Job Scheduling in Rings

Perry Fizzano*
Department of Math and CS
Dartmouth College
Hanover, NH

David Karger†
Department of Computer Science
Stanford University
Stanford, CA

Clifford Stein‡
Department of Math and CS
Dartmouth College
Hanover, NH

Joel Wein§
Department of Computer Science
Polytechnic University
Brooklyn, NY

Abstract

We give distributed approximation algorithms for job scheduling in a ring architecture. In contrast to almost all 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 algorithms must balance both computational load and communication time.

The algorithms are simple, require no global control, and work in a variety of settings. All come with small constant-factor approximation guarantees; the basic algorithm 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 give better results than our worst-case analysis.

*Research partially supported by NSF grant CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award. email: perry@cs.dartmouth.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. email: karger@cs.stanford.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

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 many parallel systems there is competition among a number of processes for the resources of the system, such as processing power or communications bandwidth. In order to achieve maximum performance, we must develop algorithms which 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*, in the setting in which the processing units are configured in a ring. 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 [2, 10, 11, 18] or in the use of a parallel system to process batches of transactions or independent sequential programs. We restrict our attention to the ring, 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 [3, 5, 9, 15, 16, 20, 21]. In practice the ring is either the basis of or an essential component of many parallel and distributed architectures [12, 13, 22, 23, 19].

The element of our model which most differentiates it from previous scheduling algorithms is that task migration from one processor to another takes time proportional to the distance between those two processors in the communications network. Our contribution is to give simple, robust algorithms that produce schedules of length within a small constant factor of the best possible schedule, *without using any centralized knowledge or control.* Our algorithms work in a variety of settings, including unit-sized jobs, jobs of different sizes, and any ratio of job size to communication time. For unit-sized jobs, our algorithm produces schedules of length no more than a factor of 4.22 times optimal, and for all the other settings it produces schedules no longer than 5.22 times optimal. We also report on simulations of our algorithm, which yield significantly better results than the worst-case analysis.

There is a wealth of literature on parallel machine scheduling (see [14] for some examples) but almost all of it fails to capture the full complexity of many real scheduling problems. Many of the proposed algorithms have significant control overhead. 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 algorithm has low control overhead, accounts for communication constraints and does not require global control.

We know of only one paper which fully addresses the latter two issues: that of Awerbuch, Kutten and Peleg [4], who study the problem of distributed dynamic job scheduling in general distributed networks. Due to the generality of the setting they give algorithms with performance guarantees that are polylogarithmic in the problem size. Despite the importance of this work the bounds are quite loose when applied to specific networks that arise in practice. Therefore it is an important question to develop and precisely analyze scheduling algorithms that exploit the structure of specific real networks.

We know of no previous papers which give algorithms for this distributed ring scheduling problem. The general techniques of [4] can be applied to the ring to yield a constant-approximation algorithm. However, even when a number of refinements to their approach and analysis are incorporated in this algorithm, the performance guarantees of our algorithm are much better. In addition, the control structures of our algorithm are much simpler than the application of their general approach to the ring.

Other papers have included communication cost in basic scheduling models, and, while not applicable to our problem, are related to our work in spirit. Deng et. al. [7] seem to be the first to have studied the parallel machine scheduling problem in a network. They give a number of centralized off-line algorithms, in addition to several distributed algorithms for special cases and models. Phillips, Stein and Wein [17] improved on their results by giving a centralized off-line 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.

The rest of this paper is organized as follows. In Section 2 we precisely define the problem. In Section 3 we present an algorithm for the most basic version of the problem and prove that it is a 4.22-approximation algorithm. In Section 4 we give extensions of the algorithm to more general settings, such as variable job size and non-unit communication time or processor speed. We present a lower bound on the performance of any distributed algorithm in Section 5, and in Section 6 we report on computational experiments with the algorithm. All of these results apply to a model in which the capacity of the communication links in the ring is unbounded; in Section 7 we give an approximation algorithm for the model in which the capacity of each network link is restricted to one job per unit time. We conclude with some open problems in Section 8.

2 Model and Problem Statement

We begin by giving a precise definition of a simple version of the problem, and defer discussion of more general and realistic versions to later sections. 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. 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 each job to be processed on exactly one processor without preemption. In one unit of time we assume that each processor can receive some jobs from each neighbor, send some jobs to each neighbor, and process one unit of work; this is the model of [4], and their assumptions are supported by current technology [6]. If a processor sends a job to a neighbor at time t, the neighbor receives the job at time $t+1$. We assume that the job granularity is large enough so that the time for simple control operations, such as simple arithmetic, is negligible. For most of the paper we assume no bounds on the capacity of each network link in the ring, as in [4]; in Section 7 we consider a model in which each link has unit capacity.

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 $\mathcal{I}$. If algorithm A always yields a schedule of length no more than $\rho\mathcal{L}(\mathcal{I}) + O(1)$ we call A a *ρ-approximation algorithm.*

Note that this problem is related to, but different from, *load balancing*, which is another common problem in parallel and distributed systems (see [1, 8]). 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 do work and the balancing of the work over those neighborhoods.

