The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems[‡]

Siu-Lun Au[†]
Newbridge Networks Corporation
600 March Road
Kanata, Ontario K2K 2E6, Canada
terrence_au@newbridge.com

Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

Abstract

A simple fork and join type of job structure has been extensively used for performance evaluation of processor scheduling policies in multiprocessor systems. However, parallel programs often exhibit a more complicated structure. It is not clear how the program structure affects the performance of processor scheduling policies. This paper studies the impact of the program structure on the performance of processor scheduling policies that are appropriate for uniform memory access (UMA) shared-memory systems.

We consider four types of parallel program structures that are frequently employed in parallel applications. These are the fork-and-join, divide-and-conquer, Gaussian elimination, and state space search programs. The impact of these four job structures on the performance of four processor scheduling policies is considered. These are the job-based round robin (RRJob), coscheduling, dynamic (Dynamic), and the preemptive smallest cumulative demand first (PSCDF) policies. These policies cover a variety of policies – preemptive vs. non-preemptive, space-sharing vs. time-sharing, *a priori* job knowledge vs. no job knowledge. The PSCDF policy is not implementable but is included for comparison purposes only. The results presented here show that the time-sharing processor scheduling policy RRJob performs as well as or better than the space-sharing Dynamic policy. This result contradicts the results of some previous studies and we clearly identify the problems associated with these previous studies. Thus, the major contribution of this paper is to correct the prevailing intuition that dynamic space-sharing outperforms time-sharing and show that, for the workloads considered here, time-sharing policies are useful for multiprocessor systems.

Key words: Processor scheduling, shared-memory multiprocessors, performance evaluation, time-sharing, space-sharing.

1. Introduction

Processor scheduling in shared-memory multiprocessor systems has received considerable attention in recent years. A wide variety of scheduling algorithms has been proposed in the literature. Performance evaluation of these policies is often done by means of simulation. A select few are actually implemented. From the performance evaluation point of view, implementations are preferred because they take into account all system and software overheads, thus leading to a more accurate performance evaluation. However, implementations are often expensive and involve modifying the kernel. The coding effort can be reduced by implementing the scheduling policies in the application layer [13]. The disadvantage of this scheme is that the overheads are different than when implemented in the kernel. Simulation modelling is often used to evaluate the performance of processor scheduling policies.

The accuracy of simulation modelling depends on several factors. One of the key components that significantly affects the performance of a scheduling policy is the workload model, which refers to the program or job characterization used in the simulation. The parallelism in a job, which is of interest here, can be characterized at a number of different levels of detail [16]. Several analytical and simulation studies have assumed a simple *fork-and-join* type of job structure [10,12,14,17]. In this model, a job is assumed to be composed of a set of independent tasks that can all be executed on the system concurrently. A job is said to be complete when all of its component tasks are completed. Tasks within a job do not communicate with each other; all the tasks of a job participate in the synchronization phase at the end. However, parallel programs often exhibit a more complicated structure. It is not clear how the program structure affects the performance of the processor scheduling policies. This paper studies the impact of the program structure on the performance of processor scheduling policies that are appropriate for the uniform memory access (UMA) shared-memory systems (and not for the non-uniform memory access (NUMA) shared-memory systems and distributed-memory systems).

[‡]This paper appears in *Int. J. of Computers and Their Applications,* 1996.

[†]This work was done while the author was at the School of Computer Science, Carleton University.

We consider four types of parallel program structures, including the fork-and-join structure, that are frequently employed in parallel applications. These are the fork-and-join, divide-and-conquer, Gaussian elimination, and state space search programs. Section 2 presents details on these four program structures. Section 2 also presents the workload model used in the simulation.

We consider four processor scheduling policies proposed in the literature. These are the job-based round robin (RRJob), coscheduling (Coscheduling), dynamic (Dynamic), and preemptive smallest cumulative demand first (PSCDF) policies. These policies cover a wide spectrum of policies – preemptive vs. non-preemptive, space-sharing vs. time-sharing. The PSCDF policy is not implementable but is included for comparison purposes only. Details on these policies are presented in Section 3.

The results of the simulation experiments are presented in Section 4. These results show that the job structure has a significant impact on the performance of the space-sharing Dynamic policy. The results presented here show that the time-sharing processor scheduling policy RRJob performs as well as or better than the space-sharing Dynamic policy. This is in contrast to the results reported in the literature that showed that the Dynamic is better than the RRJob [13,19]. The previous results have not included the job waiting time in the response time calculation. However, when the job waiting time is included in the response time calculation, RRJob performs as well as or better than Dynamic policy. In particular, the performance superiority of RRJob increases with increasing variance in job service demand and increased parallelism in jobs. The main reason for this is that Dynamic uses FCFS processor allocation and tends to allocate larger processor share to bigger jobs. Therefore, despite the recent results, time-sharing processor scheduling policies are useful for multiprocessor systems.

2. Job and Workload Characterization

This section presents the four types of job structures (Section 2.1) and the workload model (Section 2.2) used in this study.

2.1. Types of Parallel Programs Considered

Details about the four types of parallel programs – fork-and-join, divide-and-conquer, Gaussian elimination, and state space search – are presented in this section.

2.1.1. Fork-and-Join Programs

In this type of algorithms, work can be decomposed into independent sub-computations. A simple example is the algorithms that consist of a single loop and each instance of the loop can be executed independently. The parallel implementation of these types of algorithms can be done by creating tasks to work on the sub-computations. The structure of this type of parallel job is shown in Figure 2.1, which shows one phase of the fork-and-join structure.

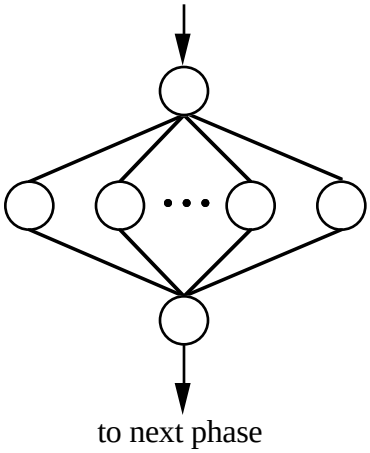


Figure 2.1 The fork-and-join job structure

The fork-and-join (FJ) job structure, for example, is reasonable for the class of problems with a solution structure that iterates through a communications phase and a computations phase. This class is exemplified by the n-body simulations of stellar or planetary movements, in which the movement of each body is governed by the gravitational forces produced by the system as a whole [8]. In this case, the model used here can be considered to represent one computation phase [6]. In this paper, only a single phase that involves one synchronization point is assumed. Similar job structure has been used in several previous studies [3,4,5,10,12,14,17].

2.1.2. Divide-and-Conquer Programs

Implementation of the divide-and-conquer (DC) algorithms on parallel systems has received significant attention in the last decade (for example, see [7]). Large numbers of parallel programs have been developed that are based on this strategy, especially in the area of symbolic computation, where the computation works on a discrete data structure. In general, the divide-and-conquer strategy does not restrict the number of sub-problems into which a given problem is partitioned. However, for convenience, we focus only on the binary case, where a given problem is divided into two sub-problems. Thus, the parallelism in these programs steps through the sequence 2^0, 2^1, . . ., 2^{n-1}, 2^n, 2^{n-1}, . . ., 2^1, 2^0 during their execution.

The parallel implementation of divide-and-conquer algorithms exhibits a partitioning structure shown in Figure 2.2 [11]. The tasks created by this type of programs perform one of the following three operations: 1) splitting the inputs, 2) working on the inputs sequentially and 3) combining the outputs. Tasks that divide the input structures are called *DIVIDE* tasks. Tasks that work on the input structures are called *WORK* tasks. Tasks that combine the output are called *MERGE* tasks. Generally, the execution time of the tasks decreases level by level over the upper half of the structure, and increases level by level over

the lower half of the structure. Synchronization is done at the *MERGE* stage. For some applications, such as the quick sort, MERGE stage does not involve much work; it simply provides synchronization. On the other hand, some applications, for example database queries, DIVIDE phase is trivial and MERGE nodes involves work (for example, sorting the answer). Efficient parallel implementations of divide-and-conquer algorithms have been developed in many areas of application, such as, database management, numerical integration, computational geometry and VLSI circuit design and combinatorial problems (see [1] for a detailed survey).

