Time Space Sharing Scheduling: A Simulation Analysis

Atsushi Hori, Yutaka Ishikawa, Jörg Nolte,
Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo

Tsukuba Research Center
Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, JAPAN
TEL:+81-298-53-1661, FAX:+81-298-53-1652
E-mail:{hori,ishikawa,jon,konaka,m-maeda,tomokiyo}@trc.rwcp.or.jp

Abstract. We explain a new job scheduling class, called "Time Space Sharing Scheduling" (TSSS) for partitionable parallel machines. TSSS is a combination of time-sharing and space-sharing job scheduling techniques. Our proposed "Distributed Queue Tree" (DQT) is an instance of TSSS. We evaluate and analyze DQT behavior in more detail with a number of simulations. The result shows that DQT performs very well in low-load to high-load situations, almost independent of system size and task size distribution. We also compare our DQT and ScanUp batch scheduling, and we find that our DQT performs as well as ScanUp scheduling in processor utilization, but that both DQT and ScanUp have drawbacks in terms of scheduling fairness. Finally, we find that TSSS can inherently achieve higher processor utilization.

1 Introduction

Work on job scheduling on parallel machines has mostly been on batch scheduling with space-sharing where processors are partitioned and jobs are allocated on these partitions [1, 5, 9]. Some commercially available parallel machines have time-sharing facilities [8]. Few of them, however, tackle both time-sharing and space-sharing [2, 3]. Table 1 shows a categorization of job scheduling systems targeting parallel machines. Time-shared job scheduling on parallel machines can take on a different aspect from that on sequential machines, if the targeting parallel machine supports variable partitions. This is because space-sharing with variable partitions[1] can bring an extra dimension to time-shared job scheduling.

In this paper, we explain a new job scheduling class called **Time Space Sharing Scheduling (TSSS)** to create an interactive programming environ-

[1] "Variable partition" means that partitioning can be controlled using software. "Dynamic partition" enables the partition in which a task or a job is running to be dynamically changed on hardware capable of variable partitioning. This definition was proposed by Larry Rudolph at IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing

Table 1. Categorization of job scheduling for a parallel machine

	Batch	Time Sharing
Fixed partition	Conventional scheme	**CM-5 (CMOST)** [8]
Variable partition	Many works [1, 5, 9] and so on.	**Distibuted Hierarchical Control** [2] **Distributed Queue Tree** [3]

ment for parallel machines. TSSS is a combination of a time-sharing and space-sharing job scheduling techniques for parallel machines with variable partitions. We have already proposed the **Distributed Queue Tree (DQT)** [3], and DQT is an instance of TSSS. In the primary report on DQT [3], we described some of its characteristics and proposed several task allocation policies. We did not, however, reveal enough DQT characteristics. In this report, DQT is evaluated and analyzed with a number of simulations. Here, we focus on: i) the scalability of scheduling performance, ii) the relation to task size (the number of processors required by a task) distribution, and iii) fairness of scheduling.

In the next section, we summarize Scan batch scheduling [5], and then explain TSSS. In Section 3, DQT, as an instance of TSSS, is introduced briefly. Then the simulation results are shown in Section 4. The behavior of DQT is analyzed in Sections 4.2 and 4.4, and compared with some batch scheduling techniques proposed so far in Section 4.3.

2 Job Scheduling for Parallel Machines

2.1 Scan Scheduling

Krueger et al. proposed a space-sharing scheduling technique called Scan [5]. The queuing system consists of multiple queues, and each queue is responsible for a partition size. Scan scheduling is centralized and relatively simple. One of the queues is selected and the tasks in the queue are scheduled. If the queue becomes empty, then the next queue becomes the current queue. If the current queue pointer moves toward a queue of larger size, then the scheduling is called ScanUp; the opposite is called ScanDown. It has been reported that ScanUp always exhibits better performance than ScanDown [5]. Scan scheduling is considered to be the best job scheduling scheme among those proposed so far.

