Randomized Load Balancing for Tree-structured Computation

Soumen Chakrabarti* Abhiram Ranade† Katherine Yelick*
Computer Science Division, University of California, Berkeley CA 94720
{soumen,ranade,yelick}@cs.berkeley.edu

Abstract

In this paper, we study the performance of a randomized algorithm for balancing load across a multiprocessor executing a dynamic irregular task tree. Specifically, we show that the time taken to explore a task tree is likely to be within a small constant factor of an inherent lower bound for the tree instance. Our model permits arbitrary task times and overlap between computation and load balance, and thus extends earlier work which assumed fixed cost tasks and used a bulk synchronous style in which the system alternated between distinct computing and load balancing steps. Our analysis is supported by experiments with application codes, demonstrating that the efficiency is high enough to make this method practical.

1 Introduction

In this paper we study a popular randomized strategy for load balancing dynamic tree-structured task graphs on large scale message passing multiprocessors. First, we show analytically that with high probability, the randomized strategy results in parallel running time that is within a constant factor of an inherent lower bound for the task graph. Second, using task graphs generated from application codes, we estimate the constant factor. The randomized strategy we consider has already been used in applications like a symbolic polynomial equation solver and an eigenvalue program and has been found to give good speedups [2, 4]. These results represent a step towards bridging the gap between the theory and practice of dynamic load balancing algorithms.

Tree structured task graphs arise frequently in paradigms such as divide and conquer, backtrack search, and heuristic search using branch and bound. A sequential algorithm maintains a pool of generated but unexpanded tasks, and repeatedly selects the most promising task (based on some measure) and solves it, possibly generating more tasks that are added to the pool. For the parallel case, the following strategy has been proposed and analyzed [5, 8]: every processor maintains its own local pool, and when a processor needs work, it selects the best task from its local pool for execution. If any new tasks are generated during the execution of one task, they are sent to the pools of randomly chosen processors in the machine. Karp and Zhang show that under reasonable assumptions, independent of the problem instance, the scheme gives linear speedup to within a constant factor, with high probability. However, their analysis assumes unit time tasks. For many applications, the wide variation of task times limits the applicability of their model.

We extend Karp and Zhang's strategy to deal with non-uniform task times. Neither the resulting strategy nor its analysis require prior knowledge of the distribution of the task times. Further, as with the analysis of Karp and Zhang, no knowledge of the structure of the task graph is required (except that it is a tree). Under reasonable assumptions, we can still establish that the speedup is linear with high probability, where the probability space is over the random choices made by the load balancing algorithm and not the structure of the tree instance or task times.

These results are supported experimentally. We present results from simulations of the machine model and algorithm, using execution traces from two applications: a symbolic algebra program and an eigenvalue finder. For comparison, we also provide actual speedup on the CM-5 multiprocessor. The performance results show that the model is faithful in the applications we considered, and the efficiency of parallel execution is high.

2 Related Work

Classical scheduling theory deals with exact or bounded approximate schedules for task graphs with their execution times and dependencies known in advance. For irregular applications, task time and dependency information, represented by the task graph structure, are usually not available at compile time, so decisions are made dynamically. Two possible approaches to dynamic load balancing are *sampling* and *oblivious* algorithms. A sampling algorithm monitors task statistics to use in scheduling decisions. An oblivious algorithm does scheduling without such statistics.

Lucco and Polychronopoulos, among others, have addressed the problem of dynamically scheduling `forall` loops in which the execution time of the loop iterations are unpredictable [6, 7]. However, there are no dependencies between tasks in the analysis, and the task pool only shrinks after initialization; there is no dynamic task creation. Lucco's sampling algorithm, `taper`, is a probabilistic version of guided self-scheduling proposed by Polychronopoulos [7]. `Taper`

*Supported in part by ARPA under contract DABT63-92-C-0026, by NSF (numbers CCR-9210260 and CDA-8722788), and by Lawrence Livermore National Laboratory. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.
†Supported by NSF-DARPA grant # CCR-9005448.

In the 1994 IEEE Scalable High Performance Computing Conference, Knoxville, TN.

collects tasks into clusters, large at first to reduce overhead, small later for even finishing time. However, dependencies and data locality are two important aspects that are not addressed. In addition, the analysis of `taper` assumes the variance of task times is known; in the implementation the true variance is approximated by sampling and using the variance of completed tasks.

The strategy we analyze is oblivious and gives optimal (to within constants) schedules in the presence of irregular task times *and* dependencies for an important computation structure: tree-structured task graphs. Decentralized scheduling is crucial for distributed memory multiprocessors, but is also essential for scalable shared memory programs to avoid memory contention.

