Improving First-Come-First-Serve Job Scheduling by Gang Scheduling *

Uwe Schwiegelshohn and Ramin Yahyapour

Computer Engineering Institute, University Dortmund
44221 Dortmund, Germany
uwe@ds.e-technik.uni-dortmund.de
yahya@ds.e-technik.uni-dortmund.de

Abstract. We present a new scheduling method for batch jobs on massively parallel processor architectures. This method is based on the First–come–first–serve strategy and emphasizes the notion of fairness. Severe fragmentation is prevented by using gang scheduling which is only initiated by highly parallel jobs. Good worst–case behavior of the scheduling approach has already been proven by theoretical analysis. In this paper we show by simulation with real workload data that the algorithm is also suitable to be applied in real parallel computers. This holds for several different scheduling criteria like makespan or sum of the flow times. Simulation is also used for determination of the best parameter set for the new method.

1 Introduction

The job scheduling strategy is one of the key elements in the resource management of a massively parallel processor (MPP). It is therefore of interest to both, the manufacturer and the owner of such a machine. For the manufacturer a better management system may be used as an additional marketing argument while also requiring more development effort to implement the necessary system components. On the other hand, the owner would like to see a flexible system which can be fine tuned to his specific environment. Unfortunately, this usually requires significant effort from the system administrator to select the correct parameter setting. In both cases the performance evaluation of a strategy together with a parameter set is a necessity.

Unfortunately, such a performance evaluation is a difficult task. Theoretical worst case analysis is only of limited help as typical workloads on production machines never exhibit the specific structure that will create a really bad case. Further, there is no random distribution of job parameter values, see e.g. Feitelson and Nitzberg [4]. Hence, a theoretical analysis of random workloads will not provide the desired information either. A trial and error approach on a commercial machine will be tedious and may affect system performance in a significant

* Supported by the NRW Metacomputing grant

fashion. Therefore, most users will probably object to such an approach except for the final fine tuning. This just leaves simulation for all other cases.

Simulation may either be based on real trace data or on a workload model. While workload models, see e.g. Jann et al. [14] or Feitelson and Nitzberg [4], enable a wide range of simulations by allowing job modifications, like a varying amount of assigned processor resources, the concurrency with real workloads is not always guaranteed. This is especially true in the case of workloads whose characteristics change over time. On the other hand, trace data restrict the freedom to select different allocation and scheduling strategies as the performance of a specific job is only known under the given circumstances. For instance, trace data specifying the execution time of a batch job do not provide similar information, if the job would be assigned to a different subset of processors. Therefore, the selection of the base data for the simulation depends on the circumstances determined by the scheduling strategy and the MPP architecture. There are already a variety of examples for simulation based on a workload model, see e.g. Feitelson [2], Feitelson and Jette [3] or on trace data, see e.g. Wang et al. [21]. Here, we describe the use of trace data for the evaluation of different parameter settings for a preemptive scheduling strategy.

In particular, we address a specific kind of gang scheduling to be used in the management system of the IBM RS/6000 SP2, a widely available commercial parallel computer. Gang scheduling has already been subject of many studies, see e.g. Feitelson and Rudolph [5, 7], and a significant number of manufacturers, like Intel, TMC or SGI, have included gang scheduling into their operating systems. A gang scheduling prototype for the SP2 has also been presented [11, 20]. The performance of this prototype has already been analyzed by Wang et al. [21]. Although these results were in favor of gang scheduling for the SP2, typical installations do not include it yet.

In this paper we present a new scheduling strategy consisting of a combination of gang scheduling together with a First-come-first-serve approach. Moreover, we show that the performance of gang scheduling is dependent on the selected parameters of the scheduler and that significant gains are achievable. In addition, we address the problem of finding a suitable criterion for measuring the scheduler performance.

