Asynchronous Random Polling Dynamic Load Balancing

Peter Sanders

Max-Planck-Institut für Informatik,
Im Stadtwald, 66123 Saarbrücken, Germany.
E-mail: `sanders@mpi-sb.mpg.de`, WWW: `http://www.mpi-sb.mpg.de/~sanders`

Abstract. Many applications in parallel processing have to traverse large, implicitly defined trees with irregular shape. The receiver initiated load balancing algorithm *random polling* has long been known to be very efficient for these problems in practice. For any $\epsilon > 0$, we prove that its parallel execution time is at most $(1+\epsilon)T_{\mathrm{seq}}/P + \mathcal{O}(T_{\mathrm{atomic}} + h(\frac{1}{\epsilon} + T_{\mathrm{rout}} + T_{\mathrm{split}}))$ with high probability, where T_{rout}, T_{split} and T_{atomic} bound the time for sending a message, splitting a subproblem and finishing a small unsplittable subproblem respectively. The *maximum splitting depth* h is related to the depth of the computation tree. Previous work did not prove efficiency close to one and used less accurate models. In particular, our machine model allows asynchronous communication with nonconstant message delays and does not assume that communication takes place in rounds. This model is compatible with the LogP model.

1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking principle for traversing large irregularly shaped trees that are only defined implicitly by the computation [3, 4, 6, 9, 12–14, 19, 17, 21, 35]. Similar problems also play a role in parallel programming languages [1, 16]. Even for loop scheduling and some numerical problems [7, 24] like adaptive numerical integration [25] it can be useful to view the computations as an implicitly defined tree (refer to [34] for a more detailed discussion of examples).

For parallelizing tree shaped computations, a load balancing scheme is needed that is able to evenly distribute the parts of an irregularly shaped tree over the processors. It should work with minimal interprocessor communication and without knowledge of the shape of the tree. Load balancers often suffer from the dilemma that subtrees which are not subdivided turn out to be too large for proper load balancing whereas excessive communication is necessary if the tree is shredded into too many pieces.

We consider *random polling* dynamic load balancing [19] (also known as *randomized work stealing* [5, 10, 2, 11]), a simple algorithm that avoids both problems: Every processing element (PE) handles at most one piece of work (which may represent a part of a backtracking tree) at any point in time. If a PE runs out of work, it sends requests to randomly chosen PEs until a busy one is found which splits its piece of work and transmits one to the requestor.

We continue this introduction by explaining the machine model in Section 1.1 and the problem model *tree shaped computations* in Section 1.2. Section 1.3 reviews related work and summarizes the new contributions. The main body of the paper begins with a more detailed description of random polling in Section 2. In Section 3 we then give expected time bounds and show in Section 4 that they also hold with high probability (using additional measures). Finally, Section 5 summarizes the paper and discusses some possible future research.

1.1 Machine Model

We basically adopt the LogP model [8] due to its simplicity and genericity. There are P PEs numbered 0 through $P-1$. We assume a word length of $\Omega(\log P)$ bits.[1] Arithmetics on numbers of word length – including random number generation – is assumed to require constant time. All messages delivered to a PE are first put into a single FIFO *message queue*. In the full LogP model, three parameters for "latency" L, "overhead" o and "gap" g contribute to the cost of message transfer. We make the more conservative assumption that sending and receiving messages always costs $T_{\text{rout}} := L + o + g$ units of time. So the analysis also applies to the widespread messaging protocols that block until a message has been copied into the message queue of the recipient.

1.2 Tree Shaped Computations

We now abstract from the applications mentioned in the introduction by introducing *tree shaped computations* which expose just enough of their common properties in order to parallelize them efficiently. All the work to be done is initially subsumed in a single *root problem* I_{root}. I_{root} is initially located on PE 0. All other PEs start idle, i.e., they only have an *empty problem* $I_{\emptyset}$.

What makes parallelization attractive, is the property that problem instances can be subdivided into *subproblems* that can be solved independently by different PEs. For example, a subproblem could be "search this subtree by backtracking" or "integrate function f over that subinterval". We model this property by a *splitting operation* $\text{split}(I)$ that splits a given (sub)problem I into two new subproblems subsuming the parent problem. Let T_{split} denote a bound on the time required for the split operation. For example, in backtracking applications a subproblem is usually represented by a stack and splitting can be implemented by copying the stack and manipulating the copies in such a way that they represent disjoint search spaces covering the original search space [26].

