On the Competitive Analysis of Randomized Static Load Balancing

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

Abstract

Static load balancing is attractive due to its simplicity and low communication costs. We analyze under which circumstances a randomized static load balancer can achieve good balance if the subproblem sizes are unknown and choosen by an adversary. It turns out that this worst case scenario is quite close to a more specialized model for applications related to parallel backtrack search. In both cases, a large number of subproblems has to be generated in order to make good load balance possible. Nevertheless, a carefully implemented randomized static load balancer can sometimes compete with dynamic load balancing on parallel machines with slow communication. The ideas and results derived here can also be used to analyze and improve existing load balancing algorithms.

1 Introduction

One of the key tasks in parallel algorithm design is load balancing, i.e., evenly distributing subproblems over the individual processors (PEs). This is particularly difficult if the subproblem sizes are hard or impossible to predict. A very simple approach is to randomly distribute subproblems to PEs hoping that for large enough numbers of subproblems, imbalance will smooth out.

In fact, in papers like [10] it is proved that it is often sufficient to assign $O(\log n)$ subproblems[1] to each of n PEs in order to smooth out load imbalance on the average. However, the sizes of subproblems are assumed there to be independent identically distributed random variables which also have to fulfill additional properties like "increasing failure rate". Unfortunately, for many parallel applications the subproblem sizes are interrelated in some complicated way because they stem from a global problem instance which has been decomposed into subproblems in order to expose sufficient parallelism.[2] In this case, one would prefer to infer performance guarantees from properties of the problem subdivision procedure and the fact that subproblems are randomly placed.

In Section 2 we start with an abstract model where an adversary is allowed to arbitrarily subdivide a root problem into a given number of subproblems subject

[1] Throughout this paper "log" stands for the logarithm base 2.

[2] There are some applications where modeling loads as independent random variables is very accurate (e.g. for some Monte Carlo simulation [6, 1]).

to a bound on the maximum subproblem size $s_{\max}$. We show that by randomization good load balancing is achieved with high probability if $1/s_{\max} \in \Omega(n \log n)$. We then switch to a more specialized setting and look at a model for parallel tree search based on splitting subproblems into two parts using an inaccurate splitting function. In Section 3 it turns out that the worst case predictions of the abstract model are quite close to lower bounds for this tree splitting model. Then Section 4 shows how randomized static load balancing can be implemented with very little communication overhead. Section 5 summarizes the results and sketches how they can be applied to improve and analyze other load balancing algorithms for parallel tree search. It also reports some preliminary measurement results.

2 An abstract model

We consider a parallel computer with n PEs numbered 1 through n. m subproblems with sizes (sequential execution times) $s_1, \ldots, s_m$ ($s_i \geq 0$) are to be distributed to the PEs. The problem sizes are normalized such that $\sum_{i=1}^{m} s_i = 1$. The sizes are unknown to the load balancer and can be chosen by an adversary subject to the constraint that the maximum subproblem size $\max_{i=1}^{m} s_i$ must not exceed a limit $s_{\max}$. There are no computational dependencies between subproblems and it does not matter where and in which order they are processed. In order to avoid some tedious special case treatments, we assume $m \geq n > 1$, $s_{\max} \leq 1/n$ and $1/s_{\max} \in \mathbb{N}$. We consider the load balancing strategy of placing the subproblems independently and uniformly at random. The adversary does not know the random choices made by the load balancer.

Let the random variable L_j denote the load allocated to PE j, i.e,[3]

$$L_j := \sum_{i=1}^{m} s_i \cdot [\text{subproblem } i \text{ is allocated to PE } j].$$

Let $L_{\max} := \max_{i=1}^{n} L_i$ denote the maximum load. The goal of the analysis is to assess under which circumstances the random placement algorithm achieves good load balancing, i.e., $L_{\max} \leq \frac{1+\epsilon}{n}$ with high probability for some positive constant ϵ.[4] The goal of the adversary is to maximize $\mathbf{E}L_{\max}$.

