracking

Peter Sanders

University of Karlsruhe, 76128 Karlsruhe, Germany
Email: sanders@ira.uka.de

Abstract. Many algorithms in operations research and artificial intelligence are based on the backtracking principle, i.e., depth first search in implicitly defined trees. For parallelizing these algorithms, an efficient load balancing scheme is of central importance.
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 hypercubes, butterflies and related networks, and an improved scalability for meshes and hierarchical networks like fat trees.

Keywords: Analysis of randomized algorithms, depth first search, distributed memory, divide and conquer, load balancing, parallel backtracking.

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, dynamic load balancing strategies are needed 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 refer to Section 2.1): There are 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. Furthermore, a subproblem can be worked on sequentially. 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 a bound h on the height of the binary tree defined by splitting the root problem into atomic pieces.

One application domain for which this is a useful model 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 (even if the underlying tree is not binary.) Note that backtrack search is a central aspect of many AI and OR applications and of parallel functional and logical programming languages.

We now go on by describing *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. An overview of the remainder of the paper concludes this introduction.

1.1 Receiver induced tree splitting

The basic principle of *Receiver induced tree splitting* is that a PE only works on a single subproblem at a time and only activates 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
```

Fig. 1. Receiver induced tree splitting.

This approach has proved useful under a variety of circumstances [6, 14, 5, 22, 15, 25, 2, 11, 26, 7]. A major advantage of receiver induced tree splitting is that load balancing only takes place when necessary. If the sequential execution time T_{seq} is large, the average size of a transmitted subproblem is also fairly large (i.e. it represents a large execution time). Productive work done on a migrated subproblem therefore makes 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 bounded by $(1+\epsilon)\frac{T_{seq}}{n} +$ (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 PEs in order to achieve a desired efficiency. In this respect there are large differences between different load balancing strategies. We use the behavior of the lower order terms as a measure for the *scalability* of an algorithm – the smaller these terms the smaller is the problem size required for good efficiency.

In [14] it is shown that sending requests to neighboring PEs has a quite poor scalability except for the combination of 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 are quickly 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 [14, 15] a variety of other partner selection schemes is analyzed. There seems to be a dilemma between schemes based on local information on the one hand which may produce many vain requests to idle PEs, and global selection schemes on the other hand which incur additional message traffic and often suffer from contention at centralized schedulers. But *random polling*, i.e., selecting communication partners uniformly at random is identified as a promising scheme. Good speedups are reported for up to 1024 PEs. In [25] it is proved that random polling works in time $(1+\epsilon)\frac{T_{\text{seq}}}{n}+O(dh)$ 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

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

In [26] the asymptotic influence of message lengths and atomic grain sizes of subproblems is also investigated. Most other authors assume it to be constant. We also start with this assumption in order to free the analysis from rather uninteresting details. But in Section 6 we discuss consequences of a more detailed model. In [2] random polling is generalized to *fully strict multithreaded computations* allowing certain interactions between subproblems. An expected execution time in $O\left(\frac{T_{\text{seq}}}{n} + T_\infty\right)$ is proved for fully connected networks. (T_∞ denotes the sequential component of a problem.)

On SIMD computers, load balancing is done in separate load balancing phases initiated by some triggering condition [22, 27, 11]. 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 for up to 32K PEs.

1.2 Other related work

Another family of algorithms which are applicable to tree shaped computations are *dynamic tree embedding* algorithms [19, 23, 10]. Using our terminology, these algorithms are based on splitting the root problem 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 [19], it is shown in [23] how randomized dynamic tree embedding algorithms can be used to perform backtracking on butterflies and hypercubes in time $O\left(\frac{T_{\text{seq}}}{n} + h\right)$ 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 these algorithms is limited to a quite small constant value 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 [10], it is demonstrated how trees with $O(n)$ leaves can be deterministically embedded into an r-dimensional mesh in time $O\left(\sqrt[r]{nh}\right)$. It is not clear however, how useful the methods used there are 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 [3], but using it as an initialization for dynamic load balancers can yield a significant improvement. In [28], 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 on meshes achieves execution times in $(1+\epsilon)\frac{T_{\text{seq}}}{n} + O(n^{1/r})$ on the average. For $T_{\text{seq}} \in \Omega(dn)$ this is asymptotically optimal. 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. The emphasis is on algorithms which are not only interesting from a theoretical point of view but also simple and efficient in practice.