The operation $\text{work}(I, t)$ transforms a given subproblem I by performing sequential work on it for t time units. The operation also returns when the subproblem is exhausted.

[1] Throughout this paper $\log x$ stands for $\log_2 x$.

What makes parallelization difficult, is that the *size*, i.e., the execution time $T(I) := \min\{t : \mathrm{work}(I,t) = I_\emptyset\}$, of a subproblem cannot be predicted. In addition, the splitting operation will rarely produce subproblems of equal size. For the analysis we assume however that $\forall I : \mathrm{split}(I) = (I_1, I_2) \implies T(I) = T(I_1)+T(I_2)$ regardless when and where I_1 and I_2 are worked on. For a discussion when this assumption is strictly warranted and when it is a good approximation, refer to Section 5 and to [32, 34].

Next we quantify some guaranteed "progress" made by splitting subproblems. Every subproblem I belongs to a *generation* $\mathrm{gen}(I)$ recursively defined by $\mathrm{gen}(I_{\mathrm{root}}) := 0$ and $\mathrm{split}(I) = (I_1, I_2) \implies \mathrm{gen}(I_1) = \mathrm{gen}(I_2) = \mathrm{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* h which guarantees that the size of subproblems with $\mathrm{gen}(I) \geq h$ cannot exceed some *atomic grain size* T_{atomic}. For example, a backtracking search tree of depth d and maximum branching factor b is easy to split in such a way that $h \leq d \lceil \log b \rceil$. We want to exclude problem instances with very little parallelism and therefore assume $h \geq \log P$. Otherwise, we might quickly end up with less than P atomic pieces of work that cannot be split any more. Since h is the only factor that constrains the shape of the emerging "subproblem splitting tree", it can be viewed as a measure for the irregularity of the problem instance. (Obviously, very regular instances with large h are possible. But in applications where this is frequently the case, one should perhaps look for a splitting function exploiting these regularities to decrease h.)

Finally, subproblems can be moved to other PEs by sending a single message. If problem descriptions are long, the parameters of the LogP model must be adapted to reflect the cost of such a long message. The resulting time bounds will be conservative since many messages are much shorter.

The task of the algorithm analysis is now to bound the parallel execution time T_{par} required to solve a problem instance of size $T_{\mathrm{seq}} := T(I_{\mathrm{root}})$ given the problem parameters h, T_{split} and T_{atomic} and the machine parameters P and T_{rout}. The bound is represented in the form

$$T_{\mathrm{par}} \leq (1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + T_{\mathrm{rest}}(P, T_{\mathrm{rout}}, \epsilon, h, \dots) \tag{1}$$

where $\epsilon > 0$ represents some small value we are free to choose. So, for situations with $T_{\mathrm{rest}} \ll T_{\mathrm{seq}}/P$ we have a highly efficient parallel execution.

1.3 Related Work and New Results

There is a quite large body of related research so that we can only give a rough outline. Many algorithms use a simpler approach regarding tree decomposition by requiring all "splits" to occur before calls to "work" (in our terminology). However, this is only efficient for some applications since in the worst case a huge number of subproblems may have to be generated or communicated (e.g. [18, 7, 27]).

Random polling belongs to a family of *receiver initiated* load balancing algorithms which have the advantage to split subproblems only on demand by idle PEs. This adaptive approach has been used successfully for a variety of purposes such as parallel functional [1] and logic programming [16] or game tree search [12]. Randomized partner selection goes at least back to [13]. The partner selection strategy turns out to be crucial. The apparently economic option to poll neighbors in the interconnection network can be extremely inefficient since it leads to a buildup of "clusters" of busy PEs shielding large subproblems from being split [26]. Polling PEs in a "global round robin" fashion [18] avoids this because no large subproblems can "hide". Execution times $T_{\text{par}} \in \mathcal{O}(\frac{T_{\text{seq}}}{P} + hT_{\text{count}})$ can be achieved where T_{count} is the time for incrementing a global counter. However, even sophisticated distributed counting algorithms have $T_{\text{count}} \in \Omega(T_{\text{rout}} \log P / \log\log P)$ [36]. It was long known that random polling performs better than global round robin in practice although the first analytical treatments could only prove an asymptotically weaker bound $\mathbf{E}T_{\text{par}} \in \mathcal{O}(\frac{T_{\text{seq}}}{P} + hT_{\text{rout}} \log P)$ [18]. Tree shaped computations are a generalization of the α-splitting model used in [18]. The gap between analysis and practical experience was closed in [28, 29] by showing that $T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + \mathcal{O}(hT_{\text{rout}})$ with high probability using synchronous random polling.

