JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **28,** 1–18 (1995)

Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures

DANIEL A. MENASCÉ,*,1 DEBANJAN SAHA,† STELLA C. DA SILVA PORTO,‡ VIRGILIO A. F. ALMEIDA,§ AND SATISH K. TRIPATHI†,2

**Department of Computer Science, George Mason University, Fairfax, Virginia 22030-4444; †Department of Computer Science, University of Maryland at College Park, Maryland 20742; ‡Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 22453 Rio de Janeiro, Rio de Janeiro, Brazil; and §Departamento de Ciencia de Computação, Universidade Federal de Minas Gerais, 30161 Belo Horizonte, Minas Gerais, Brazil*

Most parallel jobs cannot be fully parallelized. In a homogeneous parallel machine—one in which all processors are identical—the serial fraction of the computation has to be executed at the speed of any of the identical processors, limiting the speedup that can be obtained due to parallelism. In a heterogeneous architecture, the sequential bottleneck can be greatly reduced by running the sequential part of the job or even the critical tasks in a faster processor. This paper uses Markov chain based models to analyze the performance of static and dynamic processor assignment policies for heterogeneous architectures. Parallel jobs are assumed to be described by acyclic directed task graphs. A new static processor assignment policy, called Largest Task First Minimum Finish Time (LTFMFT), is introduced. The analysis shows that this policy is very sensitive to the degree of heterogeneity of the architecture, and that it outperforms all other policies analyzed. Three dynamic assignment disciplines are compared and it is shown that, in heterogeneous environments, the disciplines that perform better are those that consider the structure of the task graph, and not only the service demands of the individual tasks. The performance of heterogeneous architectures is compared with cost-equivalent homogeneous ones taking into account different scheduling policies. Finally, static and dynamic processor assignment disciplines are compared in terms of performance. © 1995 Academic Press, Inc.

1. INTRODUCTION

A parallel system is characterized by a multiprocessor executing parallel jobs. A multiprocessor consists of a number of cooperating processors. The great majority of multiprocessors are grouped into two types of organization: shared memory and distributed memory. Multiprocessors can also be viewed as homogeneous or heterogeneous. In the first case, the system is composed of a number of identical processors. Heterogeneous multiprocessors consist of processors of different speeds and costs [5, 13, 32, 34].

A parallel job is a collection of many computational tasks, possibly with some precedence constraints. A task can be identified as a computational fragment that must be sequentially executed with no part of it executable in parallel [34].

Most parallel jobs cannot be fully parallelized. In fact, the fraction of sequential processing ranges from 10% to 30% in real parallel applications [14]. In a homogeneous parallel machine—one in which all processors are identical—the serial fraction of the computation has to be executed at the speed of any of the identical processors, limiting the speedup that can be obtained due to parallelism [4]. If a subset of the processors is replaced by a single cost-equivalent more powerful processor, we have a heterogeneous architecture in which the sequential bottleneck can be greatly reduced by running the sequential part of the job or even the critical tasks on a faster processor. Menascé and Almeida [34, 31, 32] and Andrews and Polychronopoulos [5] presented quantitative cost-performance analysis of heterogeneity. Freund discusses heterogeneity in loosely coupled systems and considers the case where different processing paradigms (e.g., SIMD, MIMD, etc) are involved [51, 15, 16]. These results led to the notions of superconcurrency and to the optimal selection theory. Even though we address heterogeneity through the notion that different processors may have different speeds, the techniques presented here can easily be extended to the case of processors of different processing paradigms. In order for the performance advantages of heterogeneity to materialize, appropriate processor scheduling must be done.

Given a multiprocessor system and a set of parallel jobs, the scheduling function is responsible for the exploitation of processors in the system, with the aim at some specific performance goals, such as minimum execution time for a single job or balanced processor utilization in a multiprogramming environment. Scheduling has been vaguely defined as a function that assigns jobs to processors [17]. In a more precise way, as defined in [50], the processor scheduling problem can be viewed as a two-step process,

1 This work was performed while this author was visiting the Institute for Advanced Computer Studies, University of Maryland at College Park, partially supported by CAPES, the National Science Foundation under Grant CCR-9002351 and by IBM Brasil.

2 Partially supported by the National Science Foundation under Grant CCR-9002351.

namely processor allocation and assignment. Processor allocation deals with the determination of the number of processors allocated to a job; the assignment phase refers to the distribution of the job's tasks to the allocated processors.

One important consideration in dealing with the scheduling problem is the kind of information that is available to the scheduler. We assume that the scheduler has information on the structure of the job, in the form of a task graph, as well as timing information on each of the individual tasks. Recent work in compilation theory shows that these assumptions are reasonable. Polychronopoulos presents a technique that can be used by a compiler to generate a task graph from a parallel program source code [42]. This technique can also be used to generate scheduling related information. Program timing and program profiling has been the subject of many recent studies. Park and Shaw [39] developed a tool based on the concept of timing schema that accepts a program and bounds for each loop, and produces predictions for best and worst case execution times. Nirkhe and Pugh [37] developed a technique, based on partial evaluation, which accurately estimates the execution time of general real-time programs employing high-level constructs. Some very recent studies have been conducted on scheduling algorithms when there is no information available to the scheduler before the jobs start their execution. These algorithms are called on-line algorithms and are reported in [46]. In these studies processors may have different speeds, but tasks are independent. Nelson and Towsley [36] used analytic models to study scheduling policies in heterogeneous environments. They consider all tasks as being independent.

A performance comparison of processor assignment disciplines requires that one be able to compute the average execution time of a parallel job described by a task graph. Simulation has been used by many authors for this purpose [2, 3, 33, 45]. The other approach to obtaining the average execution time of a task graph is analytic modeling. Thomasian and Bay [49] used a two-level model in which queuing networks are used to obtain the throughputs for each possible combination of active tasks, and a Markov chain is used to obtain the average job execution time. States in this Markov chain represent sets of tasks in simultaneous execution. The state transition rates are derived from the solution of the several queuing networks. Their construction of the Markov chain is oriented more towards static processor assignment. Menascé and Barroso [30] presented analytic models to obtain the average execution time of task graphs in shared memory multiprocessors. Their model takes into account the contention for the interconnection network and the memory modules. Kapelnikov *et al.* [20], Chu and Leung [9], and Chu *et al.* [10] also present analytic models for task graph based parallel jobs. In [20] a combined queueing network and Markov chain approach is used. In [9, 10] task graph constructs such as sequence of tasks, and-fork to and-join, or-fork to or-join, and loop are successively aggregated into a single node whose execution time is an approximation of the original graph execution time. However, not all task graphs can be aggregated in this manner. In fact, the MVA type of topology [52], the LU decomposition, and the FFT transform [45] are just a few examples of topologies which are not amenable to this aggregation approach. In this paper, we will use Markov chain based techniques in order to obtain the average execution time of task graph based jobs. For the static case, we will use Thomasian and Bay's algorithm [49] to build and efficiently solve the Markov chain. For the dynamic case we will introduce a generalization of their idea. For some particular cases, one will even be able to derive closed form expressions for the average execution time of a parallel job.

This paper is organized as follows. Section 2 explores the relationship between scheduling and the various scenarios of utilization of multiprocessors. Section 3 describes several static and dynamic processor assignment disciplines which are analyzed later. Section 4 presents the Markov chain models used to analyze the performance of various scheduling policies. The next section presents an algorithm to automatically generate the Markov chain and obtain the average execution time of a parallel job under each discipline. Section 6 presents a discussion of the numerical results obtained with the model. Appendix A presents the proofs of the theorems given in the paper. Appendix B contains the derivations of a closed form expression for the average execution time of n parallel tasks on p ($p \leq n$) processors, with one of them being faster than the remaining ones.

2. MULTIPROCESSOR SCENARIOS AND SCHEDULING ISSUES

A key dimension of scheduling policies concerns the frequency with which processor allocation and assignments are made: static vs dynamic [11, 18, 24, 27, 41, 44]. Either or both of the two phases of a scheduling policy can be static or dynamic.

With static scheduling, the decision concerning processor allocation and task assignment is made at the onset of the job execution. Static policies have low run-time overhead and the scheduling costs are paid only once, during the compilation time, for instance. As a consequence, static scheduling policies can afford to be based on sophisticated heuristics that lead to efficient allocations. Parallelism can be exploited by spreading different tasks over different processors statically. However, in order to be effective, static scheduling needs to know in advance detailed information about the job's run-time profile. Because run-time profile is dependent on the input data, static scheduling carries some degree of uncertainty. The result may be an unbalanced load [41], which leads to longer parallel execution time and low system utilization.

With dynamic scheduling, the number of processors allocated to a job may vary during the execution. Also, the task assignment to the allocated processors takes place during the execution of a job. As pointed out in [18, 27, 41], dynamic scheduling policies are complementary to static policies in both their advantages and drawbacks. Because they are implemented at execution time, dynamic policies usually incur in high run-time overhead, which may lead to a degradation of performance. Since decisions are made during job execution, scheduling should be based on simple and constant time heuristics. On the other hand, dynamic scheduling mechanisms exhibit an adaptive behavior, which leads to a high degree of load balancing [27, 38, 40, 41].

Heterogeneity in parallel systems introduces an additional degree of complexity to the scheduling problem. In addition to the problem of deciding when and how many processors to allocate, scheduling policies have also to deal with the choice among processors of different speeds. Performance of static and dynamic scheduling policies for heterogeneous parallel systems is analyzed in [2, 28, 33] through the use of simulation.

Multiprocessors have been utilized in two different modes: monoprogramming and multiprogramming. In this section we discuss the various scheduling mechanisms used to implement the two utilization scenarios.

2.1. Monoprogrammed Parallel Systems

In a monoprogramming environment, parallel jobs are executed one at a time, with each job exploiting its maximum parallelism on the processors of the system. This is the desirable environment when the main goal is to have the minimum job execution time [40]. This environment is typical in many current multiprocessor installations, especially in distributed memory multiprocessors, also called multicomputers [6, 38].

Because the entire system is dedicated to a single job at a time, the major scheduling question in this environment is how to distribute the tasks among the processors to achieve minimal execution time. When a job has more tasks than processors, a scheme is needed to map tasks onto processors. The task assignment phase can be performed either dynamically or statically.

In a monoprogramming environment, the execution of a single job at a time may result in an inefficient use of the system [18]. For instance, when the maximum level of logical parallelism of a job is smaller than the physical parallelism of the system, processors will remain idle during the job execution. Also, the fluctuation of the logical parallelism of a job during its execution contributes for the inefficient use of processors. This is the main motivation that leads to multiprogramming environments.

2.2. Multiprogrammed Parallel Systems

In a multiprogrammed multiprocessor, multiple parallel jobs are active simultaneously. The key role of scheduling policies in multiprogramming environments is the distribution of processing power among a set of parallel jobs that run at the same time. Although multiprogramming provides better system utilization and allows better service to a broad user community, it also complicates resource management [19, 24]. New problems are brought out in this environment. When the number of tasks of the parallel jobs being executed simultaneously exceeds the number of processors, each processor must be shared among multiple tasks. In this case, there are three factors that can adversely affect performance of multiprogrammed parallel systems, namely overhead of context switching between multiple tasks, synchronization primitives that require spin-waiting on a variable, and low cache-hit ratio [19, 22, 27, 47, 53].

In addition to being static or dynamic, scheduling disciplines for multiprogramming can be either preemptive or nonpreemptive. The utility of preemption was observed in situations when a parallel workload exhibits a high variability in processor demands and low variability in number of tasks per job [24].

