Performance Issues of Task Routing and Task Scheduling with Resequencing in Homogeneous Distributed Systems

Anthony Karageorgos and Helen Karatza, Ph.D.
Dept. of Informatics
Aristotle University
54006 Thessaloniki, Greece
{karageor, karatza}@csd.auth.gr

Abstract

An important part of a distributed system design is the workload sharing among the processors. This includes partitioning the arriving jobs into tasks that can be executed in parallel, assigning the tasks to processors and scheduling the task execution on each processor. In many system contexts jobs must depart in the order of their arrival, hence the resequence problem is involved.

In this paper we examine the efficiency of two task routing strategies — one static and one adaptive — and three non-preemptive task scheduling policies in conjunction with job resequencing before departure.

It is shown that the adaptive task routing strategy outperforms the static one and that when adaptive task routing is applied, the scheduling strategy affects marginally system performance. The minimum resequence delay is achieved with probabilistic task routing and $FCFS$ task scheduling.

1. Introduction

Enhancements in microprocessor performance and high-speed networking have led to increased interest in the use of distributed computing systems for parallel computing.

Distributed computing systems are composed of several loosely-coupled workstations communicating over a network. Networked workstations can offer parallel processing at relatively low cost [1], since they rely solely on commodity hardware and software.

One of the biggest issues in such systems is the development of effective techniques for the distribution of the processes of a parallel program on multiple processor nodes. A better exploitation of such organizations is achieved by partitioning each job into several tasks that can be run in parallel. A major problem is how to distribute the tasks among processing elements to achieve some performance goals such as minimizing job execution time, minimizing communication and other overhead and/or maximizing resource utilization.

[5] defines task routing as the method that tasks are assigned to processors and task scheduling as how tasks are scheduled on the assigned processor. Task routing strategies that only use information about the average behavior of the system, ignoring the current state, are called static policies [5]. Static policies may be either deterministic or probabilistic. Probabilistic policies base the routing decision on a probability distribution. In deterministic policies once the set of currently ready tasks has been specified, the routing discipline is applied. Policies that react to system state are called adaptive policies [5]. Adaptive policies are more complex but produce significantly better performance results than static policies.

Ready tasks can be arranged in a single global queue or each processor can maintain its own local queue. [5] refers to each one of these cases as centralized and distributed organization respectively.

The efficiency of the distributed task queue organization depends on various factors. In this paper we examine the impact of the job resequencing on the performance of different task routing and task scheduling strategies.

We consider an open queuing network model with N=128 homogeneous processor nodes each serving its own local task queue. Resequencing of jobs (programs) after processing ensures that jobs leave the system on a first-in-first-out basis.

Upon arrival each job is partitioned into independent tasks that can be run in parallel. The number of tasks depends on the degree of parallelism that is inherent in the job (program) and on the partitioning mechanism applied by an external task generator and scheduler.

We consider that a parallel program has a simple fork-join structure. An arriving job forks into independent tasks. Each task is attached to a processor queue according to the

1080-241X/97 $10.00 © 1997 IEEE

task routing strategy and the task scheduling strategy is applied to schedule its execution. No preemption or jockeying between task queues can occur. Executed tasks are gathered at the join point waiting for their siblings to join. Consequently, synchronization between tasks must take place.

A job is able to leave the system when all its tasks have been executed and joined. In this study we do not examine the partitioning/joining mechanism. Resequencing of jobs after service maintains the arrival order, ensuring thus that jobs leave the system on a first-in-first-out basis.

The resequencing problem is met in many system contexts including industrial production systems, computer communication systems and distributed computer systems. In such systems customers who enter the system must depart in the same order as their arrival. Hence, after service completion, disordered customers wait in a special buffer to rearrange their sequences, i.e. until the preceding customer leaves the system. The special buffer is termed usually resequencing buffer and the delay due to resequencing is called resequence delay.