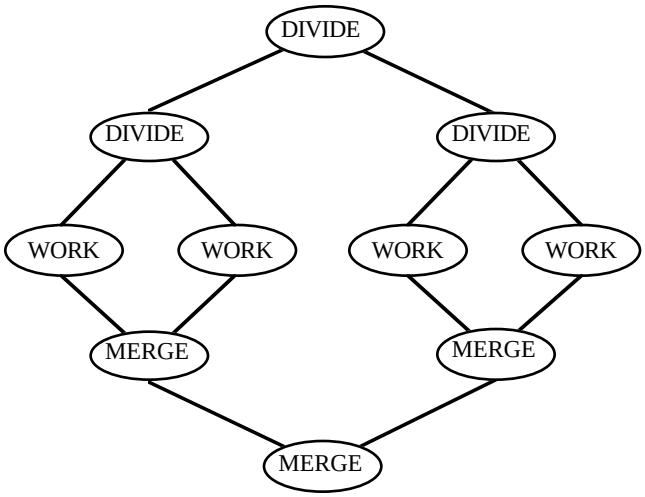


Figure 2.2 The divide-and-conquer job structure

2.1.3. Matrix Factorization Programs

To solve a matrix equation of the form $\mathbf{A}x = b$, one can decompose **A** into upper **U** and lower **L** triangular matrices and then solve it by solving $\mathbf{L}y = b$ and $\mathbf{U}x = y$. The most computationally intensive part of the algorithm is to find the LU decomposition. For this reason, parallelizing the LU decomposition has been studied by a number of researchers. One approach is to derive a task system for the LU decomposition algorithm and perform static scheduling on this task system. Other related algorithms include the Cholesky factorization and Gauss-Jordan algorithms. The structure of these parallel programs can be represented by a task system shown in Figure 2.3. T_i^j represents the task that works on column i in the j^{th} iteration (see [1] for details).

However, the actual function of the tasks are different for each of the above mentioned algorithms. In the LU decomposition and Gauss-Jordan algorithms, T_k^k finds the pivot element. For the Cholesky factorization algorithm, T_k^k computes the square root of the diagonal element and then divides the lower part of column k by this square root. In all the three algorithms, T_k^j uses the elements of column k to modify those in column j. Since matrix factorization is a way of performing Gaussian elimination, this class of programs is referred to as the Gaussian elimination (GE) programs.

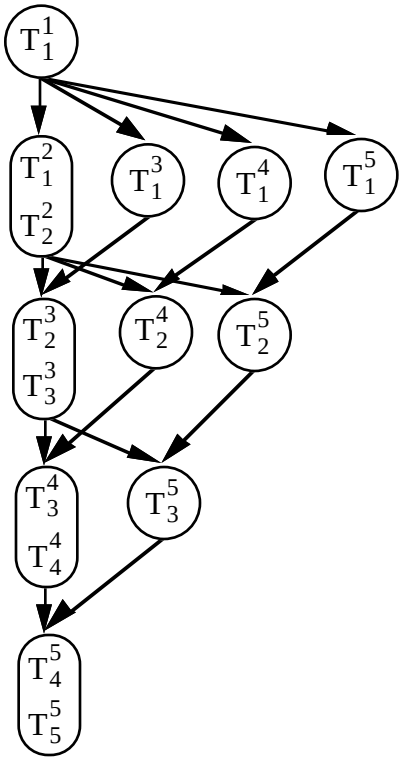


Figure 2.3 Matrix factorization job structure

2.1.4. State Space Search Programs

Searching is a common method for solving problems in artificial intelligence. In general, a search system consists of three main components. The first of these is a *database*, which describes both the current state and the goal. Consider the problem of proving a theorem. In this case, the current state consists of the assertions representing the axioms, lemma, and semantic networks. The goal is an assertion representing the theorem to be proved. The second component of the search system is a set of operators that are used to manipulate the database. In the case of a theorem-proving system, operations are the rules of inference such as modus ponens or resolution. The third component is the control strategy for deciding the next action, what operator to apply and where to apply it. The choice of a control strategy affects the contents and organization of the database. The process of problem solving is to bring the problem state forward from its initial configuration to one satisfying a goal condition by repeatedly applying an operator.

The search system mentioned above advances the search by applying an operator to produce a new state in the database. A *state-space representation* of a problem employs two kinds of entities: *states*, which are data structures that give "snapshots" of the problem at each stage of the problem solving, and *operators*, which are applied to transform the problem from one state to another. Breadth-first strategies, for example, can be used to accomplish this. Some canonical examples are the eight queen and block movement problems.

To accomplish the implementation of the breadth-first search in parallel, at each level of the search, tasks are created that correspond to every node of the state-space graph. The search for the state-space graph continues until the final state is reached. Once the final state is reached, the algorithm enters the *wait & abort* stage where the tasks that do not yield the results will be aborted. The implementation of the breadth-first search is similar to the divide-and-conquer algorithms, except that the merge tasks in this case also perform task abortion when needed. A task system which gives a high-level view of the structure of the parallel breath-first search is shown in Figure 2.4. Since this class of programs are essentially performing the state space search on a graph, these are referred to as the state space search (SSS) programs. As with the divide-and-conquer programs, we assume that each *expand* node generates just two further states.

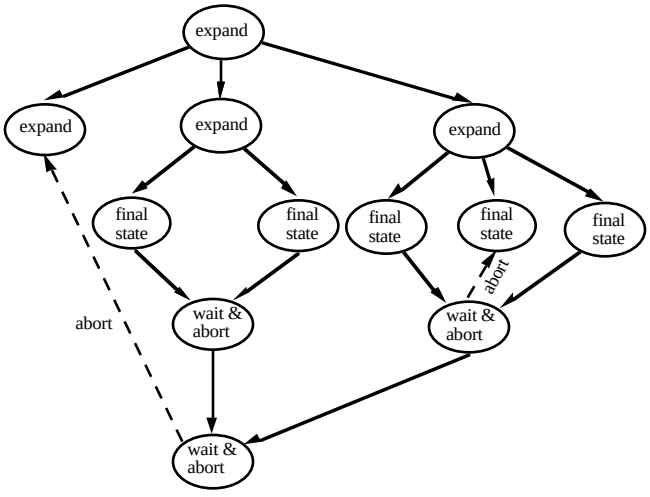


Figure 2.4 State space search job structure

2.2. The Workload Model

This section presents an abstract workload model for parallel programs. The abstract model consists of a set of input parameters (listed in Table 2.1) along with a job structure. We have examined several parallel algorithms to derive the job structure and the workload model described in this paper. For the sake of brevity, these details are not included here (interested reader is referred to [1]).

The job arrival process is characterized by two parameters: λ and CV_a. The job arrival rate is represented by λ and CV_a represents the coefficient of variation of the inter-arrival times. For all the results reported in this paper, CV_a is set to 1 (i.e., inter-arrival times are exponentially distributed).

Table 2.1 Input parameters for the simulation model

Parameter	Description
mp	maximum parallelism
CV_{mp}	coefficient of variation of maximum parallelism
d	mean service demand
CV_d	coefficient of variation of service demand
P_T	probability of task abortion
λ	arrival rate of jobs
CV_a	coefficient of variation of inter-arrival times

Maximum Parallelism (*mp*)

The maximum parallelism of a parallel program refers to the maximum number of tasks that can be run in parallel, if an unlimited number of processors is provided. Since we assume that the tasks in a fork-and-join job can be executed independently, its maximum parallelism is always equal to the number of tasks the job has. For the divide-and-conquer job model, the maximum parallelism is achieved when the job is in its work stage. For the Gaussian elimination job model, the maximum parallelism is at the second stage of the program, where independent tasks are created to work on each column of the matrix. However, as the execution progresses, the number of parallel tasks decreases as the number of columns to be solved decreases. For the parallel state-space search jobs, maximum parallelism is reached when the search enters the final state.

Mean Service Demand (*d*)

There are two ways to model the service demand of a parallel program [9,10,12]. One is to compute the overall service demand of a parallel program and distribute the overall service demand among the tasks of the program. We can also compute the service demand of each task of a parallel program and sum them up to get the overall service demand. The workload created from the first approach is called *uncorrelated* because the overall service demand of a parallel program is independent of the number of tasks. On the other hand, the model based on the second approach is called

correlated because the overall service demand of a job is directly proportional to the number of tasks.

For the uncorrelated workload, the parameter D represents the mean service demand of a parallel program. For example, if it is set to 20, then, on average, a parallel program will require 20 units of service time from the system. For the correlated workload, d is the mean service demand of the tasks in the parallel programs. For example, if d is equal to 1, then, on average, every job has a mean task service time equal to 1 time unit. The mean task service time is the sum of the service time of the tasks divided by the number of tasks in the job.

Coefficient of Variation of Service Demand (CV_d)

CV_d is the coefficient of variation of service demand of parallel programs, if the uncorrelated workload is used. It is the coefficient of variation of service demand of the tasks, if the correlated workload is used.

Probability of Task Abortion (P_T)

The probability of task abortion P_T is only used in the state space search job model. It is used to decide whether a task should be aborted by the *wait & abort* tasks.