Scheduling for parallel processing can be performed either at one or two levels [19, 27, 47]. At the higher level, two-level scheduling mechanisms perform the allocation of a set of processors to a job. At the lower level, each job does the task assignment among the processors allocated to it. On the other hand, one-level scheduling, also called single queue of runnable tasks, combines both processor allocation and task assignment at one level.

Due to its simplicity, most of the current shared-memory multiprocessors base their processor scheduling functions upon a single priority queue of runnable tasks. Examples include the DYNIX operating system which runs on the Sequent Balance machines and the Topaz, a Digital Equipment Corporation experimental system [8, 24, 47].

A key issue for scheduling policies in multiprogramming environments is how jobs share the processors of a parallel system. All policies fit into two classes: time-sharing and space-sharing [19, 27]. In the first, the system multiplexes the time of all processors among the jobs being multiprogrammed. A major goal of time-sharing schedulers is allocating resources so that competing jobs receive approximately equal portions of processor time.

Gupta *et al.* [19, 43] evaluate coscheduling, or *gang scheduling*, which schedules all executable tasks of a job to run on the processors at the same time. When a time slice ends, all running tasks are preempted simultaneously, and all tasks from another job are scheduled for the next time slice. This dynamic time-sharing policy solves the problems associated with busy-waiting locks by scheduling all related tasks at the same time. According to [25, 22], dynamic preemptive policies based on the job's characteristics, such as processor demand and number of tasks, perform well for multiprogrammed parallel systems. Leutenegger and Vernon [22] propose a Round Robin job policy that allocates an equal fraction of processing power to each job in the system. They report that for independent tasks or lock synchronization the RRjob policy generally has higher performance than co-scheduling policies. Polices that reallocate processor frequently, aiming at processor utilization, without considering cache implications may result in a poor performance. Squillante and Lazowska [47, 48] propose dynamic policies that schedule tasks on processors for which the tasks have affinity, i.e., the amount of data that a task has in a particular processor's cache. There are various factors that affect the effectiveness of affinity [19, 47, 48]. The most relevant ones are the job *footprint*, the number of intervening tasks that are scheduled on a processor between two successive times that a task is scheduled on that processor, and the length of the period of time for which a task runs on a processor once it is scheduled. The dynamic time-sharing scheduling policies reviewed here are all based on a single shared run queue.

Under space-sharing policies, an appropriate number of processors are allocated to each job. Space-sharing disciplines have a trend to provide each job a more constant allocation of a fewer number of processor than do time-sharing ones [27]. With static scheduling policies, the number of processors allocated to a job does not change during its execution. On the other hand, with dynamic policies the number of processors allocated to each job varies in response to changes in the job parallelism and the system processor availability. Multicomputers, that include several well-known commercial hypercube computers, normally operate on a space-sharing basis [6].

Space-sharing involves partitioning the processors among the parallel jobs. Partitioning schemes can be either fixed or variable. In the first case, the processor partitioning is divided into fixed-sized partitions. The partition sizes are set at system generation time and do not change [12]. This scheme is appropriate for fixed workload environments. With variable partitioning, the size of the partition allocated to each job varies, according to the job's characteristics and scheduling policy. Variable partitioning is implemented by the allocation phase of scheduling disciplines, which can be either static or dynamic.

Dowdy [11] proposes a static allocation scheme, where the processor partitioning is based on the *execution signatures* of the jobs. The characteristic speedup function which relates the job's execution time to the number of processors allocated to it is called execution signature [11]. Tripathi and co-workers [18] also propose a static processor allocation discipline based on the *processor working set*, which is defined as the minimum number of processors that maximizes speedup per unit of a cost function that incorporates the number of processors used and the associated speedup.

Zahorjan and co-workers [27] propose two-level scheduling disciplines based on space-sharing. At the higher level, the policies allocate processors to a job, forming a partition. At the lower level, the job itself assigns its tasks to the processors within the partition. They propose a quasi-static space-sharing policy called *Equipartition*, that, to the extent possible, maintains an equal allocation of processors to all jobs. *Dynamic* is a dynamic space-sharing discipline that reallocates processors in response to jobs' demands. They conclude that *Dynamic* presents a better performance than that of the proposed policies. Gupta *et al.* [19] also study the performance of two-level scheduling disciplines. They assume that processors are equally partitioned among the jobs. Within each partition, each job performs the task assignment among the processors according to some specific policy, such as priority scheduling with blocking locks. McCann and Zahorjan [26] investigate the problem of building processor allocation policies for large scale, message-passing parallel computers supporting a scientific workload.

3. PROCESSOR ASSIGNMENT DISCIPLINES

The processor assignment disciplines considered in this paper fall into the category of priority list scheduling [21], which implies that the schedulable tasks are ranked in a priority list according to a given heuristic. The highest priority task in this list is assigned to a processor according to a heuristic.

Let the *service demand* of a task t, denoted $sd(t)$, be defined as the average time taken for that task to execute on the fastest processor of the multiprocessor. Let us define *level* of task t, denoted $l(t)$ (similar to [1, 21, 45]) as the maximum of the sum of the service demands on all paths that lead from the task to the final task in the task graph. More precisely,

$$l(t) = \max_{\pi \in \Pi(t)} \sum_{t_k \in \pi} sd(t_k), \tag{1}$$

where $\Pi(t)$ is the set of paths in the task graph that start at task t and go to the final task. In order to illustrate the concept, consider the task graph of Fig. 1 and assume that the service demands of the tasks 1 through 6 are 10, 8, 3, 4, 13 and 2 time units, respectively. Then the *level* of task 1 is equal to $10 + 13 + 2 = 25$, the *level* of task 2 is equal to $8 + 4 + 2 = 14$, and the *level* of task 5 is equal to $13 + 2 = 15$.

Another dual concept is the *co-level* [1, 21] of a task t, denoted $cl(t)$, which is defined as the maximum of the sum of the service demands among all the paths from the initial task to the task in question. The formal definition of *co-level* can also be given by Eq. 1 provided the set $\Pi(t)$ be defined as the set of paths in the task graph that go from the initial task into task t. Using the same task graph, we can see that the *co-level* of task 6 is $10 + 13 + 2 = 25$.

We define the *weighted level* of task t denoted $wl(t)$ (in a similar way as in Shirazi *et al.* [45]) as the sum of the service demand of a task with the maximum of the *weighted levels* of all its successors plus the sum of the *weighted levels* of the successors of t normalized by the maximum *weighted level* among all successors of t. More precisely,

$$wl(t) = sd(t) + \max_{t_i \in succ_G(t)} wl(t_i) + \frac{\sum_{t_i \in succ_G(t)} wl(t_i)}{\max_{t_i \in succ_G(t)} wl(t_i)}, \tag{2}$$

where $succ_G(t)$ is the set of immediate successors of t in the task graph. Shirazi *et al.* [45] give an algorithm to find

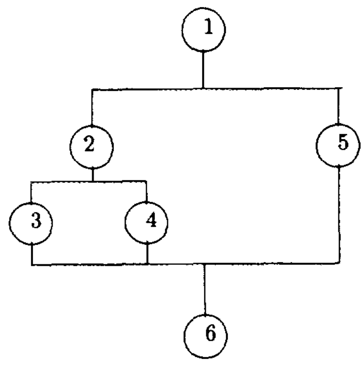

FIG. 1. Task graph example.

the *weighted level* of all tasks in the graph. In the example of Fig. 1 the *weighted levels* of tasks 1, 2, and 5 are 85.85, 16.86, and 16, respectively.

3.1. Static Processor Assignment Disciplines

In [28, 33], Menascé *et al.* presented a methodology for systematically building static heuristic processor assignment algorithms. An assignment algorithm was considered to be composed of two parts: an envelope and a heuristic. The envelope is responsible for selecting, at each step, the subset of tasks—called task domain—and the subset of processors—called processor domain—to be considered by the heuristic. The heuristic selects a (task, processor) pair from the task and processor domains, respectively. So, a static heuristic processor assignment algorithm is equal to an envelope + heuristic. The large number of simulation studies reported in [28, 33] concluded that for heterogeneous environments, the envelope that outperformed all others was the Deterministic Execution Simulation (DES). Briefly, DES carries out a straightforward deterministic simulation of the execution of the parallel job considering that task execution times are *deterministic* and equal to their estimated average values. In this simulation, any heuristic is used to select the appropriate task-processor pair from the domain. These pairs are selected for scheduling, at each instance of the simulation process, until there are no more pairs left, i.e., either the task domain or the processor domain (or both) are empty. At this point, the simulation clock is set to the time of the next event: the completion of an executing task. Then, DES updates the task domain based on the precedence graph, and the processor domain based on the set of processors which are not executing any task at this time of the simulation process. This iterative procedure ends when all tasks have been scheduled, which means that they have all been *executed* during this simulation. Note that DES is *not* a simulation, for performance analysis reasons, of the operation of a scheduling algorithm. It is a static scheduling algorithm when combined with a task and processor selection heuristic.

The three algorithms presented in [45] can be cast in terms of the envelope + heuristic approach. The envelope in that case is a modified DES, referred as MDES hereafter, which only updates the task domain when it becomes empty, as opposed to DES which updates the task domain every time a task *completes* in the execution simulation.

All static processor assignment algorithms considered here are formed by using DES as an envelope and any of the task and processor selection heuristics defined below. The heuristics are divided into task selection and processor selection ones. The three task selection heuristics are:

LTF (*Largest Task First*): selects the task from the task domain which has the largest value of service demand.

HLF (*Highest Level First*): selects the task from the task domain which has the highest *level*, as defined above.

WLF (*Highest Weighted Level First*): selects the task from the task domain which has the highest value of *weighted level*, as defined above.

Once a task is selected according to any one of the above heuristics, the processor is selected using any of the two following disciplines:

SEET (*Shortest Estimated Execution Time*): selects the processor from the processor domain which takes less time to execute the selected task.

MFT (*Minimum Finish Time*): selects the processor that minimizes the completion time of the selected task in the Deterministic Execution Simulation.

So, a heuristic results from the combination of a task selection policy with a processor selection one. For instance, LTFMFT selects the task from the task domain which has the largest service demand (LTF), and the processor that minimizes its completion time (MFT) under the deterministic execution simulation. The static processor assignment algorithms to be analyzed in Sect. 6 are: DES+LTFMFT, DES+LTFSEET, DES+HLFSEET, DES+WLFSEET, and MDES+LTFSEET.

It is interesting to note that Baxter *et al.* [7] report on an evaluation they conducted of DES and concluded that it is a viable approach for experimentation on real applications. They even incorporated a DES based scheduler into the design tools they developed for real-time control systems. Menascé and Porto [29] extended DES based schedulers to message passing heterogeneous multiprocessors.

3.2. Dynamic Processor Assignment Disciplines

We are going to consider dynamic processor assignment disciplines which use information on tasks that was available before the job started its execution, such as estimated service demands or the graph topology. The following processor assignment disciplines will be considered.

Fastest Processor to Largest Task First (*FPLTF*): This policy gives the fastest processor to the task with the highest service demand, the second fastest processor to the task with the second highest service demand task, and so on and so forth.

Fastest Processor to Highest Level Task First (*FPHLTF*): This policy gives the fastest processor to the task, among the schedulable ones, with the highest *level* first, the second fastest processor to the task with the second highest *level*, and so on and so forth.

Fastest Processor to Lowest Co-Level Task First (*FPLCTF*): This policy gives the fastest processor to the task, among the schedulable ones, with the lowest *co-level* first, the second fastest processor to the task with the second lowest *co-level*, and so on and so forth.

In order to compare the various processor assignment disciplines, we need performance models of parallel jobs in heterogeneous environments. Next section discusses the models used in this paper.

