Analysis of Delays Caused by Local Synchronization

Julia Lipman
Institute for Defense Analyses
Center for Computing Sciences

Quentin F. Stout
University of Michigan

July 26, 2010

Abstract

Synchronization is often necessary in parallel computing, but it can create delays whenever the receiving processor is idle, waiting for the information to arrive. This is especially true for barrier, or global, synchronization, in which every processor must synchronize with every other processor. Nonetheless, barriers are the only form of synchronization explicitly supplied in OpenMP, and they occur whenever collective communication operations are used in MPI.

Many applications do not actually require global synchronization; local synchronization, in which a processor synchronizes only with those processors from or to which information or resources are needed, is often adequate. However, when tasks take varying amounts of time the behavior of a system under local synchronization is more difficult to analyze since processors do not start tasks at the same time.

We show that when the synchronization dependencies form a directed cycle and the task times are geometrically distributed with $p = 0.5$, then as the number of processors tends to infinity the processors are working $2 - \sqrt{2} \approx 0.59\%$ of the time. Under global synchronization, however, the time to complete each task is unbounded, increasing logarithmically with the number of processors. Similar results apply for $p \neq 0.5$. We also present some of the combinatorial properties of the synchronization problem with geometrically distributed tasks on an undirected cycle.

Nondeterministic synchronization is also examined, where processors decide randomly at the beginning of each task which neighbors(s) to synchronize with. We show that the expected number of task dependencies for random synchronization on an undirected cycle is the same as for deterministic synchronization on a directed cycle.

Simulations are included to extend the analytic results. They show that more heavy-tailed distributions can actually create fewer delays than less heavy-tailed ones if the number of processors is small for some random-neighbor synchronization models. The results also show the rate of convergence to the steady state for various task distributions and synchronization graphs.

1 Introduction

The problem of synchronization is a very general one that appears in many contexts. It applies to any situation in which some kind of distributed agents or processors communicate information about their individual states to each other and a receiving processor cannot proceed until this information is received and/or resources are released. For example, Tabe *et al.* [23] observed that barrier synchronization on the IBM SP2 caused performance degradation due to random operating system interrupts with unexpectedly long tails, even though every processor was completing tasks that should have taken the same amount of time. Nonetheless, barriers are the only form of synchronization explicitly supplied in OpenMP, and they occur whenever collective communication operations are used in MPI.

However, in applications such as time-stepping algorithms for solving partial differential equations, a processor updating values at a site need only wait for the previous values from neighboring sites, a form of *local* synchronization. In a typical MPI send/receive program for such calculations, one region of the simulation may be a few time steps ahead of another without violating computational dependencies. Local synchronization is especially useful when the calculations can take varying amounts of time (perhaps due to the interrupts mentioned above) and delays can occur randomly at any processor or task (such as was observed in the operating system interrupts mentioned above). In this case, delays in one region may be smoothed out before they cause significant slowdown for the entire system.

We model the system as processors $p_0, ..., p_{n-1}$ connected according to some digraph G, all trying to complete a succession of tasks whose durations are independent and identically distributed with probability distribution f. Processor p_i is dependent on processor p_j to complete its r^{th} task if p_i has an incoming edge from p_j in G when p_i has finished its $(r-1)^{\text{st}}$ task. If p_i is dependent on p_j at its r^{th} task, then p_i cannot start its r^{th} task until p_j has finished its $(r-1)^{\text{st}}$ task.

We always consider a processor to be dependent on itself (Chang and Nelson [6] call such models *lateral loops*). At any given instant a processor is either executing its current task, in which case it is *working*, or waiting for one of the predecessor tasks to complete, in which case it is *idle*. Using the terminology of [1], we call the graph G the *processor graph*, and the induced graph of dependencies among processor-task pairs the *task graph*. In most of the models we study, G is the same from task to task, but in Section 4 we consider a model where dependencies can change at random. The system starts out with no processor having completed a task.

By a *level* we mean the time from when all processors have completed their $(i-1)^{\text{st}}$ task to when they have all completed their i^{th} task. Let $\mathcal{T}_{G,f}(i)$ denote the expected time per level when all processors have completed the i^{th} task. We define the (asymptotic) *expected time between levels* of G, $\mathcal{T}^*_{G,f}$, to be $\lim_{i\to\infty} \mathcal{T}_{G,f}(i)$. When G is infinite we modify $\mathcal{T}_{G,f}(i)$ to mean the expected time at any given vertex, rather than the expected time per level in the entire graph. In a finite graph the asymptotic values are the same, but in an infinite graph, for any task distribution times with unbounded support, the expected time per level in the entire graph is infinite for all i.

In Section 3 we consider cyclic synchronization graphs with task times that are geometrically distributed with $p = 0.5$. For a directed cycle of n processors we use a Markov chain approach to determine $\mathcal{T}^*$ exactly. This is given in Theorem 3.3, where it is also proven that as the number of processors approaches infinity, the fraction of time spent working converges to $2 - \sqrt{2} \approx 0.59$. When the task times are geometric with $p \neq 0.5$ the fraction of time spent working converges to a constant that can be bounded using the value for $p = 0.5$. For undirected cycles we examine the combinatorial properties of the corresponding Markov chain, relating them to Motzkin numbers.

In Section 4 we consider a model in which processors can randomly change their synchronization dependencies. Theorem 4.1 shows that for the case of random synchronization on the undirected cycle, where for each task a processor randomly selects one of its neighbors for synchronization, the expected number of task-completion times depended on is the same as for the deterministic directed cycle.

Section 5 contains extensive simulation results for a variety of synchronization models and task time distributions. Geometric, exponential, normal, and Pareto distributions are considered, on cycles, tori, and complete graphs. Deterministic and random synchronizations are evaluated, as is a "first-neighbors" synchronization model, where a processor only needs to wait until a specified number of its predecessors have completed their tasks. Results are given for both the increase in idle time as the number of processors in the system increases, and the performance of the systems as they converge to their stationary behavior.

2 Previous Work

Chang and Nelson [6] prove some bounds on the performance of a parallel system under local synchronization. "The difficulty of the analysis lies in the fact that the times processors start their i^{th} iteration differ, which is not the case for barrier synchronization. This makes an exact analysis intractable ..." they write.

Using a branching process method of Kingman [12], they show that if the task time distribution has all finite moments and the synchronization graph is of bounded degree, then the expected time for a processor to complete a task, including idle time, has an upper bound that does not depend on the number of processors. This upper bound is

$$\inf\{t \geq 1 : rm(t) < 1\}$$

where r is the expected number of dependencies that a processor has at each time step (including dependency on itself at the previous time step), $m(t)$ is defined

$$m(t) = \inf_{\theta \geq 0} e^{-\theta t}\phi(\theta)$$

and $\phi(\theta)$ is the moment generating function of the task time distribution. It is easy to see that this bound does not generally have a closed form; sometimes it is not even finite, though it is when the distribution is geometric, exponential, or normal.

Rajsbaum and Sidi [18] show that, among all "new better than used in expectation" (NBUE) distributions for f, the exponential distribution produces the worst performance. They also perform an exact analysis of the cases where G is a directed cycle or a complete graph, and f is the exponential distribution, using both combinatorial and queueing-theoretic methods. Finally, they study the behavior of a system with one slow "bottleneck" processor.

Taylor and Van Dijk [24] study a model for multicomponent systems that is similar to our model in that components may have dependencies on other components, but that also allows for the possibility of component failure. They bound the performance of these systems by constructing systems with easily computed closed-form stationary distributions and using stochastic comparability to prove that the performance of the new systems bounds the performance of the original systems.

Korniss *et al.* [13] consider the spread, or standard deviation of the tasks completed at each processor, in a synchronization model with exponentially distributed task times. They show that adding random small-world edges can result in a constant bound for the spread without creating unbounded expected synchronization times.

Legedza and Weihl [14] used simulations to study the effect of replacing barrier synchronization with local synchronization. They showed that replacing barriers with local synchronization improved program performance by 24 percent for some applications. Han *et al.* [11] proposed a compiler optimization to replace many instances of barrier synchronization with local synchronization.

Tsitsiklis and Stamoulis [25] find bounds for the communication complexity of a model where processors send messages to their neighbors in a synchronization graph both when triggered by the reception of other messages and at random, under the assumption of stochastic communication delays.

Baccelli and Liu [1] propose a queueing model for parallel systems with precedence constraints on tasks and give integral equations for the steady-state performance of these systems. "Exact analytical solutions [to these equations] seem to be rather difficult to obtain," they write, although they provide some bounds.

Malony *et al.* [15] implement a stochastic technique for predicting the performance scalability of parallel programs which transformed the graph of the original program into a "scaled" model or models. But for a neighbor-synchronized program, they conclude, "If we were to represent each task and dependency ... in the scaled model, the graph size and complexity would be unmanageable." Their solution is to transform the original graph of the neighbor-synchronized program into one with barriers to create an upper bound, and into one with no synchronization at all to create a lower bound, or to implement barriers at some points and no synchronization at others.

The problem of synchronization can also be reduced to a last-passage percolation problem. Directed last-passage percolation is the problem of finding maximal-weight lattice paths with stochastic weights on the lattice points. When the lattice is a directed grid, the problem is exactly that of finding the heaviest path through the dependency graph induced by directed-cycle synchronization. Cohn *et al* [7] proved exact results for domino tilings of the Aztec diamond; as described by Martin [16], these results can, by a series of transformations, be interpreted to provide exact asymptotic results for the last-passage percolation problem corresponding to directed-cycle synchronization with geometrically distributed task times.

One widely used method for predicting the performance of parallel programs with stochastic task duration times is to create a dependency graph of the tasks in the program and then study a similar series-parallel graph, which is much easier to analyze. The algorithm that Escribano and van Gemund [10] propose for converting non-series-parallel graphs into series-parallel graphs results in particularly large upper bounds for neighbor-synchronized graphs; in fact, it simply replaces local synchronization with barriers, as Malony's method does. Escribano and van Gemund [9] researched the extent to which adding dependencies to a task graph in order to make it series-parallel affects its expected completion time. They conjectured that any graph representing a parallel computation has an equivalent series-parallel graph which takes no longer than twice that of the original graph. However, Salamon [19] writes, "These [neighbor-synchronized] task graphs are known to provide counterexamples to [Escribano's conjecture] ... Benchmark structures that occur frequently in practice, such as neighbor synchronization, need to be analyzed further."

3 Geometric/Exponential Distribution

Here we consider the geometric distribution, where each task can be thought of as flipping a coin until it comes up heads, and the exponential distribution. For the geometric distribution on a complete graph of n processors, the expected time between levels is simply the expected time it takes for n coins all to come up heads; that is, the expected maximum of n geometric random variables. For any probability $p \in (0, 1)$ of heads, this extremal statistic grows as $\Theta(\log n)$. For the remainder of this paper, unless otherwise explicitly stated, we assume that the coin is fair, i.e., $p = 0.5$.

