Asynchronous Analysis of Parallel Dynamic Programming Algorithms

Gary Lewandowski*Anne Condon† and Eric Bach‡
Computer Sciences Department
University of Wisconsin at Madison
1210 W. Dayton St.
Madison WI 53706

January 1994

Abstract

We examine a very simple asynchronous model of parallel computation that assumes the time to compute a task is random, following some probability distribution. The goal of this model is to capture the effects of unpredictable delays on processors, due to communication delays or cache misses, for example.

Using techniques from queueing theory and occupancy problems, we use this model to analyze two parallel dynamic programming algorithms. We show that this model is simple to analyze and correctly predicts which algorithm will perform better in practice.

The algorithms we consider are a pipeline algorithm, where each processor i computes in order the entries of rows i, $i + p$ and so on, where p is the number of processors; and a diagonal algorithm, where entries along each diagonal extending from the left to the top of the table are computed in turn.

It is likely that the techniques used here can be useful in the analysis of other algorithms that use barriers or pipelining techniques.

Index Terms: Parallel dynamic programming, asynchronous algorithms, parallel models of computation, queueing theory, analysis of algorithms.

*Work supported by WARF grant 135-3094. Current address: Xavier University, Cincinnati OH 45207-4441, email lewan@xavier.xu.edu

†Work supported by NSF grant numbers CCR-9100886 and CCR-9257241 and by matching funds from Digital Equipment Corp. and the AT&T Foundation. email condon@cs.wisc.edu.

‡Work supported by NSF grant number DCR-855-2596, email bach@cs.wisc.edu.

1 Introduction

Parallel algorithms can suffer significant slowdown due to unpredictable delays in the system. These delays include contention on communication channels or cache misses, for example. Unpredictable delays lead to periods of forced idleness among the processors, during which processors are ready to work, but cannot. For example, a processor may be forced to be idle at a synchronization point because of delays to another processor.

Analytically predicting the increase in running time of an algorithm due to unpredictable delays is an important task because it provides a basis for deciding which of a set of algorithms is best for a particular system. However, there is no consensus on how best to do this. What is needed is a model of computation that is simple, general and accurate. By simple, we mean that the model should be easy to use and analyze. By general, we mean that the model should work for all algorithms, or at least all algorithms on a given architecture. By accurate, we mean the analysis should accurately reflect an experimental measure of the algorithm's performance.

We examine a very simple asynchronous model of parallel computation for task graphs. Our model assumes that the time to compute a task is random, following some probability distribution (usually the exponential distribution). The resulting variability in the task time is designed to model the effects of unpredictable delays in the system on the computation of an algorithm. With this model, we analyze the performance of two parallel dynamic programming algorithms, using techniques from queueing theory and occupancy problems. We present empirical evidence that the analysis using the model can accurately predict which of two algorithms performs better in practice. While this analysis does not prove the generality of the model, it is likely that the techniques used here can be useful in the analysis of other algorithms that use barriers or pipelining techniques.

1.1 Problem description

We view the dynamic programming problem as that of computing entries in a large table, say of dimension $n \times m$, where the computation of entry (i, j) depends on the results of its *predecessors*, which are entries $(i-1, j), (i-1, j-1)$ and $(i, j-1), 1 \leq i \leq n, 1 \leq j \leq m$. (Entry $(1, 1)$ is known in advance and the remaining entries in the first row and column have only 1 predecessor.) Throughout, we assume without loss of generality that $n \leq m$ (since the table can be transposed if $n > m$). Dynamic programming is a classic algorithmic technique. It is used, for example, to solve the Knapsack problem [10], the Longest Common Substring problem [23], queueing network models [1], and DNA sequence alignment [17]). There are many possible parallel algorithms to implement it. The two dynamic programming algorithms that we consider are the *pipeline* and *diagonal* algorithms. We also assume that the number of processors, p, is at most n.

In the *pipeline* algorithm, the ith processor computes the entries in rows $i, i+p, \ldots$, in order. A processor can compute an entry as soon as its predecessors are computed. (We assume that processors can test whether the predecessors of an entry are already computed, using locks, for example.) Almquist et al. [1] used this algorithm in solving the longest common substring problem and a problem on queueing network models. In the *diagonal* algorithm, entries along each diagonal extending from the left side to the top of the table, are computed in turn. Within each diagonal, each processor computes approximately $1/p$ of the entries. The computation of

the entries along a diagonal is not started until all entries along the previous diagonal are computed. This can be enforced using barriers, for example. Lander et al. [13] proposed this parallel algorithm for protein sequence alignment.

Note that the two algorithms are very similar in that a processor may compute exactly the same sequence of table entries in both algorithms. Thus, the time spent actually computing table entries is the same for both algorithms. The important difference between the algorithms is that in the diagonal algorithm, all processors use barrier synchronization to ensure that when computing an entry, its predecessors are already computed, whereas in the pipeline algorithm, a processor explicitly checks that the predecessors are computed. Thus, in comparing the running times of the algorithms, one must measure for the diagonal algorithm the costs of barriers and of the forced idleness of processors as they wait at barriers; and for the pipeline algorithm the costs of testing whether the predecessors of an entry are already computed, and of waiting until the predecessors are computed.

While experiments show that their performance is generally different on the same input data, synchronous models of parallel computation, such as the PRAM [6] [9] [21], give the same time complexity for both algorithms. A useful asynchronous model must capture the difference.

It is not obvious which of the two algorithms will have better performance. In the diagonal algorithm there are $\Theta(n)$ barriers, while in the pipeline algorithm, each processor must test $\Theta(n^2/p)$ times that the predecessors of an entry are computed. In our experiments on the Sequent Symmetry, we used locks for this purpose in the pipeline algorithm. (An alternative efficient scheme is to have processors read a flag associated with the predecessors of an entry, to test whether the entry has been computed. However, it is difficult to make measurements of this scheme on the Sequent.) We found that the cost in running time of executing a barrier with sixteen processors is about ten times the cost of executing a lock (and is less for fewer processors). Even for large p, the cost of $\Theta(n^2/p)$ locks will dominate the cost of $\Theta(n)$ barriers for moderately large n. Thus, if only synchronization time is considered we would expect the diagonal algorithm to be a better choice. However, it is misleading to consider only the cost of synchronization primitives. In the diagonal algorithm, if the time taken by different processors between barriers is highly variable, then the cumulative idle time of processors before the barriers may be significant. One would expect that the cost in running time due to such forced idleness would be more in the diagonal algorithm than the pipeline algorithm; furthermore our experiments show that this is indeed the case (see Section 5). Thus, it is hard to be certain, without further study, which algorithm will perform better. The purpose of our results is to compare analytically the cost of forced idleness in the diagonal and pipeline algorithms. Of course, there are yet other costs of these parallel algorithms in addition to the two discussed here. For example, in the pipeline algorithm, if testing whether a predecessor of a table entry is computed involves repeated reading of the predecessor, the resulting message volume may cause the pipeline algorithm to have a longer average task time than the diagonal algorithm (and may also be closely related to variations in task time).

1.2 Description of Model

In our random model, the time required to compute an entry in the table, given that its predecessors have already been computed, is exponentially distributed with mean $1/\mu$, where μ is some constant, $0 < \mu < \infty$. For the diagonal algorithm, we also extend our analysis to some

other distributions. The running time of an algorithm is the time to compute all the entries in the table.

This model attempts to capture the effects of processor idleness, due to unpredictable delays in a parallel system, on the running time of an algorithm. Such delays may be due to contention on communication channels or cache misses, for example. It is not the goal of this model to predict actual running time, rather it provides a basis from which to compare two algorithms without actually implementing them. We note again that other, perhaps significant, costs of the diagonal and pipeline algorithms are the cost of barrier synchronization and the cost of testing that the predecessors of an entry are already computed. These other costs are *not* included in our model, and would have to be included in a complete analysis of the running time of the computation.

We assume that task times are independent. We note that in practice, task times are probably not independent at all. However, since it may be very hard to predict the actual dependencies that may occur on a given machine, we do not attempt to incorporate such dependencies faithfully in an analysis such as ours. The independence assumption keeps the analysis as simple as possible; furthermore, there is precedent for this assumption in the literature. For example, previous papers which consider expected running times of parallel task structures [5, 11, 15, 16] also assume that task times are independent (for more details, see Section 1.4 on related work).

We use the exponential distribution for a variety of reasons. First, it makes analysis of the pipeline case feasible, since tools from queueing theory are applicable in this case. Second, our measurements in a real application (see Section 5) have shown that the variance of task times can be high; thus assuming task times follow the exponential distribution is not unreasonable. Our experiments on the two algorithms confirm that the results we get by using the exponential distribution are qualitatively accurate. Finally, performance folklore holds that as long as some variation exists, the exponential distribution is a reasonable one to use in predicting performance. Even for parallel models, there is some evidence supporting this: for example, Fromm et al. [7] measured delays in executing an instruction due to memory conflicts on one parallel system, the Erlangen General Processor Array, and concluded that the exponential distribution was the best of several distributions in predicting performance.

