PERFORMANCE ANALYSIS OF A SYNTHETIC TASK PRECEDENCE MODEL WITH THE CONTROL OF A TASK SCHEDULING ALGORITHM

Technical Report CSE-92-05

Yu Gong
and
Carolyn McCreary

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

Ian F. Akyildiz

School of Electrical Engineering
Georgia Institute of Technology
Atlanta, GA 30332
Performance Analysis of a Synthetic Task Precedence Model with the Control of a Task Scheduling Algorithm

Yu Gong    Ian F. Akyildiz    Carolyn L. McCreary

September 1992

Abstract

In a multiprocessor environment, a sequential program is first divided into various subtasks, and a task precedence relation is computed based on dependences between these tasks. This relation is represented by a directed acyclic graph (DAG), with the nodes of the DAG corresponding to individual tasks, and the arcs corresponding to restrictions on the order of task execution.

In the second phase, a task scheduling algorithm assigns the various tasks of the DAG to individual processors. In this phase, to minimize the total execution time of the DAG, a trade-off must be made between communication cost, i.e., when related tasks are assigned to different processors, and processing time, i.e., when multiple tasks are assigned to the same processor.

Design of task scheduling algorithms has been an active research subject in recent years. The performance of a scheduling algorithm has primarily been evaluated through case studies. The major problem with the case-based investigations is that multiple and exhaustive DAGs must be studied individually, which is often not feasible. This dilemma illustrates the need for studies of a synthetic task precedence model, able to capture common properties of all possible precedence relations for a class of programs.

In this paper, a synthetic task precedence model with a built-in task scheduling algorithm is developed. From this model, performance measures are computed analytically, including the mean and variance of completion time, the probability distribution of completion time and speedup. Analytical results are validated with experimental results obtained through extensive simulations.

Key words: parallel computation, task, precedence relation, task scheduling algorithm, completion time, speedup, synthetic model, performance analysis, simulation.
Performance Analysis of a Synthetic Task Precedence Model with the Control of a Task Scheduling Algorithm

Yu Gong (*), Ian F. Akyildiz (**), Carolyn L. McCreary (*)

(*) Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849
Email: gong@eng.auburn.edu
Tel: (205)844-6316(Gong)
Email: mccreary@eng.auburn.edu
Tel: (205)844-8307(McCreary)
Fax: (204)844-6329

(**) School of Electrical Engineering
Georgia Institute of Technology
Atlanta, GA 30332
Email: ian@cc.gatech.edu
Tel: (404)894-5141
Fax: (404)894-3188
1 Introduction

In a multiprocessing environment, a sequential program is first analyzed for its data and control dependencies, in order to produce a task precedence relation. The task precedence relation is customarily expressed by a directed acyclic graph (DAG), with the nodes of the DAG representing program modules, called tasks, which run on individual processors. A directed arc connecting two nodes represents a constraint on the execution order of the two corresponding tasks. The parent node of an arc must complete its execution before the child node can be executed, this restriction prohibiting the parallel execution of these particular tasks.

In the literature, there are two ways to characterize parallel programs: explicit representation or implicit representation. In the explicit representation, the DAG is assumed to be known and fixed. In this case, where the execution environment (DAG) is assumed to be known and ideal (no resource contention), the task graph model can be analyzed using stochastic PERT networks (Program Evaluation Review Technique) [1]. The only difference from task graphs is that in these networks, random variables are assigned to the arcs, while in task graphs they are assigned to the nodes. Therefore, the computation of the project completion time of a PERT network is equivalent to the computation of the overall execution time of a task graph. Exact methods exist for series-parallel graphs and tree type graphs [2]. Series-parallel graphs are called reducible task graphs [1, 3] when the task execution times appear only once in the equation defining the overall computation time. In other words, the task graph is a recursive serial and parallel combination of tasks. Task graphs representing programs written in Ada and parallel Pascal have this structure. In non-series-parallel graphs, task execution times may appear more than one time in the equation defining the overall execution time. In [1, 4] non-series-parallel graphs are transformed to series-parallel ones by duplicating or splitting arcs. They derive stochastic upper bounds on the distribution of overall execution time. In [5], it is shown that the deterministic version of the task graph gives the stochastic lower bound, while the independent version gives the stochastic upper bound on the distribution of the overall execution time.

The methods developed for explicitly represented parallel programs while taking resource contention into consideration may be classified into two groups: the hierarchical decomposition method [6, 7, 8] which is the combination of Markov chain models and queueing network models. The other group is the combination of task graph reduction methods with modified mean value analysis (MVA) algorithms of queueing network models [9]. The hierarchical method in [6] is accurate but the state explosion problem makes it difficult to apply. Kapelnikov et al. [7] propose a heuristic method to overcome this difficulty. Their method is based on the successive aggregation of portions of the task graph, called segments. Segments are subgraphs for which task executions inside occur without any interaction with the nodes outside of the segment. They analyze segments in isolation. Then they aggregate the segments to compute the overall execution time of the task graph. Ghodsi and Kant [8]
have applied the hierarchical method by considering the simultaneous resource problem. They combine Markov chain models and queueing network models for the simultaneous resource possession problem. The general idea of the MVA methods [9] is to adjust the arrival theorem of the MVA method, which is valid for product form queueing network models in order to apply it to the non-product form queueing network models which arise in the context of concurrent system modeling.