While our model permits dependencies and irregularity, it does not capture the effect of data locality. Our results will still be applicable, however, if the smallest task time is comparable to or greater than network communication time for a task. Our experiments suggest that some applications like Gröbner bases computation [2], and the eigenvalue problem [4] fit this paradigm, either because the volume of data associated with a task is small, or because caching and data replication are feasible. For the applications we consider, transmitting the task takes about 4–6μs, whereas the smallest task takes thousands of microseconds to execute. For such applications, our analysis makes authentic performance predictions.

3 Computation Model

In this section we describe the model of tree-structured computation. We consider both exhaustive search trees, in which all nodes are always expanded, and branch and bound trees, in which pruning of the tree may occur dynamically.

3.1 Tree-structured Task Graph

A tree structured computation begins with a single root task. When a task is executed, its children tasks are generated. The entire tree will be denoted by H. The time required to execute a task v will be denoted as $t(v)$, and can only be known after executing v. For simplicity, suppose all children are generated together immediately after the execution is completed. We assume time is measured in discrete steps, and for all tasks v, $t(v) \in \{1, \cdots, T\}$, where time is normalized such that $\max_{v \in H}\{t(v)\}/\min_{v \in H}\{t(v)\} = T$. The goal is to generate and execute every task that is a descendant of the root task. The process of executing a task and generating its children will be called "node expansion". For the purpose of analysis, it is assumed that even though task times and tree shape are unknown to the load balancing algorithm, they are independent of the scheduling decisions.

3.2 Exhaustive Search

Many parallel algorithms enumerate solutions from some universe by dividing the universe into successively refined partitions. This method naturally give rise to tree-structured task graphs. The sequential paradigm for such problems is backtrack search. Nodes in the task graph are partial configurations, leaves are solution configurations or "dead ends", and the goal is to expand the tree so as to enumerate all solution leaves.

3.3 Branch and Bound Graph

A branch and bound graph [5] is a tree structured graph in which each task v has an associated cost $c(v)$, with the requirements that $c(v) > c(\textsc{Parent}(v))$ and all costs are distinct. The goal is not necessarily to generate and execute all tasks in H, but only to identify the leaf node with minimum cost c^*.

3.4 Sequential Algorithm

A common strategy used by sequential algorithms is the "best-first" strategy. At any point during the execution, all unexpanded nodes are maintained in a priority queue. While the queue is non-empty, the node with least $c(v)$ is removed and expanded. It is easy to see that only the nodes in a subtree $\widetilde{H}$ are expanded, where $\widetilde{H} = \{v \in H : c(v) < c^*\}$. Let $|\widetilde{H}| = n$ and h be the height of $\widetilde{H}$. The total work in $\widetilde{H}$ is defined as $W = \sum_{v \in \widetilde{H}} t(v)$. We also define $S = \max_{\sigma \in \widetilde{H}} \left\{\sum_{v \in \sigma} t(v)\right\}$, which is the maximum work on any root-leaf path σ in $\widetilde{H}$. These definitions apply to the special case of an exhaustive search tree, where $\widetilde{H} = H$. We assume that operations on a priority queue for task selection take negligible time compared to the expansion time. In this model, the sequential algorithm takes time W.

3.5 Parallel Algorithm with Distributed Queue

Our machine model consists of processors with individual local memory connected by a communication network. Processors communicate by passing messages to each other. Communicating one task takes unit time. Similar to Karp and Zhang, we put a local priority queue of tasks in the memory of each processor; thus, priority is preserved within each local queue but not across processors. An idle processor tries to dequeue a task from its local queue. If one exists, it is expanded. Any child task is enqueued into the priority queue of a uniformly randomly chosen processor. There is no coordinated global communication for load balancing purposes. Once a processor obtains a task from its local pool and starts working at it, the task is run to completion. This assumption is important for the grain of tasks we have in mind, where pre-emptive scheduling may be expensive because of state saving and restoring costs.

Termination. In a parallel branch and bound program, each processor has to periodically propagate the cost of the least cost leaf it has expanded, so that all processors know the cost of the global best cost leaf in order to use it for pruning. Also, barrier synchronizations are required to detect situations where all local queues are empty so that the processors can terminate. We note that these can be done infrequently with low overhead, so they do not affect the time bounds we derive.