To compare the predictive power of the exponential distribution with other distributions, we did our own simulations with task times chosen according to the uniform, normal, exponential and gamma distributions. The algorithms were simulated on $n \times n$ tables, with n equal to 1000, 2000, 3000, 4000, and 5000. Table 1 details the results of the study when the diagonal and pipeline algorithms were simulated on sixteen processors (results for fewer processors were similar). For each n, the table first gives a simple lower bound on the running time, namely n^2/p, which is approximately the running time when there is no variance in the task times. Then, for each distribution we give the running time as a percentage of this lower bound. For all distributions, the diagonal algorithm is slower than the pipeline algorithm (this is consistent with our experiments, which we describe later). We observe that the running times for the exponential distribution are the most pessimistic, showing the worst slowdown. For the uniform distribution, the running time of the pipeline algorithm is actually faster than the lower bound. We conclude from this that the uniform distribution would be a poor choice in modeling slowdown due to idleness. As n increases, the percentage difference between each of the running times and the lower bound decreases. This suggests that for any of the distributions, the slowdown

due to variation in task time is a low order term in the total running time. We will see that our analytical results confirm that this is the case for the exponential distribution; moreover this is also consistent with our experimental results. From these simulations, we conclude that an analysis based on the exponential distribution may be somewhat pessimistic, but should be useful in explaining qualitatively the effects of task variance on the total running times of the algorithms.

Table	Lower	Uniform		Normal		Exponential		Gamma	
Size (n)	Bound	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
1000	62500	119%	97%	129%	102%	132%	102%	114%	101%
2000	250000	111%	96%	120%	101%	122%	101%	109%	101%
3000	562500	108%	95%	117%	101%	118%	101%	108%	101%
4000	1000000	107%	95%	115%	101%	116%	101%	107%	100%
5000	1562500	105%	95%	113%	101%	114%	101%	106%	100%

Table 1: Results of simulating the pipeline and diagonal algorithms on 16 processors. For each distribution, the running time as a percentage of the lower bound, which is $n^2/16$, is given. The uniform distribution used a mean task time of 1 on the range [0,2]. The normal distribution used a mean of 1 and a variance of 1. The exponential distribution used $\mu = 1$. The gamma distribution used a mean of 1 with order 5. Each simulated processor used an independent random number stream.

1.3 Overview of Results

1.3.1 Theoretical analysis

Our analytic results show that the expected running time of the diagonal algorithm is worse than the expected running time of the pipeline algorithm. The difference increases as the number of available processors increases, indicating that with more processors, the advantages of the pipeline algorithm would increase.

To get an idea of how our results on the two algorithms compare, Table 2 presents our results for three cases of p, in the special case when the table is of size $n \times n$ and $\mu = 1$. The table gives a lower bound on the expected running time of any algorithm in a large class of *static* algorithms (defined below), which includes both the diagonal and pipeline algorithms. It also gives an upper bound on the expected running time of the pipeline algorithm, and a lower bound on the expected running time of the diagonal algorithm. In each case, the lower bound of the diagonal algorithm is larger than the upper bound of the pipeline algorithm.

1.3.2 Summary of Experiments

The two algorithms were timed on the Sequent Symmetry, an asynchronous, shared memory parallel machine. Experiments were run using 1, 4, 8, 12 and 16 processors. Details of the experiments are given in Section 5.

In our experiments, the pipeline algorithm outperformed the diagonal on almost all data sets, although the difference in running time is small. We estimated as best we could the cost of synchronization and processor idleness in both algorithms, and concluded that processor idleness

	$p > 1$ a constant	$p = \sqrt{n}$	$p = n$
lower bound (static algorithms)	$n^2/p + \Theta(1)$ (Lemma 2.1)	$n\sqrt{n} + \Theta(\sqrt{n})$ (Lemma 2.1)	$2n - 1$ (Lemma 2.1)
upper bound, pipeline	$n^2/p + 2n + \Theta(1)$ (Theorem 3.1)	$n\sqrt{n} + 2n + \Theta(\sqrt{n})$ (Theorem 3.1)	$4n$ (Theorem 3.1)
lower bound, diagonal	$n^2/p + 2nH_{p-1}$ $-3n - n/p - 1$ (Theorem 4.1)	$n\sqrt{n} + n\log n$ $-3n - \sqrt{n} - 1$ (Theorem 4.1)	$2n\log n$ $+\Theta(n\log\log n)$ (Theorem 4.2, asymptotic)

Table 2: Running times for three cases of p ($n \times n$ table, $\mu = 1$). Here, $H_p = \sum_{i=1}^{p} 1/i (\geq \log p)$. Logs are to the base e. The lower bound result for the diagonal algorithm with $p = n$ is asymptotic.

accounted for much of the gap in the running times of the two algorithms. The difference in the idleness increased as the number of processors increased, as predicted in the theoretical analysis. We also found that the distribution of the task times has a standard deviation close to the mean, providing some empirical justification for assuming the exponential distribution in our analysis.

The rest of the paper is organized as follows. We describe related work on asynchronous models of parallel computation in Section 1.4. Our lower bound is presented in Section 2. In Sections 3 and Section 4, we present the analysis of the pipeline and diagonal algorithms, respectively. Our experimental results are reported in Section 5.

1.4 Related Work

Various models have been proposed to analytically predict the running time of algorithms on asynchronous models of parallel computation in which unpredictable delays contribute to the running time. One model, used by Anderson et al. [2] in analyzing dynamic programming algorithms on small, asynchronous parallel machines, allows an adversary to control the delays of the processors. Such a model may be useful for predicting the worst case performance of an algorithm.

Another approach is to make the running time of a task a random variable. Nishimura [19], Cole and Zajicek [5] and Martel et al. [16] describe general models of asynchronous parallel computation with such random delays. This approach appears to be a promising one, when one wants to estimate the performance of an algorithm on an average run, rather than in the worst case. Our simple model follows this approach. Our work extends the above results both by providing a sharp analysis of two commonly used algorithms, and by providing empirical evidence that the analysis is realistic.

Fromm et al. [7] use a stochastic model to analyze the performance of a parallel system, the Erlangen General Processor Array. The time needed to execute an instruction in this system depends on delays due to memory conflicts and is modeled as a random variable. Different distributions are considered, including constant, exponential and "phase-type" distributions, and the results of the analysis are compared with experimental results. The authors conclude that the analysis using the exponential distribution compares favorably with experimental results.

Mak and Lundstrom [15], describe analytic models for predicting the performance of a parallel program represented as a task graph with series-parallel structure, where the time to execute a task is exponentially distributed. Our work on the diagonal algorithm extends their results on series-parallel graphs, while our work on the pipeline algorithm provides tools for predicting the performance of task graphs with a mesh structure, where dependence between the tasks is much more complex than in a series-parallel graph.

Kruskal and Weiss [11] analyze the expected running time of p processors working on a pool of n subtasks. Each subtask can be done independently. They show that allocating an equal number of subtasks to each processor has good efficiency. Their result on the expected running time of the p processors is equivalent to the expected running time on a single diagonal of the diagonal algorithm, when p is $o(n)$ but more than a constant. Our work extends their result to give an upper bound for computing an entire table in this case, and also gives results for constant p and for $p = \Theta(n)$. The pipeline analysis examines processors that are interacting, a case not considered in their work.

2 Lower Bound for Static Algorithms

Our first analytic result is a simple lower bound on the expected running time of a general class of *static* algorithms. This class includes the pipeline and diagonal algorithms and is useful in comparing the bounds derived later for these algorithms. In a static algorithm, each processor is assigned a sequence of table entries, where the assignment of processors to entries is fixed before execution of the algorithm. A processor computes each entry of its sequence in turn and is ready to compute the kth entry in the sequence once it has completed the computation of the $(k-1)$st entry and all predecessors of the kth entry are computed. Recall that we assume in this lemma and throughout that $p \leq n \leq m$.

Lemma 2.1 *A lower bound on the expected running time of a static dynamic programming algorithm with p processors on an $n \times m$ table is $(mn/p + p - 1)/\mu$.*

Proof: A trivial lower bound on the expected running time of an algorithm is $mn/(\mu p)$, since some processor must compute at least mn/p entries, and the expected time to compute each of them is $1/\mu$. We can improve this lower bound to prove the lemma, by taking into account the fact that at the start and end of the computation, not all processors can be actively computing entries.

