Randomized Receiver Initiated Load Balancing Algorithms for Tree Shaped Computations

Peter Sanders

Max-Planck-Institute for Computer Science

66123 Saarbrücken, Germany

E-mail: sanders@mpi-sb.mpg.de

Abstract

Many applications in parallel processing have to traverse large, implicitly defined trees with irregular shape. The receiver initiated load balancing algorithm *asynchronous random polling* has long been known to be very efficient for these problems in practice. Tight bounds for the parallel execution time in the LogP model are derived based on the parameters of a problem model called *tree shaped computations*. This model incorporates the problem size, the cost for basic operations, a measure of granularity and an easy to quantify parameter which limits irregularity. Then, with *poll-and-shuffle*, an asymptotically even more efficient algorithm is introduced. By using predominantly local communications it increases the usable communication bandwidth on hypercubic networks and meshes by a logarithmic factor. These analytic results are complemented by practical refinements and implementation results which successfully apply a portable and reusable library on machines with up to 1024 processors.

Keywords: adaptive granularity control, asynchronous algorithm, hypercubic and mesh network, parallel backtrack search, parallel random permutation, randomized load balancing algorithm

1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking principle for traversing large irregularly shaped trees [8, 9, 15, 13, 19]. Similar problems also play a role in parallel programming languages [1, 12] and even for loop scheduling and some numerical problems like adaptive numerical integration it can be useful to view the computations as an implicitly defined tree. Section 2 introduces the abstract model of *tree shaped computations* which makes the common properties of these applications visible while hiding unneccessary details thereby facilitating generic algorithms and implementations.

For parallelizing tree shaped computations, a load balancing scheme is needed which 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 first consider *random polling* dynamic load balancing, a simple algorithm which avoids this problem: 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. Previous results on this algorithm, other *receiver initiated preemptive* algorithms and different approaches are surveyed in Section 3. Then random polling is analyzed in Section 4. The algorithm turns out to be very efficient for a wide range of applications and parallel architectures.

The *poll-and-shuffle* algorithm introduced in Section 5 is asymptotically even more efficient on interconnection networks like hypercubes, butterflies or meshes because it replaces the global communication of random polling with local communication and occasional random permuations.

Section 6 complements these analytical discussions with some implementation results which further underline the already well known practical merits of randomized receiver initiated load balancing for different applications and for machines ranging from workstation clusters to massively parallel computers. Finally, Section 7 summarizes the paper and discusses some possible future research.

2 The Model

2.1 Machine Model

For the random polling algorithm, we basically adopt the LogP model [6] 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 $T_{\mathrm{rout}} := L + o + g$ units of time. So the analysis also applies to the widespread messaging protocols which block until a message has been copied into the message queue of the recipient.

For the poll-and-shuffle algorithm we use a more detailed model of the interconnection network and differentiate between the time for global randomized routing and neighborhood communication which takes time proportional to the message length. Since in this context, analyzing even asynchronous routing alone poses many open problems we restrict ourselves to synchronized phases of communication and computation.

2.2 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

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

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 while 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* which 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* $\text{split}(I)$ which 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 [23].

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.

What makes parallelization difficult, is that the *size*, i.e., the execution time $T(I) := \min\{t \mid \text{work}(I, t) = I_{\emptyset}\}$, 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 $\forall I : \text{split}(I) = (I_1, I_2) \implies T(P) = T(I_1) + T(I_2)$ regardless when and where I_1 and I_2 are worked on. For a detailed discussion when this assumption is strictly warranted and when it is a good approximation, refer to [29, 30]. Even if the condition is violated, our treatment is still useful for handling the load balancing aspect of the application, while it is up to the application to minimize detrimental dependencies between subproblems. Section 6.3 presents an example.

Next we quantify some guaranteed "progress" made by splitting subproblems. Every subproblem I belongs to a *generation* $\text{gen}(I)$ recursively defined by $\text{gen}(I_{\text{root}}) := 0$ and $\text{split}(I) = (I_1, I_2) \implies \text{gen}(I_1) = \text{gen}(I_2) = \text{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* h which guarantees that the size of subproblems with $\text{gen}(I) \geq h$ 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 which cannot be split any more. Since h is the only factor which 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 $T_{\text{seq}} := T(I_{\text{root}})$ 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

$$T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + T_{\text{rest}}(h, l, T_{\text{split}}, T_{\text{atomic}}, P, T_{\text{rout}}, \epsilon) \tag{1}$$

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

3 Related Work

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. [14, 5, 24]).

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 [12] or game tree search [8]. Randomized partner selection goes at least back to [9]. 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 [23]. Polling PEs in a "global round robin" fashion [14] avoids this because no large subproblems can "hide". Execution times $T_{\mathrm{par}} \in \mathrm{O}\left(\frac{T_{\mathrm{seq}}}{P} + hT_{\mathrm{count}}\right)$ can be achieved where T_{count} is the time for incrementing a global counter. However, even sophisticated distributed counting algorithms have $T_{\mathrm{count}} \in \Omega\left(T_{\mathrm{rout}} \log P / \log\log P\right)$ [31]. 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 $\mathbf{E}T_{\mathrm{par}} \in \mathrm{O}\left(\frac{T_{\mathrm{seq}}}{P} + hT_{\mathrm{rout}} \log P\right)$ [14]. Tree shaped computations are a generalization of the α-splitting model used in [14]. The gap between analysis and practical experience was closed in [25, 26] by showing that $T_{\mathrm{par}} \leq (1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + \mathrm{O}(hT_{\mathrm{rout}})$ with high probability.

