Job Scheduling Scheme for Pure Space Sharing among Rigid Jobs

Kento Aida[1], Hironori Kasahara[2], and Seinosuke Narita[2]

[1] Department of Mathematical and Computing Sciences,
Tokyo Institute of Technology
2-12-1, O-okayama, Meguro-ku, Tokyo, JAPAN 152
aida@noc.titech.ac.jp
http://www.noc.titech.ac.jp/~aida

[2] Department of Electrical, Electronics and Computer Engineering,
Waseda University
3-4-1, Ohkubo, Shinjuku-ku, Tokyo, JAPAN 169
{kasahara,narita}@oscar.elec.waseda.ac.jp
http://www.oscar.elec.waseda.ac.jp/

Abstract. This paper evaluates the performance of job scheduling schemes for pure space sharing among rigid jobs. Conventional job scheduling schemes for the pure space sharing among rigid jobs have been achieved by First Come First Served (FCFS). However, FCFS has a drawback such that it can not utilize processors efficiently. This paper evaluates the performance of job scheduling schemes that are proposed to alleviate the drawback of FCFS by simulation, performance analysis and experiments on a real multiprocessor system. The results showed that Fit Processors First Served (FPFS), which searches the job queue and positively dispatches jobs that fit idle processors, was more effective and more practical than others.

1 Introduction

Parallel processing schemes such as SPMD [1] and Multigrain parallel processing scheme [2] are used on multiprocessor systems. In these schemes, a user or a compiler specifies the number of processors on which the job (program) is executed and optimizes the job (program) considering data locality. On the execution of these jobs, the job requests a certain number of processors. The number of processors that the job requests is specified by a user or a compiler. A job scheduler dispatches the job to the requested number of processors. Each job is exclusively executed until its completion on these processors. These jobs are called "rigid jobs" and this kind of scheduling is called "pure space sharing" [3].

The pure space sharing among rigid jobs has several advantages such that implementation is simple and any user can have optimal performance by executing the job on the requested number of processors exclusively. Although more complex scheduling schemes such as gang scheduling and adaptive space sharing are also discussed in the literature [4–6], the pure space sharing among rigid jobs is adopted on most multiprocessor systems currently installed for practical use because of these advantages.

Processor allocation strategies for the pure space sharing among rigid jobs such as Frame Sliding, First Fit, Best Fit and Non-contiguous processor allocation algorithm, which allocate processors to jobs, have been proposed [7–11]. In these strategies, processors are allocated to the job at the head of the job queue. In other words, a job scheduling is achieved by First Come First Served (FCFS) manner. However, FCFS has a drawback such that it can not utilize processors efficiently [9]. In FCFS, when the number of idle processors is less than the number of processors requested by the job at the head of the job queue, a job scheduler does not dispatch any job to processors and causes low processor utilization.

Several schemes have been proposed to alleviate the drawback of FCFS. Queue sorting is the technique that a job scheduler sorts jobs in the job queue by the number of requested processors. Both techniques that sort jobs by non-increasing order and by non-decreasing order were discussed [12,13]. Another technique, which positively dispatches jobs that fit idle processors, was also discussed. Here, "a job that fits idle processors" means "a job that requests processors whose number is not exceeding the number of idle processors." In Fit Processors First Served (FPFS), a job scheduler searches jobs in the job queue and dispatches a job that fits idle processors to processors [14–16] [1]. Backfilling is a similar scheme to FPFS. In Backfilling, when the job at the head of the job queue waits for being dispatched because there are not enough idle processors for it, a job scheduler dispatches other jobs that fit idle processors without affecting the start time of the job at the head of the job queue [13,17,18].

Several job scheduling schemes described in the previous paragraph has been evaluated either by simulation or experiments on real machines [13,16–18]. However, none of work has compared performance of all these job scheduling schemes and has evaluated their performance by an analytical method.

This paper evaluates the performance of job scheduling schemes for pure space sharing among rigid jobs. First, this paper evaluates and compares the performance of these job scheduling schemes by simulation. Next, the performance of two job scheduling schemes that showed best performance in the simulation, FPFS and Fit Processors Most Processors First Served (FPMPFS), is analyzed using queueing model and one-dimensional bin-packing problem. The authors have implemented the FPFS and FPMPFS on a multiprocessor system NEC Cenju-3 [19]. Experimental results on the Cenju-3 are also shown.

The rest of this paper is organized as follows. Section 2 describes the job scheduling model assumed in this paper. Next, Sect. 3 describes job scheduling schemes evaluated in this paper and Sect. 4 shows simulation results. Then, Sect. 5 shows performance analysis results of FPFS and FPMPFS, and Sect. 6 shows experimental results on a multiprocessor system NEC Cenju-3.

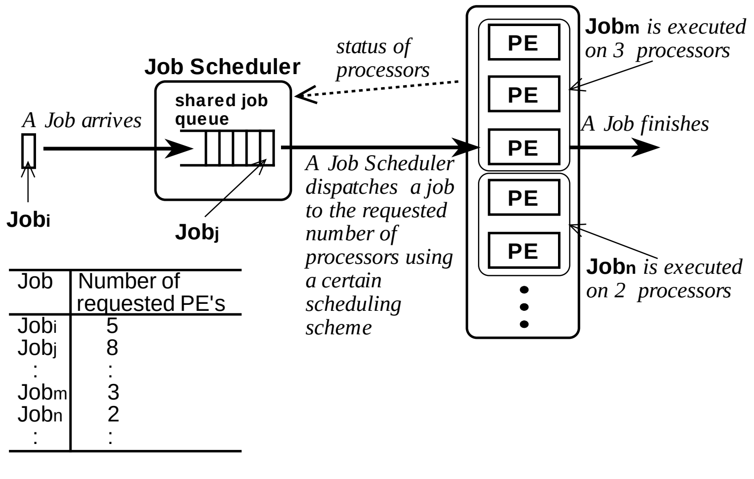
[1] This scheme is referred to as FCFS-fill in [15] and as LSF-RTC in [16]. However, this paper refers to this as FPFS for convenience.

2 Job Scheduling Model

This section describes the job scheduling model assumed in this paper.

The multiprocessor system under consideration consists of a number of processors connected equally by a crossbar network or a multi-stage interconnection network. Since the execution time of a job is insensitive to the location of each processor that executes the job on this multiprocessor system, this paper assumes that a job scheduler can dispatch a job to any idle processor existing arbitrary location.

Figure 1 illustrates the model of the job scheduler. A job arrives at the job queue dynamically. Each job requests a certain number of processors. The number of processors is specified by a user or a compiler. The job scheduler obtains status of processors. Whenever a new job arrives or a job being executed on processors finishes, the job scheduler dispatches the job in the job queue to idle processors using a certain scheduling scheme. At this time, each job is dispatched to idle processors whose number is same as the number that the job requests. The job that has been dispatched to processors is executed exclusively until its completion. For an arrived job, the job scheduler has knowledge about only the number of processors that the job requests, because it is generally difficult that a job scheduler knows the execution time of an arrived job before its execution. However, this paper assumes that the execution time of the job is also known before its execution when Backfilling is applied [2].


