Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds

Michael A. Bender*
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA
bender@cs.sunysb.edu

Michael O. Rabin
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138 USA
rabin@deas.harvard.edu

ABSTRACT

We study the problem of executing parallel programs, in particular Cilk programs, on a collection of processors of different speeds. We consider a model in which each processor maintains an estimate of its own speed, where communication between processors has a cost, and where all scheduling must be online. This problem has been considered previously in the fields of asynchronous parallel computing and scheduling theory. Our model is a bridge between the assumptions in these fields. We provide a new more accurate analysis of an old scheduling algorithm called the *maximum utilization scheduler*. Based on this analysis, we generalize this scheduling policy and define the *high utilization scheduler*. We next focus on the Cilk platform and introduce a new algorithm for scheduling Cilk multithreaded parallel programs on heterogeneous processors. This scheduler is inspired by the high utilization scheduler and is modified to fit in a Cilk context. A crucial aspect of our algorithm is that it keeps the original spirit of the Cilk scheduler. In fact, when our new algorithm runs on homogeneous processors, it exactly mimics the dynamics of the original Cilk scheduler.

1. INTRODUCTION

In this paper we study the problem of executing parallel programs, in particular Cilk programs, on processors that run at different and possibly changing speeds. We develop scheduling algorithms that are designed to run efficiently in a parallel computing environment.

In order to run efficiently, our scheduling algorithms must obey the computational constraints imposed by the parallel setting. For example, the schedulers should make *rapid* decisions about how to assign tasks to processors because otherwise the time to run the scheduler may actually delay the execution of the parallel program. Furthermore, the scheduling decisions must be made with only *partial knowledge* of the actual scheduling problem. This is because both the structure of the parallel program and the speeds of the processors are only known *online*, that is, as the computation unfolds. In addition, the entire state of the system is not automatically visible to any processor; each processor i is only aware of its *own* local state; in order to determine the state of another processor j, processor i must explicitly communicate with j and this communication has a cost. Consequently, a *centralized* scheduler that repeatedly gathers all the information about the states of the processors may be too expensive. This paper describes a scheduling algorithm that is distributed.

We call processors of different speeds *heterogeneous*, and we call identical processors *homogeneous*. In order to obtain efficient algorithms for heterogeneous processors, we must understand the pattern of speed changes so that we can optimize for the *common case*. Our algorithms are optimized for the following setting, which is common in many parallel computing environments.

1. Most of the time the processor speeds are fairly consistent, and therefore a processor can maintain a *good estimate* of its own speed. This estimate naturally is not completely accurate, but most of the time it will be mostly accurate.
2. Processor speeds may occasionally change dramatically, but these changes are limited. The efficiency of our scheduler is allowed to degrade gradually as processors become more erratic.

The general problem of executing parallel programs on heterogeneous processors been studied previously in the fields of *asynchronous parallel computing* and *scheduling theory*. However both of these fields typically assume models that differ dramatically from the parallel setting described above. For example, in the area of asynchronous parallel computing, the processor speeds are assumed to change arbitrarily and adversarially. Unfortunately, this worst-case assumption may be too pessimistic and may lead to inefficient schedules. In the area of scheduling theory, the processor speeds are assumed to remain constant, and the scheduler is allowed to have global knowledge of the state of the system,

*Supported in part by ISX Corporation and Hughes Research Laboratories.

a large amount of time to run, and offline knowledge of the structure of the computation. Based on these assumptions, the system is unrealistically predictable and the scheduler is unrealistically powerful. The model is this paper is a bridge between the assumptions of asynchronous parallel computing and those of scheduling theory. We further describe these fields and then proceed to describe the main results in this paper.

1.1 Asynchronous Parallel Computation

Executing parallel programs on heterogeneous processors is studied intensely in the area of *asynchronous parallel computation* [16, 15, 29, 28, 24, 5, 3, 2]. In this field, the goal is to run a parallel program written assuming synchronization barriers, on a collection of asynchronous processors that do not have a synchronization primitive.

Processors are assumed to be *arbitrarily erratic*. That is, a processor may initially run so slowly that it is essentially stopped, change speed abruptly so that it runs extremely (even infinitely) fast, and then stop once more. Correctness proofs typically assume that processor speeds are determined by an adversary, whose goal is to prevent the parallel program from executing correctly or efficiently. Because processors may change speeds to an arbitrary degree, processors are not assumed to have knowledge of their own speed.

This machinery is useful for mission critical applications, in which a program must run correctly and steadily, regardless of the erratic behaviors of the individual processors. On the other hand, it may not be worth paying the overhead that these schemes entail if (1) the application is not mission critical, or (2) if the processors are not arbitrarily erratic, that is, if they change speeds, but most of the time by too much.