We start by introducing a simple yet realistic model of the machine and the application in Section 2 which is later generalized in Section 6. Then, Section 3 presents a hypercube based algorithm. The PEs perform receiver induced tree splitting; communication is done with neighboring PEs. By 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. Execution times are in $(1+\epsilon)\frac{T_{\text{seq}}}{n} + O(h)$ with high probability.

The algorithm is adapted to butterflies (and related constant degree networks), r-dimensional meshes and hierarchical networks like fat trees in Section 4. Execution times are in $(1+\epsilon)\frac{T_{\text{seq}}}{n} + O(h)$, $(1+\epsilon)\frac{T_{\text{seq}}}{n} + O(h)\frac{n^{1/r}}{\log n}$ and $(1+\epsilon)\frac{T_{\text{seq}}}{n} + O(h)\sqrt{\log n}$ respectively with high probability.

Section 5 shows how the expensive random permutations can be avoided as long as PE utilization is good. A sufficiently accurate estimate of the global load can be maintained with very little communication: A PE falling idle only informs a supervising PE with probability $O(1/n)$. Finally, Section 7 summarizes the results and compares them to known lower bounds.

2 Notation

2.1 Machine and Application Model

We consider a message passing MIMD computer with n PEs numbered 0 through $n-1$. The PEs operate asynchronously but for simplicity we assume the existence of a global time. 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 $T_{\text{seq}} \in \Omega(n)$. Any subproblem can be worked on sequentially such that after working on a subproblem of size T for time t we get a subproblem of size $T-t$.

The *splitting function* is able to split a subproblem S of size T into two subproblems S_1 and S_2 of size T_1 and T_2 in unit time. For the analysis we assume that $T_1 + T_2 = T$ regardless when and where the subproblems are processed. The *generation* $\text{gen}(S)$ of a subproblem is inductively defined by $\text{gen}(\text{root problem}) = 0$ and $\text{gen}(S_1) = \text{gen}(S_2) = \text{gen}(S) + 1$. A subproblem S with $\text{gen}(S) \geq h$ must be guaranteed to be reduced to a constant atomic size T_{atomic} or smaller. An immediate consequence of the above definitions is that

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

($\log n$ means $\log_2 n$ throughout this paper.) Splitting an atomic subproblem yields the same subproblem plus an empty subproblem. All other properties of the splitting function can be chosen by an adversary. 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 [8].

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

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

i.e., the probability that X exceeds the bound f by more than a constant factor is polynomially small. In this paper, the variable used to express high probability is always n – the number of PEs.

A keystone of many probabilistic proofs are the following *Chernoff bounds* which give quite tight bounds on the probability that the sum of 0/1-random variables deviates from the expected value by some factor.

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 [18]:*

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

$$\mathbf{P}\left[X \geq \alpha np\right] \leq e^{\left(1-\frac{1}{\alpha}-\ln\alpha\right)\alpha np} \text{ for } \alpha > 1 \tag{3}$$

3 Hypercube poll-and-shuffle

3.1 The basic algorithm

Consider a $\log n$-dimensional hypercube network. Every PE performs receiver induced tree splitting. Computation time is partitioned into *phases* of constant length T_{phase}. Idle PEs are only allowed to send requests after a phase. After phase number i, requests go to the neighbor along dimension i (i.e. PE k sends a request to PE k `xor` 2^i). When we have reached phase $\log n$, we are out of fresh dimensions for communication. Therefore, we randomly permute the subproblems (we say that idle PEs contain empty subproblems) and start a new *cycle* by resetting the phase counter to 0. Figure 2 shows this partitioning of the time line for $n = 2^4$ and 2 cycles.

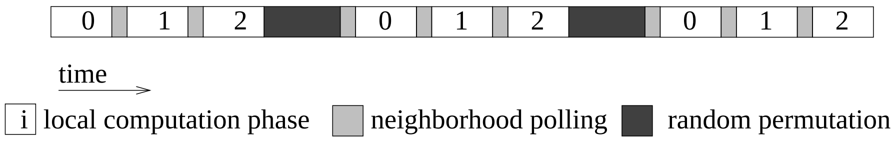

Fig. 2. Two cycles of hypercube poll-and-shuffle for $n = 2^4$.