3.6 Parallel Algorithm with Ideal Central Queue

To interpret the bound on randomized running time, we shall compare it with an idealized model in which all tasks are managed at a centralized site, so that global ordering on the priority value $c(v)$ of nodes v is ensured. An idle processor attempts to get a task from the manager. If a task is found, it is expanded and any child generated is enqueued into the central queue. We make the ideal case assumption that the task manager has high bandwidth — it can serve all P processors in the same time step if they all happened to request for work. This can only make the comparison with the randomized model more conservative.

3.7 High Probability

Analysis of the distributed algorithm is probabilistic, with the notion of *high probability* defined as follows. For a task tree with n nodes, the statement

$$t = O\big(g(P, W, n, T, S, h)\big) \quad \text{w.h.p.}^1 \tag{1}$$

means that given an arbitrary real number $\alpha > 0$, there is some constant β that possibly depends on α but not the characteristics W, n, T, S, and h of the tree instance, such that

$$\Pr\left[\, t > \beta\, g(P, W, n, T, S, h)\,\right] < n^{-\alpha}. \tag{2}$$

The probability space in the above equation is over the random choices of processors to send freshly generated tasks, and not the space of possible inputs. Thus our results hold over all possible input task trees, and no statistical properties are assumed of the input. Contrast this with sampling algorithms like Lucco's `taper`, whose efficiency depends on approximate knowledge of the variance of task times.

[1] With high probability.

4 Analysis

In this section we present the analytical results. First we make some observations about the centralized queue model described in §3.6. We get a tight bound on the running time, which sets our goal for the analysis of the randomized model described in §3.5.

4.1 Centralized Priority Queue

For exhaustive search problems with $H = \widetilde{H}$, the running time is $\Theta(W/P + S)$, analogous to the $\Theta(n/P + h)$ bound for unit task times [5]. We omit the proof of this fact, and consider the more interesting case of branch and bound trees.

Lemma 4.1 With an ideal shared priority queue, the execution time t is given by $t = \Theta\left(\frac{W}{P} + hT\right)$.

Proof. First consider the lower bound. Since a total work W has to be performed by P processors, at least W/P time is needed. The example in figure 1 shows that $\Omega(hT)$ time can be necessary. The boundary between H and $\widetilde{H}$ is shown by the broken curve in the figure. In step 1, one processor expands the root, generating P children that all P processors start expanding at time 2. $P-1$ of the nodes v are in $H - \widetilde{H}$ with $t(v) = T$, meant to keep $P-1$ processors busy for time T, so the last processor is left alone to expand part of the useful tree $\widetilde{H}$ as shown. Node r has P children and node x has $T-1$, so that when the new set of P nodes are generated, the $P-1$ processors just freed grab the new decoys. This can be arbitrarily repeated, so we effectively lose all but one processor, which has to do all the useful work.

To demonstrate the upper bound, pick any root-leaf path σ in $\widetilde{H}$ and consider the expansions of nodes on σ, which must happen in sequential order. Let u be any node in σ. Since $u \in \widetilde{H}$, u must be enqueued into the central queue (at time $g(u)$, say) and expanded at some later time $d(u)$. Let $N(u) \stackrel{\text{def}}{=} d(u) - g(u)$ be the time for which u has been "neglected". Expansion of u finishes at time $d(u) + t(u)$. Define the path time

$$t(\sigma) \quad = \quad \sum_{u \in \sigma} \Big(N(u) + t(u) \Big). \tag{3}$$

Then it is clear that $t = \max_\sigma t(\sigma)$, i.e., the total execution time is the maximum path time. We shall now bound $t(\sigma)$ for any σ. Observe that after time $g(u) + T$, every processor has had an opportunity to dequeue u, so if they pick nodes other than u, each of those nodes v must have $c(v) < c(u)$, and since $u \in \widetilde{H}$, $v \in \widetilde{H}$, too. Thus all processors must be doing useful work between $g(u) + T$ and $d(u)$ (the shaded region in figure 1). Since the length of the shaded interval is at

