Distributed Job Scheduling in Rings

Perry Fizzano ⋆

University of Puget Sound, Department of Mathematics and Computer Science

David Karger †

MIT, Laboratory for Computer Science

Clifford Stein ‡

Dartmouth College, Department of Computer Science

Joel Wein §

Polytechnic University, Department of Computer and Information Science

⋆ Research partially supported by NSF grant CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award. This work was done while the author was a graduate student at Dartmouth College. email: pfizzano@ups.edu

†Research supported by a Hertz Foundation Graduate Fellowship and by NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation and Xerox Corporation. This work was done while the author was a graduate student at Stanford University. email: karger@theory.lcs.mit.edu

‡Research partially supported by NSF grant CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award. email: cliff@cs.dartmouth.edu

§Research partially supported by NSF grant CCR-9211494 and a grant from the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications. Part of this work was done while the author was visiting DIMACS. email: wein@mem.poly.edu

Proposed Running Head:

Distributed Job Scheduling in Rings

Contact Author:

Perry Fizzano

University of Puget Sound

Dept. of Mathematics and Computer Science

1500 N. Warner

Tacoma, WA 98416

(206) 756 1522

email: pfizzano@ups.edu

Abstract

We give a distributed approximation algorithm for job scheduling in a ring architecture. In contrast to many other parallel scheduling models, the model we consider captures the influence of the underlying communications network by specifying that task migration from one processor to another takes time proportional to the distance between those two processors in the network. As a result, our algorithm must balance both computational load and communication time.

The algorithm is simple, requires no global control, and yields schedules of length at most 4.22 times optimal. We also give a lower bound on the performance of any distributed algorithm, and the results of simulation experiments which suggest better performance than does our worst-case analysis.

1 Introduction

With the promise of parallel and distributed computing comes the challenge of designing algorithms which effectively utilize the resources of a parallel or distributed system. In a parallel or distributed system there is competition amongst a number of processes for the resources of the system, such as processing power or communication bandwidth. In order to achieve maximum performance, we must develop algorithms that effectively allocate the resources of the system to these competing processes in an efficient fashion. These algorithms must be simple, so that control overhead does not nullify the performance gains due to the algorithm. The algorithms must also cope with the distributed nature of the problems: typically the algorithm has only local information.

In this paper we consider a basic resource allocation problem, *job scheduling*. Simply put, the problem is to assign each of a set of independent tasks to processors in the system so as to finish the processing of the set of tasks as quickly as possible. Job scheduling arises frequently in parallel computing, for example in algorithms for automatic loop parallelization [3, 18, 19, 28] or in the use of a parallel system to process batches of transactions of independent sequential programs.

There is a wealth of literature on the analytical evaluation of parallel machine scheduling (see Lawler *et al.* [21] for a number of examples) but much of it fails to capture the full complexity of many real scheduling problems. Many of the proposed algorithms have significant control overhead, and almost all of the literature ignores the communication constraints imposed by an underlying network architecture. In practice, however, the decision of how much work to send where can be greatly affected by the path lengths between nodes in the network. In addition, much of the scheduling literature assumes global control of task allocation.

Our goal in this paper is to design and evaluate a scheduling algorithm in a model that reflects these real-world considerations. We assume that each processing unit is at a different node of a communication network, and only has direct communication with its neighbors in the network. As a result, task migration from one processor to another takes time proportional to the distance between those two processors in the communication network; it is this element which most differentiates the model we consider from the majority of previous scheduling models.

Our work was inspired by the work of Deng, Liu, Long and Xiao [11] and Awerbuch, Kutten and Peleg [6], who introduced this distributed scheduling model. We will soon discuss their work in detail, but we note here that the major accomplishment of these papers that is relevant to our work is the algorithm of Awerbuch, Kutten and Peleg for distributed job scheduling in *any* network. Our goal in this paper is to restrict our attention to a *specific* network architecture, and as a result to design simpler algorithms that will also provide much stronger performance guarantees. The study of parallel and distributed algorithms on specific network architectures has a rich history; a good reference is the textbook authored by Leighton [22]. We focus on the *ring architecture*, which is an important network in both theory and practice. From a theoretical perspective, the ring is a basic network structure, and much work has been done on developing and analyzing algorithms for it [5, 7, 16, 22, 24, 31, 32]. In practice the ring is either the basis of or an essential component of many parallel and distributed architectures [20, 29, 33, 34].

Our primary vehicle for the evaluation of an algorithm is by a *worst-case analysis* of the quality of the schedule it produces. Our goal will be to schedule all jobs as quickly as possible, or in other words to produce a short schedule. We will let $\mathcal{I}$ denote an instance of the scheduling problem, and $\mathcal{L}(\mathcal{I})$ the length of the shortest possible schedule for $\mathcal{I}$. We call an algorithm A a *ρ-approximation algorithm* if it produces, *on any instance* $\mathcal{I}$, a schedule of length no more than $\rho\mathcal{L}(\mathcal{I}) + O(1)$. The worst-case study of approximation algorithms has a long and rich history; see [15] for an extensive treatment.

The major contribution of this paper is a *distributed* 4.22-approximation algorithm for scheduling jobs in a ring. Our algorithm has low control overhead, accounts for communication constraints and does not require global control. We note that, although our algorithm is a distributed algorithm, by virtue of our definition of a ρ-approximation algorithm we evaluate its performance by comparison to the the optimal algorithm with *centralized control*; we show that our algorithm, in a distributed manner, produces schedules of length at most 4.22 times that of the best schedule that could be computed by any algorithm with centralized control.

Deng *et al.* gave an optimal polynomial-time algorithm for this problem that requires centralized control [11]; in contrast, in this paper, we prove that *no distributed algorithm* can solve this problem optimally. Finally, although our primary vehicle of evaluation is worst-case analysis, we also report on an experimental study of our algorithm, which suggests better performance than does our worst-case analysis.

Model and Problem Statement: We are given an m-processor ring, with identical processors numbered $1, \ldots, m$. We will often discuss addition on the processor indices, and it is assumed throughout the paper that this addition will be done mod m, i.e processor $m+i$ is the same as processor i (Notice this means that processor m is also referred to as processor 0). Each processor i starts, at time 0, with x_i unit-sized jobs; we define $n = \sum_{i=1}^{i=m} x_i$. We require that each job be processed on exactly one processor without preemption. We assume that in one unit of time each processor can receive some jobs from each neighbor, send some jobs to each neighbor, and process one unit of work. Tel [34] and Choi and Esfahanian [9] argue that the assumption of overlap occurring in these three tasks is supported by current technology, and this is the model considered by Deng *et al.*, Awerbuch *et al.*, and others [6, 11, 4, 27]. If a processor, at time t, sends a job to a neighbor, the neighbor receives the job at time $t+1$. We assume that the job granularity is large enough so that the time for basic control operations, such as simple arithmetic, is negligible.

In this paper we assume, as do our predecessors [6, 11, 4], that there is no bound on the capacity of each network link in the ring; specifically, we allow a processor to send an arbitrary number of jobs to a neighbor in one time step. We do not consider this feature of the model to be overly unrealistic, since networks of increasingly high bandwidth are rapidly becoming available, and a large amount of data can be transferred in one step. Furthermore, in the scheduling of independent iterations of parallel loops, one of the major real-world sources of motivation for the problem of scheduling independent jobs on a parallel computer, a large number of iterations can be characterized by just a few numbers describing a range of loop indices [19]. Finally, we note that a related problem, that of *load-balancing*, has also received significant attention in an "unbounded link capacity model" (e.g. see [23, 10, 8, 30, 12, 13]).

