A Simulation Model of Task Cluster Scheduling in Distributed Systems

Helen D. Karatza
Department of Informatics
Aristotle University of Thessaloniki
54006 Thessaloniki, Greece
karatza@csd.auth.gr

Abstract

This paper addresses issues of task clustering—the coalition of several fine grain tasks into single coarser grain tasks called task clusters—and task cluster scheduling on distributed processors. The performance of various scheduling schemes is studied and compared for a variety of workloads. Simulation results indicate that the scheduling policy that gives priority to the cluster with the smallest cumulative service demand of all its tasks performs better than the other policies examined.

1. Introduction

In distributed systems, the efficiency of execution of parallel programs critically depends on the policies used to partition the program into modules or tasks and to schedule these tasks onto distributed nodes. In these systems communication cost incurs if two tasks are assigned to different processors. Several partition algorithms have been proposed in the literature. The goal of them is to divide the program into the appropriate size and number of tasks to balance the two conflicting requirements of low communication overhead and a higher degree of parallelism. One solution that has been proposed is to coalesce several fine grain tasks into single coarser grain tasks *called task clusters*. Upon construction, task clusters are scheduled on their assigned processors. Therefore, task clustering is a pre-process step to scheduling.

This is clearly different from the task level models ([3], [4], [9]), in which, after a job arrives to the system, it is immediately split into component tasks, and these tasks will be processed on any processor in any order as long as the precedence constraints are not violated.

Typically, a set of tasks that represents a distributed program corresponds to nodes in a directed graph with node and edge weights. Each vertex in a graph denotes a task and a weight, which represents its processing time. Each edge denotes the precedence relation between the two tasks, and the weight of the edge is the communication cost incurred if the two tasks are assigned to different processors ([5], [6], [7], [10], [11], and [12]).

The clustering problem has been shown to be NP-complete for a general task graph and for several cost functions ([5]). For example, if the cost function is the minimization of parallel time on a completely connected virtual architecture with an unbounded number of processors, then clustering is NP-hard in the strong sense. Since neither analytic nor absolute results are known, research has been done to determine the relative performance of some promising techniques.

Many authors have been studied the clustering problem ([1], [2], [5], [6], [7], [10], [11], [12]) and they have proposed various algorithms based on graph scheduling and heuristics. The primary purpose in most of these works is to find ways to distribute the tasks among the processors in order to achieve some performance goals such as minimizing job execution time, minimizing communication and other overhead and/or maximizing resource utilization.

Multitasking that has been studied in [8] is a type of clustering. In that work a distributed system with two processors is considered where some jobs consist of a set (cluster) of two sequential tasks that must be executed on the same processor, while other jobs consist of two parallel tasks that can be executed on either processor. Resequencing of jobs is required after processor service.

There are two fundamental scheduling strategies used in clustering: scheduling independent tasks in one cluster (non-linear clustering) and scheduling tasks that are in a precedence path of the directed graph in one cluster (linear clustering). Linear clustering fully exploits the parallelism in the graph while non-linear clustering reduces the parallelism by sequentializing independent tasks to avoid high communication.

In this work we consider linear clustering. A simple probabilistic task clustering method is employed to coalescing tasks into task clusters. Emphasis is given to the study of the subsequent cluster scheduling within processor queues. Tasks in a cluster must be executed sequentially on the same processor without preemption.

0-7695-0468-X/99 $10.00 © 1999 IEEE

Generally, scheduling policies for distributed systems and multiprogrammed parallel systems study the influence of the scheduling policy on processor performance only. They do not explicitly model the I/O processing, although it can significantly influence the overall system performance. However, scheduling is not an isolated issue. It is but one service provided by the operating system. The solution to the scheduling problem must be integrated with solutions to other problems, e.g. I/O management. The different parts of the system must work together to create a cohesive whole in a way that makes sense.

In this work we study task clustering in a closed queuing network model of a distributed system where we incorporated I/O equipment. The design choices that are considered include different ways to schedule task clusters for execution. We compare the performance of four cluster scheduling policies for various coefficients of variation of the processor service times and for different degrees of multiprogramming. To our knowledge, such an analysis of cluster scheduling has not appeared in the research literature.

2. Model and methodology

2.1. System and workload models

A closed queuing network model of a distributed system is considered. There are P homogeneous and independent processors each serving its own queue. We have examined the system in two cases: for $P = 4$ and for $P = 16$. This is a reasonable choice for the current existing medium-scale departmental networks of workstations. It is believed that qualitative results for other numbers of processors even for large-scale distributed, are similar for the data demonstrated here. A high-speed network with negligible communication delays interconnects the distributed nodes.

