Asynchronous Analysis of Parallel Dynamic Programming Algorithms

Gary Lewandowski
Anne Condon
Eric Bach

Technical Report #1212

February 1994

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 both simple to analyze and accurately 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.

1 Introduction

Parallel algorithms can suffer significant slowdown due to unpredictable delays in the system. These delays include such things as 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.

[*]Work support by WARF grant 135-3094, email gary@cs.wisc.edu

[†]Work supported by NSF grant number CCR-9100886, email condon@cs.wisc.edu

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

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 the model should be easy to use and analyze. By general, we mean 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, that assumes 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 on running time of unpredictable delays in the system, and the consequent idleness of processors. 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 $(i, j-1), 1 \leq i \leq n, 1 \leq j \leq m$. (The entries in the first row and column have only 1 predecessor.) Dynamic programming is a classic algorithmic technique (it is used, for example, to solve the Knapsack problem [9], the Longest Common Substring problem [22], queueing network models [1], and DNA sequence alignment [16]). There are many possible parallel algorithms to implement it. The two dynamic programming algorithms we will examine are the *pipeline* and *diagonal* algorithms.

In the *pipeline* algorithm, when the number of processors is p, 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 accomplished using barriers, for example. Lander et al. [12] proposed this parallel algorithm for protein sequence alignment. The difference between the algorithms is that in the diagonal algorithm, all processors are forced to wait at a barrier, whereas in the pipeline algorithm, a processor can compute an entry once its 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 [5] [8] [20], 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 perform better. In the diagonal algorithm there are $\Theta(n)$ barriers, while in the pipeline algorithm, each processor accesses $\Theta(n^2/p)$ locks, where p is the total number of processors. Our experiments found that the overhead cost of executing a barrier with thirteen processors on a Sequent Symmetry is no more than three times the overhead cost of executing a lock (and less for fewer processors). Thus, even for large p, the overhead costs of $\Theta(n^2/p)$ locks will dominate the overhead costs of $\Theta(n)$ barriers and 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.

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.

Our model is intended simply to model the cost to the running time of idleness of processors, due to unpredictable delays in a parallel system. 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 use the exponential distribution for a variety of reasons. Empirically, 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. Also, performance folklore holds that as long as some variation exists, mean output measures are insensitive to the distribution. Finally, we use the exponential distribution because it makes analysis of the pipeline case possible. Our experiments on the two algorithms confirm that the results we get by using the exponential distribution are qualitatively accurate.

1.3 Overview of Results

1.3.1 Theoretical analysis

Our analytic results show that the expected running time of the diagonal algorithm is significantly worse than the expected running time of the pipeline algorithm. The difference increases as the number of available processors increases, indicating that as technology provides more processors, the advantages to using the pipeline algorithm will become even larger.

To get an idea of how our results on the two algorithms compare, Table 1 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 (see Table 1).

	p a constant	$p = \sqrt{n}$	$p = n$
lower bound	$n^2/p + O(1)$	$n\sqrt{n} + O(\sqrt{n})$	$2n - 1$
upper bound, pipeline	$n^2/p + 2n + O(1)$	$n\sqrt{n} + 2n + O(\sqrt{n})$	$4n$
lower bound, diagonal	$n^2/p + 2nH_{p-1}$	$n\sqrt{n} + n\log n - 3$	$2n\log n + \Theta(n\log\log n)$

Table 1: Asymptotic running times for three cases of p ($n \times n$ table, $\mu = 1$)

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. Moreover, processor idleness was a major contributor to 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 variance 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 in the computation a random variable. Nishimura [18], Cole and Zajicek [4] and Martel et al. [15] described 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. [6] used 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. Thus, the time for an instruction is modeled as a random variable. Different distributions were considered, including constant, exponential and "phase-type" distributions, and the results of the analysis were compared with experimental results. The authors concluded that the analysis using the exponential distribution compared favorably with experimental results. More recently, Mak and Lundstrom [14], described 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 [10] analyzed the expected running time of p processors working on a pool of n subtasks. Each subtask could be done independently. They were interested in finding the best way to allocate the subtasks to each of the p processors, and concluded that giving an equal number of subtasks to each processor all at once 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.

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, it is useful to refer to the set of entries $\{(i,j) \mid i + j = d + 1\}$ as the dth diagonal of the table. We partition the computation into three phases. The first phase of the computation lasts until all the entries $(1,1), (2,1), \ldots (p-1,1)$ are computed. The second phase of the computation lasts from the end of the first phase, until all the entries lying in diagonal number $n+m-p$ are computed. The third phase begins when the second phase ends and lasts to the end of the computation.

Since the entries $(1,1), (2,1), \ldots, (p-1,1)$ must be computed sequentially, the expected time to complete the first phase is at least $(p-1)/\mu$. The computation of any entry in the pth diagonal cannot be started until the first phase is completed. Hence, in the second phase, all entries between diagonals numbered p and $n + m - p$ must be computed. Thus at least $mn - p(p-1)$ entries are computed in the second phase. The expected time to do this is at least $(mn - p(p-1))/(\mu p)$, since some processor must compute at least $mn - p(p-1)/p$ entries and the expected time to do each one is $1/\mu$. Finally, suppose that entry e is the last entry to be computed in the second phase. Then entry e lies on diagonal $n + m - p$. There is a path of $p - 1$ entries from this entry e to the bottom right entry of the table, such that the entries on this path must be computed sequentially in order. The expected time to do these entries 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

Before giving a rigorous proof of the upper bound on the expected time for the pipeline algorithm, we present here the intuition behind the analysis. Assume that n is divisible by p. This is not necessary for the analysis, but makes the intuition more clear.