4. PERFORMANCE MODELS OF PROCESSOR ASSIGNMENT DISCIPLINES

The performance models described in this section are based on Markov chains which represent the state of execution of a parallel job by indicating the tasks that are executing simultaneously, and in some cases the processors assigned to it. We assume throughout this paper that task execution times are exponentially distributed. However, we will indicate how distributions with different coefficients of variation can be treated. We also assume the general case where the number of processors allocated to the job is less than or equal to its maximum parallelism. Hence, there may be contention for processors. The inputs for the performance model include the inputs assumed to be available to the scheduler:

- $P = \{P_1, ..., P_p\}$ is the set of processors. Processors are assumed to be numbered in decreasing speed order. So P_1 is the fastest processor.
- $G = (T, \beta, R)$ is the task graph, where $T = \{t_1, ..., t_n\}$ is the set of tasks; β is an $n \times p$ matrix such that $1/\beta_{t_i,j}$ is the average execution time of task i at processor j, for $i = 1, ..., n$, and $j = 1, ..., p$; and $R \subseteq T \times T$ is the precedence relation between tasks which indicates that task t_k can only start when all tasks t_i, such that $(t_i, t_k) \in R$, have finished. Note that $1/\beta_{t_i,1}$ is the service demand of task t_i. We assume, without loss of generality, that there is a task, called the initial task, which precedes every other task in the job. Let t_1 be such a task. We also assume, that there is a task, called the final task, which must be executed after all other tasks have completed.

The performance model also takes as input the processor assignment policy.

The main performance metric we consider is the average execution time, T, of the parallel job. Other performance metrics such as F_s, the fraction of sequential processing, the average parallelism, A, defined as the average number of tasks concurrently in execution can be easily obtained as a function of the state probabilities of the Markov chain, as these probabilities are being computed. We may also compute for each processor i, its utilization U_i, as the fraction of time the processor is busy.

In general, a state description for a Markov chain that represents the execution of a parallel job is the triple (AS, AP, ET) where $AS = \{t_{i_1}, ..., t_{i_m}\}$ is the active set, i.e., set of tasks in execution concurrently, $AP = \{P_{j_1}, ..., P_{j_m}\}$ is the set of processors assigned to the tasks in AS (we assume that P_{j_k} is assigned to task t_{i_k} for all $k = 1, ..., m$), and ET is the set of tasks already completed. Our graphical representations of Markov chains will not include the set ET since this can be easily derived by inspection of the path from the initial state into the state in question. The following notation will be used, respectively, for the AS, AP, and ET components of a state $\mathbf{s}$: $as(\mathbf{s})$, $ap(\mathbf{s})$, and $et(\mathbf{s})$. In terms of state probabilities, F_s, A, and U_i may be computed as follows:

$$F_s = \sum_{\forall \mathbf{s} \text{ s.t.} |as(\mathbf{s})|=1} p(\mathbf{s}) \tag{3}$$

$$A = \sum_k k \cdot \sum_{\forall \mathbf{s} \text{ s.t.} |as(\mathbf{s})|=k} p(\mathbf{s}) \tag{4}$$

$$U_i = \sum_{\forall \mathbf{s} \text{ s.t.} P_i \in ap(\mathbf{s})} p(\mathbf{s}). \tag{5}$$

4.1. Static Processor Assignment Model

The state description for the static processor assignment case does not need to include the set AP, since this set is the same for all states.

The performance model for a given static processor assignment policy (SPAP) has the following structure:

Given the task graph G, the set of processors P, and the SPAP, do:

1. Apply SPAP to G and obtain a static assignment, $\mathcal{A}$: $T \mapsto P$, of tasks to processors. So, $\mathcal{A}(t)$ is the processor assigned to task t.
2. Given $\mathcal{A}$ and G automatically generate the Markov chain to represent the evolution of the state of execution of the parallel job. As the Markov chain is generated, compute the unnormalized sum, ΣP, of the steady state probabilities of all states assuming that the unnormalized probability of the initial state,[3] $\mathbf{s}_0$, is 1. The automatic generation of this Markov chain can be done using the algorithm presented in [49].
3. The steady state probability of the initial state, $p(\mathbf{s}_0)$, is simply $1/\Sigma P$. The completion rate of the initial task, X, is also the completion rate of the job, and is equal to $\beta_{t_1,\mathcal{A}(t_1)} \times p(\mathbf{s}_0)$. Since in every cycle there is only one job in execution, it follows by Little's law [23] that the average execution time of the job, T, is given by

$$T = \frac{1}{X} = \frac{1}{\beta_{t_1,\mathcal{A}(t_1)} \times p(\mathbf{s}_0)} = \frac{\Sigma P}{\beta_{t_1,\mathcal{A}(t_1)}} \tag{6}$$

[3] State in which the only task in execution is the initial task.

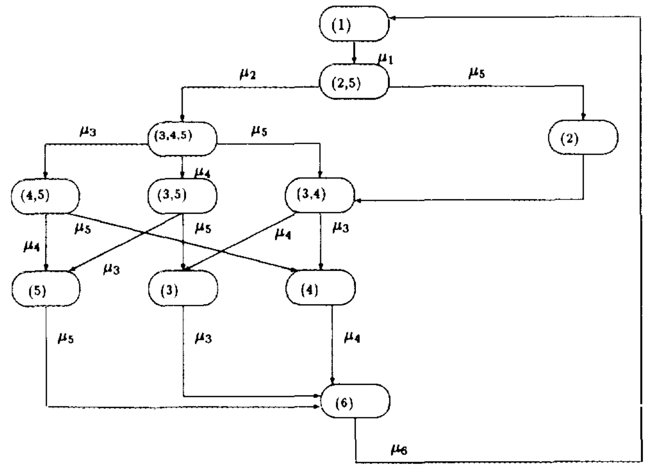

FIG. 2. MC for static processor assignment.

The MC model for the task graph in Fig. 1 is shown in Fig. 2. This model is similar to the one proposed by Thomasian and Bay in [49]. However, their work was not aimed at studying scheduling policies. The transition rates depend on the static assignment of tasks to processors. Different assignments will result in different transition rates. In general, the transition rate μ_{t_k} out of a given state due to the completion of task t_k is $\mu_{t_k} = \beta_{t_k,\mathcal{A}(t_k)}$. Since the assignment is static, μ_{t_k} is the same for every state where there is such a transition.

4.2. Dynamic Processor Assignment

In the dynamic case, the assignment of tasks to processors is state dependent. Let $\mathcal{A}(t, \mathbf{s})$ be the processor assigned to task t at state $\mathbf{s}$.

The performance model for a given dynamic processor assignment policy (DPAP) has the following structure:

Given the task graph G, the set of processors P, and the DPAP do:

1. Automatically generate the Markov chain to represent the evolution of the state of execution of the parallel job. As the Markov chain is generated, compute the unnormalized sum, ΣP, of the steady state probabilities of all states assuming that the unnormalized probability of the initial state is equal to 1. The automatic generation of this Markov chain can be done using the algorithm given in Section 5. As indicated later, the transition rates in this Markov chain are a function of the DPAP and are generated as the Markov chain is generated, since they are state dependent.

2. The steady state probability of the initial state, $p(\mathbf{s}_0)$, is simply $1/\Sigma P$. As in the static case, the average execution time of the job, T, can be obtained by Little's Law as

$$T = \frac{1}{\beta_{t_1,\mathcal{A}(t_1,\mathbf{s}_0)} \cdot p(\mathbf{s}_0)} = \frac{\Sigma P}{\beta_{t_1,\mathcal{A}(t_1,\mathbf{s}_0)}}. \quad (7)$$

States in the Markov chain are generated in a level by level basis as explained in what follows. Transitions from one state to another occur due to the completion of a single task. So, the number of completed tasks at any state of a given level is one plus the number of completed tasks at any state of the previous level. See Appendix A for a formal proof of this property. Starting with the initial state at level 0, one generates a state at level 1 due to the completion of the initial task. This new state may have several tasks in its active set. The completion of each one of them will generate new states at the next level. New states are constructed by replacing a completed task with zero or more tasks taking into account task precedence and processor constraints. The dynamic assignment policy is applied to assign new tasks to processors in each new state. A formal presentation of the algorithm used to generate the Markov chain and to compute the average execution time of the job is given in Section 5.

Figure 3 shows the Markov chain for the task graph of Figure 1. States are numbered for easy reference. Moreover, the list of processors assigned to each of the active tasks is shown in parentheses outside the oval that represents the state. In this example, we assume that there are three processors, i.e., there is no processor contention, and that we are using the FPLTF assignment discipline discussed in Section 3. Let tasks be ordered in decreasing order of service demand as t_6, t_2, t_4, t_5, t_3, and t_1. We introduced a transition from the final state into the initial state to indicate a situation where the parallel job is immediately restarted as soon as it is completed. If we ignore this transition, the Markov chain is acyclic which lends itself to an efficient computation of the steady state probabilities as shown below. As we can see in the diagram, states 6 and 7 have the same active sets but different processor assignments, therefore, they have to be represented as different states in the Markov chain. The same can be said about states 9 and 11. In the static case, states with the same active sets are indistinguishable.

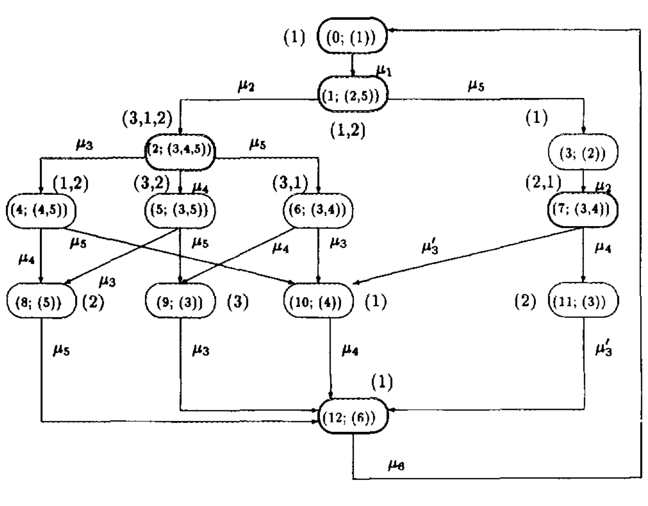

FIG. 3. Markov chain for task graph of Fig. 1.

Let $\mu_{t_k}(\mathbf{s})$ be a transition out of state **s** due to the completion of task t_k. This transition rate depends on the processor in which t_k was assigned to, and therefore is a function of the assignment configuration of this task in this state. So the transition rate out of state **s** due to the completion of task t_k is $\mu_{t_k}(\mathbf{s}) = \beta_{t_k, \mathcal{A}(t_k, \mathbf{s})}$. The different transition rate values in the diagram are a function of the assignment of tasks to processors. So, $\mu_1 = \beta_{t_1,1}$, $\mu_2 = \beta_{t_2,1}$, $\mu_3 = \beta_{t_3,3}$, $\mu_3' = \beta_{t_3,2}$, $\mu_4 = \beta_{t_4,1}$, $\mu_5 = \beta_{t_5,2}$, and $\mu_6 = \beta_{t_6,1}$.

This Markov chain has some interesting properties presented below and proved in Appendix A. Let $\mathcal{L}_k$ be the set of states generated at level k.

Theorem 1 states that the number of tasks that finished at any state in level k is exactly k. As a direct consequence of this theorem, there can be no transitions from states in $\mathcal{L}_k$ into states in $\mathcal{L}_j$ for $j < k$, except for the transition from the final state to the initial state. If that were the case, one would have a transition from a state with k executed tasks into a state with fewer completed tasks, which is impossible.

THEOREM 1. *If* $\mathbf{s} \in \mathcal{L}_k$ *then* $|et(\mathbf{s})| = k$.