As mentioned in the introduction, the two step approach to scheduling is: 1) Clustering, 2) Scheduling the clusters. The scheduling problem is equivalent to determining a mapping of the clusters to processors and then determining an ordering of the clusters within each processor queue.

Jobs are partitioned into tasks that can be run either sequentially or in parallel. A simple probabilistic clustering method is employed to coalescing tasks into task clusters. Processors are characterized by numbers 1, 2, .., P. Tasks are assigned random numbers that are uniformly distributed in the range of [1 .. P]. We consider that assignment is realized in such a way that tasks of a job with precedence constraints are assigned the same number and perform a cluster that is mapped to the processor labeled with this number. Duplication of tasks in separate clusters is not allowed.

On completing execution, a task waits at the join point for its sibling tasks of all clusters of the same job to complete execution. Therefore synchronization among tasks is required. The price that one pays for the increased parallelism is the synchronization delay that occurs when tasks wait for their siblings to finish execution.

Tasks are executed according to the cluster scheduling method that is currently employed. No migration or preemption is permitted. Once a task of a cluster starts execution, then this task and all other tasks belonging to the same cluster will also run to completion without interruption. Although tasks scheduled on two different processors communicate with each other, we do not model any communication overhead and we consider it as part of task execution time.

The degree of multiprogramming N is constant during the simulation experiment. The configuration of the model is shown in Figure 1.

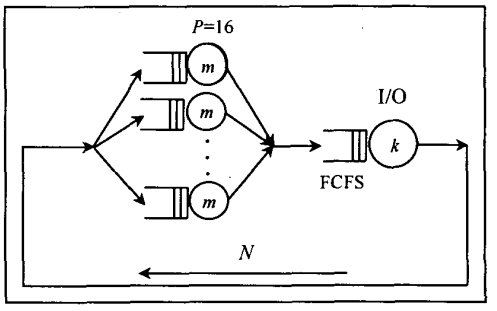

Figure 1. The queuing network model (P=16)

Since we are interested in a system with balanced program flow, we have considered an I/O channel, which has the same service capacity as the processors.

The workload considered here is characterized by three parameters: the distribution of the number of tasks per job, the distribution of task execution time, and the degree of multiprogramming. We assume that there is not correlation between the different parameters. For example, a job with a small number of tasks may have a long execution time.

Jobs consist of a set of $n \geq 1$ tasks. We assume that the number of tasks of jobs is uniformly distributed in the range of [1 .. P]. The number of tasks of a job x is represented as $t(x)$. Each time a job returns from I/O service to distributed processors, it is partitioned into a different number of tasks and it needs a different number of processors for execution. That is its degree of parallelism is not constant during its lifetime in the system.

The number of clusters of a job x is represented as $c(x)$ and is equal to the number of processors $p(x)$ required by job x. Therefore the following relations hold:

$$t(x) <= P \quad \text{and} \quad c(x) = p(x) <= t(x)$$

We also investigate the impact of the variability in task service demand on system performance. A high variability in task service demand implies that there is

proportionately a high number of service demands that are very small compared to the mean service time and a comparatively low number of service demands that are very large. When a task with a long service demand starts execution, it occupies a processor for a long interval of time and depending on the scheduling policy applied it may introduce inordinate queuing delays for the other tasks waiting for service.

The parameter, which represents the variability in task execution time, is the coefficient of variation of execution time (C). We examine the following cases:

- Task execution times are independent and identically distributed (IID) exponential random variables with mean m.
- Task execution times have a Branching Erlang distribution with two stages and are IID. The coefficient of variation is C, where $C > 1$ and the mean is m.

After a job leaves the processors, it requests service from the I/O unit. The I/O queuing discipline is FCFS. The I/O service times are exponentially distributed with mean k and are IID.

All notations used in this paper appear in Table 1.

2.2. Cluster scheduling policies

In this work we examine only non-preemptive cluster scheduling policies. All tasks belonging to the same cluster must finish execution before any other task starts processing. Tasks within a cluster are executed sequentially. We assume that the scheduler has perfect information when making decisions, i.e. it knows:

- The number of clusters of each job, that is the number of processors required by each job.
- The number of tasks of each cluster.
- The service time of tasks.

Next we describe the scheduling strategies employed in this work. As in most studies we assume that their overhead is negligible.