Independently, studies on scheduling multithreaded computations in the context of the *Cilk* project lead to the restricted class of *fully strict multithreaded computations* for which random polling (called randomized work stealing there) leads to a very efficient scheduling algorithm [3]. For many underlying applications the two models can be translated into each other. The critical path length T_∞ in Cilk then becomes $hT_{\mathrm{split}} + T_{\mathrm{atomic}}$ for tree shaped computations. Cilk is able to model certain predictable dependencies between subproblems while tree shaped computations allow for different splitting strategies which may significantly decrease h [23]. The Cilk model is natural for a multi-threaded programming language, while tree shaped computations are directly useful for a portable and reusable library [28, 30]. In the following, we concentrate on tree shaped computations. Adapting these results to Cilk or some more general model encompassing both approaches is an interesting area for future work however.

All the analytical results above make simplifying assumptions which are only warranted if communication takes place in synchronized communication rounds. However, the actual implementations are asynchronous in nature because globally synchronized communication is undesirable. Idle PEs have to wait for the next communication round and the network capacity is left unexploited most of the time. Unfortunately, we cannot fully transfer an analysis for the synchronous case to an asynchronous model since subproblems which are "in transit" cannot be split and long request queues can build up around PEs which have "difficult to split" subproblems. In Section 4 we show that the latter problems do not inhibit the efficiency of a simple asynchronous random polling algorithm.

The poll-and-shuffle algorithm considered in Section 5 was first described in [27]. The algorithm can also be viewed as an on-line tree embedding algorithm whose node load and (average) dilation is as good as previously known algorithms for this model [18, 22, 11, 10] but has the advantage that its built-in adaptive granularity control achieves efficiency arbitrarily close to one for sufficiently large problems.

Most results presented here are based on the doctoral dissertation [29] (in German).

4 Asynchronous Random Polling

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 in intervals Δt.[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.

[2] If the machine supports it, polling can be replaced by more efficient and more elegant interrupt mechanisms.

```
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 PE chosen uniformly at random
        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)                          (* do some sequential work *)
        if there is an incoming request from PE j then
            (I, I') := split(I); send I' to PE j
```

Figure 1: Basic algorithm for asynchronous random polling.

Concurrently, a distributed termination detection protocol is run which recognizes when all PEs have run out of work. We have adapted the *four counter* method [20] for this purpose. Each PE counts the number of sent and received messages which 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 [20], we use a tree based asynchronous global reduction operation. This is a simple and portable way to bound the termination detection delay by $\mathrm{O}(T_{\mathrm{rout}} \log P)$.

We do not explicitly handle reporting results of the computation here since this is quite cheap for many applications. For example, for numeric integration the results for all subproblems solved on each PE can be added together by a single global reduction.

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

In Section 4.2 we additionally show that large deviations from this time bound are improbable. 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 2. 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 3.

1. *The total amount of active CPU work expended for processing a request is bounded by* $T_{\text{split}} + \mathrm{O}(T_{\text{rout}})$.

2. *If any requests have arrived at a PE, at least one of the requests is answered every* $\Delta t + T_{\text{split}} + \mathrm{O}(T_{\text{rout}})$ *time units.*

3. *The expected elapsed time between the arrival of a message and sending the corresponding reply is in* $\mathrm{O}(\Delta t + T_{\text{split}} + T_{\text{rout}})$.

Proof. **1:** A request triggers at most one split. The total expense for sending and receiving is in $\mathrm{O}(T_{\text{rout}})$. **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 ore more subsequent load request, there is a *dead time* interval during which it cannot be reached by any other request.

Lemma 4. *All dead times can be covered by associating a dead time* $T_{\text{dead}} = \Delta t + T_{\text{split}} + \mathrm{O}(T_{\text{rout}})$ *with each request reaching a subproblem.*

Proof. Let I denote a subproblem which 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 which 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 $(k+1)(\Delta t + T_{\text{split}} + T_{\text{rout}})$. 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 which arrive during a dead time of a subproblem are "lost" for that subproblem. We therefore only consider a subset of all completed requests which has the property to be "sufficiently uniformly" distributed over time.

Definition 5. 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 6. *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 $\mathbf{P}[\exists i : \mathrm{gen}(I\langle i\rangle) < h] \leq P^{-\beta}$ (for sufficiently large P).*

Proof. We have $\mathbf{P}[\exists i : \mathrm{gen}(I\langle i\rangle) < h] \leq P\mathbf{P}[\mathrm{gen}(I\langle i\rangle) < h]$ for some fixed PE index i. So it suffices to show that $\mathbf{P}[\mathrm{gen}(I\langle i\rangle) < h] < P^{-\beta-1}$ for sufficiently large P. $\mathrm{gen}(I\langle i\rangle)$ can be bounded by the number of red requests which 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 ore 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

$$\sum_{k<h} P_k \leq P^{-\beta-1} \text{ where } P_k := \mathbf{P}[I\langle i\rangle \text{ is reached by } k \text{ red requests}] \quad .$$

There are $\binom{chP}{k}$ ways, to choose k red request which 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 which 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 $c > \frac{1}{e}$ and can be bounded from above by setting $k = h$, i.e.,

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

Now $\mathbf{P}[\mathrm{gen}(I\langle i\rangle) < h]$ 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 $h \geq c' \ln P$:

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

for an appropriate c and sufficiently large P. $\square$

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

Lemma 7. *Let $c > 0$ denote a constant. Requests can be colored in such a way that an expected work in* $\mathrm{O}(hP(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}}))$ *for all request processing suffices to process* chP *red requests.*

Proof. Let $R_1, \dots, R_m$ 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 3. $\square$

By combining lemmata 6 and 7 we get a bound for the communication expense of random polling until only atomic subproblems are left.

