Realistic Analysis of Parallel Dynamic Programming Algorithms

Gary Lewandowski
Anne Condon
Eric Bach

Technical Report #1116

October 1992

Realistic Analysis of Parallel Dynamic Programming Algorithms

Gary Lewandowski*
Anne Condon†
Eric Bach‡
Computer Science Department
University of Wisconsin at Madison

October 1992

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 unexpected delays on processors.

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 *realistic* in the sense that the analysis corresponds to experimental results on a shared memory parallel machine.

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 used 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 bus contention, cache misses, and communication delays. Unpredictable delays lead to *starvation* of processors. Starvation occurs anytime a processor is ready to work, but cannot. For example, a processor may be forced to wait at a synchronization point because the unpredictable delays have delayed 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 realistic. 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 realistic, 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 that assumes the time to compute a task is random, following some probability distribution. With this model, we analyze two parallel dynamic programming algorithms using techniques from queueing theory and occupancy problems. We present empirical evidence that the analysis using the model is realistic. While this analysis does not prove the generality of the model, it is likely that the techniques used here can be used in the analysis of other algorithms that use barriers or pipelining techniques.

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 algorithms are good to study because dynamic programming is a very common algorithmic technique (it is used, for example, to solve the Knapsack problem [9], the Longest Common Substring problem [23], queueing network models [1], and DNA sequence alignment [17]), and there are many possible parallel algorithms to implement it. The two dynamic programming algorithms we will examine are the *pipeline* and *diagonal* algorithms. While experiments show that their performance is generally much different on the same input data, synchronous models of parallel computation, such as the PRAM [5] [8] [21], give the same time complexity for both algorithms. A useful asynchronous model must capture the difference.

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.

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 capture the amount of starvation of processors due to unpredictable delays in a parallel system, which we know to be a significant cost of dynamic programming algorithms. We ignore other costs of an algorithm, such as the operating system overhead involved in synchronization. The results of our analysis can be used, together with estimates of these other costs, in an overall evaluation of an algorithm.

One would expect the pipeline algorithm to perform better than the diagonal algorithm because in the pipeline algorithm a processor can proceed to compute an entry as soon as its predecessors are computed, whereas in the diagonal algorithm a processor must wait until all processors complete the jth diagonal before working on the $(j+1)$st diagonal. Our experimental results (on a 20 processor Sequent Symmetry, a shared memory parallel machine) show that this difference in running times is largely attributable to the unexpected delays when the algorithms are running on large datasets. While the pipeline algorithm always outperforms the diagonal algorithm, as datasets get larger the diagonal algorithm actually has less synchronization overhead than the pipeline algorithm. Thus, our analysis is examining the important factor influencing the difference in running times.

The analysis accurately provides a separation in expected running time of the two algorithms, based on the effects of random delays on the two algorithms. Our empirical results on a shared memory machine show that the analysis is realistic, and that unexpected delays are the significant factor in the running time gap of the two algorithms.

1.1 Related Work

Various models have been proposed to analytically predict the increase in running time of an algorithm due to unpredictable delays. 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 [19], Cole and Zajicek [4] and Martel et al. [16] 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 considering actual algorithms rather than parallel programming paradigms and by providing empirical evidence that the analysis is realistic.

There is precedent for the use of the exponential distribution in predicting the performance of parallel programs. As early as 1983, 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. Another more recent example is the work of Mak and Lundstrom [14], who 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 [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.

1.2 Overview of Results

1.2.1 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.

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.

	p a constant	$p = \sqrt{n}$	$p = n$
lower bound, any dp algorithm	$n^2/p + O(1)$	$n\sqrt{n} + O(\sqrt{n})$	$2n - 1$
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: Analytic results for three cases of p ($n \times n$ table, $\mu = 1$)

The first analytic result is a simple lower bound on the expected running time of a general class of *static* algorithms. In these algorithms, 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. In Section 2 we show that a lower bound on the expected running time of a static dynamic programming algorithm with p processors on an $n \times m$ table is $(mn/p + p - 1)/\mu$.