Several authors have studied the resequencing delay in multiserver queues [7, 11, 15, 20, 22]. All these studies have considered open queuing network models and exponential task service times.

Multitasking and resequencing in a homogeneous distributed system with two processors is studied in [9]. That work studies closed queuing network models where half of the the total number of jobs consist of two independent tasks which can be processed in parallel.

Substantial literature exists on task routing and task scheduling strategies. [12] compared the performance of four task ready queue organizations. A study of the performance of the distributed task queue organization has been done by [2].

An extensive and thorough study on task routing and task scheduling for multiprocessor systems has been conducted by [4, 5]. These works examine many cases in detail. An open queuing network model of a multiprocessor system that consists of N=64 processors is considered and the number of tasks per job is exponentially distributed.

Analytical models for shared memory multiprocessors that execute fork-join parallel programs were developed by [21]. They show that the fork-join job structure can sufficiently model a large number of parallel applications.

[14] developed algorithms for distributed scheduling of tasks with deadlines and resource requirements. A distributed drafting algorithm for load balancing was suggested by [13]. In [18] three adaptive, decentralized controlled, job scheduling algorithms were proposed. All these studies considered random routing only and focused on scheduling methods to balance the system load. In addition, in all these works there is not any partitioning of jobs into tasks.

In this paper we consider an open queuing network model of a homogeneous distributed system consisting of N=128 processor nodes. The number of tasks per job is exponentially distributed with mean 128 or uniformly distributed between 1 and 128. We aim to compare the performance of two task routing and three task scheduling techniques, in conjunction with subsequent job resequencing, for various coefficients of variation of the processor service times (exponential and hyperexponential distributions) and for two cases of number of tasks per job (exponential and uniform number of tasks per job).

The structure of the paper is as follows. In the next section we describe our system model and in Sec. 2.2 the task routing and task scheduling strategies to be examined via simulation. In Sec. 2.3 we specify the metrics employed in assessing the performance of the various strategies. Furthermore, we describe the model input parameters and some implementation issues in Sec. 3.1 and 3.2. Finally, the results of the simulation experiments are presented and analyzed in Sec. 3.3 and conclusions are drawn in Sec. 4.

2. Model and Methodology

2.1. Model Description

An open queuing network model of a homogeneous distributed system is considered (Figure 1). It consists of N=128 independent processor nodes. Due to the decreasing cost of computer hardware the number of nodes in distributed computer systems increases. Therefore we conducted the simulation experiments using N=128.

The existence of an external central task generator and scheduler, not included in the queuing network, is assumed. This front-end subsystem is responsible for receiving arriving jobs, splitting them into tasks, and inserting the tasks into an appropriate task queue in system memory. Since our interest is in the performance of the task routing and task scheduling with job resequencing we assume that any communication delay due to the front-end operation is negligible.

The job structure is considered to be of simple fork-join type. In this context a fork-join job is composed of a set of independent tasks that are executed on the system concurrently.

After arriving at the system, jobs are partitioned into independent tasks that can be run in parallel. The number of tasks depends on the degree of parallelism that is inherent in the job (program) and on the partitioning mechanism applied by the central scheduler. We model this by associating a probability distribution with the number of tasks a job is partitioned into. In this study we consider uniform as well as exponential number of tasks per job with mean T. The aim is to compare the system behavior in the cases that the

number of tasks per job is uniform and exponential respectively.

Each processor is attached a private task queue. Tasks are assigned to a processor queue according to the task routing policy and are executed accordingly to the scheduling discipline. Tasks within a processor queue do not communicate with each other. No jockeying or preemption is permitted.

Tasks corresponding to the same job are joined before leaving the system.

A job completes execution when all its component tasks are completed. Therefore, each task that completes its service waits at the join point for the rest of the job component tasks to finish execution. The increased parallelism in fine-grain jobs results in the synchronization delay that occurs when tasks wait for their siblings to finish execution. In this work we do not examine the partitioning/joining mechanism.