Theorem 2 below is a straightforward consequence of Theorem 1 and says that states generated at different levels have different sets of executed tasks. Therefore, even if they have the same set of active tasks and the same set of assigned processors, these states must be considered as different states. This important property allows for an efficient implementation of the algorithm for generating the Markov chain, since checking for an already existing state needs only to be done at the level that is being generated. Therefore, the algorithm only needs to store two levels simultaneously.

THEOREM 2. *If* $\mathbf{s} \in \mathcal{L}_k$ *and* $\mathbf{s}' \in \mathcal{L}_j$ *for* $k \neq j$, *then* $et(\mathbf{s}) \neq et(\mathbf{s}')$.

It can also be shown (see Theorem 3) that in the particular case where there is no processor contention, if two states have the same active sets, then they have the same set of executed tasks.

THEOREM 3. *If there is no processor contention, and if* $as(\mathbf{s}) = as(\mathbf{s}')$, *then* $et(\mathbf{s}) = et(\mathbf{s}')$.

The usefulness of Theorem 3 is to provide for an optimization on the comparison of two potential duplicate states in the algorithm. In the no processor contention case, we check first for the equality of the active sets. If they are equal we do not need to compare the sets of executed tasks.

5. AUTOMATIC MC GENERATION ALGORITHM

We give below the algorithm that generates automatically the MC and solves for the average parallel job execution time T.

Let us first start with some definitions:

- $succ_G(t_i)$: set of immediate successors of task t_i in the task graph G.
- $pred_G(t_i)$: set of immediate predecessors of task t_i in the task graph G.
- $\mathcal{L}$: set of states from which new states are going to be generated from.
- $\mathcal{L}'$: set of states generated from the states in $\mathcal{L}$.
- τ': set of tasks assigned in state $\mathbf{s}'$.
- τ: set of tasks that become executable in state $\mathbf{s}'$.
- $p(\mathbf{s})$: unnormalized steady state probability of state **s**.
- ΣP: normalization constant.
- $proc(t, \mathbf{s})$: processor assigned to task t in state **s**.
- $\Sigma\mu^+$: summation of the transition rates out of state $\mathbf{s}'$.
- $\Sigma p\mu^-$: summation of the products of rates into state $\mathbf{s}'$ times the probabilities of the states from which the transition into $\mathbf{s}'$ originates.

Let us also define the operator $\oplus$ as an operator used to generate a new state $\mathbf{s}'$ from a state **s** by adding a set of tasks $\{t_1, ..., t_k\}$ to the active set of $\mathbf{s}'$. This operation, denoted as $\mathbf{s}' = \mathbf{s} \oplus \{t_1, ..., t_m\}$, has the following effect:

- $as(\mathbf{s}') = as(\mathbf{s}) \cup \{t_1, ..., t_m\}$
- $proc(t, \mathbf{s}') = proc(t, \mathbf{s}) \forall t \in (as(\mathbf{s}) \cap as(\mathbf{s}'))$
- $proc(t, \mathbf{s}') = \mathcal{A}(t, \mathbf{s}') \forall t \in \{t_1, ..., t_m\}$
- $\mu_t(\mathbf{s}') = \mu_t(\mathbf{s}) \forall t \in (as(\mathbf{s}) \cap as(\mathbf{s}'))$
- $\mu_t(\mathbf{s}') = \beta_{t, \mathcal{A}(t, \mathbf{s}')} \forall t \in \{t_1, ..., t_m\}$.

Let us define the operator $\ominus$ as an operator which deletes from a state a set of tasks from its active set. The notation used is $\mathbf{s}' = \mathbf{s} \ominus \{t_1, ..., t_m\}$ meaning that $as(\mathbf{s}') = as(\mathbf{s}) - \{t_1, ..., t_m\}$.

The algorithm is described in Fig. 4.

Algorithm *Assign*$(\tau, \mathbf{s}')$ is the part of the algorithm which is dependent on the dynamic processor assignment policy. As a function of the set, τ, of tasks that may be assigned to processors and as a function of the number of processors available for assignment, this function assigns tasks to processors returning a subset of tasks in τ that could be assigned to processors. For each such task, the assignment function $\mathcal{A}$ is computed.

The algorithm *Duplicate*$(\mathbf{s}')$ returns a true value if state $\mathbf{s}'$ already exists, and false otherwise. As a consequence of Theorem 2 we only need to check for duplicates at the level that is currently being generated. For each new state $\mathbf{s}'$ that is generated at level $\mathcal{L}'$, one has to check whether there is a state in $\mathcal{L}'$ with the same active set and the same set of assigned processors. In the affirmative case, $\mathbf{s}'$ is not considered to be a new state. When there are more than one processor with the same capacity, states with the same set of active tasks having processors of the same speed assigned to the same tasks, can be aggregated into a single state. This reduces considerably the total number of states. For instance, suppose that states $\mathbf{s}_1$ and $\mathbf{s}_2$ have tasks t_1 and t_2 active and let $ap(\mathbf{s}_1) = \{P_{i_1}, P_{i_2}\}$ and $ap(\mathbf{s}_2) = \{P_{j_1}, P_{j_2}\}$. If P_{i_1} has the same capacity as P_{j_1} and P_{i_2} has the same capacity as P_{j_2} then states $\mathbf{s}_1$ and $\mathbf{s}_2$ can be aggregated into a single state.

```
$n \leftarrow 0$;
$\vec{s} \leftarrow \{\vec{s}_0\}$; $et(\vec{s}_0) \leftarrow \emptyset$;
$p(\vec{s}_0) \leftarrow 1$; $\Sigma P \leftarrow 1$;
$\mathcal{L} \leftarrow \{\vec{s}_0\}$ ;
while $\mathcal{L} \neq \emptyset$ do
  begin
    $\mathcal{L}' \leftarrow \emptyset$ ;
    for each $\vec{s} \in \mathcal{L}$ do
      begin
        $\Sigma p\mu^- \leftarrow 0$;
        for each $t_i \in as(\vec{s})$ do
          begin
            $n \leftarrow n + 1$ ;
            { delete task $t_i$ from the active set of $\vec{s}$ }
            $\vec{s}' \leftarrow \vec{s} \ominus \{t_i\}$ ;
            { add task $t_i$ to the set of executed tasks in $\vec{s}'$ }
            $et(\vec{s}') \leftarrow et(\vec{s}) \bigcup \{t_i\}$ ;
            { include in the active set of $\vec{s}'$ all tasks which can
              be started due to the completion of $t_i$. }
            $\tau \leftarrow \{t \mid t \in succ_G(t_i) \wedge (pred_G(t) \subseteq et(\vec{s}'))\}$ ;
            $\tau' \leftarrow Assign(\tau, \vec{s}')$ ;
            $\vec{s}' \leftarrow \vec{s}' \oplus \tau'$ ;
            { If the resulting active set of $\vec{s}'$ is not empty
              then add state $\vec{s}'$ to $\mathcal{L}'$ }
            If $as(\vec{s}') \neq \emptyset$ & $\neg Duplicate(\vec{s}')$
            then begin
                  $\mathcal{L}' \leftarrow \mathcal{L}' \bigcup \{\vec{s}'\}$;
                  $\Sigma\mu^+ \leftarrow \sum_{t \in as(\vec{s}')} \mu_t(\vec{s}')$
                end;
            else $n \leftarrow n - 1$;
            $\Sigma p\mu^- \leftarrow \Sigma p\mu^- + p(\vec{s}) \cdot \mu_{t_i}(\vec{s})$
          end;{ generate all state from $\vec{s}$ }
      end; { scan all states in $\mathcal{L}$ }
    { Compute Unnormalized Probabilities of states in $\mathcal{L}'$ }
    for each $\vec{s}' \in \mathcal{L}'$ do
      begin
        $p(\vec{s}') \leftarrow \Sigma p\mu^- / \Sigma\mu^+$ ;
        $\Sigma P \leftarrow \Sigma P + p(\vec{s}')$
      end ;
    $\mathcal{L} \leftarrow \mathcal{L}'$
  end; { while }
{ Compute the Average Execution Time }
$T \leftarrow \Sigma P / \beta_{t_1, \mathcal{A}}(t_1, \vec{s}_0)$
```

FIG. 4. Markov chain generation algorithm.

Special attention has to be paid to the design of the data structure that represents a level since checking for a duplicate can involve $\mathcal{O}(n^2)$ state comparisons, where n is the number of states in a level, if a simple linked list of states is used to represent the set of states at the same level. We need a data structure which has very low cost for finding an element and that allows for a sequential scan of all the elements. The operations to be supported by this data structure are retrieval of an element and insertion of an element. We used a B^* tree where a key is the state description, which can be efficiently coded in bit vectors. This way, at each level we need to perform only $\mathcal{O}(n \log_d n)$ state comparisons, where d is the degree of the B^* tree.

6. NUMERICAL RESULTS

The Markov chain approach presented in the previous sections was used to study the performance of several static and dynamic scheduling algorithms. The results of these studies are reported in this section. A large number of situations was analyzed with the use of the Markov chain solver that was developed. The curves shown and discussed in this section are representatives of similar results obtained in our extensive studies.

In the numeric studies presented below, we assume that we have two types of processors: a fast one (P_1) and $m - 1$ slower and identical processors.

Some of the curves presented in the following subsections show a normalized execution time, T_{norm}, defined as the ratio of the average execution time of the job on the given heterogeneous architecture and the average response time of the same job on a machine with an infinite number of processors identical to the slowest processor.

The numerical cases considered in what follows explore the variation of T_{norm} as a function of two different parameters:

Processor Power Ratio (α) defined as the ratio between the speed of the fast processor and that of any of the slow ones. This ratio, first introduced in [34], measures the degree of heterogeneity of the architecture.

Fraction of Sequential Processing (F_s) defined as the fraction of time that the job would use a single processor if a pool of infinite identical processors were available to it. This measure is intrinsic to the job and is not influenced by contention for processors [44].

Several types of parallel jobs were considered as a benchmark for the evaluation of the scheduling policies discussed here. Two of them—MVA and GRAVITY—were described in [52]. MVA, shown in Fig. 5, represents several types of computation qualified as wave front, which exhibit an increasing parallelism in the first phase and a decreasing parallelism in the second phase. The MVA job considered here has maximum parallelism equal to 6.

GRAVITY represents the implementation of a clustering algorithm for the real time simulation of the interaction of a large number of stars. The topology of this job, shown in Fig. 6, exhibits 5 different and-fork constructs of identical tasks interleaved with synchronization tasks (the black ones in the figure), which must be executed sequentially. We assume here that all synchronization tasks have the same service demand, $1/\mu_s$, and that all parallel tasks have the same service demand $1/\mu_p$.

For the GRAVITY topology, one can derive a closed form expression for F_s as a function of μ_s and μ_p. This can be accomplished by solving a Markov chain where the state k is the number of tasks yet to be completed. The expression for F_s for the GRAVITY job is given by

$$F_s = \frac{5 + 4 \cdot \delta}{5 + \delta \cdot (2 \cdot H_8 + H_{12} + H_{10})} \tag{8}$$

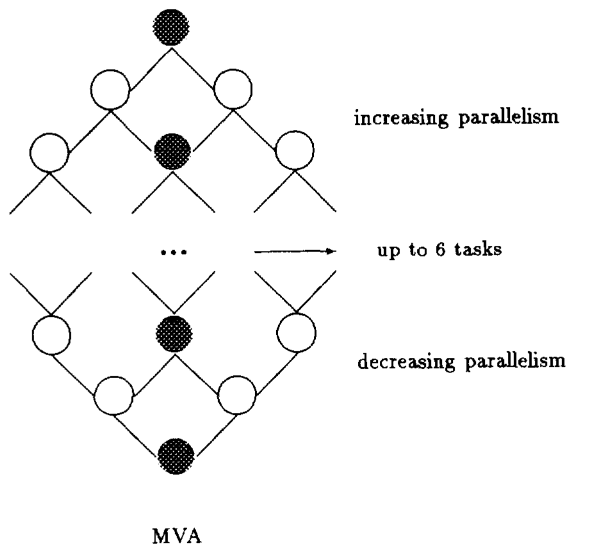

FIG. 5. MVA topology.