- *First-Come-First-Served (FCFS):* With this strategy, each cluster is scheduled into the assigned queue in the order of its arrival. This policy is the simplest form of cluster scheduling.
- *Job with the Smallest Number of Clusters First* (JSNCF): This policy gives higher priority to clusters that belong to the job with the smallest number of clusters. That is, this strategy uses the number of clusters of a job as an indicator of job granularity. Coarse-grain jobs are given higher priority. Depending on their assigned priority, clusters are inserted in the appropriate position in their assigned queue.
- *Cluster with Smallest Number of Tasks First (CSNTF)*: In this case the scheduling criterion is the number of tasks per cluster. The cluster currently including the smallest number of tasks is assigned the highest priority.
- *Shortest Cluster First* (SCF): This policy assumes that a priori knowledge about a cluster is available in form of cumulative service demand of all its tasks. When such knowledge is available, clusters in the processor queues are ordered in a decreasing order of total service demand. However, it should be noted that a priori information is not often available and only an estimate of task execution time can be obtained. In this study, task execution time estimated is assumed to be uniformly distributed within $\pm E\%$ of the exact value.

When using priorities, in the case of ties the *FCFS* method is used.

2.3. Performance metrics

Consider the following definitions:

- *Response time* of a random job is the interval of time from the dispatching of this job clusters to processor queues to service completion of the last task of this job.
- *Cycle time* of a random job is the time that elapses between two successive processor service requests of this job. In our model cycle time is the sum of response time plus queuing and service time at the I/O unit.

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

In our model the system throughput (system performance) and the mean cycle time (program performance) determine the external performance. Internal efficiency is primarily determined by the mean processor utilization.

Table 1. Notations

RT	Mean response time
K	Mean cycle time
R	System throughput
U_P	Mean processor utilization
N	Degree of multiprogramming
C	Coefficient of variation
m	Mean task execution time
k	Mean I/O service time
E	Estimation error

3. Simulation results and discussion

3.1. Model implementation and input parameters

The queuing network model was simulated with discrete event simulation models 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.

Both systems considered are balanced:

m=1.0, k = 0.625 P = 4 case

m=1.0, k = 0.531 P = 16 case

The reason we have chosen k = 0.625 (k = 0.531) for balanced program flow is that at the processors there are on average 2.5 (8.5) tasks per job in the cases of P = 4 and P = 16 respectively. So, when all processors are busy, an average of 1.6 (1.882) jobs is served each unit of time. This implies that I/O mean service time must be equal to 1/1.6 = 0.625 (1/1.882 = 0.531) if the I/O unit is to have the same service capacity.

The system was examined in cases of C = 1, 2, and 4. N was taken as 2, 4, 6, 8, and 10. In the SCF case we have also examined estimation errors ±10% and ±30%.

3.2. Performance analysis

Due to space limitations, only the following results are presented, but they are representative of the overall model performance:

- In Tables 2-3 performance parameters for the C=1 case.
- In Figures 2-3 mean cycle time (K) is plotted versus N for C=1.
- In Figures 4-9 system throughput (R) for all cases examined.
- In Figure 10 system throughput in the C=1, P=16, SCF case for E = 0, 10, 30.

Simulation results show the following:

The relative performance of the four scheduling policies is similar in the P=4 and P=16 systems. From the results we observe that, for the same degree of multiprogramming, system performance is better in the P=4 case than in the P=16 (lower utilization and throughput in the P=16 case). This is due to the fact that in the P=4 case for each processor correspond on average ($N \times 2.5$) / $P = N \times 0.625$ tasks, while in the P=16 case ($N \times 8.5$) / $P = N \times 0.531$ tasks.

For all N the performance in terms of mean cycle time and system throughput, is superior with the SCF method. This is due to the fact that with the SCF policy, clusters with small cumulative service demand are never blocked behind a cluster that has a large service time and is waiting in the queue. Blocking behind a 'large' cluster introduces inordinate queuing delays and also synchronization overhead to the sibling clusters. During that time it is most probable for the I/O unit to remain idle and then to be deluged with many jobs that are forced to delay in its queue. SCF alleviates this problem, yielding lower RT than the other policies, which results in lower mean cycle time and higher system throughput.

FCFS performs almost the same as JSNCF. CSNTF performs very slightly better than these two methods. However, there is an overhead involved with JSNCF and CSNTF due to reordering of tasks in the queues, which is not modelled in this work.

In all cases the superiority of SCF over FCFS is higher at high degrees of multiprogramming. This is due to the fact that at high N there are more tasks in the queues and therefore there are more opportunities to exploit the advantages of SCF. For $N \leq 4$ and for all C all strategies perform very close.

The simulation results reveal that the decrease in response time due to the superiority of SCF is higher than the decrease in K and the increase in R. For example, in the P=4 system case, for C=1 and N=10 the decrease in RT is 12% while the decrease in K is 6% and the increase in R is 6.4%. This is due to the fact that the overall system performance depends not only on the processors but on the I/O unit as well.