Our general bound on the expected running time of the pipeline algorithm, presented in Section 3, is obtained by modeling the algorithm as a queueing process, and applying results from queueing theory on cyclic queues. For any $p \leq n$, our upper bound is

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

Our results on the diagonal algorithm, presented in Section 4, are obtained using quite different methods. For constant p and $p = \Theta(n)$, we relate the expected running time of the diagonal algorithm to the solution of a well-known occupancy problem and use this relationship to obtain asymptotic estimates on the expected running time. When p is between these two extremes, we extend a result of Kruskal and Weiss [10] on the expected running time of p processors on k subtasks to obtain an upper bound. The following summarizes the resulting asymptotic estimates on the expected running time of the diagonal algorithm.

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. $\leq mn/(\mu p) + O(mn^{1/2}\log n)$, when $(p\log p)^{1+\epsilon} = O(n^{3/4})$, where ϵ is any positive constant.

Finally, for any $p \leq n$, we obtain a lower bound on the expected running time of the diagonal

algorithm, which is

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

(H_p denotes the pth harmonic number, defined as $\sum_{i=1}^{p} 1/i$ [20].)

1.2.2 Summary of Experiments

The two algorithms were timed on the Sequent Symmetry, a shared memory parallel machine that has twenty processors. Details of the experiments are given in Section 5. Experiments were run using 1, 4, 10 and 13 processors.

The pipeline algorithm performed up to 8% faster than the diagonal algorithm. On four processors, unexpected delays in the system accounted for at least 75% of the slowdown experienced by the diagonal algorithm, and at least 30% of the slowdown experienced by the pipeline algorithm. On ten processors, they accounted for at least 65% and 17% respectively, and on thirteen processors, they accounted for at least 60% of the slowdown on the diagonal algorithm and 13% of the slowdown on the pipeline algorithm.

The experiments also show that the difference in running time of the two algorithms is largely attributable to the random delays, when the algorithms are running on large datasets. Factoring out the time spent waiting in each of the algorithms and comparing only the overhead of synchronization (i.e. the actual time spent calling a lock or barrier), we find that as datasets get larger, the overhead of the diagonal algorithm is actually lower than that of the pipeline algorithm.

2 Lower Bound for Static Algorithms

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

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*

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

If $x=m$, there is a natural correspondence between the cyclic p-customer problem and the pipeline algorithm. We next describe this correspondence informally, and then formalize the relationship in Lemma 3.6.

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 we 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. To model 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.

Because the queueing system is cyclic, it is guaranteed that the first customer is never more than m servers ahead of the pth customer. This is important in modeling the algorithm, since the constraints of the algorithm imply that number of table entries computed by the first processor can never be more than m greater than the number computed by the pth processor. If the queueing system has x servers, where $x \leq m$, the first customer is no more than x servers ahead of the pth customer. In this case, we can we can still model the algorithm to get an upper bound on its running time, since the constraints on the customers in the queueing system are more restrictive than those on the processors executing the algorithm.

To make this precise, let $S(k,j)$ 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. 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. We claim in Lemma 3.5 that $C(k,j) \leq S(k,j)$. Since $C(k,j)$ and $S(k,j)$ are random variables, to prove this we need the following definitions and results on stochastic ordering of random variables.

Definition 3.1 *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 $\phi : \boldsymbol{R}^n \rightarrow \boldsymbol{R}$,

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

Lemma 3.2 *Suppose that $X_1, \ldots, X_n$ are independent random variables and $Y_1, \ldots, Y_n$ are independent random variables, such that $X_i \leq Y_i$, $1 \leq i \leq n$. Then*

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

Lemma 3.3 *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 $\boldsymbol{R}^{i-1}$,*

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

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

Lemma 3.2 is a special case of Lemma 3.3; both can be found in Marshall and Olkin [15]. See Arjas and Lehtonen [3] for a simple proof of Lemma 3.3. We extend these to prove a third useful lemma.