3 The Basic Algorithm

We begin by describing our algorithm in a simple to analyze but unrealistic scenario. We assume that each job may be broken up into an arbitrary number of pieces, which can then be run on different processors. We will later remove this assumption and modify the algorithm to work in the more realistic setting.

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.

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

$$\mathcal{L}(\mathcal{I}) \geq \sqrt{\frac{(k-1)^2}{4} + x_i + \ldots + x_{i+k-1}} - \frac{(k-1)}{2}.$$

Proof: Assume for convenience that $i = 1$. We focus on the work in processors $1, \ldots, k$ by assuming that there is no other work in the system. Let $\mathcal{L}(\mathcal{I})$ be the minimum amount of time necessary to process these jobs. In the best possible circumstances, processors 1 through k will each process $\mathcal{L}(\mathcal{I})$ jobs in $\mathcal{L}(\mathcal{I})$ time; in addition, processors $k+j$ and $m-j+1$, $1 \leq j \leq \mathcal{L}(\mathcal{I})$, will each process $\mathcal{L}(\mathcal{I})-j$ jobs in $\mathcal{L}(\mathcal{I})$ time, which is the maximum amount of work which can reach them from processors 1 through k in $\mathcal{L}(\mathcal{I})$ time. As a result, $\mathcal{L}(\mathcal{I})$ must satisfy

$$k\mathcal{L}(\mathcal{I}) + \mathcal{L}(\mathcal{I})(\mathcal{L}(\mathcal{I})-1) \geq x_1 + \ldots + x_k \qquad (1)$$

which implies $\mathcal{L}(\mathcal{I}) \geq \sqrt{\frac{(k-1)^2}{4} + x_1 + \ldots + x_k} - \frac{(k-1)}{2}$. □

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

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. Assume a bucket starts from processor 1; it drops off at processor i a number of jobs that brings 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. At any time step in which a processor has at least one job, it will process exactly one job. We will speak of buckets "dropping off jobs", but in reality there are no buckets; rather the processors keep some jobs and send the others along.

We now describe the communication algorithm in detail and justify its performance. For simplicity we describe it in terms of the bucket leaving processor 1. The algorithm from other processors is symmetric.

> **Basic Communication Algorithm:** A bucket B_1, containing the x_1 jobs which initiate at processor 1, leaves processor 1 at time 0, in the direction of higher-numbered processors. 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 up to the current point in time. This bucket drops off at processor j, $\min\{d_j, b\}$ jobs, where $d_j = c\sqrt{x_1 + \cdots + x_j} - a_j$ and c is a constant we will choose later. 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 also assume at this point that no bucket travels further than m steps, and will later modify the algorithm to handle this case. We call the combination of the basic communication algorithm with this processing rule the *Basic Algorithm*.

We now show that this algorithm is a ρ–approximation algorithm, where $\rho = 4.22$. To do so, we will use an adversarial model. 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. More precisely, we will actually consider a broader class of instances which contains all instances $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) = L$, and show our algorithm terminates in time at most ρL on any instance in the class.

Let us consider $\mathcal{I}$ 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. For convenience, we will restate Lemma 1 in terms of M_k, which we define to be the maximum amount of work a group of k adjacent processors can contain at time 0 if the instance has an optimum schedule of length L.

Lemma 2 *Let M_k be the maximum amount of work in k adjacent processors in an instance whose optimum schedule is of length L. Then $M_k \leq L^2 + (k-1)L$.*

Proof: Direct from Equation 1. □

Now consider a bucket B_1 that starts off with x_1 units of work, and let us determine what the worst case time to empty that bucket could be. Assume for now that the bucket empties in some time less than m, i.e. 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. An upper bound on the time for B_1 to empty is the minimum r such that

$$\sum_{k=2}^{r} c(\sqrt{x_1 + W_k} - \sqrt{W_k}) \geq x_1 \qquad (2)$$

(For convenience, we assume that B_1 does not leave any work on processor 1; the analysis can easily be modified to handle this case.) This is an upper bound rather than an exact bound on the time for B_1 to empty. Since buckets may run out of jobs before dropping off at a particular processor, B_1 may find that less than $c\sqrt{W_k}$ jobs have been dropped off at processor k; in this case B_1 will drop off its work more quickly than suggested by equation (2). We now allow an adversary to choose an instance, via the values of W_k, which maximizes r. We allow the adversary to choose from all instances which place no more than M_k work in k adjacent processors. Note that this class of instances certainly contains all $\mathcal{I}$ with $\mathcal{L}(\mathcal{I}) = L$; however, it will contain instances with larger optimal schedule lengths as well. A bound on the performance of our algorithm on any instance in this class immediately yields a bound on instances with optimal length L.

