Execution Time Analysis of Communicating Tasks in Distributed Systems

Jong Kim, *Member, IEEE*, and Kang G. Shin, *Fellow, IEEE*

Abstract—Task-execution times are one of the most important parameters in scheduling tasks. Most scheduling algorithms are based on the assumption that either worst-case task-execution times are known to the scheduler or no information on execution times is available at all. While scheduling tasks based on worst-case execution times can guarantee to meet their timing requirements, it may lead to severe under-utilization of CPUs because worst-case execution times could be one or two orders of magnitude larger than the corresponding actual values. Scheduling tasks based on the execution time distribution (instead of worst-case execution times) is known to improve system utilization significantly.

In this paper, we propose a model to predict task execution times in a distributed system. The model considers several factors which affect the execution time of each task. These factors are classified into two groups: *intrinsic* and *extrinsic*. The intrinsic factors control the flow within a task, while the extrinsic factors include communication and synchronization delays between tasks. By simplifying the extrinsic factors, we represent a distributed system with a simple queuing model. The proposed queuing model consists of two stations: one for computation and the other for communication and synchronization. Information on system utilization can be obtained by converting this queuing model to a Markov chain. The execution time of a task is then derived from the information on system utilization in the form of average and distribution. The model is extended to describe the effects of multiple tasks assigned to a single processing node. The utility of the model is demonstrated with an example.

Index Terms—Task-execution time, distributed systems, queuing analysis, communication and synchronization delays.

1 Introduction

Most scheduling algorithms known to date are based on the assumption that either worst-case task-execution times are known to the scheduler or information on task execution times is not available at all. However, it is usually difficult to estimate the worst-case execution time of a task and one must be cautious in estimating worst-case execution times. For example, over-estimation of the execution time of a task will prevent the scheduling of other tasks even if they are schedulable, whereas under-estimation will result in unbalanced system utilization or risk missing task deadlines. Hence, it is desirable to estimate task execution times as accurately as possible.

Due to their potential for high performance and fault tolerance, distributed systems are attractive for various applications. High performance is achievable by exploiting the inherent parallelism among tasks with the multiple processors in a distributed system. A task is divided into subtasks, expressing the parallelism within the task. Subtasks are assigned to different processors and communicate and/or synchronize with each other to achieve a common goal. Fault tolerance can be achieved in a distributed system by using its multiplicity of components as natural redundancy. A distributed system can be characterized by two components: tasks and inter-task communications.

The execution time of a task depends on many factors which can be classified into two groups: *intrinsic* and *extrinsic*. The intrinsic factors control the flow within a task. Example intrinsic factors include precedence relations and condition parameters that determine loops and branches within a task. The extrinsic factors, on the other hand, control interactions among tasks. Examples of extrinsic factors include resource contention, communication, and synchronization delays. Although many researchers considered the effects of intrinsic factors on the task-execution time [1], [2], [3], [4], [5], [6], [7], only the authors of [8] addressed extrinsic factors.

The analytic model proposed by Chu and Sit [8] considers the effects of intrinsic factors such as loops, branches, and the precedence relation among subtasks as well as the effects of extrinsic factors such as resource contention. They used a Timed Petri-Net to model resource contention and a queuing model to derive the execution time of a task. The final result was given in the form of the mean and variance of task-execution time. This work offered a means of including an extrinsic factor in the estimation of task-execution time. However, it is difficult to use for large systems since the number of states needed to represent resource contention is $\binom{n+v}{v}$ where n is the number of tasks and v is the number of resources to be contended for. Moreover, this model did not consider the effects of other extrinsic factors such as interprocessor communication, synchronization, and multitasking. Note also that resource contention is rather rare in loosely-coupled systems.

- *J. Kim is with the Department of Computer Science, Pohang University of Science and Technology, Pohang 790-784, Korea. E-mail: jkim@postech.ac.kr.*
- *K.G. Shin is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: kgshin@eecs.umich.edu.*

Manuscript received May 16, 1992; revised Sept. 23, 1994.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96023.

In this paper, we propose a new model by taking into account the effects of both intrinsic and extrinsic factors on task-execution times. A task (algorithm) consists of a number of subtasks in the model, each of which is in turn composed of several executable modules. The intrinsic factors determine the structure of the data dependency graph of the task and affect the execution time of a module and the number of modules in each subtask. The structure of a subtask is represented by the number of modules in it and the service demands of each module. The extrinsic factors, on the other hand, affect the delay between modules. The extrinsic factors considered here are interprocessor communication delays, synchronization delays, and multitasking. (Note that multitasking is another form of resource contention for processors (CPUs).) In the proposed model, we consider two possible subtask assignments to a processing node. The first is *single* assignment under which at most one subtask is assigned to a processing node. The other is *multiple* assignment under which more than one subtask can be assigned to a single node. The model analyzes the average of task-execution time (service time) and its distribution. The resulting service time includes the actual execution time of modules plus intermodule communication and synchronization delays.