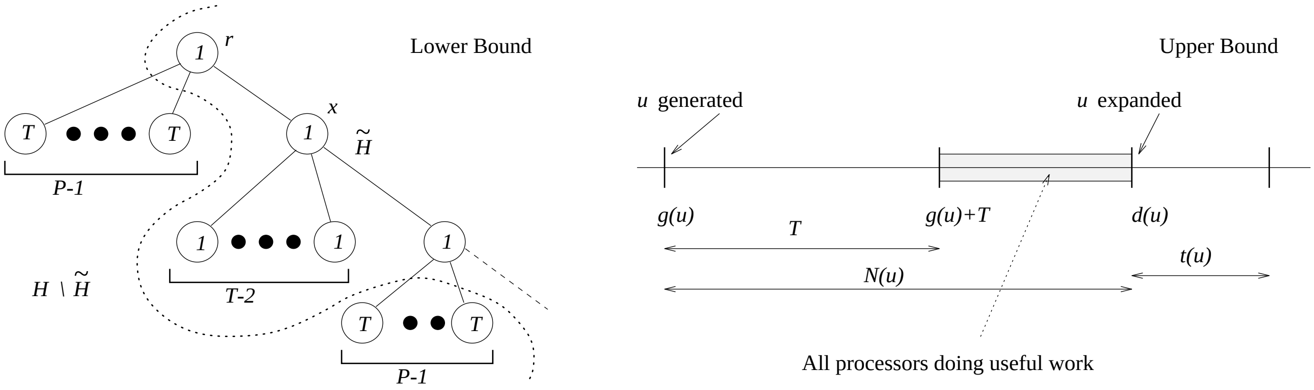

Figure 1: Illustrations for lower and upper bounds in lemma 4.1.

least $N(u) - T$ and the total useful work is W,

$$\begin{aligned}
\sum_{u \in \sigma} \Big(N(u) - T\Big) P &\leq W \\
\Rightarrow \quad \sum_{u \in \sigma} N(u) - hT &\leq \sum_{u \in \sigma} \Big(N(u) - T\Big) \leq \frac{W}{P} \\
\Rightarrow \quad t(\sigma) &\leq \frac{W}{P} + hT + \sum_{u \in \sigma} t(u) \\
&\leq \frac{W}{P} + hT + S.
\end{aligned}$$

Since $S = O(hT)$, the result follows. ∎

4.2 Distributed Priority Queue

As noted earlier, a centralized queue is unrealistic. But it provides an upper limit on the best possible performance. We show here that the distributed priority queue described earlier in fact very closely matches the upper limit.

4.2.1 The Delay Sequence Argument

The first step in the analysis (lemma 4.2) is to show that if the execution of some node s finishes at time t, then any delays in reaching s were caused by a set of nodes V that, in some sense, interfered with the execution. Further, the total execution time of the set V is greater than $t - hT - S$, so if t is very large, the nodes in V account for most of this time.

The rest of the analysis deals with estimating the likelihood that any set V of nodes interferes (in the above sense) with the execution of any node s. This is formalized using the notion of a *delay sequence*, adapted from [8]. We show that the probability of having a long execution, in which the computation is significantly delayed by interference from V, is small.

Lemma 4.2 Suppose the expansion of some node $s \in \widetilde{H}$ finishes at time t. Let $s_1, \ldots, s_{h(s)}$ denote the nodes on path from the root to s, with $s_1 = \text{root}$ and $s_{h(s)} = s$. Further, let s_j have been dequeued from queue q_j and executed by processor q_j. Then there exists a set of vertices $V \subseteq \widetilde{H}$ and a partition $\Pi_1, \ldots, \Pi_{h(s)}$ of the interval $[1, T]$ such that

- $V \cap \{s_1, \ldots, s_{h(s)}\} = \emptyset$.
- Each $v \in V$ is generated during the time interval $[1, t]$. Further, each node in V arrives into q_j during interval Π_j, for some j, $1 \leq j \leq h(s)$.
- $\sum_{v \in V} t(v) \geq t - hT - S$.

Proof. We only sketch the proof, the details being similar to a similar proof in [8]. The proof is constructive. The idea is to work backwards from the time t and consider the earliest time instant $t' < t$ such that during the interval $[t', t]$ only nodes in $\widetilde{H}$ are expanded in $q_{h(s)}$. We include all these nodes into V, and set $\Pi_{h(s)} = [t', t]$. We then continue the construction from $t' - 1$ in an analogous manner. Reasoning as in lemma 4.1, we can see that the processing times of nodes in V must cover all of $[1, t]$, except for the time spent in processing nodes $s_1, \ldots, s_{h(s)}$ (which is at most S) and the time spent in processing nodes outside $\widetilde{H}$ (which is at most hT). ∎

Definition 4.1 (Delay Sequence) A delay sequence (s, Q, Π, V) of length t, t being a positive integer, consists of the following components.

- A node $s \in \widetilde{H}$, at depth $h(s)$ in H.
- A sequence $Q = (q_1, \ldots, q_{h(s)})$ of task queues, identified by the processor on which the queue is located.
- A sequence of individually contiguous but mutually disjoint intervals $(\Pi_1, \ldots, \Pi_{h(s)})$ that partition the interval $[1, t]$.

- A set of nodes $V \subseteq \widetilde{H}$.

