Performance Analysis of Parallel Job Scheduling in Distributed Systems

Helen D. Karatza
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
karatza@csd.auth.gr

Ralph C. Hilzer
Computer Science Department
California State University, Chico
Chico, California 95929-0410 USA
hilzer@ecst.csuchico.edu

Abstract

This paper studies parallel job scheduling in a distributed system. A simulation model is used to address performance issues associated with scheduling. Five policies are used to schedule parallel jobs over a variety of workloads. Fairness is required among competing jobs. We examine a case where the distribution of the number of parallel tasks per job and also the distribution of task service demand vary with time. Simulated results indicate that although all scheduling methods have merit, one method significantly improves the overall performance and also guarantees fairness in terms of individual job execution.

1. Introduction

The scheduling of parallel jobs on the processors of a distributed system has always been an important and challenging area of research. However, in spite of extensive research it is still not always known how to efficiently schedule parallel jobs. To determine this, it is critical to partition the program into tasks properly, assign the tasks to processors and then schedule execution on a distributed processor. Good scheduling policies are needed to improve system performance while preserving individual application performance so that some jobs do not suffer unbounded 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 [2] 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 [3] 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.

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

A different type of parallel job scheduling is considered in [4], [9], [10], and [11] where parallel tasks are required to start at essentially the same time, coordinate 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. Five 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

Proceedings of the 36th Annual Simulation Symposium (ANSS'03)
1080-241X/03 $17.00 © 2003 **IEEE**

this reason this paper proposes 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).

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.

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 while the results of the simulation experiments are presented and analyzed in section 4. Finally section 5 summarizes findings and offers suggestions for further research.

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 with negligible communication delays. 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.

The effects of the memory requirements and the communication latencies are not represented explicitly in the system model. Instead, they appear implicitly in the shape of the job execution time functions. By covering several different types of job execution behaviors, we expect that various architectural characteristics will be captured.

In the simulation experiment, 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. ([8]) 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.

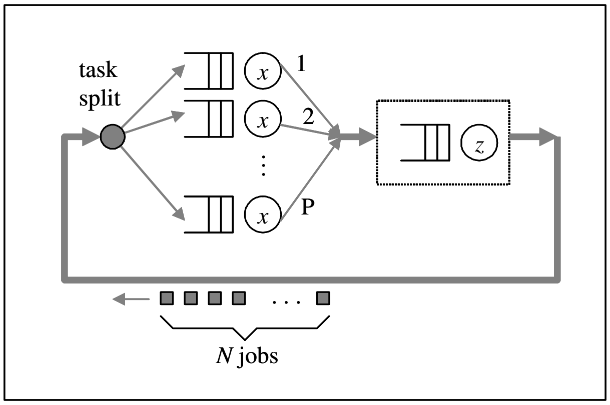

Figure 1. The queuing network model

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 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 ([1]) 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.

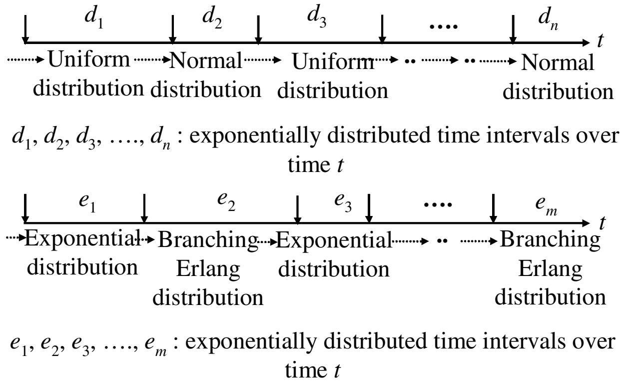

Figure 2. Exponentially varying with time distribution for job parallelism and for task service demand

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.

. **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.

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

Table 1. Notations

RT_{max}	Maximum response 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 varying with time distribution of the number of job tasks
e	Mean time interval for the varying with time distribution of task service demand
E	Estimation error in service time

R represents system performance while RT_{max} represents fairness of the policy employed. 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.

3. Experimental methodology

The queuing network model is simulated with discrete event simulation modeling ([7]) 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.

4. 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.
- 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 $C = 4$.
- 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 $C = 4$.

The following conclusions are drawn from the results:

4.1 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

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.

4.2 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 $C = 4$.

4.3 General remarks

All five 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 to make decisions every time a job arrives. However, this policy yields the best overall system performance and also it is the fairest of all the methods examined.

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.