Lemma 8. *The expected overall expense for communicating, splitting and waiting until there are no more subproblems with* $\mathrm{gen}(I) < h$ *is in* $\mathrm{O}(hP(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}}))$.

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 $\Delta t > T_{\mathrm{poll}}/\epsilon$ 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 9. *If* $\Delta t \in \Omega\left(\min(\frac{T_{\mathrm{atomic}}}{h}, T_{\mathrm{rout}} + T_{\mathrm{split}})\right)$ *and* $\mathrm{gen}(I\langle i\rangle) \geq h$ *for all PEs then the remaining execution time is in* $\mathrm{O}(T_{\mathrm{atomic}} + h(\Delta t + T_{\mathrm{split}} + T_{\mathrm{rout}}))$.

Proof. From the definition of h we can conclude that for all remaining subproblems I we have $T(I) \leq T_{\mathrm{atomic}}$. For $\frac{T_{\mathrm{atomic}}}{h} \in \mathrm{O}(T_{\mathrm{rout}} + T_{\mathrm{split}})$, $\mathrm{O}(h)$ iterations (of each PE) with cost $O(\Delta t + T_{\text{split}} + T_{\text{rout}})$ 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 $O(T_{\text{atomic}})$ no nonempty subproblems will be left. After a time in $O(T_{\text{rout}} \log P) \subseteq O(hT_{\text{rout}})$, the termination detection protocol will notice this condition. $\square$

The above building blocks can now be used to assemble a proof of Theorem 1. Choose some $\Delta t \in O(T_{\text{rout}} + T_{\text{split}}) \cap \Omega\left(\min\left(\frac{T_{\text{atomic}}}{h}, T_{\text{rout}} + T_{\text{split}}\right)\right)$ such that $\Delta t > T_{\text{poll}}/\epsilon$ (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 $T_{\text{atomic}}/h \ll T_{\text{rout}} + T_{\text{split}}$ 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 8. Therefore, after an expected time of $(1+\epsilon)\frac{T_{\text{seq}}}{P} + O(h(1/\epsilon + T_{\text{rout}} + T_{\text{split}}))$ sufficiently many requests have been processed such that only subproblems with $\text{gen}(I) \geq h$ 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 $O(P)$ times as long as a parallel execution. According to Lemma 9, an additional time in $O(T_{\text{atomic}} + h(1/\epsilon + T_{\text{split}} + T_{\text{rout}}))$ suffices to finish up the remaining subproblems and to detect termination. $\blacksquare$

4.2 Analysis with High Probability

In order to keep the algorithm and its analysis as simple as possible, Theorem 1 only bounds the expected parallel execution time. We now outline how the same bounds can be obtained with high probability.

Theorem 10. *For Δt and ϵ as in Theorem 1,*

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

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

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

The proof is largely analogous to the proof of Theorem 1 except that the waiting times due to long message queues have to be considered more closely. It suffices to show that the total waiting time suffered by $\mathrm{O}(hP)$ requests is in $\tilde{\mathrm{O}}(\Delta t + T_{\text{split}} + T_{\text{rout}})$. This is equivalent to showing that the sum of the queue lengths met by $\mathrm{O}(hP)$ requests is in $\tilde{\mathrm{O}}(hP)$. Both cases in Theorem 10 can be reduced to the following criterion:

Lemma 11. *If $\bar{q}$ is an upper bound for the queue length during a run of random polling and if $\bar{q} \in \mathrm{O}\left(\sqrt{\frac{hP}{\log P}}\right)$, then the sum of all queue lengths met by $\mathrm{O}(hP)$ requests is in $\tilde{\mathrm{O}}(hP)$.*

Proof. Consider some $\beta > 0$. Let $m \le ahP$ denote the number of requests considered and let $\bar{q} \le \sqrt{\frac{bhP}{\log P}}$ be a bound on the queue lengths. (with appropriate constants a and b). Let the random variable T_i $(1 \le i \le m)$ denote the index of the PE where request i arrives and let Y_i $(1 \le i \le m)$ denote the queue length met by request i. Let $X := \sum_{i=1}^{m} Y_i$ and let $X_i := \mathbf{E}[X \mid T_1, \ldots, T_i]$ $(0 \le i \le m)$ denote the conditional expected value of X depending on T_1 through T_i. According to [21, Theorem 4.13 and Example 4.12] X_0, $\ldots$, X_m is a Martingal sequence. Furthermore, we have $X_i = \sum_{j=1}^{i} Y_j + \mathbf{E}\sum_{j=i+1}^{m} Y_j = \sum_{j=1}^{i} Y_j + \sum_{j=i+1}^{m} \mathbf{E}Y_j$ and therefore $|X_{i+1} - X_i| = |Y_{i+1} - \mathbf{E}Y_{i+1}| \le \bar{q}$. Let c denote a constant still to be determined. Using Azumas inequality [21, Theorem 4.16] we can infer

$$\begin{aligned}
\mathbf{P}[|X - \mathbf{E}X| \ge chP] = \mathbf{P}[|X_m - X_0| \ge chP] &\le 2e^{-\frac{(chP)^2}{2m\bar{q}^2}} \\
&\le 2e^{-\frac{(chP)^2}{2(ahP)(bhP/\log P)}} = 2e^{-\frac{c^2 \log P}{2ab}} = 2P^{-\frac{c^2}{2ab}} \\
&\le P^{-\beta} \text{ for } c > \sqrt{\frac{\beta}{2ab}} \text{ and sufficiently large } P \ .
\end{aligned}$$