1.2 Scheduling on Related Processors

Executing a parallel program on heterogeneous processors is a common problem in scheduling theory. In this field there is an underlying assumption that processors may have different speeds but that the speeds do not change. The goal is to schedule a parallel program represented as a directed acyclic graph (dag) to minimize the *makespan*, that is, the maximum completion time of the jobs. Using terminology from scheduling theory, the problem is that of *scheduling precedence-constrained tasks on related processors to minimize the makespan.*

Because this problem is NP-hard [30] even when all processors have the same speed, the scheduling community has concentrated on developing approximation algorithms for the makespan. Early papers introduce $0(\sqrt{p})$-approximation algorithms [19, 20], and more recent papers propose $O(\log p)$-approximation algorithms [13, 12]. Unfortunately, some common assumptions from scheduling theory often do not apply to parallel computing, and consequently many scheduling algorithms from this field are not usable in our setting. For example, many of these scheduling algorithms run *offline*, that is, after seeing the entire structure of the parallel program. In addition, the schedulers usually have full knowledge about the state of the system and have the unlimited ability to apply the scheduling decisions.

Finally the quality of many of the scheduling algorithms are measured using the approximation ratio. Even in the homogeneous setting, it is known that the approximation ratio may be misleading [10] by a factor as large as 2. The approximation ratio is dramatically less reliable when processors are heterogeneous for several reasons that we describe shortly.

1.3 The Heterogeneous Setting

To develop intuition about the heterogeneous setting, consider the natural class of *greedy schedules*, in which no processor is allowed to stay idle if there is a task that can be assigned to it. When processors are homogeneous, all greedy schedules have essentially comparable makespans (within a factor of 2 of each other). However, when processors are heterogeneous there may be an unbounded ratio between the makespan of the best greedy schedule and the makespan of the worst greedy schedule. To obtain a schedule having a good makespan, fast processors should be assigned to longer paths in the dag and slower processors should be assigned to shorter paths. This assignment process is computationally difficult because nodes in the dag may belong to many interleaving paths of different lengths.

Thus, for any p homogeneous processors, consider p heterogeneous processors that have the same average speed. The optimal makespan in the heterogeneous setting may be much smaller than in the homogeneous setting. However, practical and computational limitations usually prevent this elusive schedule from being found. On the other hand, it is easy to encounter a poor schedule, especially when the processors' speeds can change. This is why users prefer homogeneous processors to heterogeneous ones, even though in ideal conditions the heterogeneous processors may allow shorter schedulers. Thus, in this paper the objective of an efficient scheduler is to use its heterogeneous processors *as efficiently as if they were homogeneous.*

1.4 Results

We present the following results.

1. We provide a new analysis of of an old scheduling algorithm called the *maximum utilization scheduler* [19]. In particular, we prove a bound on the makespan and on the number of preemptions. Based on this analysis, we generalize this scheduling policy and define the *high utilization scheduler*. We explain why these scheduling policies have close to optimal makespans on dags that represent most parallel programs.

   The algorithms presented so far are not directly implementable because the schedulers require too much centralized control. However, they provide insight into how to schedule parallel programs on heterogeneous systems.

2. We next focus on the Cilk platform and present the main result of the paper. We introduce a new algorithm for scheduling Cilk multithreaded parallel programs on heterogeneous processors. This scheduler is inspired by the high utilization scheduler, modified to fit in a Cilk context. A crucial aspect of our algorithm

is that it retains the original spirit of the Cilk scheduler. In fact, when our new algorithm runs on homogeneous processors, it exactly mimics the dynamics of the original Cilk scheduler.

1.5 Definitions and Notation

There are p processors labeled $1, \ldots, p$ where processor i has speed π_i steps/time. For the sake of convenience, we assume that $\pi_1 \geq \pi_2 \geq \ldots \geq \pi_p$. In much of the paper we assume that the processor speeds do not change. Let π_{tot} steps/time be the *total computing power* of all of the processors, that is, $\pi_{tot} = \sum_{i=1}^{p} \pi_i$. Let π_{ave} steps/time be the *average speed* of the processors, that is, $\pi_{ave} = \pi_{tot}/p$.

A directed acyclic graph (dag) $G = (V, E)$ describes the structure of a parallel program. The nodes of the dag represent *tasks* that the processors must complete, and the edges represent *dependencies* between the tasks. Thus, if there is an edge $(u, v) \in E$, then v cannot be executed until after u completes. In this case, we say that u is a *parent* of v. Tasks are grouped into larger segments of code called *threads*; a *thread* is a length path in the dag.