To this end we first derive a machine model and a job scheduling model for the IBM RS/6000 SP2 based on the description of Hotovy [13]. Next, we describe the workload data obtained during a period of 11 months from the Cornell Theory Center (CTC) SP2. Then, various scheduling objectives are discussed. Based on those objectives we suggest a new *fairness* criterion. In Section 5 our new scheduling concept is introduced and its parameters are explained. Finally, we present our simulation results and discuss some of them in detail.

2 The Model

2.1 The Machine Model

We assume an MPP architecture where each node contains one or more processors, main memory, and local hard disks while there is no shared memory. The system may contain different types of nodes characterized e.g. by type and number of processors, by the amount of memory or by the specific task this node is supposed to perform. An example for the last category are those nodes which provide access to mass storage. In particular, IBM SP2 architectures may presently contain three types of nodes: *thin nodes*, *wide nodes*, and *high (SMP) nodes*. A wide node usually has some kind of server functionality and contains more memory than a thin node while there is little difference in processor performance. Recently introduced high nodes contain several processors and cannot form a partition with other nodes at this time. However, it should be noted that most installations are predominantly equipped with thin nodes. In 1996 only 48 out of a total of 512 nodes in the CTC SP2 were wide nodes while no high nodes were used at all. Although in most installations the majority of nodes have the same or a similar amount of memory, a wide range of memory configurations is possible. For instance, the CTC SP2 contains nodes with memory ranging from 128 MB to 2048 MB with more than 80% of the nodes having 128 MB and an additional 12% being equipped with 256 MB. For more details see the description of Hotovy [13]. For these reasons we use a model where all nodes are identical.

Fast communication between the nodes is achieved via a special interconnection network. This network does not prioritize clustering of some subsets of nodes over others as in a hypercube or a mesh, i.e. the communication delay and the bandwidth between any pair of nodes is assumed to be constant. The network aspect and its consequences on job scheduling will be further discussed in the next subsection.

In our model the computer further supports gang scheduling by switching simultaneously the context of a subset of nodes. This context switch is assumed to be executed by use of the local processor memory and/or the local hard disk while the interconnection network is not affected except for synchronization [20]. Here, we use gang scheduling (preemption) without migration, that is no change of the node subset assigned to a job will occur during job execution. Any context switch may result in a time penalty due to processor synchronization, draining the networks of not yet delivered messages, saving of job status, page faults, and cache misses. In the simulation this penalty is considered by adding a constant time delay to the execution time of a job for each time the job is preempted. In our model no node of an affected subset is able to execute any part of a job during this time delay. Note that our scheduler deviates substantially from the prototype scheduler presented in [20] and analyzed in [21].

2.2 The Job Model

Our job model is also derived from Hotovy's description [13]. For the simulation we restrict ourselves to batch jobs as for interactive jobs the execution time cannot be assumed to be independent of the starting time and of the number of preemptions. Therefore, we just consider the batch partition of the CTC SP2 which comprises 430 nodes. As highly parallel interactive jobs are rather unlikely our scheduling schemes have little or no effect on the interactive partition as will become clear from the description of the scheduling algorithm in Section 5.

At the CTC SP2 a job is assigned to a subset of nodes in an exclusive fashion once it is supposed to be executed. This subset is described by the number of nodes it must contain from each type of node. Any subset matching this description is assumed to require the same time for execution of the job as the interconnection network does not favor any specific subset. Therefore, free variable partitioning [6] is supported. Once a job is started on a subset it is run to completion or canceled if its time limit has expired.

As we are using gang scheduling our model deviates from the description above. We require that at any time instant any node executes at most a single job while a second job may have been preempted and now waits for resumption of its execution. However, on each node there may only be a single preempted job at any moment. Note that data and status of the preempted job can be stored on the local hard disk of a node in the same way as the disk is presently used for swapping. Further, gang scheduling requires that at any time instant either all nodes allocated to a job run this job or none of them does.