These parameters along with a job structure defines the workload. The four types of job structures considered in the paper are the fork-and-join (Figure 2.1), divide-and-conquer (Figure 2.2), Gaussian elimination (Figure 2.3), and the state-space search (Figure 2.4).

3. Processor Scheduling Policies

This section discusses the scheduling policies used in the simulation experiments. These are the round robin job (RRjob) policy [10], the dynamic partition (Dynamic) policy [19] and the coscheduling (Coscheduling) policy [15]. Each of these policies has advantages and disadvantages. In addition, we will also evaluate the performance of the preemptive smallest cumulative demand first (PSCDF) policy. The PSCDF policy provides good performance and can serve as a benchmark for comparison. RRjob aims at providing an equal share of the processing power to the jobs in the system, regardless of their number of tasks. This is done by varying the quantum size for each job in the system. Since RRjob provides equal processor allocation per job, it results in good performance in multiprocessor systems [10]. Dynamic policy has the same objective as the RRjob in that it tries to make the jobs share the processing power equally [10]. In addition, Dynamic policy also aims at minimizing the number of context switches by reacting dynamically only to the changes in the job's parallelism. If the tasks of the same job require coordination among them, it would be better if these tasks can be scheduled to run at the same time. The Coscheduling policy is designed in light of this observation [15]. Even though the Coscheduling policy is shown to perform worse than the Dynamic and RRjob policies, the Coscheduling policy is considered in this paper because of the presence of synchronization in the workload. In the subsequent sections, these four scheduling policies are described briefly (a more detailed description is given in [1]).

3.1 The Coscheduling Policy

It is assumed that a linked list is used to implement the task queue. A processor window of size equal to the number of processors in the system is moved along the linked list. When a job arrives into the system, its tasks are appended to the end of the linked list. At each quantum q, tasks located in the processor window are scheduled to run. Some tasks may have service demand less than the quantum. When these small tasks are completed, their processors remain idle until the end of this quantum. All processors do the context switch at the same time. At the end of each quantum, the processor window is moved along the linked list until the head of the window is at the first task of the job that is not completely coscheduled in the previous quantum. A job is completely coscheduled, if all of its tasks have been scheduled to run in the previous quantum.

However, this definition cannot be applied to the fork-and-join parallel jobs when the number of tasks T is greater than the number of processors P in the system. For the fork-and-join parallel jobs consisting of $T > P$ tasks, it is impossible to schedule all the tasks of the job in one quantum. In this case, the job is completely coscheduled through the schedules in the previous consecutive quanta. The basic definition of complete coscheduling may not be applicable to the job models where new tasks can be created dynamically, as the tasks of the same job may not be grouped together in the queue. For these job models, a weak definition of complete coscheduling is adapted. A job is considered to be completely coscheduled, if a group of its tasks that has the same ancestor is completely coscheduled. This definition does not violate the spirit of coscheduling, because typically only the processes that are created by the same task are required to synchronize with each other.

3.2 The Preemptive Smallest Cumulative Demand First Policy

A priority queue is used to implement this policy. When a new job arrives, the system will preempt processors that are currently running the tasks of jobs having larger cumulative demand than the new job. Thus, jobs with small service demand have higher priority than those with a larger service demand. Since the service demand of the parallel divide-and-conquer jobs changes during their execution, due to task creation and termination, we use their total service demand as their job size. By total service demand, we mean the sum of the service demand of all the tasks of the given job. Their total service demand decreases when their tasks are completed. In reality, this policy cannot be implemented, as

the total service demand of a job is not known in advance. As in the previous studies [10,12], we have included this policy for comparison purposes only.

3.3 The Round Robin Job Policy

The RRjob policy was originally proposed in [12] and its performance is reported in [10]. The goal of the RRjob policy is to provide equal allocation of processors to jobs temporally. This is done by giving a longer time slice to jobs with a small number of tasks. In addition, RRjob is a dynamic scheduling policy in that the processor allocation changes dynamically.

A shared queue of parallel jobs is used to implement this policy. Each entry in the queue consists of a list of tasks of the job. Scheduling is done round robin on this queue. Tasks of the first job in the queue are scheduled to run, when there is at least one processor available. If there are M processors available, where $M \leq P$ (the number of processors in the system) and the job at the front of the queue (say job f) has N tasks waiting for service, where $N \leq M$, these N tasks are scheduled to run in the next quantum. If $N<M$, the remaining $M-N$ processors will be given to the next job on the queue. On the other hand, if $N \geq M$, only M tasks will be scheduled and the remaining tasks will have to wait for the next quantum of job f.

The quantum size of this policy varies from job to job and it depends on the number of tasks the job has. The quantum size is equal to $(P * q)/N$, where q is the unit size of the quantum, if the number of tasks in the job is less than P; otherwise, the quantum size is equal to q.

3.4 The Dynamic Policy

The dynamic policy is a two-level scheduling policy [10,19]. At the upper level, processors are partitioned among the jobs in the system. At the lower level, the task dispatcher assigns processors to the parallel jobs. The task dispatcher may release processors when the parallel jobs do not have enough parallelism to utilize them. The actions of the processor allocator are described as follows.

When a job requests one or more processors, the following action is taken [19]:

1. If there are idle processors, use them to fulfill the request.
2. Otherwise, if the job making the request is a new job, allocate it a single processor by preempting a processor from a job that currently owns more than two processors.
3. If any part of a request for processors cannot be satisfied, it remains outstanding until either a processor becomes available or the task dispatcher releases processors from other parallel jobs.

When a parallel job releases its processors, the processor allocator will search for jobs that have not yet been assigned a processor (i.e., the unsatisfied new arrivals) and assign a processor to each of these jobs. If there are more free processors after this allocation, scan the queue again, and allocate the rest of the processors on a FCFS basis.

When a processor finishes a task, the dispatcher will be invoked to either assign another task of the same job to the idle processor or hand it over to the processor allocator. The function of the dispatcher is described as follows. Let q_j be the number of job j's tasks waiting for processors.

4. If there is a newly arrived job that has not been allocated a processor, give the processor to this job.
5. Otherwise, if $q_j = 1$, the dispatcher simply dequeues the task and starts its execution.
6. If $q_j>1$ the task dispatcher will begin executing the first task in the queue.
7. Finally, if $q_j = 0$, the dispatcher will release its processors to the allocator.

4. Performance Comparison

This section presents the results of the simulation experiments to evaluate the performance sensitivity of the four scheduling policies, discussed in the last section, to the job structure. The job models given in Section 2 are used. All the simulations are parameterized for a system with 20 processors (i.e., $P = 20$). The quantum length q for Coscheduling and RRJob policies is equal to 0.1 time units. If time units are taken as seconds, this implies $q = 100$ msec, which is the quantum used by DYNIX. The context switch overhead is assumed to be 1.5% of the quantum q, that is 0.0015 time units. The context switch overhead also applies to the PSCDF and Dynamic policies, when a processor is preempted from one task to another. Note that this context switch overhead is typical of actual systems. For example, DYNIX uses a 100 msec quantum with 0.75 msec reallocation time (i.e., about 0.75%). Our use of higher than typical context switch overhead gives the benefit of doubt to the space-sharing Dynamic policy considered here. Even with this bias against the time-sharing RRJob policy, we show that the time-sharing RRJob policy performs as well as or better than the space-sharing Dynamic policy.

To reduce the effect of transients, the information about the output statistics is collected only after the measured throughput had reached 99.5% of the arrival rate. Batch method is used to compute the confidence interval (in our case, 30 batch runs were used for the results reported here and each batch contains more than 50,000 jobs). This method gives 95% confidence intervals that were less than 1% of the mean response times when the system utilization is low to moderate and less than 20% for high utilization levels (for example 90%). This technique is applied for all the experiments described here.

To provide a fair comparison across the job models, the output values of the average number of tasks per job n and the average total service demand per job D must have the same values for all the job models. This is achieved by setting appropriate values for the input parameters. The system utilization ρ is varied by changing the arrival rate of the parallel jobs. If ρ is the target system utilization, D is the average total service demand of the jobs in the workload, and there are P processors in the system, the arrival rate λ is set at $\frac{\rho P}{D}$.

Two types of workload, defined in the previous section, are considered – *correlated* and *uncorrelated*. The default values for the correlated workload are shown in Table 4.1. For the uncorrelated workload model, all the parameters remain the same as those used for the correlated workload model except for the service demand. Instead of task service demand used in the correlated model, a job service demand of 20 is used in the uncorrelated workload model.