Now, for any x_1, in order to maximize the time to empty B_1, the adversary must attempt to minimize each of the terms in the sum (2). Given values for $W_2, \ldots, W_{k-1}$, we claim that $c(\sqrt{x_1 + W_k} - \sqrt{W_k})$ is minimized by maximizing W_k. This claim follows from the following easily proved fact:

Fact 1 *For any non-negative real numbers $a, b,$ and c, $\sqrt{a+c} - \sqrt{a} \geq \sqrt{a+b+c} - \sqrt{a+b}$.*

While it is true that each W_i depends on W_2 through W_{i-1}, making any of W_2 through W_{i-1} smaller does not allow us to make W_i larger, so the best choice for the adversary is to maximize each W_i in turn. The adversary will thus make each W_i as large as possible. However, since W_k is just the work on $k-1$ consecutive processors, he is constrained by the following two rules that follow from Lemma 2:

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

Clearly, the first bound dominates if $x_1 < L$, and the second otherwise. If $x_1 < L$, the adversary sets $W_k = M_{k-1}$, otherwise he sets $W_k = M_k - x_1$.

The following lemma indicates that the adversary's best choice for x_1, in order to maximize the time to empty bucket B_1, is to set $x_1 = L$.

Lemma 3 *The adversary maximizes the distance travelled by B_1 by setting $x_1 = L$.*

Proof: First we consider the case when $x_1 \leq L$. Since the amount dropped off at processor k is $c(\sqrt{x_1 + W_k} - \sqrt{W_k})$, the fraction of the work in bucket B_1 that is dropped off at processor k is easily seen to be

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

If we differentiate this quantity with respect to x_1, we see that it is decreasing as x_1 increases. Thus as we increase the number of jobs in bucket B_1 from 0 to L, bucket B_1 will drop off a smaller fraction of its work at each step, and hence will travel farther.

For the case when $x_1 \geq L$, the fraction of work dropped is

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

which increases with increasing x_1. Thus as we decrease the number of jobs in bucket B_1 down to L, bucket B_1 will drop off a smaller fraction of its work at each step and hence will travel farther.

Thus $x_1 = L$ is the value that maximizes the distance that bucket B_1 travels. $\square$

So the adversary chooses $x_1 = L$ and accordingly sets $W_k = M_k$ in order to give the maximum possible time to empty B_1. We call this instance $\mathcal{J}$ and proceed to analyze the time required for the basic algorithm to schedule it.

Lemma 4 *Assume that no bucket travels for more than m steps. Then the basic algorithm run on instance $\mathcal{J}$ yields a schedule of length at most $4.22\mathcal{L}(\mathcal{J})$.*

Proof: Plugging the values we just obtained into the formula $c(\sqrt{x_1 + W_k} - \sqrt{W_k})$, we see that the time to empty bucket B_1 is now the minimum value of r such that

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

The left hand sum telescopes and gives

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

which solving for r, yields

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

Let

$$\alpha = (2/c + 1/c^2). \tag{3}$$

The value αL is then an upper bound on the time it takes to empty a bucket. Given this value, we can determine the maximum amount of work a processor can receive. Suppose a processor received work from j buckets to the left. Since no bucket travels farther than αL, 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 is then $c\sqrt{L(L + \alpha L)} = cL\sqrt{1+\alpha}$.

We now have the information needed to determine the schedule length. We know that 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, the algorithm is an $(\alpha + c\sqrt{1+\alpha})$-approximation algorithm.

Equation (3) gives c as a function of α, thus we can choose c to yield a bound on the algorithm of

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

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. This modification will be described in the proof of Lemma 5; from now on it is this modified algorithm which we mean when we refer to the basic algorithm.

Lemma 5 *Assume that some bucket travels for more than m steps. Then the (modified) basic algorithm run on instance $\mathcal{J}$ yields a schedule of length at most $4.22\mathcal{L}(\mathcal{J})$.*

Proof: Observe that once a bucket goes all the way around the ring it knows all the work in the system. Thus it knows what the average load should be and can send its excess around the ring so as to bring processor's loads to the average load. Thus after m time to go around the ring once and m time to balance the load, each processor has at most L work and thus we can bound the schedule length by $2m + \mathcal{L}$.

Now recall that the distance any bucket travels is at most αL. Thus, if a bucket travels around the ring we know that $m \leq \alpha L$, *i.e.* $L \geq m/\alpha$. The ratio of processing time to offline running time is then $(2m+L)/L = 1 + 2m/L \leq 1 + 2\alpha$. Given our choice of $\alpha = 1.45$ this gives a bound of 3.89 which is less than 4.22. $\square$

Combining Lemma 4 and 5 we get 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 6, indicate that this analysis is potentially not tight; the worst performance we were able to generate for the basic algorithm was 2.57 times optimal.

4 Extensions of the Basic Algorithm

4.1 Dropping off Jobs of Integral Size

We now dispense with the unrealistic assumption that a job can be split into many 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. During the execution of the basic algorithm, the following two conditions are always true. First, $D_i(t)$, the total amount of work dropped by bucket B_i from time 0 through t, satisfies $D_i(t) = \sum_{p=1}^{t} d'_{i,i+p}$. Second, $R_j(t)$, the total amount of work received by processor j from time 0 through t, satisfies $R_j(t) = \sum_{p=1}^{t} d'_{j-t+p,j}$. We wish to have the *integral algorithm*, the algorithm which only drops off whole jobs, do roughly the same thing as the basic algorithm. Since $D_i(t)$ and $R_j(t)$ are non-integral we will not be able to satisfy these constraints by dropping integral numbers of jobs; however, we will be able to satisfy slightly relaxed versions of these constraints. The following description of the integral algorithm and Lemma 6 are, without loss of generality, phrased in terms of the activity of B_1.