Previous Work: To the best of our knowledge, the first paper to consider the problem of scheduling independent jobs on a specific network of processors was that of Deng, Liu, Long and Xiao [11]. They gave a polynomial-time algorithm that produced optimal length schedules in the case when all jobs are of the same size, and an approximation algorithm in the case when jobs are of different sizes; both of these algorithms require centralized control, which means that at every point in time the entire state of the system is known to the algorithm, and the decisions of the algorithm are made and then passed to the processing units instantaneously. In addition Deng *et al.* gave several distributed algorithms for special cases and models – for the ring architecture, they showed that no distributed algorithm can produce schedules that are guaranteed to be of length less than $\frac{1}{2}m^{\frac{1}{6}}$ times optimal (for m processor rings) if the algorithm takes no advantage of the information about network load that it can gather over time. While this is an interesting result, in realistic settings an algorithm would certainly want to take advantage of this information. Phillips, Stein and Wein [27] gave a centralized 2-approximation algorithm for a very general form of the problem, as well as hardness-to-approximate results and approximation algorithms for different optimality criteria. Hoppe and Tardos [17] gave a polynomial time centralized algorithm for general networks and unit size jobs. Their algorithm was the first polynomial time algorithm that produced optimal length schedules in networks with finite capacity.

The paper most directly related to our work is that of Awerbuch, Kutten and Peleg [6], who were the first to study the problem of *distributed* dynamic job scheduling in general networks; they gave a distributed polylogarithmic-approximation algorithm. Despite the importance of this work, the bounds are quite weak when applied to specific networks such as the ring; our goal in this paper is to give a much-improved algorithm for this architecture. It is not hard to see that, when applied to the ring, their algorithm yields a (large) constant-performance guarantee; however, even when a number of refinements are incorporated into their approach and analysis, the performance guarantee of our algorithm is much better.

In addition, the control structures of our algorithm are much simpler than those arising from the application of the general approach of Awerbuch, Kutten and Peleg to the ring. In our algorithm, a job will be transmitted over the same link at most twice, whereas in the algorithm of Awerbuch, Kutten and Peleg a job may be passed over the same link $O(\log m)$ times – they essentially decompose the network into a hierarchy of intervals of logarithmic depth, and progressively distribute and process the jobs at different levels of the hierarchy.

Distributed job scheduling is related to, but different from, *load balancing*, which is another common problem in parallel and distributed systems (see [2, 14, 23, 10, 8, 30, 12, 13]). In the load balancing problem one is given a set of tasks or tokens and must distribute the tasks so that each processor in the system has approximately the same number. The scheduling problem is more complicated: just balancing the load may lead to an excessively long schedule, and a shorter one might be achieved by doing more of the work locally rather than spending the time to send it far away in order to achieve balance. The scheduling problem, in some sense, requires both the identification of neighborhoods of appropriate size in which to share work and the balancing of the work over those neighborhoods.

The model we consider is not the only attempt to incorporate the constraints imposed by communication latency into a scheduling model. A line of research which is quite different from ours, yet still has some similarity in spirit, was started by Papadimitriou and Yannakakis [25]. They modeled communication issues in parallel machine scheduling by abstracting away from particular networks and instead describe the communication time between *any* two processors by one network-dependent constant. They considered the scheduling of precedence-constrained jobs on an infinite number of identical machines in this model; the issues involved and the sorts of theorems proved are quite different from our results.

The rest of this paper is organized as follows. In Section 2 we present an algorithm for the most basic version of the problem and prove that it is a 4.22-approximation algorithm. In Section 3 we give an extension of the algorithm to more general settings. We present a lower bound on the performance of any distributed algorithm in Section 4, and in Section 5 we report on computational experiments with the algorithm. We conclude with some open problems in Section 6.

2 The Basic Algorithm

We begin by describing our algorithm under the unrealistic assumption that each job may be broken up into fractional parts, which can then be run on different processors. This assumption simplifies the analysis; we will later remove this assumption and modify the algorithm to work in the more realistic setting.

2.1 A lower bound

The intuition for our algorithm comes from the following lower bound on the performance of any scheduling algorithm for the ring, even one with centralized control. Recall that $\mathcal{L}(\mathcal{I})$ denotes the minimum amount of time necessary to process the jobs in instance $\mathcal{I}$.

Lemma 1 *Let $\mathcal{I}$ be an instance of the ring scheduling problem with x_j jobs starting at each processor j at time 0. Then for any $i, k \leq m$,*

$$\mathcal{L}(\mathcal{I}) \geq \sqrt{\frac{k^2}{4} + x_i + \ldots + x_{i+k}} - \frac{k}{2}.$$

Proof: Assume for notational convenience that $i = 0$; the other values of i follow symmetrically. Recall that processor 0 is also processor m. We focus on the work that starts on processors $0, \ldots, k$ by assuming that there is no other work in the system. Consider the time needed to process these jobs. In the best possible circumstances, processors 0 through k will each process $\mathcal{L}(\mathcal{I})$ jobs by time $\mathcal{L}(\mathcal{I})$. Processors $k+j$ and $m-j$, $1 \leq j \leq \mathcal{L}(\mathcal{I})$, which must sit idle for j time steps before any work can reach them, can process at most $\mathcal{L}(\mathcal{I}) - j$ jobs by time $\mathcal{L}(\mathcal{I})$. As a result, $\mathcal{L}(\mathcal{I})$ must satisfy

$$x_0 + \ldots + x_k \leq (k+1)\mathcal{L}(\mathcal{I}) + 2\sum_{j=1}^{\mathcal{L}(\mathcal{I})} (\mathcal{L}(\mathcal{I}) - j) = k\mathcal{L}(\mathcal{I}) + \mathcal{L}(\mathcal{I})(\mathcal{L}(\mathcal{I}) - 1). \qquad (1)$$

Solving for $\mathcal{L}(\mathcal{I})$ yields the lemma. □

Note that this lower bound holds even if we are not allowed to split jobs into pieces or if the jobs are of different sizes.

2.2 The algorithm

Our algorithm is quite simple, although its analysis is somewhat involved. Each processor sends out a "bucket" of jobs around the ring, in the direction of increasing processor number. In reality of course there are no buckets; rather the processors keep some jobs and send the others along. We will refer to the jobs kept at a processor at a given step as the jobs "dropped off" by the bucket that arrived at that processor at that step. Our algorithm also passes the initial count of jobs on a processor along with the bucket originating at that processor. Thus, at time step i, processor i will know the number of jobs that originated on processors 1 through i.

We describe the bucket that starts from processor 1; the rest are symmetric. The bucket moves continuously to the right and drops off at processor i enough jobs to bring the number of jobs at processor i up to a constant times the square root of the work that originated on processors 1 to i, as this quantity is closely related to the lower bound in Lemma 1.

We now describe our algorithm in detail and justify its performance. We assume at this point that no bucket travels further than m steps, and will later modify the algorithm to handle the case when a bucket travels further than m steps. For simplicity we describe our algorithm in terms of the bucket leaving processor 1; the algorithm from other processors is symmetric. Our algorithm uses a constant c; we leave c unspecified at this point, and will choose its value later when the rationale for the choice is clear.