Table 4.1 Default values of the input parameters for the correlated workload

Parameter	FJ	DC	GE	SSS
mp	20.0	12.444	5.667	12.444
CV_{mp}	1.0	1.0	1.0	1.0
d	1.0	1.0	1.0	1.0
CV_d	5.0	5.0	5.0	5.0
P_T	0.0	0.0	0.0	0.5

4.1 Performance as a function of system load

The purpose of the first set of experiments is to compare the performance of the scheduling policies for different job models and utilization levels. In [9], an experiment using the fork-and-join model was conducted. In this section, the experiment is repeated over different job models to see whether the performance is affected by the change in the job structure.

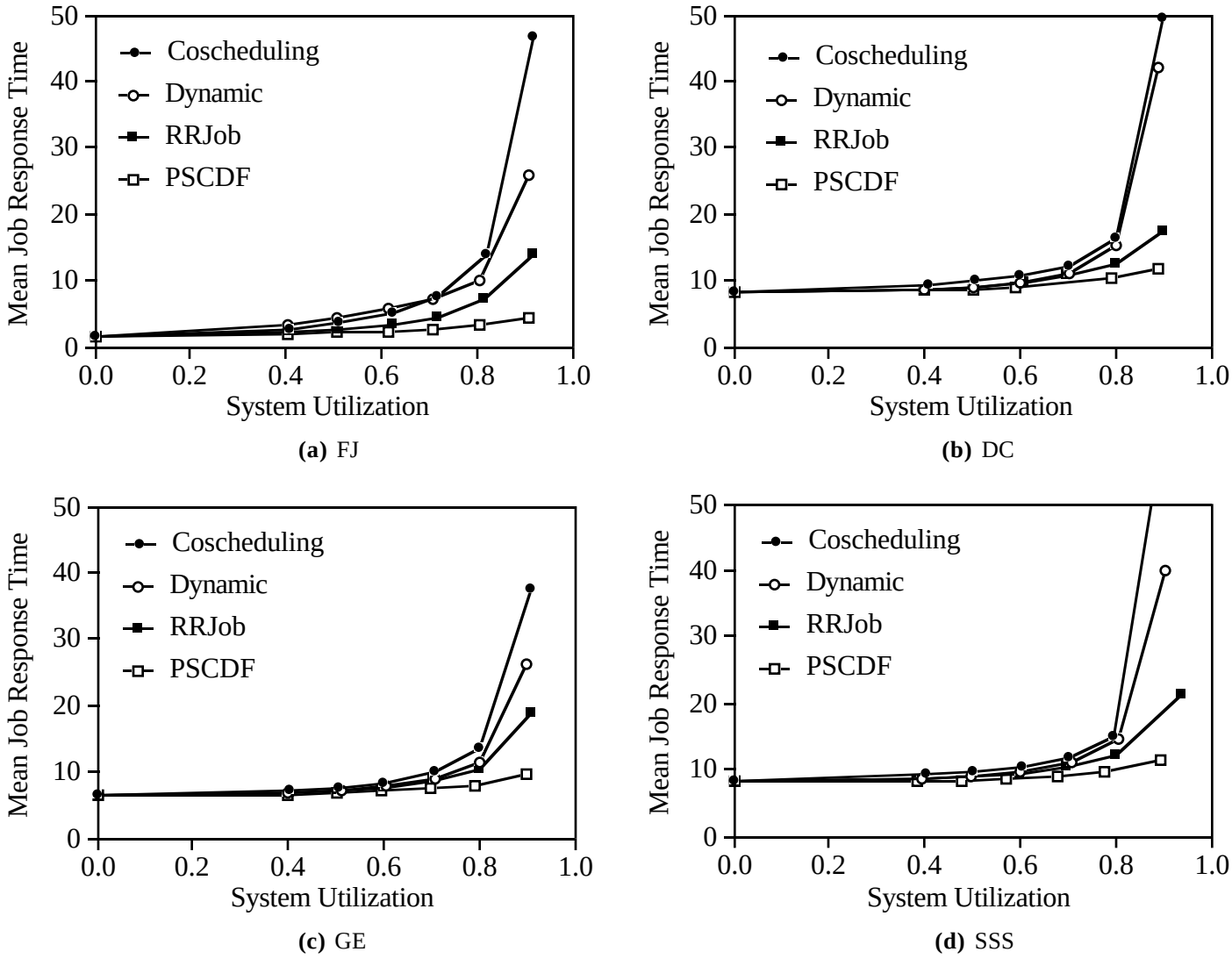
The simulation experiments were run with system utilizations of 0%, 40%, 60%, 70%, 80% and 90% for the uncorrelated and correlated workloads for the four job models. The results for the correlated workload are shown in Figure 4.1. Results for the uncorrelated workload are qualitatively similar and are not shown here for the sake of brevity (details can be found in [1]).

For all four job models, with minor exceptions, Coscheduling and PSCDF policies provide the worst and the best performance, respectively. Of the remaining two policies, RRJob provides better performance than the Dynamic policy. Note that the PSCDF policy is not a practical policy and is included for comparison purposes only.

The results for the fork-and-join job model are shown in Figure 4.1a. The results shown here are consistent with the ones reported in [9]. For the FJ job model, RRJob performs substantially better than the Dynamic policy. The Coscheduling policy performs worse than all the other policies except when the system load is low to moderate. In this case, the Dynamic policy provides the worst performance. The reason for the performance superiority of the RRJob over the Dynamic and Coscheduling policies is that the RRJob provides the jobs with an equal share of the processing power. Therefore, larger jobs (in terms of the total job service demand) do not dominate the smaller jobs. This results in better overall job response time. On the other hand, the Coscheduling policy performs poorly because it gives larger jobs a larger share of the processing power.

Dynamic policy allocates processors to jobs on a first-come-first-served (FCFS) basis. It does not adapt well to balance the processor allocation. Hence, fair allocation of processors to parallel jobs cannot be made at low utilization levels. The problem becomes serious if the workload is correlated. At high system loads, each job essentially gets only a single processor.

For the remaining three job structures, the performance of RRJob and Dynamic policies is similar up to about 70% system load. For higher system loads, the RRJob provides better performance. This is because of the fact that only the FJ job model offers the highest *maximum parallelism* (20 for the parameters considered here) and there is only one synchronization point in the job structure. Therefore, RRJob's equal processing power allocation improves the performance. With the other three workloads, there are more synchronization points and less parallelism is present in the job structure. Of these three job models, only the GE job model offers the least *maximum parallelism* (about 5.6); thus, the performance of the RRJob and Dynamic is even closer for this job model. Both policies provide similar performance for up to 80% system load. Due to the dynamic nature of these job models, processors will be released by parallel jobs during their execution. Thus, the processor allocation is balanced by the dynamic change of job parallelism and the performance of the Dynamic resembles that of the RRJob.



(a) FJ **(b)** DC **(c)** GE **(d)** SSS

Figure 4.1 Performance as a function of system load for the correlated workload model

The difference between the fork-and-join and divide-and-conquer models is the additional synchronization delay introduced by the divide and merge phases of the parallel jobs. For the SSS job model, the synchronization delays introduced by the merge tasks are reduced by having task abortion. This is due to the termination of the aborted tasks. The divide-and-conquer job model can be treated as the SSS model without task abortion (i.e., probability of task abortion $P_T = 0$). Note that, at the other extreme, when $P_T = 1$, there is no synchronization in the SSS job model because the other task that is not yet finished execution is always aborted.

Figure 4.1d shows the impact of task abortion on the performance of scheduling policies. The value of P_T in this experiment is set equal to 0.5. Increasing the probability of task abortion tends to decrease the synchronization time. Thus, on the average, there will be a decrease in the mean job response time for all the scheduling policies as the probability of task abortion increases. This change in the response time can be seen by comparing Figure 4.1b (with $P_T = 0$) with Figure 4.1d (with $P_T = 0.5$). These results also show that task abortion has no impact on the relative performance of the four scheduling policies considered here.

In general, the performance difference among the four policies tends to reduce with decreased parallelism and increased synchronization. This is demonstrated by comparing the results for the FJ job model with those for the other job models.

Related work

The results presented here show that the time-sharing RRJob performs as well as or better than the space-sharing Dynamic policy. This conclusion is in contrast to the results reported in the literature [2,13]. In the following we briefly discuss the reasons for these differences and explain the significance of our results.