$\boxtimes$

Definition 4.2 A delay sequence (s, Q, Π, V) of length t *occurs* in an execution if

- The tree expansion takes time at least t.
- $s \in \widetilde{H}$ is a node whose expansion finished no earlier than t (there has to be such a node, or time t would not have been necessary). Suppose s is at depth $h(s)$, and, starting at the root of H, the root-to-s path is $(s_1, \ldots, s_j, \ldots, s_{h(s)})$.
- s_j was enqueued into queue q_j and expanded by processor q_j.
- A set $V \subseteq \widetilde{H}$ can be identified as in lemma 4.2.

$\boxtimes$

From lemma 4.2, we can conclude that whenever execution takes time t or longer, some delay sequence of length t must have occurred.

The probability of large delays can thus be estimated by estimating the probability of occurrence of delay sequences. This is done by counting all possible delay sequences, and in turn estimating the probability of each.

Lemma 4.3 The probability of occurrence of a delay sequence (s, Q, Π, V) of length t is $\dfrac{1}{P^{h(s)+|V|}}$.

PROOF. We know that each node in V and $\{s_1, \ldots, s_{h(s)}\}$ (note that they are disjoint) was expanded during some Π_j, and further that it was enqueued into the specified q_j. However, there were P choices for each node, and thus the probability of the specified choice was $1/P$, giving the result. ∎

To count the number of delay sequences, we need to consider the number of different ways of choosing the parameters s, Q, Π and V in a consistent manner. The hardest to count is the number of choices for V. The vertices in V must be selected so that their execution times sum to at least $t - hT - S$. This part of our argument is more complicated than the one of [8] because the expansion time of a node can have arbitrary values in the range $[1, T]$.

We first describe the counting argument assuming that $t(v)$ can only take on one of the $\Theta(\lg T)$ values in $\{1, 2, 4, \ldots, 2^{\lceil \lg T \rceil}\}$. Later, in §4.2.2 we will show how this assumption can be discarded.

Lemma 4.4 For fixed s and V, the number of possible delay sequences (s, Q, Π, V) of length $t + T \lg T$ is at most $P^{h(s)} 2^{t/T}$.

PROOF. [Sketch] Given s, the processors which expanded s_j can each be chosen in P ways for $1 \leq j \leq h(s)$, and the partition Π need be specified only to a granularity of T, which can be done in $\binom{t/T + h(s)}{h(s)} \leq 2^{t/T}$ ways. Multiplying these gives the result. ∎

Lemma 4.5 The probability that some delay sequence of length at least $t + T \lg T$ occurs is at most

$$n\, 4^{t/T} \left[\frac{eW/P}{t - S - hT} \right]^{\frac{t - S - hT}{W/n}} \tag{4}$$

PROOF. [Sketch] From the previous lemmas, the probability of occurrence of a length t delay sequence for fixed s and V is at most $P^{-(h(s)+|V|)} \times P^{h(s)} 2^{t/T} = P^{-|V|} \times 2^{t/T}$.

The probability of an arbitrary delay sequence of length t occurring is at most $\sum_{s,V} P^{-|V|} 2^{t/T}$.

We choose V as follows. We first choose the amount of time t_i spent on nodes of color i in V. We choose t_i to a granularity of T, and hence this can be done in at most $\binom{t/T + \lg T}{\lg T} \leq 2^{t/T}$. Fixing t_i fixes the number of nodes of color i in V to be $t_i/2^i$. Since there are n_i nodes of color i overall, the number of choices for V consistent with the choices for t_i is at most $\prod_i \binom{n_i}{t_i/2^i}$.

The number of choices for s is at most n. Using convexity arguments and the constraint that $\sum_i t_i \geq t - S - hT$ we can upper bound the probability of occurrence of an arbitrary delay sequence of length t by the required expression. ∎

Using lemma 4.5, we can obtain our main result. We omit the proof.

Theorem 4.1 (Main Result) For any given $\alpha > 0$, there exists a β depending only on α (and not the input) such that

$$\Pr\left[t > \beta \left(\frac{W}{P} + hT \right) \right] < n^{-\alpha}, \tag{5}$$

provided $P = O(n/\lg n)$ and $P = O(n/\log T)$.

In other words, under the stated conditions,

$$t = O\left(\frac{W}{P} + hT \right) \quad \text{w.h.p.} \tag{6}$$

4.2.2 Allowing Arbitrary Task Durations

The proof above assumed that task durations could only assume values from the set $\{1, 2, 4, \ldots, 2^{\lceil \lg T \rceil}\}$. Here we indicate how to discard this requirement.