Slightly later, random polling (also called randomized work stealing) was found to be very efficient for scheduling multithreaded computations [5]. For many underlying applications, the two models can be translated into each other. The critical path length T_∞ in multithreaded computations then becomes $hT_{\text{split}} + T_{\text{atomic}}$ for tree shaped computations. Multithreading can model predictable dependencies between subproblems while tree shaped computations allow for different splitting strategies which may significantly decrease h [26]. Multi-threaded computations are most easy to use with programming language support, while tree shaped computations are directly useful for a portable and reusable library [31, 34]. In the following, we concentrate on tree shaped computations. Adapting these results to multithreading or some more general model encompassing both approaches is an interesting area for future work however.

All the analytical results above (including [28, 29]) make simplifying assumptions that are unrealistic for large systems, difficult to implement or detrimental to practical performance. The most common assumption is that communication takes place in synchronized communication rounds. This is undesirable since idle PEs have to wait for the next communication round and the network capacity is left unexploited most of the time. In fact, actual implementations are usually asynchronous. Arora et al. allow small speed fluctuations ($2C$–$3C$ instructions per round) but even that may be diffucult to attain since the number of clock cycles needed per instruction can be highly data dependent on modern processors (e.g., cache faults for large inputs). They also assume that polling and splitting take constant time ($T_{\text{rout}} + T_{\text{split}} \in \mathcal{O}(1)$ in our terminology). This is a viable assumption for moderate size shared memory machines and the thread stack of a multithreaded language. But we want an algorithm that scales to large distributed memory machines and allows more sophisticated application specific splitting functions. Using an even simpler stochastic model, Mitzenmacher was able to analyze many variants of work stealing [23].

Unfortunately, we cannot fully transfer an analysis for the above "round models" to a realistic asynchronous model since subproblems that are "in transit" cannot be split and long request queues can build up around PEs that have "difficult to split" subproblems. In Section 3 we solve these analytical problems and show that $\mathbf{E}T_{\text{par}} \leq (1+\epsilon)T_{\text{seq}}/P + \mathcal{O}(T_{\text{atomic}} + h\,(1/\epsilon + T_{\text{rout}} + T_{\text{split}}))$. In Section 4 it turns out that this bound also holds with high probability although for some values of h it may be necessary to actively trim long queues.

The time bound is not only tight for random polling but in [32] we also show a number of lower bounds which come very close: There are tree shaped computations for which a splitting overhead of $\Omega(hT_{\text{split}})$ is unavoidable so that we get an $\Omega(T_{\text{seq}}/P + T_{\text{atomic}} + hT_{\text{split}})$ lower bound. Furthermore, any receiver initiated load balancing algorithm not only needs $\Omega(h)$ communications on the critical path but also $\Omega(hP)$ full size messages overall so that the network bandwidth is fully utilized. Wu and Kung [37] show that a similar bound holds for all deterministic algorithms. Random polling *can* be slightly improved on certain networks by carefully increasing the average locality of communication [30]. At least up to constant factors, similar results can be achieved by dynamic tree embedding algorithms (e.g. [15]).

2 The Algorithm

Figure 1 gives pseudo-code for the basic random polling algorithm. PE 0 is initialized with the root problem as specified in the model. PEs in possession of nonempty subproblems do sequential work on them but poll the network for incoming messages at least every Δt time units and at most every $\alpha\Delta t$ time units for any constant $\alpha < 1$.[2] When a request is received, the local subproblem is split and one of the new subproblem is sent to the requestor. Idle PEs send requests to randomly determined PEs and wait for a reply until they receive a nonempty subproblem. Requests received in the meantime are answered with an empty subproblem. Note that an empty subproblem can be coded by a short message equivalent to a rejection of the request.

Concurrently, a distributed termination detection protocol is run that recognizes when all PEs have run out of work. We have adapted the *four counter* method [22] for this purpose. Each PE counts the number of sent and received messages that contain nonempty subproblems. When the global sum over these two counts yields identical results over two global addition rounds, there cannot be any work left (not even in transit). Instead of the ring based summing scheme proposed in [22], we use a tree based asynchronous global reduction operation. This is a simple and portable way to bound the termination detection delay by $\mathcal{O}(T_{\text{rout}} \log P)$.

