Epoch Load Sharing in a Network of Workstations

Helen D. Karatza
Department of Informatics
Aristotle University of Thessaloniki
54006 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 examines load sharing in a network of workstations (NOW). It proposes a special load sharing method referred to as epoch load sharing. With this policy, load is evenly distributed among workstations with job migration only at the end of predefined intervals. The time interval between successive load sharing is called an epoch. We compare the performance of epoch load sharing with other traditional load sharing methods. The objective is to reduce the number of times that global system information is needed to make allocation decisions, while at the same time achieving good overall performance. A simulation model is used to address performance issues associated with load sharing. Simulated results indicate that epoch load sharing frequently succeeds in this pursuit.

1. Introduction

Distributed computing based on a network of workstations (NOW) has been a topic of research for many years. As networks of workstations emerge and become viable platforms for a wide range of applications, new policies are needed to allocate workstation resources to competing users.

Generally, in distributed systems no processor should remain idle while others are overloaded. It is preferable that the workload be uniformly distributed over all of the processors. It is important to efficiently utilize computational power, especially when load distribution is necessary.

A number of load balancing and load sharing algorithms appear in the literature. In general, the purpose of load balancing is to divide work evenly among the processors; whereas, the purpose of load sharing algorithms is to ensure that no processor remains idle when there are other heavily loaded processors in the system.

With sender-initiated algorithms, load-distribution activity is initiated when an over-loaded node (sender) tries to send a task to another under-loaded node (receiver). In receiver-initiated algorithms, load-distribution is initiated by an under-loaded node (receiver), when it requests a task from an over-loaded node (sender).

Scheduling policies that use information about the average behavior of the system and ignore the current state, are called static policies. Static policies may be either deterministic or probabilistic. Policies that react to the system state are called adaptive or dynamic policies. Dynamic load balancing is an important system function designed to distribute workload among available processors and improve throughput.

The principle advantage of static policies is simplicity, since they do not require the maintenance and processing of system state information. Adaptive policies tend to be more complex, mainly because they require information on the system's current state when making transfer decisions. However, the added complexity can significantly improve performance benefits over those achievable with static policies.

This paper investigates probabilistic, deterministic and adaptive policies. Comparative results are obtained using simulation techniques.

In the probabilistic case, the scheduling policy is described by state independent branching probabilities. Jobs are dispatched randomly to workstations with equal probability. In the deterministic case, routing decisions are based on system state, so jobs join the shortest of the all workstation queues.

In the adaptive case, variations of the two scheduling policies described above are used. For example, when workstations become idle, jobs can migrate from heavily loaded workstation queues to idle workstations. This is a receiver initiated adaptive load sharing method. It balances the job load and can improve overall system performance.

Another adaptive load sharing method is referred to as epoch load sharing. With this policy, load is evenly distributed among workstations, and job migration occurs

Proceedings of the 34th Annual Simulation Symposium (SS '01)
1080-241X/01 $10.00 © 2001 **IEEE**

IEEE COMPUTER SOCIETY

only at the end of predefined intervals. The time interval between successive load sharing transfers is called an epoch.

Most research into distributed system scheduling policies focuses on improving system throughput where scheduling overhead is assumed to be negligible. However, scheduling overhead can seriously degrade performance. Therefore, the number of times the scheduler is called to make load sharing decisions can be a factor that degrades performance.

Also, jobs transferred to remote workstations incur communication costs. In this model, only queued jobs are transferred. We have considered the migration of non executing jobs. It is assumed that the job being executed is not eligible for transfer because doing so is too complex. An obvious disadvantage of preemptive migration is the need to transfer the memory associated with the migrated process; thus, migration costs for an active process is much greater than the cost of remote execution.

We compare the performance of epoch load sharing with other traditional load sharing methods. The objective is to reduce the number of times that global system information is needed to make allocation decisions, while at same time achieving good overall performance. A simulation model is used to address performance issues associated with load sharing. Simulated results indicate that epoch load sharing often succeeds in this pursuit.

Load sharing and load balancing have already been studied by many authors ([1], [2], [3], [4], [5], [6]). However, these studies do not address the issue of epoch job migration.