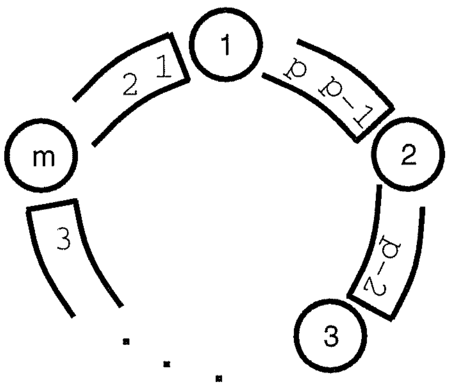

Figure 1: Cyclic p customer problem: How long for p to be served k times?

We relate the pipeline algorithm to a problem from queueing theory. Consider a cyclic queueing system where p customers endlessly circulate through m servers and queues, where the service times are exponentially distributed. 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.) If the system is in steady state, this problem has a known solution: $(s/\mu)(1 + (p-1)/m)$ (Lavenberg & Reiser [13]).

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.)

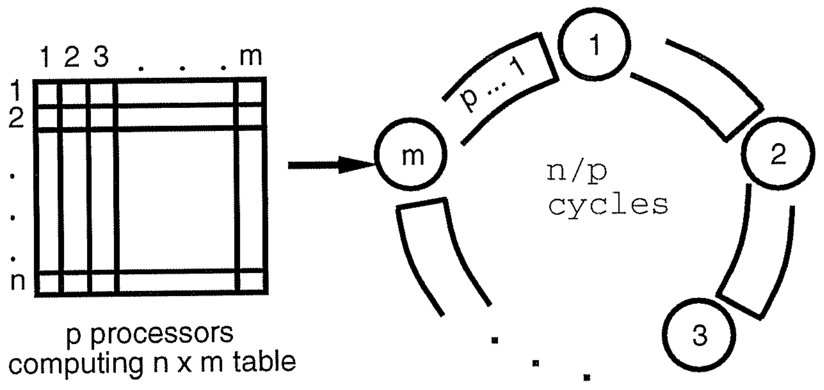

Figure 2: Pipeline algorithm: how long for the pth customer to be served $(n/p)m$ times?

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, determining the expected time for the pth customer to be served $(n/p)m$ times 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. This guarantees that every customer will have been served before 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.)

Thus, the cyclic p-customer problem gives the tightest upper bound for the expected running time of the pipeline algorithm when we minimize $(x-1+(n/p)m)(service\_time)$, where x is the number of servers in the cyclic p-customer problem and the service time is $(1/\mu)(1+(p-1)/m)$. Minimizing this equation balances the estimation cost of the startup with the cost of being served $(n/p)m$ times in a smaller system. Simple calculus provides us with the best value, which is approximately $\sqrt{mn}$ (see Theorem 3.1).

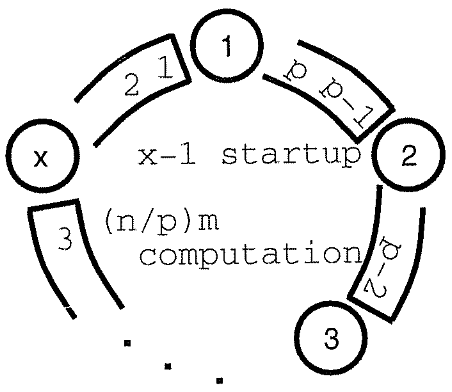

Figure 3: Pipeline upper bound: choose x such that $(x - 1 + (n/p)m) * (service\_time)$ is minimized. $(service\_time$ depends on $x)$

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 that the system is in steady-state and that service times are all exponentially distributed with mean $1/\mu$. The following lemma, which follows from a result of Lavenberg and Reiser [13], gives the expected time for a customer to be served s times.

Lemma 3.1 *In a steady-state cyclic queue 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*

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

In the pipeline algorithm, there are two stages of the computation. The first is the start-up stage, which extends from the start of the algorithm until the $(p-1)$st processor has computed one cell. The second stage extends from the end of the first stage until the end of the algorithm. Suppose we observe the queueing system when in the steady state and let the first server be the one that the first customer is at when we start observing the system. The first stage of the pipeline algorithm corresponds to the period of time from the point at which we observe the system until the time that the pth customer arrives at the queue of the first server. In the second stage of the computation, the computation of cell (i, j) by processor $(i - 1) \bmod p + 1$ corresponds to the service of the $((i - 1) \bmod p + 1)$)st customer for the $\lceil i/p \rceil$th time by the jth server.

For a cyclic queueing system with p customers and x queues, let $S_x(k, j)$ be the time for customer k to reach server 1 and then to be served j times after it has arrived in the queue of server 1. We will use $S(k, j)$ instead of $S_x(k, j)$ when the number of servers is clear from context. Let $C(k,j)$ be the time needed for processor k to compute its first j entries of the table, when executing the pipeline algorithm.

Lemma 3.2 *For $x \leq m$, $C(k,j) \leq S_x(k,j)$, for any $j \geq 1$ and $1 \leq k \leq p$.*

The proof of this lemma is a straightforward but tedious application of previous results on stochastic ordering of variables (Arjas and Lehtonen [3]); details can be found in the Appendix. Using Lemma 3.2 we get the following upper bound on the expected running time of the pipeline algorithm.

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 $x-1+m\lceil n/p\rceil$ times in the cyclic p-customer problem with $x \leq m$ servers, when the system is in the steady-state.*

Proof: From Lemma 3.2, the running time of the pipeline algorithm, which is at most $C(p, m\lceil n/p\rceil)$, is at most $S(p, m\lceil n/p\rceil)$. We now show that $S(p, m\lceil n/p\rceil)$ is at most the time for a customer to be served $x-1+m\lceil n/p\rceil$ times.