2.2 Time Space Sharing Scheduling

Time Space Sharing Scheduling (TSSS) is a new class of job scheduling techniques which provides an interactive multi-process programming environment. It is a combination of time-sharing and space-sharing job scheduling techniques and is covered in the lower right of Table 1. Figure 1 shows a schematic view

of TSSS. TSSS can inherently achieve higher processor utilization, because late-coming tasks can remove fragmentation of the processor space. One major drawback of TSSS is the heavier scheduling overhead. If TSSS performs better than batch scheduling, then it may be worth implementing TSSS from the viewpoint of resource utilization. To clarify this, we will compare our DQT and ScanUp scheduling with simulation (Sections 4.3 and 4.4).

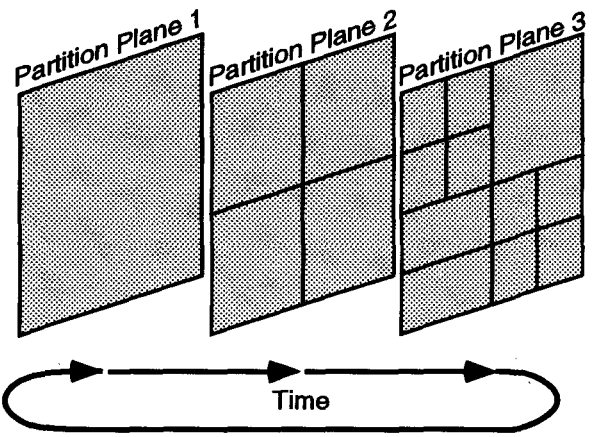


Fig. 1. Example of TSSS

In this paper, a process is defined as an execution entity of a task. With TSSS, a process exclusively occupies a partition in a certain time slot. Each processor could be multiplexed at thread level, not at process level. However, this could result in a larger working set and processor thrashing [2]. A parallel machine is multiplexed in terms of time and processor space with TSSS. A TSSS scheduler should schedule a process to a time slot and map onto a partition. A partition at a certain time slot is a virtualized parallel machine and a processor address space from the user's viewpoint.

We assume that the target parallel machine is homogeneous, and that the user cannot specify the partition to which a process allocated. Only the TSSS scheduler can decide which partition is allocated. The same situation can be seen in the conventional TSS, where the user cannot specify the time slot. The operating system views a partition as a computational resource. The TSSS scheduler is also a virtual parallel machine server of various sizes. Partitions should be allocated by a TSSS scheduler according to the status of the entire system, normally, to balance the system load.

To develop a scheduling technique with TSSS, the following items should be taken into account.

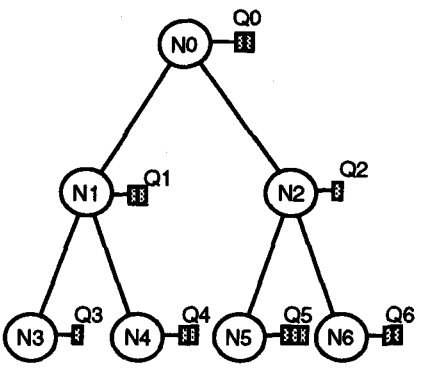
- The scheduling process should be distributed and must avoid any bottlenecks. In time-sharing, processes can often change state from running to waiting and vice versa, especially when the processes are waiting for terminal input. If a time-sharing scheduling scheme is implemented with a centralized queue system, as is the case in conventional sequential machines, this can cause a severe bottleneck.
- A TSSS scheduling scheme should be hardware independent, especially for network topology. In most cases, job scheduling is a part of the operating system and operating systems are hard to maintain. If a TSSS scheduling scheme imposes a network topology, then the portability of the operating system would be lost.
- The processor utilization ratio can be a measure of TSSS performance. The strategies of partitioning and partition allocation for a process are very important. A similar problem can be found in memory allocation [6].
- Fairness in scheduling can also be the other measure of TSSS performance. Fairness, however, may be traded off against processor utilization in some situations.
- It is desired that TSSS performance does not saturate with an increase in workload, and exhibits the same characteristics in the system size (number of processors in a system) and pattern of given workload.