McCann et al. [13] report the performance of three scheduling policies – two space-sharing policies and one time-sharing policy. The space-sharing policies considered are the Dynamic policy (discussed in this paper) and an Equipartition policy based on the process control policy proposed by Tucker and Gupta [18]. The time-sharing policy is a modified RRJob policy (with the modification their RRJob policy behaves more like the coscheduling policy). They conclude that the Dynamic policy is better than the RRJob policy. They reached this conclusion even without considering cache affinity effects. This is not surprising because their so-called RRJob is really a modified coscheduling policy and coscheduling has been shown to perform worse than other policies. Also, these conclusions should be viewed in light of the workload used by them. Of the four applications considered, three applications exhibit a job structure that is a variation of the FJ job model. One application (MVA) exhibits job structure that is representative of many wavefront computations. Also, the multiprogramming level was fixed at less than or equal to 3 (except in one workload mix that has a multiprogramming level of 4). When a job is finished, another instance of the same job is started immediately. Thus the response time they report is really the execution time of the job or application. It is not surprising, then, that the two space-sharing policies performed better than the time-sharing policy. It should be noted that the major advantage of space-sharing policies is that they context switch less frequently than does the RRJob policy. Since their study does not include the waiting time of a job in the response time computation, the major advantage of the RRJob – reducing the job waiting time – is not considered and the results are biased in favour of the space-sharing policies. Previous studies have shown that excluding jobs from service until jobs complete can hurt performance (for example, see [9]).

Crovella et al. [2] also conclude that space-sharing policies are better than time-sharing policies (one presumes from their paper that they have implemented the coscheduling policy). Furthermore, as in [13], Crovella et al. also compute only the execution time of a job (not the waiting time plus the execution time). Our results show that, when the waiting time is included in the response time computations, the time-sharing RRJob policy competes well with the space-sharing Dynamic policy.

A potential disadvantage of the RRJob policy is that the more frequent (compared to the space-sharing Dynamic policy) context switches generated by this policy may lead to more time being spent on cache reloading. Thus, when cache affinity effects are included, the performance improvement obtained due to this factor would be higher for the Dynamic policy. The significance of this effect is not clear and requires further study.

4.2 Sensitivity to service time variance

This section considers the performance sensitivity of the four scheduling policies to the variance in service demand of the jobs. The results presented in Figure 4.1 assumed a high variance in service demand (CV_d = 5, a value used in other studies including [10]). Figure 4.2 presents the result of this second set of experiments for the correlated workload[1]. Since the performance with the GE and SSS job models is similar to that of the DC job model, these results are not presented here.

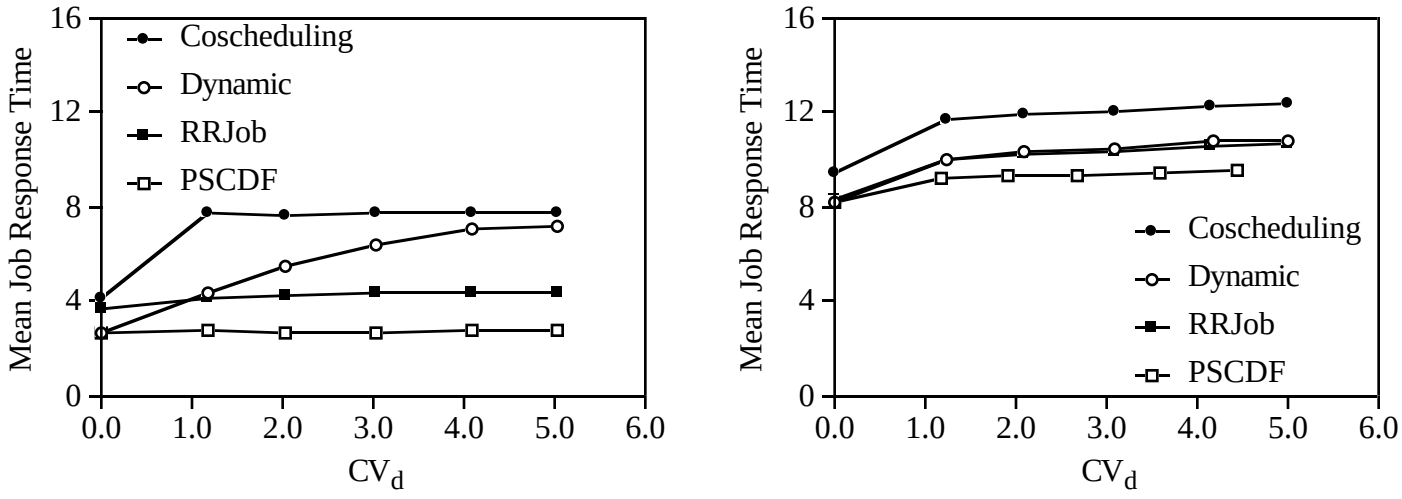
The input parameters for different job models are set such that the output parameters are $n \approx 20$, $CV_n \approx 1$, $d \approx 20$ and $CV_d \approx 1$, 2, 3, 4 and 5. For the no variation case, both CV_n and CV_d are set to zero[2]. The utilization level is fixed at around 70% by setting $\lambda = 0.7$.

Figure 4.2a presents the results of the fork-and-join model. When there is no variation in the service demand and the number of tasks in a job (i.e., $CV_d = 0$ and $CV_n = 0$), both Dynamic and Coscheduling policies perform substantially better than when $CV_n \neq 0$. The Coscheduling policy, like the other time-sharing policy, exhibits little sensitivity to the variance in service demand for $CV_d \geq 1$. However, it is sensitive to the variance in the number of tasks among the jobs. This is due to the fact that this policy tends to allocate more processors to a job that has more tasks. This disadvantage of the Coscheduling policy disappears when $CV_n = 0$. For a similar reason, Dynamic performs substantially better for the no variation case. Note that, for the no variation case, the Dynamic policy provides substantially better performance than the RRJob policy. In this case, the performance of Dynamic policy is similar to that of the PSCDF policy.

As CV_d increases, the ratio between the service demand of large jobs and that of small jobs increases. Dynamic policy is sensitive to the variance in service demand. This is because the Dynamic policy does not provide time-sharing, rather uses the FCFS allocation of processors. It is well-known that the FCFS is sensitive to service time variance. FCFS allocation of processors to jobs results in a situation where small jobs could be held up by a large job that had arrived earlier. This problem gets more serious as the variance in service demand increases. The other policies are insensitive to the variance in service demand because of the time-sharing nature. An increase in CV_d only increases the variance of job service demand and does not affect the processor allocation decision of these policies.

[1] Results for the uncorrelated workload are not presented as they are qualitatively similar (see [1] for details).

[2] Because the workload is correlated, a CV_d of zero is obtained only by making $CV_n = 0$.



(a) FJ **(b)** DC

Figure 4.2 Sensitivity to service time variance CV_d - correlated workload

Figure 4.2b presents the results for the divide-and-conquer job model. All four policies, including Dynamic policy, are insensitive to the variance in service demand for $CV_d \geq 1$. Due to the dynamic nature of this job model, processor reallocation can be used to meet the change in parallelism of a job. Thus, varying CV_d does not degrade the performance of the Dynamic policy. As RRJob and Coscheduling policies allocate the processing power on a time sharing basis, varying the CV_d does not have a significant effect on their performance.

The sensitivity of Dynamic policy (in the FJ job model) to the variance in service demand is a direct result of the maximum parallelism offered by this job model and the FCFS nature of the Dynamic policy in scheduling jobs. As a result, Dynamic policy improves its performance with decreasing variance in service demand. The other three job models reduce the sensitivity of the Dynamic policy because of the more frequent synchronization and less parallelism present in the job structures. These observations agree with the results reported by previous studies that used low variance in service demand [2,13]. For example, the workload used in McCann et al. [13] is taken from a population of at most (depending on the job mix considered) four jobs.

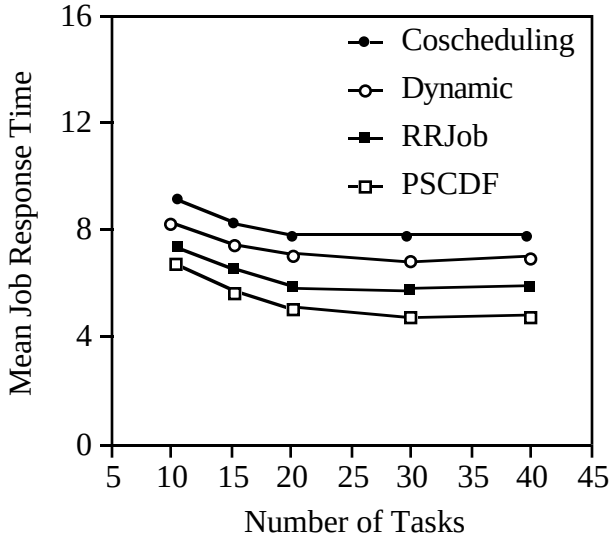
4.3 Sensitivity to number of tasks

In this section, we present the performance sensitivity to the average number of tasks per job *n*. The system utilization is fixed at 70% for this set of experiments. Other parameters are set to their default values given in Table 4.1. The experiments were conducted for values of $n \approx 10, 15, 20, 30$ and 40 for both the correlated and uncorrelated workloads. The results for the uncorrelated workload are presented first in this subsection.