$S(p, m\lceil n/p\rceil)$ is the time for customer p to reach server 1 plus the time for customer p to be served $m\lceil n/p\rceil$ times once it has reached the queue of server 1. The time for customer p to reach the queue of server 1 is at most the time for a customer to be served by $x-1$ servers. This is because customer p must pass through at most $x-1$ servers before being served by server 1 for the first time. Hence, $S(p, m\lceil n/p\rceil)$ is at most the time for a customer to be served $x-1+m\lceil n/p\rceil$ times, as required. □

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

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

This is at most

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

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*

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

4 Analysis of Diagonal Algorithm

In this section, we analyze the diagonal algorithm for dynamic programming on our random model. In Section 4.1, we prove a general lower bound on the expected running time of the diagonal algorithm. In Section 4.2, 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. In Section 4.3, we relax the assumption of an exponential distribution for certain values of p. This also yields a better result for the exponential distribution in the case of $(p \log p)^{1+\epsilon} = O(n^{3/4})$.

Before measuring the expected running time of this algorithm on our random model, it is useful to first compute its running time on a simpler, non-random model, in which every entry can be computed in exactly 1 time unit. In this case, the total time required by the algorithm is

$$2\sum_{i=1}^{n-1}(\lfloor\frac{i-1}{p}\rfloor+1)+\sum_{i=n}^{m}(\lfloor\frac{n-1}{p}\rfloor+1).$$

If p divides n, this quantity is $(mn+n(p-1))/p$. Also, $(mn+n(p-1))/p$ is a lower bound on the total running time even when n does not divide p.

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, 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 is the jth harmonic number (Solomon [21]).

4.1 Lower Bound

We now obtain a lower bound on the expected time of the diagonal algorithm. We first obtain a lower bound on $T(p,j)$. 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 $T(p,j) \geq M(p,k)$. Consider the processor which takes the most time in computing the first entry. The time taken by this processor to complete k entries is clearly a lower bound on $M(p,k)$. 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, $T(p, j) \geq M(p, k) \geq (1/\mu)(H_p + (k-1))$. Summing over all the diagonals, we arrive at the following theorem.

Theorem 4.1 *The total expected time of the diagonal algorithm is at least*

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

The proof of Theorem 4.1 can be found in the Appendix.

4.2 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 [23]. 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

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

□

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 $E[Occ(p, k)]$.

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 [17]. 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

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

$$\leq pk\binom{N}{k}(1/p)^k(1-1/p)^{N-k}, \text{ since } N \geq kp - 1.$$

If $N = \beta p$, this 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 $(1-1/p)^p \leq e^{-1}$.

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, we see that $\beta \sim k\log\beta - k\log k + k + 5/2\log p + c$, where $c = \alpha + 1/2\log\alpha - 1/2\log(2\pi)$. Equivalently,

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

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,

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

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

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

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. We defer the details of the proof to the Appendix.

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.*

4.3 Relaxing Assumption of Exponential Distribution

The analysis of the diagonal algorithm can be extended beyond the assumption of exponentially distributed task lengths to include all distributions with an increasing failure rate (defined below). Many distributions fit this criterion, including the exponential, gamma, Weibull, truncated Normal (i.e. Normal restricted to positive values), Uniform and constant distributions. This assumption is sufficient to extend the analysis in the cases where the number of processors used is more than a constant but less than the number of rows in the table.

The analysis itself is straightforward, following from results by Kruskal and Weiss [10]. They prove a theorem that can be used to asymptotically determine the maximum expected value of the time for p processors to compute K subtasks each, as long as K grows faster than p.

We can apply this theorem on each diagonal in our $n \times m$ diagonal except for the upper left-hand corner and lower right-hand corner, where the number of tasks will not be large enough to apply the theorem. Our bound is proved by overestimating these corners, and summing up the values on the rest of the diagonals.

We now present the results and definitions necessary to apply the Kruskal and Weiss result.

Definition 4.1 *Distribution function $G(x)$ has an Increasing Failure Rate or, is IFR, if*

1. $G(0) = 0$.
2. *For all $t > 0$ $(1 - G(x+t))/(1 - G(x))$ is monotone decreasing in x.*

The exponential distribution is IFR since it has a constant failure rate. Other IFR distributions include the Constant distribution, Gamma distribution when $\mu/\sigma \geq 1$, Weibull with rate ≥ 1, Truncated Normal, and the Uniform distribution on (0,C), $C > 0$.

Definition 4.2 *Let $Y_1, \ldots, Y_p$ be random variables with the common distribution function $G(x) = P(Y \leq x)$.*

The characteristic maximum $m_p \overset{\text{def}}{=} \inf\{x : 1 - G(x) \leq 1/p\}$.

Lai and Robbins [11] studied the relationship between the characteristic maximum and the expected maximum. Kruskal and Weiss [10] proved the following lemma relating the Lai and Robbins results to distributions with an increasing failure rate.

Lemma 4.3 *If $G(x)$ is IFR then no matter how $Y_1, \ldots, Y_p$ are dependent,*

$$\lim_{p \to \infty} \frac{E(max(Y_1, \ldots, Y_p))}{m_p} \leq 1$$

.

When $Y_1, \ldots, Y_p$ are independent, equality holds.

Let T be the amount of time it takes for p processors to compute pK tasks, and Y_i be the amount of time processor i spends doing its K tasks. Lemma 4.3 tells us that for p large, $E(T)/m_p \approx 1$.

Although in general estimating m_p is difficult, Kruskal and Weiss give an estimate for m_p under certain conditions, as stated in the following theorem.