For implicitly represented parallel programs the DAG is not known a priori, and also may be subject to dynamic changes. Implicitly represented parallel programs have been studied both without [10, 11, 12, 13] and with [14] resource contention. Gelenbe et al. [10] proposed a random graph model in which precedence relationships are determined in a probabilistic manner. Parallel programs are modeled by directed acyclic graphs, such a DAG consisting of $K$ nodes in which an arc from node $i$, $i = 1, 2, ..., K - 1$ to node $j, j = i + 1, i + 2, ..., K$ exists independently with a fixed probability $p$. They obtain an asymptotic speed-up for this type of task graph. Gelenbe et al. [11] analyze random series-parallel graphs for which the combination in series and parallel are defined with probabilities. They compute the probability distribution function of the overall execution time. Eager et al. [12] characterize parallel programs by a parameter called average parallelism. This is the maximum speed-up which can be reached with an infinite number of processors. They derive bounds on the performance measures. Kant and Liaw [13] characterize a family of parallel programs by using dynamic trees. These programs are generally iterative algorithms containing one or more nested parallelizable loops. Their task graphs take the form of a dynamically growing and shrinking tree. The typical input parameters for a dynamic tree to describe the underlying structure are the number of children at each level, the number of levels, and some characterization of the growth and shrinkage at each level. They model the behavior of a dynamic tree by a Markov chain with the local balance property and obtain the distribution of the overall execution time of the parallel program.

The performance of implicitly represented parallel programs on a simple parallel processing system, taking into consideration resource contention, to our knowledge has only been studied by Jiang et al. [14] so far. They propose two analytical methods to evaluate simple parallel processing systems. The average parallelism model is based on the idea of characterizing the task graphs by their average parallelism which is defined as the speed-up with an unbounded number of processors. They model a parallel program with a given average parallelism by a rectangular task graph which maintains its mean performance.

Note that although many methods have been proposed in the literature to analyze random DAGs, none of the above studies are adequate to solve the problem we address in this paper. None of the above studies take into account the effect of a task scheduling algorithm on the performance of parallel systems. The major objective of a task scheduling algorithm is to find the optimal assignment of tasks to processors, in order to achieve the shortest possible completion time of a given DAG.
Existing task scheduling algorithms include the critical path methods [15, 16, 17, 18], the list scheduling methods [19], the graph decomposition methods [20, 21], and the workload schedule methods [22]. We have observed that the performance (e.g., completion time) of task scheduling algorithms is mainly analyzed by first selecting several test DAGs, and then analyzing the algorithm with respect to the selected examples. When several algorithms are compared, the best algorithms alternate from DAG to DAG. In this case, an important question is: “What is the expected completion time for a scheduling algorithm when all DAGs are potentially possible?”

To answer the above question, a random task graph model has to be established. The random task graph model should be able to capture properties of all possible task precedence relations. The problem becomes more complex when task times are also considered to be random variables. We refer to the probabilistic task precedence model as the synthetic task precedence model.

The closest model in the literature to our study is [11]. Our model, however, considers the communication costs and also as mentioned before the effect of the scheduling algorithm on the performance. This is a significant difference from [11], since it implies that even in the case where task precedence relation permits parallel execution, a trade-off between sequential and parallel execution of tasks must still be made. For that purpose, the model proposed herein intrinsically includes a task scheduling algorithm as one of its components. The “graph decomposition algorithm” [20, 21] given in section 2 is used as the scheduling algorithm in the model. The reason for including a scheduling algorithm as an inseparable part of the model is based on the fact that the statistics for running a random DAG are closely related to how the algorithm assigns the tasks at all levels of parallelism. From the analytical model, we derive the mean and standard deviation of completion time, its probability distribution, and the speedup of the algorithm. To our knowledge, this is the first work that considers dynamic task precedence graphs together with a task scheduling algorithm.

The paper is organized as follows. In Section 2, we describe the synthetic task precedence and scheduling model. Section 3 contains the performance analysis of the model. In Section 4, we discuss various issues using numerical results obtained from analytical solutions and extensive simulations. We conclude the paper and point out possible future directions in Section 5.

2 The Model

A scheduling algorithm needs to explore the highest degree of parallelism on the physical system for a given task precedence relation. Consequently, the performance of the algorithm is closely related to the task precedence relation itself, as well as the physical characteristics of the multiprocessor. Among the many factors that may restrict the degree of parallelism, the following three are the most important.
i) Restrictions from the task precedence relation, (i.e., synchronization constraints).

ii) Processor contention.

iii) Communications cost.

Restriction i) is determined by the program logic. The other two factors are related to the physical properties of the system. Since we assume we have an infinite number of processors, processor contention, ii), is avoided.

The synthetic model consists of two parts: a probabilistic task precedence diagram and a scheduling algorithm. The first part is used to model the logical (machine independent) restrictions on the task execution order, i.e., capturing i), above. The second part provides additional restrictions based upon its decision on the trade-off between execution time and communication costs, i.e., capturing iii) above.

In this section, before we discuss the details of the two parts of the synthetic model, we first provide some background information and define some terminology.

2.1 Background and Definitions

Definition 1 (DAG). A Directed Acyclic Graph (DAG) is defined by \( G = (V, E) \) where \( V \) is the set of nodes (vertices) and \( E \) is the set of directed edges. It is important that there is no cycle in a DAG.

Definition 2 (Clan). A clan in a DAG is defined as a subset \( C \subseteq V \), such that for any nodes \( x, y \in C \), and for any node \( z \in V - C \), the following is satisfied:

\[
\begin{align*}
\text{\( x \) is reachable from \( z \) \quad \text{iff} \quad \text{\( y \) is reachable from \( z \)}}
\text{\( z \) is reachable from \( x \) \quad \text{iff} \quad z \text{ is reachable from } y
\end{align*}
\]

Informally, a clan is a set of nodes in a DAG whose member nodes are related to external nodes in exactly the same way.

One immediate implication of Definition 2 is the following: if the DAG represents the task precedence relation, the tasks in a clan can be clustered onto one processor and be treated as a single task. This is because once one of the tasks in a clan is enabled, so are all the other members. Therefore, all tasks in the clan may be executed sequentially without interruption.

Depending upon the properties of the communications hardware, clustering tasks into one processor can save communications costs. One simple example is when all nodes in a clan have but a single common parent. The communications cost then becomes zero if all tasks are clustered into the same processor as their parent's. However, one must not overlook the increased computation time.
when multiple tasks are assigned to the same processor: an optimal decision on how to cluster the tasks is certainly needed.