I1) At time t, B_1 now tries to drop off as much as it can drop off at processor t, subject to the constraint that the total it has dropped off, including the amount dropped at t, is no more than $\lceil D_1(t) \rceil$.

I2) At time t, processor j accepts as much work as possible, subject to the constraint that the total amount accepted through that time is at most $1 + \lceil R_j(t) \rceil$.

In other words, a bucket drops off at each processor the maximum it can drop off subject to constraints I1 and I2. We define the *Integral Algorithm* to run like the basic algorithm, modified to satisfy constraints $I1$ and $I2$.

Lemma 6 *Given an instance $\mathcal{I}$, let the basic algorithm return a schedule of length $\rho\mathcal{L}(\mathcal{I})$. Then the integral algorithm returns a schedule of length no more than $\rho\mathcal{L}(\mathcal{I}) + 2$.*

Proof: First, we show that B_1 will still empty in time k, as in the basic algorithm. For any $k' \leq k$, let $V(k') = \sum_{j=1}^{k'+1} R_j(j)$. In the integral algorithm buckets $B_2, \ldots, B_{k'+1}$ dropped off at most k' more work on processors 2 through $k'+1$ than would have been dropped by the basic algorithm, since each bucket, by condition $I1$, drops off at most one extra job. Despite this "extra" work that is dropped off by the integral algorithm B_1 will still be able to drop off all of its jobs in time, since by $I2$ the total "demand" of processors 1 through $k+1$, those through which B_1 travels, is at least $\sum_{j=1}^{k'+1}(1 + \lceil R_j(j) \rceil) \geq V(k') + k' + 1$ jobs. As a result, we can conclude that B_1 will empty in time k. By condition $I2$, each processor will receive at most two units more work than it would in the basic algorithm's execution, so the integral algorithm will finish at most two time units later than the basic algorithm. □

Corollary 1 *The integral algorithm is a 4.22-approximation algorithm.*

4.2 Arbitrary Job Sizes

We now generalize our algorithm to handle arbitrary sized jobs, each of which must be run on exactly one processor. Each processor i begins with $n(i)$ jobs $J_{i,1}, \ldots J_{i,n(i)}$. Job $J_{i,j}$ has processing time $p_{i,j}$. We define x_i to be the sum of the sizes of the jobs of processor i, that is, $x_i = \sum_{j=1}^{n(i)} p_{i,j}$. We let $p_{\max}$ be the maximum job size, that is, $p_{\max} = \max_{i,j}\{p_{i,j}\}$. Let $\mathcal{L}$ be the lower bound obtained by Lemma 1, with the new definition of x_i. A lower bound for the arbitrary sized job problem is $\max\{\mathcal{L}, p_{\max}\}$. We will give a natural generalization of our integral algorithm that achieves an approximation factor one greater than it. The strategy will be to simulate the integral algorithm, but instead of allowing "slack" of 1, allow slack of $p_{\max}$. Note that the processors do not know $p_{\max}$, however if they assume that $p_{\max}$ is the size of the largest job they have seen so far, the algorithm runs without modification.

We define the *Arbitrary Algorithm* to run as the integral algorithm, with constraints $A1$ and $A2$ replacing $I1$ and $I2$. Again, we refer without loss of generality to the activity of bucket 1.

A1) At time t, B_1 now tries to drop off as much as it can drop off at processor t, subject to the constraint that the total it has dropped off, including the amount dropped at t, is no more than $\lceil D_1(t) \rceil + p_{\max}$.

A2) At time t, processor j accepts as much work as possible, subject to the constraint that the total amount accepted through that time is at most $1 + \lceil R_j(t) \rceil + p_{\max}$.

Note that this algorithm is not difficult to implement– each processor goes through the bucket and greedily chooses jobs until no more can be chosen without violating one of the constraints above.

Lemma 7 *Given an instance $\mathcal{I}$, let the integral algorithm return a schedule of length $\rho\mathcal{L}(\mathcal{I})$. Then the arbitrary algorithm returns a schedule of length no more than $(\rho+1)\mathcal{L}(\mathcal{I}) + 1$.*

Proof: First, we show that B_1 will still empty in time k, as in the integral algorithm. For any $k' \leq k$, let $V(k') = \sum_{j=1}^{k'+1} R_j(j)$. In the arbitrary algorithm buckets $B_2, \ldots, B_{k'+1}$ dropped off at most k more jobs on processors 2 through k' than would have been dropped in the original algorithm, since by condition $A1$ each bucket drops off at most one extra job. This means that at most $k'p_{\max}$ more work is dropped off. Due to $A2$, however, the total demand of processors 1 through $k'+1$, those through which bucket B_1 travels in the first k' time steps, is at least $\sum_{j=1}^{k'+1}(p_{\max} + \lceil R_j(j) \rceil) \geq V(k') + (k'+1)p_{\max}$, and therefore we can conclude that B_1 will empty in time k. By condition $A2$, a processor may receive at most $p_{\max} + 1$ extra work, which can increase the running time by $p_{\max} + 1$ over that of the integral algorithm. Since $p_{\max}$ is a lower bound on the running time, this can increase the factor of the approximation by at most one. □