Theorem 4.3 *Given a probability distribution $G(x)$, such that $\int_0^\infty e^{\epsilon x} dG(x) < \infty$ for some $\epsilon > 0$, and given pK independent tasks of length distributed according to G with mean μ and variance σ^2 such that K grows faster than $\log p$,*

then

$$\lim_{\substack{p \to \infty \\ K/\log p \to \infty}} \sup \left| \frac{m_p - K\mu - \sigma\sqrt{2K \log p}}{\log p} \right| < \infty,$$

i.e.

$$m_p = K\mu + \sigma\sqrt{2K \log p} + O(\log p).$$

We now apply these results to get an upper bound on the expected running time of the diagonal algorithm.

Let $m_p(K)$ denote the characteristic maximum of p random variables, each of which is a sum of K tasks, as described in Theorem 4.3.

Consider an $n \times m$ table. Let p be the number of processors used in the diagonal algorithm. The number of tasks that each processor must compute on a given diagonal is no more than $K = \lceil r/p \rceil$, where r is the diagonal number for the diagonals numbered less than n, n for the diagonals numbered between n and m, and n minus the diagonal number for those diagonals numbered between $m+1$ and $n+m-1$. In order to apply Theorem 4.3, K must grow faster than $\log p$.

We apply Theorem 4.3 to the diagonals in the range $(p \log p)^{1+\epsilon} \ldots (m+n-1) - (p \log p)^{1+\epsilon}$, $\epsilon > 0$. Thus, our analysis is valid if the number of processors used in the diagonal algorithm is such that $p \log p$ is $o(n)$.

If we overestimate the cost of the upper left and lower right corners of the table, where the theorem does not apply, we get an asymptotic upper bound on the expected time for the diagonal algorithm of

$$2\lceil (p \log p)^{1+\epsilon} \rceil m_p(\lceil (p \log p)^{1+\epsilon}/p \rceil) + 2 \sum_{r=\lceil (p \log p)^{1+\epsilon} \rceil}^{n-1} m_p(\lceil r/p \rceil) + (m-n+1) m_p(\lceil n/p \rceil).$$

Here we have overestimated the cost of each diagonal in the corners to be the cost of the first diagonal to which the theorem is applied.

The important conclusion we can draw from this sum is stated in the following theorem.

Theorem 4.4 *If $(p \log p)^{1+\epsilon}$ is $O(n^{3/4})$, then $\frac{mn}{p}\mu + O(mn^{1/2} \log n)$ is an upper bound on the expected running time of the diagonal algorithm.*

The proof of this theorem follows from substituting $O(n^{3/4})$ for $(p \log p)^{1+\epsilon}$ in the above sum. We give the general bound on the expected time in the Appendix (see 8.3).

The bound stated in Theorem 4.4 is asymptotically lower than the bound given in Theorem 4.2 for the case where p is $\Theta(\sqrt{n})$ and the tasks are distributed exponentially.

When $(p \log p)^{1+\epsilon}$ is $\Omega(n^{3/4})$, the $\frac{(p \log p)^{2(1+\epsilon)}}{p}\mu$ term becomes significant in the overall sum. It is likely that the bound is too high in these cases, since this term comes from the overestimate of the upper left and lower right hand corners of the table.

5 Experimental Results

Experiments were run on the diagonal and pipeline algorithms using a Sequent Symmetry computer, an asynchronous, shared memory, parallel machine with twenty processors. 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 at synchronization points, and to compare this with the slowdown due to the cost of synchronization primitives.

To do this, we measured the running time of the algorithm and, in addition, we estimated the *synchronization time* and *idle 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.

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.

Our conclusions are summarized as follows.

- The diagonal algorithm is typically slower than the pipeline algorithm, although the difference in running time is small (see Table 2).
- 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 3).
- The idle time is always much more in the diagonal algorithm than in the pipeline algorithm, for 8 or more processors (see Table 4).
- 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.

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

Table 2: Total running times, in seconds

Processors	4		8		12		16	
Size of data	Diagonal	Pipeline	Diagonal	Pipeline	Diagonal	Pipeline	Diagonal	Pipeline
500	0.017	0.026	0.037	0.012	0.056	0.009	0.074	0.006
1000	0.027	0.101	0.053	0.051	0.079	0.034	0.102	0.026
1500	0.036	0.226	0.069	0.113	0.103	0.077	0.135	0.058
2000	0.045	0.401	0.076	0.202	0.076	0.135	0.111	0.103
3000	0.076	0.906	0.145	0.453	0.213	0.303	0.280	0.226
4000	0.099	1.606	0.183	0.803	0.268	0.539	0.354	0.405

Table 3: Synchronization time, in seconds

Processors	4		8		12		16	
Size of data	Diagonal	Pipeline	Diagonal	Pipeline	Diagonal	Pipeline	Diagonal	Pipeline
500	0.02	0.02	0.06	0.01	0.07	0.01	0.09	0.01
1000	0.09	0.10	0.21	0.05	0.27	0.03	0.31	0.02
1500	0.15	0.22	0.30	0.11	0.37	0.08	0.42	0.06
2000	0.23	0.40	0.41	0.20	0.57	0.13	0.63	0.11
3000	0.80	0.87	1.29	0.43	1.21	0.30	1.05	0.23
4000	0.70	1.52	1.01	0.76	1.31	0.54	1.28	0.53

Table 4: Idle time, in seconds

5.1 Timing Details

The application implemented was the Needleman-Wunsch algorithm for aligning two DNA sequences [16]. 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 cell is a very simple task. In order to increase the work between synchronization points, the algorithms do 4×4 blocks of cells 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 against other block sizes.)

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 negating 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 this model is feasible to analyze 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.

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 [7] 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 $0 < a \leq 1$.

We also plan to analyze and implement other parallel algorithms to continue evaluating the model in terms of being simple, realistic and general.