It is well-known that if f is an exponential or geometric distribution then there is a constant C_d, where $C_d = \Theta(\log d)$, such that $\mathcal{T}^*_{G,f} \leq C_d$ for any graph G of degree no more than d. See, for example, Exercise 1.4.1 of Bertsekas and Tsitsiklis [5]. The comments above about the complete graph show that this is the strongest bound possible. This result is also implied by the far more general bounds given in Chang and Nelson [6]. Here we are able to obtain the exact value of $\mathcal{T}^*$ for the directed cycle of n processors and a geometric distribution, and its limiting value as n tends to infinity.

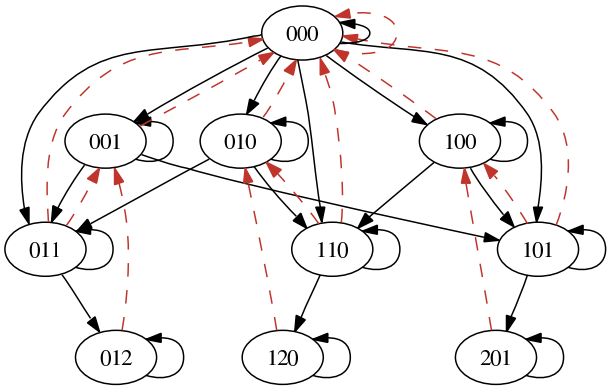

Figure 1: Markov chain for the directed cycle on 3 processors. Dashed lines indicate rezeroing.

3.1 The geometric distribution on the directed cycle

We consider the local synchronization case where G is a directed cycle: each processor must wait only for itself and its left neighbor. Even though the geometric distribution is memoryless, traditional closed-queue methods like the ones used by Rajsbaum and Sidi for the exponential distribution on the directed cycle do not apply to it.

The actions of the processors can be modeled as a discrete-time Markov chain where the states are ordered n-tuples of integers $(s_0, ..., s_{n-1})$ such that s_i corresponds to the number of tasks completed by the processor p_i. So, in a three-processor system, if the first processor has completed the first task and the second two have not completed any, the state would be $(1, 0, 0)$ (which we will abbreviate as 100). For the directed cycle, the states will consist of n-tuples where for all i, $s_i \leq s_{i+1} + 1$ (and $s_{n-1} \leq s_0 + 1$).

When every processor has completed at least one task, the state is *rezeroed*: each number is decremented. The rate of rezeroing is what we want to measure; the expected amount of time between rezeroing operations is the expected time between when every processor in the system has completed at least i tasks and when every processor in the system has completed $i + 1$ tasks. One way to represent this operation is to include states with no zeroes whose incoming edges are states with zeroes: the state 112 would have a single outgoing edge with probability 1 going to 001. However, rezeroing is considered to be instantaneous, and such an edge would imply that a time step had taken place. To avoid this problem, such a state can be collapsed with its succeeding state. The state 012, then, instead of having an outgoing edge going to 112, would instead have one going directly to 001; the rezeroing is implicitly understood. Figures 1 and 2 illustrate the Markov chains.

We call any processor that is not waiting for its neighbor to start the next task a *working* processor, and any processor that could have finished a task at the most recent time step a *last-minute processor*. More formally, a processor i is working if $s_i \leq s_{i-1}$; it is last-minute if $s_i \geq s_{i-1}$.

Calculating the steady-state probabilities of this Markov chain is much easier with the following lemma, which shows that the chain is *balanced*, i.e., that each of its states has the same number of incoming edges as it has outgoing edges.

Lemma 3.1. *The Markov chain for the directed cycle with geometrically distributed tasks is a balanced digraph.*

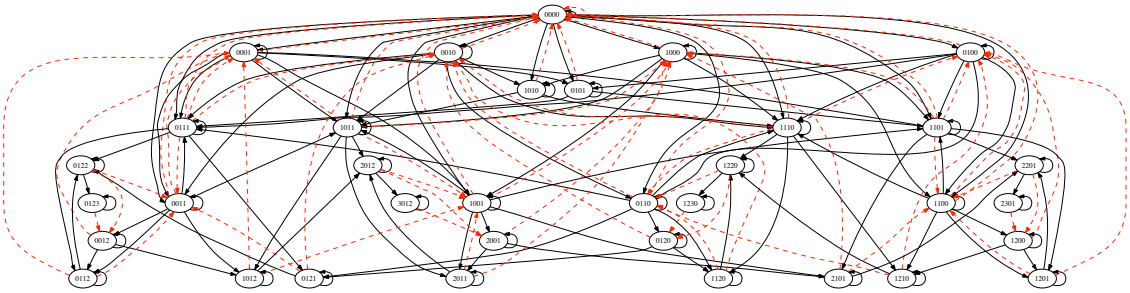

Figure 2: Markov chain for the directed cycle on 4 processors. Dashed lines indicate rezeroing.

Proof. We need to have some way to count the number of outgoing and incoming edges of a state $S = (s_0, .., s_{n-1})$. Note that a processor i is working if and only if $s_i \leq s_{i-1}$. If there are f working processors, then there are 2^f possible next states and 2^f outgoing edges, since any subset of the working processors could finish their tasks by the next time step. These edges also have equal probability, since the coin is assumed to be fair, and the coins for different processors and tasks are assumed independent.

Counting incoming edges is similar; any i for which $s_i \geq s_{i+1}$ is last-minute. To see this, observe that if $s_i = r > 0$ at time t, then s_i could have been $r - 1$ at time $t - 1$. If $s_i = 0$, then, clearly, it could not have been -1 at the previous time step. But consider the state $S' = (s_0 + 1, .., s_{n-1} + 1)$. Such a state would be one of the states with no zeroes that we removed from the chain. We can see here that the i^{th} processor is last-minute. According to our model, all of the edges that would have been S''s incoming edges will go to S. So s_i could have been incremented at the last time step, with rezeroing possibly taking place. If $s_i < s_{i+1}$, then i could not have been incremented at the last time step, because that would imply that s_{i+1} was more than 1 greater than s_i at some time. If there are l last-minute processors, then then there are 2^l possible previous states and 2^l incoming edges with equal probability.

We would like to show that $l = f$ in any possible state. To see this, observe that $s_i \leq s_{i-1}$ if and only if $s_{i-1} \geq s_i$. That is, there is a one-to-one correspondence between working processors and last-minute processors. So each state has the same number of incoming edges as outgoing edges. □

Since the Markov chain of states is a balanced digraph, and since all outgoing arcs from any given state have equal probabilities, the steady-state probability of a state is proportional to its degree, so we can calculate the steady-state probabilities of a state simply by observing which of its processors are working. By looking at the numbers of working processors of all the states in the chain, we can calculate how many processors are expected to be working at a given time, and then how much of the time any processor is expected to be working.

Lemma 3.2. *There are* $\binom{n}{k}\binom{n-1}{k-1}$ *states of length* n *with* k *working processors.*

Proof. Stanley [21] proves that the Narayana numbers $N(n, k) = \frac{1}{n}\binom{n}{k}\binom{n}{k-1}$ enumerate sequences of length $2n$ in which 1 appears n times and -1 appears n times such that each partial sum is nonnegative and there are exactly k instances of a 1 immediately followed by a -1. A variation on his proof yields a proof of this lemma.

Stanley's construction begins with two compositions, $A : a_1 + ... + a_k = n+1$ and $B : b_1 + ... + b_k = n$. There are $\binom{n}{k-1}\binom{n-1}{k-1}$ ordered pairs (A, B). From each ordered pair, he creates a circular sequence with a block of a_1 1's, then a block of b_1 -1's, then a block of a_2 1's, and so on. He shows that each such circular sequence could have been created from exactly k composition pairs, and that

there is only one way to get a linear sequence from it that starts with a 1 and has all nonnegative partial sums when this 1 is removed, so there are $\frac{1}{k}\binom{n}{k-1}\binom{n-1}{k-1} = \frac{1}{n}\binom{n}{k}\binom{n}{k-1}$ such sequences.

Our proof is similar. A state of length n with k working processors is a sequence $s_0, \ldots, s_{n-1}$ of nonnegative integers containing at least one 0 that satisfies the following three conditions:

1. It contains at least one 0,
2. For all i, $s_{i+1 \pmod n} \leq s_i + 1$
3. There are exactly k positions i at which $s_{i+1 \pmod n} < s_i + 1$.

Such a state can be obtained from a circular sequence S of the form in Stanley's construction as follows: break S into a linear sequence that starts with at least two 1's, and remove the first of these; call this new linear sequence S'. We create a new sequence T from S': At the first element (which must be a 1), write a 0. At each subsequent 1, write the partial sum at that point (take -1's into account in the calculation of the partial sums, but write partial sums only at 1's, not at -1's). Some of the terms in T may be negative. Now increase each term in T by the same amount such that the least term is 0. The sequence we construct from T in this way is a state of length n with k working processors.

There are $n-k+1$ points at which to break S into a linear sequence such that it starts with at least two 1's, since there are $n+1$ 1's and k of them are immediately followed by -1's. Breaking it at each of these points results in a different beginning state for the same reason that S could have been created from exactly k composition pairs: n and $n+1$ are relatively prime. We can also see that every length-n, k-working-processor beginning state can be made into a circular sequence of the form in Stanley's construction: go through the sequence, at each s_i writing $s_i - s_{i+1 \pmod n} + 1$ -1's followed by a 1. Finish by writing an extra 1. This will result in a circular sequence containing $n+1$ 1's (one for each entry in the state and one extra) and n -1's, since it ends up in the same place it started. Since there are k working processors, there are k positions at which $s_i - s_{i+1 \pmod n} + 1 > 0$, so there will be k groups of 1's. So the circular sequence created is one of the form in Stanley's construction.

We have shown that there are $n-k+1$ times as many beginning states of length n with k working processors as there are such circular sequences:

$$(n-k+1)\frac{1}{k}\binom{n}{k-1}\binom{n-1}{k-1} = \binom{n}{k}\binom{n-1}{k-1}$$

. □

Theorem 3.3. *For a directed cycle of n processors, with geometrically distributed tasks with $p = 0.5$, the fraction of time each processor is working is*

$$\frac{\sum_{k=1}^{n} k 2^k \binom{n}{k}\binom{n-1}{k-1}}{n \cdot \sum_{k=1}^{n} 2^k \binom{n}{k}\binom{n-1}{k-1}}$$

As n tends to infinity this converges to $2 - \sqrt{2} \approx 0.59$.

Proof. A state with k working processors has outdegree 2^k in the Markov chain of the directed cycle with geometric tasks. As we have shown, the Markov chain is a balanced digraph for $p = 0.5$, which means that the steady-state probability of a state is proportional to its (out)degree.