Analogous to the proof of Lemma 3 we have $\mathbf{E}X \le ahP$. All in all, the sum of the queue lengths met, X, is in $\tilde{\mathrm{O}}(hP)$. □

[4] Let $\tilde{\mathrm{O}}(\cdot)$ denote the following shorthand for asymptotic behavior with high probability [16]: A random variable X is in $\tilde{\mathrm{O}}(g(P))$ iff $\forall\beta > 0 : \exists c > 0 : \exists P_0 : \forall P \ge P_0 : \mathbf{P}[X \le cf(P)] \ge 1 - P^{-\beta}$

Due to the trivial bound $\bar{q} \le P$ the criterion applies for $h \in \Omega(P \log P)$. Since $h \in \Omega(\log P)$ the same is true if $\bar{q} \in \mathrm{O}\left(\sqrt{P}\right)$. ■

We see that isolated long queue lengths do not affect the overall runtime for problem instances with $h \in \Omega(P \log P)$. Furthermore, there are some simple algorithmic measures in order to keep the queues short. A radical approach would be to interrupt the normal computation when any queue length exceeds $c\sqrt{P}$ for some constant c in order to drain the system of all requests. On most systems, the effort for this would be much smaller than the effort for sending $\Omega\left(P\sqrt{P}\right)$ random requests which are (with high probability) necessary to build long queues. In addition, for $h \in \mathrm{O}\left(\sqrt{P}\right)$ it is unlikely that such a large number of requests is sent at all.

A much simpler approach would be to act only locally on long queues. Since the subproblem at this PE will have accumulated a large $\mathrm{gen}(I)$ anyway, we can simply reject some requests without splitting until the queue length has normalized.

4.3 Refinements

A rather obvious improvement is to initialize the PEs by broadcasting the root problem to all PEs and then partition it using $\log P$ subsequent splits where the bits of the PE index are used to decide which subproblem is kept. (This can be generalized for the case where P is not a power of two.) In practice, this broadcast is almost free because information common to all subproblems should be broadcast anyway in order to keep the subproblem descriptions themselves small. However, from a theoretical point of view basic random polling is already remarkably efficient. In [29] it is demonstrated that the expected number of required message exchanges until all PEs are busy is bounded by $\log P + \log\ln P + 1$ (using a synchronous model).

An important practical advantage of random polling is that it even works on inhomogeneous networks with external load by other users as long as load requests are still answered. If this becomes necessary, a search process can even switch to blocking receives when the system detects the presence of an interactive user.

5 The Poll-and-Shuffle Algorithm

Regarding the number of messages sent, random polling seems to be hardly improvable. At least no deterministic load balancer [32] nor any receiver initiated load balancer [29] can achieve good efficiency without exchanging $\Omega(hP)$ messages for some instances. On the other hand, as discussed in Section 3, the tempting solution to ask only neighbors in the interconnection network for work leads to a buildup of clusters of busy PEs resulting in insuficient splitting of large subproblems and many useless load requests. Therefore, we introduce an algorithm for hypercubes with mixed global and local communication which is then analyzed in Section 5.1. In Section 5.2 this algorithm is adapted to other interconnection networks. While in the analysis of asynchronous random polling we took great pains to stick to a practically realistic model, we now focus on the basic principle of avoiding global communciation and prefer simplicity over efficiency whenever the asymptotic bounds are not in danger.

Figure 2 gives pseudo-code for poll-and-shuffle load balancing on a $\log P$-dimensional hypercube network. Let the shorthand $I\langle j\rangle$ stand for the subproblem PE j is responsible for and let $\oplus$ stand for *exclusive-or*. The principle is similar to random polling but the PEs operate synchronously in work phases of duration Δt. $\log P$ phases form a cycle. After computation phase i within a cycle, requests can be exchanged along dimension i of the hypercube. The idea is that by using a fresh dimension for each iteration, clustering effects cannot become visible within a cycle. After a cycle, the subproblems are permuted randomly so that any possibly existing clusters would be completely dissolved.

A random permutation over PE indices can be computed in time $\mathrm{O}(\log P) + \tilde{\mathrm{O}}(\log P/\log\log P)$ on a hypercube by routing each PE index to a random PE; permuting locally; enumerating the messages using a prefix sum and rerouting the messages to the PE with this number. (For a detailed analysis refer to [29].)

var I : Subproblem

I := **if** $i_{\mathrm{PE}} = 0$ **then** I_{root} **else** $I_{\emptyset}$

while not finished **do** synchronously

  for i:=0 **to** $\log P - 1$ **do** synchronously

    $I := \mathrm{work}(I, \Delta t)$

    if $T(I) = 0$ **then** $(I, I\langle i_{\mathrm{PE}} \oplus 2^i\rangle) := \mathrm{split}(I\langle i_{\mathrm{PE}} \oplus 2^i\rangle)$

  endfor

  globally permute subproblems randomly

Figure 2: Poll-and-shuffle on a hypercube.

5.1 Analysis

We prove a bound similar to that of Theorem 1 but now the term for the communication overhead per communication is not T_{rout} but only $\mathrm{O}(l)$ where l is a bound for the message length needed to represent a subproblem. For simplicity, ϵ is treated as a constant now.

Theorem 12. $T_{\mathrm{par}} \leq \left(1 + \epsilon + \tilde{\mathrm{O}}\left(\frac{1}{\log\log P}\right)\right)\frac{T_{\mathrm{seq}}}{P} + \tilde{\mathrm{O}}\left(T_{\mathrm{atomic}} + h(l + T_{\mathrm{split}})\right)$ *for any constant* $\epsilon > 0$ *and an appropriate choice of* Δt*.*

The basic idea for the analysis is also similar to random polling. Again we start with the most difficult problem namely bounding the number of phases with many idle PEs.