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.

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] 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.

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

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

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

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

[23] 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.

8 Appendix

8.1 Proof of Lemma 3.2

As in Section 3, let $C(k, j)$ be the time for processor k to compute its jth entry and let $S(k, j)$ be the time for customer k to arrive at the queue of the first server for the first time and to be served j times after that. Note that $C(k, j)$ and $S(k, j)$ are random variables. We begin with some definitions and results on stochastic ordering of random variables.

Definition 8.1 *Let X and Y be real random variables. We say $X \leq Y$ if for all real t,*

$$Pr[X > t] \leq Pr[Y > t].$$

Let $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ be real random variables. We say $(X_1, \ldots, X_n) \leq (Y_1, \ldots, Y_n)$ if for all increasing functions $\phi : \mathbf{R}^n \to \mathbf{R}$,

$$\phi(X_1, \ldots, X_n) \leq \phi(Y_1, \ldots, Y_n).$$

Lemma 8.1 *Suppose that $X_1, \ldots, X_n$ are random variables and $Y_1, \ldots, Y_n$ are random variables, such that $X_1 \leq Y_1$ and for $2 \leq i \leq n$ and all $u \leq v$, where u, v are real vectors in $\mathbf{R}^{i-1}$,*

$$Pr[X_i > t \mid (X_1, \ldots, X_{i-1}) = u] \leq Pr[Y_i > t \mid (Y_1, \ldots, Y_{i-1}) = v].$$

Then, $(X_1, \ldots, X_n) \leq (Y_1, \ldots, Y_n)$.

See Arjas and Lehtonen [3] for a simple proof of Lemma 8.1. We extend this result to prove a second useful lemma.

Lemma 8.2 *Let $X_1, \ldots, X_n$, $Y_1, \ldots, Y_n$ be random variables such that $(X_1, \ldots, X_n) \leq (Y_1, \ldots, Y_n)$. Let Z and Z' be independent random variables, $Z \leq Z'$, such that both are also independent of the X_i's and Y_i's. Let f and f' be increasing functions from $\mathbf{R}^n$ to $\mathbf{R}$ such that on any randomly chosen instance $(x_1, \ldots, x_n)$ of $(X_1, \ldots, X_n)$, $f(x_1, \ldots, x_n) \leq f'(x_1, \ldots, x_n)$. Then*

$$(X_1, \ldots, X_n, f(X_1, \ldots, X_n) + Z) \leq (Y_1, \ldots, Y_n, f'(Y_1, \ldots, Y_n) + Z').$$

Proof: We first show that $(X_1, \ldots, X_n, f(X_1, \ldots, X_n)) \leq (X_1, \ldots, X_n, f'(X_1, \ldots, X_n))$. Let $\phi : \mathbf{R}^{n+1} \to \mathbf{R}$ be any increasing function. Then on any random $(x_1, \ldots, x_n)$, since $f(x_1, \ldots, x_n) \leq f'(x_1, \ldots, x_n)$,

$$\phi(x_1, \ldots, x_n, f(x_1, \ldots, x_n)) \leq \phi(x_1, \ldots, x_n, f'(x_1, \ldots, x_n)).$$

Hence

$$Pr[\phi(X_1, \ldots, X_n, f(X_1, \ldots, X_n)) > t] \leq Pr[\phi(X_1, \ldots, X_n, f'(X_1, \ldots, X_n)) > t],$$

which proves that

$$(X_1,\ldots,X_n,f(X_1,\ldots,X_n)) \leq (X_1,\ldots,X_n,f'(X_1,\ldots,X_n)). \tag{1}$$

We next show that $(X_1,\ldots,X_n,f'(X_1,\ldots,X_n)) \leq (Y_1,\ldots,Y_n,f'(Y_1,\ldots,Y_n))$. This follows from the fact that if ϕ is an increasing function, then $\phi(X_1,\ldots,X_n,f'(X_1,\ldots,X_n)) = \phi'(X_1,\ldots,X_n)$, where ϕ' is a different increasing function. Since $(X_1,\ldots,X_n) \leq (Y_1,\ldots,Y_n)$,

$$\phi'(X_1,\ldots,X_n) \leq \phi'(Y_1,\ldots,Y_n).$$

Hence, $\phi(X_1,\ldots,X_n,f'(X_1,\ldots,X_n)) \leq \phi(Y_1,\ldots,Y_n,f'(Y_1,\ldots,Y_n))$. This, together with (1), immediately imply that

$$(X_1,\ldots,X_n,f(X_1,\ldots,X_n)) \leq (Y_1,\ldots,Y_n,f'(Y_1,\ldots,Y_n)).$$

From this and Lemma 8.1, it follows that

$$(X_1,\ldots,X_n,f(X_1,\ldots,X_n),Z) \leq (Y_1,\ldots,Y_n,f'(Y_1,\ldots,Y_n),Z'). \tag{2}$$

Finally, note that if ϕ is an increasing function, then

$$\phi(X_1,\ldots,X_n,f(X_1,\ldots,X_n)+Z) = \phi'(X_1,\ldots,X_n,f(X_1,\ldots,X_n),Z),$$

where ϕ' is also an increasing function. We can thus deduce from (2) that

$$(X_1,\ldots,X_n,f(X_1,\ldots,X_n)+Z) \leq (Y_1,\ldots,Y_n,f'(Y_1,\ldots,Y_n)+Z').$$

□

We can now prove Lemma 3.2.