A *series parallel dag* $G = (V, E)$ is a directed acyclic graph with two distinguished vertices, a *source* s and a *sink* t. The family of series parallel graphs are described using the following grammar. A series parallel dag $G = (V, E)$ is one of the following: (1) A single edge extending from s to t, that is, $V = \{s, t\}$ and $E = \{(s, t)\}$. (2) Two series parallel graphs G_1 and G_2 *composed in parallel.* The sources s_1 and s_2 of G_1 and G_2 respectively are merged into a single source s and the sinks t_1 and t_2 of G_1 and G_2 are merged into a single sink t. (3) Two series parallel graphs G_1 and G_2 *composed in series.* The sink t_1 of G_1 and the source s_2 of G_2 are merged into a single node.

Cilk parallel programs are modeled by *fully strict dags*. A fully strict dag is series parallel, all of the nodes in the dag have outdegree at most 2, and there is one node with indegree 0 and one node with outdegree 0. The *root thread* is a path extending from the first node in the dag to the last node. A node in the root thread with outdegree 2 *spawns* another thread, which continues until it joins the root thread once more. This thread may spawn *child threads*, which may in turn spawn child threads.

Let W_1 represent the *total work*, that is the total number of nodes in the dag G. Let W_∞ represent the *critical path length* of the graph, that is, the number of nodes in the longest chain in G. Consider a modified dag G' in which all nodes that do not have indegree ≥ 2 or outdegree ≥ 2 are removed. Let S_1 represent the total number of edges in G' in the dag, and let S_∞ be the critical path in G'. Let T_p represent the *time* to execute G on p processors. A task or thread is *ready* if all of its predecessors in G have been executed.

We say that a thread is *preempted* if it is interrupted and later resumed, possibly on a different processor. We say that there is a *migration* whenever the state of the system is moved from one processor to a different processor. Thus, there may be a migration if a previously idle processor begins executing a thread because the processor may have obtained the thread from another processor. There is *not* a migration if a processor finished executing a thread and then executes a successor thread in the dag. Thus, there may be a migration without a preemption, or a preemption without a migration. All migrations entail an additional cost, which we take into account.

We say that an event E occurs *with high probability* (*w.h.p.*) if for any $c > 0$ there exists a proper choice of constants such that $\Pr\{E\} \geq 1 - n^{-c}$.

1.6 Related Work

Graham [17, 18] proved that a *list schedule* is a $(2 - 1/p)$-approximation to the optimal makespan, and this result holds for any greedy schedule. (In a *list schedule*, the jobs have fixed priorities and the processors execute the ready tasks in the system with the highest priorities.) This results derives from the following theorem:

THEOREM 1 ([17, 18, 11]). *A greedy schedule (or list schedule) has makespan*

$$T_p \leq \frac{W_1}{p} + \left(\frac{p-1}{p}\right) W_\infty.$$

Jaffe [19] shows that the following preemptive scheduling policy, called a *maximum utilization schedule* is a $O(\sqrt{p})$-approximation algorithm. At all times, maintain the following invariant: if there are i, $i < p$, ready threads, assign these threads to the i fastest processors. Note that threads may be *preempted*; that is, in the middle of the execution of a thread, a faster processor may take up the responsibility for executing the thread. Jaffe [20] then showed that the following nonpreemptive is also a $O(\sqrt{p})$-approximation algorithm for the makespan. Consider the following two schedules and select the one having the better makespan: (1) assign all jobs to the fastest processor, and (2) assign all jobs greedily to processors having speed faster than half the average. More recently, Chudak and Shmoys [13] obtained a $O(\log p)$-approximation by using a linear programming relaxation to decide at which speed each task should run. Chekuri and Bender [12] developed a combinatorial approximation algorithm having the same asymptotic approximation ratio.

Cilk Scheduler. Cilk is a parallel system with a scheduler that has provable performance guarantees. The Cilk scheduling algorithm is entirely distributed and uses the idea of *work stealing*. Namely, if a processor is idle, it randomly chooses another processor, checks if the processor has extra work, and if so, steals some. The work is stolen in a way that avoids a large increase in memory usage or in running time. The Cilk scheduler works as follows. Each processor maintains a double-ended queue, which is called a *ready deque*. Threads can be inserted and removed from either end of the ready deque. If a processor has no local work to do, it begins work stealing. The processor uses its own ready deque as a stack but other processors' deques as queues. Each processor i operates as shown in Figure 1.