[2] If the machine supports it, explicit polling can be replaced by more efficient and more elegant interrupt mechanisms which (almost) only cost time when requests arrive.

```
var I, I' : Subproblem
I := if i_PE = 0 then I_root else I_∅
while no global termination yet do
   if T(I) = 0 then
      send a request to a random PE
      repeat
         receive any message M (blockingly)
         reply requests from PE j with I_∅
      until M is a reply to my request
      unpack I from M
   else I := work(I, Δt)
      if incoming request from PE j then (I, I') := split(I); send I' to PE j
```

Fig. 1. Basic algorithm for asynchronous random polling.

3 Expected Time Bounds

This Section is devoted to proving the following bound on the expected parallel execution time of asynchronous random polling dynamic load balancing:

Theorem 1. $\mathbf{E}T_{\mathrm{par}} \leq (1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + \mathcal{O}\left(T_{\mathrm{atomic}} + h\left(\frac{1}{\epsilon} + T_{\mathrm{rout}} + T_{\mathrm{split}}\right)\right)$ *for an appropriate choice of* Δt.

The basic idea for the proof is to partition the execution time of each individual PE into intervals of productive work on subproblems and intervals devoted to load balancing. We first tackle the more difficult part and show that a certain overall effort on load balancing suffices to split all remaining subproblems at least h times. By definition of h this implies that they are smaller than T_{atomic}. As a preparation, we assign a technical meaning to the terms "ancestor", "arrive" and "reach":

Definition 1. *The* ancestor *of a subproblem* I *at time* t *is the uniquely defined subproblem from which* I *was derived by applying the operations* "work" *and* "split"*. A load request* arrives *at the point of time* t *when it is put into the message queue of a PE. A load request* reaches *a subproblem* I *at time* t *if it arrives at some PE at time* t *and (later) leads to a splitting of* I*.*

We start the analysis by bounding the expense associated with sending and answering individual requests:

Lemma 1.
1. *The total amount of active CPU work expended for processing a request is bounded by* $T_{\mathrm{split}} + \mathcal{O}(T_{\mathrm{rout}})$.
2. *If any requests have arrived at a PE, at least one of the requests is answered every* $\Delta t + T_{\mathrm{split}} + \mathcal{O}(T_{\mathrm{rout}})$ *time units.*
3. *The expected elapsed time between the arrival of a message and sending the corresponding reply is in* $\mathcal{O}(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}})$.

Proof. **1:** A request triggers at most one split. The total expense for sending and receiving is in $\mathcal{O}(T_{\mathrm{rout}})$. **2:** An additional time of Δt for sequential work can elapse until the message queue is checked the next time. **3:** Some queues might be long so that some request are delayed for a quite long time. However, there are at most P active requests at any point in time. A request arriving at a random PE will therefore encounter an expected queue length bounded by $\sum_{i<P}$ "queue length at PE i"$/P \leq 1$. ■

When a subproblem is split by one or more subsequent load request, there is a *dead time* interval during which it cannot be reached by any other request.

Lemma 2. *All dead times can be covered by associating a dead time* $T_{\mathrm{dead}} = \Delta t + T_{\mathrm{split}} + \mathcal{O}(T_{\mathrm{rout}})$ *with each request reaching a subproblem.*

Proof. Let I denote a subproblem that is reached by a request R at time t and at PE i. Let $k \geq 0$ denote the number of requests in the message queue of PE i that reach I before R. Only if I is moved to another PE j due to R, I cannot be reached by any request arriving after t until I is put into the message queue of PE j. In the worst case, the dead time is $(k+1)(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}})$. This is the case, when "work" has just been called for the ancestor of I. Then a time Δt passes until the load balancer is next activated. Subsequently, the ancestor is split with an expense of T_{split} and a subproblem is sent away. This cycle is repeated $k+1$ times. Then I is reachable on PE j. The total dead time can be distributed over the $k+1$ requests involved. ■

Now we know the various costs and delays associated with requests. If we could find out how many request are necessary to split all subproblems h times with high probability, we were almost done. However, the question is stated too imprecisely yet. Requests that arrive during a dead time of a subproblem are "lost" for that subproblem. We therefore only consider a subset of all completed requests that has the property to be "sufficiently uniformly" distributed over time.