Proof of Lemma 3.2
Order the tuples (k,j) as follows: $(k',j') \leq (k,j)$ if $j' < j$, or $j' = j$ and $k' \leq k$. We prove by induction on the ordered pairs (k,j) that

$$(C(1,1),\ldots,C(k,j)) \leq (S(1,1),\ldots,S(k,j)).$$

The lemma immediately follows from this. Throughout, we denote by E_μ an exponentially distributed random variable with mean μ. The basis is when $j = k = 1$. In this case, $C(1,1) = S(1,1) = E_\mu$, which clearly implies that $C(1,1) \leq S(1,1)$.

Next, suppose that $j = 1, k > 1$ and $(C(1,1),\ldots,C(k-1,1)) \leq (S(1,1),\ldots,S(k-1,1))$. Then, if $(u_1,\ldots,u_{k-1}) \leq (v_1,\ldots,v_{k-1})$, we claim that

$$Pr[C(k,1) > t \mid C(1,1) = u_1,\ldots,C(k-1,1) = u_{k-1}] \leq$$

$$Pr[S(k,1) > t \mid S(1,1) = v_1,\ldots,S(k-1,1) = v_{k-1}]. \tag{3}$$

From this, Lemma 8.1 immediately implies that $(C(1,1),\ldots,C(k,1)) \leq (S(1,1),\ldots,S(k,1))$. To prove (1), note that $C(k,1) = E_\mu + C(k-1,1)$, since the time until processor k computes its first entry is the time for processor $k-1$ to complete the computation of its first entry, plus the actual time to compute an entry, which is E_μ. Thus, the probability that $C(k-1,1) > t$ given that $C(k-1,1) = u_{k-1}$ is exactly the probability that $E_\mu > t - u_{k-1}$. On the other hand, $S(k,1) \geq E_\mu + S(k-1,1)$. This is because the time until the kth customer is served by server 1 is the time for the $(k-1)$st customer to be served, which is $S(k-1,1)$, plus the time to serve the kth customer once it reaches server 1, which is E_μ, plus the elapsed time from the point at which customer $k-1$ is served and the time that customer k arrives at the queue, (which we do not account for). Hence the probability that $S(k,1) > t$, given that $S(k-1,1) > v_{k-1}$, is at least the probability that $E_\mu > t - v_{k-1}$. This is at least the probability that $E_\mu > t - u_{k-1}$, since $u_{k-1} \leq v_{k-1}$.

This completes the argument when $j = 1$. We next prove that the lemma is true for (k,j) assuming that $(C(1,1),\ldots,C(k',j')) \leq (S(1,1),\ldots,S(k',j'))$, where $(k',j') = (k-1,j)$ if $k > 1$ and $(k'j') = (p,j-1)$ if $k = 1$. To do this, we show that $C(k,j)$ can be expressed as $f(C(1,1),\ldots,C(k',j'))+Z$ and $S(k,j)$ can be expressed as $f'(S(1,1),\ldots,S(k',j'))+Z'$, where f and f' have the following properties. Both f and f' are increasing functions such that on a random instance $(c_{1,1},\ldots,c_{k',j'})$ of $(C(1,1),\ldots,C(k',j'))$, $f(c_{1,1},\ldots,c_{k',j'}) \leq f'(c_{1,1},\ldots,c_{k',j'})$. Also, Z and Z' are independent random variables, and both are independent of $C(1,1),\ldots,C(k',j')$ and $S(1,1),\ldots,S(k',j')$. Then the result follows immediately from Lemma 8.2. There are a number of cases.

Case (i): $k = 1$, $j \leq x$. Then, $C(1,j) = E_\mu + C(1,j-1)$ and $S(1,j) = E_\mu + S(1,j-1)$.

Case (ii): $k = 1$, $x < j \leq m$. Then,

$$C(1,j) = E_\mu + C(1,j-1) \text{ and } S(1,j) = E_\mu + \max(S(1,j-1), S(p,j-x)).$$

Case (iii): $k = 1$, $j > m$. Then,

$$C(1,j) = E_\mu + \max(C(1,j-1), C(p,j-m)) \text{ and } S(1,j) = E_\mu + \max(S(1,j-1), S(p,j-x)).$$

Case (iv): $k > 1$. Then,

$$C(k,j) = E_\mu + \max(C(k,j-1), C(k-1,j)) \text{ and } S(k,j) = E_\mu + \max(S(k,j-1), S(k-1,j)).$$

In all cases, $Z = Z' = E_\mu$. In cases (i) and (iv), $f = f'$. In case (ii),

$$f(c_{1,1},\ldots,c_{k',j'}) = c_{1,j-1} \leq \max(c_{1,j-1}, c_{p,j-x}) = f'(c_{1,1},\ldots,c_{k',j'}).$$

Finally, in case (iii),

$$f(c_{1,1},\ldots,c_{k',j'}) = \max(c_{1,j-1}, c_{p,j-m}) \leq \max(c_{1,j-1}, c_{p,j-x}) = f'(c_{1,1},\ldots,c_{k',j'}).$$

In this case, the middle inequality follows from the fact that on any random instance $(c_{1,1},\ldots,c_{k',j'})$ of $(C(1,1),\ldots,C(k',j'))$, it must be the case that $c_{p,j-m} \leq c_{p,j-x}$. This is because the pth processor must compute the $(j-m)$th entry before computing the $(j-x)$th entry, since $x \leq m$.
□

8.2 Proofs from Section 4

Proof of Theorem 4.1: In our random model, the expected running time of the diagonal algorithm is

$$2\sum_{i=1}^{p-1} T(i,i) + 2\sum_{i=p}^{n-1} T(p,i) + \sum_{i=n}^{m} T(p,n),$$

and $T(p,j) \geq M(p,k) \geq (1/\mu)(H_p + (k-1))$.