In what follows, we refer to the set of entries $\{(i,j) \mid i + j = d + 1\}$ as the dth diagonal of the table. Consider the set of entries between diagonals numbered p and $n + m - p$. There are $mn - p(p-1)$ entries in this set; hence some processor, say i, is assigned to compute at least $(mn - p(p-1))/p$ of the entries in this set. Let e_1 and e_2 be the first and last entries in this set that processor i computes. For static algorithms, these are fixed in advance of the computation. The expected time for processor i to compute the entries in this set is at least $(mn - p(p-1))/\mu p$.

Since e_1 is in a diagonal numbered at least p, there is a sequence of entries of length at least p, starting with the top left entry of the table and ending with e_1, such that each entry in the sequence cannot be computed until the previous entry in the sequence is computed. Hence, the expected time from the start of the computation until entry e_1 can be computed is at least $(p-1)/\mu$. Similarly, since e_2 is in a diagonal numbered at most $n + m - p$, there is a sequence

of entries of length at least p, starting with e_2 and ending with the bottom left entry of the table, such that each entry in the sequence cannot be computed until the previous entry in the sequence is computed. Hence, the expected time from the time e_2 is computed until the end of the computation is at least $(p-1)/\mu$.

Hence the total expected time is at least

$$\frac{mn - p(p-1)}{\mu p} + 2\frac{p-1}{\mu} = \frac{1}{\mu}(mn/p + p - 1).$$

□

3 Analysis of Pipeline Algorithm

In this section we give an upper bound for the expected running time of the pipeline algorithm. We begin by presenting the intuition behind the proof and then give the formal proof.

3.1 Intuition

In this section, we assume that n is divisible by p. This is not necessary for the analysis, but makes the intuition more clear.

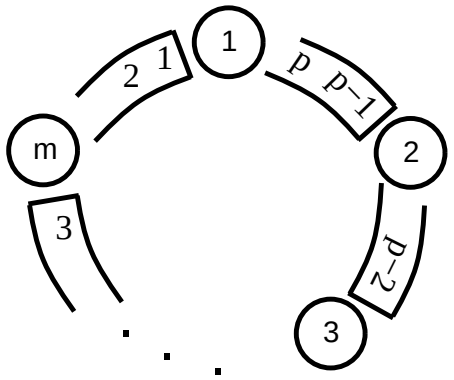

Figure 1: Cyclic p-customer problem: How long for p to be served s times, if the system starts in the steady state?

We relate the pipeline algorithm to a problem from queueing theory. Consider a cyclic queueing system in which p customers circulate through m servers and queues, where the service times are exponentially distributed. Assume that the system is in the steady state. The *cyclic p-customer problem* is to determine the expected time for the pth customer to be served s times in a cyclic queueing system with m servers (see Figure 1.) This problem has a known solution: $(s/\mu)(1 + (p-1)/m)$ (Lavenberg & Reiser [14, Equation 2.17]).

To relate the pipeline algorithm to the cyclic p-customer problem, think of the p processors as customers, and the m columns in the $n \times m$ table as servers. Each column corresponds to a server because only one processor can be working on an entry in a column at a time. (Figure 2.)

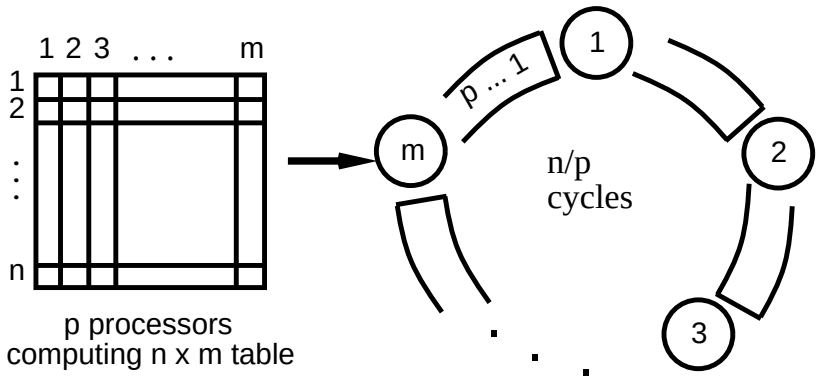

Figure 2: Pipeline algorithm: how long for the pth customer to be served $(n/p)m$ times?

If the customers start out lined up at the queue to the first server, then the expected time for the pth customer to be served $(n/p)m$ times is the expected running time of the pipeline algorithm.

Unfortunately, the cyclic p-customer problem does not correspond directly to the pipeline algorithm. Since the cyclic p-customer problem assumes the system is in steady state, the customers do not start out lined up in the first queue. Thus, the expected time for the pth customer to be served $(n/p)m$ times in steady state does not include the startup costs associated with the pipeline algorithm.

It is easy to overestimate these startup costs by making the pth customer go around the entire system an additional time. That is, the expected time for a customer to be served $m + (n/p)m$ times is an upper bound on the running time of the pipeline algorithm. This guarantees that every customer will have been served at server 1 before we start counting the services of the pth customer. However, this overestimate may be very severe if $m >> p$.

The key to our solution is to notice that the cyclic p-customer problem can model the pipeline algorithm *without using m servers*. If we use fewer servers, customers must wait more often, so the time to be served $(n/p)m$ times only increases (i.e. the estimate of the time for the pth processor to compute $(n/p)m$ cells will be at least as large the actual time). For example, we could decide to use only p servers. In this case the startup cost (one cycle through the system) would be much closer to the cost actually experienced by the pth processor, while the time to be served $(n/p)m$ times would be a large overestimate if $m >> p$. (Figure 3.)

For any $x \leq m$, an upper bound on the running time of the pipeline algorithm is

$$(x + (n/p)m)(service\_time).$$

Here, x is the number of servers in the cyclic p-customer problem and so the time for the pth customer to go around the entire system one time (the startup cost) is x times the service time. Also, the service time is $(1/\mu)(1 + (p-1)/x)$. Choosing x to minimize this equation balances the startup cost with the cost of being served $(n/p)m$ times. Simple calculus provides us with the best value, which is approximately $\sqrt{mn}$ (see Theorem 3.1).

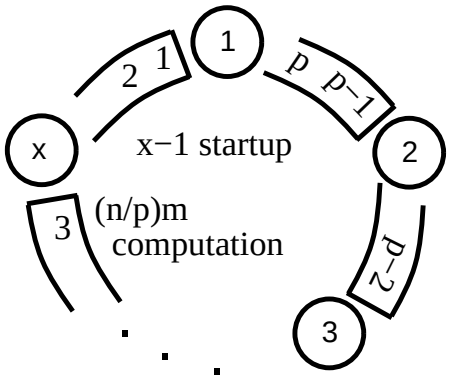

Figure 3: Pipeline upper bound: choose x such that $(x + (n/p)m)(service\_time)$ is minimized. ($service\_time$ depends on x)

3.2 Formal Proof

We model the execution of the pipeline algorithm as a cyclic queueing system. Suppose that p customers are served eternally by x first-come-first-served servers with unbounded queues, numbered $1, \ldots x$, where a customer is served by the $((i \bmod x) + 1)$st server after it has been served by the ith server. We assume service times are all exponentially distributed with mean $1/\mu$. Suppose we observe the queueing system with all customers initially in the queue of server 1, in order with 1 first and p last. Then, the computation of entry (i, j) of the table (by processor $i \bmod p$) corresponds to the service of the $(i \bmod p)$th customer for the $\lceil i/p \rceil$th time by the jth server (where here, processor p is aliased as processor 0). The termination of the algorithm corresponds to the completion of the $m\lceil n/p \rceil$th task of processor p.

Our upper bound on the pipeline algorithm exploits this relationship between the pipeline algorithm and the queueing system. Also, the bound uses Lemma 3.1, which gives the expected time for a customer to be served s times in a system in steady-state. This lemma follows directly from a result of Lavenberg and Reiser [14, Equation 2.17].

Lemma 3.1 *In a steady-state cyclic queueing system with x servers, p customers and exponentially distributed service times at each server with mean $1/\mu$, the expected time for a customer to be served s times is*

$$\frac{s}{\mu}(1 + \frac{p-1}{x}).$$

Our proof of the upper bound must bridge the gap between the system modeling the algorithm, in which the customers are all in the queue of server 1 initially, and the system of Lemma 3.1, which is initially in steady-state. We need the following notation for the proof. Define a state S of the system to be a p-tuple, where $S[i]$ is the server at which customer i resides. Let $T(x, S)$ be the expected time for customer p to be served $m\lceil n/p \rceil$ times in an x-server system that starts in state S, where we only count services of p starting with the first service at server 1 *after* customer 1 has been served there. We consider valid initial states J of the system to be states in which the the customers are in ascending circular order. That is, customer $(i \bmod p)+1$ follows customer i, $1 \leq i \leq p$. Without loss of generality, we assume that in a valid initial state J, customer 1 resides at server 1. Notice that the definition of $T(x, J)$ means that if in state J, customer p is ahead of customer 1 at server 1 (note that this is not

inconsistent with our assumption of circular ordering) then we don't start counting until the second service of customer p at server 1. Let I be the state of the queueing system in which all customers are in the queue of server 1, with customer 1 at the head of the queue. Then, $T(m, I)$ is the expected running time of the pipeline algorithm (this follows from the first paragraph of this section). The following lemma is the key to the proof.