Lemma 1. *An optimal strategy for the adversary is to assign the size $s_{\max}$ to exactly $1/s_{\max}$ subproblems and the size 0 to all remaining subproblems.*

Proof. Fix any adversary strategy which maximizes $\mathbf{E}L_{\max}$ (there must be such a strategy since the set of adversary strategies is a compact set). If it does not define a subproblem a with $0 < s_a < s_{\max}$ we are done. Else, since $1/s_{\max}$ is an integer, there must also be another subproblem b with $0 < s_b < s_{\max}$. Without loss of generality assume $s_a \geq s_b$. Let $\Omega = \{1, \ldots, n\}^{\{1,\ldots,m\}}$ denote the set of all possible subproblem placement functions, i.e., our probability space. We define the following partition for Ω.

$$\Omega = \Omega_{ab} \,\dot\cup\, \Omega_{a\bar{b}} \,\dot\cup\, \Omega_{\bar{a}b} \,\dot\cup\, \Omega_{\bar{a}\bar{b}} :$$

[3] We adopt the notation from [4] to define $[P] := 1$ if the predicate P is true and $[P] := 0$ else.

[4] We do not consider the minimum load or other characterizations of imbalance here because they do not directly influence the parallel execution time in our model.

Ω_{ab} := The placements which assign both a and b to maximum positions (i.e. positions j_a and j_b with $L_{j_a} = L_{j_b} = L_{\max}$).

$\Omega_{a\bar{b}}$:= The placements which assign a to a maximum position and b to a non-maximum position.

$\Omega_{\bar{a}b}$:= The placements which assign b to a maximum position and a to a non-maximum position.

$\Omega_{\bar{a}\bar{b}}$:= The placements which assign neither a nor b to a maximum position.

The random variable $L_{\max}$ is a mapping from Ω to $\mathbb{R}$ and its expectation can be written as

$$\mathbf{E}L_{\max} = \frac{1}{|\Omega|}\left(\sum_{\omega\in\Omega_{ab}} L_{\max}(\omega) + \sum_{\omega\in\Omega_{a\bar{b}}} L_{\max}(\omega) + \sum_{\omega\in\Omega_{\bar{a}b}} L_{\max}(\omega) + \sum_{\omega\in\Omega_{\bar{a}\bar{b}}} L_{\max}(\omega)\right).$$

If the adversary now changes s_a to $s'_a := s_a + \Delta$ and s_b to $s'_b := s_b - \Delta$ with $\Delta := \min(s_b, s_{\max} - s_a)$, $\mathbf{E}L_{\max}$ changes as follows:

- $\sum_{f\in\Omega_{ab}} L_{\max}(f)$ can only increase since every summand where a and b are assigned to the same maximum position remains the same and when a and b are assigned to different maximum positions, the summand increases by Δ.
- $\sum_{f\in\Omega_{a\bar{b}}} L_{\max}(f)$ increases by $|\Omega_{a\bar{b}}|\Delta$ because every summand increases by Δ.
- $\sum_{f\in\Omega_{\bar{a}b}} L_{\max}(f)$ decreases by at most $|\Omega_{\bar{a}b}|\Delta$.
- $\sum_{f\in\Omega_{\bar{a}\bar{b}}} L_{\max}(f)$ can only increase.

Due to $s_a \geq s_b$ we have $|\Omega_{a\bar{b}}| \geq |\Omega_{\bar{a}b}|$.[5] Therefore, the overall change of $\mathbf{E}L_{\max}$ can only be positive.