Basic Communication Algorithm: A bucket B_1, containing the x_1 jobs that initiate at processor 1, leaves processor 1 at time 0, in the direction of higher-numbered processors. On arrival at processor j, the bucket drops off as many jobs as possible subject to the constraint that processor j ends up with no more than $c\sqrt{x_1 + \cdots + x_j}$ jobs, where c is a constant to be determined later. More precisely, let b be the number of jobs currently in the bucket B_1, and a_j the number of jobs that have been dropped off at processor j by previous buckets. Bucket B_1 drops off $\min\{d_j, b\}$ jobs at processor j, where $d_j = c\sqrt{x_1 + \cdots + x_j} - a_j$. The bucket continues to travel in this direction and drop off jobs until it is empty.

In addition, on every time step, every processor that has at least one job processes a job. We call the combination of the basic communication algorithm with this processing rule the *Basic Algorithm*. Note that this is a distributed algorithm; the decisions made by a processor (i.e. the number of jobs to be dropped off at that processor at that time) are based only on information that originates at that processor or that is made known to the processor by the buckets traveling through the ring.

2.3 Analysis

We now show that this algorithm is a ρ–approximation algorithm, for $\rho = 4.22$. For an instance $\mathcal{I}$ recall that $\mathcal{L}(\mathcal{I})$ is the length of the shortest schedule for $\mathcal{I}$, where the schedule is computed optimally with complete global knowledge. This is clearly a lower bound on the performance of any distributed algorithm on $\mathcal{I}$. We will consider all instances $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) = L$ and show that our algorithm terminates in at most ρL time. In fact, we will actually consider a broader class of instances, which contains as a subset all instances $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) = L$, and show that our algorithm terminates in time at most ρL on any instance in the class and thus in particular on any instance with $\mathcal{L}(\mathcal{I}) = L$.

To analyze our performance, we will first upper bound the maximum distance a bucket can travel before emptying, and then upper bound the maximum amount of work a processor can receive; clearly the sum of these two quantities will be an upper bound on the length of the schedule computed by our algorithm. Let us consider some instance $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) = L$, and determine how long bucket B_1 starting with x_1 units of work at processor 1 can travel before emptying. We will restate Lemma 1 in terms of M_k, which we define to be the maximum amount of work a group of k contiguous processors can contain at time 0 if the instance has an optimal schedule of length L.

Lemma 2 *Let M_k be the maximum amount of work that can originate in k adjacent processors in an instance whose optimum schedule has length L. Then $M_k = L^2 + (k-1)L$.*

Proof: Direct from Equation (1). □

We now consider bucket B_1 that starts off with x_1 units of work, and determine the worst case time to empty that bucket (this will clearly apply symmetrically to any other bucket in the instance). We start by lower bounding the amount of work B_1 drops off at any time. Assume for now that the bucket empties in some time less than m, i.e. that the bucket does not travel all the way around the ring. Let $W_k = x_2 + \cdots + x_k$ be the work originating on the $k-1$ processors to the right of x_1, and let $W_1 = 0$. Define

$$V_r = \sum_{k=1}^{r} c(\sqrt{x_1 + W_k} - \sqrt{W_k}). \tag{2}$$

Lemma 3 *If bucket B_1 is not empty at time r then it has dropped off at least V_r work.*

Proof: When B_1 arrives at processor k, we know that processor k has at most $c\sqrt{W_k}$ work. Bucket B_1 then either becomes empty or raises the work at processor k to $c\sqrt{x_1 + W_k}$. In other words, it drops off at least $c(\sqrt{x_1 + W_k} - \sqrt{W_k})$ work at processor k. Summing, we find that it leaves at least V_r work over all processors it has visited up to time r. □

Corollary 1 *If $V_r \geq x_1$ then B_1 is empty at time r.*

Let r^* be the minimum r such that $V_r \geq x_1$. It follows from Corollary 1 that r^* is an upper bound on the time for B_1 to empty. We wish to upper bound the maximum distance a bucket can travel before emptying, and it therefore suffices to upper bound r^*. To do so, consider any instance $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) \leq L$. From Lemma 2, since W_k is just the work on $k-1$ consecutive processors, we know that

$$\begin{aligned} W_k &\leq M_{k-1} = L(k-1+L) - L \\ x_1 + W_k &\leq M_k = L(k-1+L) \end{aligned} \tag{3}$$

We proceed to upper bound r^* over all sets of values W_k satisfying the above constraints. Since any instance with optimum schedule length L yields values W_k satisfying the constraints, our upper bound will certainly apply to all instances with optimum schedule length L.

To do so, we consider an *adversary* who wishes to choose x_1 and $\{W_k\}$ so as to maximize r^*. We determine his optimal choices subject to the constraints (3) and prove that even with these optimal choices, r^* is not large. Since every instance with optimum schedule length L yields values W_k that obey the constraints (3), this will prove that r^*, which upper bounds the time to empty every bucket, is small.

Lemma 4 *For any $x_1, W_k \geq 0$, $\sqrt{x_1 + W_k} - \sqrt{W_k}$ decreases as W_k increases.*

Proof: This can be easily verified by differentiating $\sqrt{x_1 + W_k} - \sqrt{W_k}$ with respect to W_k and noting that this derivative is always negative. □

Lemma 5 *If $x_1 \leq L$, the adversary maximizes r^* by choosing $W_k = M_{k-1}$ for all k. For $x_1 \geq L$, the adversary maximizes r^* by choosing $x_1 + W_k = M_k$ for all k.*

Proof: To maximize r^* the adversary would like to make each term of the sum V_r as small possible, since then r^*, the number of terms needed to bring the overall sum to x_1, will increase. Lemma 4 shows that any individual term is minimized by maximizing W_k. Recall, however, that the adversary is restricted to choosing values W_k satisfying the upper bounds in Equation (3). The first constraint is a tighter upper bound when $x_1 \leq L$, and the second is a tighter upper bound when $x_1 \geq L$. Since the adversary can set W_k to the maximum allowed by Equations (3) without affecting its ability to similarly set W_{k+1} to the maximum allowed, the adversary maximizes r^* by setting, in turn, each of the W_k to either M_{k-1} or $M_k - x$ depending on the relative values of x_1 and L. □

Having determined how the adversary should set the W_k based on the value of x_1, we now determine the adversary's best choice for x_1.

Lemma 6 *The adversary maximizes r^* by choosing $x_1 = L$.*

Proof: We consider the quantity V_r/x_1. We know that r^* is the smallest r such that $V_r/x_1 \geq 1$. We show that choosing $x_1 = L$ minimizes, simultaneously, V_r/x_1 for all values of r; it follows that taking $x_1 = L$ maximizes r^*.

First we consider values $x_1 \leq L$. We have already shown that in this case the adversary will choose $W_k = M_{k-1}$ for all k. We show that increasing x_1 will decrease V_r/x_1. To see this, note that V_r/x_1 is the sum of terms

$$\frac{c(\sqrt{x_1 + M_{k-1}} - \sqrt{M_{k-1}})}{x_1}$$

(since we set $W_k = M_{k-1}$). If we differentiate this quantity with respect to x_1, we see that it is decreasing as x_1 increases. Since each term decreases with increasing x_1, we see that V_r/x_1 decreases with increasing x_1. Thus, in the range $x_1 \leq L$, we find V_r/x_1 minimized when $x_1 = L$.

Now we consider $x_1 \geq L$. In this case, V_r/x_1 is the sum of terms

$$W_k = \frac{c(\sqrt{M_k} - \sqrt{M_k - x_1})}{x_1}.$$

Differentiation shows that each term is now decreasing with *decreasing* x_1. Thus in the range $x_1 \geq L$, we again find that V_r/x_1 is minimized when $x_1 = L$. □

Lemma 7 $r^* \leq (2/c + 1/c^2)L$.