Lemma 3.2 *Let I be the state of the queueing system in which all customers are in the queue of server 1, let J be any valid initial state and let $x \leq m$. Then,*

$$T(m, I) \leq T(x, I) \leq T(x, J).$$

Both inequalities are intuitively true: The first inequality states that if the number of servers is reduced, the expected time for customer p to complete a given number of services increases. The second inequality states that the expected time for customer p to complete a given number of services is greater when the customers are in an arbitrary valid initial state, rather than when the customers are nicely lined up at server 1. The proof of Lemma 3.2 is in the appendix.

The next lemma applies Lemma 3.2 to bound the running time of the pipeline algorithm using a steady-state system.

Lemma 3.3 *The running time of the pipeline algorithm using the random delay model is at most the time for a customer to be served $x + m\lceil n/p \rceil$ times in the cyclic p-customer problem with $x \leq m$ servers, when the system is in the steady-state.*

Proof: Consider again a cyclic queueing system in the steady-state with x ordered queues, $x \leq m$, and p ordered customers, where (by definition), customer 1 is at server 1. Let $S(x)$ be the expected time for customer p to reach server 1 and then to be served $m\lceil n/p \rceil$ times after it has arrived in the queue of server 1 of an x-server system. Note that $S(x)$ is the expectation $T(x, J)$ taken over all valid initial states J. Using Lemma 3.2, we can see that the expected running time of the pipeline algorithm is at most $S(x)$. This is because we already know that the expected running time of the pipeline algorithm is at most $T(m, I)$. By Lemma 3.2, this is at most $T(x, J)$ for any valid initial state J and $x \leq m$. Let J_0 be a valid initial state for which $T(x, J)$ is minimal. Since $S(x)$ is the expectation of $T(x, J)$ over all valid initial states J, we have $T(m, I) \leq T(x, J_0) \leq S(x)$.

Also, $S(x)$ is at most the time for a customer to be served $x + m\lceil n/p \rceil$ times in an x-server system with p customers. This is because the time for customer p to reach the queue of server 1 is at most the time for a customer to be served by x servers (the worst case is when initially, customer p is ahead of customer 1 in the queue for server 1). □

Substituting the value from Lemma 3.1 into Lemma 3.3 we see that the expected time of the pipeline algorithm is at most

$$\frac{1}{\mu}(x + m\lceil n/p \rceil)(1 + \frac{p-1}{x}).$$

This is at most

$$\frac{1}{\mu}(x + m\lceil n/p \rceil + (p-1) + \frac{m\lceil n/p \rceil (p-1)}{x}).$$

To minimize this expression, we choose $x = \left\lceil \sqrt{m\lceil n/p \rceil (p-1)} \right\rceil$. Thus, we have the following theorem.

Theorem 3.1 *The expected running time of the pipeline algorithm is at most*

$$\frac{1}{\mu}\left(m\lceil n/p\rceil + p + 2\sqrt{m\lceil n/p\rceil(p-1)}\right).$$

4 Analysis of Diagonal Algorithm

In this section, we analyze the diagonal algorithm for dynamic programming on our random model. We first prove a general lower bound on the expected running time of the diagonal algorithm. In Section 4.1, we obtain asymptotic estimates for the expected running time of the diagonal algorithm for different values of p, the number of processors, using results on the solution of a well known occupancy problem. Upper bounds on the running time of the diagonal algorithm when the assumption of an exponential distribution is relaxed can be found in [4].

To compute the expected running time of the diagonal algorithm on the random model, we define $T(p,j)$ to be the time for p processors to complete an iteration in which the diagonal contains j entries, where $p \leq j$.

We first obtain an expression for the quantity $T(p,j)$. The time for any one processor to compute $k = \lfloor \frac{j-1}{p} \rfloor$ entries is the sum of k i.i.d. random variables, each of which is exponentially distributed with mean $1/\mu$. This sum has a gamma distribution with parameters μ and k. That is, the density function for the time for a processor to compute k entries is $(\mu^k/\Gamma(k))t^{k-1}e^{-\mu t}$, $t \geq 0$, where $\Gamma(k)$ is the gamma function. Let $M(l,k)$ be the time for l processors to compute k entries each; that is, $M(l,k)$ is the maximum of l i.i.d. random variables which have a gamma distribution with parameters μ and k. Then, $T(p,j)$ is the maximum of $M(l,k+1)$ and $M(p-l,k)$, where $l = j - p\lfloor \frac{j-1}{p} \rfloor$ and $k = \lfloor \frac{j-1}{p} \rfloor$.

In the special case when $j = p$, $E[T(p,j)]$ is the expected maximum of j i.i.d. random variables whose distribution is exponential with mean $1/\mu$. This is known to be $(1/\mu)H_j$, where $H_j = \sum_{i=1}^{j} 1/i$ is the jth harmonic number (Solomon [22]).

We now obtain a lower bound on the expected time of the diagonal algorithm. We first obtain a lower bound on $E[T(p,j)]$. Since in the jth diagonal, each processor must compute at least $k = \lfloor \frac{j-1}{p} \rfloor$ entries, and the time to do this is $M(p,k)$, it follows that $E[T(p,j)] \geq E[M(p,k)]$. Consider the processor which takes the most time in computing the first entry. The expected time taken by this processor to complete k entries is clearly a lower bound on $E[M(p,k)]$. The expected time taken by this processor is $(1/\mu)(k-1)$ (which is the expected time to compute $k-1$ entries, other than the first), plus the expected value of the maximum of p i.i.d. random variables which are exponentially distributed with mean $1/\mu$. We have already seen that this is $(1/\mu)H_p$. Hence, $E[T(p,j)] \geq E[M(p,k)] \geq (1/\mu)(H_p + (k-1))$. Summing over all the diagonals, we obtain the following theorem, whose proof can be found in [4].

Theorem 4.1 *For $p > 1$, the total expected time of the diagonal algorithm is at least*

$$\frac{1}{\mu}[(mn + n(p-1))/p + (m+n+1)(H_{p-1} - 2)].$$

4.1 Reduction to an Occupancy Problem and Resulting Bounds

To obtain further estimates on the running time of the diagonal algorithm, we relate the expected value of $M(p,k)$ to the solution of a well studied occupancy problem. If balls are thrown

randomly and uniformly into p bins, how many balls must be thrown in order that each bin has at least k balls? Let $Occ(p,k)$ be the random variable denoting the number of balls needed to ensure that each bin has k balls. Then we obtain the following relationship between the expected values of $M(p,k)$ and $Occ(p,k)$.

Lemma 4.1 $E[M(p,k)] = \frac{1}{p\mu}E[Occ(p,k)]$.

Proof: This result was proved by Young [24]. We give a simple proof here. Imagine that the processors compute entries forever; then $M(p,k)$ is the time at which all processors have computed at least k entries. Define an *event* to be an instant when some processor finishes computing an entry. By the memoryless property of exponential distributions, the time between events is exponentially distributed with mean μp. Thus, the expected time between events is $1/(\mu p)$. When an event occurs, the processor that finished computing an entry is random (by symmetry). Thus, the expected time for p processors to compute k entries each is

$$\begin{aligned} & \sum_{i=1}^{\infty} \Pr[Occ(p,k) = i](\text{expected time to complete } i \text{ events}) \\ = \; & \frac{1}{\mu p}\sum_{i=1}^{\infty} i\Pr[Occ(p,k) = i] \\ = \; & \frac{1}{\mu p}E[Occ(p,k)]. \end{aligned}$$

□

Using this lemma, we can apply results on the occupancy problem to obtain asymptotic bounds on the expected value of $M(p,k)$. We use the following results on $E[Occ(p,k)]$. In the next lemma, the notation $f(k) \sim g(k)$ denotes that $\lim_{k\to\infty} f(k)/g(k) = 1$.