By iterating this argument of changing two non-extremal subproblem sizes finitely often we arrive at the adversary strategy described in the lemma because in every step one subproblem size goes to 0 or $s_{\max}$. Therefore the strategy from the lemma must be at least as good as the strategy we started from. □

Theorem 2. *For every $\epsilon > 0$ and every $\beta > 0$ there is a constant c such that $\mathbf{P}\left[L_{\max} > \frac{1+\epsilon}{n}\right] \leq n^{-\beta}$ if $s_{\max} \leq \frac{1}{cn\ln n}$ and if the adversary uses the strategy from Lemma 1.*

Proof. It suffices to show that there is a c such that $\mathbf{P}\left[L_j > \frac{1+\epsilon}{n}\right] \leq n^{-(1+\beta)}$ for any j. Observing that the zero size pieces cannot contribute to $L_{\max}$ we can concentrate on the $1/s_{\max}$ subproblems of size $s_{\max}$. We number these subproblems from 1 to $1/s_{\max}$ and define
Y_{ji} := [the nonzero subproblem number i is assigned to PE j]. Then

$$L_j = s_{\max}\sum_{i=1}^{1/s_{\max}} Y_{ji}.$$

[5] I would like to thank Thomas Worsch for pointing out an elegant way to explicate this: Exchanging the positions of s_a and s_b is an injection from $\Omega_{\bar{a}b}$ to $\Omega_{a\bar{b}}$.

Since the Y_{ji} are independent for fixed j and $\mathbf{P}[Y_{ji} = 1] = 1/n$, a simple Chernoff bound[6] can be used to conclude

$$\mathbf{P}\left[L_i > \frac{1+\epsilon}{n}\right] \leq e^{-\frac{\epsilon^2}{2n s_{\max}}} \leq e^{-\frac{\epsilon^2 c n \ln n}{2n}} = n^{-\frac{\epsilon^2 c}{2}} \leq n^{-(\beta+1)}$$

for $c \geq 2(\beta+1)/\epsilon^2$. □

The mixed use of high probability and expected value results may appear strange, but we felt it to be natural that the adversary tries to make the average performance of the load balancer as bad as possible while a user prefers high probability guarantees for the performance of the load balancer. Yet, there are no technical reasons forcing that. Analogously to the proof of Lemma 1, it can be shown that the probability that $L_{\max}$ exceeds a given limit is maximized by the same adversary-strategy. Vice versa, the performance of the random placement load balancer under the worst case adversary strategy can also be analyzed in terms of $\mathbf{E}L_{\max}$ (e.g. using Theorem 5.6 in [22]).

3 A tree splitting model

In many applications it does not make sense to generate all subproblems in a single step. Often the parallel algorithm is based on the divide and conquer paradigm. In this case, subproblems are generated incrementally by recursively splitting larger problems into a small number of pieces. We can assume without loss of generality that exactly two subproblems are generated by a divide-step. If the splitting function is inaccurate and the sizes of the new subproblems are unknown, our model for randomized static load balancing is appropriate. This setting is quite typical for many backtracking algorithms. (Refer to [20] for a more detailed discussion.)

A simple way to model the splitting error in this application domain is the α-splitting model [11]:[7] The adversary can split a subproblem of size s into two pieces of size s_a and s_b subject to the constraint that $s_a + s_b = s$ and $\min(s_a, s_b) \geq \alpha s$.

Randomized static load balancing can generate 2^h subproblems by building a complete splitting tree for some fixed depth h. This can easily be translated to the abstract model from Section 2 by setting

$$s_{\max} = (1-\alpha)^h \text{ and } m = 2^h.$$

We can now use Theorem 2 to infer an upper bound for a choice of h which achieves good load balance.

Theorem 3. *For every $\epsilon > 0$ and every $\beta > 0$ randomized static load balancing using tree splitting ensures that $\mathbf{P}\left[L_{\max} > \frac{1+\epsilon}{n}\right] \leq n^{-\beta}$ if*

$$h \in x(\alpha)(\log n + \log\log n) + \Omega\left(\frac{\beta}{\epsilon^2}\right) \text{ with } x(\alpha) := \frac{1}{\log\frac{1}{1-\alpha}}.$$

[6] $\mathbf{P}\left[\sum_{i=1}^{n} Z_i > (1+\epsilon)np\right] \leq e^{-\epsilon^2 np/2}$ for independent 0/1-random variables Z_i with success probability p (refer for example to [14]).