The scheduling policy of the CTC is described in detail by Hotovy [13]. Here, we just want to give a brief summary of it: A user specifies the resource requirements of his batch job and assigns it to a batch queue. The batch queues are defined by the maximum job execution time (wall clock time), that is the permitted time limit of a job. The resource requirements include the maximum and the minimum number of nodes the job needs. This feature is disregarded in our simulation as we do not know from the workload traces how the execution time of a job would change with a different number of nodes allocated to the job. Hence, we assume that for each job the size of the subset is invariable and given by the trace data. Using the terminology of Feitelson and Rudolph [8] we therefore do not allow moldable jobs. As already mentioned, our simulation is based on a machine with identical nodes, although in general the simulator is able to handle specific resource requests as well. Jobs can be submitted at any time. Note that our model allows a combination of space and time sharing.

3 The Workload Data

As already mentioned, we use the workload traces originating from the IBM SP2 of the Cornell Theory Center. The data were compiled during the months July 1996 to May 1997. The original data include the following information:

- Number of nodes allocated to the job

– Time of job submission
– Time of job start
– Time of job completion
– Additional hardware requests of the job: Amount of memory, type of node, access to mass storage, type of adapter.

Further data like the job name, class (LoadLeveler class), type, and completion status are presently of no relevance to our simulation experiments.

For the simulation we need the node requirements, the submission time, and the execution time of a job. The execution time is determined by the difference between start and completion time of a job as nodes are assigned exclusively to a job in the batch partition. As stated in Section 2, the additional hardware request of a job is not used at the moment. But it can easily be considered in the simulator and may provide valuable information for choosing the best configuration of an MPP.

The CTC uses a batch partition with 430 nodes. As will be further explained in Section 6.1, in our simulation we assume batch partitions of 128 and 256 nodes respectively. Hence, jobs with an node allocation exceeding 128 or 256 nodes are ignored as we do not want to change individual job data. Table 1 shows that this only results in a small reduction in the total number of jobs.

	Total number of jobs	Jobs requiring at most 256 nodes		Jobs requiring at most 128 nodes	
Jul 96	7953	7933	99.75%	7897	99.30%
Aug 96	7302	7279	99.69%	7234	99.07%
Sep 96	6188	6180	99.87%	6106	98.67%
Oct 96	7288	7277	99.85%	7270	99.75%
Nov 96	7849	7841	99.90%	7816	99.58%
Dec 96	7900	7893	99.91%	7888	99.85%
Jan 97	7544	7538	99.92%	7506	99.50%
Feb 97	8188	8177	99.87%	8159	99.65%
Mar 97	6945	6933	99.83%	6909	99.48%
Apr 97	6118	6102	99.74%	6085	99.46%
May 97	5992	5984	99.87%	5962	99.50%

Table 1. Number of Jobs in the CTC Workload Data for Each Month (Submission Time)

4 The Scheduling Policy

If the amount of requested nodes for a job exceeds the nodes available on the parallel computer, a scheduling policy is used to settle those resource conflicts. This policy is supposed to reflect the objectives of the owner of the parallel computer as well as the wishes of the users. As jobs may be submitted at any time,

often some kind of First–come–first–serve (FCFS) strategy is used. Especially for parallel computers which do not support preemption and if no knowledge of future job arrival is available, various versions of FCFS strategy are frequently used. This also includes modifications like backfilling [17]. Also, an FCFS strategy reflects the notion of fairness which is probably the reason for being more or less accepted by most users. If several queues are used, FCFS may be applied separately to each queue or it may span several or even all queues. For instance, the CTC uses a modified FCFS approach for all batch queues with the exception of a queue for short running jobs (less than 15 minutes). This last queue receives a slightly higher priority.

A scheduling policy is evaluated by use of some scheduling criteria. In order to satisfy the user, his objectives must be reflected in those criteria. It can be safely assumed that a small job response time is desired for many jobs. However as already mentioned, it is typically not possible to execute each job immediately due to limited resources. In this situation most users would expect some kind of fairness policy which is a reason for the frequent use of FCFS. Of course, there may always be jobs which must be executed immediately or whose execution can be postponed to the night or the weekend. In return those jobs should be associated with some kind of either higher or lower costs. Also, a part of the parallel computer may be reserved exclusively for a specific group of jobs [13].