Lemma 4.2 *1. If p is fixed and $k \to \infty$, then $E[Occ(p,k)] \sim pk$.*

2. If k is fixed and $p \to \infty$, then $E[Occ(p,k)] \sim p(\log p + (k-1)\log\log p)$.

3. If $k = \alpha p$ and $p \to \infty$ then $E[Occ(p,k)] \leq pk(1 + O(\sqrt{\log p/p}))$.

Proof: 1 and 2 follow directly from results of Newman and Shepp [18]. We present the proof of 3. Suppose $k = \alpha p$. We first estimate the number, N, of balls needed to ensure that with probability at least $1 - 1/p$, all bins have at least k balls. Suppose that N balls are thrown in the p boxes. The probability that some bin has less than k balls is at most

$$p\left(\binom{N}{0}(1/p)^0(1-1/p)^{N-0} + \binom{N}{1}(1/p)^1(1-1/p)^{N-1} + \ldots + \binom{N}{k-1}(1/p)^{k-1}(1-1/p)^{N-(k-1)}\right).$$

To bound this sum, we first bound each term in the sum, by noting that for $1 \leq i \leq k-1$,

$$\frac{\binom{N}{i}}{\binom{N}{k}} = \frac{N!}{(N-i)!i!}\frac{k!(N-k)!}{N!} = \frac{k(k-1)\ldots(i+1)}{(N-i)(N-i-1)\ldots(N-(k-1))}$$

$$\leq \frac{k^{k-i}}{(N-(k-1))^{k-i}} \leq \frac{k^{k-i}}{(kp-k)^{k-i}} \quad (\text{since } N \geq kp-1)$$

$$= (\frac{1/p}{1-1/p})^{k-i} = (1/p)^{k-i}(1-1/p)^{i-k}.$$

Multiplying both sides by $\binom{N}{k}(1/p)^i(1-1/p)^{N-i}$, we see that

$$\binom{N}{i}(1/p)^i(1-1/p)^{N-i} \leq \binom{N}{k}(1/p)^k(1-1/p)^{N-k}.$$

Hence, the probability that some bin has less than k balls is at most

$$pk\binom{N}{k}(1/p)^k(1-1/p)^{N-k}.$$

If $N = \beta p$, then $\binom{N}{k} = \binom{\beta p}{k} \leq (\beta p)^k(1/k!)$. Hence the last expression is at most

$$pk(\beta p)^k(1/k!)(1/p)^k(1-1/p)^{\beta p-k} \leq pk\beta^k(1/k!)\exp(-(\beta-\alpha)),$$

since $(1-1/p)^p \leq e^{-1}$ and $k = \alpha p$.

We now find the asymptotic value of β that will make this last expression equal to $1/p$. Taking logarithms and applying Stirling's formula (namely, $k! = \sqrt{2\pi k}(k/e)^k(1+\Theta(1/k))$), we see that if we set this last expression equal to $1/p$ then

$$\log(1/p) \sim \log p + \log k + k\log\beta - 1/2\log(2\pi) - 1/2\log k - k\log k + k - \beta + \alpha.$$

Rearranging the terms, we have

$$k\log\beta - k\log k \sim \beta - k - 2\log p - 1/2\log k + 1/2\log(2\pi) - \alpha.$$

Since $k = \alpha p$, $1/2\log k = 1/2\log\alpha + 1/2\log p$. Letting $c = \alpha + 1/2\log\alpha - 1/2\log(2\pi)$ and dividing across by k, we have that

$$\log(\beta/k) \sim \beta/k - 1 - (5/2\log p + c)/k.$$

Let $\beta/k - 1 = \epsilon$. From the Taylor series expansion, $\log(1+\epsilon) \leq \epsilon - \epsilon^2/2$. Thus, as $p \to \infty$,

$$\epsilon - \epsilon^2/2 \geq \beta/k - 1 - (5/2\log p + c)/k.$$

Equivalently, $\epsilon^2/2 \leq (5\log p + 2c)/(2k)$, that is,

$$\epsilon \leq \sqrt{(5\log p + 2c)/k}.$$

This shows that asymptotically $\beta/k \leq 1 + \sqrt{(5\log p + 2c)/k}$ and so as $p \to \infty$,

$$N = \beta p \leq pk(1 + \sqrt{(5\log p + 2c)/k}).$$

We observe that if not all bins have k balls by N steps, we can continue for N more steps, again for another N if it is not done, and so on. At each stage, the probability of finishing is no worse than the probability that a newly begun process (starting with the all bins empty)

finishes in N steps. Now, we have a chance $\leq 1/p$ of going to the second stage, $\leq 1/p^2$ of going to the third stage, and so on. This implies that the mean waiting time is at most

$$pk(1+\sqrt{(5\log p+2c)/k})(1+1/p+1/p^2+\ldots)=pk(1+\sqrt{(5/\alpha)\log p/p}+O(1/p)).$$

□

Finally, we can obtain the following bounds on the expected running time of the diagonal algorithm with p processors by summing over all diagonals using $M(p,k)$ and applying Lemma 4.2. For details, see [4].

Theorem 4.2 *As $n \to \infty$, the expected running time of the diagonal algorithm with p processors on an $n \times m$ table, $n \leq m$, is*

1. $\sim mn/(\mu p)+\Theta(m)$, *when p is a constant,*
2. $\sim (m+n)(\log p)/\mu+\Theta(m\log\log p)$, *when $p=\Theta(n)$, and*
3. $\leq (m+\epsilon n)(n/p)(1/\mu+o(1))+O(m)$, *when $p=\Theta(\sqrt{n})$, where ϵ is any constant.*

5 Experimental Results

Experiments were run on the diagonal and pipeline algorithms using a Sequent Symmetry computer, an asynchronous, shared memory, parallel machine. The goal of our experiments was to determine empirically, for a small number of processors, the slowdown of both algorithms due to idleness of processors, and to compare this with the slowdown due to the cost of synchronization primitives. In this section, we first report on these experiments. We then compare the experimental results with predictions based on the formulas of Table 2.

5.1 Measured Running, Synchronization and Idleness Times

In our experiments, for each run of the algorithm, we measured the total running time and, in addition, we estimated the *synchronization time*, *idle time* and *average task time* of each run. These were estimated as follows.

For a given run of the diagonal algorithm, the synchronization time at a particular barrier is simply the minimum amount of time spent at the barrier by any of the processors. The total synchronization time is obtained by summing over all barriers. The idle time at a particular barrier is the difference between the maximum time any processor spends on the task prior to that barrier, and the average time spent by all processors on the task prior to that barrier. Summing up over all barriers gives the idle time. Thus, the idle time is the average time (taken over all the processors) that a processor spends idle during a run of the algorithm. Finally, the average task time is the average time (taken over all tasks of all processors) spent by a processor on a task.

We used locks on the Sequent to enable a processor to check when the predecessors of an entry are computed, since locks are a simple and efficient mechanism on the Sequent provided for this purpose. An alternative scheme, to have a special bit per table entry that indicates when it is computed, is also efficient in terms of time. We opted for locks because it enabled us

to use some measurement tools to measure the time spent on the checking of predecessors. On the Sequent, there is no good way to separate the time a processor spends executing the lock from the time the processor spends idle at the lock. Therefore, to estimate the synchronization time for the pipeline algorithm on a given data set, we first calculated the average time spent at a lock on a run of the algorithm with a single processor (where there is no idleness). Then, for a given data set and a given number of processors, we multiplied this average time by the number of locks per processor to obtain the synchronization time. The idle time is the total time spent at locks, by the processor which computes the last row of the table, minus the synchronization time. Again, the average task time is the average time (taken over all tasks of all processors) spent by a processor on a task.

We note that it may be the case that the time to execute a lock on a run of the algorithm with more than one processor is more than the time on a run of the algorithm with a single processor, due to delays in using the bus, for example. If this is the case, we are underestimating the synchronization time and overestimating the idle time for the pipeline algorithm. In spite of this, our results show that the idle time for the pipeline algorithm is less than the idle time for the diagonal algorithm.

In order to increase the work between synchronization points, both algorithms do 4×4 blocks of table entries at a time. (Various sizes were tried before settling on 4 as the block size. This block size gives reasonable performance at various sizes of sequences, compared with other block sizes.) Thus, a task consists of computing 16 table entries. For clarity in our tables, we note the size of the table, followed by the values of n and m as they should be interpreted in the previous sections. In every case, $n = m$ and n, m equal the table size divided by 4.

We draw the following conclusions from our experiments.

- The diagonal algorithm is typically slower than the pipeline algorithm, although the difference in running time is small (see Table 3).
- The difference in running time is not due to the cost of the barriers. In fact, on every run with more than one processor, the synchronization time in the diagonal algorithm is less than the synchronization time in the pipeline algorithm (see Table 4). (Recall that the number of locks executed by any one processor in the pipeline algorithm is $\Theta(n^2/p)$. This explains why, for a given table size, the synchronization time for the pipeline algorithm decreases as p increases.)
- The idle time is always much more in the diagonal algorithm than in the pipeline algorithm, for 8 or more processors (see Table 5).
- In the diagonal algorithm, the idle time is much more costly than the time to execute the barriers. In the pipeline algorithm, the idle time and synchronization times are typically close.
- Despite the simplicity of a task, the variance in computation time of a single task was high (often much larger than the task time itself) in both the diagonal and pipeline algorithms. The variance increased as the number of processors and input size increased (see Table 6). We were unable to determine why this is so. In the case that the number of processors increased, it may be because the more processors that simultaneously used the common bus in order to complete their tasks, the greater the degree of interference between the tasks, which caused some tasks to take a longer time.