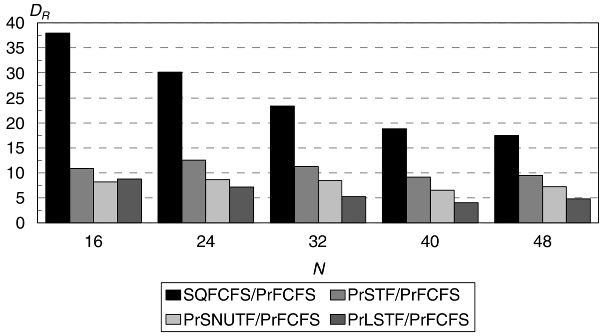

Figure 3. D_R versus N, $d = e = 10$, C=2

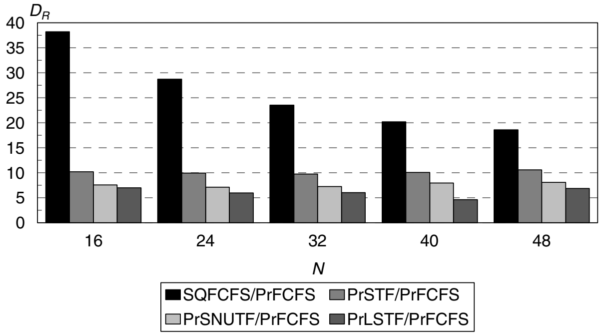

Figure 4. D_R versus N, $d = e = 20$, C=2

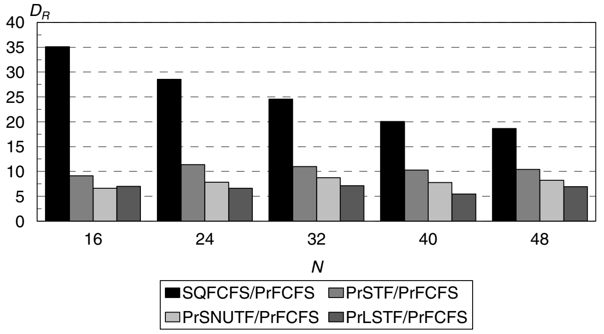

Figure 5. D_R versus N, $d = e = 30$, C=2

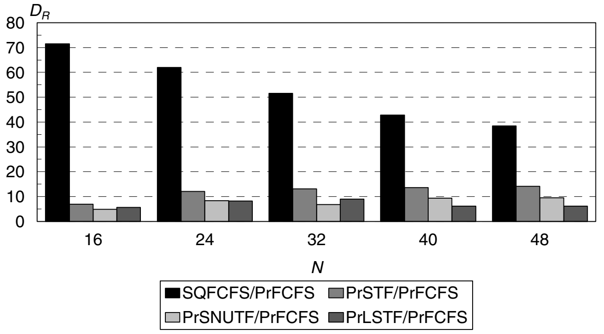

Figure 6. D_R versus N, $d = e = 10$, C=4

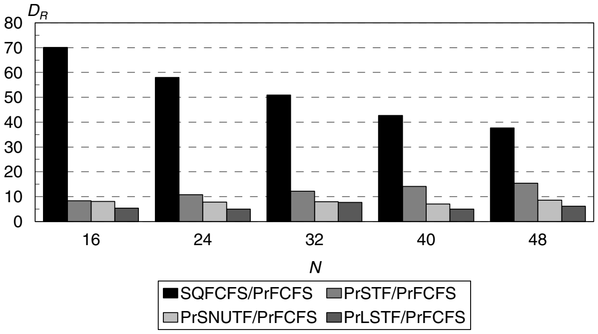

Figure 7. D_R versus N, $d = e = 20$, C=4

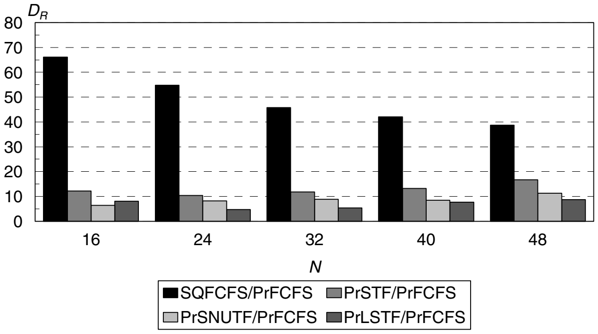

Figure 8. D_R versus N, $d = e = 30$, C=4

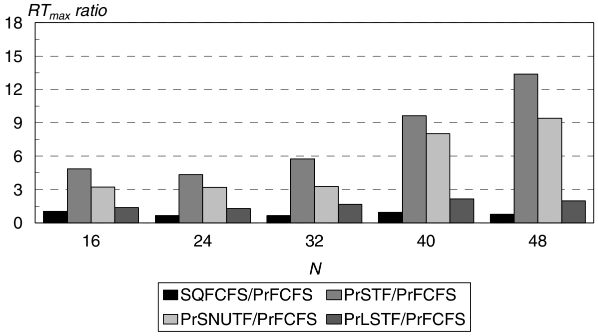

Figure 9. RT_{max} ratio versus N, $d = e = 10$, C=2

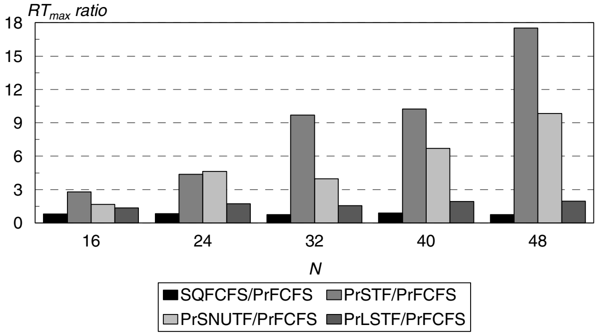

Figure 10. RT_{max} ratio versus N, $d = e = 20$, C=2

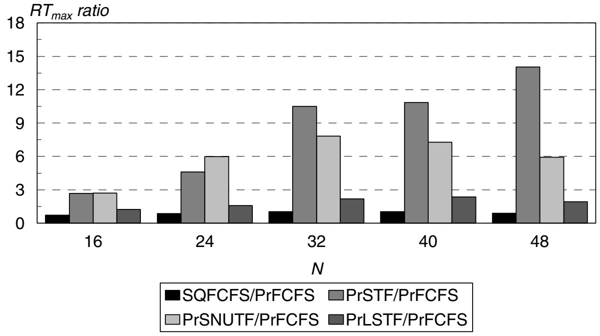

Figure 11. RT_{max} ratio versus N, $d = e = 30$, C=2

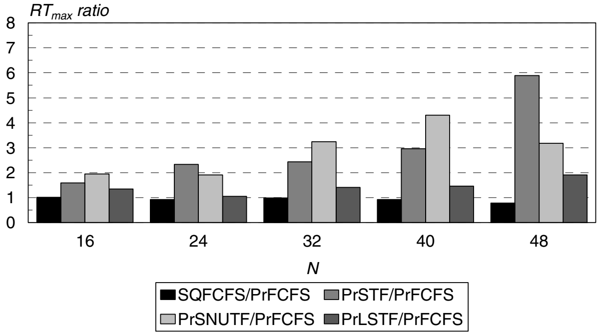

Figure 12. RT_{max} ratio versus N, $d = e = 10$, C=4

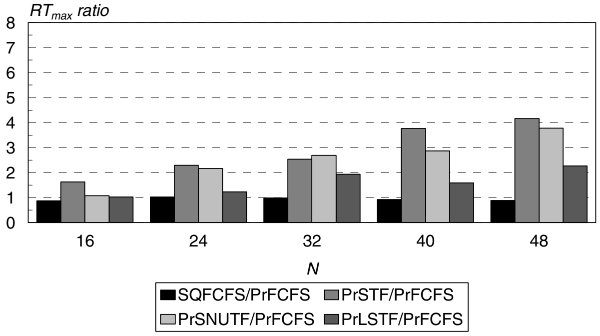

Figure 13. RT_{max} ratio versus N, $d = e = 20$, C=4

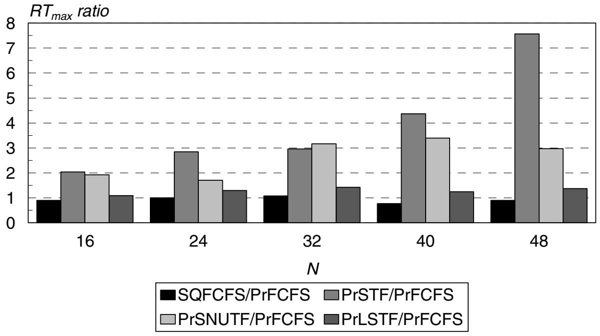

Figure 14. RT_{max} ratio versus N, $d = e = 30$, C=4

5. Conclusions and further research

This paper studies parallel job scheduling in a distributed system. It presents a comprehensive evaluation of different task scheduling alternatives using synthetic workloads. Distributions are proposed that vary with time for the number of parallel tasks per job and for task service demand. We use these workloads because they are more realistic than other distributions that have been referred in other research papers. The impact of different workload parameters on performance metrics is examined. The objective is to identify conditions that produce good overall system performance while maintaining fairness of individual job execution times. We use simulation as the means of generating results with different configurations.