Job Scheduler
shared job queue
status of processors
A Job arrives
Jobi
Jobj
A Job Scheduler dispatches a job to the requested number of processors using a certain scheduling scheme
PE
PE
PE
PE
PE
Jobm is executed on 3 processors
A Job finishes
Jobn is executed on 2 processors

Job	Number of requested PE's
Jobi	5
Jobj	8
:	:
Jobm	3
Jobn	2
:	:

Fig. 1. The model of the job scheduler

[2] Backfilling requires execution time of jobs for its scheduling process.

3 Job Scheduling Schemes

This section describes job scheduling schemes evaluated in this paper. Table 1 shows the summary of scheduling schemes evaluated in this paper.

Table 1. Scheduling schemes

schemes	sorting queue	searching queue
FCFS	*no*	*no*
MPFS	*yes*	*no*
LPFS	*yes*	*no*
FPFS	*no*	*yes*
FPMPFS	*yes*	*yes*
FPLPFS	*yes*	*yes*
Backfilling	*no*	*yes*

3.1 FCFS

First Come First Served (FCFS) is a conventional scheme that is currently used on most multiprocessor systems. A job scheduler dispatches the job at the head of the job queue when the number of processors requested by the job is not exceeding the number of idle processors.

3.2 Queue Sorting

Whenever a new job arrives at the job queue, jobs in the job queue are sorted by the number of processors that jobs request. A job scheduler dispatches the job at the head of the sorted job queue when the number of processors requested by the job is not exceeding the number of idle processors. Jobs in the job queue can be sorted by non-increasing order or by non-decreasing order. This paper refers to the former as Most Processors First Served (MPFS) and refers to the latter as Least Processors First Served (LPFS).

The basic idea of MPFS is to utilize idle processors efficiently by sorting jobs by non-increasing order of the number of requested processors. It is known that sorting items by non-increasing order of items' size improves packing efficiency in bin-packing problem [20]. The basic idea of LPFS is to reduce mean response time of jobs by dispatching a large number of small jobs, jobs that request a small number of processors, preferentially.

3.3 FPFS

The basic idea of Fit Processors First Served (FPFS) is to utilize idle processors efficiently by positively dispatching jobs that fit idle processors. The scheduling algorithm of FPFS is as follows. Here, i indicates the order of jobs in the job queue and n indicates the number of jobs in the job queue. $RequestedPE_i$ denotes the number of processors requested by Job_i and $IdlePE$ denotes the number of idle processors.

1. $i = 1$.
2. Compare $RequestedPE_i$ and $IdlePE$.
   (a) if $RequestedPE_i > IdlePE$, go to 3.
   (b) if $RequestedPE_i \leq IdlePE$, dispatch Job_i to $RequestedPE_i$ of processors. Then, $IdlePE = IdlePE - RequestedPE_i$. Go to 3.
3. $i = i + 1$. if $i \leq n \; and \; IdlePE > 0$, go to 2.
4. Go to 1.

The following algorithm is executed when a job being executed on processors finishes.

1. Relinquish processors on which the job has been executed.
2. $IdlePE = IdlePE + number \; of \; relinquished \; processors$

FPFS can be applied with the queue sorting technique. This paper refers to FPFS's with queue sorting as Fit Processors Most Processors First Served (FPMPFS) and Fit Processors Least Processors First Served (FPLPFS). In FPMPFS, a job scheduler sorts jobs in the job queue by non-increasing order of the number of requested processors and then dispatches jobs to processors in the same way as FPFS. In FPLPFS, a job scheduler dispatches jobs in the same way except sorting jobs in the job queue by non-decreasing order. However, FPLPFS causes same job scheduling results as LPFS because of the following reason. During searching jobs sorted by non-decreasing order of the number of requested processors, if a job scheduler finds the job that does not fit idle processors, no more job that remains in the job queue fits idle processors.

3.4 Backfilling

Backfilling is a similar scheme as FPFS except it does not affect the start time of the job at the head of the job queue. When the job at the head of the job queue, Job_{top}, waits for being dispatched because there are not enough idle processors for it, a job scheduler calculates the start time of Job_{top} from the remaining execution time of jobs that are in execution. Then, the job scheduler begins to search the job queue. When the job scheduler finds the job that fits idle processors, Job_{fit}, the job scheduler verifies if dispatching Job_{fit} delays the start time of Job_{top}. If dispatching Job_{fit} does not delay the start time of Job_{top}, the job scheduler dispatches Job_{fit} to idle processors . Otherwise, the job scheduler does not dispatch Job_{fit}. Dispatching Job_{fit} does not delay the start time of Job_{top} in the following cases.

1. Job_{fit} is going to finish and release processors before the start time of Job_{top}.
2. Job_{fit} is not going to finish before the start time of Job_{top}. However, another job is going to finish and release processors so that there are going to exist enough idle processors for Job_{top} before its start time.

Backfilling has difficulty for practical use, because it assumes that the execution time of each job is known before its execution. Generally, it is difficult that a job scheduler knows execution time of an arrived job before its execution. Therefore, this paper assumes that a job scheduler has knowledge about only the number of processors requested by the job. However, in order to compare the performance of Backfilling with others, this paper also assumes that the execution time of an arrived job is known before its execution when Backfilling is applied.

3.5 Avoiding Starvation

It is possible that starvation occurs in FPFS, FPMPFS, FPLPFS, MPFS and LPFS. In order to avoid starvation, *WaitLimit*, which is a deadline that a job continues waiting for being dispatched to processors, is given to each job.

In FPFS, FPMPFS and FPLPFS with *WaitLimit*, a job scheduler gives a priority to $Job_{overWaitLimit}$, which is a job waiting for being dispatched after its *WaitLimit*. The algorithm of FPFS, FPMPFS and FPLPFS with *WaitLimit* can be derived by changing 2(a) of the algorithm in Sect. 3.3 as follows.

```
2. Compare $RequestedPE_i$ and $IdlePE$.
   (a) if $RequestedPE_i > IdlePE$,
       if $Job_i$ is $Job_{overWaitLimit}$, go to 4, else go to 3.
```

In schemes that apply the queue sorting, or MPFS, LPFS, FPMPFS and FPLPFS, a job scheduler suppresses sorting jobs when there is a $Job_{overWaitLimit}$ in the job queue. In other words, a job scheduler does not enter any of newly arrived jobs at the position before $Job_{overWaitLimit}$ in the job queue.