Results for the Uncorrelated Workload

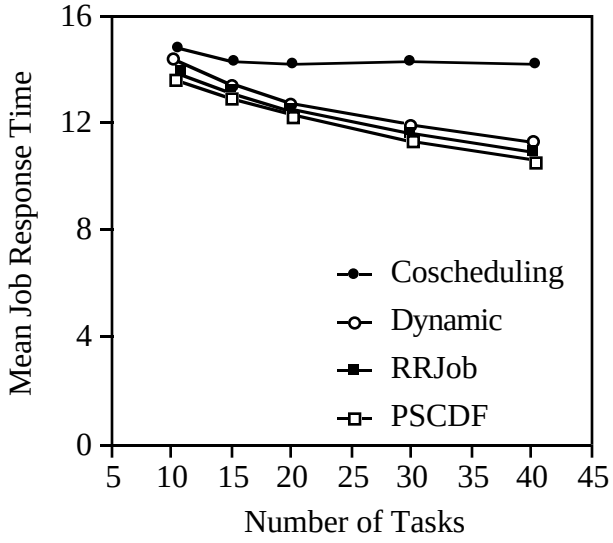
The simulation results for the fork-and-join job model is shown in Figure 4.3a. Similar experiments have also been reported in [9]. We extend the experiment to consider the cases where $n \geq 20$ (the number of processors in the system). As *n* increases the mean job response time decreases up to $n \approx 20$. After that we see only a marginal sensitivity to *n*. The initial increase in *n* introduces additional job parallelism, which results in a decrease in the response time. However, when $n \approx 20$, the advantage of additional parallelism is limited by the existing resources and, hence, no further performance improvement can be obtained.

Figure 4.3b plots *n* versus the average job response time for the divide-and-conquer job model. Performance of the other two job models is similar and is given in [1]. For the divide-and-conquer job model, within the range of *n* values considered here, the average maximum parallelism is approximately 16.0. When *n* is 40.0, the parallelism per job is equal to about the number of processors. Thus, the advantage of additional parallelism is not affected by the system resources. For the Coscheduling policy, the decrease in mean job response time is only marginal. This shows that Coscheduling is not suitable for divide-and-conquer jobs, which exhibit varying degree of job parallelism.



n	10.0	15.0	20.0	30.0	40.0
mp	10.0	15.0	20.0	30.0	40.0

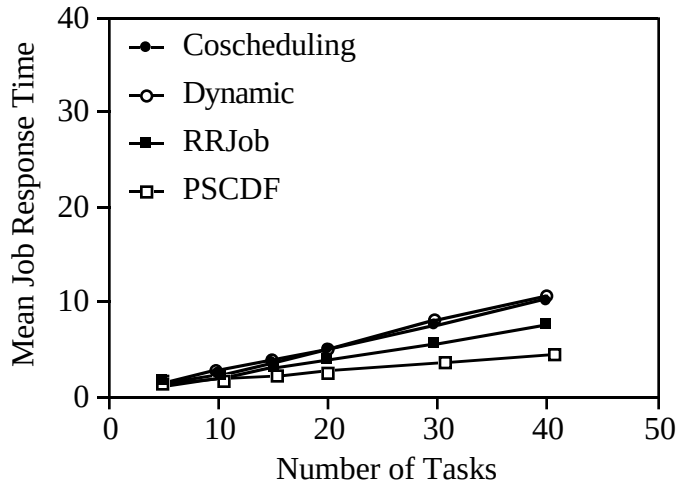
(a) FJ



n	10.36	16.04	20.04	31.05	41.46
mp	7.5	8.0	12.44	16.0	22.5

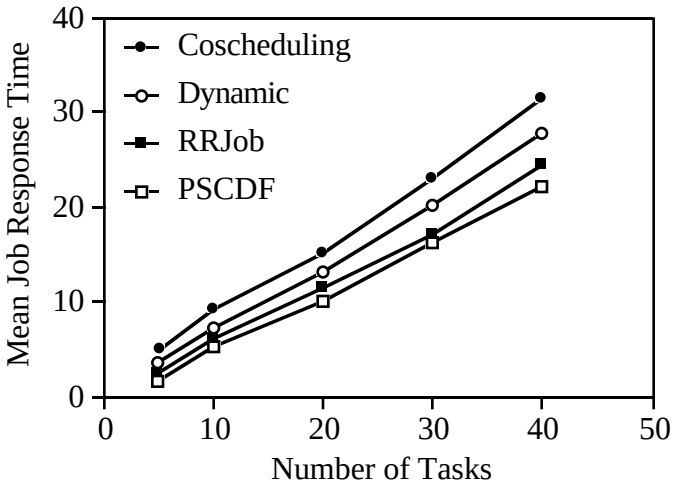
(b) DC

Figure 4.3 Sensitivity of performance to number of tasks *n* – uncorrelated workload



n	5.0	10.0	15.0	20.0	30.0	40.0
mp	5.0	10.0	15.0	20.0	30.0	40.0
d (output)	4.98	9.97	15.01	19.97	29.94	40.00

(a) FJ



n	5.03	10.36	16.04	20.0	31.05	41.46
mp	3.54	7.5	8.0	12.4	16.0	22.5
d (output)	5.13	10.13	16.56	20.34	30.99	40.94

(b) DC

Figure 4.4 Sensitivity of performance to number of tasks *n* – correlated workload

Results for the Correlated Workload

The sensitivity analysis of n is also performed for the correlated workload. The results are presented in Figure 4.4. (GE and SSS job models exhibit qualitatively similar behaviour to that of the DC job model.) Note that, in this workload, there is a positive correlation between the service demand and the number of tasks of a job. This is reasonable for applications where the service demand of individual tasks would remain unaffected by the number of tasks. For example, in Gaussian elimination, tasks are responsible for the computation of the dot-product of their associated rows and columns. Thus, an increase in the number of tasks corresponds to an increase in the size of the input matrices, which implies an increase in job service demand. A similar argument can also be made for the divide-and-conquer job model.

In this experiment, n is set to 10, 15, 20, 30 and 40. We vary the arrival rate so that the utilization of the system is approximately equal to 70% for all values of n. The default values of the other input parameters are given in Table 4.1.

Due to the correlation between the job service demand and number of tasks, the mean job response time increases as we increase n. Furthermore, there is no change in the relative performance of the four policies as we increase the number of tasks. For the fork-and-join (FJ) job model, the Dynamic policy performs marginally worse than the Coscheduling policy.

4.4 Impact of variance in number of tasks

Experiments for different levels of utilization were performed with CV_n = 0 and 5. CV_n = 0 represents the case where all jobs have the same number of tasks. CV_n = 5 represents the situation where the workload consists of large number of jobs with small number of tasks and small number of jobs with a large number of tasks. Other input parameters are set to their default values given in Table 4.1.

Results for CV_n = 0

Figure 4.5a presents the results for the fork-and-join job model for the correlated workload. Since the correlation between the service demand and the number of tasks breaks down as a result of using a constant number of tasks per job (because CV_n = 0), the results for the uncorrelated workload are similar to those for the correlated workload.

The Coscheduling policy tends to give large jobs consecutive quanta and hence increases the waiting time of small jobs. Such a problem does not exist when the number of tasks per job is fixed and, hence, the performance of Coscheduling improves. Under these circumstances, the RRJob maintains the same quantum size for each job. Thus, the scheduling sequence of this policy is similar to that of Coscheduling policy and, therefore, results in similar performance.

Fixing the number of tasks per job does not yield performance benefits by using the Dynamic policy, as can be seen from the above results (Dynamic gives the worst performance). At low to moderate system loads, the jobs with large service demand[3] tends to acquire more processors, as they stay in the system for a longer time. Small jobs that have arrived after large jobs will be allocated fewer processors than the large jobs. Such an unfair allocation of processors results in increased response time.

Figure 4.5b presents the results for the divide-and-conquer job model for the correlated workload. For reasons discussed above, the results for the uncorrelated workload are similar [1]. With the divide-and-conquer job model, the Coscheduling policy gives the worst performance among the four policies. It performs worse than the RRJob mainly due to a larger synchronization delays. Although the Coscheduling policy is designed to reduce the synchronization delay of parallel jobs, it fails to do so under the divide-and-conquer workload. The group of tasks created from different branches of a job may be scheduled in different quanta, as each group is placed at different places in the ready queue of tasks. As a result, the synchronization delay of the jobs increases. On the other hand, RRJob is implemented in such a way that the tasks of a job are placed together in the job queue. This results in minimizing the synchronization delay of a job. In addition, the large context switch overhead associated with the Coscheduling policy also affects its performance.