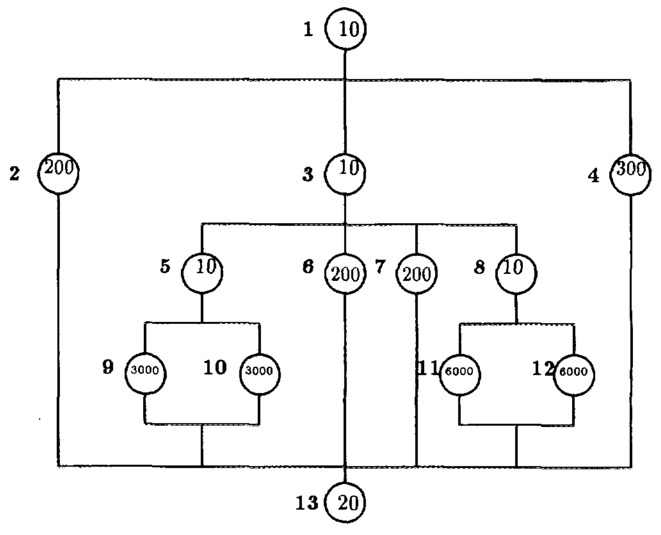

FIG. 7. Parallel job TG

where δ is defined as the ratio μ_s/μ_p and H_n is the harmonic number. From Eq. 8 we can see that when $\delta = 0$, F_s is equal to 1. The minimum value of F_s is obtained when $\delta \rightarrow \infty$. In this case, $F_s \rightarrow 4/(2 \cdot H_8 + H_{12} + H_{10}) = 0.35$. In the curves shown below, we maintained μ_p fixed at 1 and varied μ_s so that F_s would vary from its minimum value up to 1.

In order to analyze the GRAVITY topology for the dynamic case, one need not use the algorithm shown in Section 5. In fact, it is possible to derive a closed form expression for the average execution time, T_{grav}, of a GRAVITY type job with s synchronization tasks (the black ones) and $s - 1$ fork-and-join structures. Let n_i be the number of tasks of the ith fork-and-join structure, and let $T(n_i)$ be its average execution time. Then

$$T_{\text{grav}} = s \cdot \frac{1}{\alpha \cdot \mu_s} + \sum_{i=1}^{s-1} T(n_i). \tag{9}$$

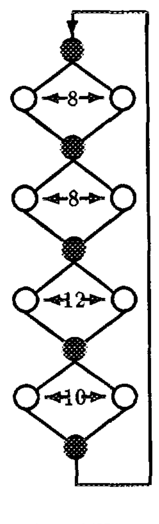

FIG. 6. GRAVITY topology.

Appendix B shows the derivation of a closed form expression for $T(n)$ as a function of the number of processors and of the processor power ratio α.

For the MVA topology, we start with all tasks identical and start to increase the service demand of all tasks along the central axis (see the black tasks in Fig. 5) by the same amount.

We also conducted experiments with the LU decomposition and Scene Analysis task graphs presented by Shirazi *et al.* [45], which along with MVA and GRAVITY form a representative workload of actual parallel applications. Besides these, we also used an artificial task graph, shown in Fig. 7 and called TG hereafter, in order to try to explore the differences among various dynamic assignment disciplines.

6.1. Static Policies

Figure 8 shows the normalized response time for GRAVITY as a function of F_s, for $\alpha = 16$ and five processors. The dashed line represents the behavior of DES+LTFMFT, while the solid curve represents the behavior of the remaining disciplines, namely DES+LTFSEET, DES+HLFSEET, DESM+WLFSEET, and DESM+LFTSEET, which for this application should yield similar results. As it can be seen, DES+LTFMFT gives the best performance among all assignment algorithms analyzed. This is true for all values of α and number of processors analyzed. However, the higher the value of α, the bigger is the difference between DES+LTFMFT and the other disciplines.

The same kind of behavior was observed for the MVA topology (see Fig. 9). As with the GRAVITY case, the DES+LTFMFT algorithm outperformed the remaining ones, especially for higher values of α.

Figure 10 depicts the variation of the normalized response time for MVA as a function of α. As it can be seen, all disciplines, except DES+LTFMFT, behave similarly

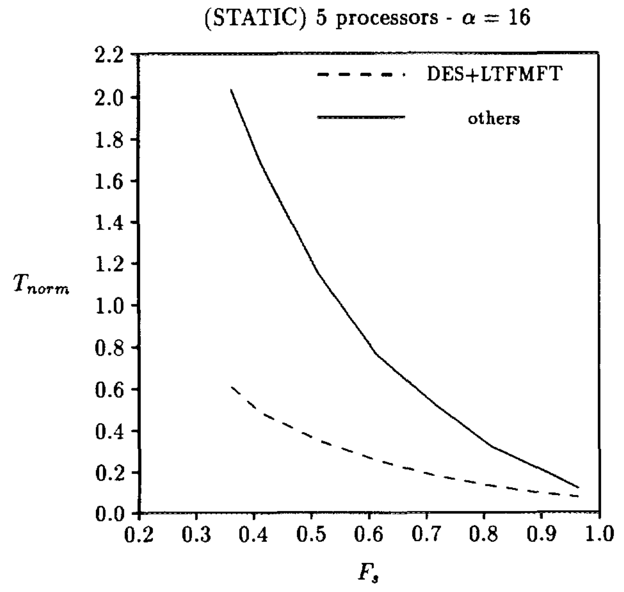

FIG. 8. T_{norm} vs F_s for GRAVITY.

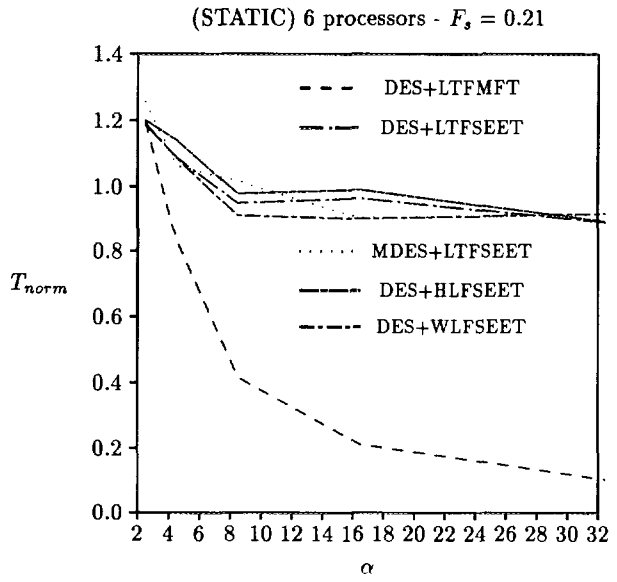

FIG. 10. T_{norm} vs α for MVA.

and are not very sensitive to the degree of heterogeneity of the architecture. On the other hand, DES+LTFMFT is able to take advantage from a heterogeneous architecture by trying to assign tasks to the processors that would make them complete faster, which may not be the fast processor depending on its load. So, the LTFMFT heuristic tries to avoid the potential bottleneck of the fast processor by diverting some tasks to the slow processors provided the job execution time is not increased. Actually, under LTFMFT, the load is more evenly balanced than under any other of the disciplines studied. The results of Fig. 10 are representative of similar results we obtained with all other topologies and values of F_s and number of processors that we experimented with. However, it should be pointed out that as the fraction of sequential processing increases, the difference between the performance of the various disciplines decreases as expected, since most of the execution time will be spent at the sequential portion of the application. This is also illustrated in Fig. 9.

As a final remark on static disciplines, it should be noted that MDES+LTFSEET, DES+WLFSEET, and DES+HLFSEET correspond to the algorithms HNF, WI, and CPM discussed by Shirazi *et al.* [45]. Our results are consistent with their findings for homogeneous environments in that these policies have very similar performance. Actually, of the static policies we investigated, the only one that exhibited a significantly superior performance in the heterogeneous environment was DES+LTFMFT.

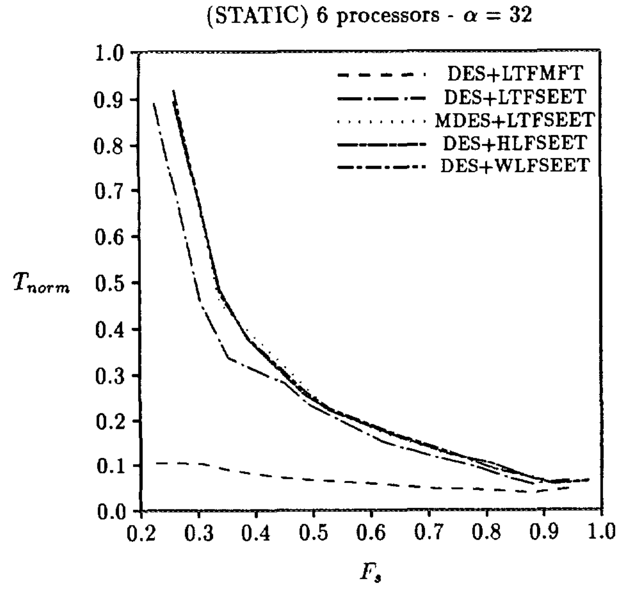

FIG. 9. T_{norm} vs F_s for MVA.

6.2. Dynamic Policies

The first set of curves for the dynamic case shows the variation of the normalized execution time T_{norm} for GRAVITY on a machine with five processors. Figure 11 shows the variation of T_{norm} as a function of the fraction of sequential processing for 4 different values of the processor power ratio α. As it can be seen, for a given value of α we obtain an improvement in the performance (i.e., a decrease in the normalized response time) as the fraction of sequential processing increases. This is expected, since as we increased F_s the sequential bottleneck caused by the synchronization tasks increases. This causes the normalization value (execution time with an infinite pool of slow processors) to increase, forcing T_{norm} down. Note that the rate of decrease of T_{norm} versus F_s is lower for the smallest value of α (2) shown in the figure than for the other values. Note also that after a certain value of α the improvement in T_{norm} is negligible: the curves for $\alpha = 16$ and $\alpha = 32$ are very close to each other. This can be explained by considering the execution of tasks in the fork-and-join con-

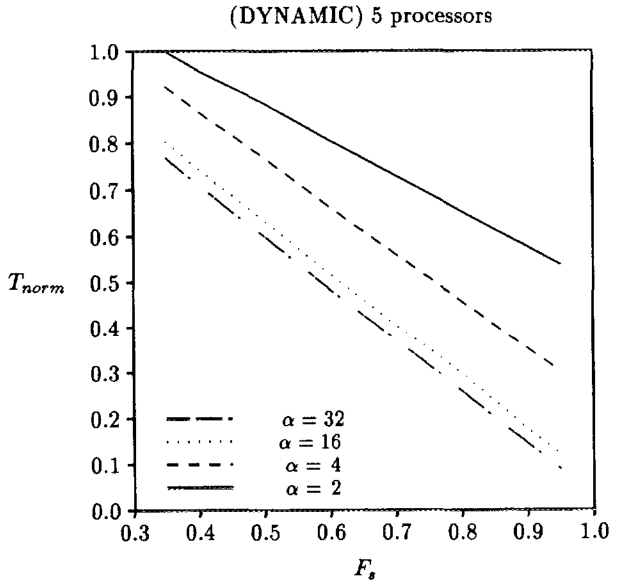

FIG. 11. T_{norm} vs F_s for GRAVITY.

structs of GRAVITY. Since we have five processors, four slow ones will start to execute one task each, while the fast processor will start to work on another task. When α is sufficiently large, the fast processor will be able to finish its task as well as all other tasks in the fork-and-join while the slow processors are still working on their tasks. At this point, there will not be any more tasks for the fast processor and it must remain idle until the slow processors complete their tasks. In other words, the bottleneck is the slow processors on the parallel parts of the job, since the sequential part will not be a bottleneck as α increases. Therefore, there is a point after which it does not pay to increase the speed of the fast processor.