4 Simulation

This section describes the performance evaluation of job scheduling schemes by simulation. In this simulation, the performance is measured by processor utilization, mean response time and the variance of response time.

4.1 Simulation Model

The multiprocessor system in this simulation is assumed to be as what described in Sect. 2.

The execution time of a job is exponentially distributed and the average execution time is 10[sec.]. The execution time includes overhead for loading a program and so on.

Job arrival is assumed to be Poisson and load is defined by (1).

$$load = \rho = \frac{\lambda \cdot p}{m \cdot \mu} \qquad (1)$$

Here, λ denotes the mean arrival rate of arrived jobs. μ denotes the mean service rate per processor, or $1/\mu$ denotes the mean execution time of a job. p indicates the mean number of processors requested by the job and m denotes the total number of processors on the multiprocessor system. Scheduling overhead for each job scheduling scheme is assumed to be negligible.

The number of processors on the multiprocessor system and the number of processors requested by a job are assumed to be the alternative of following two models.

1. *Uniform Dist.*
   The number of processors requested by a job is uniformly distributed on $[1, 256]$, and the multiprocessor system has 256 processors.
2. *Real Dist.*
   The number of processors requested by a job follows the distribution on Table 2. Table 2 is obtained by recent 10027 jobs that have been executed on a real multiprocessor system NEC Cenju-3 [3] installed in the authors' laboratory. The Cenju-3 has eight processors. In the distribution, the number of jobs that request powers of two processors is larger than others, and this characteristic is consistent with previous reports [13, 21]. Furthermore, the number of jobs that request eight processors is largest. It means that many users have attempted to execute their programs on all processors, or eight processors, because the Cenju-3 was a small system.

Table 2. The number of requested processors

number of requested processors	ratio
1	0.1698
2	0.1718
3	0.0464
4	0.1837
5	0.0295
6	0.0316
7	0.0357
8	0.3314

In the simulation, 5000 jobs are requested and executed in one replication and 100 replications with different jobs are practiced. All results have confidence intervals of 10% or less at a 95% confidence level. The performance of FPLPFS is represented by LPFS because these two schemes cause same results.

[3] See Sect. 6 for details.

4.2 Processor Utilization and Mean Response Time

Figure 2 through Fig. 6 show processor utilization and mean response time by the simulation where the number of processors requested by a job follows *Uniform Dist.*

Figure 2 and Fig. 3 show processor utilization and mean response time by the job scheduling schemes. In these results, *WaitLimit* is not given to any job. Figure 2 and Fig. 3 show that MPFS and LPFS improve processor utilization compared with FCFS. It means that the efficiency of packing jobs into idle processors is improved by queue sorting. LPFS keeps mean response time lower than MPFS and FCFS. It is caused by the reason that a large number of small jobs are dispatched prior to a small number of large jobs, which request a large number of processors, by LPFS. However performance improvement by both MPFS and LPFS is slight. On the other hand, FPFS, FPMPFS and Backfilling improve processor utilization considerably and keep mean response time much lower compared with others. It means that dispatching jobs that fit idle processors improves the efficiency of packing jobs into idle processors more considerably than MPFS and LPFS. The performance of FPFS, FPMPFS and Backfilling is almost same. However, FPMPFS shows slightly better performance than FPFS and Backfilling follows FPFS.

In MPFS, LPFS, FPFS, FPMPFS and FPLPFS, *WaitLimit* should be given to each job to avoid starvation. Figure 4 and Fig. 5 show processor utilization and mean response time where *WaitLimit*, which is equal to 600[sec.], is given to each job. Figure 4 and Fig. 5 show that processor utilization by FPFS and FPMPFS is degraded in high load while the degradation of others is slight. It is caused by the reason that searching jobs in the job queue is suppressed by *WaitLimit* in FPFS and FPMPFS. At low load, the degradation by *WaitLimit* is negligible because the number of jobs that continues waiting for being dispatched for long time is a few. However, FPFS and FPMPFS improve processor utilization compared with FCFS, MPFS and LPFS despite performance degradation by *WaitLimit*. Furthermore, the degradation for mean response time by FPFS and FPMPFS is slight. FCFS and Backfilling show no performance degradation because they did not caused starvation in the simulation. Backfilling shows the best improvement of processor utilization in this case.

Figure 6 shows processor utilization by FPFS when various *WaitLimit*'s are defined. On Fig. 6, WL denotes *WaitLimit*[sec.]. Figure 6 shows that processor utilization is degraded at high load when *WaitLimit* is given. It is clear that the scheduling result by FPFS with $WaitLimit = 0$ is same as FCFS. Therefore, processor utilization by FPFS with *WaitLimit* becomes closer to FCFS as *WaitLimit* decreases. FPMPFS with *Waitlimit* showed almost same results in the simulation.

4.3 Variance of Response Time

Table 3 shows the variance of the response time of jobs when $load = 0.5$. The number of processors requested by a job follows *Uniform Dist. WaitLimit*, which

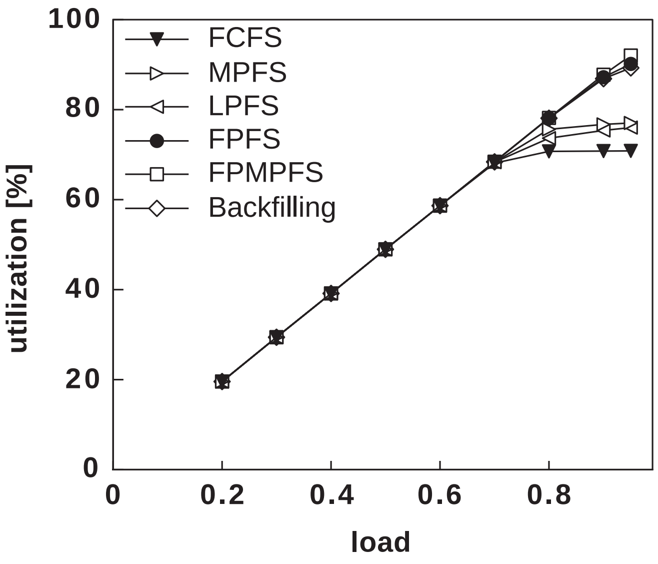


Fig. 2. Processor utilization in simulation results (*Uniform Dist.*)

