Parallel Job Scheduling in Homogeneous Distributed Systems

Helen D. Karatza
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece

Ralph C. Hilzer
Computer Science Department
California State University, Chico
Chico, California 95929-0410 USA

The performance of parallel programs such as those involving fork-join instructions is significantly affected by the method used to schedule program tasks. This paper studies parallel job scheduling in homogeneous distributed systems. A simulation model is used to address performance issues associated with scheduling. Various policies are employed to schedule parallel jobs over a variety of workloads. Fairness is required among competing jobs. We examine cases where the distribution of the number of parallel tasks per job, and also the distribution of task service demand, vary with time. We also examine the impact of overhead necessary to collect global system information about processor queues on performance. Simulated results indicate that although all scheduling methods have merit, one significantly improves the overall performance and guarantees fairness in terms of individual job execution.

Keywords: Simulation, performance, distributed systems, scheduling

1. Introduction

Distributed systems have been the focus of research for many years. They consist of several, loosely interconnected processors, where jobs to be processed are in some way apportioned among the processors, and various techniques are used to coordinate processing. However, it is still not clear how to efficiently schedule parallel jobs. To determine this, it is critical to properly assign the tasks to processors and then schedule execution on a distributed processor. Good scheduling policies can maximize system and individual application performance, and avoid unnecessary delays.

The primary focus of most existing research is to find ways to distribute tasks among the processors in order to achieve performance goals such as minimizing job execution time, minimizing communication and other overhead, and/or maximizing resource utilization. However, there are cases where task sequencing should be preserved as much as possible to achieve fairness in individual job execution. A task that is given a low priority according to the scheduling method's criteria should not be overtaken by an arbitrary number of higher priority tasks.

Parallel job scheduling has been extensively studied in the literature of parallel and distributed systems. Dandamudi in [1] conducted a thorough study of task scheduling in multiprocessor systems. Results from that study indicate that scheduling policies have substantial impact on performance when non-adaptive routing strategies are used. Dandamudi in [2] also examined the impact of node scheduling policies on the performance of sender-initiated and receiver-initiated dynamic load sharing policies. He considered two-node scheduling polices – first-come/first-served (FCFS) and round robin (RR) and he studied two types of heterogeneous systems.

The performance sensitivity of dynamic load sharing to various system and workload characteristics in distributed systems has been studied in [3]. Task assignment in heterogeneous distributed computing systems is studied in [4] where the authors propose two algorithms based on the A* technique (A* is a best-first search algorithm, which has been used extensively in artificial intelligence problem solving).

Scheduling policies in distributed systems have also been studied in [5], [6], and [7]. All these works consider jobs that consist of independent parallel tasks.

A different type of parallel job scheduling is considered in [8], [9], and [10] where parallel tasks are required to start at essentially the same time, co-ordinate their execution, and compute at the same pace.

In this paper, job tasks are independent so they can execute at any time, in any order, and at any processor. Scheduling is performed in two steps. The first step, spatial scheduling or routing, consists of assigning tasks to processors. The second step, temporal scheduling, consists of defining the sequence with which tasks at a processor queue will be executed. Six task scheduling policies are examined which combine the probabilistic or the join the shortest queue routing mechanism with four temporal scheduling methods (first come first served, and three others that take into account job characteristics or job status).

Previous research in the area of parallel job scheduling assume that the number of tasks per job is defined by a specific distribution (for example uniform or normal) and also that task service demand is defined by a specific distribution (for example exponential). However, in real systems, the variability of job parallelism and also the variability of task service demand can vary depending on the applications that run on different time intervals. For this reason this paper uses an exponentially varying with time distribution for the parallelism of jobs which represents real parallel system workloads. We also consider an exponentially varying with time distribution for the task service demand. The performance of the different scheduling policies is compared over various degrees of multiprogramming (numbers of jobs in the system).

This paper is an extension of previous research [11]. That paper considers time varying workloads in a closed queuing network model of a distributed system. Simulation results reveal that the shortest queue routing mechanism, combined with FCFS temporal task scheduling, outperforms the other methods. However, in [11] we did not take the overhead into account that is incurred while collecting global system information about processor queue length. This paper does consider that overhead, and examines its impact on the shortest queue routing algorithm performance. Therefore, conclusions can be drawn from the simulation results that are closer to realistic situations than those presented in [11]. This is possible because the collection and management of global load information incur non-trivial levels of overhead.

A closed queuing network model of a distributed system is considered which incorporates I/O equipment. The goal is to achieve high system performance while also providing fairness of job execution. To our knowledge, such an analysis of parallel job scheduling does not appear in research literature for this kind of a distributed system operating with this type of workload.

This is an experimental study in the sense that the results are obtained from simulation tests instead of from measurements of real systems. Nevertheless, we believe that the results are of practical value. Although absolute performance predictions are not derived for specific systems and workloads, we study the relative performance of differing scheduling algorithms across a broad range of workloads and we analyze how changes to the workload can affect performance.

For simple systems, performance models can be mathematically analyzed using queuing theory to provide performance measures. However, in the system presented in this paper, Branching Erlang [12] and exponential distributions are used to compute task execution time. Also, fork-join programs and scheduling policies with different complexities are involved. For such complex systems, analytical modelling typically requires additional simplifying assumptions that might have unforeseeable influence on the results. Therefore, research efforts are devoted to finding approximate methods to develop tractable models in special cases, and in conducting simulations. The precise analysis of fork-join queuing models is a well known intractable problem. For example, Kumar and Shorey in [13] derived upper and lower bounds for the mean response time when jobs have a linear fork-join structure. We chose simulations because it is possible to simulate the system in a direct manner, thus lending credibility to the results. Detailed simulation models help determine performance bottlenecks in architecture and assist in refining the system configuration.

The structure of the paper is as follows. Section 2.1 specifies system and workload models, section 2.2 describes scheduling policies, and section 2.3 presents the metrics employed in assessing the performance of the scheduling policies that are studied. Model implementation and input parameters are described in section 3.1, while the results of the simulation experiments are presented and analyzed in section 3.2. Section 4 summarizes findings and offers suggestions for further research and the last section is References.