So, the total expected time is at least

$$\frac{1}{\mu}[2\sum_{i=1}^{p-1} H_i + 2\sum_{i=p}^{n-1}(H_p + \lfloor\frac{i-1}{p}\rfloor - 1) + \sum_{i=n}^{m}(H_p + \lfloor\frac{n-1}{p}\rfloor - 1)].$$

We partition this sum into two parts A and B, where

$$A = \frac{1}{\mu}[2\sum_{i=1}^{n-1}(\lfloor\frac{i-1}{p}\rfloor + 1) + \sum_{i=n}^{m}\lfloor(\frac{n-1}{p}\rfloor + 1)]$$

and

$$B = \frac{1}{\mu}[2\sum_{i=1}^{p-1}(H_i - 1) + \sum_{i=p}^{m+n-p}(H_p - 2)].$$

Note that A equals $1/\mu$ times the total running time of the diagonal algorithm on the simple non-random model, which is at least $(mn + n(p-1))/(p\mu)$. Also, using $\sum_{i=1}^{p-1} H_i = pH_{p-1} - (p-1)$, B is at least

$$\frac{1}{\mu}(2p(H_{p-1} - 1) + (m+n-2p+1)(H_p - 2)) \geq \frac{1}{\mu}(m+n+1)(H_{p-1} - 2).$$

Combining these, we see that the expected time of the diagonal algorithm is at least

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

□

Proof of Theorem 4.2: 1. Suppose p is a constant. The expected running time of the algorithm is at most

$$2\sum_{i=1}^{n-1} E[M(p,\lfloor\frac{i-1}{p}\rfloor + 1)] + \sum_{i=n}^{m} E[M(p,\lfloor\frac{n-1}{p}\rfloor + 1)]$$

$$\sim \frac{1}{\mu}\left(2\sum_{i=1}^{n-1}(\lfloor\frac{i-1}{p}\rfloor+1)+\sum_{i=n}^{m}(\lfloor\frac{n-1}{p}\rfloor+1)\right),$$

using Lemma 4.2, part 1 and Lemma 4.1. This is equal to $mn/(\mu p)+\Theta(m)$, which can be seen using the same argument as that used to analyze the diagonal algorithm on the non-random model at the start of this section. Similarly, the expected running time of the algorithm is at least

$$2\sum_{i=1}^{n-1}E[M(p,\lfloor\frac{i-1}{p}\rfloor)]+\sum_{i=n}^{m}E[M(p,\lfloor\frac{n-1}{p}\rfloor)],$$

which is also $mn/(\mu p)+\Theta(m)$.

2. Let $p=n/k$, for some constant k. If k is an integer which divides n, then the expected running time of the algorithm is at most

$$2\left(\sum_{i=1}^{p-1}E[M(p,1)]+\sum_{i=p}^{2p-1}E[M(p,2)]+\ldots+\sum_{i=(k-1)p}^{kp-1}E[M(p,k)]\right)+\sum_{n}^{m}E[M(p,k)]$$

$$\sim\frac{1}{\mu}\left(2\left(\frac{n}{k}(\log p+0\log\log p)+\ldots+\frac{n}{k}(\log p+(k-1)\log\log p)\right)+(m-n+1)(\log p+(k-1)\log\log p)\right)$$
$$=\frac{1}{\mu}(2n\log p+n(k-1)\log\log p+(m-n+1)(\log p+(k-1)\log\log p))$$
$$=\frac{1}{\mu}((m+n+1)\log p+(m+1)(k-1)\log\log p).$$

We use Lemma 4.2, part 2, in getting to the second line from the first line. Also, the expected running time of the algorithm is at least

$$2\left(\sum_{i=1}^{p-1}E[M(p,0)]+\sum_{i=p}^{2p-1}E[M(p,1)]+\ldots+\sum_{i=(k-1)p}^{kp-1}E[M(p,k-1)]\right)+\sum_{n}^{m}E[M(p,k-1)].$$

By a similar argument, this is asymptotically equal to

$$\frac{1}{\mu}((m+n+1)\log p+(m+1)(k-2)\log\log p).$$

3. Finally, suppose that $p=\sqrt{n}/k$ for some constant k. Let l be any constant integer. Then, the expected running time of the diagonal algorithm is at most

$$2\sum_{i=\lfloor n/2^l\rfloor}^{2\lfloor n/2^l\rfloor-1}E[M(p,\lfloor\frac{i-1}{p}\rfloor+1)]+2\sum_{i=\lfloor n/2^l\rfloor}^{n-1}E[M(p,\lfloor\frac{i-1}{p}\rfloor+1)]+\sum_{i=n}^{m}E[M(p,\lfloor\frac{n-1}{p}\rfloor+1)].$$

To see this, note that the time to do the first $\lfloor n/2^l\rfloor-1$ diagonals is at most the time to do diagonals $\lfloor n/2^l\rfloor\ldots 2\lfloor n/2^l\rfloor-1$, and similarly, the time to do the last $\lfloor n/2^l\rfloor-1$ diagonals is at most the time to do diagonals $\lfloor n/2^l \rfloor \ldots 2\lfloor n/2^l \rfloor - 1$. Using Lemma 4.2, part 3, as $p \to \infty$, $E[M(p,k)] \leq k(1/\mu + o(1))$, if $k = \lfloor \frac{i-1}{p} \rfloor + 1$, where $i \geq \lfloor n/2^l \rfloor$. Hence,

$$2 \sum_{i=\lfloor n/2^l \rfloor}^{2\lfloor n/2^l \rfloor - 1} E[M(p, \lfloor \frac{i-1}{p} \rfloor + 1)] + 2 \sum_{i=\lfloor n/2^l \rfloor}^{n-1} E[M(p, \lfloor \frac{i-1}{p} \rfloor + 1)] + \sum_{i=n}^{m} E[M(p, \lfloor \frac{n-1}{p} \rfloor + 1)]$$