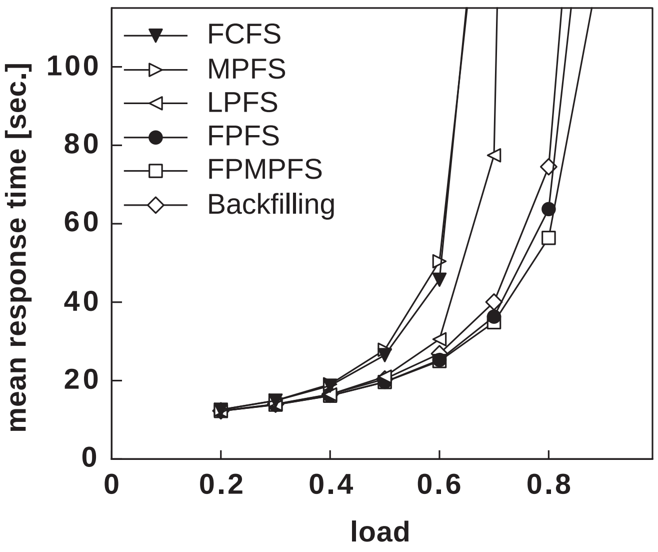


Fig. 3. Mean response time in simulation results (*Uniform Dist.*)

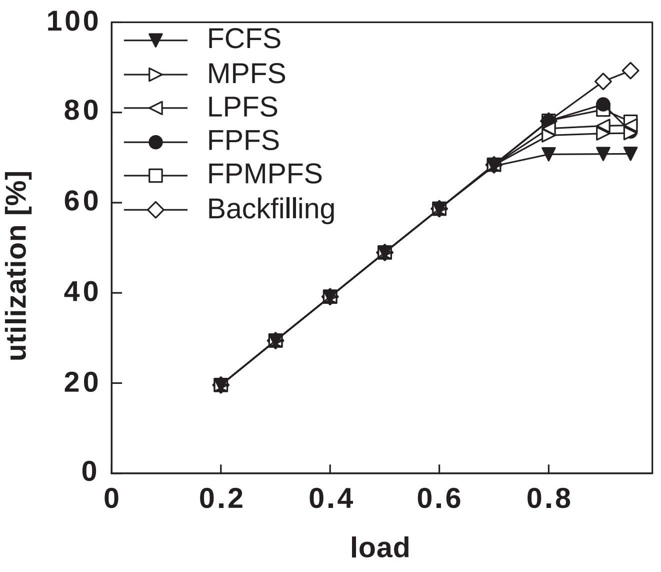


Fig. 4. Processor utilization in simulation results (*WaitLimit* = 600[sec.], *Uniform Dist.*)

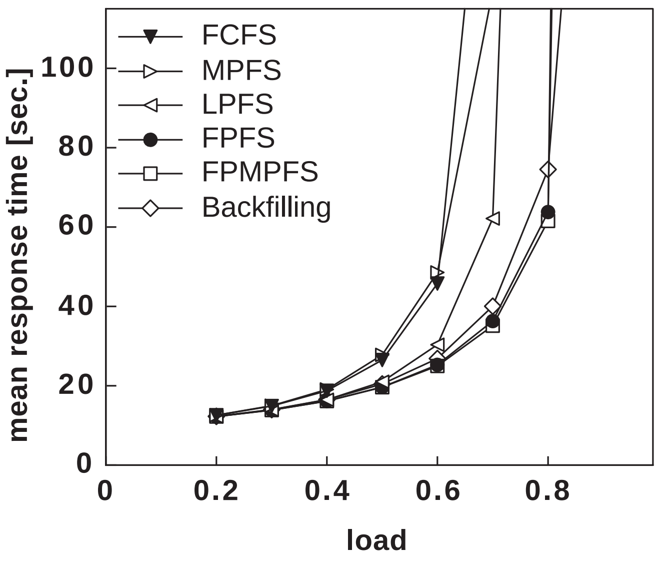


Fig. 5. Mean response time in simulation results (*WaitLimit* = 600[sec.], *Uniform Dist.*)

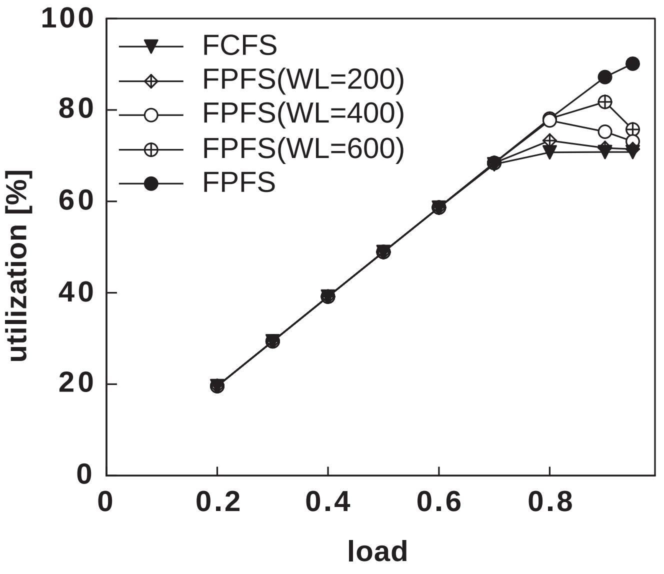


Fig. 6. Processor utilization by FPFS with various *WaitLimit* in simulation results (*Uniform Dist.*)

is equal to 600[sec.] is given to each job. MPFS and LPFS increase variance compared with FCFS. FPFS, FPMPFS and Backfilling reduce variance compared with FCFS. These results show that the technique searching the job queue and positively dispatching jobs that fit idle processors decreases the variance of the response time of jobs, however, sorting jobs in the job queue increases the variance.

Table 3. Variance of response time

schemes	variance
FCFS	639
MPFS	1765
LPFS	1058
FPFS	365
FPMPFS	498
Backfilling	370

4.4 Processor Utilization under *Real Dist.*

Figure 7 shows processor utilization by the simulation where the number of processors requested by a job follows *Real Dist.* Figure 7 shows that FPFS, FPMPFS and Backfilling improve processor utilization considerably. However, the performance of all job scheduling schemes are improved compared with results on Fig. 2. Especially, the performance improvement of MPFS is considerable. It means that the performance is sensitive to the distribution of the number of processors requested by a job. In this case, it seems that the efficiency of packing jobs that follows *Real Dist.* is higher than that follows *Uniform Dist.*, because the number of jobs that request all processors, or eight processors, is large in *Real Dist.*

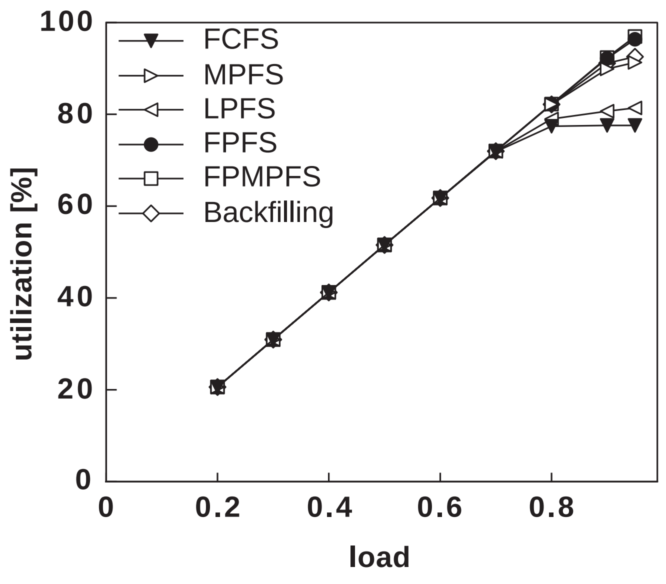


Fig. 7. Processor utilization in simulation results (*Real Dist.*)

4.5 Summary of Simulation Results

The performance of job scheduling schemes obtained by the simulation are classified into two categories.