We can find the expected number of working processors at any given time by summing, over all states, the number of working processors multiplied by the outdegree, which using what we have

shown in Lemma 3.2 is $\sum_{k=1}^{n} k2^k\binom{n}{k}\binom{n-1}{k-1}$, and normalizing by the total outdegree of all states ($\sum_{k=1}^{n} 2^k\binom{n}{k}\binom{n-1}{k-1}$). Thus the expected number of working processors is

$$\frac{\sum_{k=1}^{n} k2^k\binom{n}{k}\binom{n-1}{k-1}}{\sum_{k=1}^{n} 2^k\binom{n}{k}\binom{n-1}{k-1}} \tag{1}$$

and the fraction of time a processor is working is this divided by n.

To determine the asymptotic value as n goes to infinity, we estimate each sum. We first find the k at which the greatest term occurs in each sum. In each case, the terms in the sum are unimodal, so we look for the k at which the ratio between the $(k+1)^{\text{th}}$ and k^{th} term is closest to 1. In $\sum_{k=1}^{n} k2^k\binom{n}{k}\binom{n-1}{k-1}$, we have the equation $k2^k\binom{n}{k}\binom{n-1}{k-1} = (k+1)2^{k+1}\binom{n}{k+1}\binom{n-1}{k}$. Solving the resulting quadratic for k results in $(2-\sqrt{2})n \approx 0.59n$. Doing the same for the sum in the denominator, $\sum_{k=1}^{n} 2^k\binom{n}{k}\binom{n-1}{k-1}$, results in a value of k that approaches $(2-\sqrt{2})n$ as n goes to infinity. So the largest term of both sums occurs at $k = (2-\sqrt{2})n$.

Let the k^{th} term of the series in the numerator be a_k and that of the series in the denominator b_k, and let $C = 2-\sqrt{2}$. For some $\varepsilon > 0$, consider the ratio of two consecutive terms a_k and a_{k+1} where $k = Cn + \varepsilon n$:

$$\frac{a_{k+1}}{a_k} = \frac{2(1-C-\varepsilon)^2}{(C+\varepsilon)^2} \tag{2}$$

which is less than 1 and decreasing in ε for all $0 < \varepsilon < \sqrt{2}$.

For the series in the denominator, we get

$$\begin{aligned} \frac{b_{k+1}}{b_k} &= \frac{2n(1-C-\varepsilon)^2}{C+\varepsilon+n(C+\varepsilon)^2} \\ &\approx \frac{2(1-C-\varepsilon)^2}{(C+\varepsilon)^2} \\ &= \frac{a_{k+1}}{a_k} \end{aligned} \tag{3}$$

Similarly, we can show that the ratio of two consecutive terms a_k and a_{k-1} where $k = CN - \varepsilon n$ is $\frac{(C-\varepsilon)^2}{2(1-C-\varepsilon)^2)}$, and for the series in the denominator, we get $\frac{a_{n-1}}{a_n}$.

From 2, 3, and the similar results for decreasing terms, we see that the terms of both series below $k = (C-\varepsilon)n$ and above $k = (C+\varepsilon)n$ are falling off geometrically with constant ratios that are independent of n. Call these ratios $\kappa_{\varepsilon,1}$, $\kappa_{\varepsilon,2}$, $\kappa_{\varepsilon,3}$ and $\kappa_{\varepsilon,4}$ respectively. Since $a_k = kb_k$, we can rewrite the ratio in 1 as

$$\frac{\kappa_{\varepsilon,1}(Cn-\varepsilon n)b_{Cn-\varepsilon n} + \left(\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} kb_k\right) + \kappa_{\varepsilon,2}(Cn+\varepsilon n)b_{Cn+\varepsilon n}}{\kappa_{\varepsilon,3}b_{Cn-\varepsilon n} + \left(\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k\right) + \kappa_{\varepsilon,4}b_{Cn+\varepsilon n}}$$

The middle summation in the numerator is bounded above and below by $(Cn+\varepsilon n)\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k$ and $(Cn-\varepsilon n)\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k$ respectively. We can see that

$$\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k \geq 2\varepsilon n \max(b_{Cn-\varepsilon n}, b_{Cn+\varepsilon n})$$

since there are $2\varepsilon n$ terms and the lowest occur at the beginning and end. So we can rewrite the ratio as

$$\frac{o\left(\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k\right) + An\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k}{o\left(\sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k\right) + \sum_{k=Cn-\varepsilon n}^{Cn+\varepsilon n} b_k}$$

where $C - \varepsilon \leq A \leq C + \varepsilon$, which approaches An for large n.

Since a fraction of processors that approaches $2 - \sqrt{2}$ of the total are expected to be working at any given time, by symmetry, any given processor will be working for a proportion of the time that approaches $2 - \sqrt{2}$. □

Corollary 3.4. *When G is a directed cycle of n processors and f is the geometric distribution with $p = 0.5$,*

$$\mathcal{T}^*_{G,f} = \frac{2n \cdot \sum_{k=1}^{n} 2^k \binom{n}{k}\binom{n-1}{k-1}}{\sum_{k=1}^{n} k2^k \binom{n}{k}\binom{n-1}{k-1}}$$

which approaches $2 + \sqrt{2} \approx 3.42$ as n goes to infinity.

Proof. Whenever a processor is working, it is actively engaged in working on a task; that is, waiting for a coin to come up heads. The expected amount of time for a fair coin to come up heads is 2, so $\mathcal{T}^*_{G,f}$ is 2 divided by the fraction of time the processors are working. □

So regardless of how many processors are connected in the directed cycle, the time between levels will always approach a value that is bounded above by $2 + \sqrt{2}$. The improvement over the complete graph holds even for very small graphs. It can be shown by direct calculation that for geometrically distributed tasks with $p = 0.5$, the time between levels in a complete graph of 4 processors is $368/105 \approx 3.5$, which already exceeds our upper bound for a directed cycle of any number of processors. These results also hold for the infinite directed line.

The result for $p = 0.5$ can be used to give bounds for other values of p. For any graph G, if $\mathcal{T}^*_{G,f}$ is the asymptotic time per level using probability distribution f, then $\alpha\mathcal{T}^*_{G,f} + \beta$ is the asymptotic time using distribution $f'(\alpha x + \beta) = f(x)$, and if the cdf of f'' is less than the cdf of f then $T(f'') > T(f)$. Using this, one can show that if $p = 0.25$, then the asymptotic time per level is between $2(2+\sqrt{2}) - 1$ and $3(2+\sqrt{2}) + 2$. Similar bounds can be found for all values of p.

Baskett and Smith [3] consider a related problem, in which, at every time step, each of N processors makes a request of a memory module randomly selected from a group of M modules. When $N = M$ and N goes to infinity, the proportion of working memory modules approaches $2-\sqrt{2}$. However, for finite cases, this problem has slightly different results than the one we consider in this paper. For example, in the case of 3 processors in a directed cycle, the steady-state proportion of working processors in our model is $\frac{13}{19} \approx 0.684$. In the Baskett and Smith model with 3 processors and 3 memory modules, the steady-state proportion of working memory modules is $\frac{43}{63} \approx 0.683$.

It should be noted that for the directed cycle with exponentially distributed tasks having expectation 2, $\mathcal{T}^*_{G,f}$ approaches 4 [18], even though the exponential distribution is the continuous analogue of the geometric distribution. However, while this exponential distribution has the same mean as does the geometric distribution with $p = 0.5$, its second and all higher order moments are larger. Figure 3 shows the Markov chain induced by synchronization on a directed cycle with exponentially distributed tasks; note that states with a high concentration of working processors, like 000, have a much smaller degree here than they do in the chain for geometrically distributed tasks, shown in Figures 1 and 2.

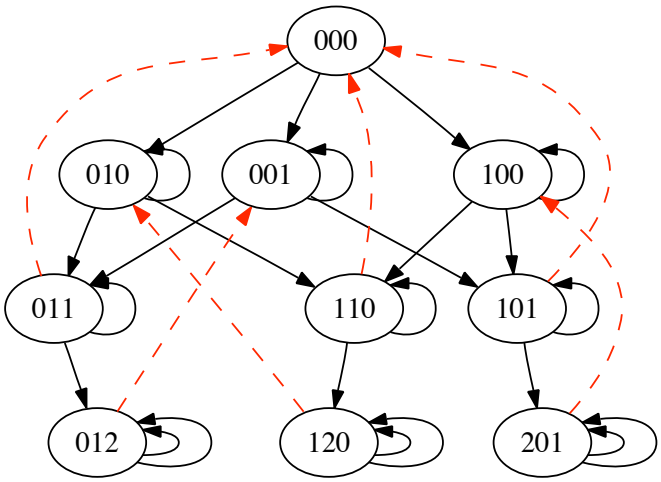

Figure 3: The discretized Markov chain induced by directed-cycle synchronization on 3 processors with exponentially distributed tasks; dashed arcs correspond to rezeroing.

3.2 The undirected cycle: combinatorial properties

The Markov chain for the undirected cycle is not a balanced digraph, so, unlike the one for the directed cycle, it does not have an obvious closed-form stationary distribution for arbitrary n. However, we can prove some facts about its combinatorial properties. Here we let S^n be the set of states in this chain.

Theorem 3.5. *The number of states in S^n is C_{n+1}, the $n+1^{\text{st}}$ central trinomial coefficient.*

Proof. The central trinomial coefficients C_n enumerate grand Motzkin paths of length $n-1$, that is, paths beginning at $(0,0)$ and ending at $(n-1,0)$ that consist of horizontal, downward diagonal and upward diagonal steps (denoted H, D and U), as shown in Figure 4. We can show that there is a bijection between states of length n and grand Motzkin paths with n steps. Given a state $s = (s_1, s_2, ..., s_n)$, we can map it to a grand Motzkin path $G(s) = m = ((0,0), (1,m_1), ..., (n,m_n))$ by setting $m_i = s_i - s_n$. For instance, the state 2101 would map to $((0,0),(1,1),(2,0),(3,-1),(4,0))$. Given a grand Motzkin path $m = ((0,0),(1,m_1),...,(n,m_n))$, we can map it to state $f^{-1}(m) = (m_1 - \min(m_1,...,m_n), ..., m_n - \min(m_1,...,m_n))$. Thus there are exactly as many states of length n as there are grand Motzkin paths with n steps. □

Theorem 3.6. *The number of working processors among all states in S^n is $n(C_n + C_{n-1})$.*

In order to prove this theorem, we partition working processors s_i into three types and enumerate each one.