The paper is organized as follows: Section 2 describes the operational principles of the distributed system under consideration. The structures of task and subtask, and the communication mechanism are discussed there. Based on these principles, the models of single and multiple assignments are given in Section 3. The distribution of execution times is derived in Section 4. The utility of the model is demonstrated with an example in Section 5. The last section summarizes our contributions.

2 Operational Principles

2.1 Task Structure

A task consists of several subtasks which communicate with one another to achieve a common goal. Since the parallelism supported by distributed systems would be mostly medium-grain to coarse-grain, each subtask has almost the same lifetime as that of the task itself. A subtask is composed of a collection of modules each of which consists of a sequence of instructions. Modules do not communicate nor synchronize with others during their execution, but subtasks communicate/synchronize with each other at the beginning or end of a module's execution. Fig. 1 shows a generic task structure. A subtask may initiate some communication after a module completes its execution. Similarly, a subtask may have to wait for some message(s) before starting the execution of a new module. The actual communication pattern depends on the data–dependency between modules and their implementation on a specific architecture.

There are two possible ways of assigning subtasks to processing nodes. The first is a multitasking or multiprogramming (MP) environment, in which more than one subtask can be assigned to a single processing node. Although multitasking is efficient in utilizing resources, it could delay the execution of a task because of resource

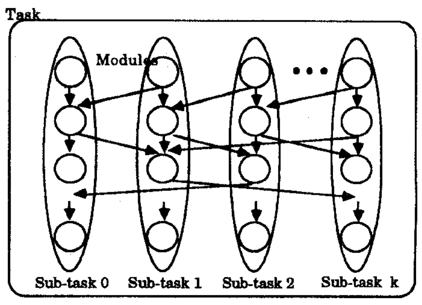


Fig. 1. A generic model for parallel execution of a task.

(CPU) contention. The second is the nonmultitasking (NMP) environment where at most one subtask is assigned to each processing node. We will analyze both MP and NMP cases.

2.2 Communication Structure

There are two types of communication primitives in parallel computation [9]: *blocking* and *nonblocking*. Upon issuing a blocking primitive, a subtask must block and wait until it receives an acknowledgment or requested information from its communicating partner. An example of a blocking communication primitive is **QUERY–READ**. A subtask using this primitive sends a message to another subtask for information. The sender subtask cannot proceed further until the receiver subtask replies back to the sender. The communication primitives used by the receiver are **READ** and **REPLY**. The receiver gets a message with **READ**, prepares the requested information, and returns it to the sender with **REPLY**. The replying subtask does not have to wait until the reply reaches the sender, i.e., it is nonblocking. Another example of this type of primitive is nonblocking **SEND**. As soon as a subtask sends a message to another subtask, the sender can resume its execution without waiting for a reply. When a subtask reaches the point where it needs information from the other task, the subtask issues **RECEIVE**. The subtask which needs information must wait until it gets the needed information. Hence, **RECEIVE** is always blocking. In this paper, we shall treat both types of communication primitives under the assumption that the probability distribution associated with the use of these primitives in each task are known in advance. This assumption is not unreasonable in view of the fact that the behaviors of tasks are tested and simulated extensively before putting them into actual use.

The communication delay depends on the system configuration, and the distributions of destination and message length. It is difficult to estimate the communication delay without considering the interconnection network and the routing algorithm used. The result for one environment could be drastically different from that for another environment. For example, circuit and packet switching methods will yield quite different communication delays even under the same system load. One cannot apply the result of circuit switching to the analysis of the system using packet switching. Similar disparities can be found in the synchronization delay. Two communicating subtasks will be synchronized at the point of each blocking communication.

Due to the difference in their service demands and execution speed, two subtasks will reach a communicating point at different times. Well-designed parallel algorithms will reduce the synchronization delay by distributing an equal amount of computation to all communicating subtasks. We assume that the synchronization delay is an independent factor affecting the task execution time.

3 Structure of the Proposed Model

When a task (algorithm) is represented by a task graph, the subtask that terminates last (takes maximum time) is called the *critical subtask.* So, the execution time of the critical subtask for a given task becomes the actual task-execution time—the time needed to complete the task after assigning all its subtasks to processing nodes. (We do not consider migration of subtasks during their execution.) It is assumed that the critical subtask consists of M modules and the execution time of each module is exponentially distributed with an average demand S, measured in basic time units or CPU cycles. (M is a random variable determined by the executing environmental factors. In this section, we assume that this is a fixed value for now but we will generalize it in Section 5.) The total service demands of the critical subtask in the absence of communication and synchronization delays is $M \cdot S$.

During the execution of a task, it will be in one of two states: *active* and *semi-active*. The *active* state represents the case when a processing node is busy executing a module. Since a module is defined as a sequence of instructions and does not communicate with others during its execution[1], only the active–state subtask requires services from the processing node. When the execution of a module is completed, either the subtask to which this module belongs may send a message to other subtask(s), or it waits for message(s) from others, or both. The waiting state is called the *semi-active* state.