Figure 12 shows the variation of T_{norm} vs α for several values of F_s. All four curves show more clearly the same phenomenon discussed above, namely a decreasing improvement in performance as the processor power ratio increases.

An interesting result is obtained if one compares the execution time of a given application when it is run on a homogeneous machine, T_{hom}, to its execution time, T_{het}, on a *cost-equivalent* heterogeneous one. A cost function is needed to discuss cost equivalence. We use here the constant returns cost function given in [35], which says that the cost of a given machine is proportional to its capacity. Under this assumption, a heterogeneous machine must have $p - \alpha$ slow processors and one fast processor in order to be cost-equivalent to a homogeneous machine with p processors, each identical to the slow one. Let us define the architecture speedup, *Sp*, as the ratio $T_{\text{hom}}/T_{\text{het}}$ [32]. Figure 13 shows the variation of the speedup for the MVA application as a function of the processor power ratio for three different values of F_s. As it can be seen, the higher the value of α the higher the speedup. Also, as the fraction of sequential processing increases, so does the speedup.

A large set of case studies were analyzed with the goal of comparing the impact of the three dynamic processor assignment disciplines discussed in Section 3: FPLTF, FPHLTF, and FPLCTF. The MVA, LU decomposition, and SCENE ANALYSIS jobs were analyzed for various values of the number of processors and various values of the processor power ratio. The three policies resulted in virtually identical performance. The reason was that, in these applications, because of the task dependencies imposed by the task graph, the average number of ready tasks when an assignment decision has to be made is very low. Actually, it is close to one in many cases. This makes all assignment policies behave similarly. In order for some differences to arise, we conducted some experiments with the task graph TG shown in Fig. 7. This graph was purposely designed with the intention to try to explore the difference in the behavior of the three policies. Consider for instance the service demands shown in Fig. 7 inside each task circle. As can be seen, some of the tasks that fork simultaneously have different service demands and different values for *level* and *co-level*. For instance, when tasks 2, 3, and 4 fork, the FPLTF policy would give the fastest processor to task 4, while FPHLTF and FPLCTF would assign it to task 3. When tasks 5, 6, 7, and 8 fork, FPLTF would give priority to either task 6 or 7, while FPHLTF would give priority to task 10, and FPLCTF would chose either task 5 or 8.

In order to compare the three dynamic assignment disciplines, we used the graph of Fig. 7 but with randomly selected service demands for tasks 2 through 12. Recall that the service demand is the *average* task execution time at the fast processor. The actual execution time is assumed to be exponentially distributed. The service demands of tasks 2, 3, and 4 were randomly chosen between 10 and

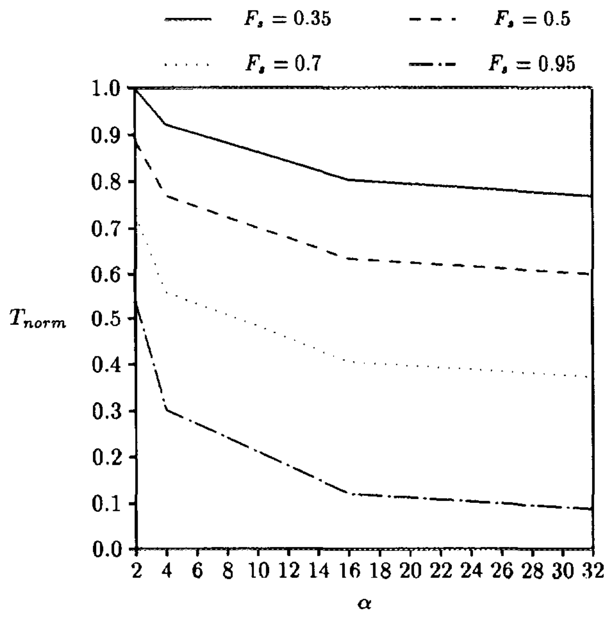

FIG. 12. T_{norm} versus α for GRAVITY (DYNAMIC).

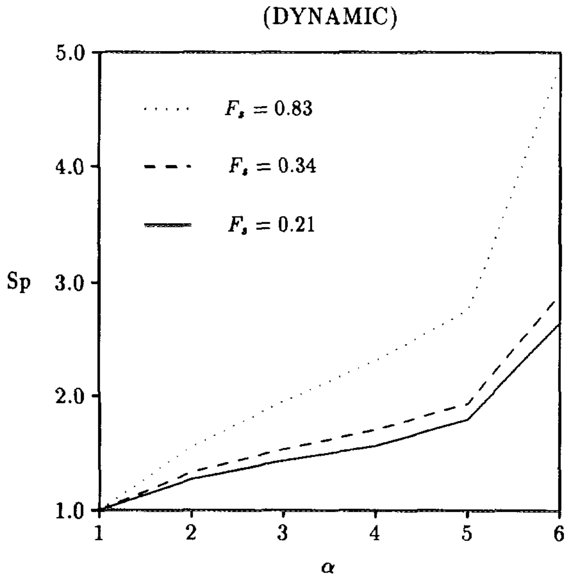

FIG. 13. (MVA) Speedup vs α.

300. Tasks 5 and 8 were assumed to have the same service demand randomly chosen between 10 and 200. The same is true for tasks 6 and 7. Tasks 9 and 10 were assumed to have the same service demand randomly chosen between 3000 and 6000. The same is true for tasks 11 and 12. Over 100 sets of service demands were generated in this way, and the Markov chain algorithm given in Section 5 was used to compare the normalized execution time averaged over all randomly generated graphs as a function of the processor power ratio. Figure 14 depicts the results of this comparison. As it can be seen, the two policies—FPHLTF and FPLCTF—that take into account the topology of the task graph combined with values of service demands perform better than FPLTF, which takes into account only the relative values of the service demands of the tasks that are to be assigned, and not those of their successors. It should be pointed out, however, that the difference in performance is not greater than 10%.

Another interesting comparison can be seen in Fig. 15, which shows the variation of the ratio between the execution time, T_{dyn}, of an application whose tasks are dynamically allocated, and the execution time of the same application assuming that its tasks were statically allocated using DES+LTFMFT (the best) discipline. The topology in this case is MVA, and the dynamic assignment policy is FPLTF, which, for MVA, gives similar results as the others. As it can be seen, only for very small values (close to 2) of α the dynamic assignment disciplines outperform the static ones. For larger values of α, the static assignment discipline exhibits superior performance. This can be explained by the fact that under DES+LTFMFT, it is possible for a ready task to wait for a "better," but busy, processor to run on, even though there is a free processor. Under the dynamic assignment disciplines we investigated, no task waits for a busy processor if there is at least one free processor for it to be assigned to.

7. CONCLUDING REMARKS

Markov chain based models were introduced here as a convenient tool to study the performance of static and dynamic processor assignment disciplines. Although we assumed task execution times to be exponentially distributed, one can also model execution time distributions with coefficients of variation different from one by using either Erlangian or hyperexponential distributions. For instance, a task with execution time distributed according to an Erlangian distribution should be replaced in the task graph by the appropriate sequence of tasks with exponentially

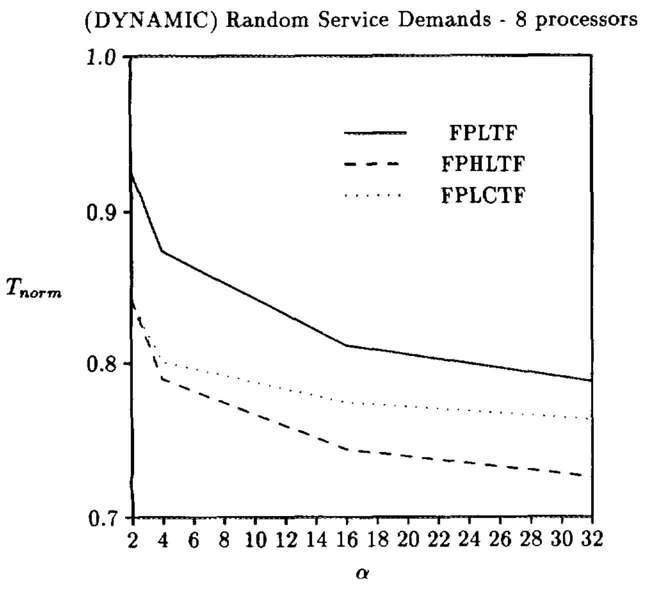

FIG. 14. Normalized response time versus α for various disciplines.

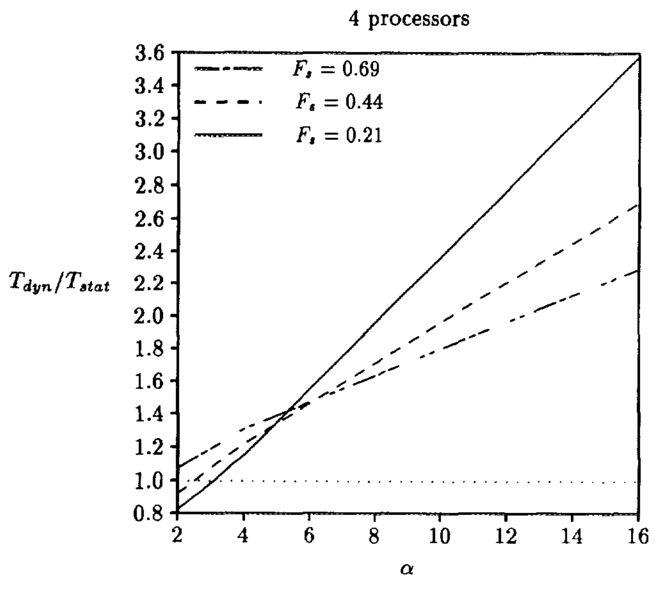

FIG. 15. Static versus dynamic for MVA.

distributed service times. Then the Markov chain analysis algorithm is applied to this modified task graph.

Among the static disciplines analyzed, DES+LTFMFT exhibited much better performance than the others. In particular, its performance improved significantly as the degree of heterogeneity of the architecture increased, making it particularly attractive for this kind of environment. The reason is that LTFMFT tries to avoid the potential bottleneck of the fast processor by diverting some tasks to the slow processors provided the job execution time is not increased. Actually, under LTFMFT, the load is more evenly balanced than under any other of the static disciplines studied. However, it should be pointed out that as the fraction of sequential processing increases, the difference between the performance of the various disciplines decreases as expected.

For the dynamic assignment case, it was shown that the normalized response time decreases with the degree of heterogeneity of the architecture. However, after a certain value of α the improvement in T_{norm} becomes negligible. This can be explained by considering the execution of tasks in a fork-and-join construct. All processors will start to execute a task each simultaneously. When α is sufficiently large, the fast processor will be able to finish its task as well as all other tasks in the fork-and-join while the slow processors are still working on their tasks. At this point, there will not be any more tasks for the fast processor and it must remain idle until the slow processors complete their tasks. In other words, the bottleneck becomes the slow processors on the parallel parts of the job, since the sequential part will not be a bottleneck as α increases. Therefore, there is a point after which it does not pay any more to increase the speed of the fast processor.