The second implication of the notion of clan is as follows: the tasks in a clan can be first assigned to processors to obtain the minimum completion time; then the entire clan is treated as a single task which has task time equal to the completion time of the clan. The original DAG is now reduced due to the shrinkage of the clan to a node. This same procedure can be repeated on new DAG until the DAG is reduced to containing only one node. This fact helps us to hierarchically design a task scheduling algorithm, called a graph decomposition algorithm [20, 21].

In [23, 24], it is proved that any DAG can be decomposed (constructed) using only three clan structures: primitive, independent, and linear, as shown in Figure 1.

**Definition 3 (Primitive Clan).** A primitive clan is one that does not contain nontrivial subclans.

**Definition 4 (Trivial/Nontrivial Clan).** A trivial subclan of a clan $C$ is either a singleton subset of $C$, or the entire set $C$. All other subclans of $C$ (if any) are nontrivial.

**Definition 5 (Independent Clan).** An independent clan contains only isolated nodes.

**Definition 6 (Linear Clan).** A linear clan is a set whose nodes are in a linear order.

### 2.2 Probabilistic Task Precedence Diagram

We consider task precedence diagrams represented by connected DAGs. A further restriction is imposed here that all DAGs have a single source node, a node that is the ancestor of all other nodes, and a single sink node, a node that is the descendant of all other nodes. The source node corresponds to the starting point of the program and the sink node is the ending point.

To illustrate the notion of random task precedence diagrams we first introduce a graphical representation method for task precedence relations. Any task precedence relation can be represented by a DAG with a single source and a sink node. All nodes in the DAG are either solid or empty as
Figure 2: An Example Representation of Random Task Precedence Relation

shown in Figure 2. A solid node indicates that the node is a clan or a task, i.e., its execution consumes processor time. To help identifying clans in a DAG, sometimes adding dummy nodes is helpful. The empty nodes in Figure 2 represent dummy nodes. Since dummy nodes are added to assist reading the DAG, they can be freely removed from the graph, provided the semantics of the precedence relation is not consequently altered.

In addition, a solid node may be labelled either by $I$, indicating that the node represents an independent clan, or by $L$, indicating that the node represents a linear clan, or not labelled at all, meaning the node is a primitive clan. Note that a single task by itself is a primitive clan.

We call a solid node that is labelled by $I$ or $L$ an $I$-node or an $L$-node, respectively. An non-labelled solid node is referred to as a primitive node. Any precedence diagram is generated probabilistically and hierarchically starting from a single $I$ or $L$ node, using the following four rules [11].

**L-Rule.** Given an $I$-node, with probability $a_n$ ($n \geq 2$), it is replaced by an independent clan consisting $n$ $L$-nodes as shown in Figure 3(a).

**L-Rule.** Given an $L$-node, with probability $b_n$ ($n \geq 2$), it is replaced by a linear clan consisting $n$ $I$-nodes as shown in Figure 3(b).

**IP-Rule.** With probability $a_1$, an $I$-node loses its label to become a primitive node.

**LP-Rule.** With probability $b_1$, an $L$-node loses its label to become a primitive node.
It can easily be seen that the IP- and LP-Rules are degenerative forms of the I- and L-Rules, respectively, for the case $n = 1$. This is the case when the dynamic expanding of the graph terminates. It is also clear from the rules that

\[
\sum_{n=1}^{\infty} a_n = 1, \quad \sum_{n=1}^{\infty} b_n = 1
\]  

Because an $I$ or $L$ node may randomly grow into complicated structures, we regard a solid node with label $I$ or $L$ as a random graph. We therefore use $I$-node ($L$-node) and $I$-graph ($L$-graph) interchangably.

We point out that the dynamic graph constructing process by the four rules described above is in the so-called canonical form. It is shown in [23, 24] that all graph decomposition/construction processes possess canonical forms. Finally, in a random graph, edges may be labelled by a weight, which is understood to be the communication cost between clans or tasks.

## 2.3 Task Scheduling Algorithm

The general idea of the graph decomposition algorithm has been introduced in Section 2.1. We now describe more details of the algorithm. These details are closely related to specific system architectures, because different systems give rise to different ways to determine the communication cost. The communication cost, in turn, is one of the two determining factors (the other being the processor time) for design of the details of a task scheduling algorithm. So we first need to make some assumptions about the underlying system in question. To exhaustively study all possible architectures is out of the scope of this paper.

We assume that the communication cost between any two processors in the system is a constant, and it is equal to $c$ time units. This assumption is valid for shared memory multiprocessors, where
tasks (and thus processors) do not require the exchange of bulky data over the communications facility. For this type of architecture, an efficient scheduling algorithm is introduced together with the method for computing the communication cost.

The algorithm works recursively, in a reverse order of the graph generating process, i.e., it starts from primitive clans, which is described below.

a) Primitive and Linear Clans.

Tasks belonging to a primitive clan or a linear clan are assigned to a single processor. The completion time for the resulting task assignment is the sum of the completion times of the components.

b) Independent Clan.

For an independent clan containing \( n \) linear components, we use \( x_1, x_2, \ldots, x_n \) to denote the completion times of the components. Without loss of generality, we assume

\[
x_1 \geq x_2 \geq \ldots \geq x_n
\]  

(2)

Moreover, \( X = \{x_1, x_2, \ldots, x_n\} \) denotes the set of the completion times, and \( \mathcal{X}(p) = \{X_1, X_2, \ldots, X_p\} \) denotes a partition of \( X \) into \( p \) non-empty subsets. Figure 4 illustrates the partition of \( X \) into \( p \) subsets. Figure 4 also illustrates the communication costs if all subsets are assigned to processors using the method described below.