Fig. 2a shows the model for a processing node in a non-multitasking (NMP) environment, where only one subtask can reside on each processor. Upon completion of module execution in the *active* state, the corresponding subtask may join the *active* state again with probability xP_f. This transition—shown in Fig. 2a as an arrow to the *active state* from the outgoing message path—represents the situation in which a subtask sends message(s) to other subtask(s) after completing the execution of a module and the node starts the execution of a new module. In such a case, the subtask does not wait for messages from other nodes. The non-blocking **SEND** achieves this type of transition.

In case a subtask needs data from other subtasks after making such requests, the branching probability becomes $P_f\,(1 - x)$ as shown in the figure with an arrow to the *semi-active state* from the outgoing message path. The blocking primitive **QUERY–READ** or **SEND–RECEIVE** achieves this type of transition.

When a subtask needs data from other subtasks, it takes the incoming message path and waits in the *semi-active* state with the transition probability $(1 - P_f)$. The blocking primitive **RECEIVE** achieves this type of transition. Branching

[1]. If this does not hold, it is not difficult to re-decompose a subtask so as to satisfy this condition.

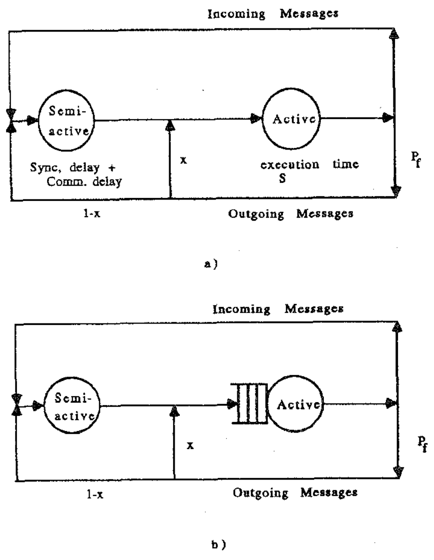


Fig. 2. Task execution model.

probabilities, P_f and x, are assumed to be given as input parameters to the model. These parameters can be determined from the nature of the parallel algorithm under consideration.

In a multitasking (MP) environment, there is a queue at each processing node to hold the ready tasks waiting for service. Each of the ready tasks will receive services based on the first-come-first-served (FCFS) principle. Note, however, that there is no queue for subtasks waiting for messages. We assumed that there are infinite servers for subtasks in the *semi-active* state, since subtasks are moved to the processing node as soon as they receive messages regardless of their arrival time at the *semi-active* state. This situation is shown in Fig. 2b.

3.1 NMP Model

The module execution rate in the *active* state is given by $\lambda_e = \frac{1}{S}$. Since the transition from the *active* state to the *active* state implies the generation of messages which are sent to other tasks, the message generation rate of a task is proportional to the service demand of the task as:

$$\lambda_g = P_f \cdot \frac{1}{S} \cdot U, \tag{3.1}$$

where U is the node utilization. This message generation rate information can be used to determine the communication delay. The transition from the *semi-active* state to the *active* state depends on both synchronization and communication delays. Since modules may have different starting times and service demands, two communicating modules may have different finishing times. This difference is in fact the synchronization delay. It is in general very difficult to analyze the synchronization delay [10] and the detailed analysis of it is beyond the scope of this paper (such an analysis deserves to be a separate paper). Instead of developing an analytic method, we measure this parameter from the corresponding task in a real/simulated environment. For simplicity, let μ_s be the rate of synchronization delay which is independent and exponentially distributed. Similarly, we assume that the average delay of a message is given and the communication delay is exponentially distributed. The summation of both synchronization and communication delays gives the time spent by a task in the *semi-active* state.

3.2 MP Model

The difference between MP and NMP models lies in the queue placed at each processing node. Upon execution of one of its modules, a subtask either sends a message or receives a message or both. After sending a message, the subtask joins the queue with probability of xP_f. The node processor selects the next subtask from the queue based on the FCFS principle. A subtask which waits for an incoming message(s) from another subtask(s) moves to the semi-active state with probability of $(1 - xP_f)$. The semi-active state behaves as an infinite server since a subtask task moves immediately from the *semi-active* state to processor queue upon receiving the message it is waiting for.

4 Model Analyses

4.1 NMP Model

We present the combined communication and synchronization delay, which is hypo-exponential in nature, as two stages of exponential distribution functions. The first stage represents the synchronization delay (exponentially distributed with rate μ_s). The second stage represents the communication delay (exponentially distributed with rate μ_{avg}). Fig. 3 shows the Markov chain for the NMP model. The arrow that originates from the *Active* state and returns to the same state, with transition rate $xP_f\lambda_e$, represents non-blocking outgoing messages. Existence of the arrow does not affect the Markov chain of Fig. 3 [11]. The semi-1 and semi-2 jointly specify the probability of *semi-active* state. Solving this three-state Markov chain, we get

$$P(act) = \frac{\mu_s \cdot \mu_{avg}}{\mu_s \cdot \mu_{avg} + \lambda_e(1 - xP_f)(\mu_s + \mu_{avg})}$$
$$P(semi) = \frac{\lambda_e(1 - P_f \cdot x)(\mu_s + \mu_{avg})}{\mu_s \cdot \mu_{avg} + \lambda_e(1 - xP_f)(\mu_s + \mu_{avg})}. \tag{4.1}$$

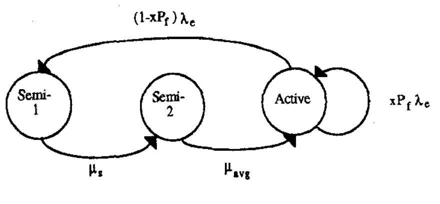


Fig. 3. A Markovian model for the NMP environment.