3 Distributed Queue Tree

We proposed a **Distributed Queue Tree (DQT)** [3], which we briefly explain in this section. DQT is a distributed tree structure for process scheduling management. Each DQT node has a process run queue. Every process in the queue requires that the number of processors does not exceed the partition size of the node. The DQT tree structure should reflect the nesting of partitioning. Each DQT node should be distributed to the processor in the partition corresponding to that node. When a process is suspended, it should be dequeued from the process run queue. In DQT, this queue operation is needed only in a processor that plays the role of a DQT node.

Figure 2 (a) shows an example of a DQT structure. Each DQT node has a process run queue that is represented by small rectangles on the right side of the node, with the number of rectangles being equal to the length of the queue. The root node, N0, is responsible for the entire processor space (full partition). Each of the nodes N1 and N2 is responsible for a halved partition. Each of the nodes N3, N4, N5, and N6 is responsible for a quartered partition.

Figure 2 (b) shows a DQT scheduling example, corresponding to the DQT in Fig. 2 (a). In this figure, the jth process in the queue Q_i of the ith node is denoted as "$Q_i(j)$". The entire processor space is assigned to $Q_0(0)$ at time slot 0 and $Q_0(1)$ at time slot 1. In time slot 2, halved partitions are assigned and two processes are simultaneously running in adjacent partitions. In time slot 3, the right-hand side halved partition is halved again, while the left-hand side halved



(a) DQT Structure

Time Slot #	PE0	PE1	PE2	PE3
0	$Q_0(0)$			
1	$Q_0(1)$			
2	$Q_1(0)$		$Q_2(0)$	
3	$Q_1(1)$		$Q_5(0)$	$Q_6(0)$
4	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(1)$
5	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(0)$
6	$Q_0(0)$			
7	$Q_0(1)$			
8	$Q_1(0)$		$Q_2(0)$	
9	$Q_1(1)$		$Q_5(0)$	$Q_6(1)$
10	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(0)$
11	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(1)$
12	$Q_0(0)$			
:	:			

(b) DQT Scheduling

Fig. 2. Example of DQT

partition is left as is since there are two processes in queue Q_1. Every process is scheduled at least once in 6 time slots in this case.

DQT scheduling consists of two major parts. One is a distributed scheduling algorithm and the other is task allocation to balance the DQT load.

3.1 DQT Scheduling

The DQT scheduling process is distributed over the DQT nodes, and the DQT nodes are distributed to the processors in a corresponding partition. Each node communicates and synchronizes with its supernode and subnodes only. Thus the scheduling process is distributed and parallelized.

A DQT node is activated when a process in the queue of a node is scheduled. At a certain time, the line connecting activated DQT nodes in a tree diagram is called a **front**. Figure 3 (a) shows an example of a front movement in a DQT. In this figure, each small rectangle in a DQT node represents a process in a process run queue in that node. If the DQT load is well-balanced, the front is a horizontal line moving repeatedly downward. Lines $t0$, $t1$ and $t2$ in Fig. 3 (a) are examples. The front moves faster on the DQT branch with the lighter load (denoted by $t3$). If a part of the front hits the bottom of the tree (right half of $t3$), then it moves back to the node where the load is unbalanced ($t4$). Consequently smaller tasks may be scheduled more often than larger tasks. This is to keep processors as busy as possible. Predictably, this strategy can cause unfairness. To clarify the effect of this scheduling strategy, we propose an alternative scheduling strategy, called "Fair-DQT." Fig. 3 (b) is an example of Fair-DQT corresponding Fig. 3 (a). Every process is scheduled exactly once in a round. The front never goes

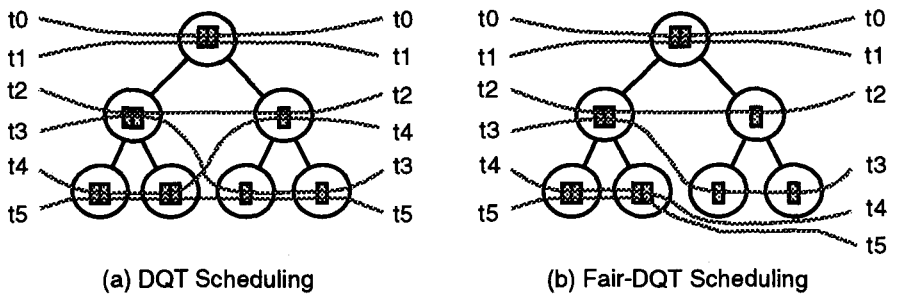


Fig. 3. Example of front movement

up until every part of front is synchronized at the bottom of the DQT. This strategy can result in fairer scheduling but lower processor utilization. We will show some simulation results to compare these two DQT scheduling strategies (Sections 4.3 and 4.4).