2. Model and Methodology

2.1 System and Workload Models

The technique used to evaluate the performance of the scheduling disciplines is experimentation using a synthetic workload simulation. A closed queuing network model of a distributed system is considered. There are P homogeneous and independent processors each serving its own queue and interconnected by a high-speed network. We examine the system for $P = 16$ processors. This is a reasonable size for current existing medium-scale departmental networks of workstations.

Since we are interested in a system with a balanced program flow, we have included an I/O subsystem which

has the same service capacity as the processors. The I/O subsystem may consist of an array of disks (multi-server disk center) but it is modeled as a single I/O node with a given mean service time. Each I/O request forks in sub-requests that can be served by the parallel disk servers.

In the simulation experiments, we assume that a fixed number of jobs N are repeatedly executed in the closed circle of parallel processors and an I/O unit shown in Figure 1. N is called the degree of multiprogramming of a simulation experiment.

Since both processors and I/O unit are involved, we need to examine the performance of both processors and the I/O in our parallel job scheduling. Rosti et al. in [14] study parallel computer systems and suggest that the overlapping of the I/O demands of some jobs with the computational demands of other jobs offers a potential improvement in performance. In Figure 1, x and z represent the mean processor and the mean I/O service time respectively.

One scheduling policy case takes communication overhead into account that is incurred when the scheduler collects global system information about processor queue assignment decisions. The remaining scheduling policies assume negligible overhead when assigning job tasks to processor queues. In Figure 1, the scheduler is modeled as a single server queuing system, with mean service time *Co,* that represents the effects of communication overhead. $Co = 0$ in other cases where overhead is ignored. In the latter case, the scheduler dispatches job tasks immediately to processor queues upon job arrival.

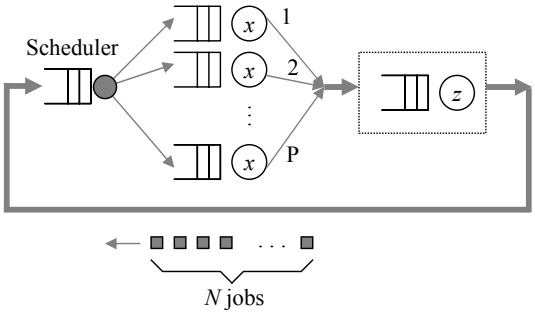

Figure 1. The queuing network model

An important part of a distributed system design is the workload shared among the processors. This involves partitioning the jobs into tasks that can be executed in parallel, assigning the tasks to processors, and scheduling the task execution on each processor.

Jobs are partitioned into independent tasks that can run in parallel. The number of tasks that a job consists of is this *job's degree of parallelism*. On completing execution, a task waits at the join point for sibling tasks of the same job to complete execution. Therefore, synchronization among tasks is required. The price paid for increased parallelism is a synchronization delay that occurs when tasks wait for siblings to finish execution.

The number of tasks of a job x is represented as $t(x)$. If $p(x)$ represents the number of processors required by job x, then the following relation holds:

$$p(x) \le t(x) \le P$$

The workload considered here is characterized by four parameters:

- The distribution of the number of tasks per job.
- The distribution of task service demand.
- The distribution of I/O service time.
- The degree of multiprogramming.

We consider that job parallelism and task service demand are not defined by a specific distribution but that the distribution changes with time. So, a time interval during which the variability in jobs parallelism is high is followed by a time interval during which the majority of jobs exhibit moderate parallelism. Also, during some time period, task service demands are highly variable while during other time periods, task service time is exponentially distributed.

Each task is assigned to one of the queues accordingly to the routing policy that is applied. Tasks in processor queues are executed according to the temporal scheduling method that is currently employed. No migration or pre-emption is permitted.

2.1.1 Distribution of the Number of Tasks per Job

We assume that the distribution of the number of tasks per job changes in exponentially distributed time intervals d_1, d_2, d_3, ..., d_n from uniform to normal and vice versa (Figure 2). The mean time interval for distribution change is d. In the uniform distribution case, the number of job tasks is uniformly distributed in the range of $[1..P]$. The mean number of tasks per job is $\eta = (1+P)/2$. In the normal distribution case we assume a "bounded" normal distribution for the number of tasks per job in the range of $[1..P]$ with mean $\eta = (1+P)/2$ and standard deviation $\sigma = \eta/4$.

Those jobs that arrive at the processors within the same time interval d_i have the same distribution for the number of tasks that they consist of. However, during the same time interval some jobs exist at the processors that arrived during a past time interval and which may have a different distribution for the number of their tasks. These jobs may wait at the processor queues or to be served.

It is obvious that jobs in the uniform distribution case present larger variability in their degree of parallelism than jobs whose number of tasks are normally distributed.

In the second case, most of the jobs have a moderate degree of parallelism (close to the mean η). Since the distribution of job parallelism changes with the time, for some time intervals, arriving applications have highly variable degree of parallelism, while during other time intervals, the majority of the arriving applications have a moderate parallelism as compared to the number of processors.

2.1.2 Distribution of Task Service Demand

We also consider that the distribution of task service demand changes in exponentially distributed time intervals e_1, e_2, e_3, …, e_m from exponential to Branching Erlang and vice versa (Figure 2). The mean time interval for distribution change is e. In both exponential and Branching Erlang cases, the mean task service demand is x.

Those jobs that arrive at the processors within the same time interval have the same distribution for their task service demand. However, during the same time interval, some jobs may have arrived during a past time interval and have a different distribution for their task service demand.

Tasks of the Branching Erlang distribution case have larger variability in their service demand than exponential tasks. A high variability in task service demand implies that there are proportionately a high number of service demands that are very small as compared with the mean service demand, and that there are a comparatively low number of service demands that are very large. When a task with a large service demand starts execution, it occupies its assigned processor for a long time interval and, depending on the scheduling policy, it may introduce inordinate queuing delays for other tasks that are waiting for service. The parameter which represents the variability in task execution time is the coefficient of variation of execution time C. In the exponential distribution case C=1 while in the Branching Erlang distribution case C >1.