Due to the resequencing problem requirements, jobs must leave the system in the order of their arrivals. Therefore, each job after execution is placed in the resequencing buffer. If all preceding jobs in the arrival order have already departed, the job leaves the system instantaneously. Otherwise it remains in the resequencing buffer until the departure of all jobs that have arrived before it. Apart from the resequence delay we consider that any other overhead caused by resequencing is negligible.

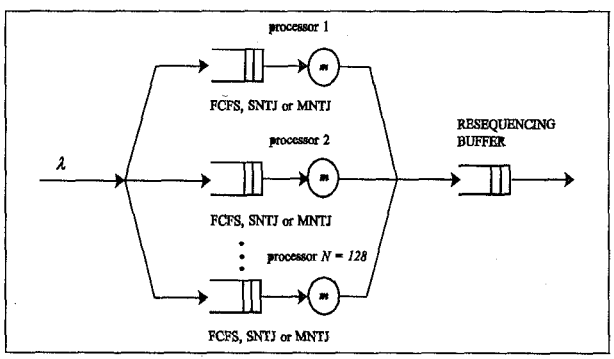

Figure 1. The queueing network model

Job interarrival times are (IID) exponential random variables (with arrival rate λ). Two cases of task service time distributions are examined:

1. Task service times are independent and identically distributed (IID) exponential random variables with mean m at each processor ($C = 1$).

2. Task service times are (IID) and hyperexponentially distributed. The coefficient of variation is C, where $C > 1$ and the mean is m at each processor.

All notations used in this paper are included in Table 1.

Table 1. Notations

N	Number of processor nodes
λ	Job arrival rate
m	Processor mean service time
C	Coefficient of variation of processor service time
T	Mean number of tasks per job
p	Maximum number of rejections for the first element in a task queue
ρ	System utilization
RT	Job response time
RD	Job resequence delay
TS	Job time in system
MRT	Mean job response time
MRD	Mean job resequence delay
MTS	Mean job time in system
UTL	Mean processor utilization
THR	System throughput rate

2.2. Task Routing and Task Scheduling Strategies

We study two different task routing strategies, one static and one adaptive. Static strategies use only statistical system information in making task routing decisions, and their principal advantage is their simplicity in mathematical analysis and implementation. However, they do not adapt to workload fluctuations. Adaptive policies are based on information about the system state and attempt to dynamically optimize the workload routing.

The following task routing strategies are examined:

- *Probabilistic (P):* Arriving tasks are assigned randomly to one of the 128 processor queues with equal probability (static policy).

- *Shortest Queue (SQ):* This policy assigns each ready task to the currently shortest processor queue (adaptive policy).

A considerable communication and decision making overhead is involved in the shortest queue case. This overhead is due to queue length information that must be updated before a routing decision is made. The problem deteriorates with increasing number of processor nodes. We do not explicitly model this overhead in this study. However it is suggested that information from only a subset of the processors could efficiently be used. [4] shows that when the number of processors probed for queue length information is reduced from N to 3, the average performance decrease is within 10% of that achieved when all processors are used.

The order in which tasks in a processor queue are executed depends on the task scheduling policy applied. Task scheduling policies can be either preemptive or non-preemptive. Preemptive policies involve suspending the execution of a task on a processor node and transferring it to another position in the task queue, i.e. at the end of the queue. In non-preemptive policies tasks, once have started execution, are not interrupted until service completion.

We examine three non-preemptive task scheduling policies:

- *First-Come-First-Served, FCFS:* This strategy schedules tasks in the same order as their arrival at the task queue.
- *Smallest Number of Tasks First, SNTF:* This policy uses the number of tasks into which a job is partitioned as an indicator of the job granularity. Coarse-grain jobs are given higher priority. Therefore tasks belonging to a job with the currently smallest number of tasks are assigned the highest priority. Upon priority assignment tasks are inserted at the appropriate position in the task queue. In case two tasks have the same priority they are scheduled on a $FCFS$ basis.
- *Maximum Number of Tasks First, MNTF:* This policy also considers the number of tasks per job as an indicator of the job granularity. However, it assigns higher priority to fine-grain jobs. Hence, the highest priority is given to each task that belongs to a job with the currently maximum number of tasks. Tasks with the same priority are scheduled in a $FCFS$ discipline.