The idea is to round up the actual duration $t(v)$ of node v to the next higher power of 2 for the purpose of the analysis. This can at most increase all times by a factor of two. As a result only the constant factors in the analysis will be affected.

We note that instead of rounding up to powers of 2, we could use any other base $b > 1$. This might be useful for getting the best estimates of the constants.

4.2.3 Exhaustive Search

For exhaustive search problems, a slightly different analysis gives

$$t = O\left(\frac{W}{P} + S\right) \quad \text{w.h.p.,} \tag{7}$$

subject to the same conditions as above.

5 Simulation and Experience

This section reports on the performance evaluation of the randomized load balancing algorithm in practice. For applications with tasks that take much more time expanding than transmitting, actual performance corroborates the above results that predict that speedups are scalable, to a small constant factor.

5.1 The Applications

There are applications for which the theory developed makes reasonable performance predictions. The Gröbner basis problem [2] and the bisection eigen value problem [4] are two examples. They can be abstracted to tree-structured computations, in which locality has little influence on performance because replication is used.

5.1.1 The Bisection Eigenvalue Algorithm

A symmetric tridiagonal $N \times N$ real matrix is known to have N real eigenvalues. It is easy to find an initial range on the real line containing all eigenvalues, and, given a real number ρ, it is possible to calculate in $O(N)$ time how many of the N eigenvalues are less than ρ. This primitive can be used to successively subdivide the real line and locate all eigenvalues to arbitrary precision [9]. The kernel is shown below. Parameters `lVal` and `rVal` represent the current endpoints of the current range and `Cnt` is the number of eigenvalues in that range.

```
Eigen ( lVal, rVal, Cnt ) {
  If Cnt = 0 Return;
  Shrink [lVal,rVal] while Cnt remains unchanged;
  If [lVal,rVal] is small
    Return, reporting convergence;
  Else {
    Split [lVal,rVal] at some mVal:
                  lVal < mVal < rVal;
    Find number mCnt of eigenvalues
                  between lVal and mVal;
    Eigen ( mVal, rVal, Cnt - mCnt );
    Eigen ( lVal, mVal, mCnt );
  }
}
```

The two recursive calls to **Eigen** generate a binary tree as the task graph. The computation in each task consists of the *Shrink* and *Split* operations, each of which is $O(N)$ flops for an $N \times N$ tridiagonal input. But *Shrink* might be repeated an unknown number of times depending on convergence. This makes task times irregular. The input array, being read-only and sparse, can be replicated across all processors. Thus locality is not an issue. It is common to choose `mVal` = (`lVal` + `rVal`)/2. This does not necessarily balance the number of eigenvalues on either side, so the tree can potentially be quite unbalanced. The first histogram in Figure 2 shows the distribution of tasks times.

5.1.2 The Gröbner Basis Algorithm

Buchberger's Gröbner basis algorithm [1] starts with a set G of multivariate symbolic polynomials and repeatedly checks if there is any pair $f, g \in G$ such that a certain scaled sum of f and g is not represented in G. If there is, G is augmented with a new polynomial to make it "more complete." Thus the computation kernel has the following structure.

```
G = UserInput;
P = {{f,g} : f,g ∈ G};
While P ≠ ∅ do {
    Remove some {f,g} from P;
    r = Combine(f,g);
    If r not represented in G {
        Add new pairs {{r,h} : h ∈ G} to P;
        Add r to G;
    }
}
```

By suitably transforming the program, we can express the parallel program as a tree-structured computation. Each task descriptor is a pair $\{f, g\}$. Solving the task involves executing the body of the while-loop above, which can take extremely diverse task times (i.e., T can be hundreds to thousands in practice). Even though the augmentation of G looks inherently sequential, the tasks in the tree can actually run with significant concurrency [2]. G is replicated across processors, and maintaining consistency is inexpensive since augmentations are rare in practice. Thus, locality is not a primary factor in performance.