The active state represents that a node is busy executing a module and hence gives information on node utilization. Recall that the execution time of the critical subtask was assumed to be $M \cdot S$ if there is no delay between the execution of modules. Since the utilization of a node is calculated as *P(act)*, the exact expression for the mean execution time of the critical subtask is given as

$$t_{comp} = \frac{(M \cdot S)}{P(act)}. \tag{4.2}$$

4.2 MP Model

Fig. 4 is the Markov state diagram when the degree of multiprogramming is K. The two-stage delay is represented as states semi-1 and semi-2 in Fig. 3, and as a single (semi-active) state in Fig. 4 for clarity. The transition rate to an active state is also given as μ. But, we use two semi-active states as shown in Fig. 3 when solving the model. The K active (Active–1 to Active–K) states in Fig. 4 represent the number of subtasks at the processor queue. Whenever a subtask moves from active–1 state to the semi-active state with rate $(1 - xP_f)\lambda_e$, the number of subtasks at the processor is decremented by one. On the other hand, the transition from the semi-active state to active–1 state increments the number of subtasks at the processor by one. This rate is shown as μ for all states. The transition rate from active–i state, $1 \le i \le K$, to itself is given by $(x\lambda_e P_f)$. Note that the semi-active state is equivalent to active–0 state representing the case of no subtask at the processor queue.

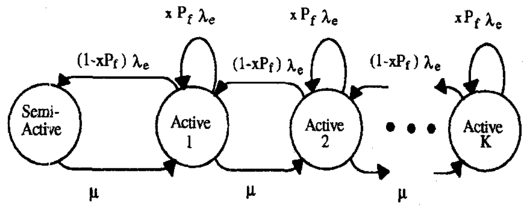


Fig. 4. A Markov model for the MP environment.

The model presented in Fig. 4 is the same as the classic $M/M/1/K$ finite storage queueing model. But, as discussed in the previous paragraph, each state represents two implied sub-states, semi-1 and semi-2. When the Markov chain of Fig. 4 is fully presented, it is extremely difficult, if not impossible, to get closed–form expressions for the above state probabilities. But this Markov chain can be solved by using a numerical package like HARP [12]. The utilization of a node is then

$$U = \sum_{i=1}^{K} P(\text{active } i).$$

The average percentage of the time that subtask i occupies the processor is U/K. Hence, the exact expression for the mean task execution time is given as

$$t_{comp} = (M \cdot S) \cdot K \Big/ \sum_{i=1}^{K} P(active\ i). \tag{4.3}$$

5 Distributions of Delays and Execution Times

5.1 Delay Distributions

In this section, we will analyze the distribution of the total communication and synchronization delay. The communication delay depends on message length, network structure, and traffic density. Let C and H be the random variables that represent the communication and synchronization delays, respectively, and are both exponentially distributed (for tractability). Recall that the average communication and synchronization delays are $1/\mu_{avg}$ and $1/\mu_s$, respectively. The combined communication and synchronization delay is hypo-exponentially distributed since it is the sum of two exponentially distributed variables [11]. Let $\Pr[C \le t]$ be the probability that a message is delivered to the destination in time t, then this distribution is given as

$$\Pr[C \le t] = 1 - exp(-t\mu_{avg}). \tag{5.1}$$

Likewise, the distribution of H is given as

$$\Pr[H \le t] = 1 - exp(-t\mu_s). \tag{5.2}$$

We need a probability density function to compute the distribution of total delay. Since the probability density function of total delay is the convolution of density functions of all random variables, we must find the individual density functions first. The derivatives of distribution functions of C and H are

$$\begin{aligned} f_C &= \Pr[C = t] = \mu_{avg} \cdot exp(-t\mu_{avg}) \\ f_H &= \Pr[H = t] = \mu_s \cdot exp(-t\mu_s). \end{aligned} \tag{5.3}$$

Taking the Laplace transforms of the above equations, we get

$$\begin{aligned} \mathcal{L}(f_C) &= \mu_{avg} \cdot \frac{1}{s + \mu_{avg}} \\ \mathcal{L}(f_H) &= \mu_s \cdot \frac{1}{s + \mu_s}. \end{aligned} \tag{5.4}$$

The Laplace transform of probability density function for the total delay is the direct multiplication of individually transformed functions. Let T be the random variable representing the total delay, then

$$\begin{aligned} \mathcal{L}(f_T) &= \mathcal{L}(f_C) \cdot \mathcal{L}(f_H) \\ &= \frac{\mu_s}{s + \mu_s} \cdot \frac{\mu_{avg}}{s + \mu_{avg}}. \end{aligned}$$

The probability density function f_T can then be found by taking the inverse Laplace transform as

$$f_T = \frac{\mu_s \mu_{avg}}{\mu_{avg} - \mu_s}\left[exp(-t\mu_s) - exp(-t\mu_{avg})\right]. \tag{5.5}$$

By integrating f_T, one can find the total delay to be hypo-exponentially distributed.