Lemma 3.4 *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 $\phi : \mathrm{R}^{n+1} \rightarrow \mathrm{R}$ 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 3.2, 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 proceed to show that $C(k,j) \leq S(k,j)$, where $C(k,j)$ is the time for processor k to compute its jth entry and $S(k,j)$ is the time for customer k to arrive at the queue of the first server for the first time and to be seved j times after that.

Lemma 3.5 *$C(k,j) \leq S(k,j)$, for any $j \geq 1$ and $1 \leq k \leq p$.*

Proof: 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 3.3 immediately implies that $(C(1,1), \ldots, C(k,1)) \leq (S(1,1), \ldots, S(k,1))$. To prove (3), 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 3.4. 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 $x \leq m$. □

Lemma 3.6 *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 3.5, 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.

Note that $S(p, m\lceil n/p\rceil)$ 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. □

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

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

Proof: From Lemmas 3.1 and 3.6, it follows directly that for any positive integer $x \leq m$, 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 $x = \left\lceil\sqrt{m\lceil n/p\rceil(p-1)}\right\rceil$. This gives the bound stated in the theorem. □

4 Analysis of Diagonal Algorithm

In this section, we analyze the diagonal algorithm for dynamic programming on our random model. On an $n \times m$ table, the diagonal algorithm proceeds in iterations. At the dth iteration, the entries on the dth diagonal are computed, where the dth diagonal is the set of entries (i, j) such that $i + j = d + 1$. If there are j entries in the diagonal, then l processors compute $\lfloor(j-1)/p\rfloor + 1$ entries and $p - l$ processors compute $\lfloor(j-1)/p\rfloor$ entries, where $l = j - p\lfloor\frac{i-1}{p}\rfloor$. 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

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

In the first sum, the first p terms are 1, the next p terms are 2, and so on. Hence the first sum can be rewritten as follows.

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

Hence, the total running time is

$$p\lfloor\frac{n-1}{p}\rfloor(\lfloor\frac{n-1}{p}\rfloor+1)+2(n-1-p\lfloor\frac{n-1}{p}\rfloor)(\lfloor\frac{n-1}{p}\rfloor+1)+(m-n+1)(\lfloor\frac{n-1}{p}\rfloor+1)$$

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

Note that 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.

In our random model, the expected running time of the diagonal algorithm is

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

where $T(p,j)$ is 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 [22]).

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, the total expected time of the diagonal algorithm 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

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

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 [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

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

□

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 [18]. We give a brief outline of 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, Lemma 4.2 can be applied in a straightforward way to obtain the following bounds on the expected running time of the diagonal algorithm with p processors.

Theorem 4.1 *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.*

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

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 whose tails are no worse than exponential (i.e. all distributions G such that $dG(x) = O(e^{-\alpha x})$, $\alpha > 0$). 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.

This analysis uses results from Kruskal and Weiss [10]. We begin by presenting these results.

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

Definition 4.1 *The characteristic maximum* $m_p \stackrel{def}{=} inf\{x : 1 - G(x) \leq 1/p\}$.

Kruskal and Weiss [10] proved the following theorem based on results by Lai and Robbins [11].

Theorem 4.2 *Given a probability distribution $G(x)$, such that for some $\epsilon > 0$*

$\int_0^\infty e^{\epsilon x} dG(x) < \infty$,

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.

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

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 4.2.

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 4.2, K must grow faster than $\log p$.

We will apply Theorem 4.2 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.