Definition 2. *A request may be colored* red *if there are at most P other red requests during a time interval T_{dead} after its arrival.*

Lemma 3. *Let $I\langle i\rangle$ denote the subproblem at PE i. For every $\beta > 0$ there is a constant $c > 0$, such that after processing cPh red requests*

$$\mathbf{P}\left[\exists i : \mathrm{gen}(I\langle i\rangle) < h\right] \leq P^{-\beta} \textit{(for sufficiently large } P\textit{).}$$

Proof. For some fixed PE index i, we have

$$\mathbf{P}\left[\exists i : \mathrm{gen}(I\langle i\rangle) < h\right] \leq P\mathbf{P}\left[\mathrm{gen}(I\langle i\rangle) < h\right] \enspace .$$

So it suffices to show that $\mathbf{P}\left[\mathrm{gen}(I\langle i\rangle) < h\right] < P^{-\beta-1}$ for sufficiently large P. We can bound $\mathrm{gen}(I\langle i\rangle)$ by the number of red requests that reach $I\langle i\rangle$. Uncolored requests can be ignored here w.l.o.g.: Although it may happen that an uncolored request reaches $I\langle i\rangle$ and causes one or more subsequent red requests to miss

$I\langle i\rangle$, this split will be accounted to the next following red request and its dead time suffices to explain that the subsequent red requests miss $I\langle i\rangle$. Using a combinatorial treatment, we now show that $\sum_{k<h} P_k \leq P^{-\beta-1}$ where

$$P_k := \mathbf{P}\left[I\langle i\rangle \text{ is reached by } k \text{ red requests}\right] .$$

There are $\binom{chP}{k}$ ways, to choose k red request that are to reach $I\langle i\rangle$. The probability that they are all heading for PE i is P^{-k}. Since there are at most P red requests in the dead time after a request, there are at least $chn - kP$ remaining red request that do not reach $I\langle i\rangle$. The probability of this event is

$$(1-1/P)^{chP-kP} \leq e^{-(ch-k)} .$$

All in all, we have

$$P_k \leq \binom{chP}{k} P^{-k} e^{-(ch-k)} \leq \left(\frac{chPe}{k}\right)^k P^{-k} e^{-(ch-k)} = \left(\frac{che^2}{k}\right)^k e^{-ch}$$

using the Stirling approximation $\binom{m}{k} \leq (me/k)^k$. Since $k < h$, it is easy to verify that the k-dependent part of the above expression is monotonously increasing with k for $c > 1/e$ and can be bounded from above by setting $k = h$, i.e.,

$$P_k \leq \left(ce^2\right)^h e^{-ch} = e^{-h(c-\ln c-2)} .$$

Now $\mathbf{P}\left[\mathrm{gen}(I\langle i\rangle) < h\right]$ can be bounded by

$$he^{-h(c-\ln c-2)} = e^{-h\left(c-\ln c-2-\frac{\ln h}{h}\right)} \leq e^{-h\left(c-\ln c-2-\frac{1}{e}\right)} .$$

Since we assume that $h \in \Omega(\log P)$ there is a c' such that $h \geq c' \ln P$:

$$\mathbf{P}\left[\mathrm{gen}(I\langle i\rangle) < h\right] \leq P^{-c'\left(c-\ln c-2-\frac{1}{e}\right)} \leq P^{-\beta-1}$$

for an appropriate c and sufficiently large P. ■

Now we bound the expense for all requests in order to have cPh red ones among them.

Lemma 4. *Let $c > 0$ denote a constant. Requests can be colored in such a way that an expected work in $\mathcal{O}(hP(\Delta t + T_{\text{split}} + T_{\text{rout}}))$ for all request processing suffices to process chP red requests.*

Proof. Let $R_1, \ldots, R_m$ denote all the requests processed and let $t(R_1) \leq \cdots \leq t(R_m))$ denote the arrival time of R_i. Going through this sequence of requests we color P subsequent requests red and then skip the requests following in an interval of T_{dead}, etc. Since there can never be more than P requests in transit there can be at most $2P$ uncolored requests whose executions overlaps an individual red interval. Therefore, the expense for P red requests can be bounded by PT_{dead} plus the expense for processing $3P$ requests. The expense for this is given in Lemma 1. ■

By combining lemmata 3 and 4 we get a bound for the communication expense of random polling until only atomic subproblems are left.

Lemma 5. *The expected overall expense for communicating, splitting and waiting until there are no more subproblems with* $\mathrm{gen}(I) < h$ *is in* $\mathcal{O}(hP(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}}))$.