The last two task scheduling strategies are vulnerable in the extreme cases where the number of tasks per job is either too big or too small respectively. Since queues at the processors are rearranged each time a new task is inserted to them, it is possible for some jobs to never be scheduled. As resequencing requires the departure of all previously arrived jobs before a job leaves the system, the resequence buffer can become the system bottleneck.

This problem is eliminated by limiting the number of times p that a specific task can be rejected from the first queue position when a higher priority task has just been inserted. Any further attempts to reject this particular element from the first queue position due to priority rules are deferred. This constraint is applied to every queue element when it is transferred to the first queue position.

2.3. Performance Metrics

In examining the system behavior we use the following performance criteria:

- *Response Time, RT:* The response time of a random job is the time taken from the job arrival to the system to the job service completion.
- *Resequence Delay, RD:* The resequence delay of a random job is the time between the job service completion and the job departure from the system.
- *Time in System, TS:* The time in system of a random job is the time measured from the job arrival to the job departure from the system.

In addition we consider system throughput rate (THR).

The system behavior without taking into account job resequencing is shown by the mean response time, MRT. The impact of the resequencing operation on system performance is indicated by the mean resequence delay, MRD. The overall performance of the system is determined from the system throughput rate, THR, and from the mean time in system, MTS.

3. Results and Discussion

3.1. Workload and input parameters

We obtained the results using simulation. Our model was realized by building over the Sim++ software toolkit as described in Sec. 3.2.

The simulation was performed according to the method of independent replications [10]. We calculated 95% confidence intervals for every mean value we used. All confidence intervals were less than 5% of the mean values.

The number of processor nodes was $N = 128$.

Two task service time distributions were employed, exponential and hyperexponential, both having the same mean $m = 1$. The system was examined for $C = 1$ (exponential distribution) and $C = 2, 4, 6$ (hyperexponential distribution) coefficients of variability. We have not studied cases for $C > 6$ because in real systems job service times tend to have low variations.

Two cases for the number of tasks per job were studied. In the first case the number of tasks per job was exponentially distributed with mean $T = N = 128$, and in the second one the number of tasks per job was uniformly distributed in the range $[1..N]$. In each case the aim was to study a system with a utilization ρ around 0.8. System utilization is given from the following formula:

$$\rho = \frac{\lambda \cdot T}{N \cdot \frac{1}{m}} \quad (1)$$

where λ is the arrival rate, T denotes the mean number of tasks per job, N denotes the number of processors and m denotes the mean task service time.

In our study $m = 1$, $N = 128$, and T values were 128 and 64.5 in the exponential and the uniform number of tasks per job cases respectively. For these cases from the above formula we obtain $\lambda = 0.8$ and $\lambda = 1.59$ respectively.

The maximum number of rejections of the first element in a task queue was taken as $p = 2$.

3.2. Model implementation

A modified version of Sim++ version 1.01 — a Simpack descendant — was used to develop the model. Simpack, [3], is a software toolkit suitable for the development of event scheduling and queuing model simulations. It consists of a set of software library components oriented to a variety of potential simulation applications. The major Simpack components are intended for constraint models, functional models, spatial models, multimodels and other autonomous tools for modeling and visualization. Simpack is implemented in the C programming language.