3.2 Task Allocation

The policy used to decide which partition is to be allocated when a process is created is very important to balance the DQT load. A well-balanced DQT exhibits not only good processor utilization, but also a shorter response time and fair scheduling [3]. We proposed various task allocation policies [3].

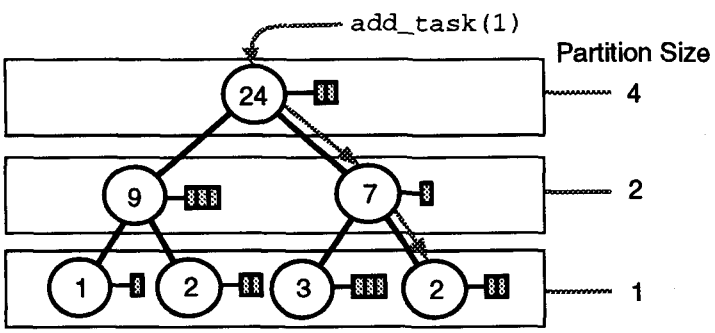


Fig. 4. Example of task allocation

Figure 4 shows an example of one of the proposed policies. The number on each DQT node represents the load on the branch. In this case, it is the number of virtual processors needed to schedule all processes in the sub-DQT at once.

For example, the number on the left node in the second level is 9, because the node itself requires 6 virtual processors (three processes times partition size two) and 3 virtual processors in the subnodes (one for the left subnode and two for the right subnode). The **add_task** message is sent to the root when a partition for a new process is required. This message is forwarded to the subnode whose load is lighter, until it reaches the node that has the required partition size (in this case, the required size is one). This task allocation policy performs very well in various situations [3], and is very simple. In this paper, every simulation of DQT uses this task allocation policy.

4 Simulation

4.1 Simulation Conditions and Measurements

Each task size is rounded up to the nearest partition size, in this case a power of 2. The processor utilization numbers shown in these simulation results were optimistic, since we ignore internal fragmentation and processors which are idle because of communication delays, synchronization and/or waiting for an I/O operation. The task size distributions we simulated are uniform, proportional, and inversely proportional to the rounded size of the task. Tasks that require the full configuration are omitted because they behave in the same way as a single queue system. The distribution of the ideal execution time (task length) is exponential with an average of 1,000 time units. Every task is independent, and can be scheduled and preempted at any time. The simulation time is 10^6 time units, and the time quantum is one time unit. The scheduling overhead is ignored.

The task arrival distribution is geometric, and the mean task arrival time ($\overline{T_{interval}}$) is calculated as

$$\overline{T_{interval}} = \frac{\overline{task\_size} \times \overline{task\_length}}{P \times W_{target}}$$

where $\overline{task\_size}$ is the mean value of the task size (calculated from system size and task size distribution), $\overline{task\_length}$ is the mean value of the task service time, P is the number of processors in the system (system size), and W_{target} is the ratio of workload to system size.

To evaluate scheduling behavior, the system size (number of processors) is varied at 128, 256, 512, 1024, 2048 and 4096. The targeting workload is also varied at 0.2, 0.4, 0.6, 0.8, 0.9, 0.95, 0.97 and 0.99. Since task size, task arrival and task service time are randomized independently, the actual workload may differ from the target workload. So, we define the actual workload (W_{actual}) as

$$W_{actual} = \frac{\sum_{l=0}^{L-1} (task\_size_l \times task\_length_l)}{P \times T}$$

where $task\_size_l$ is the task size of the lth task, $task\_length_l$ is the task length of the lth task, L is the number of tasks arrived, and, T is the simulation time.