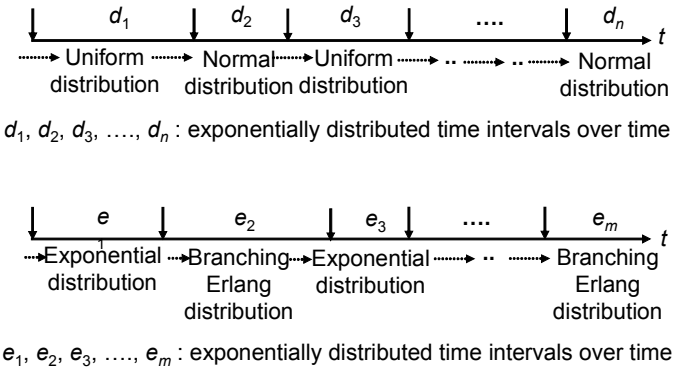

Figure 2. Exponentially varying with time distribution for job parallelism and for task service demand

2.1.3 Distribution of I/O Service Time

After a job leaves the processors, it requests service on the I/O unit. The I/O service times are exponentially distributed with mean z.

Each time a job returns from I/O service for scheduling on distributed processors, it is partitioned into a different number of tasks even if it arrives during the same time interval d_i in which case it executed last. All notations used in this paper are described in Table 1.

2.2 Scheduling Strategies

. Probabilistic routing – First-Come-First-Served (PrFCFS). With this policy, a task is dispatched randomly to processors with equal probability. The task dispatcher chooses one of the P processors based on the outcome of an independent trial in which the i^{th} outcome has probability $p_i = 1 / P$. Thereafter, the FCFS temporal scheduling policy is applied. This policy is the simplest to implement since it involves only a negligible amount of overhead when generating random numbers.

. Shortest Queue routing – FCFS (SQFCFS). This policy assigns each ready task to the currently shortest processor queue. The FCFS method is applied to the respective queue. In this case we do not take into account the overhead required to collect global system information about processors queues.

. Shortest Queue routing – FCFS - *Co* (SQFCFS-*Co*). In the SQFCFS case, assume that $Co = 0$. However, the shortest queue method invokes the scheduler every time a job demands processing service. Invoking the scheduler requires considerable communication overhead to collect load information from all processors. For this

reason, the SQFCFS-*Co* policy is considered where the overhead of collecting global system information is not negligible. The time required by the scheduler to make an assignment decision is considered to be uniformly distributed over the interval [*a*, *b*] with a mean of $Co > 0$. The availability of global system information comes with some communication costs.

. **Probabilistic routing – Shortest – Task - First (PrSTF)**. This policy assumes a-priori knowledge about a task in form of service demand. When such knowledge is available, tasks in the processor queues are arranged in a decreasing order of service demand. However, it should be noted that a-priori information is often not available and only an approximation of task execution time is available. Estimated processor service times are assumed to be uniformly distributed within ± *E*% of the exact value.

. **Probabilistic routing – Task of the job with the Smallest Number of Uncompleted Tasks First (PrSNUTF)**. This policy gives higher priority to tasks belonging to the job that has the smallest number of uncompleted tasks. This number is an indication of how close the job is to completion. This method does not use information about task execution time but it is obvious that it incurs an additional overhead, as the scheduler has to know which task in a queue belongs to the job that is closest to completion.

The PrSTF and PrSNUTF task scheduling strategies are vulnerable in the extreme cases where the service demand of a task or the number of uncompleted tasks of a job are too big. Since processor queues are rearranged each time a new task is entered in them, it is possible that some jobs are never scheduled. This problem is eliminated by the following policy, which is a version of STF.

. **Probabilistic routing – Limited STF (PrLSTF)**. With this policy, the STF method is applied $l = 10$ times and then the oldest task in the queue is scheduled. Therefore, the number of times that a task can be rejected from the first queue position when higher priority tasks have been inserted is limited.

When using priorities and a tie occurs, the FCFS method is used.

. **I/O scheduling**. For the I/O subsystem, the FCFS policy is employed.

2.3 Performance Metrics

Response time of a random job is the time interval from the dispatching of a job's tasks to processor queues, to service completion of the last task of the job.

Cycle time of a random job is the time that elapses between two successive processor service requests of this job. In our model cycle time is the sum of scheduling delay, plus response time, plus queuing and service time at the I/O unit.

Parameters used in later simulation computations are presented in Table 1.

Table 1. Notations

RT	Mean response time
RT_{max}	Maximum response time
ScD	Mean scheduling delay
K	Mean cycle time
R	System throughput
U	Mean processor utilization
N	Degree of multiprogramming
C	Coefficient of variation
x	Mean task service demand
z	Mean I/O service time
d	Mean time interval for the time varying distribution of the number of job tasks
e	Mean time interval for the time varying distribution of task service demand
Co	Mean communication overhead that is required to collect global system information
SD	Mean synchronization delay of tasks
E	Estimation error in service time

R represents system performance and K represents program performance. In the case of the SQFCFS-*Co* strategy, fairness is expressed as the maximum of the sum of the scheduling delay and the response time Max(ScD+RT), while in the remaining cases fairness is expressed by RT_{max}.

When each policy is compared to the PrFCFS, the relative (%) increase in R is represented as D_R. We also study the ratio of RT_{max} in each one of the SQFCFS, PrSTF, PrSNUTF, and PrLSTF cases over the corresponding value of the PrFCFS case.

When SQFCFS-*Co* is compared to the SQFCFS, the relative (%) increase in K is represented as D_K.

3. Simulation Results and Discussion

3.1 Model Implementation and Input Parameters

The queuing network model is simulated with discrete event simulation modeling ([15]) using the independent

replication method. For every mean value, a 95% confidence interval is evaluated. All confidence intervals are less than 5% of the mean values. The system considered is balanced (refer to Table 1 for notations):

$$x = 1.0, \quad z = 0.531$$

The reason $z = 0.531$ is chosen for balanced program flow is that there are on average 8.5 tasks per job at the processors. So, when all processors are busy, an average of 1.882 jobs are served each unit of time. This implies that I/O mean service time must be equal to 1/1.882 = 0.531 if the I/O unit is to have the same service capacity.

When the distribution for the task service demand changes from exponential ($C = 1$) to Branching Erlang ($C > 1$), in one set of experiments Branching Erlang distribution is considered with $C = 2$, while in the other set $C = 4$.