Given a partition \( \mathcal{X}(p) \) as shown in Figure 4, the algorithm assigns the first subset \( X_1 \) to the parent processor \( P \) of the independent clan, thus there is no communication cost, indicated by 0 in
Figure 4. The parent processor $P$ is the one which enables the independent clan during the program execution. All other subsets from $X_2$ to $X_p$ are assigned to separate processors other than $P$. There are communication costs between these processors and the parent processor $P$, indicated by $c$ in Figure 4. Note that we assume the communication costs $c$ are the same for all processors. When the processors complete their executions, they send messages to processor $P$ to synchronize the entire independent clan. Thus, if there is a subsequent independent clan following that one, $P$ can enable it after the synchronization. The notation $c$ on the edges of Figure 4 represents the communication costs of this assignment.

We have made the following assumptions about the system communication metric:

1. Broadcasting a message to other processors takes a constant $c$ time units.

2. No contention is encountered when independent tasks send messages to the same destination.

3. The execution time of nodes placed on the same processor is the sum of the individual execution time.

Denote by $t(\mathcal{X}(p))$ the completion time of the independent clan shown in Figure 4, given the partition $\mathcal{X}(p)$. Under the above assumptions, the completion time for the entire clan is the maximum time among all branches, calculated from:

$$t(\mathcal{X}(p)) = \max\{t(X_1), t(X_2) + 2c, \ldots, t(X_p) + 2c\}$$

where $t(X_i)$ denotes the optimal completion time for the $i$-th linear component ($1 \leq i \leq p$).

Clearly, the algorithm must find the optimal partition that has the smallest completion time. We denote this optimal completion time by $t(X)$. Thus,

$$t(X) = \min_{\forall \mathcal{X}(p)} t(\mathcal{X}(p))$$

The problem of finding the optimal partition is, in general, an NP-complete problem. We now state a result which helps the algorithm to avoid an exhaustive search.

**Lemma 1:** If an independent clan has $n$ linear components. The completion times of the $n$ components are given in the set $X = \{x_1, x_2, \ldots, x_n\}$, for $x_1 \geq x_2 \geq \ldots \geq x_n$ (WLOG), then

(i) A partition of $X$, $\mathcal{X}(p) = \{X_1, X_2, \ldots, X_p\}$, has the shortest completion time if

1. $X_1 = \{x_1, x_2, \ldots, x_J\}$, such that

$$\sum_{i=1}^{J} x_i \geq 2c \quad \text{and} \quad \sum_{i=1}^{J-1} x_i < 2c$$


2. All other sets contain only one component each, i.e.,

\[ \forall i (2 \leq i \leq p) : |X_i| = 1 \] (6)

(ii) For the optimal partition, the completion time of the clan is:

\[ t(X) = \max \{ \sum_{i=1}^{J} x_i, x_{J+1} + 2c \} \] (7)

Proof. It is easy to see, using (3), that for any optimal partition the condition (6) is true. We now prove that for any optimal partition, satisfying (6), the partition can be modified, without enlarging the completion time, to satisfy the condition (5). During the modification, the cost for the entire clan is not increased.

Step 1: Assume there are \( x_i \) and \( x_j \), where \( x_i < x_j \). Furthermore, \( x_i \) is in the first set but \( x_j \) is outside. Then the completion time for the entire clan will not be increased if \( x_i \) is taken out of the first set, and assigned to a separate processor.

Step 2: Assume the first set contains the first \( j \) largest linear components. Furthermore,

\[ \sum_{i=1}^{j} x_i < 2c. \] (8)

Then the completion time for the entire clan is determined by \( t_{j+1} + 2c \), i.e., the \((j + 1)\)st component, on a separate processor. Using (8), we have

\[ \sum_{i=1}^{j+1} x_i < x_{j+1} + 2c \] (9)

This last inequality implies that adding \( x_{j+1} \) will not increase the completion time of the entire independent clan.

Lemma 1 indicates the following: to form the first set \( X_1 \) of an optimal partition for an independent clan in Figure 4, one can repeatedly collect components into \( X_1 \), until the total completion time of \( X_1 \) exceeds \( 2c \). At that point, \( X_1 \) is assigned to the parent processor of this independent clan to avoid communication costs. All the remaining components are assigned to separate processors. Interestingly, Lemma 1 verifies intuition, i.e., if \( c \) is zero, assigning all components to separate processors is optimal, on the other hand, if \( c \) approaches the infinity, all components should be grouped into one processor.
3 Analysis of the Model

3.1 Probability Distributions of Completion Time

In the last section, the communication cost $c$ is defined as a constant. We now further assume that the processing time required by any primitive clan (counting all tasks in the clan) follows a geometric distribution with parameter $\tau$ ($0 < \tau \leq 1$). If we use $P(k)$ to denote the probability that the completion time of a primitive clan is equal to $k$ time units, according to the geometric distribution, we have that

$$P(k) = (1 - \tau)^{k-1}\tau \quad k = 1, 2, \ldots \tag{10}$$

$Q_{I/L}(k)$ is the probability for an $I/L$-node to have its completion time no more than $k$.

$$Q_{I/L}(k) = \sum_{i=1}^{k} P_{I/L}(i) \quad k = 1, 2, \ldots \tag{11}$$

where $P_{I/L}(k)$ is the probability of completion time being $k$ units for an I-node or an L-node. $I/L$ means either I or L.

For an L-node, $P_L(k)$ can be expressed by:

$$P_L(k) = b_1P(k) + \sum_{n=2}^{\infty} b_n \sum_{\kappa} P_I(k_1)P_I(k_2) \ldots P_I(k_n) \tag{12}$$

where $\kappa = (k_1, k_2, \ldots, k_n)$ is an integer vector, satisfying $k_1, k_2, \ldots, k_n \geq 1$ and $k_1 + k_2 + \ldots + k_n = k$.

Equation (12) is derived using the Law of Total Probability and a convolution operation for summation of $n$ random variables (the second term in (12)). The convolution is used because the completion time for a linear clan is the sum of the completion times for the $n$ independent clans.