5.2 Measurements

Randomly placing tasks give good empirical performance for many applications. However, it is difficult to isolate the effects of the load balancing strategy from several other factors (e.g. locality) that affect overall performance. By simulating execution traces we overcome this problem. For each application, we added instrumentation to the sequential program to emit the task tree with task times, and input this tree to a simulator that simulated the parallel execution of the randomized load balancing algorithm. We also

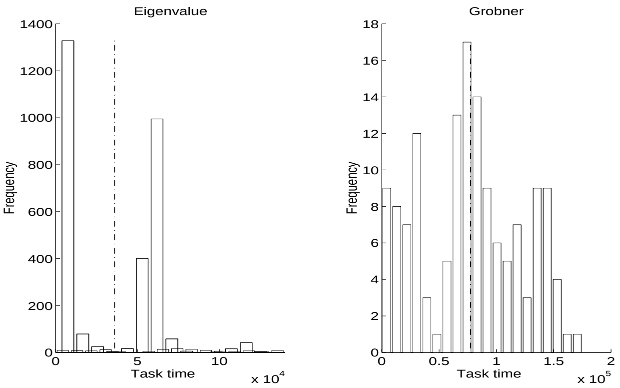

Figure 2: Distribution of task times for (a) the eigenvalue example, and (b) the Gröbner basis example. The average task time is shown by a vertical broken line.

measured actual speedup on the CM-5 multiprocessor. Comparing the speedup curves enabled us to judge the closeness of the simulation to reality.

Given a computation tree $H = \widetilde{H}$, the minimum execution time is at least $\max\{W/P, S\}$. Using more than W/S processors cannot increase speedup, so we plot efficiency against the number of processors only where $P < W/S$. Note that $\max\{W/P, S\}$ may not be achievable even by an optimal run, so our efficiency estimate is pessimistic.

5.2.1 Gröbner Basis Algorithm

Over a standard set of benchmarks, the Gröbner basis problem shows a remarkable variation of tree structure, both in shape and task times. We present data from one example in figure 3, but the speedup profiles from other examples are virtually identical in shape. For our example, the tree has $W = 11053339\mu$s, $n = 142$, $h = 11$, $T = 174860\mu\text{s} \div 1184\mu\text{s} \approx 148$, $S = 474880\mu$s, and $W/S \approx 23$. The simulation is worse than actual runs because they did not work with the same tree; it is hard to control what tree is expanded as the algorithm is highly indeterminate [2].

5.2.2 Bisection Eigenvalue Algorithm

Depending on the input, dynamic load balancing may or may not be necessary for the bisection algorithm. If the eigenvalues are distributed uniformly on the number line, static load balancing usually does quite well. To measure the effectiveness of our load balancing algorithm, we used an input that has one large cluster of eigenvalues that would be assigned to one processor by a typical static scheduling strategy. For this problem, the tree has $W = 108247480\mu$s, $n = 2999$, $h = 20$, $T = 184377\mu\text{s} \div 4486\mu\text{s} \approx 41$, $S = 237917\mu$s, and $W/S \approx 455$. The simulation fits the actual performance closely.

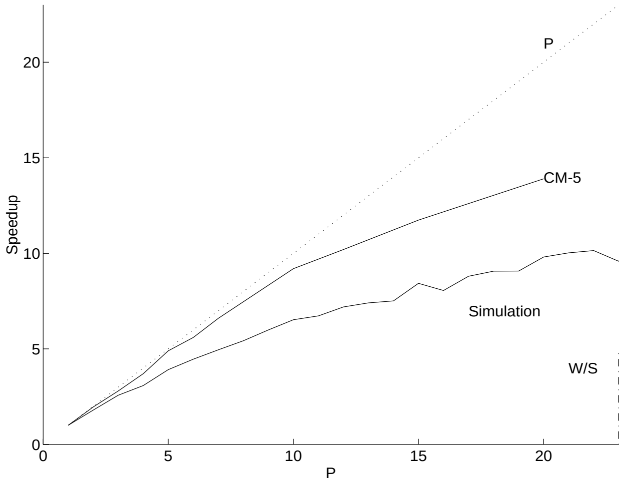

Figure 3: Performance on the Gröbner basis example. Speedups were averaged over 20 runs.

5.2.3 Communication Latency

The random scattering of tasks does not take into consideration the topology of the connection network or the cost of communication. Thus the analysis would be accurate for PRAMs, but it would also be acceptable for networks that are effectively modeled as being completely connected (for example, the LogP model [3]), provided transmitting a task has a negligible cost compared to executing it.

To study the effect of network latency on efficiency, we fixed the number of processors P and varied network latency L. The simulator models network latency by delaying the remote enqueue event of a task by L from the message send event. The send itself is instantaneous, so the sender can continue with computation right away. The variation of efficiency η with latency, L, is plotted, with latency shown in units of $\bar{t} = W/n$, the average task time. The point $L = W/n$ is instructive: since each task suffers about a equal time overhead in communication, one might expect that $\eta(W/n) \approx \eta(0)/2$. $\eta(W/n)$ turns out to be better owing to the overlap of message transfer with computation. For the CM-5, $nL/W \approx 1.66 \times 10^{-4}$, so the loss of efficiency is negligible.