Proof: It follows from the previous lemma that the adversary will maximize r^* by setting $x_1 = L$ and accordingly setting $W_k = M_{k-1}$ for all k. This defines our values V_r, and r^* is the smallest value of r for which $V_r \geq x_1$. Taking the adversarial $x_1 = L$ and expanding V_r, we find that we must find the smallest r for which

$$\sum_{k=1}^{r} c(\sqrt{L(L+k)} - \sqrt{L(L+k-1)}) \geq L$$

The left hand sum telescopes and gives

$$c(\sqrt{L(L+r)} - \sqrt{L^2}) \geq L$$

which, solving for r, yields

$$r^* = (2/c + 1/c^2)L.$$

□

Lemma 8 *Consider any instance $\mathcal{I}$ on which the basic algorithm has no bucket that travels for more than m steps. Then the basic algorithm run on instance $\mathcal{I}$ yields a schedule of length at most $4.22\mathcal{L}(\mathcal{I})$.*

Proof: Let

$$\alpha = (2/c + 1/c^2). \tag{4}$$

The value αL is then an upper bound on the time it takes to empty a bucket in any instance with optimal schedule length L. Given this value, we can determine the maximum amount of work a processor can receive *in any instance* with optimal schedule length L. Suppose a processor received work from j buckets to the left. Since no bucket travels farther than αL before emptying, we know $j \leq \alpha L$. Then the amount of work seen by processor j is at most M_j, which means that the amount of work the processor keeps is at most $c\sqrt{M_j}$. This is clearly maximized by maximizing j, *i.e.* taking $j = \alpha L$. The amount of work received is then at most $c\sqrt{L(L+\alpha L)} = cL\sqrt{1+\alpha}$.

We now have the information needed to determine an upper bound, *for any instance* with optimal schedule length L, on the overall length of the schedule constructed by our algorithm. We know that, for any instance with optimal schedule length L, at time αL, all the buckets are empty. At this time, each processor will have all of the at most $cL\sqrt{1+\alpha}$ work it is going to receive, and will process it in that amount of time. Therefore, the total time to finish processing all the jobs is at most $(\alpha + c\sqrt{1+\alpha})L$. In other words, our algorithm is an $(\alpha + c\sqrt{1+\alpha})$-approximation algorithm.

Equation (4) gives c as a function of α, thus we can choose c to yield a bound on the algorithm of

$$\begin{aligned} \alpha + c\sqrt{1+\alpha} &= (2/c + 1/c^2) + c\sqrt{1 + (2/c + 1/c^2)} \\ &= 1 + c + 2/c + 1/c^2. \end{aligned}$$

We choose c so as to minimize the above quantity. Choosing $c = 1.77$ sets $\alpha = 1.45$ and yields a schedule of length $4.22L$. $\square$

We need now consider the case when some bucket travels all the way around the ring in m time steps. To handle this case we must modify the basic algorithm slightly. Intuitively, if a bucket goes all the way around the ring then there is so much work in the system that the time spent thus far is small compared to the amount of processing that still needs to be done. The modification of the basic algorithm will be described in detail in the proof of Lemma 9; from now on it is this modified algorithm which we mean when we refer to the basic algorithm.

Lemma 9 *Consider any instance $\mathcal{I}$ on which the basic algorithm has a bucket that travels for more than m steps. Then the (modified) basic algorithm run on instance $\mathcal{I}$ yields a schedule of length at most $4.22\mathcal{L}(\mathcal{I})$.*

Proof: First, note that a trivial lower bound on the length of the optimal schedule is just the total number of jobs in the network divided by the number of machines. Observe that once a bucket goes all the way around the ring it knows all the work in the system. Thus it knows the average number of jobs on each processor. So each processor can keep that average and send its excess around the ring so as to bring the number of jobs at each processor to the average. Thus after m time to go around the ring once and m time to balance the work, each processor has at most L work and we can bound the schedule length by $2m + L$.

Now recall that the distance any bucket travels is at most αL. Thus, if a bucket travels all the way around the ring we know that $m \leq \alpha L$, and therefore $L \geq m/\alpha$. The ratio of the distributed schedule length to the optimal schedule length is then $(2m + L)/L = 1 + 2m/L \leq 1 + 2\alpha$. Given our choice of $\alpha = 1.45$ this gives a bound of 3.90 which is less than 4.22. □

Combining Lemma 8, which covers the case when no bucket travels more than m steps, and Lemma 9, which covers the case when some bucket travels more than m steps, and noting that each Lemma is true when $c = 1.77$, we obtain our main result.

Theorem 1 *If $\mathcal{I}$ is an instance of the basic problem and $c = 1.77$, the Basic Algorithm returns a schedule that is of length at most $4.22\mathcal{L}(\mathcal{I})$.*

Our experimental results, to be discussed in Section 5, indicate that this analysis is potentially not tight; the worst performance we were able to generate for the basic algorithm was 2.87 times optimal.

3 The Integral Algorithm

We now dispense with the unrealistic assumption that a job can be split into fractional pieces and instead require that each job be processed entirely by one processor. Let $d_{i,j}$ be the amount of work dropped off in the basic algorithm by bucket B_i on processor j. Let t_i be the time that it takes bucket B_i to empty in the basic algorithm and let r'_{ij} be the number of jobs remaining in bucket B_i when it arrives at processor j in the integral algorithm. Our integral algorithm will simulate the basic algorithm, but instead B_i will drop off $\min\left\{\lceil d_{ij}\rceil, r'_{ij}\right\}$ jobs on processor j, as opposed to the potentially fractional d_{ij}. Note that this simulation requires that each processor and bucket keep track of what they would have done in the basic algorithm, in order to know what d_{ij} is, and hence what $\lceil d_{ij}\rceil$ is, upon arriving at processor j.

Lemma 10 *Given an instance $\mathcal{I}$, assume that the basic algorithm runs for $\rho\mathcal{L}(\mathcal{I})$ steps. Then the integral algorithm runs for no more than $\rho\mathcal{L}(\mathcal{I})+1$ steps.*

Proof: Assume without loss of generality that B_1 is the last bucket to empty in the integral algorithm. B_1 will still empty by time t_1 in the integral algorithm, since at any time, bucket B_1 has dropped at least as much in the integral algorithm as it has in the basic algorithm. Thus, it will empty no later than it did in the basic algorithm.

Now we need to show that no processor will complete its work much later in the integral algorithm than in the basic algorithm. Notice that in the analysis of the basic algorithm we did not take into account any overlapping of processing and communication. We will now overlap the two, and require that whenever a processor receives an extra job (as the result of rounding) it processes that extra job on the next step. Thus, at any time a processor can have at most 1 extra unprocessed job. In particular, at time t_1, each processor has at most one additional unprocessed job more than it did in the basic algorithm. Thus, no processor will finish more than one time unit later in the integral algorithm than in the basic algorithm. □

Corollary 2 *The integral algorithm is a 4.22-approximation algorithm.*

4 A Lower Bound

In this section we prove that there is no distributed ρ-approximation algorithm for the basic problem (as defined in Section 2) with $\rho \leq 1.06$. This is admittedly a weak constant; the major significance of this result is to show that no distributed algorithm for this problem can be asymptotically optimal. We will assume we have a distributed 1.06-approximation algorithm A and show that it must do worse than 1.06 times optimal for one of two different classes of problem instances.