For an I-node, the first step, using the Law of Total Probability, is immediate:

$$P_I(k) = a_1P(k) + \sum_{n=2}^{\infty} a_n R_n(k) \tag{13}$$

where $R_n(k)$ is the probability that "an independent clan with $n$ linear components has its completion time equal to $k". We calculate $R_n(k)$ using Lemma 1.

According to Lemma 1, the sum of completion times for the first $J$ largest linear components are used to determine the partition. Thus, $R_n(k)$ depends heavily on the random index $J$ which satisfies (5). To capture $J$, we define $R_n(k, j, s_1, x)$ to have the same meaning with $R_n(k)$ except that there are two more restrictions.

i) Among the first $j$ largest components, $j'$ of them are assigned to the first set $X_1$ in the optimal partition. $R_n(k, j, s_1, x)$ is not concerned with the exact value for $j'$; however, it requires that
the total completion time of the \( j' \) components is \( s_1 \). (For this reason, we use the subscript 1 in \( s_1 \)).

ii) The next largest completion time (i.e., the \((j + 1)\)-st) is not larger than \( x \).

It is easy to see that

\[
R_n(k) = R_n(k, 0, 0, k) \tag{14}
\]

Therefore, the problem is to compute \( R_n(k, j, s_1, x) \) for \( n \geq j, k \geq s_1 \). It turns out that \( R_n(k, j, s_1, x) \) can be recursively computed. If \( j = n \), then it is easy to obtain:

\[
R_n(k, n, s_1, x) = \begin{cases} 
1 & \text{if } k = s_1 \\
0 & \text{otherwise}
\end{cases} \tag{15}
\]

In the following, we organize the presentation by further distinguishing several cases, all under \( j < n \).

**Case 1: \( s_1 > k \).** Clearly,

\[
R_n(k, j, s_1, x) = 0 \quad \text{if } s_1 > k \tag{16}
\]

since it is not possible for the independent clan to complete with time \( k \).

**Case 2: \( s_1 = k \).** Now the first \( j' \) (\( j' \leq j \)) components being assigned to the first set \( X_1 \), have their completion times sum up to \( k \). If the independent clan is to complete with time \( k \), this sum \( s_1 \) (and therefore \( k \) also, since \( s_1 = k \)) must not be less than \( 2c \). Otherwise, (5) in Lemma 1 requires more members to be added into the first set \( X_1 \), in order to achieve an optimal partition. Obviously, adding more members into \( X_1 \) results in a completion time larger than \( k \). Therefore,

\[
R_n(k, j, s_1, x) = 0 \quad \text{if } s_1 < 2c \tag{17}
\]

On the other hand, if \( s_1 \geq 2c \), \( s_1 \) must be the completion time of \( X_1 \) in an optimal partition. According to Lemma 1, all other linear components are assigned to separate processors, and their completion times must not be larger than \( k - 2c \).

Hence,

\[
R_n(k, j, s_1, x) = \begin{cases} 
Q_L^{n-j}(k - 2c) & \text{if } s_1 = k, x > k - 2c \\
Q_L^{n-j}(x) & \text{if } s_1 = k, x \leq k - 2c \\
0 & \text{if } s_1 = k < 2c
\end{cases} \tag{18}
\]

Recall that \( Q_L(k - 2c) \), equation (9), is the probability for a linear clan to have its completion time no more than \((k - 2c)\). The \((n - j)\)-th power in (18) takes into account the fact that there are \((n - j)\) remaining linear components. All of them should have their completion times no more than the smaller value of \((k - 2c)\) and \(x\).
Case 3: $s_1 < k$. Similar with Case 2, here we also need to further distinguish two sub-cases, depending on whether $s_1 < 2c$ or $s_1 \geq 2c$.

Case 3.1: $s_1 \geq 2c$. In this case, similar with Case 2 above, $s_1$ must be the total completion time of $X_1$. Due to the fact that $s_1 < k$, the completion time $k$ for the independent clan must be attributed to one of its linear components not in the first set $X_1$. That clan has a completion time equal to $k - 2c$. Moreover, all those linear clans not in $X_1$ must have completion times not greater than $(k - 2c)$. Therefore,

$$R_n(k, j, s_1, x) = Q_L^{n-j}(k - 2c) - Q_L^{n-j}(k - 2c - 1)$$

if $2c \leq s_1 < k \quad (19)$

In (19), the right-hand side of the equation is the probability that all remaining $(n - j)$ linear clans have completion times less than or equal to $(k - 2c)$, and at least one of them is equal to $(k - 2c)$.

Case 3.2: $s_1 < 2c$. In this case, the first set $X_1$ in the optimal partition includes the $j$ largest components. These $j$ components have their completion times sum up to $s_1$. In addition, the $(j+1)$-st largest component should also be included into $X_1$. Denote the $(j+1)$-st largest completion time of the linear components by $x'$. From the definition of $R_n(k, j, s_1, x)$, $x'$ can only be selected from $\{1, 2, \ldots, x\}$. However, there can be as many as $m$ components $(1 \leq m \leq n - j)$ to share the next largest completion time $x'$. Thus,

$$R_n(k, j, s_1, x) = \sum_{m=1}^{n-j} \binom{n-j}{m} \sum_{x'=1}^{x} P_L^m(x') R_n(k, j + m, s_1 + m'x', x' - 1)$$

if $s_1 < k, s_2 < 2c \quad (20)$

where $m'$ is simply $m$, if $s_1 + mx' < 2c$. When $s_1 + mx' \geq 2c$, $m'$ is the integer that satisfies

$$1 \leq m' \leq m; s_1 + m'x' \geq 2c; s_1 + (m' - 1)x' < 2c \quad (21)$$

In other words, provided that there are $m$ components having completion times all equal to $x'$, we would want to add the fewest of these components into the first set $X_1$, to make its total completion time exceed $2c - 1$. $m'$ is this minimum number.