- *Type I*: $s_i = 0$ and $s_j \neq 0$ for all $j \neq i$. Example: 11012.
- *Type II*: s_i is such that it cannot be replaced with $s_i - 1$ to create a legal state, either because doing so would result in a negative number or because it would result in a processor that is more than one task ahead of a neighbor. Also, it is not of Type I. Example: 01211.
- *Type III*: $s_i \neq 0$ and $s_{i-1} = s_i = s_{i+1}$. Example: 00111

Lemma 3.7. *Every working processor can be classified into exactly one of these three types.*

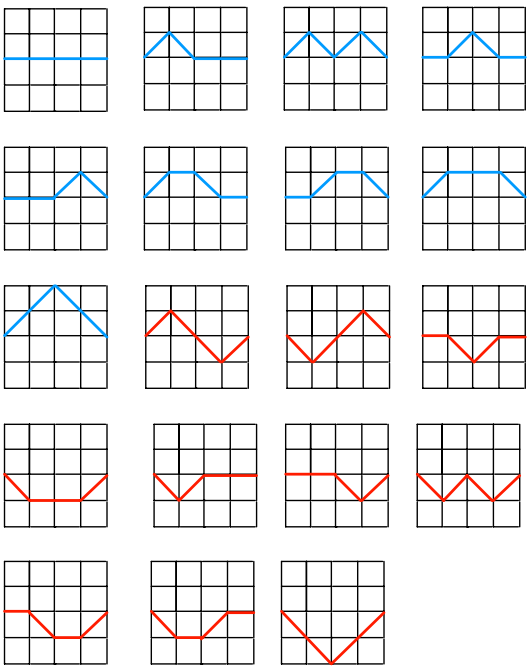

Figure 4: The 19 grand Motzkin paths of length 4. The first 9 are also Motzkin paths, since they do not go below the center line.

Proof. Consider a working processor s_f in a state s. If $s_f = 0$ and is the only 0 in s, then it is of Type I. If $s_f = 0$ and it is not the only 0 in s, then, since it is not of Type I and cannot be replaced with any lower number, it must be of Type II.

Now consider the case where $s_f > 0$. As a working processor, it must have neighbors that are greater than or equal to it. If at least one of its neighbors is greater than it, then it must be of Type II, since it cannot be replaced with anything smaller without creating an illegal state. If, however, both of its neighbors are equal to it, then it is by definition of Type III. □

Lemma 3.8. *There are nM_{n-1} working processors of Type I among all states of S^n, where M_n is the n^{th} Motzkin number (Sloane's A001006).*

Proof. The Motzkin numbers M_n enumerate Motzkin paths of length $n-1$, that is, paths beginning at $(0,0)$ and ending at $(n-1,0)$ that consist of upward diagonal, downward diagonal and horizontal steps, and that do not go below the line $x = 0$, as shown in Figure 4. We can show that there is a bijection between length-n states that begin with a Type I processor and Motzkin paths of length $n-2$. Consider such a state $s = (s_1, s_2, ..., s_n)$. We know that $s_1 = 0$ and that $s_i > 0$ for all $i > 1$. We can map it to the path $m(s) = ((0, s_2 - 1), (1, s_3 - 1), ..., (n-1, s_n - 1))$. Since s_2 and s_n must both be equal to 1, and no other s_i can be less than 1, this path starts and ends at height 0 and never goes below 0, making it a legal Motzkin path. Similarly, we can map a Motzkin path of length $n-2$ to a length-n state that begins with a Type I processor.

We have shown that there are exactly as many length-n states that begin with a Type I processor as there are Motzkin paths of length $n-2$. Since every such state has n distinct cyclic permutations, and every state with a Type I processor is a cyclic permutation of a state with a Type I processor at the beginning, there are n times as many Type I processors as there are states that begin with a Type I processor; that is, nM_{n-1}. □

Lemma 3.9. *There are nC_n length-n working processors of Type II among all states of S^n.*

Proof. We showed in Theorem 3.5 that there are T_{n+1} total states in S^n. We can show that there are exactly n times as many working processors of Type II among all states of S^n as there are states in S^{n-1}.

For each state in S^{n-1}, there are n places where we can insert a processor to get a state in S^n. There is exactly one choice for the new processor such that it is a working processor of Type II: the least possible number that could go there to create a legal state. For example, from the state 0121, we can obtain 00121, 00121, 01121, 01211 and 01210.

For each working processor of Type II among the states of S^n, there is exactly one state in S^{n-1} into which it could have been added in this way; removing a working processor cannot create an illegal state, since its two neighbors cannot differ from each other by more than 1.

So there are exactly n times as many working processors of Type II among all states of S^n as there are states in S^{n-1}; that is, nC_n. □

Lemma 3.10. *There are $n(C_{n-1} - M_{n-1})$ working processors of Type III among all states of S^n.*

Proof. We can obtain a grand Motzkin path $G(s, s_f)$ from a state $s = (s_1, ..., s_n)$ with a working processor s_f of Type III as follows: starting at s_f, we write H, D or U at each s_i depending on whether s_i is equal to, less than or greater than s_{i-1}, respectively.

A Type III working processor must be equal to both of its neighbors, so this path must begin with HH. We also know that s_f must be greater than 0, but that the state s must contain a 0 somewhere for it to be a legal state. This means that at some point, $G(s, s_f)$ must go below the line $x = 0$, since the path started at $(0, 0)$. So $G(s, s_f)$ is a grand Motzkin path, but not a Motzkin path, since Motzkin paths must stay above that line. Ignoring the beginning HH common to all grand Motzkin paths obtained in this way, we can think of $G(s, s_f)$ as a grand Motzkin path from $(0, 0)$ to $(n - 2, 0)$.

We can also show that each grand Motzkin path G that is not also a Motzkin path can be obtained in this way from exactly n ordered pairs (s, s_f) of a state in S^n and a Type III working processor in that state. Taking the $(n - 2)$-tuple of the path's y-coordinates, incrementing each member by the same amount such that the least is 0 and appending the first member to both the beginning and the end to account for the ignored HH, we get the state s that created this path, rotated so that the Type III working processor s_f is at the beginning. So there are n ordered pairs that could have been mapped to this path, since there are n cyclic permutations of a length-n state.

There are $C_{n-1} - M_{n-1}$ grand Motzkin paths of length $n - 2$ that are not also Motzkin paths. Since we have shown that there are n times as many working processors of Type III among all the states of S^n, there are $n(C_{n-1} - M_{n-1})$ such working processors. □

Now we are ready to prove Theorem 3.6. Since all working processors can be classified into one of the three types, there are $nM_{n-1} + nC_n + n(C_{n-1} - M_{n-1}) = n(C_n + C_{n-1})$ total working processors among all states of S^n. Additionally, $C_n + C_{n-1} = A_n$ is the number of UDU-free paths of $n - 1$ upsteps and $n - 1$ downsteps (Sloane's A025565) [20].

Now we can calculate the average number of working processors among all states of S^n.

Theorem 3.11. *The average number of working processors among all states of S^n is $\frac{4}{9}n$.*

Proof. The chain has C_{n+1} total states and $n(C_n + C_{n-1})$ working processors among those states. So the average number of working processors among all states is

$$\frac{n(C_n + C_{n-1})}{C_{n+1}} \tag{4}$$

From [20], we know that C_n asymptotically approaches $d \cdot 3^n/\sqrt{n}$ for a constant $d \approx 0.5$. So the expression in 4 approaches

$$\frac{n\left(\frac{d\cdot 3^n}{\sqrt{n}} + \frac{d\cdot 3^{n-1}}{\sqrt{n-1}}\right)}{\frac{d\cdot 3^{n+1}}{\sqrt{n+1}}} \to \frac{4}{9}n \tag{5}$$

Thus, as n approaches infinity, the proportion of working processors approaches $\frac{4}{9}$. □

This implies that if each state in this chain had equal stationary probability, as does each state in the discretized chain for the directed cycle with exponentially distributed tasks, $\mathcal{T}^*_{G,f}$ would approach 4.5. In practice, our simulations show that the actual value for $\mathcal{T}^*_{G,f}$ in the undirected cycle with exponentially distributed task times is ≈ 4.77.

We can in fact easily modify the chain for the undirected cycle to construct one with the same states where every state has equal stationary probability; there is a bijection between states with indegree a and outdegree b and states with indegree b and outdegree a, so edges can be redirected from states with greater indegree to states with lesser indegree. This method always redirects edges from states with fewer working processors to states with more working processors, so it would appear to be a lower bound on the original chain. However, neither the original chain nor the modified one is stochastically monotone, so the standard techniques for proving that the modified chain is a bound, like those described in [22], do not apply. Van Houtum *et al.* [26] describe a similar method for constructing bounds on queueing systems considered as Markov reward processes by redirecting edges, but their methods of proof are not readily applicable here.

4 Random-Neighbor Synchronization

A further extension of our model allows processors to select a random subset of its neighbors in the synchronization graph with which to synchronize. In the random-neighbor model, a processor p_i selects uniformly at random a subset of c_i neighbors at the beginning of its m^{th} task, where selections at all processors and all levels are independent.

When p_i and all of its randomly selected neighbors have finished their mth tasks, p_i begins its $(m+1)^{\text{st}}$ task (selecting another random subset of c_i neighbors). Figure 5 illustrates the dependencies that can result from this process.

This allows us to study synchronization where each processor waits for a constant number of neighbors despite the fact that the synchronization graph may have degree unbounded in n (e.g., random-neighbor synchronization with c neighbors on a complete graph for some constant c). We can also look at graphs with bounded degree where processors synchronize with the same number of neighbors as they do in graphs of lower degree.

In the following result, we show that the expected number of dependencies in the undirected infinite line with random one-neighbor synchronization is the same as the number of dependencies in the directed infinite line.

Theorem 4.1. *For a given processor p in the infinite line with random synchronization, the expected number of task-completion times that p depends on to start the m^{th} task is $\frac{m(m+1)}{2} - 1$, which is the same as the number for a processor in the (deterministic) infinite line.*

Proof. We prove this by induction. First, we show that for a given $r < m$, the expected number of task-completion times $T_j(m-r)$ on which p_k depends to start the m^{th} task (which we will also refer to as times on which $T_k(m)$ depends) is $m - r + 1$. For a given t, let $N(T_k(m), t)$ be the number of task completion times $T_j(t)$ on which $T_k(m)$ depends.

We know that $E[N(T_k(m), m-1]$ is 2, since it can depend only on its own time at the previous task, $T_k(m-1)$, and exactly one of $T_{k-1}(m-1)$ or $T_{k+1}(m-1)$, whichever corresponds to the randomly selected dependency. Now assume that $N(T_k(m), m-r) = \ell$. We will show that $E[N(T_k(m), m-r-1] = \ell + 1$.