We have analyzed thus far both the communication and synchronization delays when each of these delays is exponentially distributed. This analysis, however, can be applied to any distribution of communication and synchronization delays as long as their Laplace transforms exist.

5.2 Execution Time Distribution

We will derive the distribution of task-execution time under the following assumptions.

A1. The probability distribution of the number of modules is known and represented as $\Pr[M = m]$ where M is a random variable representing the number of modules in the critical subtask.

A2. The execution time of each module is independent and identically distributed.

A1 does not impose any problem in practice, since programs are usually tested and simulated extensively before putting them in use. Moreover, an algorithm (program) would be inefficient if environmental factors are not considered during its design stage. Since the communication between subtasks depends on the environmental factors, the number of modules also depends on these factors. Environmental factors are usually known during the planning period and thus **A1** is not unreasonable. **A2** is needed for tractability and can be justified as follows. The execution time of each module is a function of its environment. For example, let us consider conditional branches and loops inside of a module, which are determined by the module's environmental conditions. These conditions are in turn determined by the module's input and working data. Some of the working data are exchanged among subtasks, and subsequently affect module-execution times. This situation implies that modules have no strict 'logical' relation among them. On the other hand, a task is partitioned into subtasks and modules either by a programmer or by a compiler [13]. To increase execution efficiency (by reducing the synchronization delay), an equal load distribution to modules is preferred, thus making module-execution times identically distributed.

As defined in Section 3, let S be the random variable representing the execution time of a module. When S is exponentially distributed with rate λ_e, the probability density function of module-execution time becomes:

$$f_S = \lambda_e \cdot exp(-t\lambda_e). \tag{5.6}$$

If the critical subtask is composed of m modules, then the probability density function (*pdf*) of its execution time without considering the inter-module delays is

$$\mathcal{L}(f_E) = \mathcal{L}(f_S)^m = (\lambda_e)^m \left[\frac{1}{s + \lambda_e}\right]^m \tag{5.7}$$

where f_E represents the density function of the total execution time of the critical subtask without considering the inter-module delays. By taking the inverse Laplace transform, we can get the probability density function f_E:

$$f_E(t) = \lambda_e^m \frac{t^{m-1}}{(m-1)!} \cdot exp(-t\lambda_e). \tag{5.8}$$

By integrating f_E, one can get the probability distribution of total execution time. Let this distribution be $\Pr[E \le t \mid M = m]$. To reflect the inter-module delays into the total execution time, one can apply (4.2) or (4.3) to the final distribution. Then $\Pr[t_{comp} \le t \mid M = m] = \Pr[E \le t \cdot U/K \mid M = m]$. Since we know the distribution of the number of modules in the critical subtask ($\Pr[M = m]$), the final distribution of execution time is

$$\Pr[t_{comp} \le t] = \sum_{m=1}^{max} \Pr[t_{comp} \le t \mid M = m] \cdot \Pr[M = m]. \tag{5.9}$$

The first part of (5.9) ($\Pr[t_{comp} \le t \mid M = m]$) represents the distribution of task execution time when the number of modules is fixed. The second part of (5.9) ($\Pr[M = m]$) represents the distribution of number of modules in the critical task.

6 Results and Discussion

To demonstrate the utility and power of the proposed model, we applied the model to a parallel algorithm which has potential use in many applications. The selected algorithm is a solution to the 0/1 knapsack problem which is known to be NP-complete. To comprehend the potential use of the knapsack problem for applications, let us consider the problem of assigning a fixed number of resources to multiple objects (e.g., modules or subtasks) in time. Let us assume that each object requires a different number of resources and yields a reward if it has the required resources. Each resource, however, can be assigned only to a single object. Under this setting, we want to find an assignment within a deadline which gives the maximum reward. This is one instance of the 0/1 knapsack problem.

A parallelized solution to this problem was reported in [14]. The task-flow of this parallelized algorithm is given in Fig. 5. In this algorithm, n, c, and p denote the number of data elements, the size of knapsack, and the number of processes, respectively. Each process is assumed to be assigned to a processor, so that the terms "process" and "processor" are used interchangeably. The critical subtask in this example is the task that includes the top module in Fig. 5. The total number of modules in the critical subtask is $M = 3 \log p + 1$ and the overall computation time of this algorithm without considering intercommunications is $M \cdot S = (c \cdot n/p + c(c+1)(1 - 1/2p)) \cdot t_{unit}$, where t_{unit} represents the time to execute a block of instructions which are common to all modules. The total communication time for this algorithm is $(n + 2c \cdot \log p) \cdot d_{unit}$ where d_{unit} is the time needed to send one block of data/information to a corresponding task. In this example, the main (critical) subtask always has the largest piece of computation which is divided into modules. Hence, modules in the main subtask finish last among all communicating modules, so the main subtask does not suffer any synchronization delay even though others do. The communication pattern used in this example is **SEND–RECEIVE** after executing a module. Hence, $x = 0$ and $P_f = 1$.