Sim++ is the object oriented version of Simpack. It is developed according to the object oriented paradigm and it is implemented in C++. Sim++ comprises two API's (Application Programmer Interfaces) for both procedural and object-oriented programming. In both versions the aim is to free the simulation programmer from repetitive tasks and unnecessary detail. The Sim++ user is expected to build his own model within the existing framework and to extend the Sim++ capabilities by writing his own code. Therefore Sim++ includes routines — either in procedural or object oriented form — to handle most of the common operations of a discrete event model, i.e. event list manipulation, event handling, random number generation, statistics gathering, report generation. In particular three event handling mechanisms are available — manual, semi-automatic and automatic — that are based on the same discrete event modeling principles.

Apart from exploiting the Sim++ offered potential, and building upon the existing framework, we also had to proceed to some modifications in the Sim++ internal structure to be able to realize all the details of our model. The major changes involve extending the Sim++ queue storing capabilities — in Sim++ queues can be stored as dynamic data structures, i.e. objects — to handle the specific requirements of our queuing problem. The main point was to limit the times that the first element in a queue can be rejected. Furthermore, statistics calculation had to be seriously enhanced to correspond to all the metrics required to debug and conduct this study. Finally, model execution was facilitated by extending Sim++ to interact with multiple input parameter files and to produce customized, problem specific reports.

3.3. Performance Analysis

Figure 2 shows the performance (MRT) of the three task scheduling policies we considered as a function of coefficient of variation C of task service time. This is done for both probabilistic and shortest queue task routing strategies. These results are obtained considering exponential number of tasks per job. Figure 3 presents MRT versus C for the same cases but with uniform number of tasks per job.

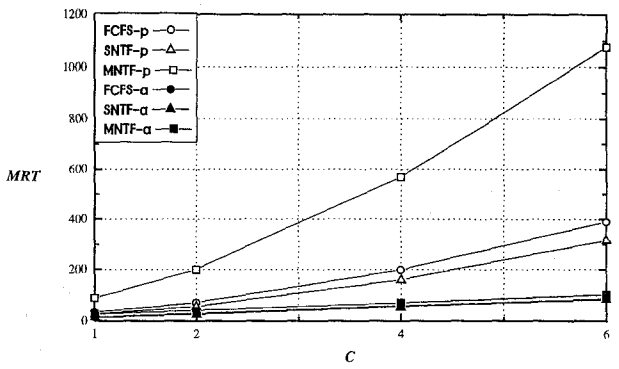

Figure 2. MRT versus C. Exponential number of tasks per job.

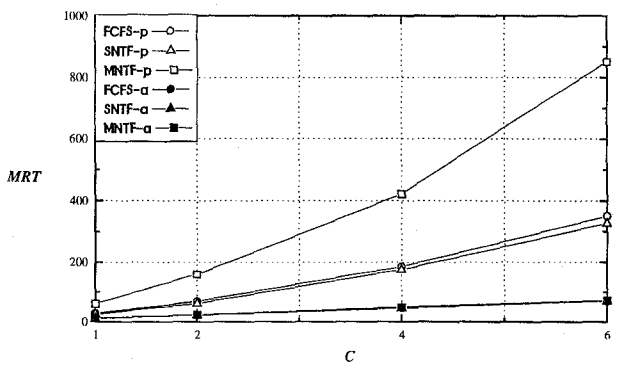

Figure 3. MRT versus C. Uniform number of tasks per job.

Similarly, in Figures 4-9, the other performance parameters are plotted versus C for both exponential and uniform number of tasks per job. The simulation results reveal a substantial performance advantage of the shortest queue strategy over the probabilistic one. Generally, system performance deteriorates with increasing coefficient of variation C of task service demand.

When SQ task routing is considered, all three task scheduling policies affect marginally MRT for both cases of number of tasks per job (Figures 2, 3). Furthermore, SQ task routing results in lower MRT values than the probabilistic one for all cases examined in this study. When task routing is probabilistic, there is a significant effect of the scheduling policy. The worst strategy is clearly the $MNTF$ in this case. In the cases of probabilistic and SQ task routing and $FCFS$ and $SNTF$ task scheduling, [4] presents similar results concerning MRT based on a system that consists of $N = 64$ processors. The difference in performance of the scheduling methods increases with increasing C values. This is because a high variability in task service demand implies that there is proportionally a high number of tasks in the system with small service demand and a comparatively low number of tasks with very large service demand. Hence, when "large" tasks are served by some processors, they occupy them for a long time and introduce inordinate queuing delays to the other tasks that wait in their queues. Therefore, there are more opportunities at high C for the different scheduling policies to present their different capabilities.