In the theoretical scheduling community the criteria most frequently used in this context are either makespan (=completion time of the last job in all queues) or the sum of the completion or flow/response times (= completion time - submission time) of all jobs [9]. The makespan is closely related to the utilization throughput and represents an owner centric point of view [8]. Adding the completion or flow time corresponds to a consideration of the individual user criteria 'minimization of the turnaround time'. Note that the completion time is of very limited interest in a practical setting where individual jobs are not submitted at the same time. Further, it must be emphasized that addressing only the makespan or only the flow time criterion is often not good enough, as an optimal makespan will not necessarily guarantee a minimal or even small sum of flow times and vice versa [18].

Apart from these general criteria, there may be some specific priority criteria based upon the intentions of the owner of a parallel computer. The owner may wish to prioritize highly parallel jobs over mostly sequential jobs or the other way around (throughput maximization) [9]. Also, some users may receive a larger allocation of compute time while others are only allowed to use selected time slots. Altogether, it is possible to think of a large number of different strategies. Some of these strategies can be implemented by using weights in connection with each job resulting in a weighted flow time criterion. In this case, the priority strategy must still be linked with the necessary selection of job weights. Hotovy [13] pointed out that in a real machine environment the selection of a priority strategy and its implementation is a complex and iterative process. But the overall goal of the policy established by the CTC is never explicitly stated and the final strategy is the result of several experiments which were evaluated

with respect to backlog and average waiting time. Lifka [17] addressed a similar problem by describing the properties of his *universal scheduler*.

In our simulation, we use the following three criteria:

1. Makespan of the whole schedule
2. Sum of flow times over all jobs
3. Sum of weighted flow times over all jobs with a special selection of the weights

Note that the simple adding of flow times (second criterion) does not consider the resource use of a job. In order to minimize the sum of flow times it is preferable to immediately schedule a large number of jobs requiring few nodes and running only for a short time and postpone all highly parallel jobs which require a long time to execute. Thus, this criterion indirectly prioritizes small jobs over others. While it may also increase throughput, the application of this criterion may contradict the purpose of a parallel computer to run long and highly parallel jobs. Moreover, it is beneficial for a user to split a parallel job into many sequential jobs even if this results in increased resource usage.

Therefore, we suggest to consider the objectives fairness and resource consumption by selecting weights appropriately. In particular, we define the weight of a job by the product of the execution time of the job and the nodes required by the job. This weight can also be used as the cost of the job for the purpose of accounting. This way there is no gain in splitting a parallel job into many sequential jobs or in combining many independent short jobs into one long job or vice versa. Only the overhead of a parallel job in comparison to its sequential version must be paid for. Using this type of weight (cost) for the purpose of accounting has the additional advantage that the user is only charged for the actual resource consumption instead of the projected value. This is especially important for long running jobs which fail immediately due to the lack of an input file or a similar cause. As the execution time of a job is not certainly known before the termination of the job, the actual weight is not available at the start of a job. Therefore, the scheduling algorithm cannot take the actual weight of a job into account when making its decisions. Only the maximum weight is provided due to the time limit of a job. A theoretical analysis of a scheduler based on this kind of weight selection is given in [19].