To find the relation between t_{unit} and d_{unit}, we assumed that system utilization decreases to 75% of one processor utilization when we use two processors for the NMP case. The actual decrease is implementation-dependent and can be measured from the actual system. From these parameters, we computed average execution times using (4.2) and (4.3) for all possible numbers of processes when $n = 300$ and $c = 30$, and the computed results are plotted in Fig. 6. The vertical axis represents a total execution time divided by the execution time when a job is executed in a single processor. The figure is represented in percentile. The solid line represents execution times for the NMP case and the dotted line represents the MP assignment with $K = 2$. As was expected in [14], the execution time decreases until it reaches the saturation point and increases again as we increase the number of processes involved in computation. The reason for this is that the communication time overhead becomes a dominant factor as there is only a small gain in the speed of computation.

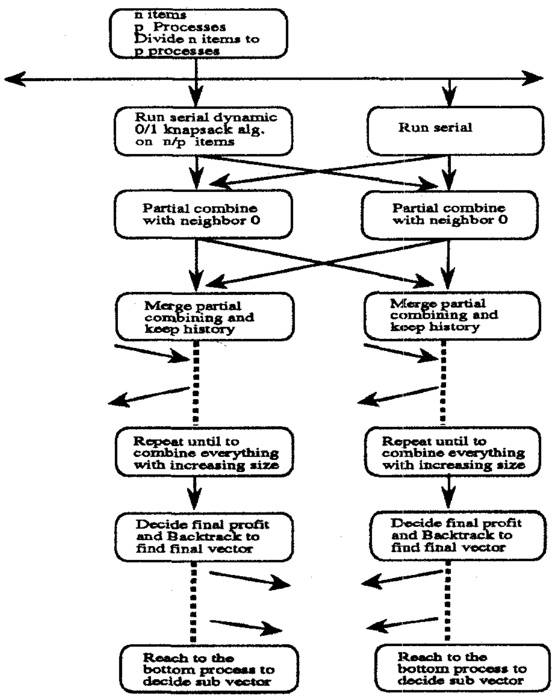

n items
p Processes
Divide n items to p processes
Run serial dynamic 0/1 knapsack alg. on n/p items
Run serial
Partial combine with neighbor 0
Partial combine with neighbor 0
Merge partial combining and keep history
Merge partial combining and keep history
Repeat until to combine everything with increasing size
Repeat until to combine everything with increasing size
Decide final profit and Backtrack to find final vector
Decide final profit and Backtrack to find final vector
Reach to the bottom process to decide sub vector
Reach to the bottom process to decide sub vector

Fig. 5. Task graph of 0/1 knapsack parallel algorithm.

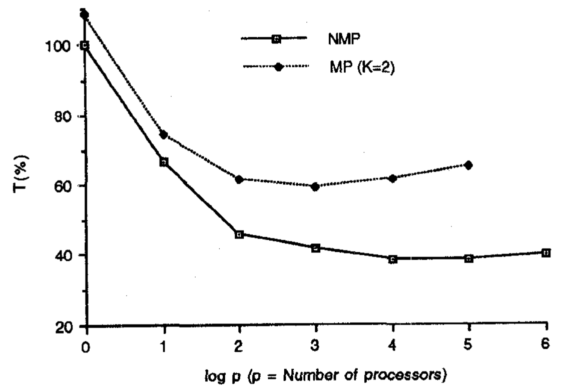


Fig. 6. Execution time comparison for different number of processes.

The density function of execution time for the knapsack problem is plotted in Fig. 7. In this experiment, we assumed that the task is divided into eight parallel subtasks. The average service demand of a module is 20 milliseconds when t_{unit} is assumed 101 microseconds in addition to the conditions used in Fig. 6. The average service demand is computed using the relations: $M = 3 \log p + 1$ and $M \cdot S = (c \cdot n/p + c(c+1)(1 - 1/2p)) \cdot t_{unit}$. This algorithm has a fixed number of modules for each subtask since it depends on the number of subtasks. The execution time of algorithm is a function of the number of data elements. Hence, we modified (5.9) to reflect the distribution of data elements. The modified equation is

$$\Pr\left[t_{comp} \le t\right] = \sum_{k=min}^{max} \Pr\left[t_{comp} \le t \mid n = k\right] \cdot \Pr\left[n = k\right].$$

We used a uniform distribution of n in the interval [200, 500]. The dotted line in this figure represents the probability density function of single module execution time. The solid line represents the probability density function of the total execution time of the critical subtask with the communication delay included.