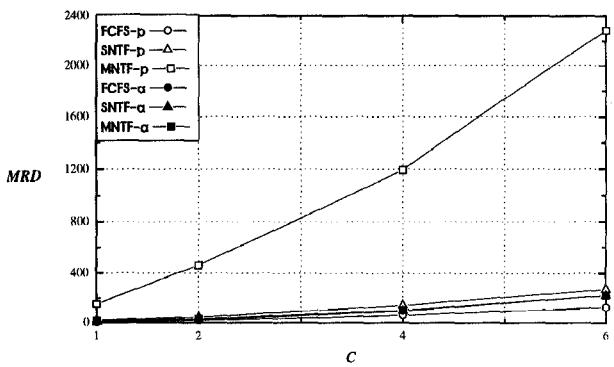

Figure 4. *MRD* versus C. Exponential number of tasks per job.

The performance superiority of the SQ task routing method over the probabilistic one increases with increasing C. This is due to the large variability of task sizes at high C values, which results in unbalanced processor queues and consequently in better exploitation of the advantages of the shortest queue policy.

The least resequence delay (Figures 4, 5) from all cases examined is observed when probabilistic task routing with $FCFS$ task scheduling is used. It appears that the arrival order is mostly preserved in this case, requiring thus less job resequencing after service. When $STNF$ or $MTNF$ task scheduling is employed, SQ task routing yields lower resequencing delay than the probabilistic one. Both number of tasks per job cases produce similar results regarding MRD.

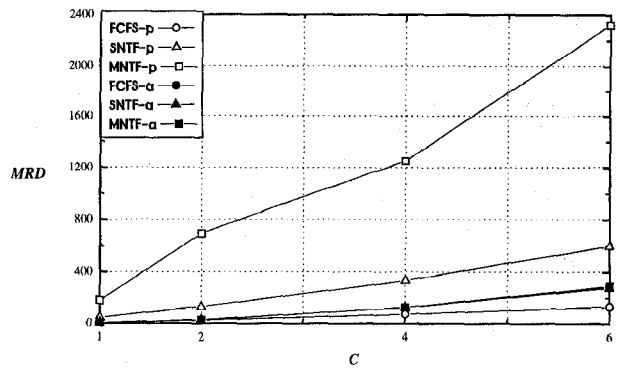

Figure 5. *MRD* versus C. Uniform number of tasks per job.

The resequencing delay increases with increasing C values. This is because at high C values more service times are produced that are much shorter than m and fewer ones that are much longer than m. Therefore, when a task with high service time is processed in a processor, other subsequent jobs with smaller tasks eventually finish execution at other processors and wait in the resequencing buffer. The uniform number of tasks per job case results in higher resequencing delay for all policies.

Although probabilistic task routing with $FCFS$ task scheduling produce the least resequencing delay, SQ task routing results in lower job mean time in system and higher system throughput for all three task scheduling methods and for both cases of number of tasks per job. Figures 6-9 show that the best combination to obtain the best overall performance (MTS and THR) is adaptive task routing and either task scheduling policy. Obviously the worst combination is probabilistic task routing and $MNJF$ scheduling.

For both cases of number of tasks per job system throughput deteriorates with increasing task service time C (Figures 8, 9). This is due to the fact that increased C values result in increased MRT and MRD values and consequently in higher MTS. Probabilistic task routing results in lower system throughput than the shortest queue for all three task scheduling methods. The worst case is when probabilistic routing and $MNJF$ is used.