We will break this sum into the following three parts:

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

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

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

Applying Theorem 4.2, these parts are

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

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

$$C = (m - n + 1)(\lceil \frac{n}{p} \rceil + \sigma\sqrt{2\log p \lceil \frac{n}{p} \rceil} + O(\log p)).$$

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

Lemma 4.3 *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 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 4.3 *Assuming that the lengths of tasks follow a distribution with an exponentially bounded tail, 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: We first write down all the parts in expanded form, using Lemma 4.3:

$$\begin{aligned}
A \;\leq\;& 2\frac{(p\log p)^{2(1+\epsilon)}}{p}\mu + 2(p\log p)^{\frac{3}{2}(1+\epsilon)}\sigma\sqrt{2\log p/p} + 2\lceil (p\log p)^{1+\epsilon}\rceil(\sigma 2\log p + \mu + O(\log p)) \\
&+ 2\frac{(p\log p)^{1+\epsilon}}{p}\mu + 2\sigma\sqrt{2\frac{(p\log p)^{1+\epsilon}}{p}\log p}; \\
B \;\leq\;& \frac{n^2}{p}\mu - \frac{n}{p}\mu - \frac{(p\log p)^{2(1+\epsilon)}}{p}\mu + \frac{\lceil (p\log p)^{1+\epsilon}\rceil}{p}\mu + \frac{4}{3}n^{3/2}\sigma\sqrt{2\log p/p} \\
&- \frac{4}{3}\sigma\sqrt{2\log p/p}(p\log p)^{3/2(1+\epsilon)} + (\mu + \sigma\sqrt{2\log p} + O(\log p))n \\
&- (\mu + \sigma\sqrt{2\log p} + O(\log p))\lceil (p\log p)^{1+\epsilon}\rceil; \\
C \;\leq\;& \frac{mn}{p}\mu - \frac{n^2}{p}\mu + \frac{n}{p}\mu + \sigma\sqrt{2\log p/p}(mn^{1/2} - n^{3/2}) \\
&+ \sigma n^{1/2}\sqrt{2\log p/p} + (m - n + 1)(O(\log p) + \mu + \sigma\sqrt{2\log p/p}).
\end{aligned}$$

Many of the terms from these three parts cancel out. We have the following remains.

From A, little is cancelled out and we are left with

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

Several terms of B cancel with terms in C and A, leaving

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

From C we are left with

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

Observing that $(\mu + \sigma\sqrt{2\log p/p} + O(\log p))$ is $O(\log n)$, everything beyond the first two terms of each part can be lumped together and we see that $A + B + C$ is no more than

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

□

For cases where $(p\log p)^{1+\epsilon}$ is $O(n^{3/4})$, this upper bound is $\frac{mn}{p}\mu + O(mn^{1/2}\log n)$, which is asymptotically lower than our previous analysis 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

5.1 Introduction/Equipment

Experiments were run on the diagonal and pipeline algorithms using a 20 processor Sequent Symmetry computer. The Sequent is a shared memory parallel machine.

The goal of our experiments was to determine empirically the effect of delays on the two algorithms. Below, we first describe the experiments, then discuss the results, and then give a more detailed explanation of how the measurements were taken.

5.2 Methodology

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.

The application implemented as an example of the algorithms is 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 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.)

Each algorithm was run ten times on ten different data sets, ranging from sequences of length 500 to sets of length 1400. Data sets of length less than 500 provide only a few rows for each processor to work on in the pipeline algorithm. Data sets much longer than 1400 characters required more shared memory than were available on our Sequent.

The experiments were run while the machine was not busy, so there should be little effect from any other jobs. The number of processors used in the algorithms was varied between 1, 4, 10, and 13.

5.3 Results

Tables 2 and 3 show the average running times of the two algorithms (with standard deviations) over the ten runs, for the various data set sizes and number of processors. The tables also include the time it took the algorithm to run on one processor without doing any synchronization. This run allows us to get an idea of the synchronization overhead involved in the two algorithms.

These tables indicate that the pipeline algorithm generally outperforms the diagonal algorithm, running between one and eight percent faster.

In our further discussion of these results, we center on three issues. We first look at the relationship of synchronization delays in the algorithms to the overall slowdown experienced in the algorithms. Next, we look at the relationship between the amount of synchronization delay and the number of processors used in each algorithm. Finally, we discuss the effect of waiting time on choosing which algorithm is better.