If we omit all edges along dimensions $i, \ldots, \log P - 1$, the hypercube is partitioned into i-dimensional subcubes we call *i-cubes*. Let S, T denote some arbitrary subproblems. Then $S \overset{i}{—} T$ denotes the event that S and T are neighbors along dimension i. In random polling, a subproblem was equally probable to receive a request from any idle PE. Similarly, our analyis is now based on the fact that (almost) all pairings $S \overset{i}{—} T$ are equiprobable in phase i:

Lemma 13. *Consider the communication after any i-th phase of a cycle. Let S denote an arbitrary subproblem and $\mathcal{T}$ the set of subproblems* not *in the same i-cube as*

S. Empty subproblems are assumed to be distinguishable. Then, for any $T, T' \in \mathcal{T}$, $\mathbf{P}\left[S \overset{i}{-} T \mid (S,\mathcal{T})\right] = \mathbf{P}\left[S \overset{i}{-} T' \mid (S,\mathcal{T})\right]$ where $(S,\mathcal{T})$ denotes the subset of the event space leading to the same S and $\mathcal{T}$ at the considered point in time.

Proof. Wlog., assume that "work" und "split" work deterministically (otherwise we could make a case distinction for any possible outcome of random choices in these operations).

Let M denote the subgroup of the permutation group $\mathbf{S}_P$ defined by exchanging i-cubes and mirroring individual i-cubes. (In [29] the proof is executed in more detail using formal definitions of the permutations involved.) From elementary group theory, we know that M (like any supgroup) can be used to partition $\mathbf{S}_P$ into equivalence classes by defining the coset $C_\pi := \pi M = \{\pi m \mid m \in M\}$ for any $\pi \in \mathbf{S}_P$.

Now let π denote the permutation applied in the last shuffling operation. Let $C_T := \left\{\sigma \in C_\pi \mid S \overset{i}{-} T\right\}$ and $C_{T'} := \left\{\sigma \in C_\pi \mid S \overset{i}{-} T'\right\}$. It suffices to show that $|C_T| = |C_{T'}|$. A simple and elegant way to do this is to show that there is a bijective mapping $f_{TT'} \in {C_{T'}}^{C_T}$. By symmetry it suffices to give an injection. The following sequence does the job: First mirror the i-cube where T is located until T and T' have the same position within their respective i-cube. Then swap the i-cubes where T and T' are located. Since this $f_{TT'}$ is a sequence of injective mappings from the group M, it is injective itself and also an element of M. So, $|C_T| = |C_{T'}|$ and this settles the claim. □

Now we have a tool for proving that in (most) phases with low PE utilization, any subproblem is split with some minimum probability. Consider some constant $\gamma \in (0,1)$ we are free to choose. Call a phase *red* if $i < \log P - \log \frac{2}{\gamma}$ and at least γP PEs are idle at its end.

Lemma 14. *$\tilde{\mathrm{O}}(h)$ red phases suffice to reduce all subproblems to size at most T_{atomic}.*

Proof. Consider a fixed subproblem S. After a red phase at least $\gamma P - 2^i$ subproblems outside the i-cube of S are empty. Color $\gamma P - 2^i$ of those *red*. By Lemma 13 each red subproblem is equiprobable to be a neighbor of S along dimension i and will therefore lead to a split. So, the probability that S is split by a request from a PE with a red subproblem is $\frac{\gamma P - 2^i}{P - 2^i} \geq \gamma/2$ for $i < \log P - \log \frac{2}{\gamma}$ and these probabilities are independent. Define a random variable $X_j := 1$ iff S is split in the j-th red phase by a PE with a red subproblem. Using a standard Chernoff-bound argument and similar to the proof of Lemma 6 it follows that $\mathbf{P}\left[\text{gen}(S) < h\right] \leq \mathbf{P}\left[\sum_{j \leq 2c\beta h/\gamma} X_j < h\right] \leq P^{-\beta-1}$ for $h \in \Omega(\log P)$, any constant β and an appropriately choosen constant c. The probaility that any subproblem has generation less then h (and thereby possibly nonatomic size) can be at most a factor of P larger. □

We now outline how the proof of Theorem 12 can be completed: First, note that a cycle with $\log P$ phases can be completed in time $\log P\left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log P}\right)\right)$ where c is a constant only depending on the speed of the communication links. This is true even if we use unpipelined dimension order packet routing for finding and performing the random permutation [17, Theorem 3.34].

There can be at most $\left\lceil \frac{T_{\text{seq}}}{P(1-\gamma)\Delta t} \right\rceil$ phases with at least $(1-\gamma)P$ busy PEs at the end because after that no work could be left.

By Lemma 14, $\tilde{\mathrm{O}}(h)$ red phases suffice to reduce all subproblems to atomic size and $\lceil T_{\text{atomic}}/\Delta t \rceil$ subsequent phases suffice to finish them up. In each cycle, a constant number of phases with $i \geq \log P - \log \frac{2}{\gamma}$ might neither have high PE utilization nor substantially contribute to splitting but for sufficiently large P their contribution to the overall execution time is negligible.

A termination detection can be performed using a global and-reduction once per cycle. The costs for this can be charged to the $\log P$ neighborhood exchanges per cycle.