The epoch load sharing examined here is different from the epoch scheduling studied in [8]. That paper, focuses on the scheduling of memory constrained jobs in distributed memory parallel computers. Only policies that provide co-scheduling are considered, and the number of processors allocated to a job may be changed during its execution. All nodes are reallocated to jobs at each reallocation point.

We do not consider co-scheduling. Instead of reallocating nodes to jobs, we consider load sharing via job migration at the end of predefined intervals. Epoch job migration is studied in a closed queuing network model of a NOW where the I/O subsystem is incorporated.

Generally, other papers found in the distributed and parallel systems literature study processor scheduling only. They do not explicitly model I/O processing, even though it can significantly influence overall system performance. However, scheduling is not an independent issue. Different components of the system must work together to create a cohesive whole in a way that makes sense. A Rosti et al. ([9]) study of large-scale parallel computer systems suggests that by overlapping the I/O demands of some jobs with the computational demands of other jobs, performance can be improved.

Scheduling algorithms involve overhead which is attributed to two factors: the scheduling algorithm's execution time, and the number of task migrations the scheduler produces.

It is well known that shortest queue scheduling performs better than probabilistic scheduling. However, the shortest queue method invokes the scheduler every time a job demands processing service. The objective of this study is to find an epoch that performs well when compared with the shortest queue method and which involves less overhead. In this work, migration overhead impacts on performance. Therefore, we define scheduling optimality as minimizing the number of times the scheduler is activated since invoking the scheduler requires considerable overhead to collect load information from all workstations.

Performance is examined for different epoch sizes, different communication costs and for various degrees of multiprogramming (various system loads). Simulation indicates that in many cases, epoch load sharing performs close to the shortest queue method while involving much less overhead. To our knowledge, such an analysis of epoch load sharing has not appeared in the research literature.

The structure of this paper is as follows. Section 2.1 specifies system and workload models, section 2.2 describes the scheduling strategies, and section 2.3 presents the metrics employed while assessing the performance of the scheduling policies. 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 provides conclusions and suggestions for further research.

2. Model and methodology

2.1 System and workload models

This paper uses a simulation model to address load sharing issues. A closed queuing network model of a NOW is considered. P = 16 homogeneous workstations are available, each serving its own queue. A high-speed network connects the distributed nodes. This is a representative model for many existing departmental networks of workstations.

A multi-server disk center is included in the system. Since files can be stripped across a variable number of disks, a natural way to capture stripped tasks is by using a fork-join system. Each I/O request forks in sub-requests that are served by parallel disk servers. Although the I/O subsystem consists of an array of disks, it is modeled as a single I/O node with a given mean service time. Since we are interested in a system with balanced program flow, an I/O subsystem with the same service capacity as the processors is considered.

IEEE COMPUTER SOCIETY

The degree of multiprogramming N is constant during each simulation run. N jobs circulate alternately between the processors and the I/O subsystem. The configuration of the model is shown in Figure 1.

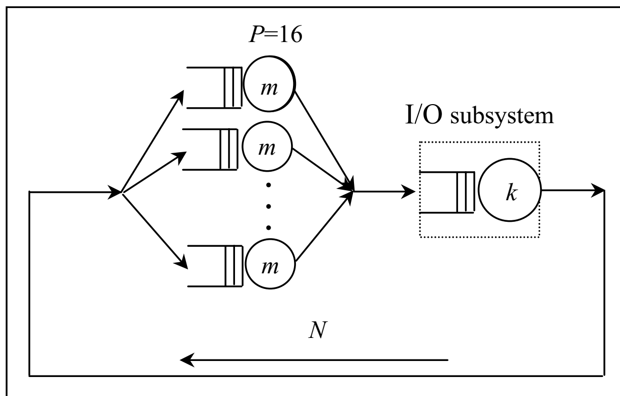

Figure 1. The queuing network model

The jobs examined are highly independent. For example, once a job commences execution, no job ever idly waits for communication with (i.e., synchronizes with) other jobs.

We consider the problem of resource management in a NOW, and focus on load sharing as an efficient strategy to improve throughput. However, the load sharing activity comes at the expense of useful computation, incurs communication overhead, and consumes memory space to maintain load sharing information.