The first instance will be one in which all the work in the system is placed on one node. The optimal algorithm for this case is to keep the square root of the work and send the remainder of the work equally in each direction. In order for algorithm A to perform well in this case it must do something similar to the optimal solution. The second instance will be one in which all the work in the system is split between two different processors. In order for algorithm A to perform well on this instance it must send somewhat less work towards the other processor and more towards the outside of the region between the two processors.

We will prove the lower bound by exploiting the inability of a distributed algorithm to distinguish between the two cases for some time; therefore A must do something similar to the optimal schedule for the first instance. However, the work from the two processors will collide at some point, but by then there will be too much work in a small region to process as fast as is necessary for A to be a 1.06-approximation algorithm.

Let $\rho = 1.06$ and assume A is a ρ-approximation algorithm. Let W and z be parameters which we will set later. The two problem instances we consider are:

- $\mathcal{I}_1$: Processor number i, which we will denote by p_1, receives W unit-size jobs at time 0, and all other processors receive no jobs.
- $\mathcal{I}_2$: W unit-size jobs are placed on p_1 and W unit-size jobs on processor number $i + 2z + 1$, which we will denote by p_2. We will refer collectively to processors number $i + z$ and $i + z + 1$ as the "midpoint."

For both instances the ring size m is the same and is chosen large enough so that $m - (2z + 1) >> \mathcal{L}(\mathcal{I})$, i.e. no jobs will go around all the way around the ring.

We first lower bound the optimal schedule length for each instance. We know from Lemma 1 that the optimal schedule length for $\mathcal{I}_1$ is at least $\sqrt{W}$. We now establish the length of the optimal schedule for instance $\mathcal{I}_2$.

Lemma 11 *For instance $\mathcal{I}_2$ the maximum amount of work done by any algorithm by time $u > z$ is $2u^2 - (u-z)^2 + u - z$. Specifically, the shortest schedule for $\mathcal{I}_2$ is of length t, where t satisfies*

$$2W \;=\; 2t^2 - (t-z)^2 + t - z. \tag{5}$$

Proof: At time 0, only processors p_1 and p_2 can process any work. For each of the next z time steps, an additional four processors can receive work, thus the total number of jobs processed in time 0 through z is $\sum_{i=0}^{z} 2 + 4i$. At this point, $4z + 2$ processors have work. After time z work has reached the midpoint from both sides, hence for each subsequent step only two additional new processors – those on the "outside" – can receive work. Thus the total amount of work processed in u steps, where $u > z$ and no work goes around the ring, is at most

$$\begin{aligned}
&\sum_{i=0}^{z}(2+4i) + \sum_{i=z+1}^{u-1}(4z+2+2(i-z)) \\
&= \sum_{i=0}^{z}(2+4i) + \sum_{i=z+1}^{u-1}(2z+2) + \sum_{i=1}^{u-1}(2i) - \sum_{i=1}^{z}(2i) \\
&= 2(z+1) + \frac{4(z)(z+1)}{2} + (2z+2)(u-z-1) + \frac{2(u-1)u}{2} - \frac{2z(z+1)}{2} \\
&= u^2 + 2zu - z^2 + u - z \\
&= 2u^2 - (u-z)^2 + u - z.
\end{aligned}$$

This establishes the first part of the lemma. To establish the second part of the lemma, we substitute t for u and observe that the total work in $\mathcal{I}_2$ is $2W$. $\square$

Now we will exploit our freedom to choose z and W. For any z, and for a constant ϵ, which we will choose later, we will choose the value of W by the formula

$$W = \frac{1-\epsilon^2/2}{(1-\epsilon)^2} z^2 + \frac{\epsilon}{2(1-\epsilon)} z. \tag{6}$$

Plugging this value for W into (5) and solving for t we see that $t = \frac{1}{1-\epsilon}z$. Now choosing $z \geq 11$ and $\epsilon = 0.7$, we see that $t \geq 35$. Plugging the identity $t = \frac{1}{1-\epsilon}z$ into (6) and applying the bounds on t and ϵ, we get that

$$W = \frac{1-\epsilon^2/2}{(1-\epsilon)^2}z^2 + \frac{\epsilon}{2(1-\epsilon)}z = (1-\epsilon^2/2)t^2 + \frac{\epsilon}{2}t \leq (1.01 - \frac{\epsilon^2}{2})t^2 \leq .765t^2. \tag{7}$$

To summarize, we can choose $\mathcal{I}_2$ so that $W \leq .765t^2$, $t = \frac{10}{3}z$, $z \geq 11$ and $t \geq 35$.

Now consider the operation of algorithm A on $\mathcal{I}_1$. Since A is a ρ-approximation algorithm, it must finish by time $\rho\sqrt{W}$. Plugging in the bound from (6), we see that the algorithm must finish by time

$$u = \rho\sqrt{.765t^2} = .875\rho t. = 2.92z$$

In other words, by time $u = 2.92z$ no processor has any remaining jobs.

Now we consider $\mathcal{I}_2$. The key idea is that by time u, the processors that cannot distinguish whether the input is $\mathcal{I}_1$ or $\mathcal{I}_2$ will have to finish all their work. We formalize this in the following lemma:

Lemma 12 *Consider the performance of algorithm A on instance $\mathcal{I}_2$. If $u > z + 20$ then there are at least $V(u) = 2W - 2u^2 + \frac{19}{20}(u-z)^2$ units of unprocessed work remaining in the system, all of which are within distance $u - z$ of the midpoint, i.e. within a region of width $2(u-z)$.*

Proof: What is the state of the system at time u? It takes at least z time for any information from p_1 to meet any information from p_2 at the midpoint of p_1 and p_2. From time z on, processors farther and farther from the midpoint can learn about the work from p_i on the other side of the midpoint. However, at time u, any processor p at distance greater than $u - z$ from the midpoint must be in the same configuration that it would if the instance was $\mathcal{I}_1$, since it cannot yet have learned of the work from the other side of the midpoint.

Applying Lemma 11 gives that the amount of work that could have been processed up to time u is $2u^2 - (u-z)^2 + u - z$. Therefore, at time u there are at least $V(u) = 2W - 2u^2 + (u-z)^2 - u + z$ remaining unprocessed jobs. $V(u) \geq 2W - 2u^2 + \frac{19}{20}(u-z)^2$ as long as $\frac{1}{20}(u-z)^2 \geq (u-z)$ which is true for $(u-z) > 20$.