Tables 4 and 5 give the slowdown of the algorithms. The percentage slowdown of the algorithms decreases as the datasets get larger. This coincides with the theoretical analysis that indicates the delay of the algorithms will be linear, while the time to compute the cells will be $\Theta(n^2)$. Hence, the time to actually compute the cells should dominate the time spent waiting as the datasets get larger.

The slowdown of the algorithms as processors are increased can be broken down roughly into three categories. Two of these categories relate to the synchronization delays of the algorithm. We subdivide these delays into the overhead of executing the locks or barriers, and the waiting time at them. The third category is the increase in task length due to other causes such as cache misses. Tables 6 and 7 indicate the percentage of the slowdown that was due to the synchronization delays, and the percentage that was due only to the waiting time at the synchronization points. In both algorithms, these percentages go down with an increase in processors, indicating the effects of increased cache-to-cache misses on the Sequent.

The diagonal algorithm was greatly affected by synchronization delays, which account for

$75 - 85\%$ of the slowdown on four processors, $65 - 70\%$ on ten processors and $60 - 65\%$ of the slowdown on thirteen processors.

Because a great deal of time was spent waiting at the barriers, the overhead cost of executing a barrier is only a small portion of the total slowdown, accounting for only three or four percent.

The opposite case occurs in the pipeline algorithm, where the slowdown due to waiting at a lock is relatively small (never more than nine percent) while the overhead cost of executing all the locks involved in the synchronization accounts for $28 - 49\%$ of the slowdown on four processors, $14 - 42\%$ on ten processors, and $11 - 34\%$ on thirteen processors.

Overall, the pipeline algorithm was affected less by synchronization delays than by other factors that increase the task times. For four processors, the waiting accounted for $30 - 55\%$ of the slowdown. For ten processors, it decreased to $17 - 40\%$, and for thirteen processors it accounted for only $13 - 38\%$ of the slowdown.

The relation of synchronization delays to the number of processors used is quite different in the two algorithms (see Tables 8 and 9). In the pipeline algorithm, the synchronization delays on the last processor go down as the number of processors increases. In the diagonal algorithm, the synchronization time increases as the number of processors increases. This difference in the effect of increasing processors reflects the difference in synchronization method. In the pipeline algorithm, an increase in processors means fewer rows for each processor to work on and thus fewer locks. As the theoretical analysis indicated, processors do not spend a great deal of time waiting at a lock, so the overhead of actually executing a lock dominates the time spent at a lock. The diagonal algorithm also fits the theoretical analysis, which indicated that an increase in processors would cause and increase in the waiting time at a barrier, since the maximum time spent on a task increases.

The time spent waiting at a synchronization point is extremely important in choosing which algorithm to use. At first glance, one might believe that the pipeline algorithm is always a better choice than the diagonal algorithm. One indication that this is not the case, however, is seen in Tables 2 and 3; the one-processor diagonal algorithm performs better than the pipeline algorithm in the case of the size 500 data set.

Another point in favor of the diagonal algorithm is that it has many fewer barriers than the pipeline algorithm has locks and so spends less overhead on synchronization calls. The diagonal algorithm has $\Theta(n)$ barriers, while the pipeline algorithm has $\Theta(n^2)$ locks. The overhead on a barrier is no more than three times the cost of a lock for thirteen processors, and less for fewer. Thus if no waiting were done beyond the calls to synchronization, the diagonal algorithm would be a better choice than the pipeline.

However, considering only the call to the barrier as the overhead in synchronizing using barriers is misleading. When using p processors in the diagonal algorithm, $(p-1)/p$ of the diagonals will have at least one processor waiting for other processors to do an extra task on

the diagonal. This extra waiting at the barrier is a sort of overhead that one must pay for using the diagonal algorithm.