[7] We actually use a slight generalization of the original model.

Proof. Solving $(1-\alpha)^h = \frac{1}{cn \ln n}$ for h using the c from the proof of Theorem 2 yields[8]

$$h = \frac{1}{\log \frac{1}{1-\alpha}}(\log n + \log\log n) + \Omega\left(\frac{\beta}{\epsilon^2}\right).$$

So, good load balance is guaranteed for the claimed h even if the adversary were allowed to use its optimal strategy from Lemma 1. □

This result is actually quite tight because we can easily infer a lower bound on h which is only by an $O(\log\log n)$ additive term smaller:

Theorem 4. *No load balancing algorithm which considers 2^h subproblems as atomic that are generated by recursively splitting the root problem, can achieve good load balancing under the α-splitting model if*

$$h < x(\alpha)\log n.$$

Proof. The adversary can always split the problems in the ratio α to $1-\alpha$. After less than $\frac{1}{\log \frac{1}{1-\alpha}} \log n$ splits there will always be one subproblem of size at least $\frac{1}{(1-\alpha)n}$. □

The upper and lower bounds can be interpreted in terms of the number of subproblems $m = 2^h$ to be generated for good load balance. We have $m \geq n^{x(\alpha)}$. So the exponent is heavily dependent on the value of α, i.e., on the quality of the splitting function. Figure 2 shows the behavior of the exponent depending on α. The gap between the upper bound from Theorem 3 and the lower bound from Theorem 4 is a polylogarithmic factor. The variation due to the tolerated imbalance and the degree of high probability is a constant factor.

The α-splitting model can be modified and generalized in various ways. Requiring a guaranteed minimum quality for every split operation as in α-splitting can be quite unrealistic and we do not really need this for the proof of Theorem 3. It is sufficient to specify in some arbitrary way how many splits are necessary to sufficiently reduce the size of the generated subproblems. $O(\log n)$ splits which do not yield any size reduction can be tolerated if this is compensated later.

In [19] the splitting factor is considered to be a random variable with range $[0,1]$ whose density is symmetric around $1/2$. The inaccuracy of the splitting function is modeled by the distribution's standard deviation σ. The lower bounds inferred there are similar to the ones presented here and can be translated by setting $\sigma = 1/2 - \alpha$. The upper bounds only hold on the average and are looser than here.

4 Efficient implementation

So far, we have ignored the overhead incurred by generating and distributing the subproblems. Now we describe how this can be done quite efficiently.

We stick to the tree splitting model from the previous section but the basic ideas are also valid for many other ways of generating subproblems. Only two

[8]The factor needed to convert between log and ln can be shifted into the Ω.

communication steps are performed: Broadcasting the root problem in the beginning, and collecting the results[9] after all subproblems have been exhausted. The PEs independently generate the subproblems they have to process. In order to make this possible, we modify the problem placement process. Instead of placing problems independently at random we use a random permutation $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$ over the subproblems. PE i generates and processes the subproblems $\pi(\frac{m}{n}(i-1)+1), \ldots, \pi(\frac{m}{n}i)$. Figure 1 summarizes this algorithms and shows how subproblems are generated from the root problem by splitting it $h = \log m$ times keeping only the currently relevant piece. For simplicity, n is assumed to be a power of two.[10]

```
(* Input: *)
R : Problem                                   (*  Root problem *)
h : N          (*  (2^h = m, Number of subsequent splits to be performed *)

Determine a random permutation π : {1,...,2^h} → {1,...,2^h}
Broadcast R to each PE
FOR PE i := 1 TO n DOPAR (* asynchronously *)
    FOR j := (i − 1)2^h/n + 1 TO i2^h/n DO
        P := R (* Problem under consideration *)
        let (b_0, ..., b_{h−1}) be the bit representation of π(j) − 1
        FOR l := 0 TO h − 1 DO
            IF b_l = 0 THEN
                P := left part of splitting P into two parts
            ELSE
                P := right part of splitting P into two parts
        search subproblem P sequentially
collect results
```

Figure 1: Generic Algorithm for randomized static load balancing.