Corollary 2 *The arbitrary algorithm is a 5.22-approximation algorithm.*

4.3 Other Variations

Our algorithm can be modified to handle the case when all m processors have a speed s and/or all the links have a transit time $\tau \neq 1$. If all m processors have a speed s, we can simply form an equivalent instance $\mathcal{I}'$ in which all processors have speed 1 and job $J_{i,j}$ has processing time $p'_{i,j} = p_{i,j}/s$. We can then apply Corollary 2 and obtain a 5.22-approximation algorithm. If all links have a transit time $\tau \neq 1$, we can just scale time, so that τ becomes 1. To do so, we set the transit time to one, and the machine speeds to $1/\tau$. We then apply the algorithm described above for speed $s = 1/\tau$, obtain a schedule and adjust the schedule length by a factor of τ.

5 A Lower Bound

In this section we prove that there is no distributed approximation algorithm for the basic problem (as defined in Section 2) with performance guarantee better than 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 intuition behind this argument is that a distributed algorithm can not differentiate between the two classes and therefore must do the same thing for both until it is too late.

Let $\delta = .06$ and assume A is a $(1+\delta)$-approximation algorithm. The two problem instances we consider are:

- $\mathcal{I}$: W unit-size jobs are placed on each of two processors at distance $2z+1$ apart in the ring. We will call the two processors p_1 and p_2, and let their numbers be i and $i+2z+1$ respectively. We will refer collectively to processors number $i+z$ and $i+z+1$ as the "midpoint."
- $\mathcal{J}$: Only p_1 receives W unit-size jobs at time 0, and all other processors receive no jobs.

For both instances the ring size m is the same and is chosen large enough so that $m-(2z+1) >> \mathcal{L}(\mathcal{I})$.

Lemma 8 *The shortest schedule for $\mathcal{I}$ is of length t, where t satisfies $2W = 2t^2 - (t-z)^2 + t - z$.*

Proof: At time 0, only processors p_1 and p_2 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, $4z+2$ have work. After time z work has reached the midpoint from both sides, hence for each subsequent step only two additional processors can receive work. Thus the total amount of work processed in t steps, where $t > z$ and no work goes around the ring, is

$$\sum_{i=0}^{z} 2+4i + \sum_{i=z+1}^{t-1} 4z+2+2(i-z).$$

As stated above we choose m large enough so that no work goes around the ring, so setting this equal to $2W$ yields the lemma. □

Note that if $z = (1-\epsilon)t$, $0 < \epsilon < 1$, and we choose W large enough, then W can be made arbitrarily close to $(1-\epsilon^2/2)t^2$. Since our theorem is actually true for a value somewhat larger than $\delta = .06$, assuming $W = (1-\epsilon^2/2)t^2$ will not affect our $(1+\delta)$ lower bound.

Now consider the performance of A on $\mathcal{J}$. The optimal schedule for $\mathcal{J}$ is of length $\sqrt{W} = t\sqrt{1-\epsilon^2/2}$. Since we have assumed that A is a $(1+\delta)$-approximation algorithm, it must finish W work in time

$$u = (1+\delta)t\sqrt{1-\epsilon^2/2}.$$

In other words, by this time no processor has any remaining jobs.

Lemma 9 *Consider the performance of algorithm A on instance $\mathcal{I}$. At any time $u > z$ there are at least $V = 2W - 2u^2 + (u-z)^2$ units of unprocessed work remaining in the system, all of whom 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 be in if the instance was $\mathcal{J}$, since it can not yet have learned of the work from the other side of the midpoint.

Applying Lemma 8 gives that the amount of work which could have been processed up to time u is $2u^2 - (u-z)^2 + u - z$. By similar arguments as above we can take this to be $2u^2 - (u-z)^2$. Therefore, at time u there are at least $V = 2W - 2u^2 + (u-z)^2$ remaining unprocessed jobs. Since processors that are not within $u-z$ of the midpoint must be in the same configuration at time u for both $\mathcal{I}$ and $\mathcal{J}$, and these processors must have completed processing by time u in $\mathcal{J}$, all of the unprocessed work must be within distance $u-z$ of the midpoint, i.e. within a region of width $2(u-z)$. □

Using Lemma 1 to calculate a lower bound q for processing V units of work which start in a region of width $2(u-z)$, we see that A must take at least $u+q$ time to complete instance $\mathcal{I}$. By choosing ϵ to be .71 we obtain $u+q > (1+\delta)t$, which contradicts our assumption that A is a 1.06-approximation algorithm. Thus we have shown the following:

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

6 Experimental Work

6.1 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 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; 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. 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 buckets so as to have the square root of the work that has *passed by* (in contrast to C in which a processor brings itself up to the square root of the work *it knows about*).