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 i 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. Due to space constraints, we can only outline this and all subsequent proofs and have to refer to the full paper for details. During the current cycle, S can only have interacted with $2^i < \frac{\gamma}{2}n$ PEs. Sufficiently many of the remaining PEs will issue a request and with probability $\geq \gamma/2$ one of these PEs will be $S's$ neighbor along dimension i. This is shown by partitioning the set of all possible permutations into equivalence classes of equal size – one class for each pairing of S with another subproblem. ∎

Building on this we can bound the number of phases necessary to reduce all subproblems to atomic size.

Lemma 4. *For any constant $\gamma > 0$, $\tilde{O}(h)$ phases with at least γn idle PEs and a phase number less than $\log n - \log \frac{2}{\gamma}$ are sufficient such that every existing subproblem S has $gen(S) \geq h$.*

Proof. By viewing the event that a subproblem is hit by a request as a Bernoulli trial with success probability at least $\gamma/2$, we can use Chernoff bounds to show that the probability that a particular subproblem receives less than h requests after $O(h)$ phases is polynomially small. Since taking the minimum over all subproblems only increases this probability by a factor at most n, the same is true for all subproblems together. ∎

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. Based on this observation we can bound the parallel execution time:

Theorem 5. *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*

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

Hypercube poll-and-shuffle is asymptotically optimal because its execution time meets the lower bound of Equation (1). The algorithm is better than any known tree embedding scheme because efficiencies arbitrarily close to 1 are possible even if $T_{\text{atomic}} \ll 1$, and it is better than previously known tree splitting algorithms because the lower order term $\tilde{O}(h)$ is smaller than the $\tilde{O}(h \log n)$ term for random polling.

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. From the analysis of randomized routing algorithms (e.g. [18]) 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 [21] 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.

4 Other Networks

Our starting point is the idea to adapt the hypercube poll-and-shuffle algorithm to other networks by embedding a hypercube into the real network in such a way that poll-and-shuffle works efficiently.

This is quite easy for *hypercubic networks* in the sense of [18] (e.g. butterflies, cube-connected-cycles, perfect shuffle, DeBruijn) because poll-and-shuffle uses the hypercube dimensions one after the other. Using the quite general results from [17] on routing and [18, Section 3.3.3] on emulating *normal* hypercube algorithms we can conclude:

Theorem 6. *Hypercube poll-and-shuffle can be adapted to constant degree hypercubic networks in such a way that*

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

Proof. Neighborhood polling and permutation routing takes a constant factor longer than on the hypercube. But this can be compensated by making T_{phase} correspondingly larger. Everything else is completely analogous to the proof of Theorem 5. ∎

By introducing some minor modifications into the poll-and-shuffle concept we can also derive good algorithms for meshes and hierarchical networks like fat trees.

4.1 Meshes

Consider an r-dimensional mesh (n a power of 2, $d = r2^{\lceil \log n/r \rceil} - r$ and r constant). A hypercube can be embedded in such a way that every j-dimensional subcube is embedded into a submesh of diameter $r2^{\lceil j/r \rceil} - r$ (e.g. [15, 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 $O(n^{1/r})$ [18].

The only complication we have to deal with is that the proof of Theorem 5 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 $r2^{\left\lceil \frac{\log n - r \log\log n}{r} \right\rceil} - r \in O\left(\frac{n^{1/r}}{\log n}\right)$):

Theorem 7. *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 each cycle omitted. For every $\epsilon > 0$ there is a choice of the constant c such that*

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

Proof. Analogous to the proof of Theorem 5. $r\log\log n$ takes the role of $\log\frac{2}{\gamma}$ and we have to substitute the appropriate execution times for polling and random permutations. ∎

Using the results from [1] on emulating mesh algorithms on the mesh of trees network we can conclude that same performance is possible on meshes of trees.

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 [20]. 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 8. *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*

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

Proof. Similar to proof of Theorem 7. 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. ■

The same pattern can be applied to any network: If possible, embed subcubes into subnetworks in such a way that intra-subnetwork routing is faster than global routing. If the saving is sufficiently large to make up for the random permutations, we get an algorithm superior to random polling.