We conducted additional simulation experiments to assess the impact of service time estimation error on the performance of the SCF method. Figure 10 shows the effect of service time estimation error on system throughput for the C=1, P=16 case. The estimation error in this figure is set at ±0%, ±10% and ±30%. The graph shows that the estimation error in task service time affects marginally system performance. Therefore, no profit can be gained from the a priori knowledge of exact task service times.

For all N, the superiority of SCF over the other three methods examined is generally increasing with increasing C. This is due to the fact that as C increases the variability in task execution time increases too and this results in better exploitation of the SCF strategy. The superiority of SCF over FCFS was more significant for P=4 than for P=16. For example, regarding R, in the P=4, N=10, C=4 case the superiority is about 10%, while in the P=16, N=10, C=4 case it is about 6%.

4. Conclusions and further research

In this work we studied task clustering in distributed systems. We used simulation as the means of obtaining results. Four scheduling policies were considered. First-Come First-Served (FCFS), Job with the Smallest Number of Clusters First (JSNCF), Cluster with the Smallest Number of Tasks First (CSNTF), and Smallest Cluster First (SCF). Their performance was studied and compared for various degrees of multiprogramming N and coefficients of variation C of task execution times. The simulation results reveal the following:

- In all cases examined SCF performed better than the other policies.
- FCFS performed very close to JSNCF and CSNTF.

- The superiority of SCF over the other methods is increasing with increasing N and increasing C.
- For low N all methods perform close.
- The SCF policy can tolerate estimation error in task service time.

FCFS and SCF policies have their merits. System performance is better in the case of SCF but this method assumes a priori knowledge of an approximate task execution time. When the scheduler does not have this information, then the FCFS method should be preferred from the rest of the methods examined, as it performs very close to them, it is easier to implement, and it produces less overheard.

This work is a case study. It should be extended so that task communication overhead to be considered and also pre-emptive service disciplines to be studied.

Table 2. C=1, SCF case, p = 4

N	U_p	RT	K	R
2	0.41	2.28	3.03	0.66
4	0.59	3.08	4.17	0.96
6	0.70	3.78	5.22	1.15
8	0.77	4.53	6.33	1.26
10	0.81	5.21	7.45	1.34

Table 3. C=1, SCF case, p = 16

N	U_p	RT	K	R
2	0.26	3.49	4.10	0.49
4	0.41	4.36	5.13	0.78
6	0.51	5.15	6.10	0.98
8	0.58	5.98	7.10	1.13
10	0.64	6.76	8.07	1.24