Our study examines various methods of allocating jobs for load sharing using simulation. Traditional methods of scheduling are compared with epoch scheduling. The scheduling policies (described next) are probabilistic, deterministic, and adaptive. In the adaptive case, job migration is performed for load sharing. This is a receiver-initiated load sharing method. Our research seeks to enhance system performance in terms of throughput by dynamic load sharing.

When a job is transferred to a workstation for remote processing, the job incurs a communication cost. In this model, only jobs in the scheduling queue are transferred. The latest-job-arrived selection policy is used to select a job for transfer from the sending workstation to the receiver workstation. We believe that the average transfer cost for a nonexecuting job, although nonegligible, is quite low relative to the average job processing costs. The communication channel is modelled as a single server queuing system, whose service time is an exponentially distributed random variable, with mean *Co,* in order to deal with the effects of communication overhead. The benefits of migration depend on migration costs.

When jobs leave a workstation, they request service on the I/O subsystem. The I/O queuing discipline is FCFS.

A technique used to evaluate the performance of the scheduling disciplines is experimentation using a synthetic workload simulation. In studies like this, the use of synthetic workloads is usually necessary because real workloads cannot be simulated efficiency and real systems with actual workloads are not normally available for experimentation. Also, useful analytic models are difficult to derive because subtleties between various disciplines are difficult to model.

The workload considered here is characterized by three parameters:

- The distribution of processor service time.
- The distribution of I/O service time.
- The degree of multiprogramming.

Processor and I/O service times are independent and identically distributed (IID) exponential random variables with means of m and k respectively.

All notations used in this paper appear in Table 1.

2.2 Job scheduling policies

The following scheduling strategies are employed in our simulations.

- *Probabilistic* (*Pr*)
  With this policy, a job is dispatched randomly to one of the workstations with equal probability. The job dispatcher chooses one of the P workstations based on the outcome of an independent trial in which the ith outcome has probability $p_i = 1 / P$. Then the FCFS policy is applied. Therefore, with this method the scheduler is never activated to make decisions which depend on system state.

- *Probabilistic with Migration* (*PrM*)
  Jobs are assigned to processor queues in the same way as in the *Pr* case. However, when a processor becomes idle and there are jobs waiting at the other processor queues, a job migrates from the most heavily loaded processor to the idle processor. This is a receiver-initiated algorithm, since load-distributing activity is initiated by an idle node (receiver), which tries to get a job from an overloaded node (sender). For stability reasons, sender nodes can only have a queue length greater than one. This policy activates the scheduler only when a processor becomes idle.

- *Shortest Queue (SQ)*
  With this strategy, a job is assigned to the shortest

IEEE COMPUTER SOCIETY

processor queue. Therefore the scheduler is activated every time a job arrives. Each job is entered into its assigned queue in the order of its arrival.

- *SQ with Migration* (*SQM*)
  This is a variation of SQ, where migration takes place in the same way as in PrM. Therefore with this strategy the scheduler is called on job arrival and also when a processor becomes idle after a job departure.

- *Epoch Load Sharing (ELS)*
  With this policy, load is evenly distributed among workstations using job migration which occurs only at the end of predefined intervals called epochs. At the end of an epoch, the scheduler collects information about the status of all workstation queues, evaluates the mean of all queue lengths, and places processor queue lengths into increasing order in a table. Then it moves jobs from the most heavily loaded processors to the lightly loaded ones until either all processors have queue lengths equal to the mean or some of them differ at most by one job.

All five of the above scheduling schemes have merit. Pr is the simplest method since it involves only a negligible amount of overhead when generating random numbers. It is apparent that Pr results in suboptimal performance. However, this method never activates the scheduler as it does not need decisions that depend on system state. The SQ method requires global knowledge of queues on job arrival and so the scheduler is called upon to make decisions every time a job arrives. The migratory versions of these policies invoke the scheduler when a processor becomes idle and they also involve overhead each time a job migrates. The migration overhead is taken into account in this study. Therefore, a major concern is the number of times the system scheduler is called to collect information about processor queues in order to manage the information and to make transfer decisions.

The collection and management of global load information as well as transfer decision making require non-trivial amounts of overhead. However, this overhead is necessary to implement even a moderately effective scheduler. In this study, the parameter that reflects the scheduling policy complexity is the number of times that the scheduler is activated to make decisions which depend on system state.