By choosing $\Delta t = 2(1 + \frac{1}{\epsilon})(T_{\text{split}} + cl)$ and $\gamma = 1 - 1/\sqrt{1 + \epsilon/2}$ this information can be combined to yield the desired bound for sufficiently large P. The calculations needed for this are simple yet take about three pages in [29] so we omit them. ■

5.2 Other Networks

Note that unless l is very small, Theorem 12 is no improvement over random polling for some hypercube variants which can exploit their large bisection width to get $T_{\text{rout}} \in$

$\tilde{\mathrm{O}}(\log P + l)$. However, poll-and-shuffle is a normal hypercube algorithm in the sense of [17] and can therefore be emulated with constant slowdown on hypercubic networks such as butterfly, de Bruijn, shuffle-exchange or cube-connected-cycle. These constant degree networks have a bisection width in $\Theta(P/\log P)$ and therefore poll-and-shuffle can be a factor $\Theta(\log P)$ faster than random polling there. Since we can charge the subproblem migrations required by the hypercube emulation to the neighborhood exchanges, the local computations can be considered to have perfect speedup yielding an equally strong bound as for the full hypercube:

Corollary 15. *For poll-and-shuffle on hypercubic networks*

$$T_{\mathrm{par}} \leq \left(1 + \epsilon + \tilde{\mathrm{O}}\left(\frac{1}{\log\log P}\right)\right)\frac{T_{\mathrm{seq}}}{P} + \tilde{\mathrm{O}}\left(T_{\mathrm{atomic}} + h(\frac{1}{\epsilon} + l + T_{\mathrm{split}})\right)$$

for any constant $\epsilon > 0$ and an appropriate choice of Δt.

Similarly, it is easy to embed a hypercube into an r-dimensional mesh in such a way that a cycle can be completed in time $\log P(\Delta t + T_{\mathrm{split}}) + \mathrm{O}(lP^{1/r})$ if we use worst-case efficient algorithms like sorting for the random permutations.

Corollary 16. *For poll-and-shuffle on meshes*

$$T_{\mathrm{par}} \leq (1+\epsilon)\frac{T_{\mathrm{seq}}}{P} + \tilde{\mathrm{O}}\left(T_{\mathrm{atomic}} + h\left(\frac{1}{\epsilon} + \frac{lP^{1/r}}{\log P} + T_{\mathrm{split}}\right)\right)$$

for any constant $\epsilon > 0$ and an appropriate choice of $\Delta t \in \Theta\left(T_{\mathrm{split}} + \frac{lP^{1/r}}{\log P}\right)$.

A hypercube can even be embedded into a mesh where P is not a power of two. In this case some mesh PEs have to emulate several hypercube PEs. But as for networks of interactive workstation this does not impede the possibility to achieve high efficiencies.

Also note, that the $\Theta(\log P)$ factor improvement over random polling implies a larger useable *bandwidth* so that it can also materialize on machines with fast hardware routers where *latencies* are usually assumed to be independent of the network diameter.

5.3 Refinements

On current relatively small machines, basic poll-and-shuffle cannot compete with asynchronous random polling. The penalty for synchronization and periodic global permutations more than outweighs the asymptotically dominant locality advantage.

A simple observation is that it is sufficient to synchronize neighboring PEs along dimension i after phase i. Also, the analysis is unaffected if busy PEs waiting for synchronization continue their work and idle PEs can ask for work along a dimension $j < i$ (on a mesh they can even ask anywhere in their i-cube).

Expensive permutations can be saved if they are only triggered when the PE utilization is low. Also, local requests could be mixed with occasional global requests to avoid some global permutations. One variant of this analytically difficult but practically appealing idea is considered in Section 6.2.

6 Practical Aspects

The results of the analysis of random polling are quite clear, backed up by a lot of previous implementation results and are easy to interpret. As long as T_{atomic} is not huge it can be neglected. T_{split} is easy to measure and of little significance for slow networks. The most important parameter is h which can be estimated based on the problem size and the properties of the splitting function. Together with the theoretical bounds these simple considerations suffice to estimate the performance for worst case instances. Load balancing will be efficient as long as T_{seq}/P is large compared to the time for $\mathrm{O}(h)$ global message exchanges and splits. This is important in practice and cannot be replaced even by the most detailed empirical evaluation which can never make reliable predictions beyond the instances measured.

In this experimental section, we therefore concentrate on orthogonal results which are not covered by the theoretical analysis. In Section 6.1 we discuss implementation aspects. Section 6.2 studies the impact of some improvements to random polling. An application going beyond the model of tree shaped computations is investigated in Section 6.3.

6.1 Portability, Reusability, Efficiency

The library PIGSeL (Parallel Implicit Graph Search Library) [28, 7] was built around the concept of tree shaped computations. An application has to implement the operations "work" and "split" together with functions for packing and unpacking of subproblems, for initialization and for solution handling.[5] The parallelization has random polling as its default load balancer and can be reused by different applications.

The library is also portable except for a thin communication layer which has been adapted to MPI, PVM, Cosy [4] and PARIX so far, so that it runs on most MIMD machines. The performance of the library at least equalizes previous less portable implementations. The same application code runs equally well on a network of workstations and on a massively parallel machine. Perhaps most interesting are measurements on a Parsytec GCel/3 with a 32×32 mesh of transputers.[6] A heuristic backtrack search for optimal "Golomb rulers" [2, 28, 29] achieves an almost perfect speedup of 958 for a quite small problem instance with (12.4 s parallel execution time) even though the GCel has a high penalty for global communication. For a very small problem instance with parallel execution time 0.88 s, the efficiency is still above 1/2 (speedup 578). Previous research on massively parallel search in irregular trees only achieves good efficiency for problem instances which are an order of magnitude larger [15, 24].