Processors	1		4		8		12		16	
Table size	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
500(n, m=125)	16.80	16.79	4.36	4.30	2.33	2.19	1.67	1.54	1.35	1.16
1000(n, m=250)	67.91	67.55	17.33	17.08	9.02	8.69	6.28	5.76	4.93	4.43
1500(n, m=375)	153.93	152.97	39.03	38.43	20.01	19.23	13.72	13.12	10.61	9.91
2000(n, m=500)	274.67	272.03	69.28	68.06	35.26	34.34	24.95	25.33	19.74	17.51
3000(n, m=750)	622.82	613.54	156.94	153.85	79.78	76.98	53.98	51.71	40.97	38.70
4000(n, m=1000)	1111.18	1142.65	279.44	273.09	140.67	136.59	94.83	92.07	72.23	72.70

Table 3: Total running time, in seconds, for several values p and table size.

Processors	4		8		12		16	
Table size	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
500(n, m=125)	0.017	0.026	0.037	0.012	0.056	0.009	0.074	0.006
1000(n, m=250)	0.027	0.101	0.053	0.051	0.079	0.034	0.102	0.026
1500(n, m=375)	0.036	0.226	0.069	0.113	0.103	0.077	0.135	0.058
2000(n, m=500)	0.045	0.401	0.076	0.202	0.076	0.135	0.111	0.103
3000(n, m=750)	0.076	0.906	0.145	0.453	0.213	0.303	0.280	0.226
4000(n, m=1000)	0.099	1.606	0.183	0.803	0.268	0.539	0.354	0.405

Table 4: Synchronization time, in seconds, for several values p and table size.

Processors	4		8		12		16	
Table size	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
500(n, m=125)	0.02	0.02	0.06	0.01	0.07	0.01	0.09	0.01
1000(n, m=250)	0.09	0.10	0.21	0.05	0.27	0.03	0.31	0.02
1500(n, m=375)	0.15	0.22	0.30	0.11	0.37	0.08	0.42	0.06
2000(n, m=500)	0.23	0.40	0.41	0.20	0.57	0.13	0.63	0.11
3000(n, m=750)	0.80	0.87	1.29	0.43	1.21	0.30	1.05	0.23
4000(n, m=1000)	0.70	1.52	1.01	0.76	1.31	0.54	1.28	0.53

Table 5: Idle time, in seconds, for several values p and table size.

5.2 Predicted Running, Synchronization and Idleness Times

To compare our experimental results with our theoretical analysis, we predicted the total running time of the algorithms, and the costs of synchronization and idleness as follows. The synchronization times for the pipeline and diagonal algorithms are ln^2/p and $b_p(2n-1)$ respectively, where l, the cost of a lock is estimated to be 6 microseconds and b_p, the cost of a barrier with p processors is given in Table 7. The cost of a barrier increases as the number of processors increases. These costs for the barriers are based on average measurements taken when the processors executed no instructions other than barriers (which may explain why these numbers are less than our experimental measurements in Table 4).

For our estimate of running times, we first computed the formulas in Table 2, first column (that is, with p a constant) and added the synchronization cost. Based on Table 6, we used 1080 microseconds as our estimate of average task time, $1/\mu$. Thus, our formulas are:

- Predicted running time (pipeline) $= 1/\mu(n^2/p + 2n) + ln^2/p$.

Processors		1		4		8		12		16	
Table size		Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
500	Avg	1959	1065	1061	1063	1070	1064	1080	1065	1090	1066
$(n, m{=}125)$	Var	827	787	1290	1133	3919	1295	6902	1374	9986	1578
1000	Avg	1070	1071	1071	1068	1080	1069	1080	1069	1102	1070
$(n, m{=}250)$	Var	1028	769	2252	1170	7460	1307	13567	1300	21233	1395
1500	Avg	1078	1077	1077	1079	1082	1074	1087	1075	1093	1075
$(n, m{=}375)$	Var	933	21727	1945	4884	4796	4189	7891	4167	114555	4365
2000	Avg	1082	1077	1078	1073	1080	1074	1085	1075	1088	1075
$(n, m{=}500)$	Var	12397	5765	1816	4845	3825	4707	6022	33293	13121	31820
3000	Avg	1089	1079	1084	1077	1084	1077	1086	1077	1088	1077
$(n, m{=}750)$	Var	3075	4517	1546	4245	2685	4164	3902	4223	5342	4235
4000	Avg	1093	1132	1091	1078	1089	1077	1091	1081	1089	1093
$(n, m{=}1000)$	Var	1986	3141770	1363	14289	2129	22126	3055	337709	3979	34954374

Table 6: Average task time and variance (in microseconds) for several values p and table size.

p	1	4	8	12	16
b_p	6	21	34	48	62

Table 7: Estimates of b_p, the cost of a barrier with p processors, on the Sequent (in microseconds).

- Predicted running time (diagonal) $= 1/\mu(n^2/p + 2nH_{p-1} - 3n - n/p - 1) + b_p(2n - 1)$.

To estimate the cost due to idleness, we computed the lower bound given in Table 2 – this is our estimate of the running time, not counting costs of synchronization and idleness. We subtracted this value and the time for synchronization from the predicted running time. Thus,

- Predicted idleness (pipeline) $= 1/\mu(n^2/p + 2n) - 1/\mu(n^2/p + p - 1)$.
- Predicted idleness (diagonal) $= 1/\mu(n^2/p + 2nH_{p-1} - 3n - n/p - 1) - 1/\mu(n^2/p + p - 1)$.

All of the predicted values are listed in Table 8. Our conclusions are as follows.

- The predicted running times are qualitatively correct for 8 or more processors, that is, they predict correctly that the diagonal algorithm is slower than the pipeline algorithm. (Our lower bound for the diagonal algorithm given in Table 2 is not good for 4 processors, leading to a poor prediction). The difference in running time between the diagonal and pipeline algorithms is small and increases as p increases. This is as expected, since the work done in both algorithms is quadratic in n whereas the cost due to idleness is predicted to be linear in n (for fixed p). However, our predictions are somewhat pessimistic, that is, most of the numbers overestimate the actual running time. The percentage error of the predicted results tends to decrease as the input size increases.
- For the input sizes we considered, synchronization costs are insignificant, representing less than 1% of the total running time in both algorithms. Since the the cost of barriers is linear in n and the running time of the algorithm is $\Theta(n^2/p)$, the relative cost of barriers increases as p increases and decreases as n increases. In contrast, since the cost of locks

is $\Theta(n^2/p)$, the percentage of total running time due to locks is fairly constant, at about .6%.

- The cost of idleness is much larger than the cost of synchronization for all entries in the table, being highest in the diagonal algorithm when $p = 16$, where for $n = 125$ it is 30% of the total running time and decreases as the input size increases to be about 5% of the total running time when $n = 1000$. Clearly, as n increases further, the relative cost of idleness continues to decrease in both algorithms.

- Since as n increases, the relative cost of locks stays constant but the relative cost of barriers and idleness decreases, we should expect that eventually the diagonal algorithm should beat the pipeline algorithm (for any fixed number of processors). Using our numbers for average task time, locks and barriers, with $p = 16$, the "break even" point at which the predicted performance of both algorithms is equal occurs when n is approximately 4,000 (that is, the table size is 16,000). Since our estimates appear to be pessimistic however, especially for the pipeline algorithm, the break even point is probably smaller on the Sequent.

5.3 Timing Details

The application implemented was the Needleman-Wunsch algorithm for aligning two DNA sequences [17]. In aligning the two sequences, gaps may be inserted in one or the other of the sequences in order to make the overall alignment better. The best alignment is determined by scoring all possible alignments using a dynamic programming algorithm. The scoring of a table entry is a very simple task.

Both algorithms were implemented in the C programming language, using the synchronization routines provided in the Sequent Parallel Programming Library. The diagonal algorithm was implemented using barriers, the pipeline using locks. Each algorithm was run ten times on six different data sets, ranging from sequences of length 500 to sets of length 4000. The experiments were run on a dedicated machine, (thus avoiding effects of other jobs on the dynamic programming algorithms). The runs were done on 1, 4, 8, 12 and 16 processors.

The Sequent provides a good timing facility (the *getusclk()* function). Both algorithms take the same number of timings, minimizing the effect of timing the algorithms on the relative performance. The results of the timings were buffered and output after the algorithm completed, preventing variations in time due to i/o effects.

6 Conclusions

We have examined a very simple model of parallel computation that models unpredictable delays on processors. We have shown that it is feasible to analyze this model for two parallel dynamic programming algorithms. We have experimentally shown that the analysis is also realistic, accurately reflecting the effects of unexpected delays on the running times of the two algorithms. The techniques used in our analysis can likely be useful in analyzing other parallel algorithms, since they apply to situations using pipelining or barriers. Our experimental work supports our analysis.