Observe that if $T_k(m)$ depends both on $T_{j-1}(m-r)$ and $T_{j+1}(m-r)$, then it also must depend on $T_j(m-r)$. This follows from the fact that if p_k depends on $T_x(t)$ to start a task, then it also must depend on $T_x(t')$ for all $t' < t$, since self-dependencies always apply. By the definition of dependency, any task-completion time on which p_k depends to start the m^{th} task must have a path in the dependency graph to $T_k(m)$. So if $x < k$ and p_k depends on the task-completion time $T_x(t)$ for some t to start the m^{th} task, then $T_x(t)$ must appear on a path to $T_k(m)$ in the dependency graph, and that path must go through $T_{x+1}(\tau)$ for some $\tau > t$; similarly, if $x > k$, then the path must go through $T_{x-1}(\tau)$ for some $\tau > t$. But then we also know that $T_k(m)$ depends on $T_{x+1}(\tau')$ (or for $x > k$, $T_{x-1}(\tau')$) for all $\tau' < \tau$.

So the ℓ task-completion times of the form $T_j(m-r)$ on which $T_k(m)$ depends must all correspond to consecutive processors. Let j_0 be the least j for which this is true; then $j_0 + \ell - 1$ is the greatest. We already know then that $T_k(m)$ must depend on $T_{j_0}(m-r-1), T_{j_0+1}(m-r-1), ..., T_{j_0+\ell-1}(m-r-1)$, due to the self-dependency constraint. $T_{j_0}(m-r)$ has probability $\frac{1}{2}$ of depending on $T_{j_0-1}(m-r-1)$, which does not appear on that list, and probability $\frac{1}{2}$ of depending on $T_{j_0+1}(m-r-1)$, which does. Similarly, $T_{j_0+\ell-1}(m-r)$ also has probability $\frac{1}{2}$ of adding another dependency to the list, and all other $m-r$ task-completion times have probability 0 of doing so. So $E[N(T_k(m), m-r-1] = \ell + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = \ell + 1$.

By this result and linearity of expectation,

$$\begin{aligned} E\left[\Sigma_{r=1}^{m-1} N(T_k(m), m-r)\right] &= \Sigma_{r=1}^{m-1} E[N(T_k(m), m-r)] \\ &= E[N(T_k(m), m-1] + (E[N(T_k(m), m-1] + 1) + ... + \\ &\quad E[N(T_k(m), m-1] + (m-2) \\ &= 2 + (2+1) + ... + m \\ &= \frac{m(m+1)}{2} - 1 \end{aligned}$$

The fact that this is the number that occurs in the undirected case is quite straightforward. □

5 Simulation results

To extend the analytic results to additional synchronization graphs and models, we carried out a variety of simulations. All of the simulations were done in C++ using random number generators from the GNU Scientific Library.

One of the challenging aspects of some computing tasks is that they can have heavy-tailed distributions, typically power laws (Pareto distributions) with a pdf of the form $x^{-\lambda}$. For example, file sizes and network traffic have such distributions [2, 8, 17]. For power laws, the probability of sampling a large value is greatly higher than it is for the other distributions considered, and thus one would expect that reducing synchronization is even more important. We also studied normally distributed tasks to show the effects of a distribution that is less heavy-tailed than geometric or exponential.

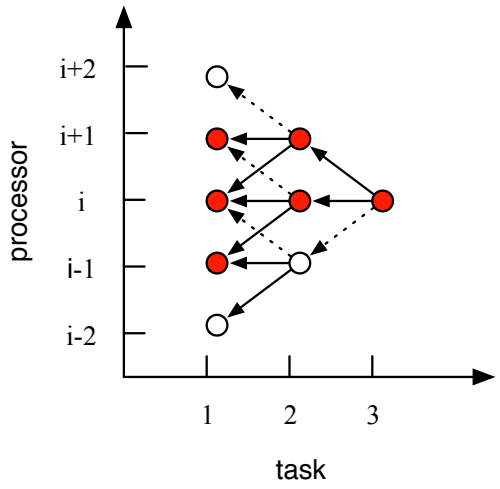

Figure 5: A possible dependency graph for randomized synchronization on the undirected line. Solid arrows represent active dependencies; shaded nodes represent the processor-task pairs on which processor i depends for its 3rd task.

Throughout this section, all of the distributions considered have an expected task time of 2. We consider the geometric distribution with $p = 0.5$, the exponential distribution with $\lambda = 0.5$, the Pareto distribution with $\lambda = 2$, and the truncated normal distribution with $\mu = 2$ and $\sigma = 0.5$. However, while these distributions have the same mean, their order statistics are quite different. The expected maximum of n samples (i.e., the expected time per level in the complete graph of n processors) grows as $\Theta(\log n)$, $\Theta(\log n)$, $\Theta(\sqrt{n})$, and $\Theta(\sqrt{\log n})$, respectively.

5.1 Local synchronization

We ran simulations to model local synchronization under a variety of conditions in order to extend our theoretical results.

In Figure 6, the task time distribution is geometric with $p = 0.5$. Each simulation ran for 5,000,000 time steps. As n increases, the time between levels in the directed cycle quickly approaches what we have shown to be its limit, $2+\sqrt{2}$. The time between levels in the undirected cycle appears to display similar behavior, increasing rapidly and then leveling out around 3.96. The plot of the increasing time between levels in the square torus looks much like those for the other two graphs.

We simulated synchronization on the same graphs for the exponential, Pareto, and truncated normal distributions in Figures 7, 8 and 9 respectively. Each of these simulations ran for 500,000 time steps, with the first 100 tasks for each processor not counted in order to allow the simulation to approach any possible steady state.

In Figure 10, we compare the values computed from Chang and Nelson's upper bounds for $\mathcal{T}^*_{G,f}$ with the results given by simulation, for exponentially distributed tasks on a variety of synchronization graphs. Recall that their bounds depend only on the degree of the graph, not the specific graph, and hence the bounds for the undirected cycle and directed 2-dimensional torus are the same, while the observed behavior is different.

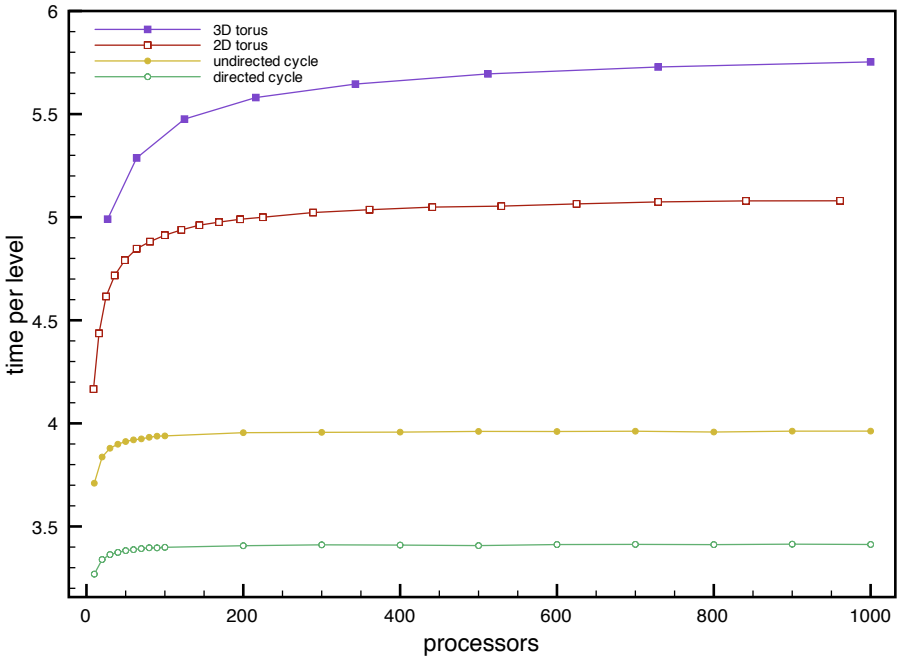

Figure 6: Asymptotic time per level versus n, geometrically distributed task times with p=0.5.

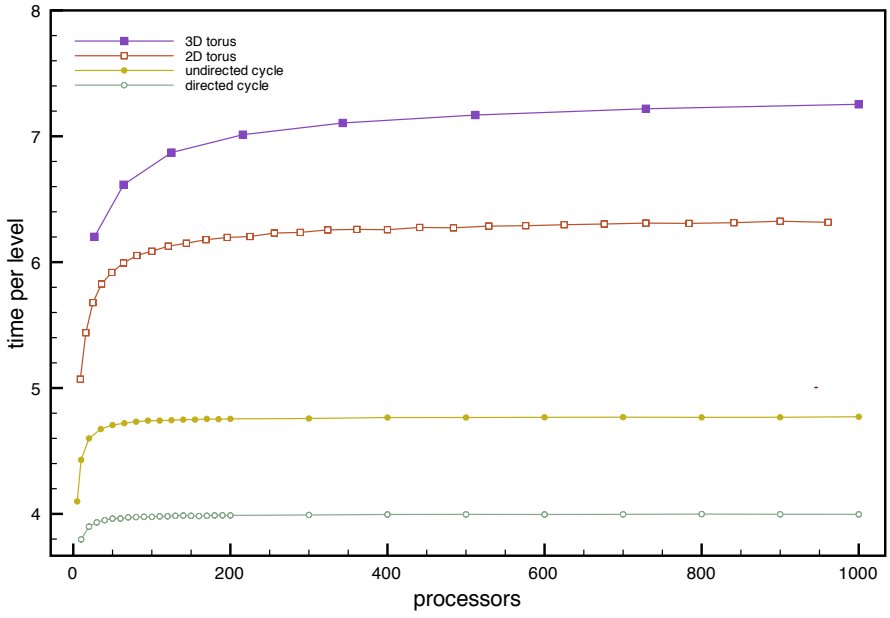

Figure 7: Asymptotic time per level versus n, exponentially distributed task times with $\lambda = 0.5$.

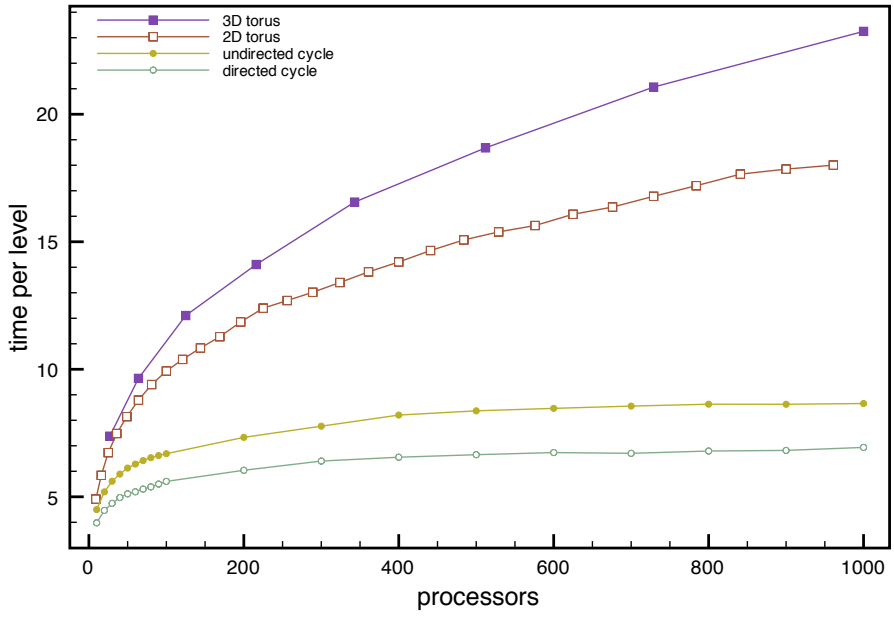

Figure 8: Asymptotic time per level versus n, Pareto distributed task times with $\lambda = 2$.