Five parallel scheduling policies are analyzed and compared. Simulation results reveal that the SQFCFS policy which combines the Shortest Queue routing criteria and the FCFS temporal task scheduling method performs much better than the other methods examined and also is the fairest policy. Its superiority is higher at lower degrees of multiprogramming and also is higher when the varying with time distribution of task service demand involves time intervals with large differences in the service demand variability.

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

Overhead associated with the SQFCFS method is not accounted for in this current work. However, since this is not a large distributed system and overhead was not considered with the other methods either, we expect this policy would still outperform methods if overhead were considered. A logical extension to this research is to examine large distributed systems and to include the impact on them of overhead required to collect and process global system information.

References

[1] Bolch, G., S. Greiner, H. De Meer, and K.S. Trivedi, *Queueing Networks and Markov Chains*, J. Wiley & Sons Inc., New York, 1998.

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

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

[4] H.D. Karatza, “A Simulation-based Performance Analysis of Gang Scheduling in a Distributed System”, In *Proceedings of 32nd Annual Simulation Symposium*, IEEE Computer Society, San Diego, CA, April 11-15, 1999, pp. 26-33.

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

[6] H.D. Karatza, “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), Dec. 2000, pp. 12-20.

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

[8] E. Rosti, G. Serazzi, E. Smirni, and M. Squillante, “The Impact of I/O on Program Behavior and Parallel Scheduling”, *Performance Evaluation Review*, ACM, New York, USA, Vol. 26 (1), 1998, pp. 56-65.

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

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

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