Considering the overall system efficiency it can be seen from the above that the best results are achieved by using SQ task routing. Furthermore with SQ task routing the choice of a task scheduling strategy is not important. Therefore the $FCFS$ policy is preferred since it is the simplest to implement, produces less overhead and provides more fair scheduling than the other two task scheduling methods.

4. Conclusions

In this work we used simulation to compare the performance of two task routing and three task scheduling policies in conjunction with resequencing in a homogeneous distributed system model. We considered two cases of number of tasks per job based on exponential and uniform distribution respectively.

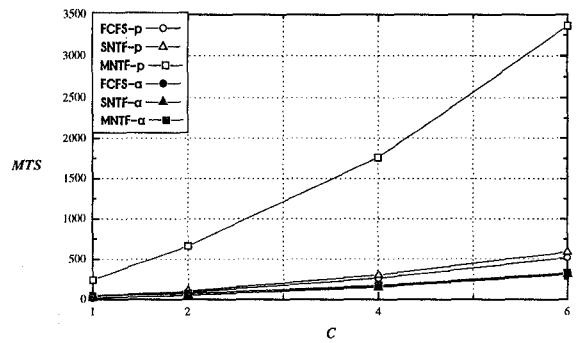

Figure 6. *MTS* **versus** C**. Exponential number of tasks per job.**

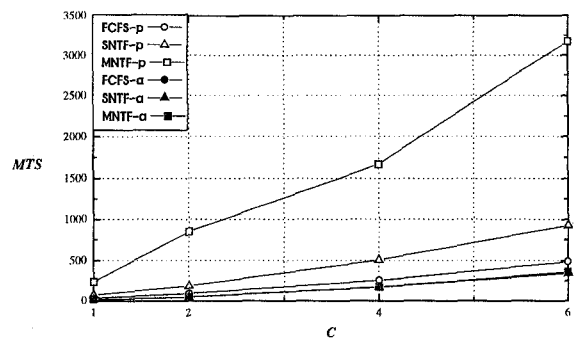

Figure 7. *MTS* **versus** C**. Uniform number of tasks per job.**

Our analysis shows that in all cases examined the least resequence delay is observed when probabilistic task routing and $FCFS$ task scheduling are employed. When $STNF$ or $MNTF$ task scheduling is used, the SQ task routing policy reduces the resequence delay for both distributions of number of tasks per job. In all cases the resequencing delay increases with increasing C values. Since MRT also increases with increasing C, this results in overall performance degradation.

The results show that task scheduling policies affect significantly the overall system performance (MTS, THR) when probabilistic routing is used. In this case the best task scheduling strategy regarding the overall system performance is $FCFS$. The difference in performance among task scheduling policies increases with increasing coefficient of variation C of task service time distribution.

Our results also indicate that when the SQ task routing policy is applied, the task scheduling policy has only a marginal effect on overall system performance (MTS, THR). In this case the $FCFS$ task scheduling strategy is therefore preferable since it is easier to implement and produces less overhead.

Our conclusion is that the use of the shortest queue task routing strategy produces the best overall system performance in all cases examined. Generally, system performance deteriorates with increasing coefficient of variation C of task service demand. The performance superiority of the shortest queue policy over the probabilistic one increases with increasing C values.

Priorities of our future research directions include the extension of the current model to take into account the system state overhead, and also to include cases where task routing and scheduling is applied to groups of tasks, instead of individual tasks only.