The degree of multiprogramming N is 16, 24, 32, 40, 48. The reason various numbers of programs N are examined is because it is a critical parameter that reflects the system load. In cases where estimation of service time is required, we have also examined estimation errors of ±10%, ±20%, and ±30%.

The mean time interval for distribution change is considered as $d = e = 10$, 20, 30. These are reasonable choices considering that the mean service time of tasks is equal to 1.

The results of two communication overhead cases are presented for the SQFCFS-*Co* policy. In one case, the communication overhead is uniformly distributed over the interval [0.49, 0.51] ($Co = 0.5$), while in the second case, the interval is [0.59, 0.61] ($Co = 0.6$). The $Co < 0.5$ case is also examined. However, those results are not presented because they are very close to the $Co = 0.5$ case results.

3.2 Performance Analysis

A large number of simulation experiments were conducted, but to conserve space, only a representative sampling of the experimental results is presented in this paper.

In Table 2, the range of mean processor utilization is presented for all cases examined when Co=0.

In Tables 3-8 performance metrics are presented for the SQFCFS and SQFCFS-*Co* policies in the $d = e = 30$ case for Co=0.5, and 0.6.

D_R versus N, for $d = e = 10$, 20, and 30 respectively in Figures 3, 4, and 5 for $C = 2$, and in Figures 6, 7, and 8 for $C = 4$ when Co=0.

Figures 9, 10, 11 show the RT_{max} ratio for $d = e = 10$, 20, and 30 at $C = 2$. Figures 12, 13, and 14 show the RT_{max} ratio at $C = 4$ when Co=0.

D_K versus N, for $d = e = 10$, 20, and 30 respectively in Figures 15, 16, and 17 for $C = 2$, and in Figures 18, 19, and 20 for $C = 4$. In these Figures $Co = 0.5$, and 0.6.

The following conclusions are drawn from the results.

3.2.1 Overall System Performance

With regard to processor load, in all cases examined the lower (higher) mean processor utilization is presented in the PrFCFS (SQFCFS) case respectively. PrSNUTF and PrLSTF yield almost the same utilization. At low N, the utilization in the PrSTF case is close to the utilization of the PrSNUTF and PrLSTF cases while at high N it is larger (Table 2).

Table 2. Mean processor utilization range (Co= 0)

Scheduling policy	$d = e = 10$	$d = e = 20$	$d = e = 30$
	U range, $C = 2$		
PrFCFS	0.64 – 0.84	0.64 – 0.83	0.65 – 0.83
SQFCFS	0.88 – 0.98	0.89 – 0.99	0.89 – 0.98
PrSTF	0.71 – 0.91	0.71 – 0.92	0.72 – 0.91
PrSNUTF	0.69 – 0.90	0.69 – 0.90	0.70 – 0.90
PrLSTF	0.70 – 0.88	0.69 – 0.90	0.70 – 0.89
	U range, $C = 4$		
PrFCFS	0.48 – 0.69	0.48 – 0.70	0.49 – 0.70
SQFCFS	0.82 – 0.96	0.82 – 0.96	0.82 – 0.96
PrSTF	0.51 – 0.80	0.52 – 0.81	0.55 – 0.81
PrSNUTF	0.50 – 0.76	0.52 – 0.76	0.53 – 0.78
PrLSTF	0.51 – 0.74	0.51 – 0.74	0.53 – 0.76

In all cases, the SQFCFS method performs better than all the other methods, while the worst performance is encountered in the PrFCFS policy. The PrSTF method performs better that the PrSNUTF and PrLSTF policies. However, the difference in performance between SQFCFS and PrSTF is much higher than the difference between PrSTF and each one of PrSNUTF and PrLSTF. In some cases, PrSNUTF performs better than PrLSTF while in other cases the two methods exhibit similar performance.

The superiority of SQFCFS over the rest of the methods decreases with an increasing degree of multiprogramming. This is due to the fact that when the probabilistic routing policy is employed, it is more probable for the processors to be idle due to unbalanced processor queues at small N than at large N. Therefore, at low N, the abilities of the SQFCFS policy are better exploited. The change in performance for the rest methods due to increasing N does not follow a specific

pattern and also is less significant than the change in performance of SQFCFS.

Also, the superiority of the SQFCFS strategy over the other methods is more significant at $C = 4$ than at $C = 2$. This is due to the fact that tasks present larger variability in their service demand when $C = 4$ than when $C = 2$. When a task with a large service demand starts execution, it may introduce inordinate queuing delays to other tasks. This may cause long synchronization delays in their sibling tasks. Furthermore, during this time the I/O subsystem may starve only to later become deluged with jobs that must spend large amounts of time waiting in the I/O queue. The SQFCFS method alleviates this problem as it does not send tasks to a queue that is already long.

However, the variability in task service demand impacts the performance of the other methods to a lesser degree than the performance of SQFCFS. Furthermore, in some cases these methods perform slightly better in the $C = 2$ case than at $C = 4$, while in other cases they perform better for $C = 4$ than for $C = 2$.

Additional simulation experiments were conducted to assess the impact of service time estimation error on the performance of scheduling methods that require a-priori knowledge of task service demands (PrSTF and PrLSTF strategies). The estimation error in these experiments is set at ±10%, ±20%, and ±30%. Simulation results reveal that the estimation error in processor service time marginally affects performance. Therefore, no profit is gained from the a priori knowledge of exact service times.

3.2.2 Fairness of Job Service

In all cases, the smallest RT_{max} ratio is presented in the SQFCFS case and it is less than or equal to 1. Therefore, the SQFCFS method is the fairest of all other methods that we examined. The PrLSTF method gives larger RT_{max} than the PrFCFS method, but RT_{max} is much smaller in the PrLSTF case than in the PrSTF and PrSNUTF cases.

In most cases the most unfair method is the PrSTF in that it results in long queuing delays for large tasks in processors queues. In few cases, PrSNUTF gives larger RT_{max} than PrSTF. This is because scheduling the job that has the smallest number of uncompleted tasks first may result in giving priority to some tasks that are large. In these cases, the strategy results in larger queuing delays than the PrSTF method.