5 Adaptively initiating permutations

The periodically invoked random permutation is a quite expensive operation during which no work on the subproblems is possible. In addition, there is actually no reason to redistribute subproblems as long as the PE utilization is good. A simple approach to avoiding unnecessary work is to determine the average number of idle PEs during a cycle and to trigger a random permutation only if PE utilization is low. For example, we can trigger when the average number of idle PEs raises above γn for some appropriate constant γ. More sophisticated triggering conditions, which take the current cost for load balancing and the past development of PE utilization into account, are described in [22, 11].

A simple way to implement this idea is to count the number of idle phases during a cycle on each PE and to determine the average number of idle PEs by globally adding all these values after each cycle. But although a global add may be considerably cheaper than migrating all subproblems, we would still have an expensive global operation which is invoked periodically even though PE utilization may be fairly stable.

Counting can be made more adaptive by letting idle PEs notify a monitoring PE (say PE 0) about their idleness. (Refer to [16] for another approach to adaptively determining global load changes which is used in the context of balancing independent work packets.) However, if *every* idle PE sent a message, the monitoring PE would become a terrible bottleneck. Here, randomization comes to the rescue again: Whenever a PE becomes idle or an idle PE enters a new phase, it notifies PE 0 with probability[1] $\frac{c}{n}$ (for some constant c still to be determined). If PE 0 receives m notifications during k phases, the average number of idle PEs can be estimated to be $\frac{mn}{ck}$. (The effect of message latency is compensated by only using notifications from phases sufficiently far back such that all their messages must have arrived with high probability.)

We first show that the arrival rate of notifications is sufficiently low in order to avoid contention at PE 0.

Lemma 9. *The number of notifications sent during* $\log n$ *phases is in* $\tilde{O}(\log n)$.

[1] For fat trees, the probability needs to be set to $c\sqrt{\log n}/n$ in order to achieve the right balance of messages. For meshes we can choose some higher probability in $O(n^{1/r})$ without creating a hot spot.

Proof. Using Chernoff bounds by viewing the decisions whether a notification is sent as independent Bernoulli trials with success probability $\frac{c}{n}$. ∎

Since $\log n \in O(d(n))$ for all the networks we are considering, the contention at PE 0 will not asymptotically change the message latency.

We go on by showing that a significant underestimation of the fraction of idle PEs is improbable.

Lemma 10. *Let $\bar{\gamma}$ denote the average fraction of idle PEs over $\alpha \log n$ phases ($\alpha > 0$). Let $\hat{\gamma}$ denote the estimate of $\bar{\gamma}$ based on the notifications sent by idle PEs. For every $\epsilon > 0$ there is a choice of the constant c such that $\hat{\gamma} \geq \bar{\gamma} - \epsilon$ with probability at least $1 - \frac{1}{n}$.*

Proof. Similar to the proof of Lemma 9. ∎

Finally, the effect of overestimations of the number of idle PEs can be compensated by never initiating random permutations after less than $\log n$ phases. We now have all the required components for analyzing a variant of poll-and-shuffle which bases triggering decisions on the estimate of the fraction of idle PEs.

Theorem 11. *Let T_{par} denote the execution time of the poll-and-shuffle algorithm with probabilistic triggering of shuffling. For every $\epsilon > 0$ there is a choice of the phase length T_{phase}, the constant c, and a triggering policy such that*

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

Proof. Due to message latencies there is a fraction of the current cycle about which PE 0 is not informed. T_{phase} must be sufficiently large such that this fraction is small. c must be sufficiently large such that the number of phases needed to accumulate sufficiently many notifications is small compared to $\log n$. Lemma 9 shows that contention of notification messages at PE 0 is not an issue if $c \ll T_{\text{phase}}$. We can use Lemma 10 in order to show that with high probability only a small number of cycles with low PE utilization will go undetected. A possible strategy for triggering permutations is to trigger whenever the estimate for the number of idle PEs exceeds $\frac{\epsilon}{2}n$. In order to avoid an accumulation of estimation errors, outdated information has to be discarded from time to time, for example by considering only the last $\alpha \log n$ phases for some appropriate α. ∎

6 Generalizing the problem model

In Section 2 we made a lot of simplifying assumptions about the machine and problem model in order to concentrate on load balancing, irregularity and locality. Now we want to outline how relaxing some of these assumptions affects the performance of our algorithms. For example, the global time can be replaced by local synchronization between the phases. (for details refer to the full paper.)