Up to this point, we have developed all formulas necessary for computing $P_{I/L}(k)$ from equations (10) and (11).

3.2 Conditional Expected Completion Times and Their Standard Deviations

Once the probability distributions $P_{I/L}(k)$ are obtained, we can easily compute the expected completion time $T_{I/L}(k)$ for an I/L-node given that its completion time is no more than $k$: 
\[ T_{I/L}(k) = \frac{1}{Q_{I/L}(k)} \sum_{i=1}^{k} i P_{I/L}(i) \] (22)

The standard deviation of completion time for an I/L-node, given that its completion time is no more than \( k \) is:
\[ D_{I/L}(k) = \sqrt{\frac{1}{Q_{I/L}(k)} \sum_{i=1}^{k} i^2 P_{I/L}(i) - T_{I/L}^2(k)} \] (23)

### 3.3 Conditional Expected Number of Primitive Nodes in a Random Graph

We use \( n_{I/L} \) to denote the number of primitive nodes in a (random) I/L-node. Then we define \( M_{I/L}(k) \) by
\[ M_{I/L}(k) = E(n_{I/L} | \text{completion time is } k) \] (24)

where \( E(\cdot) \) means the expectation of the random variable \( \cdot \). \( 1_A \) is a (random) boolean flag which gives 1 when event \( A \) occurs, 0 otherwise. Thus, \( M_{I/L}(k) \) is the expectation of the total number of primitive nodes in an I/L-graph, which completes in \( k \) time units.

\( M_L(k) \) satisfies
\[ M_L(k) = b_1 P(k) + \sum_{n=2}^{\infty} b_n \sum_{\forall K} P_L(k_1)P_L(k_2)\ldots P_L(k_n) \sum_{i=1}^{n} M_I(k_i) \] (25)

where \( K \) has the same meaning as in (12). (25) clearly indicates that \( M_{I/L}(k) \) is simply a sum of products of probabilities with the corresponding number of primitive nodes in the graph. Thus, the computation of \( M_{I/L} \) follows a similar approach with which we computed the probability distribution \( P_{I/L}(k) \) in Section 3.1. Then the expected total number of primitive nodes generated by an I/L-node, given that the completion time for the I/L-node is no more than \( k \), can be expressed by:
\[ N_{I/L}(k) = \frac{1}{Q_{I/L}(k)} \sum_{i=1}^{k} M_{I/L}(i) \] (26)

Since the derivations for \( M_{I/L} \) are similar to in section 3.1, we omit the details here.

We define by \( R_n'(k) \) the expectation of the total number of primitive nodes in an \( I \)-graph having completion time \( k \), given that the graph has \( n \) immediate linear components. Note that \( R_n'(k) \) is similar to \( R_n(k) \) as in Section 3.1.
\[ M_I(k) = a_1 P(k) + \sum_{n=2}^{\infty} a_n R_n'(k) \] (27)

Next we define \( R_n'(k, j, s_1, x, z) \) to have the same meaning as \( R_n'(k) \) except for the following three more restrictions:
i) The completion times for the first $j'$ ($j' \leq j$) components is $s_1$, where $j'$ satisfies the same conditions for $j'$ as in Section 3.1.

ii) The next largest completion time should not be larger than $x$.

iii) The total number of primitive nodes in the first $j$ components is $z$.

Notice that $R_n'(k, j, s_1, x, z)$ has one more argument $z$ compared to $R_n(k, j, s_1, x)$ in Section 3.1.

\[ R_n'(k) = R_n'(k, 0, 0, k, 0) \]  \hspace{1cm} (28)

For $j = n$, it is easy to see that

\[ R_n'(k, n, s_1, x, z) = \begin{cases} 
z & \text{if } k = s_1 \\
0 & \text{otherwise} 
\end{cases} \]  \hspace{1cm} (29)

Using similar arguments as in derivation of $R_n$ in section 3.1 we derive following formulas for $R_n'(k, j, s_1, x, z)$, under $j < n$:

\[ R_n'(k, j, s_1, x, z) = 0 \]  \hspace{1cm} \text{if } s_1 > k \hspace{1cm} (30)

\[ R_n'(k, j, s_1, x, z) = \begin{cases} 
Q_L^{n-j}(k-2c)[x + (n-j)N_L(k-2c)] & \text{if } s_1 = k \geq 2c, x > k-2c \\
Q_L^{n-j}(x)[x + (n-j)N_L(x)] & \text{if } s_1 = k \geq 2c, x \leq k-2c \\
0 & \text{if } s_1 = k < 2c 
\end{cases} \]  \hspace{1cm} (31)

\[ R_n'(k, j, s_1, x, z) = Q_L^{n-j}(k-2c)[x + (n-j)N_L(n-2c)] - Q_L^{n-j}(k-2c-1)[x + (n-j)N_L(n-2c-1)] \]  \hspace{1cm} \text{if } 2c \leq s_1 < k \hspace{1cm} (32)

\[ R_n'(k, j, s_1, x, z) = \sum_{m=1}^{n-j} \binom{n-j}{m} \sum_{x'=1}^{x} P_L^m(x')R_n'(k, j + m, s_1 + m'x', x'-1, z + mM_L(x')) \]  \hspace{1cm} \text{if } s_1 < k, s_1 < 2c \hspace{1cm} (33)

where $m'$ satisfies the condition (21) in Section 3.1.

3.4 Conditional Speedup

The speedup $S_{I/L}(k)$ for an I/L-node given that its completion time is no more than $k$ is:

\[ S_{I/L}(k) = \frac{1}{Q_{I/L}(k)} \sum_{i=1}^{k} \frac{M_{I/L}(k)}{k\tau} \]  \hspace{1cm} (34)

Note that $1/\tau$ is the expected computation time of a primitive node, and thus $M_{I/L}(k)/\tau$ gives the expected total computation time for those clans with completion time $k$, if executed on a parallel architecture.
3.5 Unconditional Performance Measures

If $k$ approaches infinity in equation (22), (23), and (34), we obtain the unconditional expected completion time, standard deviation, and speedup, respectively. The values for these unconditional performance measures may be infinite, depending upon if the random graph is unstable.