CILK SCHEDULER

1. The processor chooses a victim processor j uniformly at random.
2. If the victim j's ready deque is empty, processor i attempts to steal again.
3. Otherwise, it steals the thread T from the *top* of the deque and begins executing it. The processor begins working on thread T until one of three situations:
   (a) Thread T spawns a thread T'. In this case, the processor puts T on the *bottom* of the ready deque and starts work on thread T'.
   (b) The thread T returns or terminates. If the deque is not empty, the processor begins working on the *bottom* thread. If the deque is empty, it tries to steal and execute thread T's parent. Otherwise, if the parent is busy, the processor attempts to work steal.
   (c) The thread reaches a synchronization point. In this case, the processor attempts to work steal. (Note that the deque is empty.)

Figure 1: The Cilk Scheduler.

2. HIGH UTILIZATION SCHEDULES

We now provide a new analysis of the maximum utilization scheduling policy. This scheduler maintains the following invariant. During each time interval in which there are exactly i ready threads, for each $i < p$, the fastest i processors execute these tasks. If there are $i \geq p$ ready threads, then all of the processors work. Beyond this basic restriction, any processor may execute any task. Note that in order to maintain this invariant, the scheduling policy must allow preemptions.

The maximum utilization scheduling policy is a $O(\sqrt{p})$-approximation algorithm but there are other scheduling algorithms that have comparable approximation ratios and that do not even require preemptions. As a result, the maximum utilization strategy has languished in relative obscurity. However, many of the other scheduling strategies suffer from the following drawbacks: either (1) they are too complicated to be implemented efficiently, or (2) they produce schedules that are qualitatively unsatisfactory.

The maximum utilization schedule has a straightforward generalization, which we call a *high utilization schedule.* In this scheduler we relax the invariant so that at all times: if there are i, $i < p$, ready threads, the fastest idle processor is at most β times faster than the slowest busy processor. Thus, when $\beta = 1$, we obtain a maximum utilization schedule. This makespan of a high utilization schedule may be inferior to the makespan of a maximum utilization schedule, but may have the advantage of fewer preemptions.

We will demonstrate two advantages of high utilization schedules: (1) in the common case in parallel computing, high utilization schedules are almost optimal, and (2) they convey a straightforward message to practitioners, run your parallel program on the fastest processors that you can find, and this may be all the optimization that is required. On actual system such as the Cilk platform, the unembellished high utilization schedule may be too complicated to implement. However, the straightforward concept of using the fastest processors that you can find can be generalized so that it is practical. Thus, high utilization strategies are important because of the guidance that they give in actual situations.

THEOREM 2. *Any maximum utilization schedule has makespan*

$$\begin{aligned} T_p &\leq \frac{W_1}{p\,\pi_{ave}} + \left(\frac{\pi_2}{\pi_1} + \frac{\pi_3}{\pi_2} + \cdots + \frac{\pi_p}{\pi_{p-1}}\right)\frac{W_\infty}{p\,\pi_{ave}} \\ &\leq \frac{W_1}{p\,\pi_{ave}} + \left(\frac{p-1}{p}\right)\frac{W_\infty}{\pi_{ave}}\,. \end{aligned}$$

PROOF. We introduce an accounting tool. We postulate $p-1$ disjoint *shadow threads* $ST_2, ST_3, \ldots ST_p$. Each shadow thread is an imaginary chain of tasks. When a processor i is unable to do any work on an *actual thread*, we say that the processor begins working on its *shadow thread* ST_i.

Consider any time interval in which processor i is idle and thus working on its shadow thread ST_i. Since not all processors have actual work, we are assured that progress is being made on the critical path at the rate of the slowest working processor. That is, since only faster processors $1 \ldots i-1$ may be working on the computation, the critical path is advancing at a rate of at least π_{i-1} steps/time.

Because the critical path has length W_∞, processor i can work on ST_i for $\pi_i/\pi_{i-1}\,W_\infty$ time units. Processor 1 is never idle. Therefore the total amount of work the processors dedicate to actual and shadow threads is at most $W_1 + (\pi_2/\pi_1 + \pi_3/\pi_2 + \ldots + \pi_p/\pi_{p-1})\,W_\infty$. Because the processors operate at π_{tot} steps/time we obtain the desired bound. □

Note that from the Theorem 2, we obtain Theorem 1 as a corollary. The makespan can be marginally improved by more strategically placing processors on threads. Namely, put the i-th fastest processor on the i-th longest critical path. This policy guarantees that the critical path always progresses at least at the average speed of the working processors.

CLAIM 3. *Suppose that the maximum utilization strategy additionally maintains the invariant that the i-th fastest processor executes the thread that is i-th farthest from the end of the dag. This amounts to putting the fastest processor on the critical path. Then the computation has makespan.*

$$T_p \leq \frac{W_1}{p\,\pi_{ave}} + \left[\frac{\pi_2}{\pi_1} + \frac{2\,\pi_3}{\pi_1+\pi_2} + \frac{3\,\pi_4}{\pi_1+\pi_2+\pi_3} + \cdots + \frac{(p-1)\,\pi_p}{\pi_1+\pi_2+\ldots+\pi_{p-1}}\right]\frac{W_\infty}{p\,\pi_{ave}}.$$

Unfortunately, this gain in makespan seems small in comparison to the potentially infinite number of additional preemptions that this policy entails.