PrSTF and PrSNUTF strategies present significantly larger RT_{max} ratios in the $C = 2$ case than at $C = 4$. This is because when $C > 1$, when a large task is served by a processor and there are other tasks waiting in this processor queue, the response time mainly depends on the large task execution time. This is because queued tasks most probably have a very small service demand as compared with the large task. Therefore, the sequence they are served does not significantly affect the response time. Since the variability in task service demand is higher at $C = 4$ than at $C = 2$, RT_{max} ratios are larger at $C = 2$ than at $C = 4$.

3.2.3 The impact of the Communication Overhead

In Tables 3-8, observe that the mean "scheduling delay + response time" of jobs in the SQFCFS-*Co* case is larger than the mean response time in the SQFCFS case. However, the delay of jobs at the scheduler in the SQFCFS-*Co* case has as a consequence of yielding smaller mean task synchronization delays and smaller mean response times than that of the SQFCFS case.

Results show that when $Co = 0.5$ the performance of SQFCFS and SQFCFS-*Co* does not differ significantly. Similar results (not presented here) were observed for $Co < 0.5$ cases.

However, for $Co = 0.6$ the SQFCFS-*Co* case communication overhead degrades performance. This is because program flow is not balanced in the SQFCFS-0.6 case, since the scheduler service capacity is smaller than the service capacity of the I/O subsystem and processors ($1/Co < 1.882$). Therefore, programs are blocked while the scheduler collects global system information to make assignment decisions. This results in lower mean processor utilization as shown in Tables 3-8. Results also show that for $Co = 0.6$, performance deterioration due to scheduling communication overhead increases with increasing degree of multiprogramming (Figures 15-20). Also, the difference in performance between the SQFCFS and SQFCFS-0.6 cases is almost the same for all d, and is larger in the $C = 2$ case than in the $C = 4$ case. For the SQFCFS-0.6 case, the largest D_K is about 10 percent.

The maximum "scheduling delay + response time" of jobs in the SQFCFS-*Co* case is not significantly larger than RT_{max} of the SQFCFS case (Tables 3-8). Therefore, SQFCFS-*Co* is almost as fair as SQFCFS.

3.2.4 General Remarks

All of the above scheduling schemes have merit. PrFCFS is the simplest to implement since it involves only a negligible amount of overhead when generating random numbers. It is apparent that PrFCFS results in sub-optimal system performance. However, it never activates the scheduler, as it does not make decisions that depend on system-state or job characteristics.

The SQFCFS method requires global knowledge of queue length on job arrival and also sorts queues into decreasing queue length order. So, the scheduler is called upon every time a job arrives. This policy yields the best overall system performance and also is the fairest of all the methods examined. Furthermore, when scheduling overhead is taken into account (SQFCFS-*Co* method) up

to a certain value of *Co* (which depends on system characteristics), performance is very close to SQFCFS.

The PrSTF and PrLSTF methods need a-priori information about service demand of local tasks when they make decisions. However, advance information comprised of even an approximation of task service demand is available only in some cases. On the other hand, the PrSNUTF method needs information about the status of siblings of all local tasks and it needs to process this information in order to determine which task has the smallest number of uncompleted sibling tasks. Among these three methods, PrSTF performs better but its superiority is not significant as compared with the performance of the SQFCFS method. Furthermore, in most cases PrSTF is the most unfair method. The fairest method among these three policies is PrLSTF. However, in most cases it performs worse than the other two methods. The extent of its superiority over the PrFCFS policy does not justify its complexity.