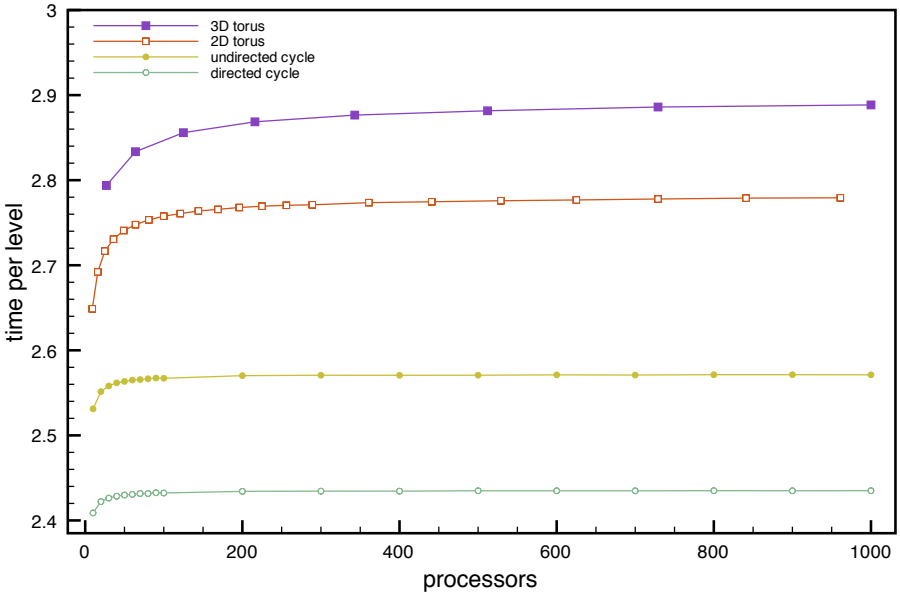

Figure 9: Asymptotic time per level versus n, normally distributed task times with $\mu = 2$, $\sigma = 0.5$.

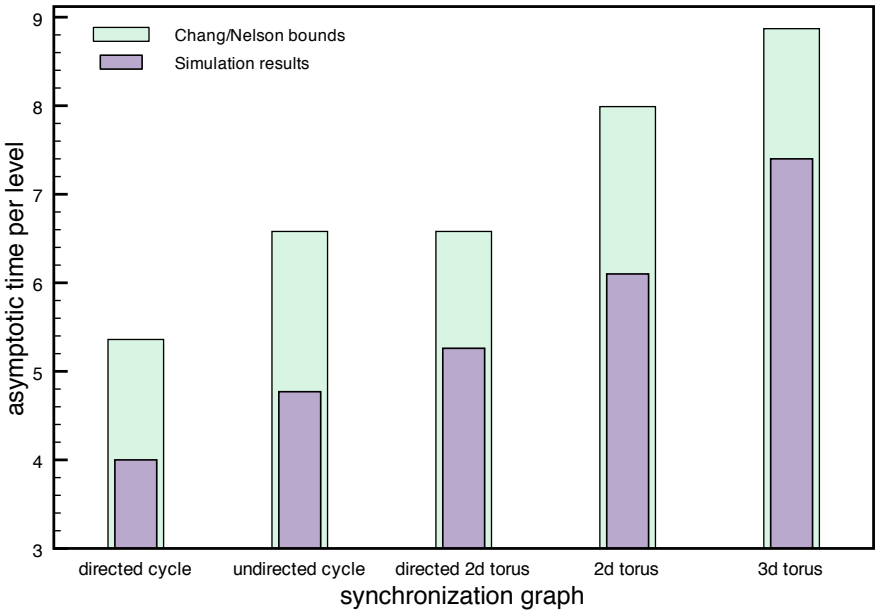

Figure 10: Comparison of Chang and Nelson's upper bounds with our simulation results, exponentially distributed task times with $\lambda = 0.5$.

5.2 First-neighbor and random synchronization

A relaxation of the synchronization requirements is to allow processor p_i to start the next task as soon as c_i of its neighbors have completed their task, where c_i is no more than the in-degree of p_i. We call this the *first-neighbors* model. Note that first-neighbors synchronization is less restrictive than random synchronization since the c_i neighbors are not specified in advance. We simulated first-neighbors and random synchronization for various synchronization graphs and task time distributions, where all the c_i are the same. Our results comparing the undirected cycle with random dependencies with the directed cycle under exponential task times are shown in Figure 11. Here we see that the behavior is nearly identical. This corresponds with our results in Section 4, where we show that the expected number of task-completion dependencies in the dependency graph for the random undirected cycle is the same as for the directed cycle.

In Figure 12, we consider the case of first-neighbors synchronization on the complete graph where processors wait for almost all of their neighbors, $c = n - \sqrt{n}$ and $c = n - \lg(n)$ respectively, for both exponential and Pareto task times. For the $n - \sqrt{n}$ case, we see that the system with Pareto task times is faster for fewer than ≈ 380 processors and then becomes slower. For the $n - \lg(n)$ case, the system with Pareto task times is faster for fewer than ≈ 110 processors.

We compare first-neighbors and random synchronization on a square torus with exponential and Pareto task times in Figure 13. Our results show that, for all n we tested, Pareto-distributed task times lead to faster synchronization with the first-neighbors model where processors wait for the first 2 out of the 4 possible neighbors in the square torus. However, when processors wait for 3 out of 4 of their neighbors, Pareto synchronization is faster only up to $n = 196$, even though the expected value of the second largest of 4 Pareto random variables is ≈ 1.83, while the expected value of the second largest of 4 exponential random variables is ≈ 2.17.

These simulations also show that random-neighbor synchronization on the square torus, even

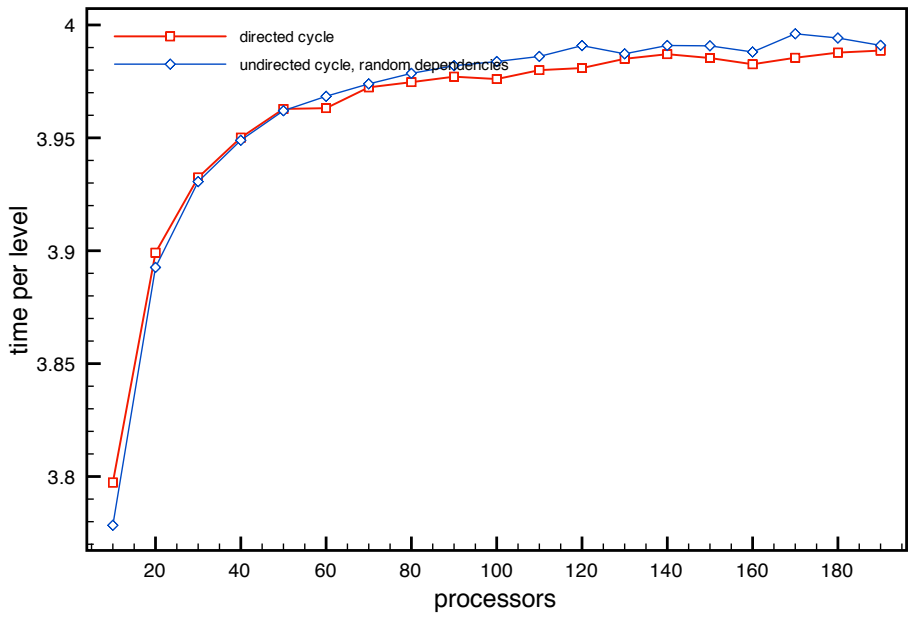

Figure 11: Comparison of the undirected cycle with random dependencies and the directed cycle, with exponential task times.

when processors wait for only one neighbor out of 4, is slower than first-neighbors synchronization, even when processors wait for 3 out of 4 neighbors. These results hold for even small values of n. With random-neighbor synchronization, we also see the characteristic difference between the exponential and Pareto distributions that does not seem to appear with first-neighbors synchronization. With random-neighbor synchronization, varying the number of neighbors waited for seems to matter less than changing the distribution, whereas with first-neighbors synchronization, the data for synchronization with 1 neighbor and Pareto-distributed task times appears more similar to that for synchronization with 1 neighbor and exponential-distributed task times than it does to that for synchronization with 2 neighbors and Pareto-distributed task times.

In Figure 14, we compare synchronization times on three models where each processor depends on one neighbor — directed cycle, undirected cycle with random dependencies, and the complete graph with one random neighbor — with those on three models where each processor depends on 2 neighbors — undirected cycle, square torus with 2 random dependencies, and the complete graph with 2 random neighbors. For $n > 300$, we find that the complete graph with one random dependency is actually slower than the undirected cycle.

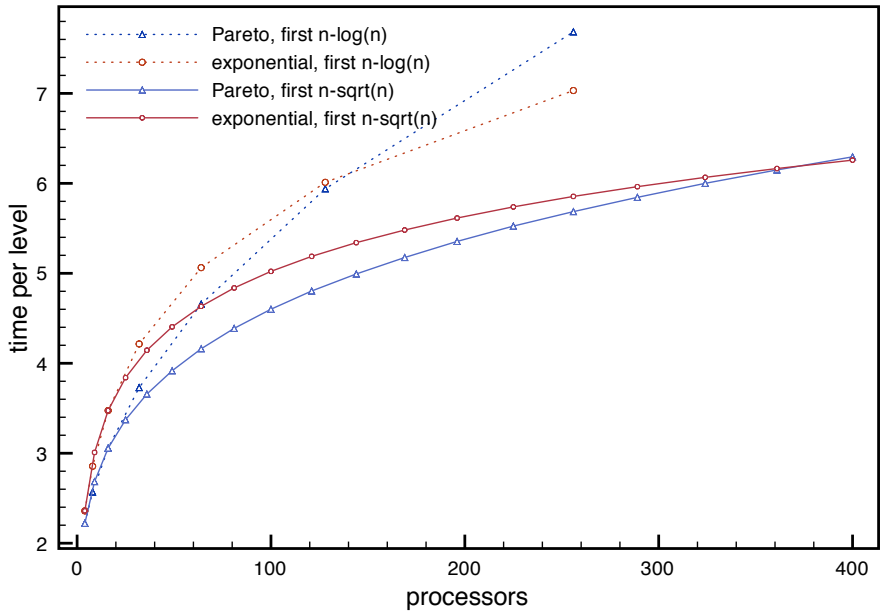

Figure 12: Comparison of Pareto and exponential task times on a complete graph, waiting for the first $n - \lg(n)$ and $n - \sqrt{n}$ neighbors.