Processors	4		8		12		16	
Table size	Diag	Pipe	Diag	Pipe	Diag	Pipe	Diag	Pipe
500(n, m=125)	4.28	4.52	2.40	2.39	1.81	1.69	1.55	1.33
% Actual								
Running Time	98%	105%	102%	109%	108%	110%	115%	115%
Synchronization	.01	.03	.01	.01	.01	.01	.01	.01
Idleness	.05	.27	.27	.26	.48	.26	.46	.25
1000(n, m=250)	17.00	17.52	9.01	9.03	6.45	6.20	5.21	4.78
% Actual								
Running Time	98%	103%	100%	104%	102%	107%	106%	108%
Synchronization	.01	.11	.02	.06	.02	.04	.04	.03
Idleness	.11	.53	.27	.26	.48	.26	.46	.25
1500(n, m=375)	38.15	39.02	19.84	19.91	13.89	13.54	10.99	10.36
% Actual								
Running Time	98%	102%	99%	104%	101%	103%	104%	105%
Synchronization	.02	.25	.03	.12	.04	.08	.05	.06
Idleness	.26	.81	.81	.79	1.18	.79	.33	.79
2000(n, m=500)	67.74	69.01	34.89	35.05	24.14	23.72	18.87	18.06
% Actual								
Running Time	98%	101%	99%	102%	97%	94%	96%	103%
Synchronization	.02	.44	.03	.22	.05	.15	.06	.11
Idleness	.22	1.07	1.10	1.08	1.58	1.06	1.91	1.06
3000(n, m=750)	152.23	154.46	77.65	78.04	53.08	52.57	40.95	39.83
% Actual								
Running Time	97%	100%	97%	101%	98%	102%	100%	103%
Synchronization	.03	.98	.05	.49	.07	.33	.09	.25
Idleness	.33	1.61	1.66	1.51	2.39	1.61	2.89	1.60
4000(n, m=1000)	270.46	272.13	137.28	138.02	93.28	92.73	71.48	70.09
% Actual								
Running Time	97%	100%	98%	101%	98%	101%	99%	96%
Synchronization	.04	1.8	.07	.88	.09	.58	.12	.44
Idleness	.44	2.15	2.21	2.15	3.19	2.15	3.74	2.14

Table 8: Predicted running time, in seconds, for several values of p and table size. The predicted running time as a percentage of the actual running time is listed. The predicted costs of synchronization and idleness are also listed.

In future work, we would like to extend the analysis for the pipeline case for some non-exponential distributions, as in the diagonal algorithm. A theorem by Glynn and Whitt [8] can be used to get a very weak upper bound of $(m\lceil n/p \rceil)/\mu + O(nm^{1-a/2})$, when $p = \Theta(m^a)$, for $0 < a \leq 1$.

7 Acknowledgements

B. Narendran pointed out an error in an earlier proof of Theorem 3.1 and provided invaluable help in correcting the proof. Thanks also to Anton Rang, Vikram Adve, Jim Dai, Deborah Joseph, Tom Kurtz, Rajesh Mansharamani, Prasoon Tiwari and Mary Vernon for many helpful comments on the work described in this paper, and to Jeff Hollingsworth and Bruce Irvin for their assistance in obtaining our experimental measurements. Finally, we thank the anonymous

referees for their careful comments which greatly improved the presentation.

References

[1] Almquist, K., R. J. Anderson and E. D. Lazowska. The measured performance of parallel dynamic programming implementations, *Proceedings of the International Conference on Parallel Processing*, III, pages 76-79, 1989.

[2] Anderson, R. J., P. Beame and W. L. Ruzzo. Low overhead parallel schedules for task graphs, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11-21, 1990.

[3] Arjas, E. and T. Lehtonen. Approximating many server queues by single server queues, *Mathematics of Operations Research* 3, pages 205-233, 1978.

[4] Lewandowski, G, A. Condon and E. Bach. Asynchronous Analysis of Parallel Dynamic Programming Algorithms. Technical Report Number 1212, Computer Sciences Department, University of Wisconsin, Madison, WI 53706, February 1994.

[5] Cole, R. and O. Zajicek. The expected advantage of asynchrony, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 85-94, 1990.

[6] Fortune, S. and J. Wylie. Parallelism in random access machines, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114-118, 1978.

[7] Fromm, H., U. Hercksen, U. Herzog, K. John, R. Klar and W. Kleinoder. Experiences with performance measurement and modeling of a processor array, *IEEE Transactions on Computers*, 32(1), pages 15-31, 1983.

[8] Glynn, P. W. and W. Whitt. Departures from many queues in series, Technical Report Number 60, Department of Operations Research, Stanford University.

[9] Goldschlager, L.M. A unified approach to models of synchronous parallel machines, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 89-94, 1978.

[10] Horowitz, E. and S. Sahni. Computing partitions with applications to the knapsack problem, *Journal of the ACM* 21, pages 277-292, 1974.

[11] Kruskal, C.P. and A. Weiss. Allocating independent subtasks on parallel processors, *IEEE Transactions on Software Engineering*, Vol SE-11, No. 10, pages 1001-1016, 1985.

[12] Lai, T.L. and H. Robbins. Maximally dependent random variables, *Proceedings of the National Academy Sciences*, vol 73, no. 2, pages 286-288, 1976.

[13] Lander, E., J. P. Mesirov and W. Taylor IV. Study of protein sequence comparison metrics on the Connection Machine CM-2, *The Journal of Supercomputing*, 3, pages 255-269, 1989.

[14] Lavenberg, S. S. and M. Reiser. Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers, *Journal of Applied Probability*, pages 1048-1061, 1980.

[15] Mak, V. W. and S. F. Lundstrom. Predicting performance of parallel computations, *IEEE Transactions on Parallel and Distributed Systems*, 1(3), pages 257-270, 1990.

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

[17] Needleman, S. B., C.D. Wunsch. A general method applicable to the search for similarity in the amino acid sequence of two proteins, *Journal of Molecular Biology* 48, pages 443-454, 1970.

[18] Newman, D. J. and L. Shepp. The double dixie cup problem. *The American Mathematical Monthly*, 67, pages 58-61, 1960.

[19] Nishimura, N. Asynchronous shared memory parallel computation. *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 76-84, 1990.

[20] Purdom, P. W. and C. A. Brown. *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.

[21] Savitch, W.J., M. Stimson. Time bounded random access machines with parallel processing, *Journal of the ACM*, 26, pages 103-118, 1979.

[22] Solomon, F. Residual lifetimes in random parallel systems, *Mathematics Magazine*, 63(1), pages 37-48, 1990.

[23] Wagner, R.A., M.J. Fischer. The string-to-string correction problem, *Journal of the ACM* 21, pages 168-173, 1974.

[24] Young, D. H. Moment relations for order statistics of the standardized gamma distribution and the inverse multinomial distribution, *Biometrika*, 58(3), pages 637-640, 1971.

A Appendix

Proof of Lemma 3.2

In the proof, it is useful to use the following numbering of services of customer i, in a queueing system with x servers that starts in a valid initial state J. The services of customer 1 (which by definition, is at server 1) are numbered $1, 2, 3$ and so on. If $i \neq 1$, then service number 1 of customer i is the first service at server 1 *after* customer 1 has been served there. Successive services following service number 1 are numbered 2, 3 and so on. The jth service of customer i that precedes service number 1, if any (which occurs at server numbered $x - j + 1$) is numbered $1 - j$, where $j \leq x$. When $j = x$, customer i is ahead of customer 1 in the initial state J; this is not inconsistent with our assumption of circular ordering, since customer 1 may not be at the head of the queue. Fact A.1 follows directly from this numbering scheme and the fact that the customers remain in circular order at all times.

Fact A.1 *Consider an x-server queueing system that starts in a valid initial state J. Let $a_i(t)$ denote the number of the last completed service of customer i at time t.*

1. *If $i > 1$ then $a_i(t) \leq a_{i-1}(t)$ and if $i = 1$, $a_i(t) \leq a_p(t) + x$.*
2. *If $i > 1$ then $a_i(t) = a_{i-1}(t)$ if and only if customers $i + 1$ and i are in the same queue at time t (with $i - 1$ just ahead of i). If $i = 1$ then $a_i(t) = a_p(t) + x$ if and only if customers 1 and p are in the same queue at time t with p just ahead of 1.*

Note that $T(x, J)$ is the expected time for customer p to complete its service numbered $m\lceil n/p \rceil$, if the system is in state J at time 0.

Proof: (of Lemma 3.2). We first show that $T(m, I) \leq T(x, I)$. The idea is that both of these expected values are integrals over random number sequences that define the time for each service (the sequences are defined precisely below). To show that $T(m, I) \leq T(x, I)$, we show that for each random number sequence and each value t of time, the system with m servers has at least as many completed services of each customer as does the system with x servers.