We also plotted the probability density function of execution time for the MP case. We assumed that tasks are divided into eight subtasks and two subtasks are assigned to a single processing node. The figure indicates that the MP case takes more time to complete tasks than the NMP case, due mainly to resource (CPU) contention.

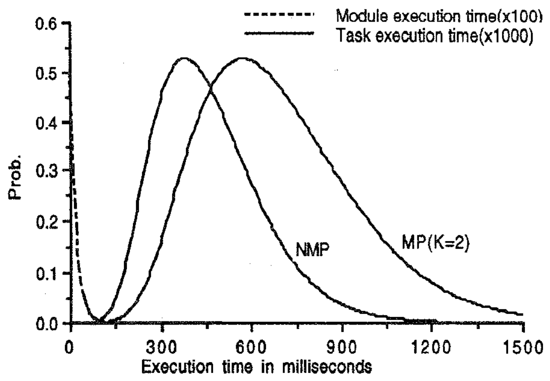


Fig. 7. The *pdf* of task execution time for 0/1 knapsack problem.

To show the relation between system utilization and parameter P_f, we generated a random task and compared the result with the analytical result. Fig. 8 shows the variation of node utilization as a function of P_f for both single (NMP) and multitasking ($K = 2$) cases. As P_f increases, the node utilization increases. This means that the critical subtask only sends messages to other subtasks. The solid line shows the analytical results and the dotted line shows the simulation results. Fig. 9 represents the variation of execution time as a function of P_f for both NMP and MP execution models. For both figures, the number of modules M in the critical subtask is assumed to be 840, 50 units of average service demand of module (S), 10 units of synchronization delay (H), and 25 units of average communication delay (C). As expected with higher P_f, the node is busy most of the time, and thus, does not have a significant semi-active state delay. Note that the actual parameterization of M, S, P_f, and μ_s is algorithm dependent. Here, we used arbitrary values to demonstrate how to apply the proposed model to the analysis of task behavior with given actual parameter values.

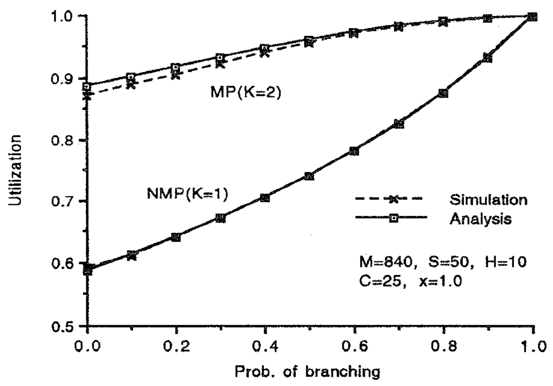


Fig. 8. Utilization versus P_f.

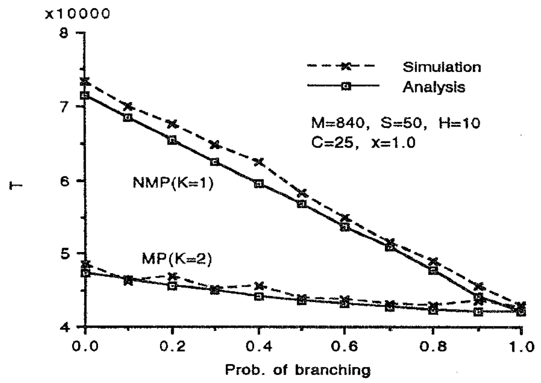


Fig. 9. Execution time versus P_f.

7 Conclusion

In this paper, we proposed a model to estimate the execution times of communicating tasks in distributed systems. We classified all factors affecting the execution time of a task into two groups: intrinsic and extrinsic. The effects of intrinsic factors are reflected into a task model, in which a task is represented with subtasks and modules. It was assumed that the execution time of a task is the same as its critical subtask which finishes last within the task. Tasks are characterized by the service demand of each module, the number of modules in the critical subtask, and the branching probabilities for communication (P_f, x). This characterization depends solely on intrinsic factors. On the other hand, the effects of extrinsic factors are reflected into communication and synchronization delays. For example, the effects of the interconnection network and the routing algorithm are reflected into the communication delay parameter. The environmental difference between processing nodes is reflected into the synchronization delay. These two delays are explicitly specified to show their effects on the execution time. By simplifying all factors, we represented a distributed system with a simple queuing model of two stations: one for computation and the other for communication and synchronization delays. Information on system utilization is obtained by converting this queuing model to a Markov chain. The execution time of a task is derived from the information on system utilization. The analysis results are given in two forms: average and distribution. The distribution of task execution times is desired for the analysis of other important measures such as dynamic failure. Several examples are also presented to show the effectiveness of the model.

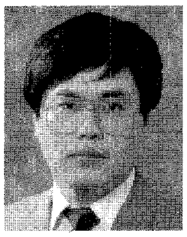
The main difference between the proposed model and previous results reported in the literature lies in the inclusion of extrinsic factors such as communication and synchronization delays in the proposed model. The proposed model can be used for any distributed system since the model includes the abstracted effects of intrinsic and extrinsic factors. The task structure depends on the nature of a parallel algorithm and its implementation on a specific architecture. Similarly, communication and synchronization delays are affected by the interconnection structure and the routing algorithm. As long as we can transform all factors which affect the execution task time into task structure parameters and communication delays, we can derive the execution time distribution using the proposed model.