The actual workload could exceed 1.0, even though the target workload is less than 1.0. We discard the simulation results in such a situation.

Processor utilization (U_t) at time t is defined as $U_t = P_t^*/P$, and the average processor utilization ($\overline{U}$) from time t_1 to time t_2 is defined as

$$\overline{U} = \frac{\sum_{t=t_1}^{t_2} P_t^*)}{P \times (t_2 - t_1 + 1)}$$

where P_t^* is the number of potentially busy processors (or the sum of the numbers of processors in activated partitions) at the tth time quantum, and P is the number of processors in the entire system.

Here, the **Real Execution Time Ratio (RETR)**[2] of the lth task (R_l^{RET}) is defined as

$$R_l^{RET} = \frac{t_l^{task\_end} - t_l^{task\_entry}}{task\_length_l}$$

where $t_l^{task\_entry}$ is the time of task entry, and $t_l^{task\_end}$ is the time the task ends. This RETR represents how much slower the execution of a task is than the ideal situation.

4.2 DQT Simulation Results

Figure 5 shows the DQT simulation result. The three graphs in the upper row show the processor utilization curves when varying the workload on each system size with the same scale. In those graphs, actual workload values are given on the horizontal axis. To show the difference more clearly, only part of high load situations are drawn. The graphs in the lower row show mean RETR curves in the same scale. In those graphs, in both rows from left to right, the task size distributions are proportional, uniform, and inversely proportional to task size respectively.

Overall, those simulation results show good linearity to the given workload. In all graphs, DQT behaves almost independently of system size. Generally, the saturation of processor utilization means that the number of remaining tasks in the system increases rapidly. These simulation results also show the stability of DQT in high-load situations.

DQT exhibits a somewhat poor performance with uniform task size distribution. With a proportional distribution, the processor utilization can remain high because of larger tasks, while with an inversely proportional distribution, a number of small tasks can be enough to balance the DQT load in high-load situations.

[2] **Real Execution Time** is defined as the total duration that a task is in a process run queue. In this sense, RETR is different from elapsed time.

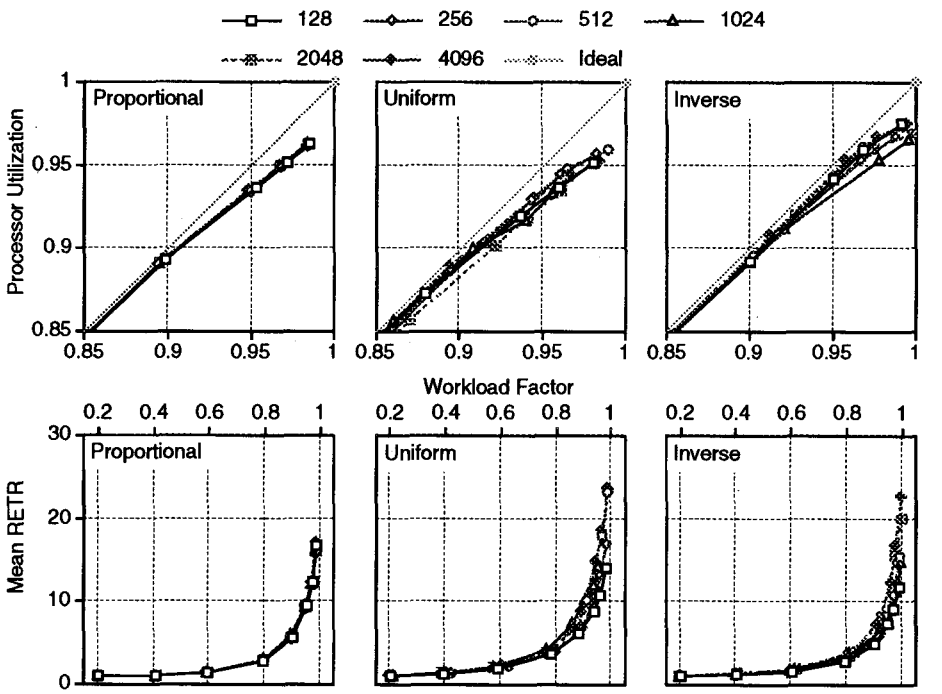


Fig. 5. DQT Simulation Results

4.3 DQT vs. Batch

Figure 6 shows graphs of the simulation results of DQT, Fair-DQT, First-Come-First-Serve scheduling with Binary Buddy allocation strategy (FCFS-BB), and ScanUp using exactly the same situation as in Fig. 5, but with the number of processors fixed at 1024. Overall, ScanUp scheduling also exhibits good processor utilization for proportional and uniform distribution (the RETR curves of DQT and ScanUp are almost overlapped in the figure). However, it exhibits relatively lower processor utilization for inversely proportional distribution.