MPFS and LPFS, which sort the job queue by the number of requested processors, showed almost same performance that is as follows.

1. Both MPFS and LPFS improved processor utilization and reduced mean response time slightly compared with FCFS.

2. The performance degradation by *WaitLimit* of these schemes was slight.
3. These schemes increased the variance of the response time of jobs compared with FCFS.

FPFS, FPMPFS and Backfilling, which search the job queue and positively dispatch jobs that fit idle processors, showed almost same performance that is as follows.

1. FPFS, FPMPFS and Backfilling improved processor utilization and reduced mean response time considerably compared with MPFS, LPFS and FCFS.
2. Although processor utilization by FPFS and FPMPFS was degraded when *WaitLimit* was given to each job, they maintained better performance than FCFS, MPFS and LPFS. In this case, Backfilling showed the best improvement of processor utilization. The degradation of processor utilization depended on value of *WaitLimit*.
3. These schemes decreased the variance of the response time of jobs compared with FCFS.

According to these results, it can be assumed that FPFS, FPMPFS and Backfilling caused better performance improvement than others. It shows that the effectiveness of the technique that searches the job queue and positively dispatches jobs that fit idle processors. However, Backfilling has difficulty for practical use, because it assumes that the execution time of each job is known before its execution. Therefore, this paper assumes that FPFS and FPMPFS cause best performance improvement among the job scheduling schemes discussed in this paper. Simulation results in previous work [13,15] also showed effectiveness of a part of these job scheduling schemes, LPFS and FPFS (or Backfilling). Results obtained in this section are consistent with them.

5 Performance Analysis

This section describes the performance analysis of job scheduling schemes that showed best performance in the simulation, or FPFS and FPMPFS. The performance analysis of FCFS is also described to compare the performance of FPFS and FPMPFS with FCFS. In this analysis, the performance is measured by processor utilization and stability condition.

5.1 Processor Utilization and Stability Condition

In $M/M/m$ queueing model, which assumes that the inter-arrival time and the service time of jobs are exponentially distributed and there are m servers, the stability condition is given by

$$\rho = \frac{\lambda}{m \cdot \mu} < 1 \qquad (2)$$

and the utilization of servers, U, is given by

$$U \begin{cases} = \frac{\lambda}{m \cdot \mu}, & \rho < 1 \\ \to 1, & \rho \geq 1. \end{cases} \tag{3}$$

Here, λ denotes the mean arrival rate of a job and μ denotes the mean service rate per server. The stability condition is the condition to keep the system stable. If the stability condition is satisfied, the mean response time of a job is stable[22]. In other words, if stability condition is not satisfied, the response time of a job rises suddenly.

In the job scheduling model assumed in this paper, an arrived job is executed on the requested number of processors simultaneously. Therefore, the following formula is derived from (2) where p denotes the average number of processors requested by a job.

$$\rho = \frac{\lambda \cdot p}{m \cdot \mu} < 1 \tag{4}$$

In $M/M/m$ queueing model, all of m servers are active when there are jobs waiting for being dispatched in the job queue. However, in the job scheduling model assumed in this paper, when the number of idle processors is not zero but less than the number of processors requested by any job in the job queue, idle processors remain idle. In other words, the number of active servers when there are jobs in the job queue is equal to or less than m. Therefore, the stability condition in the job scheduling model assumed in this paper is given by

$$\frac{\lambda \cdot p}{\upsilon \cdot m \cdot \mu} < 1. \tag{5}$$

Here, $\upsilon \cdot m$ denotes the mean number of active servers when there are jobs waiting for being dispatched in the job queue. In other words, υ is the ratio of active servers to all processors in the multiprocessor system when there are jobs in the job queue. Then, the following formula is derived from (4), (5).

$$\rho < \upsilon \tag{6}$$

and processor utilization, U, is given by

$$U \begin{cases} = \frac{\lambda \cdot p}{m \cdot \mu}, & \rho < \upsilon \\ \to \upsilon, & \rho \geq \upsilon \end{cases} \tag{7}$$

from (4), (6).

The υ defines the stability condition and an upper limit of processor utilization in the job scheduling model assumed in this paper. Therefore, the processor utilization (the upper limit of processor utilization) and the stability condition can be improved by increasing the value of υ.

5.2 Derivation of υ

The value of υ can be derived from one-dimensional bin-packing problem. There are one-dimensional bins, B_j, $(j = 1, \cdots m)$, and a list of one-dimensional items,

$p_i, (i = 1, \cdots n)$. The capacity of B_j is C. The length of p_i, $s(p_i)$, is $s(p_i) \leq C$. One-dimensional bin-packing problem attempts to minimize the number of bins to pack all items in the list satisfying $\sum_{p_i \in Bj} s(p_i) \leq C$. Next Fit (NF), First Fit (FF) and First Fit Decreasing (FFD) are proposed as algorithm for one-dimensional bin-packing problem [20].

The job scheduling model assumed in this paper can be considered as one-dimensional bin-packing problem in which a job scheduler attempts to pack jobs into the idle processors. Here, the job corresponds to the item and "the number of processors requested by the job" is $s(p_i)$. Similarly, the idle processors correspond to the bin and "the number of idle processors" is C. Then, it can be assumed that the value of υ is same as the utilization of B_j in one-dimensional bin-packing problem. In several previous works, two-dimensional bin-packing problem has been used for the job scheduling model in which the job has two-dimensional quantity, the number of requested processors and the execution time [23]. However, one-dimensional bin-packing is suitable for the job scheduling model assumed in this paper because this paper assumes that the execution time of a job is unknown.