Bounding the expense for sequential work – i.e. calls of "work" – is easy. Let T_{poll} denote the (constant) expense for probing the message queue unsuccessfully. It suffices to choose $\Delta t > T_{\mathrm{poll}}/(\epsilon\alpha)$ to make sure that only $(1+\epsilon)T_{\mathrm{seq}}$ time units are spent for those iterations of the main loop where the local subproblem is not exhausted and no requests arrive. All other loop iterations can be accounted to load balancing.

As the last component of our proof, we have to verify that atomic subproblems are disposed of quickly and that termination detection is no bottleneck.

Lemma 6. *If* $\Delta t \in \Omega\big(\min(\frac{T_{\mathrm{atomic}}}{h}, T_{\mathrm{rout}} + T_{\mathrm{split}})\big)$ *and* $\mathrm{gen}(I\,\langle i\rangle) \geq h$ *for all PEs then the remaining execution time is in*

$$\mathcal{O}(T_{\mathrm{atomic}} + h(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}})) \ .$$

Proof. From the definition of h we can conclude that for all remaining subproblems I we have $T(I) \leq T_{\mathrm{atomic}}$. For $\frac{T_{\mathrm{atomic}}}{h} \in \mathcal{O}(T_{\mathrm{rout}} + T_{\mathrm{split}})$, $\mathcal{O}(h)$ iterations (of each PE) with cost $\mathcal{O}(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}})$ each suffice to finish up all subproblems. Otherwise, a busy PE spends at least a constant fraction of its time with productive work even if it constantly receives requests.[3] Therefore, after a time in $\mathcal{O}(T_{\mathrm{atomic}})$ no nonempty subproblems will be left. After a time in $\mathcal{O}(T_{\mathrm{rout}} \log P) \subseteq \mathcal{O}(hT_{\mathrm{rout}})$, the termination detection protocol will notice this condition. ■

The above building blocks can now be used to assemble a proof of Theorem 1. Choose some $\Delta t \in \mathcal{O}(T_{\mathrm{rout}} + T_{\mathrm{split}}) \cap \Omega\big(\min\big(\frac{T_{\mathrm{atomic}}}{h}, T_{\mathrm{rout}} + T_{\mathrm{split}}\big)\big)$ such that $\Delta t > T_{\mathrm{poll}}/(\epsilon\alpha)$ (where T_{poll} is the constant time required to poll the network in the absence of messages). This is always possible and for the frequent case $T_{\mathrm{atomic}}/h \ll T_{\mathrm{rout}} + T_{\mathrm{split}}$ there is also a very wide feasible interval for Δt. Every operation of Algorithm 1 is either devoted to working on a nonempty subproblem or to load balancing in the sense of Lemma 5. Therefore, after an expected time of $(1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + \mathcal{O}(h(1/\epsilon + T_{\mathrm{rout}} + T_{\mathrm{split}}))$ sufficiently many requests have been processed such that only subproblems with $\mathrm{gen}(I) \geq h$ are left with high probability. The polynomially small fraction of cases where this number of requests is not sufficient cannot influence the expectation of the execution time since even a sequential solution of the problem instance takes only $\mathcal{O}(P)$ times as long as a parallel execution. According to Lemma 6, an additional time in $\mathcal{O}(T_{\mathrm{atomic}} + h(1/\epsilon + T_{\mathrm{split}} + T_{\mathrm{rout}}))$ suffices to finish up the remaining subproblems and to detect termination. ■

[3] In the full LogP model even $\Delta t \in \Omega\Big(\min(\frac{T_{\mathrm{atomic}}}{h}, \max(T_{\mathrm{split}} + o, g))\Big)$ suffices.

4 High Probability

In order to keep the algorithm and its analysis as simple as possible, Theorem 1 only bounds the expected parallel execution time. In [33] it is also shown how the same bounds can be obtained with high probability. The key observation is that Martingale tail bounds can be used to bound the sum of all queue lenghts encounterd by requests if the maximal queue length is not too large.