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

I) Structured Test Cases

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

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	100 500	20 40 80

Table 1: The three different input distributions

thousand nodes; the number of jobs on the heavily loaded processors ranged from a thousand to a hundred thousand; and the distributions ranged from heavily concentrated to widely dispersed. 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 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 3 might construct. Notice that the evil adversary could construct any one of a variety of instances and still follow the strategy that we laid out. For example, the adversary can pick any lower bound, L, and also can pick any size region, k, for the instance. Our six evil adversary test cases varied these two parameters and the details are in part III of Table 1. This amounts to 51 test cases in all.

6.2 Analysis

In order to obtain empirical factors of approximation we needed to compute the optimum length schedules for each instance, or at least lower bounds on the optimum schedule length for each instance. For many of the test cases we actually computed the optimum schedule length. It was previously known how to compute the optimum length schedule for an instance [7] with unit-size jobs. This approach, however, requires roughly n^2m space, which for some of our test cases is 10^{15}. We developed another more space efficient approach which uses m^2 space which for the largest of our test cases is only 10^6. Needless to say, this approach made the problem more computationally feasible. However, even with this modified approach some instance's optimum schedule lengths still eluded us. For this handful of cases we used one of two lower bounds. The first is the lower bound explained in Lemma 1 and the second is just $\lceil \frac{n}{m} \rceil$. So the empirical approximation factors obtained for these instances are somewhat pessimistic.

The simulation results show that the algorithms perform better than our analysis suggests. Figures 2 through 7 give histograms of the approximation factors found in our test cases. Each bar of the histogram corresponds the frequency which we obtained the approximation factor in the range specified by the x axis.

Algorithm C1, for which we showed a worst case bound of 4.22 in Section 3, performed no worse than 3.09 times optimal in the simulations. This was on an instance whose optimum we could not calculate exactly; the worst performance of C1 on an instance whose optimum we knew was 2.57 times optimal. Also note that Figure 4 shows that many of the experiments for C1 had an approximation factor of 1.2 or less. In fact all six algorithms show good performance most of the time. The best algorithm empirically was algorithm A2 which performed no worse than 1.65 times optimal in any of the 51 test cases. We speculate that the A algorithm does the best because it performs slightly better local load balancing, as it works only with the work it sees going by. In general the bidirectional versions of the algorithms were somewhat better, but this feature did not offer improvements anywhere close to a factor of 2. Note also that Algorithm B performed the worst despite working with a more accurate estimation of the lower bound.

It is interesting to note that for algorithm $C1$ the examples constructed according to the adversary's strategy did not produce the worst approximation factors. In addition, for all the algorithms, the worst approximation factors were reported on instances for which we do not have the exact value of the optimal solution.

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

7 Capacitated Rings

In some environments the assumption of unlimited available bandwidth on the network links is the appropriate model, while in others it is more realistic to model the bandwidth as limited. In this section we explore the difference between these models and give a distributed algorithm for job scheduling in a ring in which each link has capacity one. In this model only one job and one message can be passed over a link in a single time step; each message sent will simply be a number describing how many unprocessed jobs the sending processor has resident.

The basic idea of the algorithm is for each processor to know the state of its neighbors at the previous time step, and then pass a job to either or both neighbors if that neighbor is in danger of being idle on the next time step. The details appear in Figure 1, where we use j_i to denote the number of unprocessed jobs on processor i. Note that in this description of the algorithm two messages can be sent over a link in one step; it is not hard to reduce this to one.

We wish to prove that this algorithm produces schedules of length close to optimal. We first show a lower bound similar to that in Lemma 1.

Lemma 10 *If the optimal schedule is of length L then no consecutive group of k processors can start with more than $(k+2)L$ total jobs.*

Proof: Omitted. Similar to the proof of Lemma 1. □

This lemma exposes a significant restriction on the way work can be distributed among the processors. For example, no pair of adjacent processors, at time 0, can contain more than $4L$ jobs. There are also significant restrictions on the conditions under which jobs can be passed.

Lemma 11 *Given a processor i, let t be the earliest time that $j_i \leq 1$. Then*

a) p_i receives no jobs before time t.

b) After time t, $j_i \leq 3$.

c) Let $t' > t$ be the first time that $p_j \leq 3$, where j is a neighbor of i. For any k, $1 \leq k \leq t' - t$, p_j passes a job to p_i in at least half of the time steps between time t and $t + k$, inclusive.

Proof: Part a) is clear from the description of the algorithm. For part b), we observe that due to the time delay between the actual state of a processor and the state that its neighbor is aware of, processor i can receive jobs for two consecutive time steps. When it first receives jobs it must have zero jobs, thus after receiving jobs it has at most two jobs. During the next step, it will process one job, and receive up to two jobs, thus having at most three jobs at the end of the step. However, if it had two at the beginning of the previous step it will receive no more jobs on the next step, and will receive no more jobs until it has processed all of its jobs and has zero remaining. Hence there is no way for the number of jobs to rise above three.