2.3 Performance metrics

Parameters used in simulation computations (presented later) are shown in Table 1.

Table 1: Notations

N	Degree of multiprogramming
m	Mean processor service time
k	Mean I/O service time
R	System throughput
NSA	Number of times that the system Scheduler is Activated
D_R	Relative (%) increase in R when one of the above described methods is employed instead of the Pr policy
D_{NSA}	Relative (%) decrease in NSA when one of the above described methods is employed instead of the SQ policy
Co	Mean communication delay due to job migration

3. Simulation results and discussion

3.1 Model implementation and input parameters

The queuing network model described above is implemented with discrete event simulation ([7]) using the independent replication method. For every mean value, a 95% confidence interval was evaluated. All confidence intervals were found to be less than 5% of the mean values.

A balanced system with m=1.0 and k = 0.0625 was considered.

The degree of multiprogramming N was 16, 32, 48, 64, and 80. The reason for examining different degrees of multiprogramming is that it is a critical parameter in determining system load.

Epoch size was 1, 2, 4, and 8. We chose epoch length 1 as a starting point for the experiments because the mean processor service time is equal to 1, and also because with this epoch size NSA was smaller than in the SQ case. We expected that larger epoch sizes would result in even smaller NSA.

Mean communication delay Co was 0.1, 0.2, 0.25, 0.5. That is, we chose mean communication delay equal to m/10, m/5, m/4, and m/2. These are reasonable choices for low to high communication delays because we considered the migration of non executing jobs.

3.2 Performance analysis

Only the following results are presented due to space limitations. These results represent the overall relative performance of the different policies very accurately.

IEEE COMPUTER SOCIETY

- In Figures 2, 5, 8, 11: R versus N, for Co = 0.1, 0.2, 0.25, and 0.5 respectively.
- In Figures 3, 6, 9, 12: D_R versus N, for Co = 0.1, 0.2, 0.25, and 0.5, respectively.
- In Figures 4, 7, 10, 13: D_{NSA} versus N, for Co = 0.1, 0.2, 0.25, and 0.5 respectively.

Simulation results demonstrate the following:

In all cases, the overall performance in terms of system throughput is superior with the SQ and SQM methods. Actually, these two methods produce close to the same throughput. This is due to the fact that in the SQ case jobs are always assigned to the shortest queue, so it is probable that the load is evenly distributed among processors. Therefore, there are only rare opportunities for migration. In the SQM case, when migration occurs to an idle processor, the next arriving job will be dispatched to another idle processor because of the shortest queue criterion.

The superiority of the SQ method is intuitive. However, the intent of this study is to determine how much better it is when compared with other methods, and if the extent of its superiority justifies overhead required to maintain knowledge of processor queue lengths.

The worst method is the probabilistic policy Pr. The difference in performance between SQ and Pr decreases with increasing N. For N = 16, D_R is about 67%, while for N=80, D_R is about 19%. The migration of jobs when using PrM significantly improves overall performance as compared with Pr. This is because using the Pr policy there are cases where processors have unbalanced queues. For all N and all Co, the difference in performance between PrM and Pr methods varies between 15% and 36%.

For all epoch sizes, epoch load sharing performs better than Pr. Performance in terms of throughput improves with decreasing epoch size.

For each epoch size the performance deteriorates with increasing Co. The deterioration is larger for smaller epoch sizes than for larger ones. However, in all cases the deterioration is very slight. For epoch size 8, R is almost the same for all Co. Co affects the performance of the PrM method more than the performance of the epoch policy. However, even in the PrM case the difference in performance due to varying Co is not significant. For example, in the PrM case for N = 32, when Co increases from 0.1 to 0.5, then the relative decrease in R is about 5%. This is the largest difference observed as a result of varying Co in all of the cases examined.

For Co = 0.1, N = 64, and N = 80, the PrM and SQ algorithms perform very close to ELS for epoch sizes 1 and 2. However, as Co increases, with these degrees of multiprogramming, the decrease in R is larger in the PrM case than in the ESL case. For Co = 0.5, and N = 64 and 80, PrM performs worst than ELS (for epoch sizes 1 and 2) while ESL still performs close to SQ.