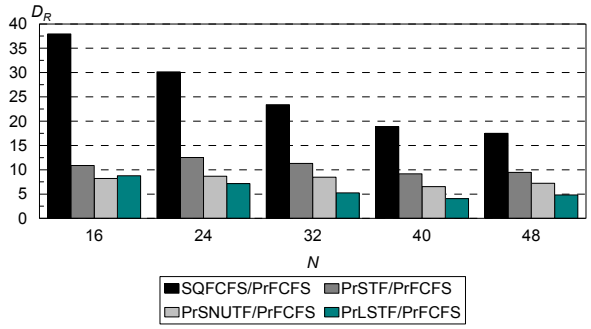

Figure 3. D_R versus N, $d = e = 10$, C=2

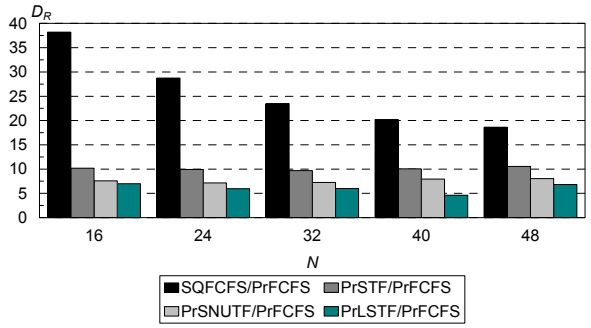

Figure 4. D_R versus N, $d = e = 20$, C=2

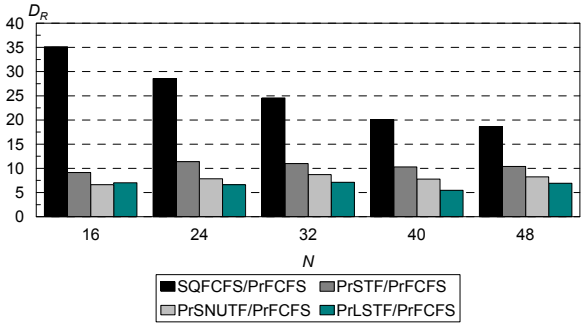

Figure 5. D_R versus N, $d = e = 30$, C=2

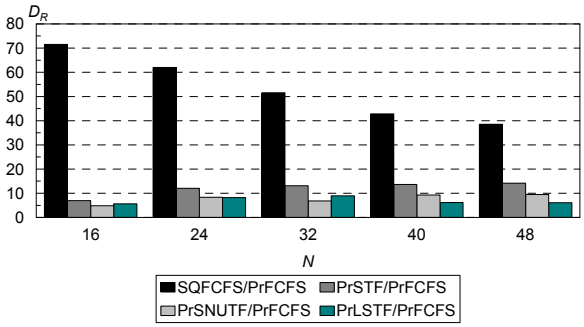

Figure 6. D_R versus N, $d = e = 10$, C=4

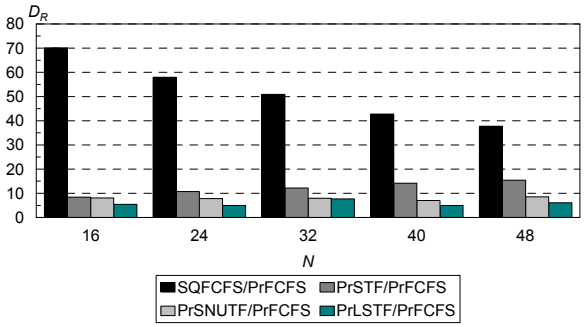

Figure 7. D_R versus N, $d = e = 20$, C=4

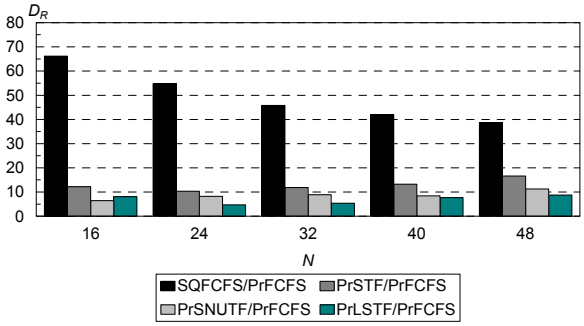

Figure 8. D_R versus N, $d = e = 30$, C=4

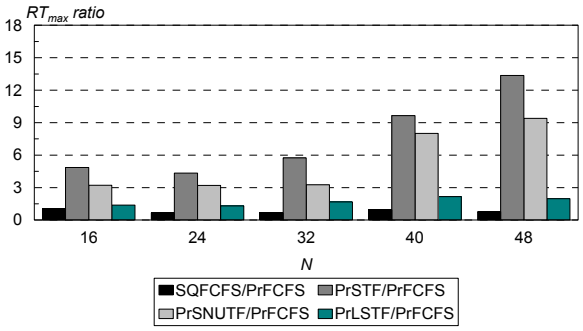

Figure 9. RT_{max} ratio versus N, $d = e = 10$, $C=2$

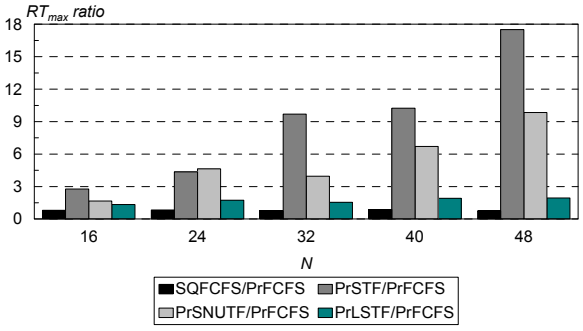

Figure 10. RT_{max} ratio versus N, $d = e = 20$, $C=2$

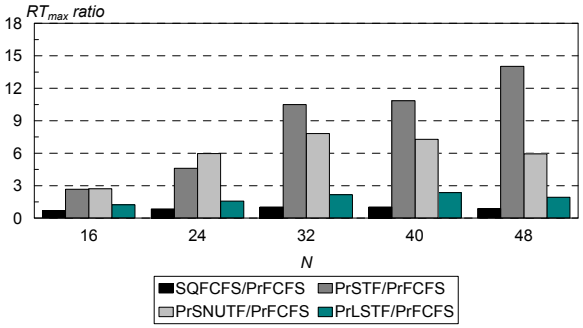

Figure 11. RT_{max} ratio versus N, $d = e = 30$, $C=2$

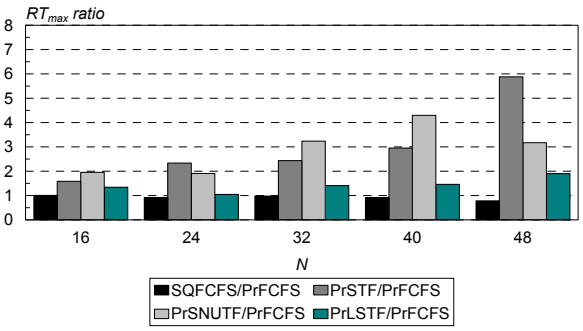

Figure 12. RT_{max} ratio versus N, $d = e = 10$, $C=4$

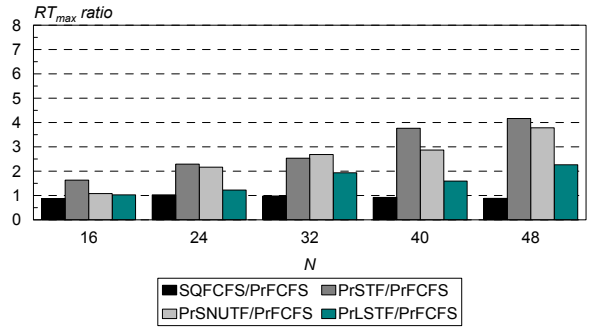

Figure 13. RT_{max} ratio versus N, $d = e = 20$, $C=4$

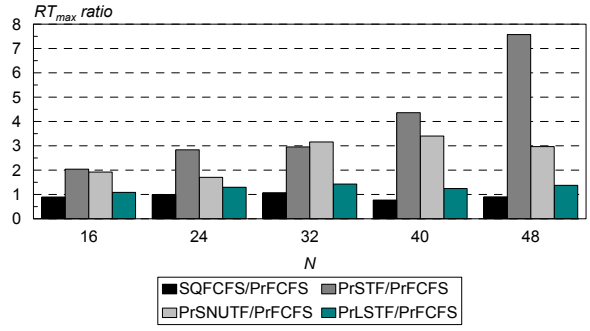

Figure 14. RT_{max} ratio versus N, $d = e = 30$, $C=4$

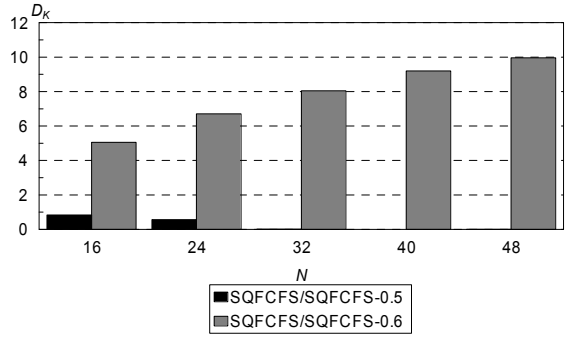

Figure 15. D_K versus N, $d = e = 10$, $C=2$

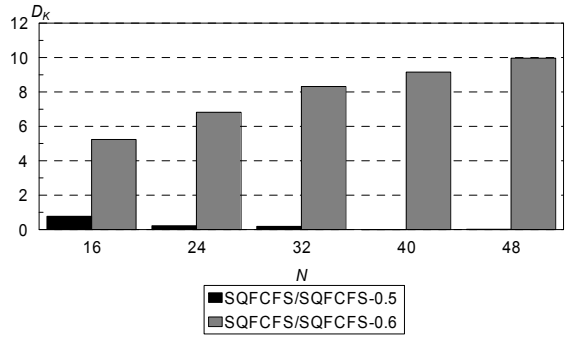

Figure 16. D_K versus N, $d = e = 20$, $C=2$

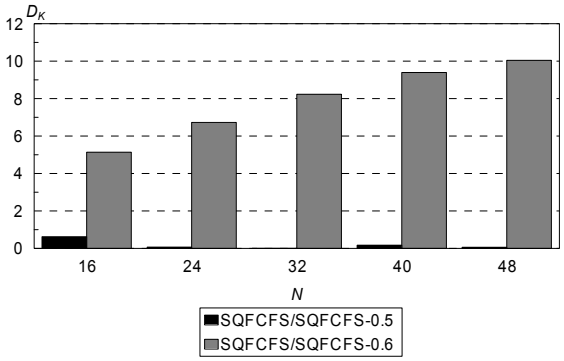

Figure 17. D_K versus N, d = e = 30, C=2

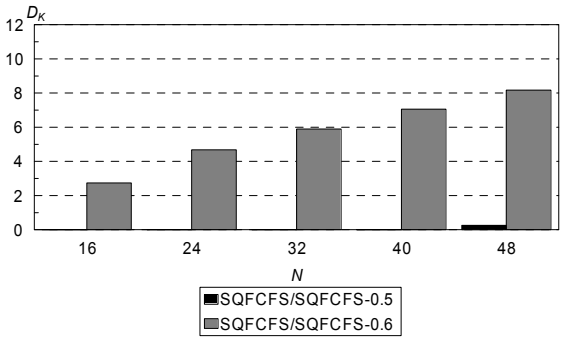

Figure 18. D_K versus N, d = e = 10, C=4

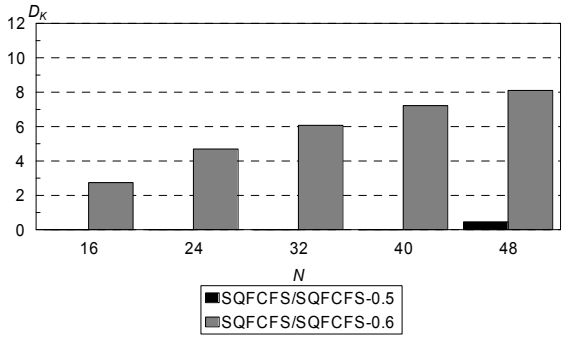