6.2 Improving Random Polling?

In order to test some algorithmic ideas for improving random polling we made measurements with PIGSeL and the *15-puzzle* [13] which is a well known toy widely used as a benchmark in AI. You have to shift 15 scrambled squares in a 4×4 frame into the right order. In order to get a large sample of problem instances of widely varying size we used

[5] There is a higher level interface centered around an abstraction for search tree nodes.

[6] We thank the Paderborn Center for Parallel Computing (PC^2) for making this machine available.

the 100 instances for the 15-puzzle given in [13]. Each instance requires a series of iteratively deepened searches. We used all failing iterations as individual test instances since those are fully modeled by tree shaped computations and exhibit no speedup anomalies.

It turns out that the fast initialization from Section 4.3 can increase speedup by a factor up to three for very tiny instances which achieve almost no speedup otherwise. This is useful for the initial iterations of iterative deepening searches since it obviates specialized treatments as they are used in other studies such as [24]. If the instances are known to be small it pays to disable dynamic load balancing completely because this reduces communication and simplifies termination detection. In this case, load balancing can be improved by randomly generating more than one subproblem per PE. (For details refer to [29].)

Figure 3 shows speedups for the 15-puzzle instances on 1024 PEs of a Parsytec GCel. In addition to the basic algorithm and the algorithm with fast initialization, it covers an algorithm which uses more localized communication without incurring the overhead of the poll-and-shuffle algorithm. To this end, the fast initialization is modified such that the P initial subproblems are implicitly placed using a pseudorandom permutation (otherwise clusters could build up immediately). During dynamic load balancing, only a subset of PEs "close" to an idle PE is considered when sending a request. The size of this "neighbor set" is doubled after every request sent (in a local unsynchronized way). After all PEs have become reachable, the size of the neighbor set is reset to 4.

While fast initialization alone gains only a small improvement (the larger improvements for tiny problems are difficult to see in this graph), the localized algorithm is noticably faster. Perhaps more noteworthy than this moderate improvement is the fact that many other schemes like using some fixed communication radius, or trying to increase the neighbor set more slowly turned out to be slower than global random polling. Also, the randomized initialization is essential. This indicates that polling "almost" globally is really practically important and not only a requirement imposed by the approach to the analysis. For very large instances, all three variants level at a speedup below 900 because

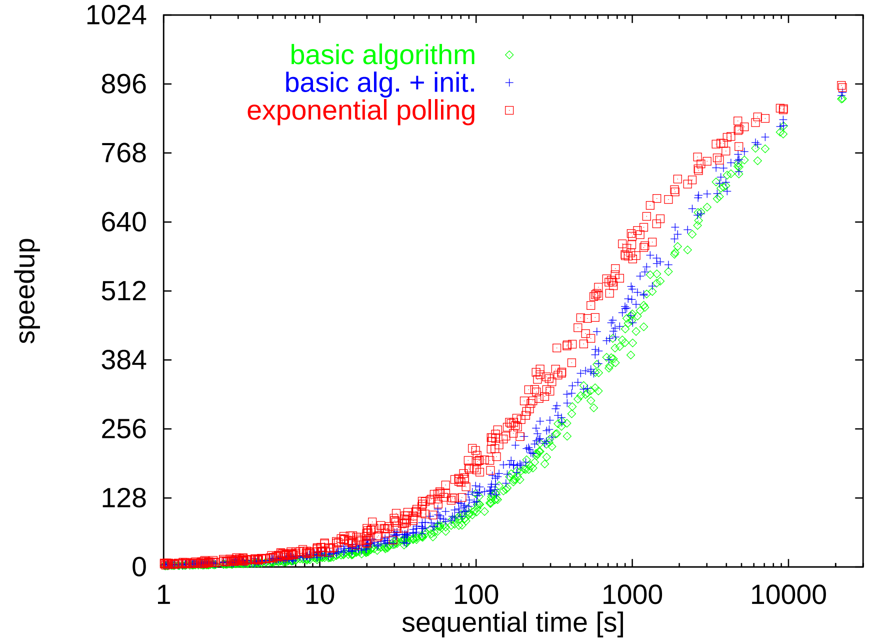

Figure 3: Speedup for 15-puzzle instances on a 1024 PE GCel.

in this measurement Δt was set to get high performance for small instances. With a larger Δt speedups go up to 960.

6.3 Depth First Branch-and-Bound

The *0-1 knapsack problem* is one of the most intensively studied problems in combinatorial optimization [19]. An instance is defined by m items with weight w_i and profit p_i and a knapsack of capacity M. We are looking for $x_i \in \{0, 1\}$ such that $\sum p_i x_i$ is maximized subject to the constraint $\sum w_i x_i \leq M$, i.e., we want to achieve a maximal profit with items in the knapsack without exceeding its capacity.

For large m and arbitrary w_i, the best known algorithms are based on a very fine-grained depth first branch-and-bound search [19]. The branch-and-bound heuristic implies that a new solution found at one PE may affect the size of a subproblem searched elsewhere. Therefore, most instances of the knapsack problem are no tree shaped computations in the strict sense. Nevertheless, random polling works surprisingly well, if it is supplemented by a fast, bottleneck free algorithm for updating bounds (the solution han-