For all Co and for all N, the relative decrease in the number of scheduler activations (D_{NSA}) is very high with all epoch sizes (i.e., in the range 89-99%). D_{NSA} is lower in the PrM case (in the range 66% – 86%). NSA is slightly larger in the SQM case than in the SQ case due to job migration.

For any Co and for any N, D_{NSA} increases with increasing epoch size. For a given epoch size and a given N, D_{NSA} is almost the same for all Co.

As was already mentioned, for N = 64, and 80, and for all Co, ELS performs close to SQ for epoch sizes 1 and 2. D_{NSA} in this case is not much smaller than the D_{NSA} of the 4, and 8 epoch size cases. From the point where N = 48, as N decreases ELS starts to decline with SQ. For $N \leq 48$, PrM performs better than ESL but in many cases its superiority over ESL is not significant if overhead is considered. Actually, there are cases where D_{NSA} is much higher in the ESL case than in PrM. For N=16, the SQ method performs much better than all other methods. Therefore, even though the overhead that this method incurs can degrade its superiority, its performance is expected to remain high in comparison with other methods.

In all cases, system load (mean workstation utilization) varied between 0.50 (Pr, N=16) and 0.99 (SQ, N=80).

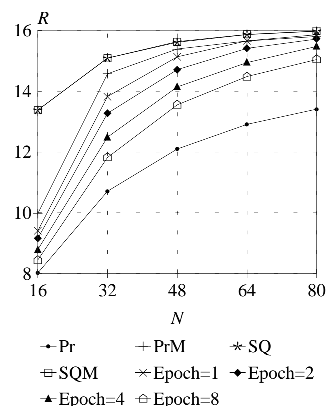

Figure 2. *R* versus *N*, *Co* =0.1

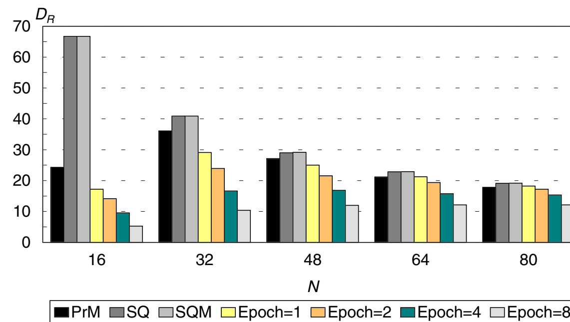

Figure 3. D_R versus *N*, *Co* =0.1

Proceedings of the 34th Annual Simulation Symposium (SS '01)
1080-241X/01 $10.00 © 2001 IEEE
IEEE COMPUTER SOCIETY