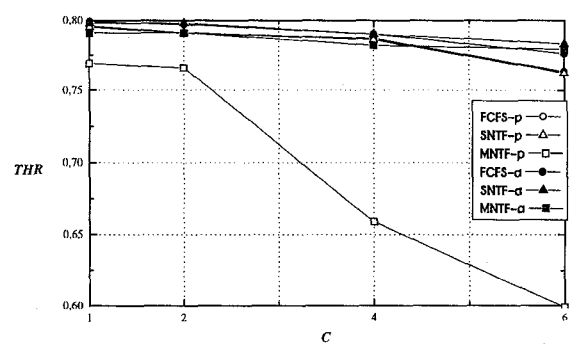

Figure 8. *THR* **versus** C**. Exponential number of tasks per job.**

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] T. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives

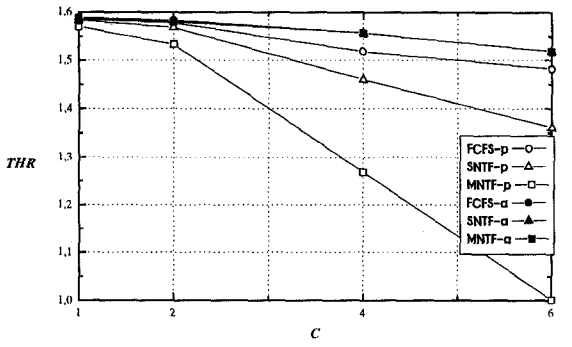

Figure 9. *THR* **versus** *C*. **Uniform number of tasks per job.**

for shared memory multiprocessors. *IEEE Transactions on Computers*, C-38(12):1631–1644, December 1989.

[3] R. M. Cubert and P. A. Fishwick. Sim++: Version 1.0. postscript file, available from WWW: http://www.cis.ufl.edu, December 1995.

[4] S. P. Dandamudi. A comparison of task scheduling strategies for multiprocessor systems. In *Proceedings of IEEE Symposium on Parallel and Distributed Processing*, pages 423–426, Dallas, TX, December 1991.

[5] S. P. 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*, pages 348–353, Ischia, Italy, May 1994.

[6] P. A. Fishwick. *Simulation Model Design & Execution: Building Digital Worlds*. Prentice Hall, New York, January 1995.

[7] I. Iliadis and Y. Lien. Resequencing delay distribution for a queuing system with two heterogeneous servers under threshold-type scheduling. *Data Communication Systems and Their Performance*, pages 359–373, 1989. Elsevier Science Publishers B.V., IFIP.

[8] A. Jean Marie. Load balancing in a system of two queues with resequencing. In P. J. Courtois and G. Latouche, editors, *Performance 87*, pages 75–88, 1988.

[9] H. D. Karatza. Simulation study of multitasking and resequencing in a homogeneous distributed system. In *Proceedings of the Eurosim Congress 95*, pages 541–546, Vienna, Austria, September 1988.

[10] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, Inc, New York, second edition, 1991.

[11] Y. C. Lien. Evaluation of the resequencing delay in a queuing system with two heterogeneous servers. *Computer Networking and Performance Evaluation*, pages 189–197, 1986. Elsevier Science Publishers B.V., IFIP.

[12] R. Nelson, D. Towsley, and A. N. Tantawi. Performance analysis of parallel processing systems. *IEEE Transactions on Software Engineering*, SE-14(4):532–540, April 1988.

[13] L. M. Ni, C.-W. Xu, and T. B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, October 1985.

[14] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, August 1989.

[15] I. Sasase and S. Mori. Resequencing delay for a queuing system with multiple servers under Threshold-type scheduling. In *Proceedings of the Tenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 391–399, December 1991.

[16] C. H. Sauer and K. Chandy. *Computer Systems Performance Modeling*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[17] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.

[18] J. A. Stankovic. Simulations on three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, (8):199–217, 1984.

[19] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, second edition, 1991.

[20] T. Takine and T. Hasegawa. Resequencing delay in preemptive priority m/m/2 queues. In *Performance 90*, pages 109–121, 1990.

[21] D. Towsley, C. G. Rommel, and J. A. Stankovic. Analysis of fork-join program response times on multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):286–303, July 1990.

[22] S. Varma. Optimal allocation of customers in a two server queue with resequencing. *IEEE Transactions on Automatic Control*, 36(11):1288–1293, November 1991.