Figure 19. D_K versus N, d = e = 20, C=4

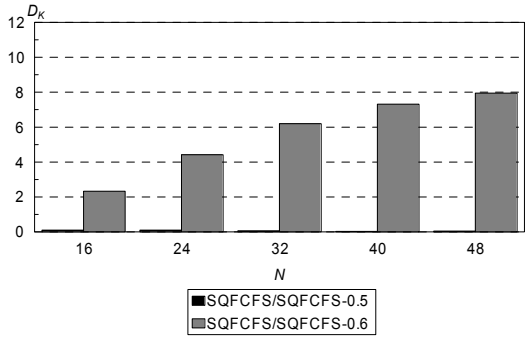

Figure 20. D_K versus N, d = e = 30, C=4

Table 3. d=30, C=2, SQFCFS

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.89	6.72	6.72	5.84	52.92	9.63	1.66
0.93	8.78	8.78	7.09	57.09	13.63	1.76
0.96	10.81	10.81	8.16	60.31	17.76	1.80
0.97	12.82	12.82	9.15	64.65	21.93	1.82
0.98	14.90	14.90	10.08	69.61	26.14	1.84

Table 4. d=30, C=2, SQFCFS-0.5

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.88	5.65	7.01	5.11	53.22	9.69	1.65
0.93	7.16	9.06	6.09	58.36	13.64	1.76
0.96	8.76	11.06	7.00	60.52	17.76	1.80
0.97	10.48	13.01	7.89	67.27	21.97	1.82
0.97	12.21	15.03	8.74	74.84	26.16	1.83

Table 5. *d*=30, C=2, SQFCFS-0.6

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.84	4.93	7.84	4.60	54.51	10.12	1.58
0.87	5.39	11.43	4.93	58.69	14.55	1.65
0.88	5.58	15.74	5.05	61.53	19.22	1.66
0.88	5.67	20.42	5.10	67.29	23.99	1.67
0.89	5.71	25.10	5.11	74.90	28.77	1.67

Table 6. *d*=30, C=4, SQFCFS

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.82	7.89	7.89	7.23	151.19	10.47	1.53
0.89	10.07	10.07	8.87	152.40	14.41	1.67
0.92	12.21	12.21	10.37	156.51	18.42	1.74
0.94	14.41	14.41	11.73	158.31	22.57	1.77
0.96	16.34	16.34	12.88	163.84	26.77	1.79

Table 7. *d*=30, C=4, SQFCFS-0.5

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.81	6.75	8.03	6.33	151.25	10.48	1.53
0.89	8.50	10.21	7.66	153.32	14.42	1.67
0.92	10.26	12.38	8.91	159.05	18.43	1.74
0.94	12.08	14.45	10.08	159.93	22.58	1.77
0.95	13.79	16.44	11.13	163.99	26.78	1.79