Since processors that are not within $u - z$ of the midpoint must be in the same configuration at time u for both $\mathcal{I}_2$ and $\mathcal{I}_1$, and these processors must have completed processing by time u in $\mathcal{I}_1$, all of the unprocessed work must be within distance $u - z$ of the midpoint, i.e. within a region of width $2(u-z)$. $\square$

For $u > z + 20$, let $q(u)$ be a lower bound on the time needed to process the remaining $V(u)$ units of work. Then A must take $u + q(u)$ units of time to complete instance $\mathcal{I}_2$. We can use Lemma 1 to calculate $q(u)$ knowing that there are $V(u)$ jobs in a region of $2(u - z)$. In particular,

$$\begin{aligned} q(u) &= \sqrt{\frac{(2(u-z))^2}{4} + 2W - 2u^2 + \frac{19}{20}(u-z)^2} - \frac{2(u-z)}{2} \\ &= \sqrt{\frac{39}{20}(u-z)^2 + 2W - 2u^2} - u + z \end{aligned}$$

Therefore,

$$u + q(u) = z + \sqrt{\frac{39}{20}(u-z)^2 + 2W - 2u^2}$$

Now we set $u = 2.92z$, as that is the time that $\mathcal{I}_1$ must have completed, and substitute into the above equation. A simple calculation shows that $u = 2.92z > z + 20$ when $z > 11$, so the above bound applies.

$$u + q(u) = z + \sqrt{\frac{39}{20}(1.92z)^2 + 2W - 2(2.92z)^2}. \tag{8}$$

We can lower bound W as a function of z^2 using equation (6),

$$W = \frac{1 - \epsilon^2/2}{(1-\epsilon)^2} z^2 + \frac{\epsilon}{2(1-\epsilon)} z \geq \frac{1 - \epsilon^2/2}{(1-\epsilon)^2} z^2.$$

Substituting $\epsilon = .7$, we get that $W \geq 8.38z^2$. Plugging this into (8) and simplifying, we get that the algorithm finishes no earlier than time

$$u + q(u) \geq 3.62z = 3.62\frac{3}{10}t \geq 1.08t$$

which contradicts our assumption that A is a ρ-approximation algorithm for some $\rho \leq 1.06$. Thus, we have shown the following:

Theorem 2 *There does not exist a distributed ρ-approximation algorithm for ring scheduling for any $\rho \leq 1.06$.*

5 Experimental Work

5.1 Approach

We performed simulations of three similar algorithms (A,B and C) for scheduling unit sized jobs; each algorithm drops off only integral amounts of work. Algorithm C is the integral algorithm which we have shown above to be a 4.22-approximation algorithm. Recall that in algorithm C a bucket drops off work so as to bring a processor up to a constant times the square root of the work it knows about in the system. However, the square root of the work is not actually a lower bound on the optimal schedule length; the actual lower bound is given by Lemma 1. Algorithm B is a variant of our algorithm in which buckets drop off jobs so as to bring the work at a processor up to the best lower bound the bucket knows based on Lemma 1. Algorithm C proved easier to analyze and is slightly simpler than B, but one might expect B to be a better algorithm. Algorithm A represents our initial idea for solving this problem and is included for comparison purposes. In algorithm A, a processor removes jobs from the bucket passing by on step i so as to have the square root of the work that has *passed by* in the previous i buckets. This is in contrast to algorithm C in which each processor removes jobs from the bucket passing by on step i so as to have the square root of the work that *originated* on the previous i processors.

Each algorithm was simulated with passing in one direction (A1,B1,C1), as in our analysis, as well as passing jobs in two directions (A2, B2, C2). For passing jobs in both directions we just split the work that would go in a bucket evenly and send a bucket in each direction.

We tested the algorithms on a variety of examples. Most of our tests were generated according to three parameters: ring size, the number of jobs on the heavily loaded processors, and distribution. The ring sizes ranged from ten to a thousand nodes; the number of jobs on the heavily loaded processors ranged from a thousand to a hundred thousand; and the distributions were heavily concentrated on one node or a region of nodes. In the heavily concentrated cases we had variants where the remainder of the ring was empty as well as randomly loaded according to a uniform distribution. A precise listing of these test cases appears

I) Structured Test Cases

Ring Sizes	Distributions	The number of jobs on each heavily loaded processor
10	1) Concentrated on one node, zero elsewhere	Huge = 100,000
100	2) Concentrated in a region, zero elsewhere	Large = 10,000
1000	3) Concentrated on one node, rand(100) elsewhere	Big = 1,000
	4) Concentrated in a region, rand(100) elsewhere	

II) Random Test Cases

Ring Sizes	Distributions
10	Random 0 - 100 per processor
100	Random 0 - 500 per processor
1000	Random 0 - 1000 per processor

III) Evil Adversary Test Cases

Ring Size	Adversary's choice of the lower bound, L	Adversary's choice of the region size, k
1000	100	20
	500	40
		80

Table 1: The three different input distributions

in part I of Table 1. We tested each algorithm on all 36 combinations of these parameters. We also tested all the algorithms on nine uniform random load distributions. For these cases we used random loads drawn uniformly from 0 to 100, 500, and 1000 per processor and varied the size of the ring as in the previous examples. The uniform random load cases are detailed in part II of Table 1. Finally, we used six more test cases which correspond to a distribution that the "evil adversary" from Section 2 might construct. Notice that the evil adversary can construct any one of a variety of instances and still follow the strategy that we laid out. For example, the adversary is allowed to pick any lower bound, L, and also can pick any size region, k, in which to place work. Once the adversary picks these two parameters for a region then the amount of work placed on each processor is determined by the arguments we presented. Note the adversary only places work in a single region of size k for each instance. Our six evil-adversary test cases varied these two parameters; the details are in part III of Table 1. This amounts to 51 test cases in all.

5.2 Computing the Optimal Schedule Length

In order to obtain empirical factors of approximation we need to compare the lengths of the schedules produced by our algorithms to the optimal schedule length. For each instance, we computed the length of the optimal centralized schedule (a centralized scheduler is given global knowledge of the system). It was previously known how to compute the optimal schedule length for an instance with unit size jobs [11] [17]. The former of these two approaches, due to Deng. *et al* [11], formulated the problem as a minimum cost flow problem. Their algorithm is difficult to implement because it requires the use of a very wide range of costs. For instance, an instance with a thousand machines would require that the costs range from 1 to 100^{1000}. Even if we shifted the values so that the largest value is representable then the smallest value is too small to be represented. The latter of the approaches, due to Hoppe and Tardos [17], requires the use of a polynomial time procedure for convex function minimization as a subroutine [35]. We developed a different polynomial time algorithm that uses less than m^3 space and only requires a maximum flow subroutine. This approach made the problem more tractable than the algorithm of Deng *et al.* [11] and easier to code than Hoppe and

Tardos' algorithm [17].

The algorithm we implemented is a polynomial time algorithm that works for general graphs with infinite capacity links. We begin by describing a variation of this algorithm and then discuss the necessary refinements to make it run in polynomial time. We will use notation similar to what we have used thus far. Namely, we have m machines or processors labeled $1, \ldots, m$ and n jobs $j_1, \ldots, j_n$.

This algorithm is a decision procedure. For a given time T it decides whether or not a schedule of length T exists. Then binary search is employed to find the minimum T under which the schedule completes. First, we construct a bipartite graph $G = (V \cup W, E)$ where the n vertices on the left, V, represent the set of jobs, J, and the mT vertices on the right, W, represent machines at various times. Let vertex $v_i \in V$ represent job j_i, and vertex $w_{p,k} \in W$ represent processor p at time $k \leq T$. An edge is placed between v_i and $w_{p,k}$ if job j_i could be processed by processor p at time k. Now we perform a maximum bipartite matching on G. A matching of cardinality n corresponds to a schedule that will process all of the jobs by time T. If there is no matching of cardinality n then there is no way to process all the jobs given T units of time. As we said above, we use binary search to find the smallest value of T for which all the jobs finish. Notice that the actual schedule can be extracted from the matching in the following way. An edge in the matching from vertex v_i to vertex $w_{p,k}$ corresponds to processing job j_i on processor p at time k. This gives a correct schedule because no two jobs will be processed by the same machine at the same time step. Similarly, each job is processed if the matching has cardinality n.

A well-known method for solving maximum cardinality bipartite matching uses maximum flow. Connect a source, s, to each vertex in V and connect each vertex in W to a sink, t. The edges between V and W are as described above. Finally, give every edge unit capacity. (see [1] for details). This leads to an algorithm that is very simple to understand and code but unfortunately it could require nmT space which is far too large for the large instances in Table 5. We refer to this procedure as FIRST-OPT.