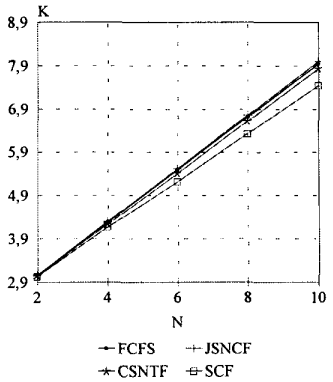

Figure 2. K versus N, p=4, C=1

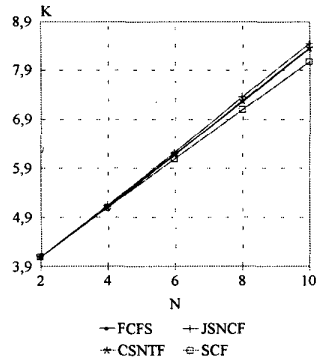

Figure 3. K versus N, p=16, C=1

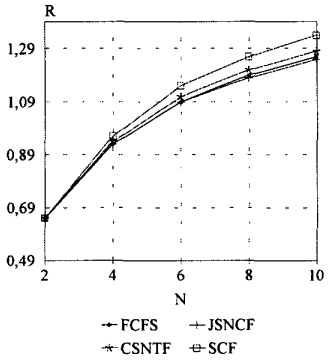

Figure 4. R versus N, p=4, C=1

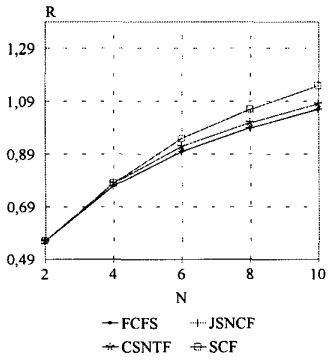

Figure 5. R versus N, p=4, C=2

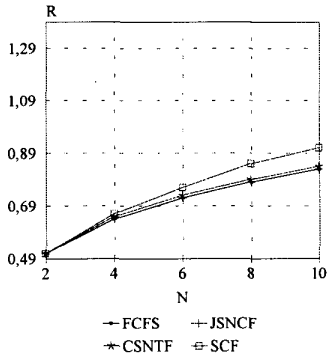

Figure 6. R versus N, p=4, C=4

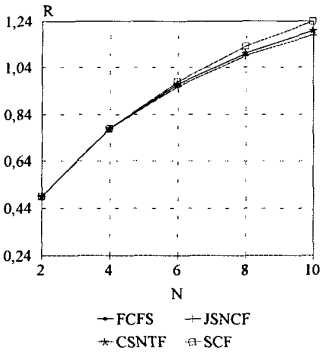

Figure 7. *R* versus *N*, *p*=16, *C*=1

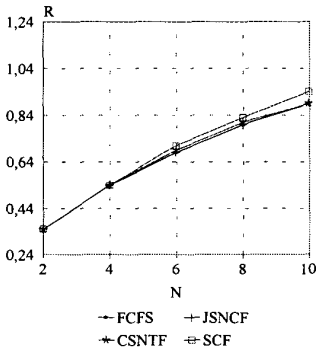

Figure 8. *R* versus *N*, *p*=16, *C*=2

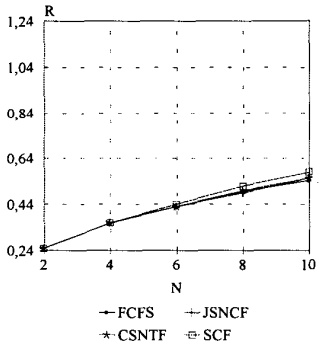

Figure 9. *R* versus *N*, *p*=16, *C*=4

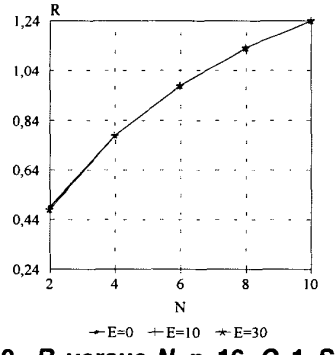

Figure 10. *R* versus *N*, *p*=16, *C*=1, SCF case

5. References

[1] J. Aguilar, and E. Gelenbe, "Task Assignment and Transaction Clustering Heuristics for Distributed Systems", *INFORMATION SCIENCES*, Elsevier Science Inc., March 1997, pp. 199-219.

[2] T. Bultan, and C. Aykanat, "A New Mapping Heuristic Based on Mean Field Annealing", *Journal of Parallel and Distributed Computing*, Academic Press, Vol. 16, 1992, pp. 292-305.

[3] S. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems", *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Dec. 1991, pp. 423-426.

[4] S. Dandamundi, "Performance implications of task routing and task scheduling strategies for multiprocessor systems", *Proceedings of the IEEE-Euromicro Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 1994, pp. 348-353.

[5] A. Gerasoulis, and T. Yang, "A comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors", *Journal of Parallel and Distributed Computing*, Academic press, Vol. 16, 1992, pp. 276-291.

[6] A. Gerasoulis, and T. Yang, 1993, "On the granularity and clustering of directed acyclic task graphs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No.6, June 1993, pp. 686-701.

[7] A. Gerasoulis, J. Jiao and T. Yang, "Experience with Scheduling Irregular Scientific Computation", *Proceedings of the First IPPS Workshop on Solving Irregular Problems on Distributed Memory Machines*, IEEE Computer Society Press, Santa Barbara, CA, April 1995, pp. 1-8.

[8] H.D. Karatza, "Simulation Study of Multitasking and Resequencing in a Homogeneous Distributed System", *Proceedings of the Eurosim Congress '95*, Elsevier Publishers P.V., Vienna, Sept. 11-15, 1995, pp. 541-546.

[9] H.D. Karatza, "Simulation Study of Task Scheduling and Resequencing in a Multiprocessing System", *Simulation Journal*, Special Issue: Modelling and Simulation of Computer Systems and Networks: Part Two, SCS, April 1997, pp. 241-247.

[10] L.C. McCreary, A.A. Khan, J. J. Thomson, and M. E. McArdle, "A Comparison of Heuristics for Scheduling DAGS on Multiprocessors", *Proceedings of Eighth International Parallel Processing Symposium*, IEEE Computer Society Press, Cancun, Mexico, April 1994, pp. 446-451.

[11] M.A. Palis, J.-C. Liou, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No.1, Jan. 1996, pp. 46-54.

[12] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No.9, 1994, pp. 951-967.