When this extra cost is factored into the measurement of overhead, the pipeline algorithm is the better choice as the number of processors increase. However, as Table 10 shows, for a smaller number of processors, the diagonal algorithm has less overhead than the pipeline algorithm on large data sets. That is, if there were no waiting at a barrier or lock, the diagonal algorithm would be a better choice in those cases. Thus, the influence of waiting time becomes significant as the size of data sets grow.

5.4 Timing Details

Timings were done using the *getusclk()* function provided by the Sequent timing facility. 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.

For the diagonal algorithm, the time of a task was defined to be the time the algorithm spent between barrier operations. The waiting time at a particular barrier was considered to be the difference between the maximum time spent on the task prior to that barrier and the average time spent by all processors on the task prior to that barrier. The overhead of a given barrier is the minimum amount of time spent at the barrier by any of the processors. Thus, the overhead measurement does not overlap the waiting time measurement.

For the pipeline algorithm, the time of a task was defined to be the time the algorithm spent between a pair of locking operations. The waiting time was defined to be the time at the lock. This includes the overhead of the lock (there is no easy way to avoid including the overhead in the timings). In order to get an idea of the overhead caused by the locks, we calculate an average overhead per lock for each data set by running the pipeline algorithm with one processor on that data set and then dividing by the total number of locks used during that run. The waiting time of the algorithm is the total waiting time of the processor that computes the last row of the table.

The *slowdown* of a parallel algorithm on p processors is the difference between the actual running time on p processors and the time of the algorithm on one processor, divided by p. This definition leaves several possibilities to be considered as the one processor version of the algorithm. One possibility is to take the sequential version of the algorithm; another is to run the parallel algorithm on only one processor; a third is to run the parallel algorithm without synchronization on one processor.

The results presented above use the one processor, no synchronization version of the algorithms. This allows the measurement of slowdown due to increased overhead as the number of processors and size of data sets increases.

6 Conclusions

We have examined a very simple model of parallel computation that models unexpected delays on processors. We have shown that this model is simple 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 likely can be used in analyzing other parallel algorithms, since they apply to situations using pipelining or barriers.

Our experimental work supports our analysis, and indicates that unexpected delays on processors are the most important difference in the running time of the two algorithms.

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. Thanks also 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 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] Marshall, A.W. and I. Olkin, *Inequalities: Theory of majorization and its applications*, New York: Academic Press, 1979.

[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, Rhinehart and Winston, 1985.

[21] Savitch, W.J., M. Stimson. Time bounded random access machines with parallel processing, *Journal 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 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.

Processors Size of data	1 (no synch)	1	4	10	13
500	12478243.20 (2255)	12472891.60 (2559)	3238878.40 (623)	1450118.90 (447)	1184672.10 (490)
600	18072069.00 (4539)	18072317.50 (4091)	4668570.90 (1258)	2062288.70 (785)	1674693.90 (837)
700	24649953.90 (9362)	24989848.10 (517992)	6342885.10 (1589)	2777989.30 (885)	2241275.60 (810)
800	32394014.50 (10183)	32396818.40 (13856)	8285103.10 (1911)	3596705.80 (1427)	2895731.90 (3352)
900	40881820.80 (16078)	40910215.50 (83005)	10464597.90 (2272)	4523639.60 (1735)	3626344.20 (1836)
1000	50540925.70 (16256)	50521212.20 (11860)	12908946.50 (2473)	5545659.70 (1201)	4436447.80 (1919)
1100	61489308.50 (26260)	61482684.20 (22272)	15684339.30 (2429)	6707540.30 (2403)	5344876.70 (2562)
1200	73774369.30 (20396)	73511145.20 (323006)	18671564.00 (4558)	7930990.40 (2223)	6294657.10 (2732)
1300	86019335.50 (17722)	86040906.10 (28449)	21885205.90 (4586)	9261423.30 (2257)	7339761.70 (2117)
1400	99889741.30 (58597)	99972650.50 (248890)	25370583.30 (6420)	10686687.90 (2263)	8451358.70 (2989)