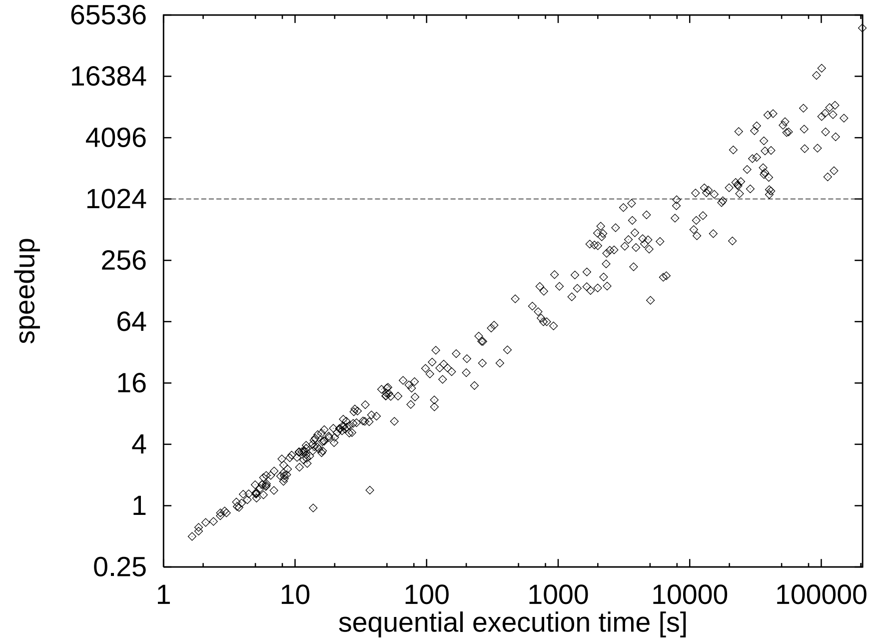

Figure 4: Speedup for 256 instances of the knapsack problem on 1024 PEs.

dling component of PIGSeL implements such a mechanism) and a tree splitting function which takes into account that most subtrees are pruned.

A standard way for generating difficult to solve test instances is to choose w_i uniformly at random from an interval $[w_{\max}, w_{\min}]$ and p_i from a correlated interval $[w_i + p_{\min}, w_i + p_{\max}]$. The double-logarithmic plot in Figure 4 shows the relation between speedup and sequential execution time for 256 random instances with $m = 2000$, $w_i \in [0.01, 1.01]$, $p_i \in [w_i + 0.1, w_i + 0.125]$ and $M = \sum w_i/2$ on 1024 PEs of a Parsytec GCel. The problem parameters were chosen in order to get instances with large m which are tractable but not easy to solve. Other tractable random instances would either contain almost no parallelism or have a small m and are easier to parallelize.

Beginning at per PE loads of about 10 seconds we start to observe good performance. Very large problem instances show a considerable superlinear speedup. For these instances, the sequential algorithm appears to have run into some kind of "dead end". The parallel algorithm is more robust because it follows multiple search paths at once. The overall parallel execution time for 1024 PEs is 1410 times smaller than the sequential time. (Cache effects and the like can be excluded since transputers have no cache and the fast on-chip memory cannot be used for the application code on the system. Also, the parallel version actually searches less nodes.)

7 Conclusions

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.

The asynchronous random polling variant of receiver initiated load balancing parallelizes tree shaped computations provably efficiently, namely with $T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + \mathrm{O}\bigl(T_{\text{atomic}} + h\left(\frac{1}{\epsilon} + T_{\text{rout}} + T_{\text{split}}\right)\bigr)$ with high probability, where h is the maximal splitting depth, T_{rout} the overhead for a message exchange, T_{split} the splitting overhead and T_{atomic} the atomic granularity.

The asynchronous algorithm avoids the waiting costs of synchronized variants and our analysis shows that the unreachability of subproblems in transit is no problem. Message queues may become longer than in the synchronized case but this has no influence on the expected execution time and queue lengths can be efficiently controlled if desired.

The global communication of random polling has its good reasons. Nevertheless, the poll-and-shuffle algorithms replaces this by local communication without increasing the number of communications by more than a constant factor, yielding an asymptotically even faster algorithm.

Although tree shaped computations span a remarkably wide area of applications, an important area for future research is to generalize the analysis to models which cover dependencies between subproblems. The predictable dependencies modelled by fully strict multithreaded computations [3] are one step in this direction. But in many classic search problems the main difficulty are heuristics which prune the search tree in an unpredictable way. Our experiments with the knapsack problem and results like the "Young Brothers Wait Concept" for parallel game tree search [8] indicate that it is often a good strategy to combine random polling load balancing with an application specific splitting function and a protocol for efficiently propagating information which leads to tree pruning.

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] G. S. Bloom and S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, pages 356–368, Santa Fe, 1994.

[4] R. Butenuth, W. Burke, and H.-U. Heiß. COSY – an operating system for highly parallel computers. *ACM Operating Systems Review*, 30(2):81–91, 1996.

[5] 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.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subromonian, 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.

[7] P. Diefenbach and P. Sanders. *PIGSel Manual.* Universität Karlsruhe, LIIN, 1995.

[8] 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.

[9] R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[10] Heun and 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.

[11] 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.

[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] 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.

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

[18] 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.

[19] S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley, 1990.

[20] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[21] J. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

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

[23] V. N. Rao and V. Kumar. Parallel depth first search. *International Journal of Parallel Programming*, 16(6):470–519, 1987.

[24] A. Reinefeld. Scalability of massively parallel depth-first search. In *DIMACS Workshop*, 1994.

[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.

[27] P. Sanders. Better algorithms for parallel backtracking. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 333–347, Lyon, 1995. Springer.

[28] 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.

[29] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. PhD thesis, University of Karlsruhe, 1997.

[30] 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.

[31] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal Parallel and Distributed Processing, Special Issue on Parallel and Distributed Data Structures*, 1998. to appear.

[32] I. C. Wu and H. T. Kung. Communication complexity of parallel divide-and-conquer. In *Foundations of Computer Science*, pages 151–162, 1991.