5 The Algorithm

We already mentioned the close relation between FCFS and fairness in the previous section. Note however that while FCFS guarantees that jobs are started in the order of their arrivals the same property does not necessarily hold for the completion of the jobs due to differences in job execution times. A fair schedule strategy can also be defined as a strategy where the completion time of any job does not depend on any other job submitted after it. Lifka's backfilling [17] method tries to obey this kind of fairness principle as only those jobs are moved up in the queues which are supposed not to affect the execution time of any job submitted before them. However, if job execution times are not known, this

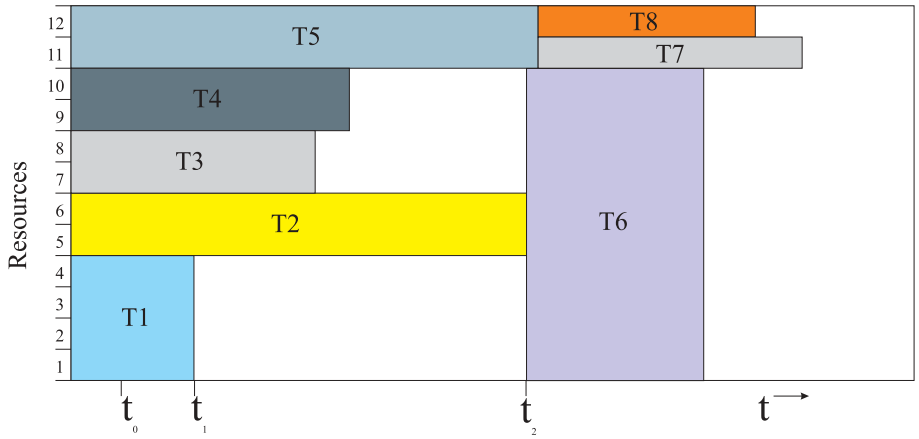

Fig. 1. Example for Fragmentation in FCFS by Wide Jobs

notion of fairness can only be preserved by aborting those backfilled jobs which would delay the start of a job submitted before then. Also, a preemptive schedule will not be fair in general even if the jobs are started in FCFS order. In this paper we therefore use a more general fairness notion called κ–fairness which has been introduced in [19]:

A scheduling strategy is κ-fair if all jobs submitted after a job i cannot increase the flow time of i by more than a factor κ.

Note that the term κ-fairness is not closely related to other schedule criteria. Therefore, we are looking for a scheduling strategy, which produces schedules with small deviations for makespan, sum of (weighted) flow times, and κ from the optimal values.

Since it is much harder to approximate the optimal total weighted flow time than the optimal total weighted completion time, as shown by Leonardi and Raz [16], many theoretical researchers focused on the completion time problem. This may be justified as both measures differ from each other by only a constant. Therefore, it is sufficient to consider just one of both criteria for the purpose of comparing two schedules.

For the sake of completeness we now repeat a few theoretical results which were already presented in other publications.

Using a pure FCFS schedule will obviously minimize κ. If all jobs are sequential it will also guarantee small deviations from the optimum values provided our weight selection is used with respect to the total weighted completion time. Here we denote by h_i and w_i the execution time and the weight of job i respectively. The following theorem can be derived from [15, 12, 10] and has been stated in this form in [19]. The makespan and the total completion time of a schedule S are described by m_S and c_S, respectively.

Theorem 1. *For all FCFS schedules S with only sequential jobs i and $w_i = h_i$ the following properties hold*

1. $m_S < 2 \cdot m_{opt}$,
2. $c_S < 1.21 \cdot c_{opt}$, *and*
3. S *is* 1*-fair.*

For the parallel case, it is usually assumed that FCFS scheduling may possibly generate big deviations for the makespan and the sum of flow times from the optimal values because fragmentation may occur and lead to a large number of idle nodes. The amount of fragmentation grows if jobs require large subsets of nodes, see Fig. 1. In the displayed example, job T6 cannot be started before time t_2 although it became the first job of the FCFS queue at time t_0. However, if the maximum degree of parallelism is restricted, FCFS still guarantees small deviations from the optimal case for parallel job systems with limited resource (node) numbers. Here we use r_i to denote the required nodes. Further R is the maximum number of nodes. The following results are also presented in [19].