Table 2: Time in μ secs (standard deviation), Diagonal Algorithm

Processors / Size of data	1 (no synch)	1	4	10	13
500	12270613.10 (1321)	12487084.10 (997)	3212986.40 (596)	1346213.70 (2293)	1060563.10 (4128)
600	17750036.00 (3001)	18065073.10 (1930)	4621585.85 (4335)	1874435.10 (4838)	1514662.20 (4735)
700	24170492.50 (5902)	24605586.00 (6573)	6250204.50 (7727)	2598433.40 (6488)	2045927.20 (8505)
800	31583729.20 (5120)	32147597.40 (3173)	8119519.10 (7332)	3314655.40 (6857)	2658502.00 (10693)
900	39984861.50 (6748)	40702515.00 (10619)	10352875.30 (4186)	4251429.20 (8665)	3340050.00 (17484)
1000	49375508.40 (9777)	50257330.80 (11278)	12763127.20 (3465)	5145017.60 (7651)	4108269.80 (18001)
1100	60175123.10 (234220)	61154486.00 (6555)	15493261.60 (19613)	6338617.90 (10137)	4971757.40 (25498)
1200	71576096.70 (16300)	72839887.60 (12274)	18385477.60 (25736)	7420469.70 (8915)	5853417.20 (1052)
1300	84149125.30 (243032)	85553536.60 (6479)	21657106.90 (4219)	8808685.90 (3821)	6742332.60 (10464)
1400	97563507.50 (11778)	99344616.70 (69876)	25115406.60 (15840)	10094690.90 (8918)	7824081.80 (11684)

Table 3: Time in μ secs (standard deviation), Pipeline Algorithm

Processors	4		10		13	
Size of data	μs	%	μs	%	μs	%
500	119317.60	3.8	202294.58	16.2	224807.24	23.4
600	150553.65	3.3	255081.80	14.1	284534.75	20.5
700	180396.63	2.9	312993.91	12.7	345125.30	18.2
800	196599.47	2.3	357304.35	11.0	403884.63	16.2
900	244142.70	2.4	435457.52	10.7	481588.75	15.3
1000	273715.08	2.2	491567.13	9.7	548684.28	14.1
1100	312012.17	2.0	558609.45	9.1	614929.89	13.0
1200	327971.67	1.8	593553.47	8.1	650474.85	11.5
1300	380372.02	1.8	659489.75	7.7	722889.74	10.9
1400	398147.98	1.6	697713.77	7.0	767532.45	10.0

Table 4: Slowdown of Diagonal Algorithm

Processors	4		10		13	
Size of data	μs	%	μs	%	μs	%
500	145333.12	4.7	119152.39	9.7	116669.78	12.4
600	184076.80	4.1	99431.50	5.6	149274.82	10.9
700	207581.37	3.4	181384.15	7.5	186658.55	10.0
800	223586.80	2.8	156282.48	4.9	228984.37	9.4
900	356659.93	3.6	252943.05	6.3	264291.42	8.6
1000	419250.10	3.4	207466.76	4.2	310153.77	8.2
1100	449480.83	3.0	321105.59	5.3	342901.78	7.4
1200	491453.43	2.7	262860.03	3.7	347563.61	6.3
1300	619825.57	2.9	393773.37	4.7	269322.96	4.2
1400	724529.73	3.0	338340.15	3.5	319196.61	4.3

Table 5: Slowdown of Pipeline Algorithm

Processors	4		10		13	
Size of data	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd
500	77.6	73.8	67.8	65.0	64.6	62.1
600	76.2	72.5	66.6	63.9	64.0	61.6
700	75.3	71.7	65.9	63.3	63.1	60.7
800	86.5	82.4	67.3	64.6	63.9	61.6
900	75.6	72.2	65.5	63.1	62.8	60.6
1000	77.7	74.2	65.4	63.1	62.3	60.8
1100	78.1	74.7	65.4	63.1	62.7	60.1
1200	85.6	82.2	67.2	64.8	64.0	61.8
1300	79.1	75.9	66.2	63.9	63.6	61.5
1400	84.3	80.9	67.4	65.0	64.7	62.6