Table 8. *d*=30, C=4, SQFCFS-0.6

U	*RT*	*ScD+RT*	*SD*	*Max(ScD+RT)*	*K*	*R*
0.79	6.12	8.64	5.82	152.42	10.71	1.49
0.84	7.15	12.01	6.70	154.37	15.04	1.60
0.87	7.92	15.65	7.26	159.16	19.56	1.64
0.87	8.33	19.85	7.55	161.88	24.22	1.65
0.88	8.78	24.07	7.83	166.39	28.90	1.66

4. Conclusions and Further Research

This paper studies parallel job scheduling in homogeneous distributed systems. It presents comprehensive evaluations of task scheduling alternatives using synthetic workloads. Distributions that vary with time are used to represent the number of parallel tasks per job and task service demand. These workloads are used because they are more realistic than other distributions found in other research papers. The impact of different workload parameters on performance metrics is also examined. The objective is to identify conditions that produce good overall system performance while maintaining fair individual job execution times. Simulation is used to generate results for different configurations.

A number of parallel scheduling policies are analyzed and compared. Simulation results indicate that the SQFCFS policy performs much better than the other methods examined and also is the most fair. It is more superior at lower degrees of multiprogramming and when task service demand involves time intervals with large differences in the service demand variability. Furthermore, results show that, up to a certain value of the mean communication overhead, performance of the SQFCFS method is only marginally affected. Generally, it can be concluded that even though overhead can degrade SQFCFS, in many cases its performance remains high in comparison to other methods.

The worst system performance is produced by the PrFCFS algorithm, which uses probabilistic routing and FCFS task scheduling. All of the remaining methods use probabilistic routing and need information about jobs to make their decisions. They perform better than PrFCFS but are not as fair, and involve considerable implementation overhead.

A logical extension to this research is to examine the impact of the communication overhead on the performance of an open queuing network model of a distributed system.

5. References

[1] Dandamudi, S. 1994. Performance Implications of Task Routing and Task Scheduling Strategies for Multiprocessor Systems. *Proceedings of the IEEE-Euromicro Conference on Massively Parallel Computing Systems*, IEEE Computer Society, pp. 348-353, 2-6 May, Ischia, Italy.

[2] Dandamudi, S. 1997. The Effect of Scheduling Discipline on Dynamic Load Sharing in Heterogeneous Distributed Systems. *Proceedings of the 5th International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS '97), IEEE Computer Society, pp. 17-24, 12-15 January, Haifa, Israel.

[3] Dandamudi, S. 1998. Sensitivity Evaluation of Dynamic Load Sharing in Distributed Systems. *IEEE Concurrency*, July-September: 62-72.

[4] Kafil M., and I. Ahmad. 1998. Optimal Task Assignment in Heterogeneous Distributed Computing Systems. *IEEE Concurrency*, July-September: 42-51.

[5] Karatza, H.D. 1997. Simulation Study of Task Scheduling and Resequencing in a Multiprocessing System. *Simulation Journal*, Special Issue: Modelling and Simulation of Computer Systems and Networks: Part Two, SCS, San Diego, CA, USA. 68(4): 241-247.

[6] Karatza, H.D. 2000. Scheduling Strategies for Multitasking in a Distributed System. *Proceedings of the 33rd Annual Simulation Symposium*, IEEE Computer Society, pp. 83-90, 16-20 April, Washington D.C., USA.

[7] Karatza, H.D. 2000. A Comparative Analysis of Scheduling Policies in a Distributed System using Simulation, *International Journal of SIMULATION Systems, Science & Technology*, UK Simulation Society, Nottingham, UK, Vol. 1(1-2): 12-20.

[8] Talby, D., and D.G. Feitelson. 1999. Supporting Priorities and Improved Utilization of the IBM SP2 Scheduler using Slack-based Backfilling. *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, IEEE Computer Society, pp. 513-517, 12-16 April, San Juan, Puerto Rico.

[9] Zhang, Y., H. Franke, J. Moreira, and A. Sivasubramaniam. 2000. The Impact of Migration on Parallel Job Scheduling for Distributed Systems. *Proceedings of Europar*, pp. 242-251, 29 August - 2 September, Munich, Germany.

[10] Zhang, Y., and A. Sivasubramaniam. 2001. Scheduling Best-effort and Real-time Pipelined Applications on Time-shared Clusters. *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures,* ACM, pp. 209-219, 4-6 July, Crete Island, Greece.

[11] Karatza, H.D., and R.C. Hilzer. 2003. Performance Analysis of Parallel Job Scheduling in Distributed Systems. *Proceedings of the 36th Annual Simulation Symposium* IEEE Computer Society, pp. 109-116, 30 March - 2 April, Orlando, Florida.

[12] Bolch, G., S. Greiner, H. De Meer, and K.S. Trivedi. 1998. *Queueing Networks and Markov Chains*. New York, NY: 1998.

[13] Kumar, A., and R. Shorey. 1993. Performance Analysis and Scheduling of Stochastic Fork-Join Jobs in a Multicomputer System. *IEEE Transactions on Parallel and Distributed Systems*, IEEE Computer Society, 4(10): 1147-1162.

[14] Rosti, E., G. Serazzi, E. Smirni, and M. Squillante. 1998. The Impact of I/O on Program Behavior and Parallel Scheduling. *Performance Evaluation Review*, ACM, 26(1): 56-65.

[15] Law, A., and D. Kelton. 1991. *Simulation Modelling and Analysis*. New York, NY: McGraw-Hill.

Helen D. Karatza *is an Associate Professor in the Department of Informatics at the Aristotle University of Thessaloniki, Greece. Her research interests include Computer Systems Performance Evaluation, Multiprocessor Scheduling, Mobile Agents, Distributed Systems and Simulation. Her email is <karatza@csd.auth.gr>.*

Ralph C. Hilzer *is a Professor of Computer Science at the California State University, Chico, USA. His research interests are in the areas of Operating Systems, Parallel Processing, Languages and Compilers. His email is <rhilzer@csuchico.edu>.*