Coscheduling performs worse than the Dynamic mainly because it incurs large amount of context switch overhead. On the other hand, Dynamic performs preemption only when a new job arrives or a job changes its parallelism. Therefore, the amount of overhead will be much less than that in the Coscheduling policy.

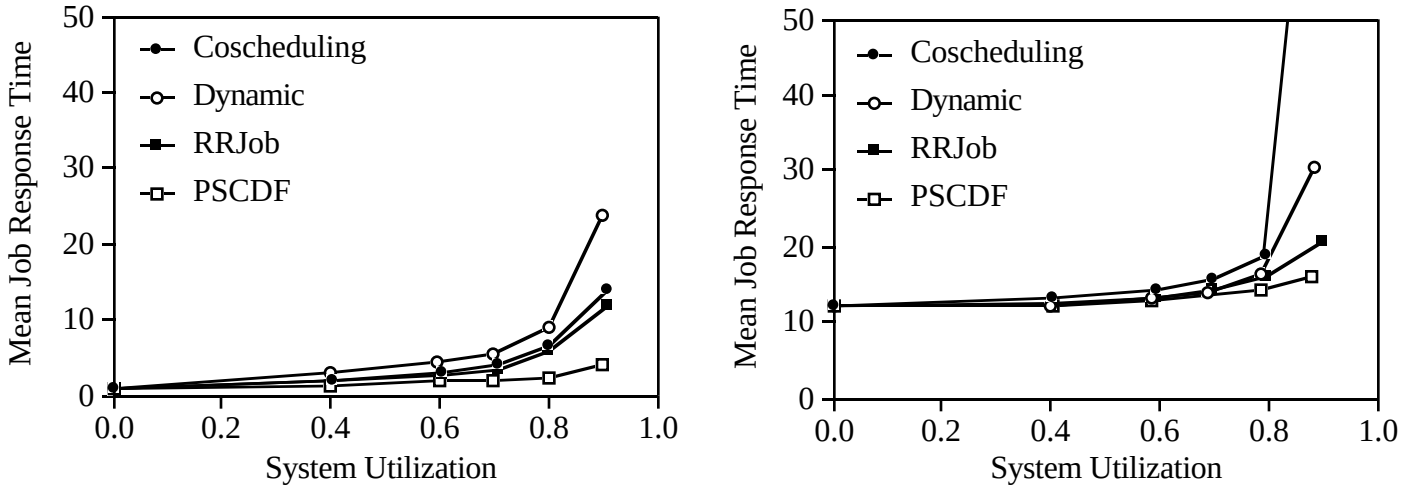
Dynamic has a smaller context switch overhead by minimizing the number of context switches per job. However, it yields a longer synchronization delay per job. On the other hand, although the RRJob gives a shorter synchronization delay per job, its context switch overhead is larger than that of the Dynamic policy. The performance of these two policies is comparable up to 80% system load, then the performance of the Dynamic degrades as the number of jobs exceeds the number of processors.

Results for CV_n = 5

The results for the correlated and uncorrelated workloads for the fork-and-join job model with CV_n = 5 are presented in Figure 4.6.

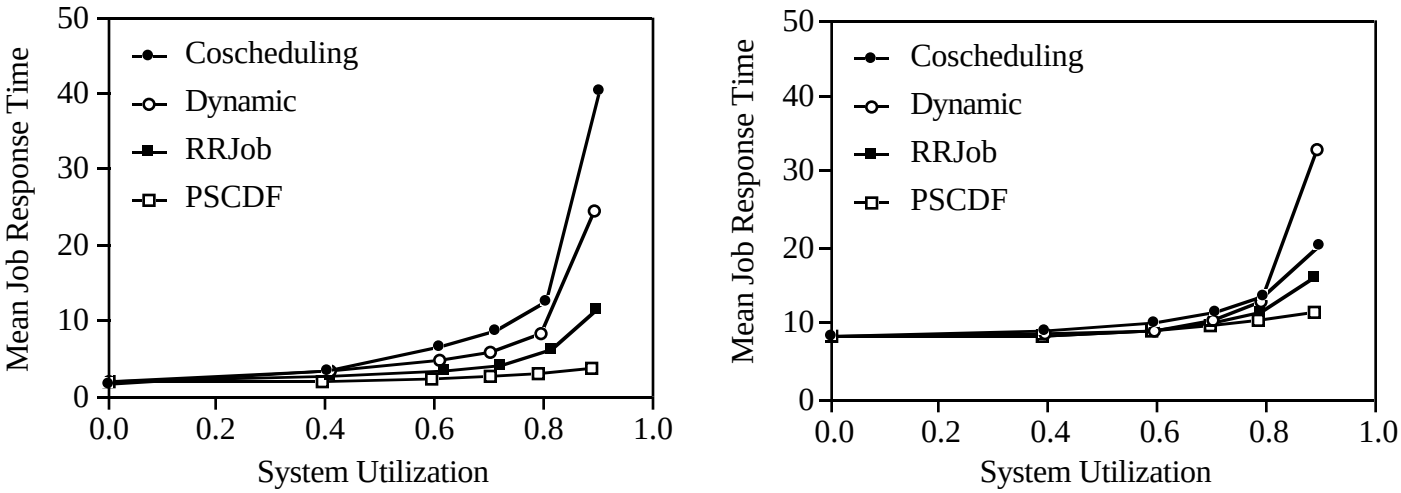
---

3 Note that even though the number of tasks per job is constant, the service demand of jobs has a CV of 5.



(a) FJ **(b)** DC

Figure 4.5 Performance of scheduling policies for the correlated workload (CV_n = 0)

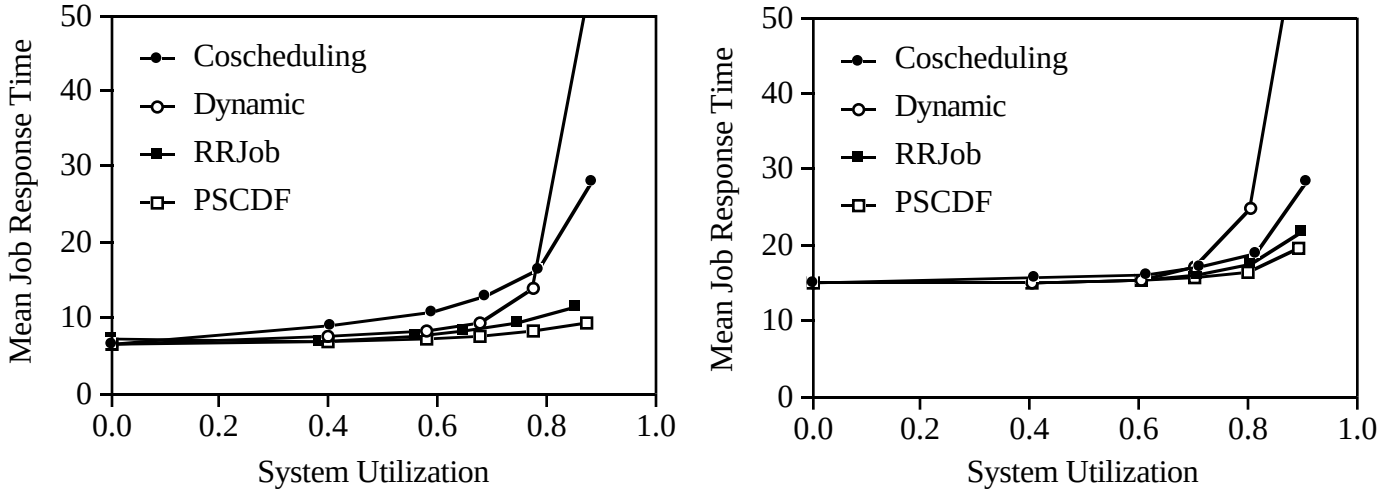


(a) correlated **(b)** uncorrelated

Figure 4.6 Performance of scheduling policies for the FJ job model (CV_n = 5)

The performance of the policies under the correlated workload is discussed first. In this workload, there are 20.7% of sequential jobs, 66.3% of jobs with 1 to 10 tasks, 7.5% of jobs with 10 to 20 tasks and 5.5% of jobs with more than 20 tasks. When the workload is correlated, jobs with large number of tasks tend to have a larger service demand. Thus, Coscheduling performs poorly, as a result of providing disproportionate share of the processing power to larger jobs. As discussed in the previous section, the Dynamic policy uses FCFS processor allocation policy and tends to give a larger share of the processing power to bigger jobs. Note that for the correlated workload, larger number of tasks implies larger service demand. This causes the policy to perform worse than the RRJob. However, Dynamic policy performs better than the Coscheduling because Dynamic causes less frequent context switching.