$$\leq \left(2 \sum_{i=\lfloor n/2^l \rfloor}^{2\lfloor n/2^l \rfloor - 1} (\lfloor \frac{i-1}{p} \rfloor + 1) + 2 \sum_{i=\lfloor n/2^l \rfloor}^{n-1} (\lfloor \frac{i-1}{p} \rfloor + 1) + \sum_{i=n}^{m} (\lfloor \frac{n-1}{p} \rfloor + 1) \right) (1/\mu + o(1))$$

$$= ((n^2/p)(1/2^{2l-1} + 1/2^{2l}) + mn/p)(1/\mu + o(1)) + O(m)$$

$$= (m + \epsilon n)(n/p)(1/\mu + o(1)) + O(m),$$

where $\epsilon = 1/2^{2l-1} + 1/2^{2l}$. □

8.3 General Upper Bound on Diagonal Algorithm

Recall that we can overestimate the cost of the upper left and lower right corners of the table, where Theorem 4.3 does not apply, to get an asymptotic upper bound on the expected time for the diagonal algorithm of

$$2\lceil (p \log p)^{1+\epsilon} \rceil m_p(\lceil (p \log p)^{1+\epsilon}/p \rceil) + 2 \sum_{r=\lceil (p \log p)^{1+\epsilon} \rceil}^{n-1} m_p(\lceil r/p \rceil) + (m - n + 1) m_p(\lceil n/p \rceil).$$

We will break this sum into the following three parts:

$$A = 2\lceil (p \log p)^{1+\epsilon} \rceil m_p(\lceil (p \log p)^{1+\epsilon}/p \rceil),$$

$$B = 2 \sum_{r=\lceil (p \log p)^{1+\epsilon} \rceil}^{n-1} m_p(\lceil r/p \rceil),$$

$$C = (m - n + 1) m_p(\lceil n/p \rceil).$$

The following lemma gives upper bounds on the separate parts of the sum.

Lemma 8.3 *A is asymptotically no more than*

$$2\frac{(p \log p)^{2(1+\epsilon)}}{p}\mu + 2(p \log p)^{\frac{3}{2}(1+\epsilon)} \sigma \sqrt{2 \log p / p} + 2\lceil (p \log p)^{1+\epsilon} \rceil (\sigma \sqrt{2 \log p} + \mu + O(\log p))$$

$$+ 2\frac{(p \log p)^{1+\epsilon}}{p}\mu + 2\sigma \sqrt{2\frac{(p \log p)^{1+\epsilon}}{p} \log p},$$

B is asymptotically no more than

$$\frac{\mu}{p}(n^2 - n - (p \log p)^{2(1+\epsilon)} + \lceil (p \log p)^{1+\epsilon} \rceil) +$$

$$\frac{4}{3}\sigma\sqrt{2\log p/p}(n^{3/2} - (p\log p)^{3/2(1+\epsilon)})$$
$$+(\mu + \sigma\sqrt{2\log p} + O(\log p))(n - \lceil (p\log p)^{1+\epsilon}\rceil),$$

and C is asymptotically no more than

$$(m-n+1)(\frac{n\mu}{p} + \sigma\sqrt{\frac{2n}{p}\log p} + O(\log p) + \mu + \sigma\sqrt{2\log p/p}).$$

Proof:

The bounds on A and C are obtained by applying Theorem 4.3 and overestimating the ceiling function.

Applying Theorem 4.3 and overestimating the ceiling function, the sum B is no more than

$$2\sum_{r=\lceil (p\log p)^{1+\epsilon}\rceil}^{n-1} (\frac{r}{p}\mu + \mu + \sigma\sqrt{2\log p\frac{r}{p}} + \sigma\sqrt{2\log p} + O(\log p)).$$

Next, we observe that

$$\begin{aligned}\sum_{r=\lceil (p\log p)^{1+\epsilon}\rceil}^{n-1} r &= (n^2 - n - \lceil (p\log p)^{1+\epsilon}\rceil^2 + \lceil (p\log p)^{1+\epsilon}\rceil)/2\\ &\leq (n^2 - n - (p\log p)^{2(1+\epsilon)} + \lceil (p\log p)^{1+\epsilon}\rceil)/2\end{aligned}$$

and

$$\sum_{r=\lceil (p\log p)^{1+\epsilon}\rceil}^{n-1} \sqrt{r} \leq \int_{r=\lceil (p\log p)^{1+\epsilon}\rceil}^{n} \sqrt{r} \;\leq\; \frac{2}{3}(n^{3/2} - (p\log p)^{3/2(1+\epsilon)})$$

Substituting these observations back into B gives the upper bound from the lemma statement. □

The expected time of the diagonal algorithm is no more than the sum of A, B and C. This sum looks unwieldy at first glance, but the following theorem puts it in a better perspective.

Theorem 8.1 *Assuming that the lengths of tasks follow a distribution with an increasing failure rate, the expected time of the diagonal algorithm is no more than*

$$\frac{mn}{p}\mu + \frac{(p\log p)^{2(1+\epsilon)}}{p}\mu + \sigma\sqrt{2\log p/p}(mn^{1/2} + \frac{1}{3}n^{3/2} + \frac{2}{3}\frac{(p\log p)3/2(1+\epsilon)}{p}\mu) + O(m\log n).$$

Proof: Writing down all the parts in expanded form, using Lemma 8.3, allows many terms to cancel. Then, observing that $(\mu + \sigma\sqrt{2\log p/p} + O(\log p))$ is $O(\log n)$, we can get the theorem statement. □

Substituting $O(n^{3/4})$ for $(p\log p)^{1+\epsilon}$ gives the bound stated in Theorem 4.4.