The proof of Theorem 2 extends to prove the following theorem that provides a bound on the makespan of a high utilization schedule.

THEOREM 4. *Any high utilization schedule has makespan*

$$T_p \leq \frac{W_1}{p\,\pi_{ave}} + \left(\frac{p-1}{p}\right)\frac{\beta\,W_\infty}{\pi_{ave}}$$

We now provide a bound on the number of migrations in a high utilization schedule. of the execution from another

THEOREM 5. *Consider a high or maximum utilization schedule of an arbitrary dag. If there are a total of S_1 threads, then there are at most $2S_1$ migrations.*

PROOF. We divide the computation into phases, S_1, S_1-1, ..., 2, 1, where in phase Π the computation has Π (incomplete) threads. Within a phase, a computation has no migrations at all. A phase begins when the number of *active threads* (e.g., threads currently being executed by processors) changes.

Assume without loss of generality (w.l.o.g.) that at most one thread completes at any time. (If two threads complete simultaneously, we break the tie arbitrarily.) There are two cases for the dynamics of the schedule when a thread completes. (1) When a thread T_α completes, no new threads active become active. Then the slowest currently-active processor k migrates to the idle pool, and the processor j on T_α migrates to k's thread. (If we are lucky, the slowest currently-active processor k is already on thread T_α.) (2) When a thread T_α completes, x new threads become active. Then $x-1$ processors migrate from the idle pool to a new active thread and one processor migrates from the completed thread T_α to a new active thread. □

Thus, if there is a bound M on the time to migrate, then we have a bound on the increase in makespan from Theorem 6 when migrations have a cost, namely $2MS_1/p$. The quantity M may include the cost to send the system state from one processor to another or even may include the cost to restart a thread from some previous checkpoint. One could balance the parameters M and β to optimize the makespan, e.g., only preempt and migrate if there is a substantial gain.

2.1 Performance in the Common Case

Even though the high utilization schedule is a $O(\sqrt{p})$ approximation algorithm for general dags, on dags that represent most parallel programs, the algorithm has a substantially better performance. In most parallel programs $W_1/p \gg W_\infty$ [10]. An interpretation of this inequality is that the parallel program has enough inherent parallelism to justify the use of p processors. Observe that in Theorems 2 and 4, W_1/π_{tot} is a lower bound on the makespan, and when $\beta > 1$ is sufficiently close to 1, this quality dwarfs $\beta W_\infty/\pi_{ave}$. Therefore, even though the high utilization schedule is a $O(\sqrt{p})$ approximation for general dags, in the case of dags representing typical parallel programs, it is almost optimal. This is not true of the nonpreemptive $O(\sqrt{p})$ approximation algorithm.

3. AN ENHANCED CILK SCHEDULER

Direct implementation of the the scheduling policies in the previous section are impractical because they rely on global control. However, the general design principle of high utilization is critical, and we apply this concept in Cilk scheduling. In this section we describe an enhanced Cilk scheduler that runs correctly and robustly even when processors have different speeds. Moreover, when the processors run at *similar* speeds, our new schedule behaves identically to the standard Cilk scheduler. Thus, an important feature of our scheduler is that it is extremely similar to the original scheduler at a small cost in algorithmic complexity.

In this algorithm there are two kinds of migrations: (1) *steals* and (2) *muggings*. In a steal, a processor does not interrupt a thread. Instead, a processor begins working on a thread at the top of another processor's ready deque. In a mugging, there is no work on another processor's ready deque, and so the processor "mugs" a slower processor and takes the thread that the slower processor was working on.

Thus, if all processors operate at speeds within an β factor of each other, then there are no muggings and the scheduler behaves like the standard Cilk scheduler. The parameter β can be tuned to optimize system performance.

3.1 Design Assumptions

We make the following additional assumptions: (1) Each processor steals at a rate proportional to its speed. (2) Steals and steal attempts are completed in an amount of time that is proportional to the speed of the processor doing the stealing/mugging. It is important to have a platform so that the steal responses do not depend on the speed of the victim processor because otherwise the slowest processor can delay the entire system.[1] There are several ways to ensure this design principle. For example, if there are at most two magnitudes of difference between the fastest and slowest processor speeds, then the times for steal attempts, muggings, and steals can be calculated accordingly. We could also require some mechanism for communicating steal attempts, such as a shared memory, that allows one processor to look directly

[1] If the steal attempts run at the speed of the victim processor then the work-stealing approach may not have guaranteed good performance. This is because the root thread of the computation may reside on a processor that is entirely stopped, and the computation cannot proceed.