Before we can go on to analyze this algorithm, we must first explain the differences to the abstract model from Section 2. First, switching to random permutations does not affect the optimal strategy for the adversary. The proof of Lemma 1 carries over. Furthermore, a random permutation distributes the subproblems more evenly than a random placement strategy and we therefore assume that the upper bound from Theorem 2 still applies.

The next problem is that computing a truly random permutation of such a large size would be very expensive. In particular, it would ruin the communication economy we wanted to gain with our approach. But for most practical cases it is sufficient to compute some pseudorandom permutation. One possibility is to use a linear congruence generator of the form $x_{n+1} = ax_n + c \bmod m$. According to Knuth [9] this yields reasonable pseudorandom number generators with full period length for appropriate choices of a and c. Unfortunately, little is

[9]This is often possible by applying an associative commutative operation to the results of the subproblems (like "number of solutions" or "best solution"). In this case, collecting the results is very simple and efficient because it can be done locally first followed by a single global reduction operation.

[10]By introducing a slight additional imbalance, this constraint can be lifted.

known about using the full sequence as a permutation. We are not even aware of well established empirical tests for assessing the randomness of a permutation. Since $m = 2^h$ is a power of two, pseudorandom permutations can also be computed using the theory of finite fields. Let p be a primitive polynomial modulo 2 of degree h. Then the polynomial $x^l \bmod p$ (l relatively prime to $m-1$) is a generating element of the multiplicative group of $GF(2^h)$, i.e., the sequence $((x^l)^1, \ldots, (x^l)^{m-1})$ enumerates the nonzero polynomials modulo p.[11] Inserting the 0-polynomial at some random place in this sequence yields the desired pseudorandom permutation. By interpreting the coefficients of the polynomials as bits of a computer word, polynomial arithmetic mod p is quite efficient. Both classes of permutations can be generated quickly and without communication using an integer-power computation for each PE's first evaluation of π and a multiplication for all subsequent evaluations.

4.1 Analysis

Let d denote the network diameter, l the length of the description of the root problem, T_{seq} the sequential execution time (up to know this was 1, i.e. our unit of time) and T_{split} the time for splitting a subproblem. Let us further assume that collecting results is asymptotically as fast as broadcasting the root problem. Let m (or equivalently h) be chosen in such a way that good load balance is achieved, i.e., $L_{\max} \leq (1+\epsilon)T_{\text{seq}}$. The pseudorandom permutations described above can be computed so quickly that they do not contribute to the asymptotic execution time since they need to be evaluated only every h splits. Then the parallel execution time can be estimated as the subproblem solving time on the most highly loaded PE plus the time for broadcasting and collecting results plus the time for doing $\frac{h2^h}{n}$ splits, i.e.,

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}(d+l) + \frac{h2^h}{n}T_{\text{split}}.$$

Theorem 3 allows us to express h in terms of n and the splitting factor α. Let $x(\alpha) := \frac{1}{\log \frac{1}{1-\alpha}}$ then

$$T_{\text{par}} \in (1+\epsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}(d+l) + T_{\text{split}}\mathrm{O}\left(n^{x(\alpha)-1}(\log n)^{x(\alpha)+1}\right).$$

The exponent $x(\alpha)$ is depicted in Figure 2.