Table 6: Percentage of Slowdown in Diagonal Algorithm from Synchronization

Processors	4		10		13	
Size of data	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd
500	34.3	6.5	17.1	3.3	13.4	2.6
600	38.8	7.0	28.4	3.6	15.2	2.8
700	46.5	8.2	21.9	4.0	16.5	3.9
800	56.0	8.7	32.1	5.6	17.6	3.1
900	45.1	8.7	25.7	4.7	19.3	3.5
1000	47.2	8.8	37.9	5.9	20.3	3.9
1100	53.0	8.2	30.0	4.2	22.1	3.1
1200	57.9	8.5	43.5	7.0	26.8	4.8
1300	54.4	8.1	34.5	5.2	38.2	4.4
1400	53.7	7.6	46.0	5.6	37.4	6.5

Table 7: Percentage of Slowdown in Pipeline Algorithm from Synchronization

Processors	4		10		13	
Size of data	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd
500	92615.35	88091.15	137062.60	131488.40	145295.25	139685.95
600	114774.20	109182.50	169937.41	163003.21	182111.82	175298.62
700	135757.48	129263.48	206315.76	198003.16	217734.38	209548.18
800	161416.10	153818.90	240346.34	230694.34	258222.03	248929.83
900	184532.25	176150.25	285076.28	274681.78	302205.81	291711.11
1000	212677.33	203140.83	321642.85	310000.95	345202.78	333419.08
1100	333528.47	233029.77	365424.11	352375.51	385752.19	372663.19
1200	280597.92	269439.72	398649.02	384396.82	416269.67	402272.87
1300	301010.45	288557.85	436860.61	421655.61	459678.25	444354.95
1400	335504.10	321948.40	470082.55	453752.75	496909.72	480458.52

Table 8: Average total synchronization time, Diagonal Algorithm

Processors	4		10		13	
Size of data	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd	w/ovrhd	w/o ovrhd
500	49897.00	9407.80	20363.50	3914.70	15662.80	3009.90
600	71451.40	12941.10	28258.40	3622.50	22679.50	4202.60
700	96489.80	16976.90	39697.50	7169.50	30847.00	7354.60
800	125174.40	19428.40	50174.90	8705.90	40202.90	7027.70
900	160744.70	31143.70	65072.30	11843.50	51026.60	9369.30
1000	197749.10	36830.10	78680.30	12269.50	63039.50	11954.30
1100	238443.80	36985.80	96397.20	13443.90	75779.80	10602.20
1200	284581.50	41544.50	114310.00	18374.40	93261.10	16512.60
1300	337434.40	50418.40	135873.40	20367.40	102950.30	11945.30
1400	389304.00	55403.00	155602.70	19006.70	119413.00	20760.40

Table 9: Average total synchronization time, Pipeline Algorithm

Processors	4		10		13	
Size of data	Diagonal	Pipeline	Diagonal	Pipeline	Diagonal	Pipeline
500	80502.7	40489.2	98344.7	16448.8	101406.2	12652.9
600	97145.2	58510.3	116209.8	24635.9	121375.1	18476.9
700	113304.4	79512.9	137499.8	32528.0	141399.1	23492.4
800	129284.3	105746.0	155404.4	41469.0	160935.9	33175.2
900	145137.0	129601.0	176110.6	53228.8	180549.7	41657.3
1000	162018.4	160919.0	193810.6	66410.8	200066.6	51085.2
1100	178196.0	201458.0	215112.7	82953.3	219593.3	65177.6
1200	193580.9	243037.0	232742.5	95935.6	238595.2	76748.5
1300	210110.3	287016.0	253693.3	115506.0	258507.0	91005.0
1400	226532.9	333901.0	271397.6	136596.0	278991.8	98652.6

Table 10: Overhead of diagonal algorithm plus waiting due to unequal number of tasks on diagonals versus overhead of pipeline algorithm