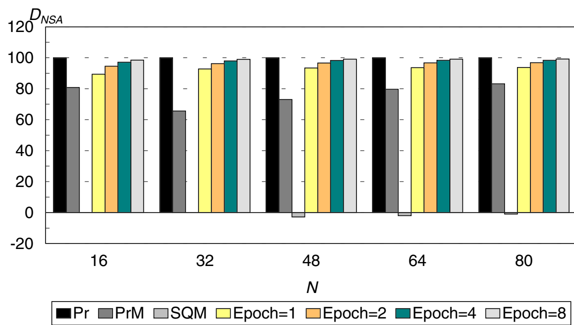

Figure 4. D_{NSA} versus N, Co = 0.1

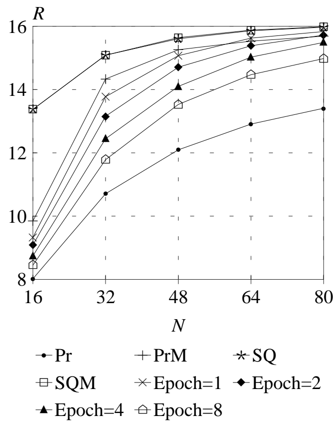

Figure 5. R versus N, Co =0.2

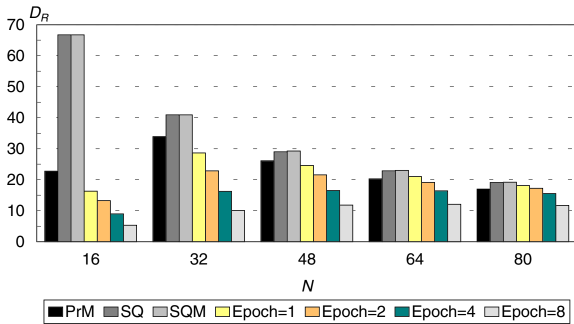

Figure 6. D_R versus N, Co = 0.2

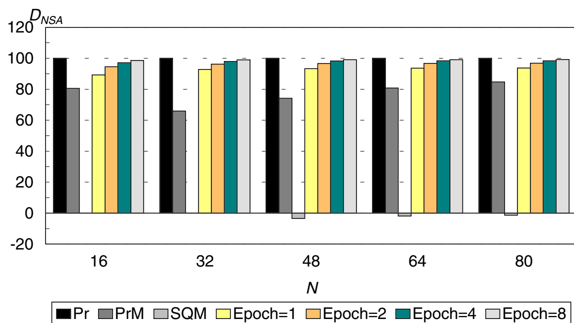

Figure 7. D_{NSA} versus N, Co = 0.2

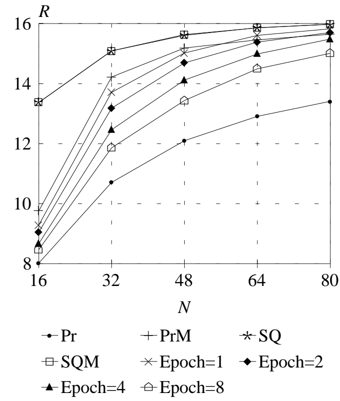

Figure 8. R versus N, Co =0.25

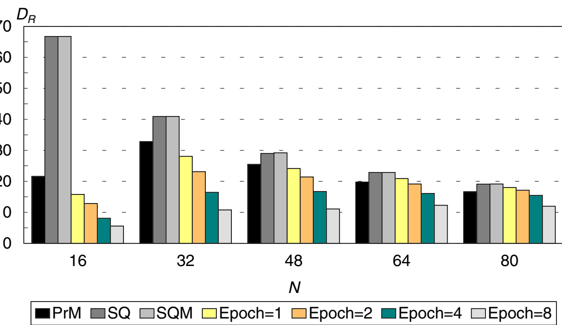

Figure 9. D_R versus N, Co = 0.25

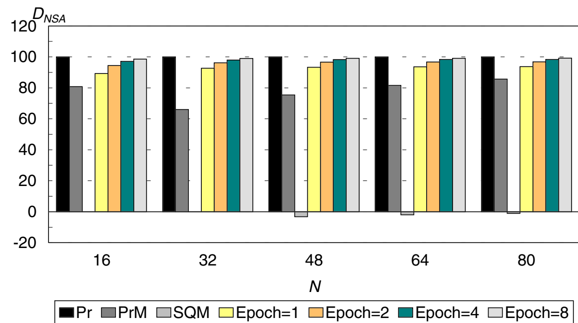

Figure 10. D_{NSA} versus N, Co = 0.25

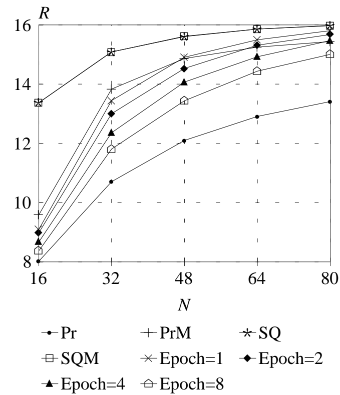

Figure 11. R versus N, Co =0.5

IEEE COMPUTER SOCIETY

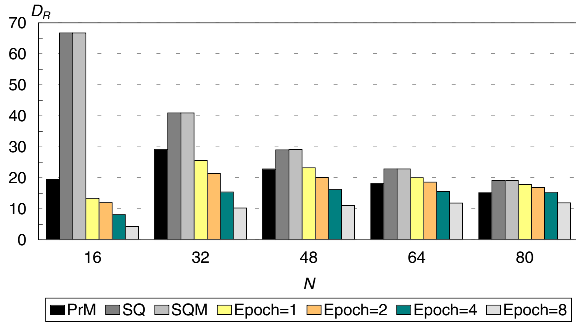

Figure 12. D_R versus N, Co = 0.5

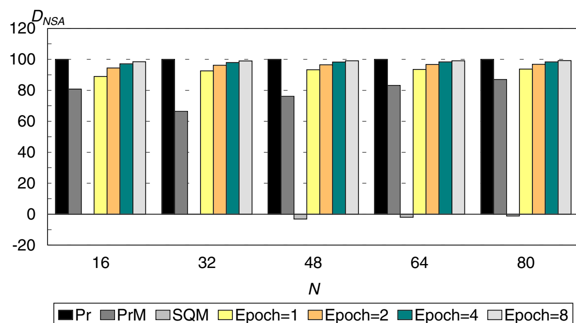

Figure 13. D_{NSA} versus N, Co = 0.5

4. Conclusions and further research

This paper studies load sharing policies in a network of workstations. Simulation is used to generate results needed to compare different configurations.

A new policy called Epoch Load Sharing (ELS) is proposed. Its performance is compared with other scheduling methods for different epoch sizes, different migration overhead, and for various degrees of multiprogramming *N*.

Simulation results reveal the following:

- For all levels of migration overhead, all *N*, and for all epoch sizes, ELS involves much less overhead than the shortest queue (SQ) policy, and involves less overhead than the Probabilistic Migratory (PrM) method, in terms of the collection of global system information.
- For high loads, ELS with small epoch size is preferred since it performs very close to the SQ method.
- For moderate loads, in some cases the PrM method is best while in other cases ESL with a small epoch size is preferred.
- For light loads, the SQ method is recommended.

This paper is a case study. It can be extended to cases where:

- An estimate of job service time is known in advance and can be considered, so that very small jobs will not be migrated.