We now refine FIRST-OPT in order to reduce the space and time needed by the algorithm. The intuition for the first refinement comes from the following observation. Consider a collection of vertices in V that correspond to jobs that originate on the same machine. Notice that the edges emanating from these vertices go to exactly the same vertices in W. We can reduce this redundancy by just having m vertices in V. Each vertex, v_i, now denotes the set of jobs that originated on processor p_i. The only other modification we need to make is that the capacity of each edge of the form (s, v_i) is set to the number of jobs originating on processor p_i. Now we are searching for the minimum T under which we get a flow of value n. Again this schedule is correct because a flow of value n corresponds to processing all n jobs and no machine can process more than one job per time step because each edge from a vertex in W to t has capacity one. We will refer to this approach as MODIFIED-OPT.

Notice that MODIFIED-OPT as stated requires $O(m^2T)$ space. If T is larger than m then we can reduce the space needed in the following way. Notice that if the time, t, is greater than the diameter of the network, δ, then there is an edge between every vertex in V and every vertex $w_{p,t}$. Thus there is no need to even include nodes $w_{p,t} \in W$ for any $t > \delta$. Once we determine that the maximum flow up to time δ is f then we can compute the amount of time needed to finish the schedule by just calculating the quantity $\lceil (n-f)/m \rceil$ since any job can be processed on any processor at this point. Let $S = \min(\delta, T)$, and notice that $\delta < m$ for any graph. Thus, Theorem 3 shows that MODIFIED-OPT with this final refinement is a polynomial time algorithm.

Theorem 3 *We can compute the optimal schedule length for an instance with n jobs and m machines in $O(m^2S)$ space and $O(MF(mS, m^2S)\log n)$ time, where $MF(a,b)$ is the running time of maximum flow on a graph with $O(a)$ nodes and $O(b)$ edges.*

We make a brief note that computing the optimal schedule length even with all the refinements discussed above was still not possible for the largest instances (1000 node rings). To compute the optimal schedule length for these instances we used a version of MODIFIED-OPT that did not store any edges explicitly (because 10^9 edges were too many for our machines). Instead, we recomputed whether an edge was present every time we needed to check. This trick let us compute, albeit very slowly, the optimal schedule lengths for the remainder of the instances.

5.3 Analysis

The simulation results show that the algorithms perform better than our analysis suggests. Figures 1 through 6 give histograms of the approximation factors found in our test cases. The height of each bar of the histogram corresponds to the frequency that we obtained the approximation factor in the range specified by the x axis. For example, the leftmost bar in each figure corresponds to the number of times we obtained an approximation factor between 1.0 and 1.2.

Algorithm C1, for which we showed a worst case upper bound of 4.22 in Section 2, performed no worse than 2.87 times optimal in the simulations. Also note that Figure 5 shows that many of the experiments for C1 had an approximation factor of 1.2 or less. The instances that provided the worst performance for C1 were the evil adversary instances. The factors for these instances increased with increasing k but dis not surpass 2.87 for any value of k. Other instances on which C1 performed poorly were the instances with heavily loaded regions.

For Algorithm A, we noticed almost a factor of two improvement in the bidirectional version of the algorithm, A2, over the unidirectional version, A1. Furthermore, A2 was the best algorithm empirically. It performed no worse than 1.64 times optimal in any of the 51 test cases. We speculate that algorithm A does the best because it performs slightly better local load balancing, as it works only with the jobs it sees going by.

In general, the bidirectional versions of the algorithms were somewhat better than the unidirectional versions, but this feature did not usually offer improvements of a factor of 2. Note also that Algorithm B performed the worst despite working with a more accurate estimation of the lower bound.

One possible reason that the experimental work would indicate approximations of a higher quality than Theorem 1 is that, in addition to the inherent nature of worst-case analysis, our theoretical analysis does not take into account the potential overlap of the dissemination of work and the processing of work except in Lemma 10. In the simulations we overlapped processing and communication at every opportunity.

6 Conclusions and Open Problems

In this paper we have given a constant-approximation algorithm for scheduling jobs in a ring of processors. The algorithm and its variants are extremely simple and do not rely on global knowledge or centralized control. Simulations of this algorithm and related algorithms suggest that its worst-case performance guarantee may be better than what is shown in Section 2.

An interesting open problem is whether simple, small-constant approximation algorithms which require no centralized control exist for the other networks, such as the mesh. As stated earlier, Awerbuch et al. [6] give a distributed algorithm for job scheduling in general networks. When applied to the mesh their algorithm is a constant-approximation algorithm. It is an interesting open problem to design a distributed approximation algorithm using simple control structures that does significantly better than this, possibly by adapting the approach presented here.