ENHANCED CILK SCHEDULER

1. Processor i chooses a victim processor j uniformly at random.
2. If the victim j's deque is not empty, it steals the thread T from the *top* of the deque.
3. If the victim j's deque is empty, but the victim is working on a thread T and *its speed is β times slower than processor i,* then i *mugs* j, that is, i interrupts j and takes the thread T.
4. If processor i has located a thread T, i works on T until one of four situations:
   (a) Thread T spawns a thread T'. In this case, the processor puts T on the *bottom* of the ready deque and starts work on thread T'.
   (b) The thread T returns or terminates. If the deque is not empty, the processor begins working on the *bottom* thread. If the deque is empty, it tries to steal and execute thread T's parent. Otherwise, if the parent is busy, the processor attempts to work steal.
   (c) The thread reaches a synchronization point. In this case, the processor attempts to work steal. (Note that the deque is empty.)
   (d) Processor i is mugged and the thread T is migrated to another processor. In this case, processor i attempts to work steal.
5. Otherwise, there is a failed steal attempt; processor i tries to steal again.

Figure 2: The Enhanced Cilk Scheduler.

into the deques of other processors. The pseudocode for the Enhanced Cilk Scheduler appears in Figure 2.

3.2 Analysis

We now analyze the running time of the Enhanced Cilk Scheduler. We prove the following performance guarantee.

THEOREM 6. *W.h.p., the execution time T_p of the enhanced Cilk Scheduler is bounded as follows.*

$$T_p \leq \frac{W_1}{p\,\pi_{ave}} + O\left(\frac{W_\infty}{\pi_{ave}}\right).$$

We use an accounting argument to prove Theorem 6. Observe that at all times a processor is either (1) executing an instruction, or (2) attempting to steal (and perhaps actually stealing or mugging). For simplicity of analysis, we assume that each of these operations requires one unit of work. (In fact, executing an instruction is likely to be much faster and so in our analysis we can group multiple instructions together.)

We postulate two *buckets* that we use for accounting, a *work bucket* and a *steal bucket*. Each time a processor completes a unit of work on the dag it puts one dollar into the work bucket; each time a processor completes a steal attempt (successful or not) it puts one dollar into the steal bucket. (This approach was used in the original paper of [10] and in much of the subsequent work on Cilk.) There are π_{tot} dollars that enter the buckets per unit of time. Therefore, if at the end of the computation, there are a total of D dollars in both buckets, then the computation ran in time D/π_{tot}.

Computing the number of dollars in the work bucket is straightforward, because each time the processor completes one unit of work, it puts a dollar in the work bucket.

Observation 1. *At the end of the computation there are a total of exactly W_1 dollars in the work bucket.*

We now use a potential-function argument to prove a bound on the number of dollars in the steal bucket. This argument is an extension of the result in [1, 7] and begins with some definitions.

Definitions. For any (nonroot) node v, suppose that node u is the last of v's parents to be executed. Then we say that the execution of node u *enables* node v. Node u is called the *designated parent* of v and edge (u, v) is called the *enabling edge.* The graph composed of all the enabling edges is called the *enabling tree.* The node that is being executed at time t by processor i is called the *assigned node of processor i.* We assign weights to all of the nodes, so that we can use these weights in a potential function argument. Let $d(u)$ denote the *depth* of node u in the dag. Each node u has *weight* $w(u) = W_\infty - d(u)$.

Now supplied with these definitions, we describe the Structural Lemma of the deques. This lemma guarantees that for any deque at all times during the execution if the work stealing algorithm, the designated parents of the nodes in the deque lie on the root-to-leaf path in the enabling tree.

LEMMA 7 ([1, 7]). *Let k be the number of (ready) nodes in a given deque at any time t, and let $v_1, v_2, \ldots, v_k$ denote these nodes ordered from bottom to top. Let v_0 be the assigned node. In addition, for $i = 1 \ldots k$, let u_i be the designated parent of v_i. Then for $i = 1 \ldots k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, although it may be that $u_0 = u_1$, for $i = 2 \ldots k$, $u_{i-1} \neq u_i$. Thus, the weights of the nodes increase from bottom to top, that is, $w(v_0) \leq w(v_1) < w(v_2) < \ldots < w(w_k)$.*

Proof sketch: The proof is by induction on times in which the structure of the deque changes, as in [1, 7]. There are five possible ways that the deque may change: (S) The top node of the deque is stolen; (E0) The assigned node enables 0 children; (E1) The assigned node enables 1 children; (E2) The assigned node enables 2 children; (M) The processor is mugged and the assigned node is moved to a faster processor.