6 Open Problems

Even though the asymptotic optimality of random task placement within the defined model has been settled in this paper, many important problems remain to be solved. First, our analysis guarantees asymptotically linear speedup up to the point where running time is dominated by the sequential paths in the tree. Better constant factors should be possible when P is not close to the limit of parallelism (e.g., the eigenvalue algorithm efficiency is much higher than the best predicted constant between 1 and 64 processors,

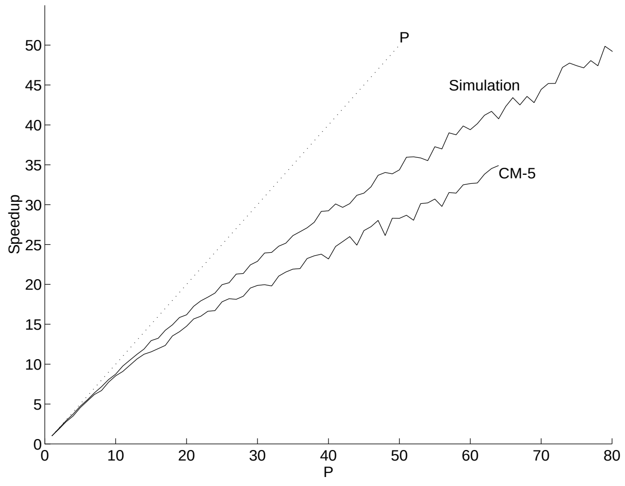

Figure 4: Performance on the eigenvalue example. Speedups were averaged over 20 runs.

because $W/S \gg 64$). Another problem is to extend the results to account for the effects of locality. For unit time tasks in a finite tree, with the restriction that the task tree is also the communication graph, Wu and Kung [10] have described a method based on controlled global information.

7 Conclusion

Dynamic scheduling algorithms are designed to ensure load balance when the computational load is not known in advance. The problem is complicated by one or more of the following: variable task times, dependencies between tasks, and need for data locality to reduce communication cost. While systems exist that address some or all of these concerns, little is known about their theoretical performance bounds under even two of three complicating factors. We have presented and analyzed a load balancing algorithm, using a randomized strategy that has been found useful in application programming. Our main result is that, with high probability, the randomized running time is optimal to within a small constant factor. This improves on previous work by handling both irregular task times and dependencies. Our experimental results suggest that the constant factor in the analysis is small, and the effects of communication latency are tolerable. These results bring the theoretical understanding of load balancing algorithms closer to the techniques used in practice.

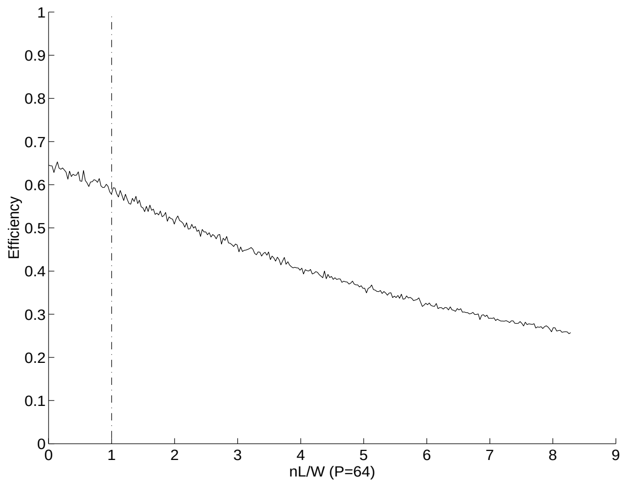

Figure 5: Effect of latency on efficiency.

References

[1] Bruno Buchberger. Gröbner basis: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Company, 1985.

[2] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 169–178, San Diego, California, May 1993.

[3] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Schauser, Eunice Santos, Ramesh Sumbramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[4] Inderjit Dhillon and James Demmel. Private communication., March 1994.

[5] Richard M. Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. *JACM*, 40:765–789, 1993. Preliminary version in *ACM STOC* 1988, pp290–300.

[6] Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 200–211, 1992.

[7] C. D. Polychronopoulos. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C–36(12):1425–1439, December 1987.

[8] Abhiram Ranade. A simpler analysis of the Karp-Zhang parallel branch-and-bound method. Technical Report UCB/CSD 90/586, University of California, Berkeley, CA 94720, August 1990.

[9] David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley and Sons, 1991.

[10] I.-C. Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *32nd IEEE FOCS*, pages 151–162, 1991.