Worst Case Analysis. The performance of NF, FF and FFD in the worst case is given as follows [20]. Here, $h_{wc}(A)$ denotes the number of bins to pack n items by algorithm A in the worst case and $h(opt)$ denotes the number of bins to pack n items in the optimal case.

$$\frac{h_{wc}(NF)}{h(opt)} = 2.0, \ n \rightarrow \infty \tag{8}$$

$$\frac{h_{wc}(FF)}{h(opt)} = \frac{17}{10}, \ n \rightarrow \infty \tag{9}$$

$$\frac{h_{wc}(FFD)}{h(opt)} = \frac{11}{9}, \ n \rightarrow \infty \tag{10}$$

Since it is safe to say that there is no fragmentation in bins in the optimal case, utilization by NF, FF and FFD is the reciprocal of (8), (9) and (10) respectively. Therefore, values of υ by FCFS, FPFS and FPMPFS in the worst case, $\upsilon_{wc}(FCFS)$, $\upsilon_{wc}(FPFS)$ and $\upsilon_{wc}(FPMPFS)$ are given as follows respectively.

$$\upsilon_{wc}(FCFS) = \frac{1}{2} = 0.5 \tag{11}$$

$$\upsilon_{wc}(FPFS) = \frac{10}{17} = 0.588 \tag{12}$$

$$\upsilon_{wc}(FPMPFS) = \frac{9}{11} = 0.818 \tag{13}$$

These results show that FPFS and FPMPFS improve processor utilization and stability condition in the worst case as compared with FCFS. These results also show that FPMPFS improves performance more than FPFS.

Average Case Analysis. The performance of NF in the average case where $s(p_i)$ is uniformly distributed on $(0, 1]$ is given as follows [24]. Here, $h_{ac}(A)$ denotes the number of bins to pack n items by algorithm A in the average case and $h(opt)$ denotes the number of bins to pack n items in the optimal case.

$$\frac{h_{ac}(NF)}{h(opt)} = \frac{4}{3}, \ n \to \infty \tag{14}$$

Next, the performance of FF in the average case is given by

$$h_{ac}(FF) = h(opt) + \theta(n^{\frac{2}{3}}) \tag{15}$$

[24,25]. Since $s(p_i)$ is uniformly distributed on (0,1], $h(opt) = n/2$. Then,

$$\frac{h_{ac}(FF)}{h(opt)} = 1 + \frac{\theta(n^{\frac{2}{3}})}{n/2} = 1 + \theta(n^{-\frac{1}{3}}) \tag{16}$$

is derived from (15) and then,

$$\frac{h_{ac}(FF)}{h(opt)} = 1, \ n \to \infty. \tag{17}$$

In the same way, the performance of FFD in the average case is given by

$$h_{ac}(FFD) = h(opt) + \theta(n^{\frac{1}{2}}) \tag{18}$$

[24]. Then,

$$\frac{h_{ac}(FFD)}{h(opt)} = 1, \ n \to \infty. \tag{19}$$

From (14), (17) and (19), values of v by FCFS, FPFS and FPMPFS in the average case, $v_{ac}(FCFS)$, $v_{ac}(FPFS)$ and $v_{ac}(FPMPFS)$ are given as follows respectively.

$$v_{ac}(FCFS) = \frac{3}{4} = 0.75 \tag{20}$$

$$v_{ac}(FPFS) = 1 \tag{21}$$

$$v_{ac}(FPMPFS) = 1 \tag{22}$$

These results show that FPFS and FPMPFS improve processor utilization and stability condition in the average case as compared with FCFS.

5.3 Comparison with Simulation Results

Figure 8 shows processor utilization obtained by the simulation (on Fig. 2) and results of average case analysis. On Fig. 8, solid lines indicate processor utilization in the simulation and dotted lines indicate results of the average case analysis. Figure 8 shows that processor utilization by FCFS is saturated at nearby $load = 0.75$ or more, while processor utilization by FPFS and FPMPFS maintain to improve at high load.

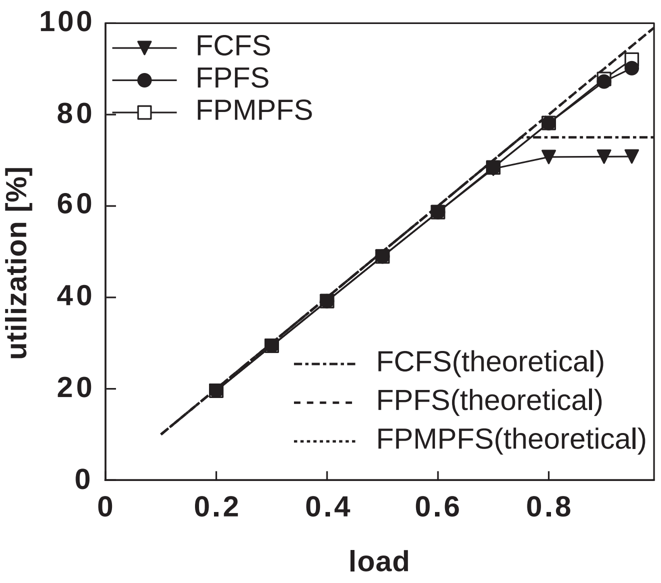


Fig. 8. Processor utilization in simulation results and average case analysis

Furthermore, on Fig. 3, mean response time by FCFS rises suddenly when load is less than 0.75, because the stability condition is not satisfied at $load \geq 0.75$. On the other hand, FPFS and FPMPFS keep mean response time lower than FCFS, because FPFS and FPMPFS improve the stability condition compared with FCFS. These results show that the performance of FCFS, FPFS and FPMPFS in the simulation follows results in the average case analysis.

6 Experiments on a Cenju-3

This section describes experimental results of FPFS and FPMPFS on a multiprocessor system NEC Cenju-3.

6.1 Architecture of a Cenju-3

Cenju-3 is composed of up to 256 processor elements (PE's) connected by a multistage interconnection network like baseline network. The system is connected to the workstation, which acts as a host computer. Each PE has a microprocessor VR4400 and up to 64 ,MB local memory. The multi-stage interconnection network is composed of 4×4 switches, and its maximum throughput is 40 ,MB/s [19].

Figure 9 illustrates architecture of a multiprocessor system NEC Cenju-3 Model 8S that is used for the experiments. Cenju-3 Model 8S has eight PE's.

The peak performance of a single PE is 33.3 MFlops and each PE has 32 ,MB local memory in the system. The host computer, NEC EWS4800/330 EX, is connected to the system.

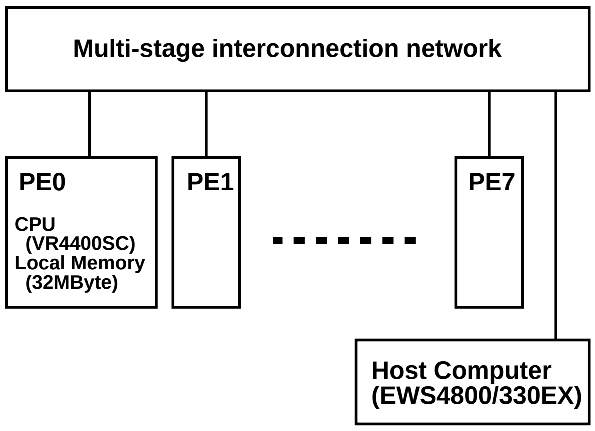


Fig. 9. Architecture of a Cenju-3 Model 8S

6.2 Job Scheduling Mechanism on a Cenju-3

The job scheduling mechanism on a Cenju-3 is achieved by two processes executed on a host computer, Job Scheduler and Maser. Job Scheduler decides the scheduling of arrived jobs and sends the scheduling result to MASER. MASER watches information about PE's, sends the information to Job Scheduler and executes the job on PE's following the scheduling result sent by Job Scheduler [26].