6.1 Message lengths

For many backtracking applications our assumption of constant message lengths is quite realistic: Requests, rejections and other control information only need a constant number of machine words. And, if we initially broadcast the root problem to all PEs, subproblems can often be represented by sequences of $O(h)$ bits indicating how to derive the subproblem from the root problem by subsequent split operations (e.g. [12]). Under the usual assumption that $O(\log n)$ bits fit into a machine word our assumption is strictly justified for the frequent case [15] $h \in O(\log T_{\text{seq}})$ and T_{seq} polynomial in n (Larger problems are easy to load balance anyway).

Still, in other cases it is better to treat the message length as an additional variable of the problem.[2] Let us assume that l is an upper bound on the length of the representation of a subproblem depending on the root problem only. If we stick to packet routing, we can easily adapt our analysis by charging $O(dl)$ instead of $O(d)$ time for transmitting a message. But we might be able to do better by using cut-through routing [15] (i.e. chopping messages into constant size pieces) which makes it possible to send a subproblem in time $O(l + d)$ as long as there is no network contention. For networks with a high bisection width of $\Omega(n)$ (e.g. hypercubes, fat trees, multi-stage butterflies) and long messages ($l \in \Omega(d)$) poll-and-shuffle now has no advantage over random polling since the transmission time is in $\tilde{O}(l)$ regardless of the distance between communication partners. (Refer to [26] for a discussion why the network traffic is sufficiently uniform to justify this conclusion.)

However, for networks with limited bisection width like meshes or butterflies, cut-through routing has no asymptotic advantage over packet routing when the network is highly loaded. And indeed, any efficient receiver induced tree splitting algorithm must sometimes handle $\Omega(nh)$ load transfers which is a constant fraction of the total number of messages for random polling and poll-and-shuffle. (For details refer to the full paper.)

This observation has a consequence of some practical importance. One might be tempted to regard poll-and-shuffle as only of theoretical interest since many contemporary machines with mesh architecture have a low latency hardware router such that the physical *distance* between communication partners is only of secondary importance for the case of low network load. But poll-and-shuffle can nevertheless yield a significant improvement on these machines because the usable *bandwidth* per message is higher than for random polling: For example, consider the poll-and-shuffle algorithm for r-dimensional meshes from Section 4.1. During every phase of a cycle at least $\Omega(n^{\frac{r-1}{r}} \log n)$ PEs can communicate in parallel via cut-through routing without any contention while for random polling the figure is only $O(n^{\frac{r-1}{r}})$.

[2] Note that the tree embedding schemes from Section 1.2 can only achieve an efficiency in $O\left(\frac{1}{l}\right)$ for nonconstant message lengths.

6.2 The splitting function

In Section 2 we assumed that 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 such that that $T_1 + T_2 = T$ regardless when and where the subproblems are processed.

Relaxing the assumption that the splitting function takes unit time does not yield anything new as long as splitting a subproblem does not take longer than communicating it. For example, this is always the case if $T_{\mathrm{split}} \in O(l)$, i.e., splitting is linear in the length of a the representation of a subproblem. Similarly, the effect that splitting often performs productive work on the ancestor problem in order to find an acceptable place for splitting is quite uninteresting.

However, T_1 and T_2 can be quite dependent on the order of subproblem evaluation in general. Usually, the reason is that the evaluation of one subproblem yields information which can be used to prune (or reduce in size) other subproblems. The resulting *speedup anomalies* are a complex issue by themselves. For some classes of problems, large superlinear speedups (compared to sequential depth first search) can be achieved [4, 24], others show wildly varying speedups over several orders of magnitude [29] and search algorithms with strong pruning heuristics like game tree search are very hard to parallelize efficiently [5, 7]. We did not incorporate these effects into our model because they are quite application dependent and because we wanted to concentrate on the load balancing aspect of parallel search.