Interestingly, only a small degradation in processor utilization can be found in high-load situations with Fair-DQT. The possible explanation of this phenomenon is that the latecoming tasks cancel fragmentation. The linearity of DQT in processor utilization found in Fig. 5 is also obtained for the same reason. Figure 7 shows RETR curves of Fair-DQT under the same condition as in Fig. 6. The DQT scheduling strategy always results in shorter mean RETRs. In these graphs, we show mean RETR values. However, the situation is the same with the maximum values of RETR. Thus the DQT scheduling strategy is very

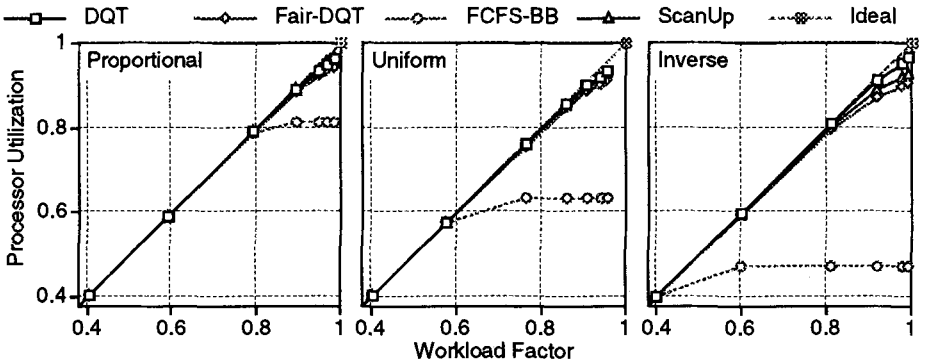


Fig. 6. DQT vs. Batch Scheduling

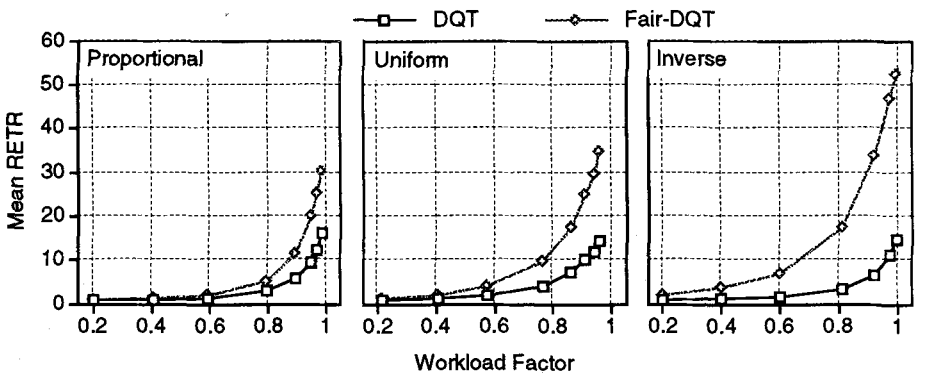


Fig. 7. RETR of Fair-DQT

efficient to reduce RETR values. This means that the strategy can result in a shorter response time.

As expected, FCFS-BB exhibits the worst performance. The processor utilization depends strongly on the task size distribution. This phenomenon comes from external fragmentation. Larger tasks are forced to wait for the space occupied by smaller and longer task(s).