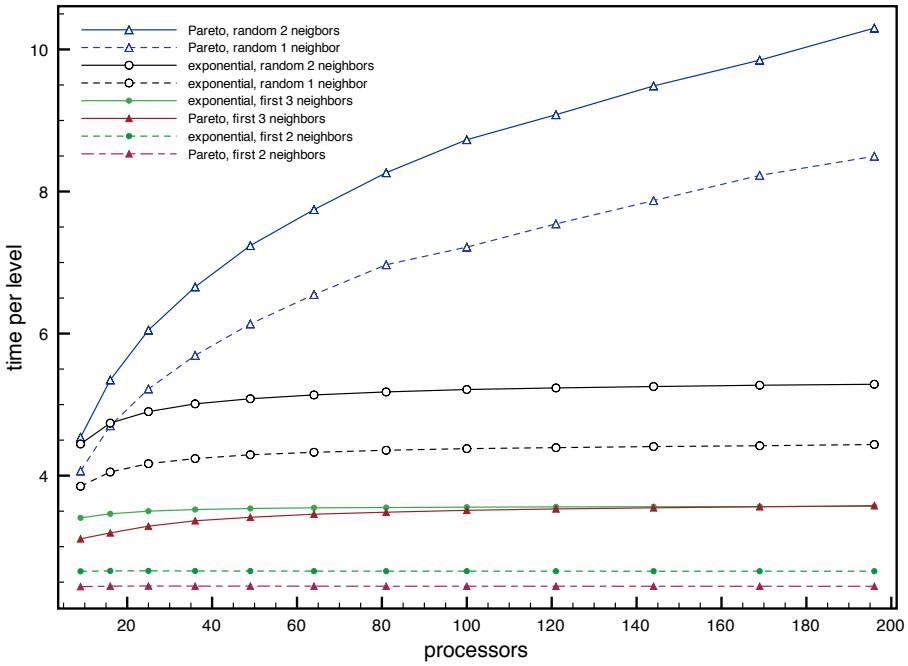

Figure 13: Comparison of random and first-neighbors synchronization on a square torus with Pareto and exponential task times.

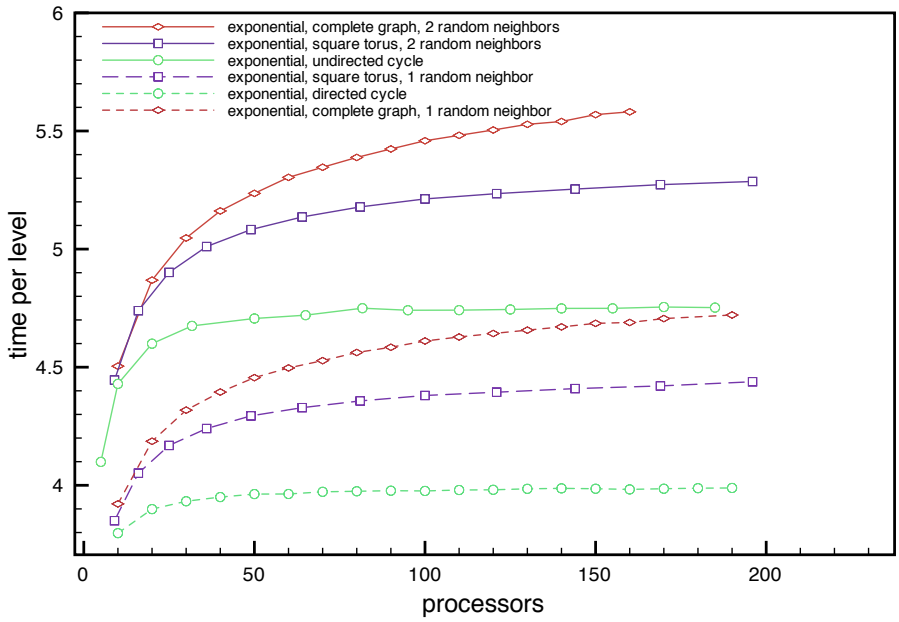

Figure 14: Comparison of synchronization with 1 and 2 neighbors with different graph shapes.

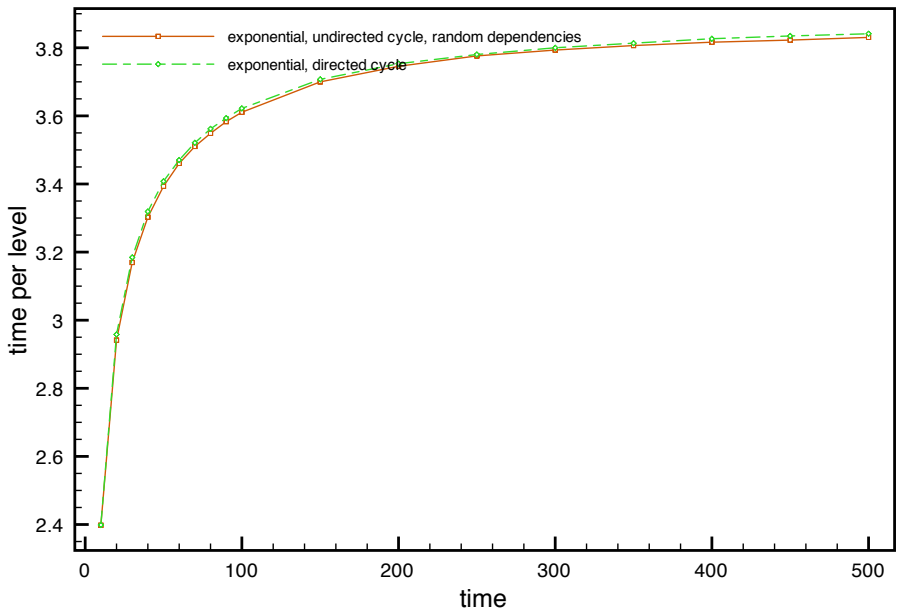

Figure 15: Comparison of time to converge with exponentially distributed tasks on the directed cycle and the undirected cycle with random dependencies.

5.3 Comparison of convergence times

We know that the expected synchronization time in the models we study will converge to a constant if the number of processors n is held constant while the time of the simulation goes to infinity as long as the task time distribution has finite mean, since the synchronization time cannot exceed the expected maximum order statistic of n random variables. But how quickly does it approach that constant? To study this, we simulated synchronization on various graphs with several distributions, ending the simulation after increasing amounts of time t. That is, the simulation is run up to time t, and at that point, the total number of tasks completed over all processors is calculated, which is then used to computer the average time per level over all of the processors. For each t, the simulation was run 1,000,000 times, and an average was taken over all simulations. The error bars on some of the graphs represent the sample variance over these 1,000,000 runs of the simulation.

In Figure 15 we see that the synchronization times with exponential task times and 20 processors on the directed cycle and the undirected cycle with random dependencies are nearly identical for all of the ending times we studied. Again, this agrees with our results in Section 4.

In Figure 16, we compare synchronization on the directed cycle with exponential task times for different numbers of processors. After 500 time units, we found that the cycle with 10 processors had nearly reached its steady-state value while the ones with 20 and 50 were much slower to converge.

In Figures 17 and 18, we compare synchronization on the directed cycle with normal and Pareto task times, respectively, for different numbers of processors.

In Figure 19, we compare convergence for synchronization with exponential and Pareto task times. Our data shows that for small values of t, less than ≈ 100, the directed cycle with Pareto task times is actually faster, but it becomes slower for larger values.

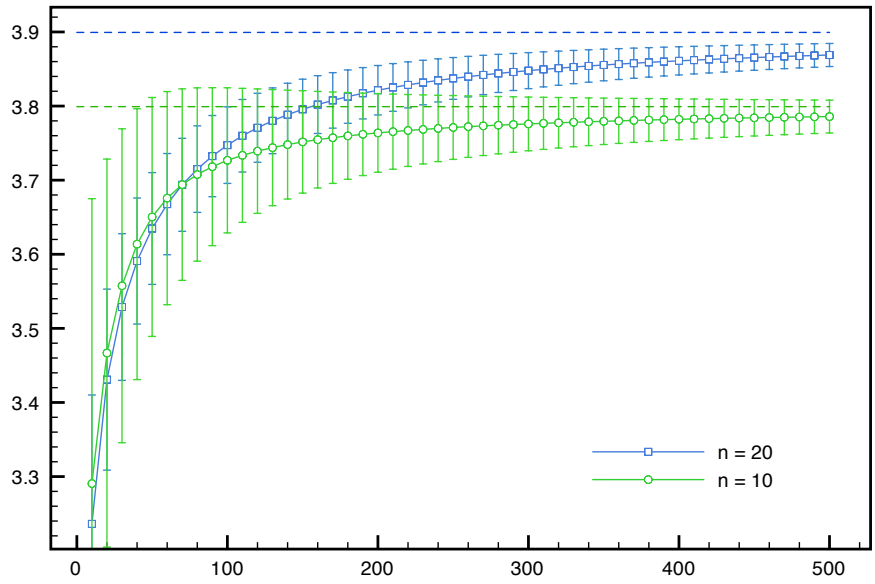

Figure 16: Comparison of time to converge on the directed cycle with exponential task times and 10, 20 and 50 processors.

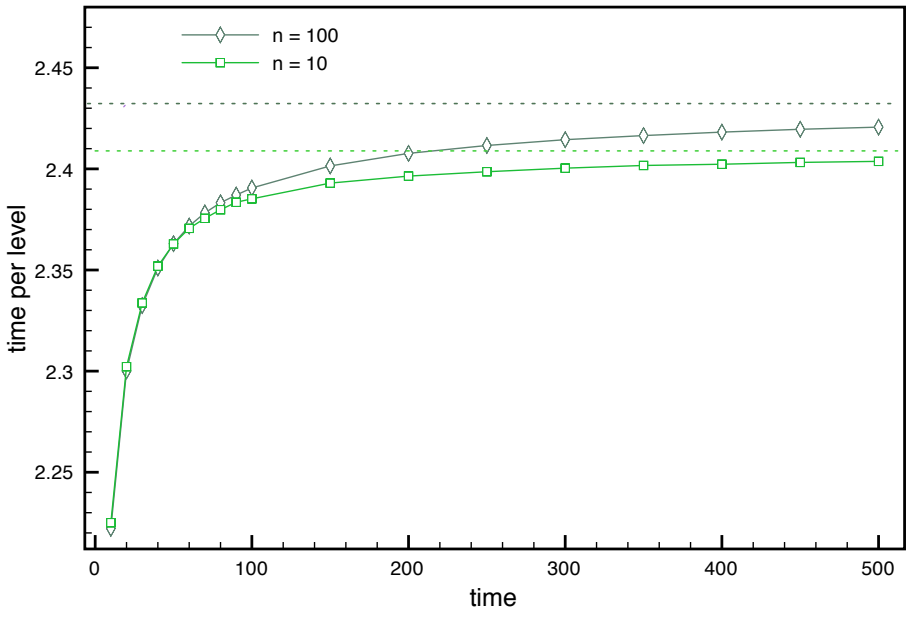

Figure 17: Comparison of the time to converge on the directed cycle with normal task times and 10 and 100 processors.