Also, there are many applications for which our model is quite accurate: All but the last iteration of iterative deepening search algorithms like IDA* [13] are independent of the evaluation order, and in the last iteration it is a quite good approach to work on several randomly selected subproblems until the first solution is found [9, 4, 24].

A practically useful application of branch-and-bound is to verify that a known (heuristic) solution is within a certain percentage of the optimum. In this case the search tree does not depend on the execution order. Similarly, in many applications optimal solutions are usually found quickly but verifying their optimality takes very long such that our assumption holds for the main part of the search.

6.3 Problem granularity

If we consider T_{atomic} to be an additional problem variable, we trivially get an additional lower bound $T_{\mathrm{par}} \in \Omega(T_{\mathrm{atomic}})$ and it is quite simple to change the analysis of our algorithms to show that there is simply an additional $O(T_{\mathrm{atomic}})$ term in the upper bound for the parallel execution time if $T_{\mathrm{seq}} \in \Omega(nT_{\mathrm{atomic}})$. (For random polling this has been done in [26].)

7 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 $O(h)$. This is asymptotically optimal since the sequential component of the problem instances 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 $O(\log n)$) than the best previously known algorithms. In the important case of logarithmic depth trees ($h \in O(\log n)$), the PE load of $O(d)$ required for constant efficiency is asymptotically optimal. The new algorithms for fat trees are by a factor $\sqrt{\log n}$ better than the best previously known ones.

In fact, the algorithms turn out to be optimal in so many cases that we were tempted to title this paper "Towards asymptotically optimal algorithms for parallel backtracking." However, the discussion in Section 6 shows that optimality is so dependent on the underlying model that such a title would be misleading.

8 Acknowledgements

I would like to thank H. Rust, A. C. Achilles, H. Fernau, T. Minkwitz and T. Worsch for many constructive discussions which helped to shape the algorithms and their analysis.

References

1. A.-C. Achilles. Optimal emulation of meshes on meshes-of-trees. In *EURO-PAR International Conference on Parallel Processing*, 1995.
2. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, 1994.
3. O. I. El-Dessouki and W. H. Huen. Distributed enumeration on between computers. *IEEE Transactions on Computers*, C-29(9):818–825, September 1980.
4. W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.* Dissertation, TU München, 1992.
5. R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 94–103, 1994.
6. R. Finkel and U. Manber. DIB— A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, Apr. 1987.
7. H. Hopp and P. Sanders. Parallel game tree search on SIMD machines. In *Workshop on Algorithms for Irregularly Structured Problems*, LNCS, Lyon, 1995. Springer.
8. D. Ierardi. 2d-bubblesorting in average time $O(\sqrt{N \lg N})$. In *ACM Symposium on Parallel Architectures and Algorithms*, 1994.
9. V. K. Janakiram, E. F. Gehringer, D. P. Agrawal, and R. Mehotra. A randomized parallel branch-and-bound algorithm. *International Journal of Parallel Programming*, 17(3):277–301, 1988.

10. 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.
11. G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1057–1072, 1994.
12. J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
13. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
14. V. Kumar and G. Y. Ananth. Scalable load balancing techniques for parallel computers. Technical Report TR 91-55, University of Minnesota, 1991.
15. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.
16. T. Lauer. *Adaptive dynamische Lastbalancierung.* PhD thesis, Max Planck Institute for Computer Science Saarbrücken, 1995.
17. 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.
18. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, 1992.
19. 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.
20. C. E. Leiserson. Fat trees: Universal networks for hardware efficient supercomputing. In *International Conference on Parallel Processing*, pages 393–402, 1985.
21. T. Minkwitz. Personal communication. Department of Informatics, University of Karlsruhe, 1995.
22. C. Powley, C. Ferguson, and R. E. Korf. Depth-fist heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.
23. A. Ranade. Optimal speedup for backtrack search on a butterfly network. *Mathematical Systems Theory*, pages 85–101, 1994.
24. V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.
25. P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.
26. 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.
27. 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.
28. P. Sanders. Randomized static load balancing for tree shaped computations. In *Workshop on Parallel Processing*, TR Universität Clausthal, Lessach, Austria, 1994.
29. A. P. Sprague. Wild anomalies in parallel branch and bound. Technical Report CIS-TR-91-04, CIS UAB, Birmingham, AL 35294, 1991.