Theorem 2. *For Δt and ϵ as in Theorem 1,*

$$T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{par}}}{P} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(\tfrac{1}{\epsilon} + T_{\text{split}} + T_{\text{rout}})\right)$$

if $h \in \Omega(P \log P)$ or queue lengths in $\Omega\left(\sqrt{P}\right)$ are avoided by algorithmic means.[4]

5 Discussion

Tree shaped computations represent an extreme case for parallel computing in two respects. On the one hand, parallelism is very easy to expose since subproblems can be solved completely independently. Apart from that they are the worst case with respect to irregularity. Not only can splitting be arbitrarily uneven (only constrained by the maximum splitting depth h) but it is not even possible to estimate the size of a subproblem. Considering the simplicity of random polling and its almost optimal performance (both in theory and practice) the problem of load balancing tree shaped computations can largely be considered as solved.

Although tree shaped computations span a remarkably wide area of applications, an important area for future research is to generalize the analysis to models that cover dependencies between subproblems. The predictable dependencies modeled by multithreaded computations [2] are one step in this direction. But in many classic search problems the main difficulty are heuristics that prune the search tree in an unpredictable way.

References

1. G. Aharoni, Amnon Barak, and Yaron Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, 9:163–174, 1993.
2. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.
3. S. Arvindam, V. Kumar, V. N. Rao, and V. Singh. Automatic test pattern generator on parallel processors. Technical Report TR 90-20, University of Minnesota, 1990.

[4] Let $\tilde{\mathrm{O}}(\cdot)$ denote the following shorthand for asymptotic behavior with high probability [20]: A random variable X is in $\tilde{\mathrm{O}}(g(P))$ iff $\forall \beta > 0 : \exists c > 0 : \exists P_0 : \forall P \geq P_0 : \mathbf{P}[X \leq cf(P)] \geq 1 - P^{-\beta}$

4. G. S. Bloom and S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.
5. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foudations of Computer Science*, pages 356–368, Santa Fe, 1994.
6. M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.
7. S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, 1994.
8. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, 1993.
9. W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*. Dissertation, TU München, 1992.
10. P. Fatourou and P. Spirakis. Scheduling algorithms for strict multithreaded computations. In *ISAAC: 7th International Symposium on Algorithms and Computation*, number 1178 in LNCS, pages 407–416, 1996.
11. P. Fatourou and P. Spirakis. A new scheduling algorithm for general strict multithreaded computations. In *13rd International Symposium on DIStributed Computing (DISC'99)*, Bratislava, Slovakia, 1999. to appear.
12. 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.
13. R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.
14. C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, number 1332 in LNCS, Cracow, Poland, 1997.
15. V. Heun and E. W. Mayr. Efficient dynamic embedding of arbitrary binary trees into hypercubes. In *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, number 1117 in LNCS, 1996.
16. J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
17. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
18. V. Kumar and G. Y. Ananth. Scalable load balancing techniques for parallel computers. Technical Report TR 91-55, University of Minnesota, 1991.
19. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.
20. 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.
21. S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, 1990.
22. F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.
23. M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.

24. A. Nonnenmacher and D. A. Mlynski. Liquid crystal simulation using automatic differentiation and interval arithmetic. In G. Alefeld and A. Frommer, editors, *Scientific Computing and Validated Numerics*. Akademie Verlag, 1996.
25. W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.
26. V. N. Rao and V. Kumar. Parallel depth first search. *International Journal of Parallel Programming*, 16(6):470–519, 1987.
27. A. Reinefeld. Scalability of massively parallel depth-first search. In *DIMACS Workshop*, 1994.
28. P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.
29. 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.
30. P. Sanders. Better algorithms for parallel backtracking. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 333–347, 1995.
31. P. Sanders. A scalable parallel tree search library. In S. Ranka, editor, *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, Honolulu, Hawaii, 1996.
32. P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. PhD thesis, University of Karlsruhe, 1997.
33. P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.
34. P. Sanders. Tree shaped computations as a model for parallel applications. In *ALV'98 Workshop on application based load balancing*. SFB 342, TU München, Germany, March 1998. `http://www.mpi-sb.mpg.de/~sanders/papers/alv.ps.gz`.
35. E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In C. D. Houstis, E. N.; Papatheodorou, T. S.; Polychronopoulos, editor, *Proceedings of the 1st International Conference on Supercomputing*, number 297 in LNCS, pages 985–993, Athens, Greece, June 1987. Springer.
36. R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal Parallel and Distributed Processing, Special Issue on Parallel and Distributed Data Structures*, 49:135–145, 1998.
37. I. C. Wu and H. T. Kung. Communication complexity of parallel divide-and-conquer. In *Foudations of Computer Science*, pages 151–162, 1991.