The first four cases are described and analyzed in the proof in [1, 7]. However, the case of muggings is unique to the heterogeneous setting. This case can be integrated into the correctness proof using arguments similar to those used in the cases of (S) and (E0). □

We now present the potential function that we will use [1, 7]. Let R_t be the set of ready nodes at time t. Each node is either in some deque or assigned to and executed on some processor. For each ready node $v \in R_t$, we define its potential $\phi_t(v)$ as

$$\phi_t(v) = \begin{cases} 3^{2 \cdot w(v)-1} & \text{if } v \text{ is assigned;} \\ 3^{2 \cdot w(v)} & \text{otherwise.} \end{cases}$$

We let $\Phi_t(i)$ denote the sum of the potentials of the nodes on processor i at time t. We let $\Phi_t = \sum_{i=0}^{p} \Phi_t(i)$ be the value of the potential function at time t. Thus, the initial potential is $3^{2 \cdot W_\infty}$ because the root node has depth 0 and is initially unassigned. The final potential is 0 because all nodes have been completed.

Observation 2. *For any processor at time t during the execution of the scheduling algorithm, the potential of the topmost nodes in the deques contributes at least 3/4 of the potential associated with the processors that have nonempty deques.*

We now divide the computation into *phases*, which are defined inductively by when steal attempts occur. The first phase begins at time $t = 0$, the start of the computation, and it ends after $(\beta + 2)p$ steal attempts have occurred. The i-th phase begins at the end of the $(i-1)$-th phase and completes, as before, after $(\beta+2)p$ additional steal attempts have been made.

THEOREM 8. *There is at least a constant probability that within each phase, the potential drops by at least a constant factor. Therefore, there are at most $O(\log n)$ phases, both expected and with high probability.*

PROOF. At any time t we partition the potential $\Phi_t = D_t + S_t + F_t$ into 3 disjoint components. The component D_t is associated with processors whose deque contains nodes. The rest of the potential is associated with processors that have empty deques, but which may have assigned nodes. We divide this remaining potential into components associated with processors we define as *slow* and *fast* respectively. A processor i is called *slow* in phase ℓ, if during phase ℓ, the processor does not have time to finish executing the node that it was working on when the phase began. A processor i is called *fast* otherwise.

We first consider the potential D_t associated with the set of processors whose deques are *not* empty. Recall that at least 3/4-th of the potential from nodes in the deques is exposed at the top of the deques. Consequently, because there are $(2+\beta)$ steal attempts in any phase, the probability that there is no steal attempt in a deque is at most $e^{-(2+\beta)}$. When the node at the top of the deque is stolen, the potential of this node decreases by a factor of 2/3 because the node is now assigned to a processor.

Let value Q be the sum of the potentials of the nodes at the top of the deques. Then the expected value of the remaining potential of these nodes after the phase ends is at most $e^{-(2+\beta)}\, Q + (1 - e^{-(2+\beta)})\, 2Q/3$. Therefore, by the Markov inequality, there is at least a constant probability that the potential associated with these nodes decreases by at least a constant factor. Consequently, by Corollary 2, with at least a constant probability the potential associated with all the nodes in those deques decreases by at least a constant factor.

We now examine the component F_t of the potential, that is, the potential associated with fast processors having empty deques at the start of phase ℓ. For any such processor i, the completion of i's assigned node causes the potential to decrease by at least a constant factor because i's original assigned node will be completed.

Finally, we examine the component S_t of the potential, that is, the potential associated with slow processors having empty deques at the start of phase ℓ. In order to reduce the potential of a slow processor i that contributes to S_t, another processor j must (1) choose to mug processor i, and (2) complete one node of the thread that it obtained from processor i. In order to mug i, processor j must be more than β times faster than processor i. How many steal attempts are there in phase ℓ that satisfy these conditions? Any processor that makes $\beta + 2$ steal attempts in the phase must be more than β times faster than processor i, which does not even finish executing one node. Consequently, in $(\beta + 2)p$ steal attempts, there will be at least p steal attempts that satisfy all of these conditions. Therefore, the probability that any given slow processor is not mugged is at most $1/e$. Let value Q' be the sum of the potential of the nodes being executed by the slow processors. Then the expected value of the remaining potential of these nodes after the phase ends is at most Q'/e. Therefore, by the Markov inequality, there is at least a constant probability that the potential associated with these nodes decreases by at least a constant factor.