The authors have developed the new job scheduler based on the original Job Scheduler to evaluate performance of FPFS and FPMPFS. Figure 10 shows basic scheduling routines on the new job scheduler. On Fig. 10, FCFS_Scheduler or FPFS_Scheduler dispatches jobs to processors by FCFS or FPFS respectively. A job scheduler executes one of these routines repeatedly. FPMPFS_job_spooler registers arrived jobs into the job queue and sorts jobs by non-increasing order of the number of requested processors. FPFS_Scheduler is also used for FPMPFS. Processor_relinquishment relinquishes processors on which the job finishes execution.

6.3 Results on a Cenju-3

In the experiment, the authors executed 500 jobs composed of three application programs, Electro-magnetic field analysis [27], 3D multigrid [28] and Sparse matrix solver by Gauss-Seidel method. Arrival of these jobs are Poisson. All PE's

```
queue[QUEUEMAX];        /* job queue */
queue[i].job;            /* a job at i th position in a job queue */
queue[i].job.penum;      /* number of demanded PE's by queue[i].job */
queue[i].job.wl;         /* WaitLimit  of queue[i].job */
QueueLength;             /* number of jobs in a job queue */
IdlePeNumber;            /* number of idle processors */
ArrivalJob;              /* arrival job */
ArrivalJob.penum;        /* number of requested PE's by ArrivalJob */
FinishJob;               /* job which finished its execution */
FinishJob.penum;         /* number of demanded PE's by FinishJob */

FCFS_Scheduler()
{
   while (){
      if(queue[1].job.penum <= IdlePeNumber){
         Dispatch_queue[1].job_to_Processors;
         IdlePeNumber - = queue[1].job.penum;
      }
}

FPFS_Scheduler()
{
   while (){
      i = 1;
      while(i <= QueueLength && IdlePeNumber > 0){
         if(queue[i].job.penum <= IdlePeNumber){
            Dispatch_queue[i].job_to_Processors;
            IdlePeNumber - = queue[i].job.penum;
            i++;
         } else {
            if(queue[i].job.wl <= waiting_time_of_queue[i].job )
                break;
            else
                i++;
         }
      }
   }
}

FPMPFS_job_spooler()
{
   i = QueueLength;
   while(i > 0){
      if(queue[i].job.wl <= waiting_time_of_queue[i].job )
         break;
      if(queue[i].job.penum < Arrivaljob.penum)
         i - -;
      else
         break;
   }
   Insert_Arrivaljob_into_(i+1 )th_position_in_a_job_queue;
}

Processor_relinquishment()
{
   Release_processors_which_executed_Finishjob;
   IdlePeNumber += FinishJob.penum;
}
```

Fig. 10. Basic structure of job scheduling routines on a Cenju-3

on the Cenju-3 are dedicated to the experiment. Among these 500 jobs, the number of processors requested by a job is uniformly distributed on [1,8] and the average execution time of a job including time for loading a program and so on is 32[sec.]. Each job is given $WaitLimit$, which is equal to 600[sec.], for FPFS and FPMPFS.

Figure 11 and Fig. 12 show processor utilization and mean response time in the experiment. FCFS(native) denotes the native version of FCFS implemented on a Cenju-3 generally and FCFS(improved) denotes the improved version of FCFS that the authors have developed newly. Difference between FCFS(native) and FCFS(improved) is scheduling overhead. In FCFS(native), Job Scheduler inquires MASER periodically to obtain the number of idle processors and this process requires large overhead. In FCFS(improved), Job Scheduler has a local data, or table, to watch the number of idle processors.

Figure 11 and Fig. 12 show that FPFS and FPMPFS improve processor utilization and keep mean response time lower compared with FCFS. FPFS improves processor utilization by 9[%] compared with FCFS(improved) and by 19[%] compared with FCFS(native) at $load = 0.9$. Mean response time by FCFS(native) and FCFS(improved) rises suddenly at $load = 0.7$ and $load = 0.8$ respectively, however, FPFS and FPMPFS keep mean response time below 116[sec.] at $load = 0.8$.

Processor utilization by FPFS and FPMPFS is degraded to 81[%] at $load = 0.95$ because of the influence by $WaitLimit$. Processor utilization by FPMPFS is degraded by 3[%] compared with FPFS at $load = 0.9$, because searching jobs in the job queue in FPMPFS was suppressed 12 times more than FPFS. In other words, FPMPFS suffered the influence by $WaitLimit$ more than FPFS. FCFS(improved) shows much better performance than FCFS(native), because FCFS(native) requires larger overhead to watch the number of idle processors than FCFS(improved).

These experimental results on a Cenju-3 show that FPFS and FPMPFS improve processor utilization and keep mean response time lower, or improve stability condition, compared with FCFS as results in the simulation and the performance analysis.

7 Conclusions

This paper evaluated the performance of job scheduling schemes for pure space sharing among rigid jobs. More complex scheduling schemes such as gang scheduling and adaptive space sharing are discussed in the literature. However, the discussion of job scheduling schemes for pure space sharing among rigid jobs is still important, because these schemes are adopted on most multiprocessor systems currently installed for practical use. The performance of these job scheduling schemes has been discussed in the literature. In most of the previous work, the performance has been evaluated by either simulation, performance analysis or experiments. However, this paper evaluated performance of the job scheduling