4.4 Scheduling Fairness

In Fig. 8, the correlation coefficient between task size dimension ($log_2(task\_size)$) and RETR obtained at the simulation results in Fig. 6 on DQT, Fair-DQT,

and ScanUp scheduling is plotted . In most cases, the correlation coefficients of DQT are positive. This means the larger the task size, the less the opportunity for scheduling. As described in Section 3.1, this phenomenon happens because smaller tasks tend to be scheduled more often to keep processors as busy as possible. As supposed, Fair-DQT exhibits better fairness.

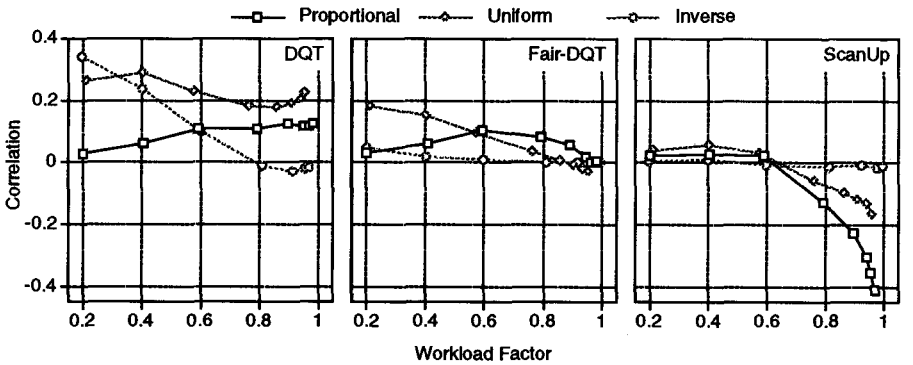


Fig. 8. Task Size Correlation (2^{10} processors)

With uniform and inversely proportional distributions, the correlation coefficient of DQT tends to be higher in lower-load situations. This phenomenon may be because DQT (and Fair-DQT as well) may not have a sufficient number of tasks to fill the load imbalance. A positive correlation coefficient has one advantage; users might hesitate to enter large tasks and this situation could prevent the thoughtless wasting processor utilities.

It was reported that ScanUp scheduling has good scheduling fairness [5]. However, we found that the correlation coefficient between task size and response time depends on the task size distribution and workload. With the inverse distribution, fairness is almost guaranteed. However, when larger tasks are entered more often, the response time for larger tasks becomes shorter for higher workloads.

5 Summary

We have explained a new scheduling class called Time Space Sharing Scheduling combining time-sharing and space-sharing for partitionable parallel machines. On evaluation through simulations, we found that our proposed DQT [3], which is an instance of TSSS, performs as well as ScanUp scheduling which is possibly the best batch scheduling system so far [5]. DQT shows good linearity over given workloads, and good independency from task size distribution and system size.

In terms of fairness, however, both DQT and ScanUp scheduling exhibit some dependency on task size distribution and workload.

Importantly, this paper shows that TSSS can inherently achieve higher processor utilization due to the cancellation of external fragmentation of processor space by late-coming tasks. Thus, the scheduling overhead of TSSS may be acceptable.

DQT scheduling will be implemented on the RWC-1, parallel machine that is under development in our RWC project [7]. RWC-1 will be implemented with some architectural support for TSSS. With TSSS and architectural support [4], an interactive programming environment can be implemented on parallel machines and it is as practical as time-sharing on sequential workstations.

References

1. P.-J. Chuang and N.-F. Tzeng. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. *IEEE Transactions on Computers*, 41(4):467–479, 1992.
2. D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *COMPUTER*, pages 65–77, May 1990.
3. A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo. A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume II, pages 173–182. IEEE Computer Society Press, January 1995.
4. A. Hori, T. Yokota, Y. Ishikawa, S. Sakai, H. Konaka, M. Maeda, T. Tomokiyo, J. Nolte, H. Matsuoka, K. Okamoto, and H. Hirono. Time Space Sharing Scheduling and Architectural Support. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1995.
5. P. Krueger, T.-H. Lai, and V. A. Dixit-Radiya. Job Scheduling Is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):488–497, 1994.
6. J. L. Peterson and T. A. Norman. Buddy System. *Communication of the ACM*, 20(6):421–431, June 1977.
7. S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Proceedings of 1993 International Conference on Supercomputing*, pages 251–260, 1993.
8. Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.
9. Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.