- Epoch load sharing is applied in a heterogeneous NOW.

References

[1] B.A. Blake, “Assignment of Independent Tasks to Minimize Completion Time”, *Software-Practice and Experience*, Vol.22(9), John Wiley & Sons, Inc., New York, 1992, pp. 723-734.

[2] D.L. Eager, E.D. Lazowska, and J. Zahorjan, “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing”, *Performance Evaluation*, Vol.6, Elsevier Science Publishers B.V., Amsterdam, 1986, pp. 53-68.

[3] M. Harchol-Balter, and A.B. Downey, “Exploiting Process Lifetime Distribution for Dynamic Load Balancing”, In *Proceedings of Sigmetrics ’96*, ACM, Philadelphia, May 23-26, 1996, pp. 13-24.

[4] H.D. Karatza, “Simulation Study of Sender-Initiated Load Sharing with Resequencing”, In *Proceedings of Summer Computer Simulation Conference, SCSC’96*, SCS, V.W. Ingalls, J. Cynamon, and A.V. Saylor (editors), SCS, Portland, Oregon, July 21-25, 1996, pp. 497-501.

[5] H.D. Karatza, “Sender-Initiated versus Receiver-Initiated Adaptive Load Sharing with Resequencing”, In *Proceedings of the 8th European Simulation Symposium & Exhibition*, SCS, A. Bruzzone and E. Kerckhoffs (editors), Genoa, Italy, October 24-26, 1996, pp. 546-550.

[6] H.D. Karatza, “Assignment of programs in a Distributed System with Resequencing”, In *Proceedings of the 31th Annual Simulation Symposium*, IEEE Computer Society Press, Boston, MA, April 5-9, 1998, pp.34-41.

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

[8] C. McCann, and J. Zahorjan, “Scheduling Memory Constraint Jobs on Distributed Memory Parallel Computers”, In *Proceedings of the 1995 ACM Sigmetrics Conference*, ACM, Ottawa, Canada, May 15-19, 1995, pp. 208-219.

[9] E. Rosti, G. Serazzi, E. Smirni, and M. Squillante, “The Impact of I/O on Program Behaviour and Parallel Scheduling”, In *Proceedings of SIGMETRICS 98*, ACM, Madison, WI, June 1998, pp. 56-65.

1080-241X/01 $10.00 © 2001 IEEE

IEEE COMPUTER SOCIETY