By considering all three cases, we conclude that there is at least a constant probability that the total potential decreases by at least a constant factor. Therefore, by applying Chernoff Bounds, we conclude that after at most $O(W_\infty)$ phases the potential has decreased until it is zero, both expected and with high probability. □

From Lemma 8, we conclude that there are at most $O(\beta W_\infty p)$ steal attempts and consequently $O(\beta W_\infty p)$ dollars in the steal bucket. Therefore, the running time of the algorithm is $W_1/(p\pi_{ave}) + O(\beta W_\infty \pi_{ave})$, which concludes the proof of Theorem 6. □

Finally, we end this section by observing that it is not even necessary in the previous argument to define a particular value of β. That is, the argument works if processor i mugs another processor j as long as $\pi_i > \pi_j$. The advantage of introducing β, is that it reduces the number of migrations.

4. CHANGING SPEEDS AND DISCUSSION

So far we have assumed that the processor speeds are fixed. Our algorithms also run correctly when the speeds change, but possibly at an additional cost. To understand why, first reconsider high utilization schedules. Even when speeds change, the high utilization requirement can still be maintained through additional migrations. The same holds for the high utilization scheduler. The value of β can be chosen to smooth out the schedule so that small fluctuations in processor speeds do not lead to as many additional migrations.

The same advantages apply to our enhanced Cilk scheduler. Our scheduler uses no global control, and in its place only brief interactions between pairs of processors. Processors do not even have to store information about the speeds of the other processors, which might quickly become out of date. Consequently, this algorithm easily adapts to changing speeds. As speeds are modified, there may be additional steal attempts and muggings. As before, the value of β can be chosen to remove unnecessary muggings. Thus, the performance of the scheduling algorithm degrades gracefully as the speeds become more erratic.

5. ACKNOWLEDGMENTS

The first author warmly thanks Charles Leiserson for suggesting this problem, for enjoyable meetings in the earlier stages of this work, and for much excellent advice.

6. REFERENCES

[1] N. Arora, R. Blumofe, and G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[2] Y. Aumann, M. A. Bender, and L. Zhang. Efficient execution of nondeterministic parallel programs on asynchronous systems. *Information and Computation*, 139(1):1–16, 25 Nov. 1997. An earlier version of this paper appeared in the *8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1996.

[3] Y. Aumann, K. Palem, Z. Kedem, and M. O. Rabin. Highly efficient asynchronous execution of large grained parallel programs. In *Proceedings of the 34th Annual Symposium on the Foundations of Computer Science*, pages 271–280, November 1993.

[4] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proceedings of the 33rd Annual Symposium on the Foundations of Computer Science*, pages 147–156, 1992.

[5] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. *Theoretical Computer Science*, 128:3–30, 1994.

[6] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 198–205, May 1999.

[7] R. Blumofe. Scheduling multithreaded computations by work stealing. Seminar Talk. Joint work with N. Arora C. Leiserson, and G. Plaxton, 1998.

[8] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[9] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, California, May 1993.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, Nov. 1994.

[11] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

[12] C. Chekuri and M. A. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. In *Integer Programming and Combinatorial Optimization*, volume 1412, pages 383–393, 1998.

[13] F. A. Chudak and D. B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds (extended abstract). In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, New Orleans, Louisiana, 5–7 Jan. 1997.

[14] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[15] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. of the ACM Symposium on Parallel Architectures and Algorithms*, pages 85–94, 1989.

[16] P. B. Gibbons. A more practical PRAM model. In *Proc. of the 1st ACM Symposium on Parallel Architectures and Algorithms*, pages 158–168, June 1989.

[17] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[18] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, Mar. 1969.

[19] J. M. Jaffe. An analysis of preemptive multiprocessor job scheduling. *Mathematics of Operations Research*, 5(3):415–421, Aug. 1980.

[20] J. M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, Aug. 1980.

[21] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Annual ACM Symposium on the Principles of Distributed Computing*, pages 211–221, 1989.

[22] P. Kanellakis and A. Shvartsman. Effecient parallel algorothms on restartable fail-stop processors. In *Proceedings of the 10th Annual ACM Symposium on the Principles of Distributed Computing*, pages 23–36, 1991.

[23] P. Kanellakis and A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.

[24] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformation for resilient parallel computation via randomization. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[25] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, May 1991.

[26] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22rd Annual ACM Symposium on Theory of Computing*, pages 138–148, May 1990.

[27] J. W. W. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. *North-Holland*, pages 349–353, 1974.

[28] C. Martel, A. Park, and R. Subramonian. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science*, pages 590–599, 1990.

[29] N. Nishimura. Asynchronous shared memory parallel computation. In *Proc. of the 2nd ACM Symposium on Parallel Architectures and Algorithms*, pages 76–84, 1990.

[30] J. Ullman. NP-complete scheduling problems. *Journal Computing System Science*, 10:384–393, 1975.