Definition 7. (Stable/Unstable Random Graph) We call a random graph stable if the expected number of primitive nodes in the graph is finite. Otherwise, the graph is unstable.

The following result can be used to determine if a random graph is stable.

Lemma 2: A random graph is stable iff

$$\left(\sum_{n=2}^{\infty} na_n\right) \left(\sum_{n=2}^{\infty} nb_n\right) < 1$$

(35)

Proof: Denote by $N_{I/L}$ the expected number of primitive nodes generated by an $I/L$-node. Thus,

$$N_I = a_1 + \sum_{n=2}^{\infty} a_n (nN_L)$$

(36)

$$N_L = b_1 + \sum_{n=2}^{\infty} b_n (nN_I)$$

(37)

Hence,

$$N_I = \frac{a_1 + b_1 (\sum_{n=2}^{\infty} na_n)}{1 - (\sum_{n=2}^{\infty} na_n) (\sum_{n=2}^{\infty} nb_n)}$$

(38)

$$N_L = \frac{b_1 + a_1 (\sum_{n=2}^{\infty} nb_n)}{1 - (\sum_{n=2}^{\infty} na_n) (\sum_{n=2}^{\infty} nb_n)}$$

(39)

It is then clear that (35) must hold in order to have finite $N_{I/L}$.

\[\square\]

4 Numerical Results

In this section, we present representative numerical results. Some of them are obtained from both analytical formulas and from extensive simulations. The experimental results are collected through generating well over 10,000 random graphs to guarantee a degree of confidence at least 95%. The simulation was conducted mainly for validating the correctness of the analysis. It is worth noting that for the simulator to terminate with probability 1, the probability vectors $a = (a_1, a_2, \ldots)$ and $b = (b_1, b_2, \ldots)$ must satisfy the stability condition in (35).

17
It is not difficult to see that when we deal with real applications, the stability condition is often violated. For those unstable cases, performance evaluation through simulation becomes not applicable. However, the analytical formulas derived in the last section can be used no matter if the system is stable. For this reason, to obtain analytical procedures to predict the performance of the scheduling algorithm, as is done in the last section, is both necessary and very important.

We point out that using the analytical formulas, the performance of the task scheduling algorithm may be examined in many ways through numerical results. Due to space limit, in this section, we will only focus our attention on

i) examining the impact of the logical restriction on parallelism.

ii) examining the impact of communication cost on the performance of the algorithm.

We explained in the beginning of Section 2 that those are two out of three major factors that may affect efficient execution of a sequential program on a parallel architecture.

4.1 Validation of the Analysis

In this subsection, we assume

\[ a = b = (0.55, 0.4, 0.025, 0.025) \]

and

\[ c = 4, \tau = 0.05 \text{ or } 0.5 \]

Under these assumptions, the stability condition in (35) is satisfied, i.e., the task graph is stable. Therefore the simulation method can be used. In this example, the vector \( a \) places a strong restriction on parallelism, i.e., branching out three or more parallel components is rare. The speedup is therefore very low.

In Figure 5 and 6, we show \( T_I(k) \), the conditional expected completion time, and \( S_I(k) \), the conditional expected speedup, respectively, both for an \( I \)-node. These figures, as well as many other results not shown here, clearly demonstrate that the analysis presented in the last section is correct.

In addition, these figures show increasing completion time and speedup, when \( k \) increases. Here we point out that \( k \) can be roughly understood as a parameter to reflect the maximum "size" of a program. With this understanding, Figures 5 and 6 show that completion time and speedup increase almost linearly with the maximum size of a program. This is a rather desirable performance, but unfortunately, it is only valid for stable precedence relations. For the unstable cases, we will show shortly that this conclusion does not hold.
Figure 5: Conditional Expected Completion Time of an I-Node

Figure 6: Conditional Speedup for an I-Node
In Figure 7, we plot the probability distribution of completion time for $I/L$-node, computed from analysis. With this figure, we show the difference between the performance of the algorithm on $I$-node and $L$-node. Because we have selected $a = b$, the difference, if any, must be attributed to the behavior of the algorithm.

In general, as expected, there is little difference between an $I$-node and an $L$-node on their probability distributions of completion time, since $a = b$. One can observe that when $k < 2c = 8$, both types of nodes possess the same completion time distributions. This is expected because the algorithm uses $2c$ as the criterion to determine the content of the first set $X_1$. So, when $k < 2c$, it clusters all the components generated by an $I$-node into one processor, thus there is no difference between an $I$-node and an $L$-node. Starting from 8, the effect of the algorithm on an $I$-node can be seen. Due to an optimal task assignment by the algorithm, the completion time for an $I$-node tends to be shorter than if it were an $L$-node. It is therefore expected that the mass of probability for an $I$ node is shifted towards the left of the figure.

Finally, in Figure 7, we plot the standard deviation for completion time. In general, we can observe that this deviation increases with $k$. We can also observe another (although small) effect of an $I$-node: its standard deviation on completion time tends to be smaller than an $L$-node. In addition, larger $1/\tau$ (corresponding to programs with larger average primitive modules) tend to incur higher standard deviation on completion time. This fact may be understood by the result for geometric
distribution: larger $1/\tau$ (smaller $\tau$) gives rise to larger variance $\sqrt{1 - \tau/\tau}$. In summary, Figure 7 indicates that the maximum size of the program, the nature of the program, (i.e., if it corresponds to an I node or to an L-node), and the maximum program size may all affect the accuracy of the expected performance values.