A slightly more careful look at the proof of part b) will suffice to establish part c). At time t, if p_i has one job, then at times $t+1$ and $t+2$ it will receive a job from each neighbor that has more than three unprocessed jobs. This is due to the one unit time delay between the actual state of a processor and the state that the processor's neighbor is aware of. It then takes p_i at most two steps to process received jobs until it returns to having one job; the cycle continues until p_i's neighbors run out of work to pass. So in at least two out of every four steps, passing occurs; furthermore, the passing occurs in the first two steps after p_i becomes idle. This establishes the claim. □

We now show that the algorithm in Figure 1 is a 2-approximation algorithm.

Lemma 12 *Let S' be the schedule in which no processor ever passes a job and let S be the schedule produced by the algorithm in Figure 1. Then S is no longer than S'.*

Proof: Let $m(t)$ be the maximum number of jobs on any processor at time t. In schedule S' it is always the case that $m(t+1) = m(t) - 1$. We will show that in schedule S, $m(t+1) \leq m(t) - 1$, thus proving the lemma.

We observe that in S the only processors that pass jobs have more than three jobs, and that the processor that had the maximum number of jobs $m(t)$ at time t has at most $m(t) - 1$ jobs at time $t + 1$. Thus, when $m(t) > 3$, the processors with $m(t)$ jobs decrease by at least one, and by part b) of Lemma 11, no processor's load increases above 3, so $m(t)$ decreases. When $m(t) \leq 3$, no passing occurs. Therefore at each step of S $m(t)$ decreases by at least 1, which implies that the length of S is at most $m(0)$, which is the length of S'. □

Now using the previous three lemmas we show that the capacitated ring scheduling algorithm gives schedules of length within a factor of two of optimal.

Theorem 3 *Let L be the length of the optimal schedule. Then the capacitated ring scheduling algorithm produces a schedule of length no more than $2L + 2$.*

Proof: There are two cases to consider.

<u>Case 1:</u> No processor starts with more than $2L$ work.

By Lemma 12 we know that the schedule length does not increase by passing jobs, thus the maximum schedule length for this case is $2L$.

<u>Case 2:</u> Some processor starts with more than $2L$ work.

Let processor p_i be a processor that starts with more than $2L$ work, i.e. processor p_i has $2L + x$ work (for $0 < x \leq L$). Its neighbors, p_{i-1} and p_{i+1}, start with at most $2L - x'$ (for $x' > x$ by Lemma 10). Assume p_{i+1} starts with $2L - x'$ work, and that this is no less than what p_{i-1} starts with. At time no later than $2L - x'$ p_{i+1} becomes idle. At this point it may receive work from both its neighbors. By part c) of Lemma 11 we know that during at least half the time steps in the interval from time $2L - x'$ through the time when j_i first goes below 3, p_i will pass jobs to p_{i+1}.

At time $2L - x'$ p_i has at most $x + x'$ work. Assume for simplicity that p_i has passed no jobs to its neighbors at any time up to $2L - x'$. It will now begin to pass jobs to its neighbors until it has only three jobs left. In $\lceil \frac{x+x'-3}{2} \rceil$ time it can pass $\lceil \frac{x+x'-3}{4} \rceil$ work to each neighbor and process $\lceil \frac{x+x'-3}{2} \rceil$ jobs. It will then spend three units of time processing the final three jobs. So p_i completes all its jobs in time no more than $2L - x' + \lceil \frac{x+x'-3}{2} \rceil + 3$ which is less than or equal to $2L + 2$ since $x' > x$. □

We note that a more careful analysis (on a slightly modified algorithm) which goes through a number of cases for the last three steps can be used to show a bound of exactly $2L$. We omit the details due to space limitations.

receive messages from neighbors $i-1$ and $i+1$
set *left* and *right* equal the values of the messages from $i-1$ and $i+1$ respectively
if $j_i \neq 0$
    process a job, set j_i to $j_i - 1$
    if $j_i > 3$ and $right \leq 1$ then pass a job to neighbor p_{i+1} and set j_i to $j_i - 1$
    if $j_i > 3$ and $left \leq 1$ then pass a job to neighbor p_{i-1} and set j_i to $j_i - 1$
tell neighbors that p_i has i jobs.

Figure 1: One step of the capacitated ring scheduling algorithm for processor i.

8 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 variations are extremely simple and do not rely on global knowledge or centralized control. Simulations of this algorithm and related algorithms suggest that the performance guarantee may be better than what is shown in Section 3.

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. [4] 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 in this paper.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] 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*, 1992.

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

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

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

[11] 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.

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

[13] H. Kanada. Construction of a distributed computer network containing ring connections. In *Systems and Computers in Japan*, pages 25–34, 1993.

[14] 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 pp 445–522. North-Holland, 1993.

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

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

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

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

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

[20] 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.

[21] 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.

[22] J. Takahashi, S. Hattori, T. Kimura, and A. Iwata. A ring array processor for highly parallel dynamic time warping. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34:1310–1318, 1986.