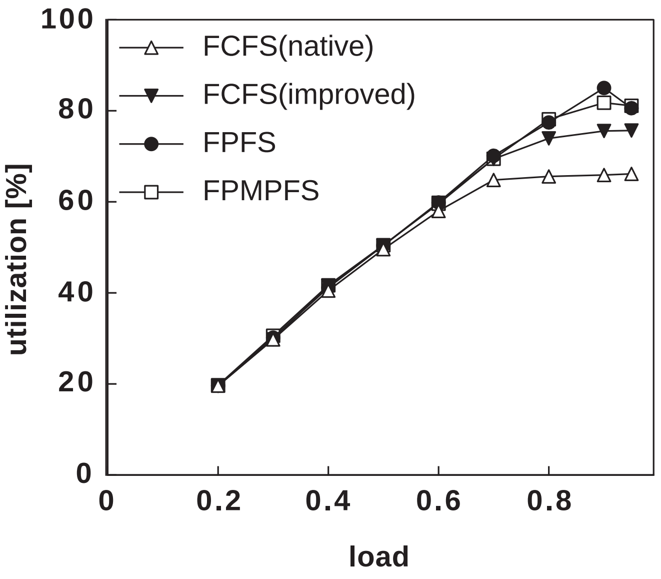


Fig. 11. Processor utilization on a Cenju-3

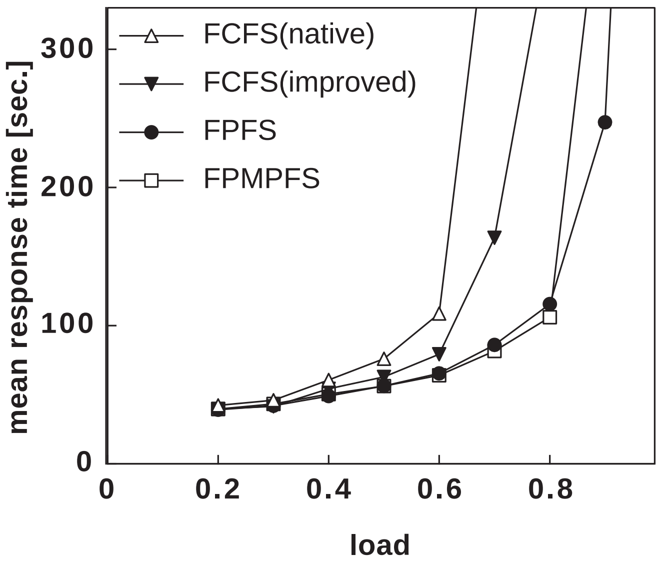


Fig. 12. Mean response time on a Cenju-3

schemes by combination of simulation, performance analysis and experiments to verify the effectiveness and the practicality of these schemes.

Simulation results showed that FPFS, FPMPFS and Backfilling caused considerable performance improvement compared with others. This result means that searching the job queue and positively dispatching jobs that fit idle processors can utilize processors more efficiently and keep mean response time lower than others. However, Backfilling has difficulty for practical use, because it assumes that the execution time of each job is known before its execution. Performance analysis of job scheduling schemes that showed best performance in the simulation, or FPFS and FPMPFS, showed that these schemes improved processor utilization and stability condition compared with FCFS in both worst case and average case. The comparison of results in the simulation and those in the average case analysis showed that simulation results followed the analysis. Experimental results on a multiprocessor system NEC Cenju-3 also showed the advantage of FPFS and FPMPFS as the simulation and the performance analysis.

According to these results, this paper concludes that,

1. FPFS and FPMPFS, which search the job queue and positively dispatch jobs that fit idle processors, are more effective and more practical than other job scheduling schemes discussed in this paper. Although Backfilling can also improve performance, it has difficulty for practical use because it requires knowledge about the execution time of an arrived job before its execution.
2. Performance improvement by FPFS and FPMPFS is almost same. Therefore, FPFS is more practical than FPMPFS because the algorithm of FPFS is simpler than FPMPFS in average case.

Results in Sect. 4.4 showed that the performance of job scheduling schemes was sensitive to the distribution of the number of processors requested by a job. Further investigation on the sensitivity is required as future work.

References

1. High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0, 1993.
2. H. Kasahara, H. Honda, K. Aida, M. Okamoto, and S. Narita. OSCAR Fortran Compiler. In *Proc. Workshop on Compilation of Languages for Parallel Computers*, pages 30–37, 1991.
3. D. G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1162*, pages 1–26. Springer-Verlag, 1996.
4. S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proc. of 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.
5. A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proc. of 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

6. C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, 11(2):146–178, 1993.
7. K. Li and K. Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *J. Parallel and Distributed Computing*, 12:79–83, 1991.
8. P. Chuang and N. Tzeng. An Efficient Submesh Allocation Strategy for Mesh Computer Systems. In *Proc. of International Conference on Distributed Computing Systems*, pages 256–263, 1991.
9. Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected parallel Computers. *J. Parallel and Distributed Computing*, 16:328–337, 1992.
10. V. Lo, K. J. Windisch, W. Liu, and B. Nitzberg. Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers. *IEEE Trans. on Parallel and Distributed Systems*, 8(7):712–726, 1997.
11. D. G. Feitelson and L. Rudolph. Parallel Job Scheduling: Issues and Approaches. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 949*, pages 1–18. Springer-Verlag, 1995.
12. K. Li and K. Cheng. Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):413–422, 1991.
13. J. Subhlok, T. Gross, and T Suzuoka. Impact of Job Mix on Optimizations for Space Sharing Scheduler. In *Proc. of Supercomputing '96*, 1996.
14. K. Aida, H. Kasahara, and S. Narita. A Scheduling Scheme of Parallel Jobs to Processor Groups on a Multiprocessor System. *Trans. of IEICE*, J80-D-I(6):463–473, 1997, (in Japanese).
15. R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1291*, pages 58–77. Springer-Verlag, 1997.
16. E. W. Parsons and K. C. Sevcik. Implementing Multiprocessor Scheduling Disciplines. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1291*, pages 166–192. Springer-Verlag, 1997.
17. D. A. Lifka. The ANL/IBM SP Scheduling System. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 949*, pages 295–303. Springer-Verlag, 1995.
18. J. S. Skovira, W. Chan, and H. Zhou. The EASY - LoadLeveler API Project. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1162*, pages 41–47. Springer-Verlag, 1996.
19. T. Maruyama, Y. Kanoh, T. Hirose, T. Nakata, K. Muramatsu, Y. Asano, and Y Inamura. Architecture of a Parallel Machine: Cenju-3. *IEICE Trans. The Institute of Electronics, Information and Communication Engineers*, J78-D-I(2):59–67, 1995, (in Japanese).
20. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin-packing - An Updated Survey. In *Algorithm Design for Computer System Design*, pages 49–106. Springer-Verlag, 1984.
21. S. Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1162*, pages 27–40. Springer-Verlag, 1996.
22. R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.
23. Y. Zhu and M. Ahuja. On Job Scheduling on a Hypercube. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):62–69, 1993.

24. E. G. Coffman and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms.* Wiley, 1991.
25. P. W. Shor. The Average-case Analysis of Some On-line Algorithms for Bin Packing. *Combinatorica*, 6(2):179–200, 1986.
26. NEC. *NEC Parallel Computer Cenju-3 User's Manual*, 1994, (in Japanese).
27. T. Sakamoto, Y. Maekawa, S. Wakao, T. Onuki, and H. Kasahara. Parallelization of the Electro-magnetic Field Analysis Application Using Hybrid Finite Element and Boundary Element Method. In *Proc. 52th Annual Convention IPSJ*, pages 4L–8, 1996, (in Japanese).
28. D. Bailey, E. Barszcz, J. Barton, D. Browningand R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report BNR-94-007, NASA Ames Research Center, 1994.