4.2 The Impact of Logical Restriction on Parallelism

In this subsection, we fix

$$b = (0.5, 0.5), c = 1, \text{ and } \tau = 0.1$$

and we examine three cases where $a$ takes one of the following three vector values.

$$\bar{a}_4 = (0, 0, 0, 1)$$

$$\bar{a}_8 = (0, \ldots, 0, 1)$$

$$\bar{a}_{16} = (0, \ldots, 0, 1)$$

Therefore, $\bar{a}_4, \bar{a}_8, \text{ and } \bar{a}_{16}$ correspond to three similar cases where program parallelism becomes increasingly higher. We would like to see how speed up and completion time varies with the changing degree of parallelism.
Figure 9: Conditional Speedup for an I-Node

Note that the sequential component of the model (i.e., the vector $b$) gives even chances to the two possibilities: to terminate the graph or to further expand it. We have also conducted examinations by fixing $a$ and allow $b$ to take one of the following values.

$$\bar{b}_4 = (0, 0, 0, 1)$$

$$\bar{b}_8 = (0, \ldots, 0, 1)$$

$$\bar{b}_{16} = (0, \ldots, 0, 1)$$

The conclusion is similar with the one to be summarized in this subsection, except that $b_4$, $b_8$, $b_{16}$ appear to have the effect of increasingly lowering the degree of parallelism. Due to space limit, we will not show the results obtained through varying the vector $b$.

In Figures 9 and 10, we plot the conditional speedup and completion time, respectively, for an $I$-node. The same performance measures are plotted in Figures 11 and 12, for an $L$-node.

Different from the observation from the last subsection, Figure 9 indicates that speedup does not increase linearly with $k$. For all three cases, speedup increases with $k$ up to certain point. We call this turning point critical point. The lesson learned from this Figure is clear: it is beneficial to estimate the maximum completion time for a program before it is run on the parallel architecture. If it is certain that the maximum completion time of the program would pass the critical point too
Figure 10: Conditional Completion Time for an I-Node

Figure 11: Conditional Speedup for an L-Node
far, it is necessary to either use another (more appropriate) scheduling algorithm, or simply run the program sequentially.

Figure 10, shows that increasing the degree of parallelism also increases the expected completion time. The reason behind this phenomenon is that higher degree of parallelism leads to more nodes to be generated, the expected completion time may thus be higher.

Finally, Figures 11 and 12 indicate that it is definitely not beneficial to run an L-node on the parallel architecture with the example in this subsection.

4.3 The Impact of Communication Cost

In this subsection, we assume

\[ \overline{a_\gamma} = (0, \ldots, 0, 1), b = (0.5, 0.5) \]

and examine two classes of examples. In the first class, the expected completion time of a primitive clan is larger than the communication time c. We call this class of cases large grain situations. Another class of cases considered, called fine grain situations, is when the communication time c is larger or comparable to the expected completion time of a primitive clan.
Figure 13: Conditional Speedup for an I-Node

In Figure 13, we plot speedup of an I-node for the following parameters:

\[ c = 1, \tau \in \{0.1, 0.01, 0.001\} \]

The expected lengths (1/\( \tau \)) for a primitive clan for the three cases can be computed to be 10, 100, and 1000, respectively. These three cases all belong to large grain situations. Intuitively, when \( \tau \) approaches 0, speedup should approach the one for \( c = 0 \). Therefore, in addition to the above three cases, we also examine another case where

\[ c = 0, \tau = 0.1 \]

In Figure 13 we observe non-linear behaviors of speedup with \( k \). A more important fact shown in Figure 13 is that the case when the communication is zero, cannot be taken as a limiting situation for \( \tau \to 0 \). The implication of this fact is important: when analyzing the task scheduling algorithm, one may not overlook the communication cost, no matter how small it may be. The algorithm analyzed in this herein, provides a good example to demonstrate a possible loophole in performance analysis.

Finally, in Figure 14, we plot conditional speedup for the fine grain case. It is especially clear from this figure that when communication cost exists, fine grain task graph leads to poor system performance.
5 Conclusions and Future Research

To capture the restrictions imposed by both program logic and by hardware cost, we have considered two important concepts in the model: a probabilistic task precedence relation generator, and a task scheduling algorithm. At every step, the parallel execution of a probabilistic task precedence graph is controlled by the scheduling algorithm, with the goal being to achieve the best performance from parallelism. In our analysis, we assumed constant communication costs between any pair of processors and an unlimited number of processors, and under these conditions we determined the effect of the computation-communication trade-off in a multiprocessor environment. With a selected task scheduling algorithm, we derived formulas for computing the expected completion time of a random diagram on a parallel multiprocessor, its standard deviation, the speedup of the task scheduling algorithm, and the probability distribution of the completion time. The results are validated by extensive simulations. Well over 10,000 random DAGs were generated for each example.

Analysis of random task precedence graphs, although discussed in literature prior to this work, has never considered the scheduling algorithms. Task scheduling algorithms were analyzed using deterministic task graphs. Our work, thus, represents the first attempt to capture both the task precedence graphs and the scheduling algorithm in one analytical model. We demonstrated that for those task scheduling algorithms which do not have NP-complete problems, performance analysis may be possible, even with the consideration of a random task generator.
Among the many results shown in this paper, the following are the most important.

- We showed that the evaluation of the model is not always amenable to simulation methods for all possible cases. In fact, simulation cannot be used at all for most practical situations.

- We demonstrated that, in general, speedup does not increase linearly with the maximum program size. Given a class of programs (i.e., those sharing the same \(a, b,\) and \(r\) parameters), there is an optimal value for program size which results in maximum speedup.

- We show that neglecting communication cost, no matter how small it may be, may lead to incorrect performance results.

There are numerous directions in which this work may be extended. In this paper and also in [11], we assume the probability vectors \(a\) and \(b\) are fixed at all levels. A more general model would consider different values from level to level. A second extension may be to allow a primitive task to take different probability distributions. However, this is heavily dependent upon digesting some empirical data about the primitive tasks in a program. Finally, any other scheduling algorithm could be considered to obtain performance results and their performances could be compared.

References