On the other hand, under the uncorrelated workload, the performance of the Coscheduling is closer to that of the RRJob policy. When CV_n = 5, there are large number of jobs that are either sequential or have small amount of parallelism. Moreover, in this case, there is no correlation between the service demand and number of tasks, so the jobs with large number of tasks may not have a correspondingly large service demand. Hence, even though the jobs with large number of tasks may take up series of quanta, their effects will be very limited. The Dynamic policy performs poorly at moderate to high utilization levels. Since the Dynamic policy only allows as many jobs into the system



(a) correlated **(b)** uncorrelated

Figure 4.7 Performance of scheduling policies for the DC job model (CV_n = 5)

as there are processors, the remaining jobs are kept in the queue until a job finishes execution. As a result, Dynamic policy behaves similar to the first-come-first-served discipline and provides performance that is worse than that of Coscheduling policy.

The results of the correlated and uncorrelated versions of the divide-and-conquer workload model with CV_n = 5 is shown in Figure 4.7. As explained before, because of the reduced parallelism and increased synchronization associated with this job structure, the performance difference among the policies (excluding the Coscheduling) tends to decrease. The relative performance of the policies under GE and SSS job models is similar to that in the fork-and-join model [1]. Once again it shows that the presence of task abortion does not change the relative performance of the four policies.

5. Conclusions

A simple *fork-and-join* type of job structure has been extensively used for performance evaluation of processor scheduling policies in multiprocessor systems. However, parallel programs often exhibit a more complicated structure. This paper has studied the impact of program structure on the performance of processor scheduling policies.

We have considered four types of parallel program structures that are frequently used in parallel applications. These are the fork-and-join, divide-and-conquer, Gaussian elimination, and state space search programs. The impact of these four job structures on the performance of four processor scheduling policies is considered. These are the job-based round robin (RRJob), coscheduling, dynamic, and preemptive smallest cumulative demand first (PSCDF) policies. These policies cover a variety of policies – preemptive vs. non-preemptive, space-sharing vs. time-sharing, *a priori* job knowledge vs. no job knowledge.

The following observations summarize the results presented in this paper for the four types of job structures considered in this paper.

- The performance of the Coscheduling policy is the worst for all the various parameters and the types of job structures considered. This is partly because of the types of jobs considered in this study. The Coscheduling policy is devised for those jobs where inter-task communication is dominant. In this case, coscheduling the tasks of a job would improve the overall system performance. Since the focus of this paper is on the job structure and task synchronization, the main feature of the Coscheduling policy is not exploited. The results presented here show that the Coscheduling performs very poorly when the inter-task communication is not dominant. Similar results were obtained by other researchers as well.
- The PSCDF policy provides the best performance among the four processor scheduling policies. Thus, *a priori* job service demand knowledge improves performance substantially. Since such *a priori* information is difficult to obtain in practice, this policy is not of practical importance, but is included only for comparison purposes.
- The performance of the Dynamic policy is sensitive to the workload. The results presented here show that the job structure has a significant impact on the performance of this policy. When there is little variation in job parallelism, as exemplified by the fork-and-join job model used here, Dynamic performs worse than RRJob policy. This is in contrast to the results reported in the literature that showed that Dynamic is better than RRJob. The previous results have not included job waiting time in the response time

calculation. When interested only in the job execution time, the previous results show that the space-sharing Dynamic policy is better than the time-sharing RRJob policy. This is mainly because space-sharing policies tend to cause less frequent context switching. However, when the job waiting time is included in the response time calculation, RRJob performs as well as or better than the Dynamic policy. In particular, the performance superiority of the RRJob increases with increasing variance in job service demand and increased parallelism in jobs. The main reason for this is that the Dynamic uses FCFS processor allocation and tends to allocate larger processor share to bigger jobs. Thus, despite the recent results, time-sharing processor scheduling policies are useful for multiprocessor systems.

- With few exceptions, varying the job structure has a significant impact only on the performance of the Dynamic policy. For the remaining three policies, the relative performance is not affected by the job structure or other parameters of the workload except when the job structure is fork-and-join and $CV_n = 0$. In this case, the Coscheduling policy performs as well as the RRJob policy.
- Relative performance of the scheduling policies is not affected by task-abortion.

Acknowledgements

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council (NSERC) and Carleton University. A shorter version of this paper appears in *Proc. ISCA International Conference on Parallel and Distributed Computing Systems*, Las Vegas, October 1994.

References

[1] S. L. Au, *Characterization of Some Parallel Programs and Its Impact on Processor Scheduling Policies*, M.C.S. Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1992.

[2] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, E. Markatos, "Multiprogramming on multiprocessors," *Third IEEE Symp. Parallel and Distributed Processing*, Dallas, Texas, December 1991, pp. 590-597.

[3] S. Dandamudi, "A comparison of task scheduling strategies for multiprocessor systems," *Third IEEE Symp. Parallel and Distributed Processing*, Dallas, Texas, December 1991, pp. 423-426.

[4] S. P. Dandamudi, "Performance implications of task routing and task scheduling strategies for multiprocessor systems," *Proc. IEEE Int. Conf. Massively Parallel Computer Systems*, Ischia, Italy, 1994, pp. 348-353.

[5] S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 1, January 1995, pp. 1-16.

[6] A. Duda, "On the Tradeoff between Parallelism and Communication," in *Modelling Techniques and Tools for Computer Performance Evaluation*, R. Puigjaner and D. Potier (Eds.), Plenum Publishing Co., 1988.

[7] E. Horowitz and A. Zorat, "Divide-and-conquer for parallel processing," *IEEE Trans. on Computers*, Vol. C-32, No. 6, June 1983, pp. 582-585.

[8] P. Jones and A. Murta, "Practical Experience of Run-Time Link Reconfiguration in a Multi-Transputer Machine," *Concurrency - Practice and Experience*, Vol. 1, No. 2, December 1989, pp. 171-189.

[9] S. T. Leutenegger, *Issues in Multiprogrammed Multiprocessor Scheduling*, Technical Report #954, Computer Science Dept., University of Wisconsin-Madison, August 1990.

[10] S. T. Leutenegger, and M. K. Vernon, "The performance of multiprogrammed multiprocessor scheduling policies," *Proc. ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1990, pp. 226 - 236.

[11] S. Madala and J. B. Sinclair, "Performance of synchronous parallel algorithms with regular structures," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 1, January 1991, pp. 105-116.

[12] S. Majumdar, D. L. Eager and R. B. Bunt, "Scheduling in multiprogrammed parallel systems," *Proc. ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1988, pp. 104-113.

[13] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors," *ACM Trans. Computer Systems*, Vol. 11, No. 2, May 1993, pp. 146-178.

[14] R. Nelson, D. Towsley, and A. N. Tantawi, "Performance analysis of parallel processing systems," *IEEE Trans. Software Engineering*, Vol. SE-14, No. 4, April 1988, pp. 532-540.

[15] J. K. Ousterhout, "Scheduling techniques for concurrent systems," *Proc. of 3rd International Conference on Distributed Computing Systems*, October 1982, pp. 22-30.

[16] K. C. Sevcik, "Characterizations of parallelism in applications and their use in scheduling," *Proc. ACM SIGMETRICS Conference on Measurement and*

Modelling of Computer Systems, May 1989, pp. 171-180.

[17] D. Towsley, C. G. Rommel, and J. A. Stankovic, "Analysis of fork-join program response times on multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 286-303.

[18] A. Tucker and A. Gupta, "Process Control and Scheduling issues for multiprogrammed shared-memory multiprocessors," *Proc. of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 159-166.

[19] J. Zahorjan and, C. McCann, "Processor scheduling in shared-memory multiprocessors," *Proc. ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1990, pp. 214-225.

List of Abbreviations

λ	Job arrival rate
ρ	System utilization
Coscheduling	Coscheduling time-sharing policy
CV_a	Coefficient of variation of inter-arrival times
CV_d	Coefficient of variation of service demand
CV_n	Coefficient of variation of n
d	Mean service demand of a task
D	Mean service demand of a job
DC	Divide-and-conquer job structure
Dynamic	Dynamic space-sharing scheduling policy
FJ	Fork-and-Join job structure
GE	Gaussian elimination type job structure
mp	Maximum parallelism of a job
n	average number of tasks per job
NUMA	Non-uniform memory access shared-memory systems
P	Number of processors
PSCDF	Preemptive smallest cumulative demand first scheduling policy
P_T	Probability of task abortion in SSS jobs
RRJob	Job-based round robin scheduling policy
SSS	State space search type job structure
UMA	Uniform memory access shared-memory systems