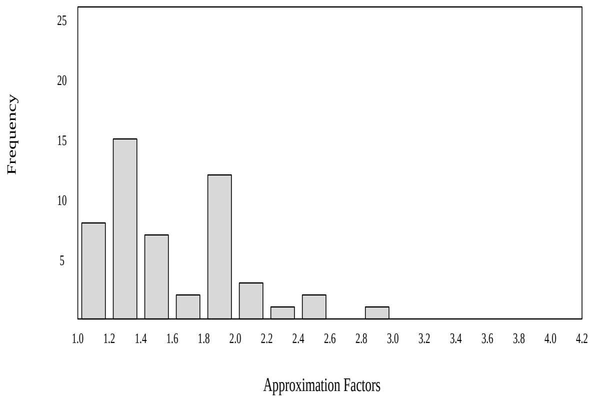

Figure 1: Approximation factors for 51 runs of A1

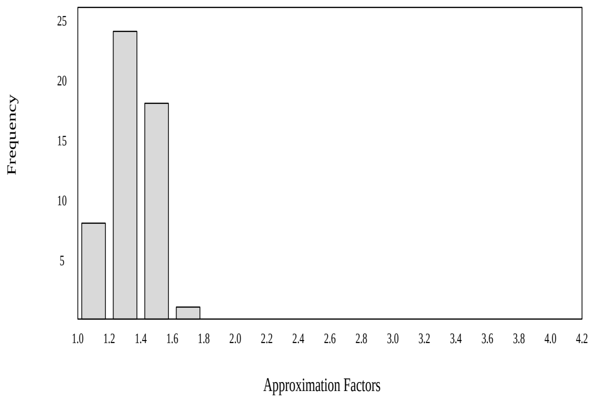

Figure 2: Approximation factors for 51 runs of A2

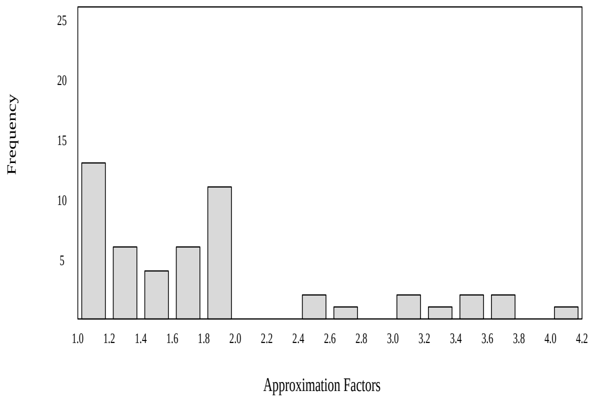

Figure 3: Approximation factors for 51 runs of B1

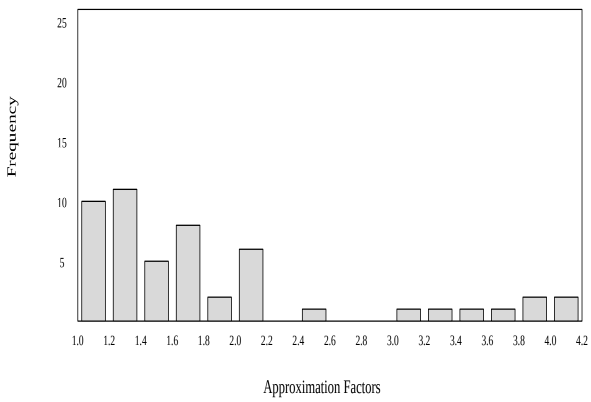

Figure 4: Approximation factors for 51 runs of B2

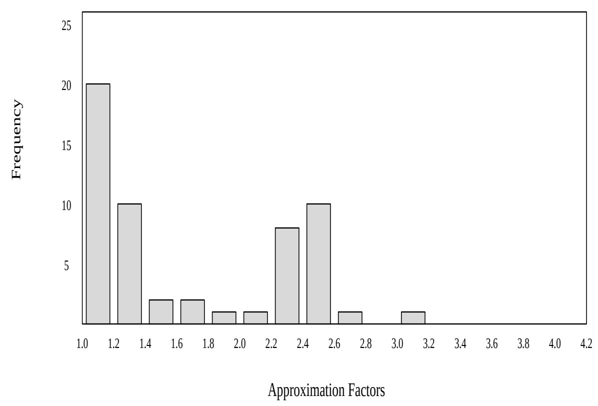

Figure 5: Approximation factors for 51 runs of C1

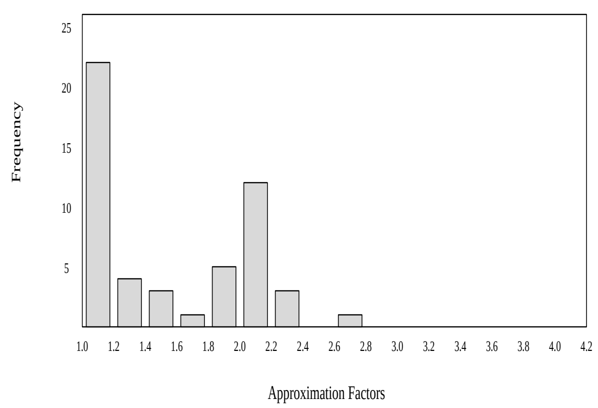

Figure 6: Approximation factors for 51 runs of C2

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows.* Prentice Hall, 1993.

[2] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 632–641, 1993.

[3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[4] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 334–343, 1992.

[5] H. Attiya, M. Snir, and M. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, 1988.

[6] B. Awerbuch, S. Kutten, and D. Peleg. Competetive distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 571–580, 1992.

[7] L. Bhuyan, D. Ghosal, and Q. Yang. Approximate analysis of single and multiple ring networks. *IEEE Transactions on Computers*, 38:1027–1040, 1989.

[8] J. E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[9] H. Choi and A. Esfahanian. A message routing strategy for multicomputer systems. *Networks*, 22:627–646, 1992.

[10] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 2:279–301, 1989.

[11] X. Deng, H. Liu, J. Long, and B. Xiao. Deterministic load balancing in computer networks. In *Proceedings of 2nd IEEE Symposium on Parallel and Distributed Processing*, 1990.

[12] B. Ghosh and S. Muthukrishnan. Dynamic load balancing on parallel and distributed networks by random matchings. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 226–235, 1994.

[13] Bhaskar Ghosh, S. Muthukrishnan, and M.H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 72–81, 1996.

[14] M. Herlihy, B. Lim, and N. Shavit. Low contention load-balancing on large-scale multiprocessors. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 219–227, 1992.

[15] Dorit Hochbaum, editor. *Approximation Algorithms*. PWS, 1997. to appear.

[16] Y. Hong and T. Payne. Parallel sorting in a ring network of processors. *IEEE Transactions on Computers*, 38:458 – 464, 1989.

[17] B. Hoppe and E. Tardos. The quickest transhipment problem. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[18] S. Flynn Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5/6):743–765, 1991.

[19] S. Flynn Hummel, E. Schonberg, and L. E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Comm. of the ACM*, 35(8):90–101, 1992.

[20] D. Hutchinson. *Local Area Network Architectures*. Addison-Wesley, 1988.

[21] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, *Handbooks*

in Operations Research and Management Science, Vol 4., Logistics of Production and Inventory, pages 445–522. North-Holland, 1993.

[22] F. T. Leighton. *An Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, CA, 1991.

[23] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.

[24] Y. Mansour and L. Schulman. Sorting on a ring of processors. *Journal of Algorithms*, 11:622–630, 1990.

[25] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–514, 1988.

[26] C. A. Phillips, C. Stein, and J. Wein. Task scheduling in networks. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, July 1994.

[27] C. A. Phillips, C. Stein, and J. Wein. Task scheduling in networks. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, July 1994.

[28] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Computers*, 12:1425–1439, 1987.

[29] J. Rothnie. Kendall square research introduction to the ksr1. In *Dartmouth Institute for Advanced Graduate Studies in Parallel Computation*, pages 200–210, 1992.

[30] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, 20:853–868, 1994.

[31] W. Sung and S. Mitra. Multiprocessor implementation of recursive least squares algorithms using a parallel block processing method. In *IEEE International Symposium on Circuits and Systems*, pages 2939–2942, 1991.

[32] W. Sung, S. Mitra, and K. Kum. Mapping locally recursive sfgs upon a multiprocessor system in a ring network. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 560–573, 1992.

[33] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1989.

[34] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, U.K., 1994.

[35] P. M. Vaidya. A new algorithm for minimizing convex functions over convex sets. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 338–343, 1989.

Professional Biographies

Perry Fizzano is an Assistant Professor in the Department of Mathematics and Computer Science at the University of Puget Sound. He holds a Ph.D. and an M.S. in Computer Science from Dartmouth College (1995,1993) and a B.S. in Computer Science from Widener University (1990). His research interests include scheduling, approximation algorithms, parallel and distributed computation and computational biology.

David Karger is an Assistant Professor at the Massachusetts Institute of Technology. He holds a B.A. in Physics and Computer Science from Harvard University (1989), a certificate of advanced Study in Mathematics from Cambridge University (1990), and a PhD. in Computer Science from Stanford University (1994). He was a postdoctoral fellow at AT&T Bell Labs in the fall of 1994, and joined the faculty of MIT in January 1995. His research interests include analysis of algorithms, with an emphasis on applications of randomization; combinatorics and combinatorial optimization; and information retrieval.

Cliff Stein is an Assistant Professor in the Department of Computer Science at Dartmouth College. He holds a Ph.D. in Computer Science from the Massachusetts Institute of Technology (1992), an M.S. in Computer Science from the Massachusetts Institute of Technology (1989) and a B.S.E. in Computer Science from Princeton University (1987). His research interests include combinatorial optimization, computational biology and scheduling.

Joel Wein is an Assistant Professor in the Department of Computer and Information Science, Polytechnic University, Brooklyn, NY. He holds a Ph.D. in Mathematics from the Massachusetts Institute of Technology (1991) and a B.A. in Applied Mathematics and Computer Science from Harvard University (1985). His research interests are in combinatorial optimization, scheduling theory, and the theory and practice of parallel and distributed computing.