A random number sequence $r_1, r_2, r_3, \ldots$ defines the durations of services in both queueing systems as follows. The duration of service number k of customer i is r_u, where $u = i + p(k-1)$. Since we are considering both systems starting from the special initial state I, all services of all customers are numbered by a positive number.

Let the m-server system and the x-server system be called system A and system B, respectively. In system A, for each customer i, let $a_i(t)$ be the number of the last completed service at time t. Initially $a_i(0) = 0$ for $1 \leq i \leq p$. Also, let $a'_i(t)$ be the amount of time until the next service is completed. By definition, $a'_i(t)$ is always positive, because if customer i completes a service at time t, it is considered to be at the queue of the next server and $a_i(t)$ is the amount of time until service is completed at this new server. Initially, $a'_i(0) = r_1 + r_2 + \ldots + r_i$. Define $b_i(t)$ and $b'_i(t)$ similarly for system B.

We say system A *is ahead of* system B at time t if it is the case that for all i, (a) $a_i(t) \geq b_i(t)$ and (b) if $a_i(t) = b_i(t)$ then $a'_i(t) \leq b'_i(t)$. We next show by induction on t that system A is ahead of system B at all times t. The first inequality of the lemma follows from this.

The base case is when $t = 0$. In this case, by definition, $a_i(0) = b_i(0) = 0$ and $a'_i(0) = b'_i(0) = r_1 + r_2 + \ldots + r_i$ for all i.

Now, suppose that system A is ahead of system B at time t_1 and consider time $t_2 > t_1$ such that no service is completed after t_1 and before t_2. We need to show that for each customer i, (a) $a_i(t_2) \geq b_i(t_2)$ and (b) if $a_i(t_2) = b_i(t_2)$, then $a'_i(t_2) \leq b'_i(t_2)$. There are three cases.

(i) The service of customer i is not completed in either system. That is, $a_i(t_2) = a_i(t_1)$ and $b_i(t_2) = b_i(t_1)$. By the inductive hypothesis, $a_i(t_1) \geq b_i(t_1)$ and so $a_i(t_2) \geq b_i(t_2)$, proving that (a) holds. To prove that (b) holds, we note that since $t_2 - t_1$ time has elapsed, $a'_i(t_2) = a'_i(t_1) - (t_2 - t_1)$ and $b'_i(t_2) = b'_i(t_1) - (t_2 - t_1)$. If $a_i(t_2) = b_i(t_2)$, then $a_i(t_1) = b_i(t_1)$ and so by part (b) of the inductive hypothesis, $a'_i(t_1) \leq b'_i(t_1)$. Substituting this into the equations of the previous sentence, we conclude that $a'_i(t_2) \leq b'_i(t_2)$.

(ii) The service of customer i is completed in system A but not in system B at time t_2. That is, $a_i(t_2) > a_i(t_1)$ and $b_i(t_2) = b_i(t_2)$. From the inductive hypothesis, $a_i(t_1) \geq b_i(t_1)$. Putting these three inequalities together we conclude that $a_i(t_2) > b_i(t_2)$. Therefore (a) holds, and (b) holds vacuously.

(iii) The service of customer i is completed in system B at time t_2. That is, $b_i(t_2) = b_i(t_1)+1$ and $a_i(t_2) \geq a_i(t_1)$. If $a_i(t_1) > b_i(t_1)$ then clearly $a_i(t_2) \geq b_i(t_2)$. If $a_i(t_1) = b_i(t_1)$ we claim that the service of customer i in system A is completed also at time t_2, that is, $a_i(t_2) = a_i(t_1)+1$ (and so again $a_i(t_2) \geq b_i(t_2)$). This is because at time t_1, by the inductive hypothesis, the amount of time until the service of customer i is completed in system A is less than or equal to the amount of time until the service of customer i is completed in system B, that is, $a'_i(t_1) \leq b'_i(t_1)$. Thus if the service of customer i completes at time t_2 and no service is completed between times t_1 and t_2, it must be the case that the service of customer i is also completed in system A at exactly

time t_2. We conclude that $a_i(t_2) \geq b_i(t_2)$ in case (iii).

It remains to show that (b) holds for case (iii), that is, if $a_i(t_2) = b_i(t_2)$ then $a'_i(t_2) \leq b'_i(t_2)$. Therefore suppose that $a_i(t_2) = b_i(t_2)$. Then the next service of customer i to be completed in both systems has duration r_u where $u = i + pa_i(t_2)$ $(= i + pb_i(t_2))$. If in system A there is nothing ahead of customer i on the queue at which customer i resides at time t_2 then $a'_i(t_2) \leq r_u$. Also $b'_i(t_2) \geq r_u$ since the next service of customer i in system B has not yet started. Therefore $a'_i(t_2) \leq b'_i(t_2)$, as required. Otherwise in system A, in the queue of customer i at time t_2, there is some customer ahead of customer i. This is customer $i-1$ if $i > 1$ and is customer p if $i = 1$. That is, from Fact A.1, $a_{i-1}(t_2) = a_i(t_2)$ if $i > 1$ and $a_p(t_2) + m = a_i(t_2)$ if $i = 1$. We will suppose that $i > 1$ and complete the argument by referring to customer $i-1$. The argument is similar when $i = 1$, with $a_p(t_2) + m$ and $b_p(t_2) + m$ substituted for $a_{i-1}(t_2)$ and $b_{i-1}(t_2)$, respectively. We claim that also in system B, in the queue of customer i at time t_2, customer $i-1$ must be ahead of customer i. To see this, we know from the proof so far and the inductive hypothesis that $a_{i-1}(t_2) \geq b_{i-1}(t_2)$. Also, we are assuming that $b_i(t_2) = a_i(t_2) = a_{i-1}(t_2)$. Hence, $b_i(t_2) \geq b_{i-1}(t_2)$. By Fact A.1 part 1, it must be that $b_i(t_2) = b_{i-1}(t_2)$ and thus by part 2, both i and $i-1$ are at the same queue, with $i-1$ ahead of i, as we claimed. Therefore in both systems, in the queue of customer i at time t_2, customer $i-1$ is just ahead of customer i. We conclude that customer $i-1$ did not complete a service in either system at time t_2. Case (i) above therefore implies that $a'_{i-1}(t_2) \leq b'_{i-1}(t_2)$. Also, $a'_i(t_2) = a'_{i-1}(t_2) + r_u$ and $b'_i(t_2) = b'_{i-1}(t_2) + r_u$. The last three inequalities immediately yield that $a'_i(t_2) \leq b'_i(t_2)$.

We next show that $T(x, I) \leq T(x, J)$. The proof of this is similar to the proof that $T(m, I) \leq T(x, I)$. Namely, both $T(x, I)$ and $T(x, J)$ are expectations over random number sequences, and we show that for each random number sequence and each value t of time, in the system with initial state I, which we call system A, there are at least as many completed services of each customer as in the system with initial state J, which we call system B.

A random number sequence $\ldots, r_{-2}, r_{-1}, r_0, r_1, r_2, \ldots$ defines the duration of services in both queueing systems as before. Recall that services of customer i with a non-positive index represent services of customer i that are completed before service number 1 of customer i, where service number 1 of customer i is the first service of customer i at server 1 that occurs after customer 1 has already been served there. The duration of service number k of customer i is r_u, where $u = i + p(k-1)$. Now however, since some of the initial services of customer i may have negative numbers (that is, k may be negative), we extend our random number sequence to have numbers with negative indices. Thus, $r_0, r_{-p}, r_{-2p}, \ldots$ are the durations of services numbered $0, -1, -2, \ldots$ of customer p; $r_{-1}, r_{-p-1}, r_{-2p-1} \ldots$ are the durations of services numbered $0, -1, -2, \ldots$ of customer $p-1$ and so on. The lowest possible index needed is $2-px$; this is the time for the service numbered $-(x-1)$ of customer 2 (if needed).

We define $a_i(t), a'_i(t), b_i(t)$ and $b'_i(t)$ just as in the previous proof and again use induction to show that system A is ahead of system B at all times t. With our set-up, the proof is almost identical except for the base case. Now when $t = 0$, $a_i(0) = 0$ for all i, and $b_i(0) \leq 0$ for all i (if the first service of i in system B is numbered f, where $-(x-1) \leq f \leq 1$, then $b_i(0) = f - 1$). Also, $a'_i(0) = r_1 + r_2 + \ldots + r_i$. If $b_i(0) = 0$ then $b'_i(0) = r_1 + r_2 + \ldots r_i$.

The only other difference in the proof is in case (iii) of the inductive proof, where m should be replaced everywhere by x, since now system A is an x-server system and not an m-server system.

□