Ignoring polylogarithmic factors, and assuming that l and T_{split} grow slowly when T_{seq} is scaled, we can achieve good net efficiency if the root problem size is scaled as a polynomial function of n with an exponent which is heavily dependent on the quality of the splitting function. For α close to 1/2, the parallelization overhead can be dominated by the communication expense. For example for $\alpha > 0.37...$ ($\alpha > 0.29...$) communication costs asymptotically dominate splitting costs on a mesh-connected (ring-connected) machine. In these cases, randomized static load balancing outperforms the best known dynamic load balancing algorithm because these involve an $\Omega(dl)$ term for the communication expense [20]. (But note that these algorithms work efficiently for any α and even for splitting functions which cannot be characterized by any $\alpha > 0$.)

[11] At least from the point of view of cryptographers this method seems to yield quite random permutations since inverting them is related to the discrete logarithm problem. (I would like to thank Thorsten Minkwitz and Jörn Müller-Quade for pointing this out.)

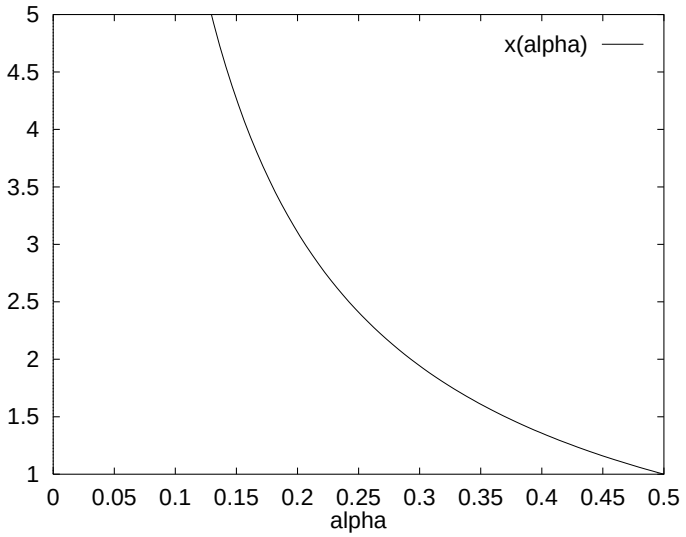

Figure 2: Development of the exponent $x(\alpha)$.

5 Conclusion

Even if subproblem sizes are unknown and determined by an adversary, randomizes static load balancing can achieve good load balancing with high probability provided there is a bound on the subproblem size such that a logarithmic number of subproblems cannot overload a processor. This is a significant generalization compared to previous results which show that good load balance can be achieved on the average if the PE loads are sums of logarithmically many independent and "well-behaved" random variables. If the bound on the subproblem size is large compared to the average subproblem size this result is "bad news" because the handling of large numbers of tiny subproblems severely limits efficiency.

For divide-and-conquer applications where subproblems are generated by an inaccurate splitting function, this situation of very unevenly distributed subproblem sizes occurs quite naturally. On the other hand, randomized static load balancing can be implemented using only a single broadcast of the root problem and a locally computed pseudorandom permutation. If communication is slow and the splitting error is not too large, this approach of trading computation for communication is superior to otherwise more robust dynamic load balancing algorithms.

5.1 Applications to existing tree search algorithms

The approach to use static tree decomposition for parallelizing tree search is quite old [2]. Assigning a single subproblem to each PE achieves only very poor balancing but it can be helpful as an initialization scheme for dynamic load balancing [13, 12, 17, 7]. If the (pseudo)random permutations described in Section 4 are used, the initialization can actually lead to an asymptotic improvement: In any subnetwork of sufficient size the average load will be about the same. So, global communication (which is for example important for the basic random polling algorithm [17]), is no more necessary. It can be shown that on an r-dimensional mesh the communication distance can be reduced by a logarithmic factor for any splitting factor $\alpha > 0$.

In [16] a combined static/dynamic load balancing algorithm is described which starts by allocating several subproblems to each PE. Later, idle PEs get work from nearby busy PEs. In this phase, subproblems are never split in order to avoid the negative effects of splitting in the context of neighborhood communication described in [15]. The author reports problems due to uneven sizes of the subproblems. This can be explained quite naturally by our splitting model – the lower bound from Theorem 4 also applies to this algorithm.