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

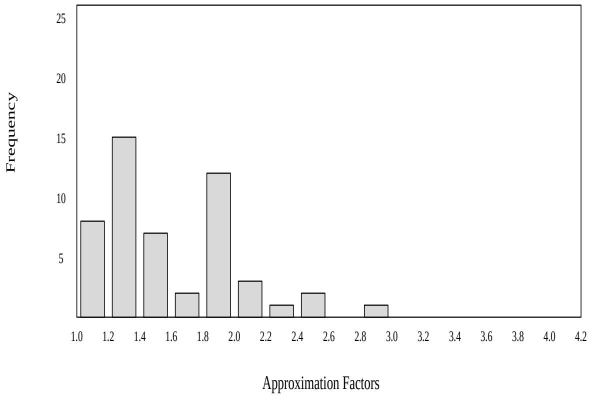

Figure 2: Approximation factors for 51 runs of A1

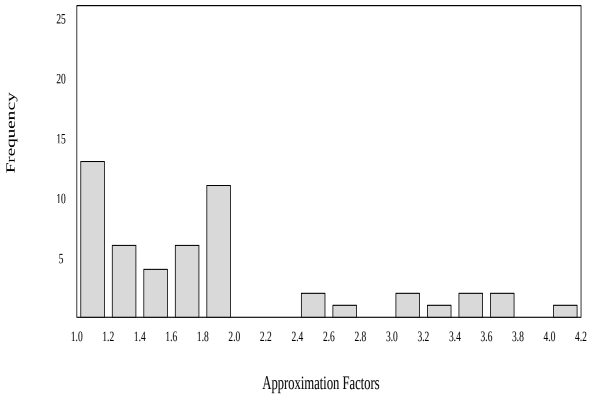

Figure 3: Approximation factors for 51 runs of B1

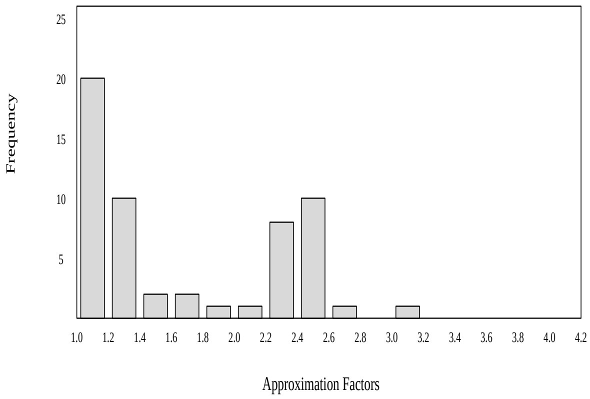

Figure 4: Approximation factors for 51 runs of C1

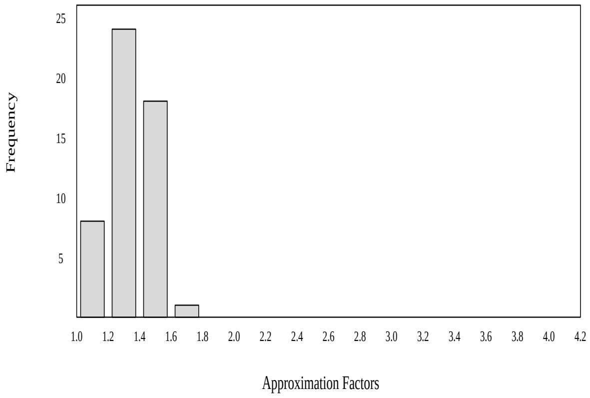

Figure 5: Approximation factors for 51 runs of A2

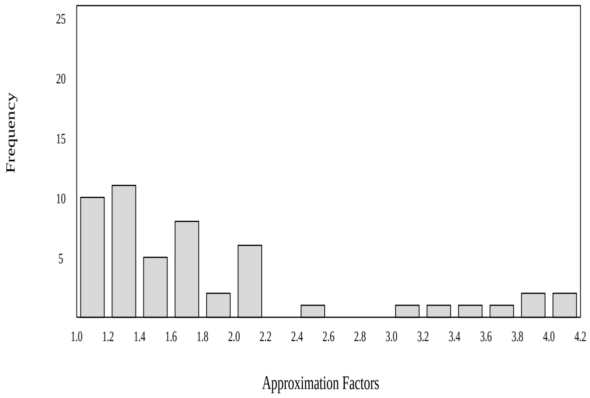

Figure 6: Approximation factors for 51 runs of B2

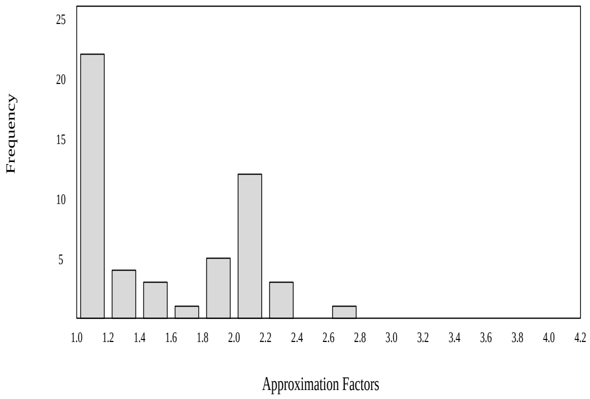

Figure 7: Approximation factors for 51 runs of C2