Acknowledgments

The work reported in this paper was supported in part by the U.S. National Science Foundation under Grant MIP-9203895 and by the U.S. Office of Naval Research under Grant N000-94-1-0229. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agencies.

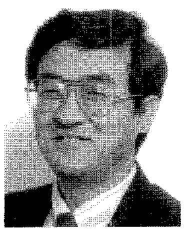
References

[1] M.H. Woodbury and K.G. Shin, "Evaluation of the Probability of Dynamic Failure and Processor Utilization for Real-Time Systems," *Proc Ninth Real-Time System Symp.*, pp. 222-231, Dec. 1988.
[2] M.H. Woodbury, "Analysis of the Execution Time of Real-Time Tasks," *Proc. Seventh Real-Time System Symp.*, pp. 89-96, Dec. 1986.
[3] V. Haase, "Real-Time Behavior of Programs," *IEEE Trans. Software Eng.*, vol. 7, pp. 494-501, Sept. 1981.
[4] W. Chu and K.K. Leung, "Task Response Time Model and Its Applications for Real-Time Distributed Processing Systems," *Proc. Fifth Real-Time System Symp.*, pp. 225-236, Dec. 1984.
[5] C.Y. Park and A.C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Computer*, pp. 48-57, 1991.
[6] A. Stoyenko, "A Real-Time Language with a Schedulability Analyzer," Doctoral dissertation CSRI-206, Univ. of Toronto, Dec. 1987.
[7] K.B. Kenny and K.J. Lin, "Measuring and Analyzing the Performance of Real-Time Programs," *IEEE Software*, vol. 8, no. 5, pp. 41-49, Sept. 1991.
[8] W. Chu and C.M. Sit, "Estimating Task Response Time with Contentions for Real-Time Distributed Systems," *Proc. Ninth Real-Time System Symp.*, pp. 272-281, Dec. 1988.
[9] D. Peng and K.G. Shin, "Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 500-516, Apr. 1987.
[10] R. Nelson, D. Towsley, and A.N. Tantawi, "Performance Analysis of Parallel Processing Systems," *IEEE Trans. Software Eng.*, vol. 14, pp. 532-539, Apr. 1988.
[11] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, N.J.: Prentice Hall, 1982.
[12] J.B. Dugan, R. Geist, and K.S. Trivedi, "The Hybrid Automated Reliability Predictor," *AIAA J. Guidance, Control, and Dynamics*, vol. 9, no. 3, pp. 319-331, 1986.
[13] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, Mass.: MIT Press, 1989.
[14] J. Lee, E. Shragowitz, and S. Sahni, "A Hypercube Algorithm for the 0/1 Knapsack Problem," *Proc. 1987 ICPP*, pp. 699-706, Aug. 1987.



Jong Kim received the BS degree in electronic engineering from Hanyang University, Seoul, Korea, in 1981, the MS degree in computer science from the Korea Advanced Institute of Science and Technology, Seoul, Korea, in 1983, and the PhD degree in computer engineering from Pennsylvania State University in 1991.

Since 1992, he has been an assistant professor in the Department of Computer Science and Engineering, Pohang University of Science and Technology, Pohang, Korea. From 1991 to 1992, he was a research fellow in the Real-Time Computing Laboratory of the Department of Electrical Engineering and Computer Science at the University of Michigan. He was a research assistant in the ECE department of Pennsylvania State University from 1987 to 1990. From 1983 to 1986, he was a system engineer in the Korea Securities Computer Corporation, Seoul, Korea. His major areas of interest are fault-tolerant computing, performance evaluation, and parallel and distributed computing.



Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is a professor and the director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, Michigan.

He has authored/coauthored more than 400 technical papers (more than 150 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He is currently writing (jointly with C.M. Krishna) a textbook *Real-Time Systems* which is scheduled to be published by McGraw Hill in 1996. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from the University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called **HARTS**, and middleware services for distributed real-time fault-tolerant applications.

Dr. Shin has also been applying the basic research results of real-time computing to multimedia systems, intelligent transportation systems, and manufacturing applications ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes.

From 1978 to 1982, he was a member of the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, International Computer Science Institute, Berkeley, California, IBM T.J. Watson Research Center, and Software Engineering Institute. He also chaired the Computer Science and Engineering Division, EECS Department, the University of Michigan for three years beginning in January 1991.

Dr. Shin is an IEEE fellow, was the program chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chairman of the 1987 RTSS, the guest editor of the 1987 August special issue of *IEEE Transactions on Computers* on real-time systems, a program co-chair for the 1992 International Conference on Parallel Processing, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, was a distinguished visitor of the Computer Society of the IEEE, an editor of *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of the *International Journal of Time-Critical Computing Systems*.