Cost-equivalent homogeneous and heterogeneous architectures were compared. It was shown that heterogeneous architectures exhibit an increasing speedup with the value of α. Also, as the fraction of sequential processing increases, so does the speedup.

The MVA, LU decomposition, and SCENE ANALYSIS jobs were analyzed for various values of the number of processors and various values of the degree of heterogeneity of the architecture. For these topologies, the three policies resulted in virtually identical performance. The reason for that is that in these applications, because of the task dependencies imposed by the task graph, the average number of ready tasks when an assignment decision had to be made is very low. Actually, it is close to one in many cases. This makes all assignment policies to behave similarly.

For randomly generated task graphs, it was observed that the two policies—FPHLTF and FPLCTF—that take into account the topology of the task graph combined with values of service demands performed better than FPLTF which only takes into account the relative values of the service demands of the tasks that are to be assigned, and not of their successors. It should be pointed out, however, that the difference in performance was not greater than 10%.

Finally, it was found that the dynamic assignment disciplines outperform DES+LTFMFT, a static discipline, for very small values (close to 2) of α only. For larger values of α, the static assignment discipline exhibits superior performance. This can be explained by the fact that under DES+LTFMFT, it is possible for a ready task to wait for a "better," but busy, processor to run on, even though there is free processor. Under the dynamic assignment disciplines we investigated, no task waits for a busy processor if there is at least one free processor for it to be assigned to.

APPENDIX A. PROOFS OF THEOREMS

Before the following theorems are stated and proved, some definitions are in order:

- $pred^*_G(t_i)$: Set of all immediate and nonimmediate predecessors of task t_i in the task graph G.
- $succ^*_G(t_i)$: Set of all immediate and nonimmediate successors of task t_i in the task graph G.
- $\mathcal{L}_k$: Set of states generated at level k of the algorithm.

Proof of Theorem 1. Let us prove this theorem by induction on k.

Basis of Induction. For $k = 0$, there is only one state in $\mathcal{L}_0$ which is the initial state. Since no tasks completed before the initial state, the theorem holds trivially for $k = 0$.

Induction Hypothesis. Assume the theorem is true for level $k - 1$.

Induction Step. Let **s** be any state in $\mathcal{L}_k$. By construction of the MC, state **s** is obtained from states in $\mathcal{L}_{k-1}$. The transition from any state in $\mathcal{L}_{k-1}$ into state **s** occurs due to the completion of a single task. Then, the number of completed tasks in **s** is equal to the number of completed tasks in any of the predecessors of **s** (which by assumption is equal to $k - 1$) plus 1. Hence, the number of completed tasks in state **s** is equal to $k - 1 + 1 = k$. ■

Proof of Theorem 2. By Theorem 1, $|et(\mathbf{s})| = k$ and $|et(\mathbf{s}')| = j$. Since $j \neq k$, it follows immediately that $et(\mathbf{s}) \neq et(\mathbf{s}')$. ■

Proof of Theorem 3. This theorem will be proved by giving a necessary and sufficient condition for a task to be in $et(\mathbf{s})$. Since this condition only depends on the active set of **s** and since **s**′ has the same active set, the theorem follows immediately.

Let us define the following sets:

1. $S_1 = as(\mathbf{s})$. Clearly tasks in S_1 are executing by definition and therefore $\notin et(\mathbf{s})$.
2. $S_2 = \bigcup_{t_k \in as(\mathbf{s})} succ^*_G(t_k)$. A task in S_2 cannot be in $et(\mathbf{s})$ since the successors of the tasks in the active set of **s** could not have been started already, and thus cannot be completed. This is recursively true for the successors of the successors of the former tasks.

3. $S_3 = \bigcup_{t_k \in as(s)} pred^*_G(t_k)$. Clearly all tasks in S_3 are in $et(\mathbf{s})$ since for all tasks in the active set of state **s** to be active, all their predecessors in G must have completed. This applies recursively to all predecessors of the predecessors.

4. $S_4 = (\bigcup_{t_k \in S_3} succ^*_G(t_k)) - (S_1 \cup S_2)$. This is the set of immediate and nonimmediate successors of all the immediate and nonimmediate predecessors of the tasks in the active set of state **s**, excluding the tasks in the active set of **s** and their immediate and not immediate successors. Let us prove that all tasks in $S_4 \in et(\mathbf{s})$. Assume that there is a task $t \in S_4$ that has not finished. By definition of S_4, t cannot be executing otherwise it would be in $as(\mathbf{s})$. If it is not executing, then it has a predecessor task t' which has not finished. t' cannot be executing since if it were, it would be in $as(\mathbf{s})$ and would belong to S_2, and t would not be in S_4, which is a contradiction. So t' must have at least one immediate predecessor, t'', which did not finish. By the same reasoning, t'' cannot be executing. If we continue applying this argument all the way up in the hierarchy, we will end up concluding that the initial task has not finished yet. But this is a contradiction, since if the initial task had not completed, none of the tasks in $as(\mathbf{s})$ would be executing. So all tasks in S_4 are in $et(\mathbf{s})$.

So, in order for a task t to be in $et(\mathbf{s})$, it is necessary that

$$t \notin (S_1 \cup S_2) \wedge t \in (S_3 \cup S_4). \tag{10}$$

It remains to show that condition (10) is also sufficient for a task t to be in $et(\mathbf{s})$. Assume that there is a task t that completed and that did not satisfy condition (10). Then

$$t \in (S_1 \cup S_2) \vee t \notin (S_3 \cup S_4). \tag{11}$$

Since $S_1 \cap S_2 = \emptyset$, the left hand part of condition (11) is false. So, in order for condition (11) to be true it must be true that t is neither in S_3 nor in S_4. Since t finished, all of its predecessors, including the initial task, finished. The initial task is clearly in S_3. So, since task t is a successor of the initial task and it finished (so it is neither in S_1 nor in S_2), it follows, by definition of S_4, that $t \in S_4$. This shows that it is impossible for t to have finished and not satisfy condition (10).

Hence, condition (10) is a necessary and sufficient condition for a task t to be in $et(\mathbf{s})$. Since all the definitions of sets S_1, S_2, S_3, and S_4 rely only on the active set of state **s** and since state **s**′ has the same active set as state **s**, it follows immediately that $et(\mathbf{s}) = et(\mathbf{s}')$. ■

APPENDIX B. ANALYTIC MODEL OF A FORK-AND-JOIN

This appendix presents an analytic model of a fork-and-join application where n tasks fork and join when all of them have completed. There are p ($p \le n$) processors, and one of them is α times faster than the others. Let this processor be called the fast processor and the remaining ones be called slow processors. It is assumed that the task service demands of all n tasks are identically distributed exponential random variables with mean $1/\mu$.

Let us define a Markov chain whose state (i, j, k) indicates that i tasks are running on slow processors, j ($j = 0, 1$) tasks are running on the fast processor and there are k tasks yet to be completed. Figure 16 shows the state transition diagram for this Markov Chain.

Let $\mathcal{S}$ be a state of this Markov Chain, and consider the following definitions:

- $P_k = Pr[\mathcal{S} = (p - 1, 1, k)]$ for $k = p, \ldots, n$,
- $Q_i = Pr[\mathcal{S} = (i, 1, i + 1)]$ for $i = 0, \ldots, p - 2$, and
- $R_i = Pr[\mathcal{S} = (i, 0, i)]$ for $i = 1, \ldots, p - 1$.

If we write the flow equilibrium equations for this Markov chain we get that

$$P_n = P_{n-1} = \cdots = P_{p+1} \tag{12}$$

$$Q_i = Q_{p-1} \cdot \prod_{j=1}^{p-i-1} \frac{p - j}{p - j - 1 + \alpha}. \tag{13}$$

Since $P_p = Q_{p-1}$, and $P_p = P_n$ we have that

$$Q_i = P_n \cdot F(i) \quad i = 1, \ldots, p - 2, \tag{14}$$

where

$$F(i) = \prod_{j=1}^{p-i-1} \frac{p - j}{p - j - 1 + \alpha}. \tag{15}$$

Now,

$$R_{p-1} = \frac{\alpha}{p - 1} \cdot P_p \tag{16}$$

$$R_{p-2} = \frac{(p - 1)R_{p-1} + \alpha Q_{p-2}}{p - 2}. \tag{17}$$

Substituting (16) into (17) and noting that $P_p = Q_{p-1}$, we may write that

$$R_{p-2} = \alpha \frac{Q_{p-1} + Q_{p-2}}{p - 2}. \tag{18}$$

In general, we have that

$$R_i = \frac{\alpha}{i} \sum_{j=i}^{p-1} Q_j \quad i = 1, \ldots, p - 1. \tag{19}$$

Using (14) in (19) we have that

$$R_i = P_n \frac{\alpha}{i} \sum_{j=i}^{p-1} F(j) \quad i = 1, \ldots, p - 1. \tag{20}$$

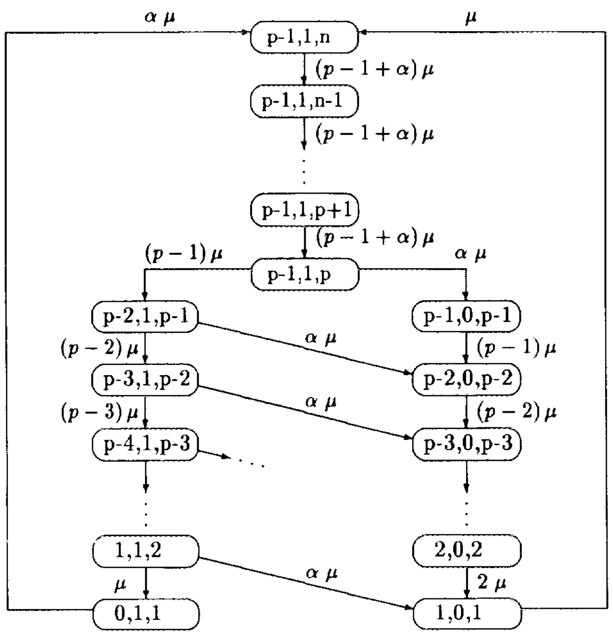

FIG. 16. Markov chain for fork and join.

Also,

$$Q_0 = \frac{1}{\alpha} Q_1 = \frac{1}{\alpha} P_n F(1). \tag{21}$$

So, if we use the fact that all probabilities must sum to one, and use Eqs. (12) to (21) we may solve for P_n:

$$P_n = \left[n - p + 1 + \sum_{i=1}^{p-2} F(i) + \frac{F(1)}{\alpha} + \alpha \sum_{j=1}^{p-1} \frac{1}{j} \sum_{i=j}^{p-1} F(i) \right]^{-1}. \tag{22}$$

The average execution time, T, of the fork and join can be obtained by applying Little's law as we did in Eq. (6). So,

$$T = \frac{1}{(p-1+\alpha)\cdot\mu\cdot P_n}. \tag{23}$$

For $n = p$ (no processor contention) and $\alpha = 1$ (homogeneous case), Eq. (23) reduces, as expected, to

$$T = \frac{1}{\mu} H_n, \tag{24}$$

where H_n is the harmonic number.