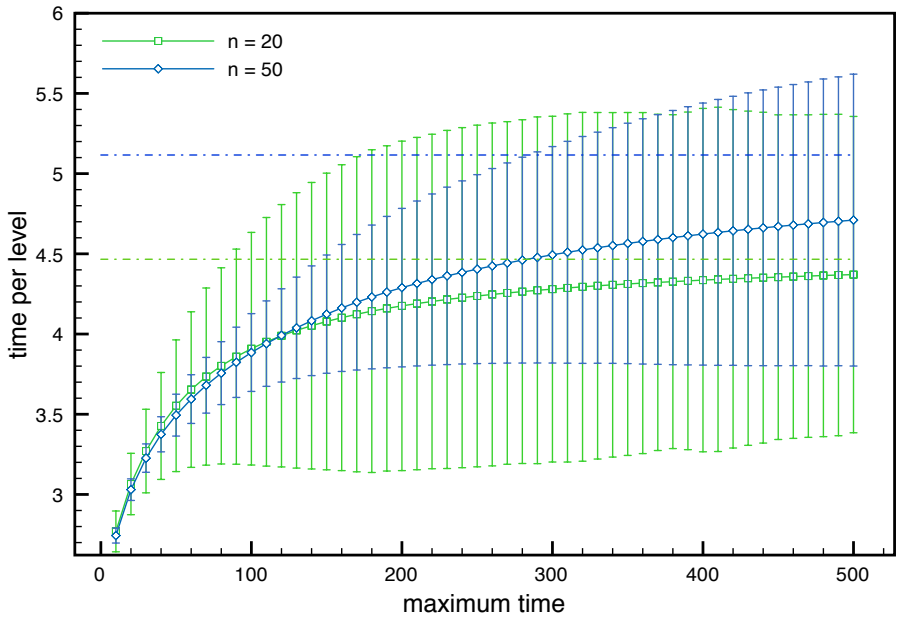

Figure 18: Comparison of the time to converge on the directed cycle with Pareto task times and 20 and 50 processors.

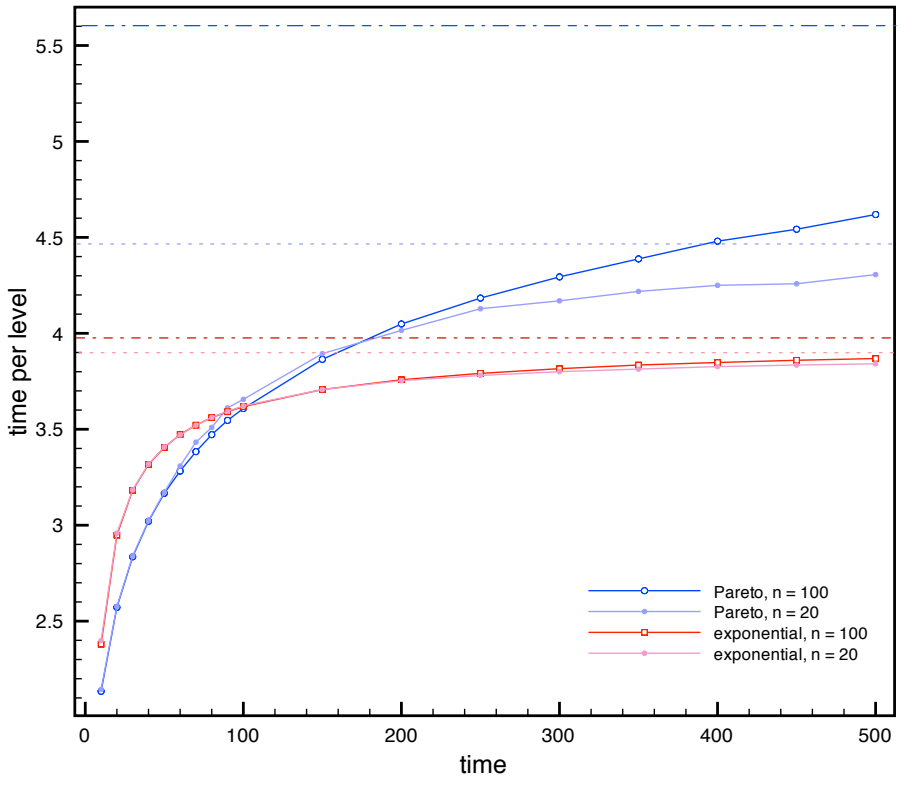

Figure 19: Comparison of the time to converge on the directed cycle with exponential and Pareto task times.

6 Conclusion and Future Work

There are other models of synchronization that correspond to various parallel computing techniques. One possible new model involves slack, in which the synchronization constraints are enforced with a "grace period." This is sometimes used in simulating nonblocking communication. A further model that may lend itself to the kind of analyses carried out here allows the possibility of processor failure, similar to the model described by Taylor and Van Dijk [24].

Another way to reduce the effects of synchronization on program efficiency is to batch tasks, with each processor solving several before synchronizing. For example, this is often used in branch-and-bound calculations, where occasionally global bounds need to be determined. More generally, it occurs in many distributed manager-worker programs in which work is occasionally rebalanced, or in simulations where the workload shifts from one region to another. For this latter situation typically the shift involves many nearby sites and persistent imbalance for a number of cycles, in which case the imbalance does not smooth itself out. However, it may be more efficient to wait many cycles before taking a global picture of the situation, rather than doing so after every cycle. We have some results on the effects of synchronization with batching.

In addition to the approaches with which we have obtained results, there are other directions that may yield more advances in this area. One method is a probabilistic-method bounding technique. In a directed square grid of size n, which corresponds to the task graph of a directed cycle of n processors after t tasks where $n \geq t$, we can show that there are $2^{n-d-2}\binom{n+d-1}{d}$ pairs of paths that differ in exactly $d < n-1$ places. Using a bounding technique like the Chung-Erdös second-moment method, it may be possible to use this information to get bounds on quantities like the expected number of paths that contain a certain number of extreme order statistics or values above a certain threshold.

Many of the Markov models for local synchronization are "almost martingales" in the sense of Bellare and Impagliazzo [4] when viewed as Markov reward processes where the cost at each state is the number of waiting processors; both the conditional expectation and greatest possible value of the change in cost after one time step are bounded. Using the almost martingale extension of Azuma's inequality in [4], one may be able to find upper bounds on the probability of being in a state with very few working processors. Since their model does not assume memorylessness, one may also be able to use it to obtain results for non-memoryless task distributions. In particular, the Taylor and Van Dijk model mentioned above can be used to construct Markov models that obey arbitrarily strict conditional-expectation bounds.

References

[1] F. Baccelli and Z. Liu. On the execution of parallel programs on multiprocessor systems – a queuing theory approach. *Journal of the ACM*, 37(2):373–414, 1990.

[2] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: characteristics and caching implications. *World Wide Web*, 2:15–28, 1999.

[3] F. Baskett and A. Smith. Interference in multiprocessor computer systems with interleaved memory. *Communications of the ACM*, 19(6):327–334, 1976.

[4] M. Bellare and R. Impagliazzo. A tool for obtaining tighter security analyses of pseudorandom function based constructions, with applications to PRP → PRF conversion, 1999.

[5] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computing: Numerical Methods*. Prentice Hall, 1989.

[6] C. S. Chang and R. Nelson. Bounds on the speedup and efficiency of partial synchronization in parallel processing systems. *Journal of the ACM*, 42(1):204–231, 1995.

[7] H. Cohn, N. Elkies, and J. Propp. Local statistics for random domino tilings of the aztec diamond. *Duke Mathematical Journal*, 85:117–166, 1996.

[8] A. Downey. The structural cause of file size distributions. In *Proc. 2001 SIGMETRICS*, pages 328–329, 2001.

[9] A. G. Escribano, V. C. Payo, and A. van Gemund. On the loss of parallelism by imposing synchronization structure. In *Proceedings 1st EURO-PDS Int'l Conference on Parallel and Distributed Systems*, pages 251–256, June 1997.

[10] A. G. Escribano and A. van Gemund. An algorithm for transforming NSP to SP graphs. Technical report, Technical University, Delft, The Netherlands, 1996.

[11] H. Han, C. Tseng, and P. Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 26(5):591–612, 1998.

[12] J. F. C. Kingman. The first-birth problem for an age-dependent branching process. *Annals of Probability*, 3(5):790–801, 1975.

[13] G. Korniss, M. Novotny, H. Guclu, Z. Toroczkai, and P. Rikvold. Suppressing roughness of virtual times in parallel discrete-event simulations. *Science*, 299:677–699, 2003.

[14] U. Legedza and W. Weihl. Reducing synchronization overhead in parallel simulation. In *Workshop on Parallel and Distributed Simulation*, pages 86–95, 1996.

[15] A. Malony, V. Mertsiotakis, and A. Quick. Stochastic modeling of scaled parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 274–279, December 1994.

[16] J. Martin. Last-passage percolation with general weight distribution. *Markov Processes and Related Fields*, 12:273–299, 2006.

[17] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1:226–251, 2003.

[18] S. Rajsbaum and M. Sidi. On the performance of synchronized programs in distributed networks with random processing times and transmission delays. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):939–950, 1994.

[19] A. Salamon. Task graph performance bounds through comparison methods. Technical report, Department of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, Jan. 2001. MSc Dissertation (141 pages).

[20] N. Sloane. *The On-Line Encyclopedia of Integer Sequences*. http://www.research.att.com/~njas/sequences/.

[21] R. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.

[22] D. Stoyan. *Stochastic Orders and Their Applications*. Wiley, 1983.

[23] T. Tabe, J. Hardwick, and Q. Stout. Statistical analysis of communication on the IBM SP2. *Computing Science and Statistics*, 27:347–351, 1995.

[24] P. Taylor and N. van Dijk. Strong stochastic bounds for the stationary distribution of a class of multicomponent performability models. *Operations Research*, 46:665–674, 1998.

[25] J. Tsitsiklis and G. Stamoulis. On the average communication complexity of asynchronous distributed algorithms. *Journal of the ACM*, 42(2):382–400, 1995.

[26] G. J. van Houtum, W. H. M. Zijm, I. J. B. F. Adan, and J. Wessels. Bounds for performance characteristics: a systematic approach via cost structures. *Stochastic Models*, 14:205–224, 1998.