A somewhat different model of parallel tree search underlies the idea to search for a single solution in a tree with unevenly distributed solution nodes using randomized successor ordering [8, 3]. In some cases, this can yield large superlinear average speedups compared to a sequential depth first search. But if there is no solution at all, the speedup is smaller than one. Our approach offers a way to improve this situation. Even if subproblems are decomposed and searched randomly, they are always guaranteed to remain disjoint. If there is no solution, the efficiency analysis from Section 4.1 predicts good speedups for large problems.

5.2 Current work

Randomized static load balancing has been implemented as one of the load balancers in the parallel tree search library PIGSeL [18]. However, it cannot currently compete with the "Random Polling" dynamic load balancing algorithm [17] which achieves almost perfect speedup on a 1024-PE mesh-connected transputer system for quite small problem instances of irregular tree search problems like the knapsack problem, the 15-Puzzle and search for "Golomb rulers" [21].[12] This may change with the development of better and faster splitting functions.

On the theoretical side the next step is to reconcile the abstract model with the implementation by finding a "universal"[13] set of permutations which are easy to compute and which can be proved to perform similarly well as a truly random placement.

References

[1] C. Casari, C. Catelli, M. Guanziroli, M. Mazzeo, G. Meola, S. Punzi, A. Scheinine, and P. Stofella. Status report on ESPRIT project p9519 palace: Parallelization of GEANT. In *International Conference Massively Parallel Processing Applications and Development*, Delft, 1994. Elsevier.

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

[3] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.* PhD thesis, TU München, 1992.

[4] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison Wesley, 1992.

[5] R. Gupta, S. A. Smolka, and S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, 1994.

[12] A note to the referees: If desired I can expand on some of these measurements.

[13] In the sense which is used for the term *universal hashing* [5].

[6] P. Heidelberger. Discrete event simulations and parallel processing: Statistical properties. *SIAM Journal of Statistical Computation*, 9(6):1114–1132, 1988.

[7] D. Henrich. *Lastverteilung für Branch-and-bound auf eng-gekoppelten Parallelrechnern.* PhD thesis, Universität Karlsruhe, 1994.

[8] V. K. Janakiram, E. F. Gehringer, D. P. Agrawal, and R. Mehotra. A randomized parallel branch-and-bound algorithm. *International Journal of Parallel Programming*, 17(3):277–301, 1988.

[9] D. E. Knuth. *The Art of Computer Programming — Seminumerical Algorithms*, volume 2. Addison Wesley, 2nd edition, 1981.

[10] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Computers*, 11(10):1001–1016, 1985.

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

[12] R. Lüling and B. Monien. Load balancing for distributed branch & bound algorithms. In *Int. Parallel Processing Symposium (IPPS)*, 1992.

[13] R. P. Ma, F. S. Tsung, and M. H. Ma. A dynamic load balancer for a parallel branch and bound algorithm. In *3rd Conference on Hypercubes, Concurrent Computers, and Applications*, pages 1505–1530, Pasadena, 1988. ACM.

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

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

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

[17] 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.

[18] P. Sanders. Portable parallele Baumsuchverfahren: Entwurf einer effizienten Bibliothek. In *Transputer Anwender Treffen*, pages 168–177, Aachen, 1994. IOS Press.

[19] P. Sanders. Randomized static load balancing for tree shaped computations. In *Workshop on Parallel Processing*, TR Universität Clausthal, pages 58–69, Lessach, Österreich, 1994. ISSN 0943-3805.

[20] 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.

[21] P. Sanders. Some implementation results on random polling dynamic load balancing. Technical Report 40/95, Universität Karlsruhe, August 1995.

[22] J. S. Vitter and P. Flajolet. Average case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. Elsevier, 1990.