The fraction of sequential parallelism, F_s, is simply the sum of the probabilities of the states where there is only one task active. So

$$F_s = Q_0 + R_1 = P_n \cdot \left[F(0) + \alpha \sum_{i=1}^{p-1} F(i) \right]. \tag{25}$$

The average parallelism, A, is given by

$$A = p \cdot P_n \cdot (n - p + 1) + \sum_{i=1}^{p-1} i\,[Q_{i-1} + R_i]. \tag{26}$$

REFERENCES

1. Adam, T. L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. *J. Assoc. Comput. Mach.* **17,** 12 (Dec. 1974).
2. Almeida, V. A., Vasconcelos, I. M. M., and Árabe, J. N. C. The effect of heterogeneity on the performance of multiprogrammed parallel systems. *Proc. Workshop on Heterogeneous Processing.* Beverly Hills, CA, 1992.
3. Almeida, V. A., and Vasconcelos, I. M. M. A simulation study of processor scheduling policies in a multiprogrammed parallel system. *Proc. 1991 Summer Computer Simulation Conference (SCSC 91).* Baltimore, 1991.
4. Amdahl, G. Validity of the single processor approach to achieving large scale computing capability. *Proc. AFIPS Spring Joint Computer Conference 30.* 1967.
5. Andrews, J. B., and Polychronopoulos, C. D. An analytical approach to performance/cost modeling of parallel computers. *J. Parallel Distrib. Comput.* **12** (Aug. 1991), 343–356.
6. Athas, W., and Seitz, C. Multicomputers: Message-passing concurrent computers. *IEEE Comput.* **21,** 8 (Aug. 1988).
7. Baxter, M. J., Tokhi, M. O., and Fleming, P. J. Heterogeneous architectures for real-time control systems: Design tools and scheduling issues. *Proc. 12th IFAC Workshop on Distributed Computer Control Systems.* 1994.
8. Black, D. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Comput.* **23,** 5 (May 1990).
9. Chu, W. W., and Leung, K. K. Module replication and assignment for real-time distributed processing systems. *Proc. IEEE* (May 1987), 547–562.
10. Chu, W. W., Sit, C., and Leung, K. K. Task response time for real-time distributed systems with resource contentions. *IEEE Trans. Software Engrg.* **17,** 10 (Oct. 1991).
11. Dowdy, L. W. On the partitioning of multiprocessor systems. Technical Report 88-06, Department of Computer Science, Vanderbilt University, 1988.
12. Dussa, K., Carlson, B., Dowdy, L. W., and Park, K. Dynamic partitioning in a transputer environment. *Proc. ACM Sigmetrics Conference.* 1990.
13. Ercegovac, M. D. Heterogeneity in supercomputer architectures. *Parallel Comput.* **7** (1988), 367–372.
14. Flatt, H., and Kennedy, K. Performance of parallel processors. *Parallel Computing* **12** (1989).
15. Freund, R. F. Optimal selection theory for superconcurrency. *Proc. Supercomputing '89.* IEEE Computer Society/ACM Sigarch, Reno, NV, 1989, pp. 699–703.
16. Freund, R. R., and Conwell, D. S. Superconcurrency, a form of distributed heterogeneous supercomputing. *Supercomput. Rev.* (Jun. 1990).
17. Gajski, D., and Peir, J. Essential issues in multiprocessors. *IEEE Comput.* **18,** 6 (Jun. 1985).
18. Ghosal, D., Serazzi, G., and Tripathi, S. K. Processor working set and its use in scheduling multiprocessor systems. *IEEE Trans. Software Engrg.* (May 1991).
19. Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods of the performance of parallel applications. *Proc. ACM Sigmetrics Conference.* 1991.

20. Kapelnikov, A., Muntz, R. R., and Ercegovac, M. D. A modeling methodology for the analysis of concurrent systems and computations. *J. Parallel Distrib. Comput.* **6** (1989), 568–597.
21. Kasahara, H., and Narita, S. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Comput.* **33,** 11 (Nov. 1984).
22. Leutenegger, S., and Vernon, M. The performance of multiprogrammed multiprocessor scheduling policies. *Proc. ACM Sigmetrics Conference.* 1990.
23. Little, J. D. C. A proof for the queuing formula: $L = \lambda W$. *Oper. Res.* **9** (1961), 383–389.
24. Majumdar, S., Eager, D., and Bunt, R. Scheduling in multiprogrammed parallel systems. *Proc. ACM Sigmetrics Conference.* 1988.
25. Majumdar, S. Processor scheduling in multiprogrammed parallel systems. Ph.D. Thesis, Department of Computational Science, Research Report 88-6, University of Saskatchewan, 1988.
26. McCann, C., and Zahorjan, J. Processor allocation policies for message-passing parallel computers. *Proc. 1994 ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems.* Nashville, TN, 1994.
27. McCann, C., Vaswani, R., and Zahorjan, J. A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors. Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington, 1991.
28. Menascé, D. A., da Silva Porto, S. C., and Tripathi, S. K. Static heuristic processor assignment in heterogeneous multiprocessors. *Int. J. High Speed Comput.* **6,** 1 (Mar. 1994), 115–137.
29. Menascé, D. A., and da Silva Porto, S. C. Scheduling on heterogeneous message passing architectures. *J. Comput. Software Engrg.* **1,** 3 (1993).
30. Menascé, D. A., and Barroso, L. A. A methodology for performance evaluation of parallel applications in shared memory multiprocessors. *J. Parallel Distrib. Comput.* **14,** 1 (Jan. 1992).
31. Menascé, D. A., and Almeida, V. A. F. Heterogeneous supercomputing: Why is it cost-effective? *Supercomput. Rev.* **4,** 8 (Aug. 1991).
32. Menascé, D. A., and Almeida, V. A. F. Heterogeneity in high performance computing. *Proc. 2nd Symposium on High Performance Computing* (M. Durand and F. El Dabaghi, Eds.). Elsevier, Montpellier, France, 1991.
33. Menascé, D. A., da Silva Porto, S. C., and Tripathi, S. K. Processor assignment in heterogeneous parallel architectures. *Proc. International Parallel Processing Symposium (IPPS'92).* Beverly Hills, CA, 1992.
34. Menascé, D. A., and Almeida, V. A. Cost–performance analysis of heterogeneity in supercomputer architectures. *Proc. ACM-IEEE Supercomputing '90 Conference.* New York, 1990.
35. Mendelson, H. Economies of scale in computing: Grosh's law revisited. *Comm. ACM* **30,** 12 (Dec. 1987), 1066–1072.
36. Nelson, R., and Towsley, D. Comparison of threshold scheduling policies for multiple server systems. IBM Research Report RC 11256 (#50704), July 1985.
37. Nirkhe, V., and Pugh, W. A partial evaluator for the maruti hard real-time system. *Proc. of the IEEE Real-Time Systems Symposium.* San Antonio, TX, 1991.
38. Park, K., and Dowdy, L. W. Dynamic partitioning of multiprocessor systems. *Int. J. Parallel Programming* **18,** 2 (1989).
39. Park, C., and Shaw, A. C. Experiments with a program timing tool based on source-level timing schema. *Proc. of IEEE Real-Time Systems Symposium.* Lake Buena Vista, FL, 1990.
40. Polychronopoulos, C. D. Multiprocessing vs. multiprogramming. *Proc. 1989 International Conference on Parallel Processing.* 1989.
41. Polychronopoulos, C. D. Parallel programming issues. *Proc. 9th International Conference on Computational Sciences and Engineering.* Paris, 1990.
42. Polychronopoulos, C. D. The hierarchical task graph and its use in auto-scheduling. *Proc. of the 1991 International Conference on Supercomputing.* Cologne, 1991.
43. Seager, M., and Stichnoth, J. Simulating the scheduling of parallel supercomputer applications. Technical Report UCRL 102059, Lawrence Livermore National Laboratory, 1988.
44. Sevcik, K. C. Characterization of parallelism in adaptation and their use in scheduling. *ACM Sigmetrics Performance Rev.* **17,** 1 (May 1989).
45. Shirazi, B., Wang, M., and Pathak, G. Analysis and evaluation of heuristic methods for static task scheduling. *J. Parallel Distrib. Comput.* **10** (Nov. 1990), 222–232.
46. Shmoys, D. B., Wein, J., and Williamson, D. P. Scheduling parallel machines on-line. *Proc. of the 32nd Annual Symposium on Foundations of Computer Science.* IEEE Comput. Society, San Juan, Puerto Rico, 1991.
47. Squillante, M. Issues in shared memory multiprocessor scheduling: A performance evaluation. Ph.D. Dissertation, Department of Computer Science and Engineering, University of Washington, 1990.
48. Squillante, M., and Lazowska, E. Using processor cache affinity information in shared-memory multiprocessor scheduling. Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, 1990.
49. Thomasian, A., and Bay, P. F. Analytic queueing network models for parallel processing of task systems. *IEEE Trans. Comput.* **C-35,** 12 (Dec. 1986).
50. Tripathi, S. K., and Ghosal, D. Processor scheduling in multiprocessor systems. *Proc. First International Conference of the Austrian Center for Parallel Computation.* Springer-Verlag, Berlin/New York, 1991.
51. Wang, M.-C., Kim, S.-D., Nichols, M. A., Freund, R. F., Siegel, H. J., and Nations, W. G. Augmenting the optimal selection theory for superconcurrency. *Proc. Workshop on Heterogeneous Processing.* Beverly Hills, CA., 1992.
52. Vaswani, R., and Zahorjan, J. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. Technical Report No. 91-03-03, Department of Computer Science and Engineering, University of Washington, 1991.
53. Zahorjan, J., Lazowska, E., and Eager, D. The effect of scheduling discipline on spin overhead in shared memory multiprocessors. Technical Report 89-07-03, Department of Computer Science, University of Washington, 1989.

DANIEL A. MENASCÉ is a professor of computer science and the Associate Director of the Center for the New Engineer at George Mason University. Prior to that, he was a faculty at the Pontifícia Universidade Católica, Rio de Janeiro, for 14 years, where he also served as department chair. Professor Menascé received the Ph.D. degree in computer science from the University of California at Los Angeles in 1978, and the M.Sc. in computer science and a B.S.E.E., both from the Pontifícia Universidade, Católica in Rio de Janeiro, in 1975 and 1974, respectively. Dr. Menascé has been actively involved in research related to distributed systems, performance evaluation, and high performance computer systems.

DEBANJAN SAHA received a B.Tech. degree in computer science and engineering from the Indian Institute of Technology in 1990, and an M.S. degree in computer science from the University of Maryland, College Park, in 1992. He is currently a Ph.D. candidate in the Computer Science Department at the University of Maryland. He is working on multimedia transport on ATM networks for his dissertation. He is also interested in high performance computing and fault tolerance.

STELLA C. S. PORTO is an assistant professor at the Federal University in Niteroi, Brazil. She received the M.Sc. degree in computer science from the Pontifícia Universidade Católica, Rio de Janeiro, in 1991, and she is a Ph.D. candidate at the same institution. Her research interests include parallel processing and multiprocessor scheduling.

VIRGILIO A. F. ALMEIDA is a professor of computer science at the Federal University of Minas Gerais, Brazil. He received the Ph.D. degree in computer science from Vanderbilt University in 1986, and the M.Sc. degree, also in computer science from the Pontifícia Universidade Católica, Rio de Janeiro, in 1980. His research interests include high performance computing, performance evaluation, and capacity planning.

SATISH K. TRIPATHI is a professor in and the Chairman of the Department of Computer Science at the University of Maryland, College Park. He attended the Banaras Hindu University, the Indian Statistical Institute, the University of Alberta, and the University of Toronto. He received his Ph.D. in computer science from the University of Toronto. He has been on the faculty at the University of Maryland since 1978. For the last fifteen years he has been actively involved in research related to performance evaluation, networks, real-time systems, and fault tolerance. Dr. Tripathi has served as a member of the Program Committee and as Program Chairman for various international conferences.

Received June 14, 1992; accepted February 19, 1993