Theorem 2. *For all FCFS schedules with $r_i \leq \frac{R}{2}$ and $w_i = r_i \cdot h_i$ there is*

1. $m_S < 3 \cdot m_{opt}$,
2. $c_S < 2 \cdot c_{opt}$, *and*
3. S *is* 1*-fair.*

From Table 1 we can see that highly parallel jobs are only a very small part of the workload. This explains the acceptable performance of FCFS schedulers in commercial MPPs in contrast to the bad behavior predicted by theoretical worst case analysis [19]. However, if more jobs with a small degree of parallelism are moved from MPPs to SMP workstations, it can be assumed that FCFS performance will suffer more severely.

As the percentage of idle nodes caused by jobs that require only a few nodes is rather small, our new scheduling strategy, called preemptive FCFS (*PFCFS*), especially cares for the highly parallel jobs. While in FCFS scheduling idle nodes must stay idle until the next job in line is started (with the notable exception of backfilling), we allow highly parallel jobs to interrupt the execution of currently running jobs.

Intuitively, we can describe PFFS by using two different schedules for small and for highly parallel jobs, which are then partially interleaved.

It is shown in [19] that PFCFS generates good theoretical results for makespan and completion time.

Theorem 3. *If the execution times of all jobs are unknown and $w_i = r_i \cdot h_i$ holds for each job and $n \to \infty$ and $\delta \geq \Delta$, then for all schedules S produced by the PFCFS Algorithm there is*

1. $m_S < (4 + 2\bar{p})m_{opt}$,
2. $c_S < (3.56 + 3.24\bar{p})c_{opt}$, *and*

3. S is $(2 + 2\bar{p})$-fair.

For this concept to be applied in practice, theoretical analysis is not sufficient. Therefore, parameters must be chosen to fine tune the strategy. In particular our new strategy and its parameters can be described as follows:

1. Any job requiring less than $x\%$ of the total number of nodes in the batch partition (in the following called *small job*) is scheduled in an FCFS fashion.
2. Any job requiring at least $x\%$ of the nodes (*wide job*) will preempt other running jobs under the following conditions:
   (a) The wide job is the next job to be started in FCFS order.
   (b) No other previously submitted wide job is still active.
   (c) A time span of δ seconds has passed since the moment the previous two conditions became valid.
3. If a wide job has been determined to start in a preemptive fashion a suitable subset of nodes (*preemptive subset*) is selected.
4. Once the machine is in a preemptive mode caused by a wide job, there is a context switch in the subset every Δ seconds up to a maximum of n context switches.

Note that not all currently running nodes must be preempted by a wide job. A wide job only preempts enough running jobs to generate a sufficiently large preemptive subset. Those small jobs are selected by a simple greedy strategy to preempt as few jobs as possible. The strategy also tries to adjust the size of the preemptive subset as much as possible to the requested node number. If n is odd, the wide job will be run to completion after the maximum number of context switches is exhausted. Otherwise those small jobs which are allocated to nodes of the preemptive subset all complete before the wide job if the latter has not finished at the last permitted context switch. Whether the size of the preemptive subset may change during the preemptive mode depends on the implementation. Altogether, the strategy is characterized by the selection of the four parameters x, δ, Δ, and n. It is the task of our simulation experiments to determine the best parameter set.

Fig. 2 gives an example for such a schedule. The system is executing jobs T1 - T5 with wide job T6 being due to execution at time t_0. That means T_6 is the next job to be scheduled in the FCFS queue. After a time delay $\delta = t_1 - t_0$, a sufficient large resource set to execute T6 is selected. The delay is introduced to prevent a possible starvation of smaller jobs if many wide jobs arrive in succession.

At this time t_1, all small jobs in the selected preemptive subset are interrupted. In the example, wide job T6 requires 9 resources for which jobs T1 to T4 must be preempted. After the preemption the wide job is started in the now available node subset. The wide job runs for a parameterized amount of time $\Delta = t_2 - t_1$. Next, we preempt the wide job and resume the execution of the previously preempted jobs (t_2). This cycle between the wide job and the small job gang will be repeated until a gang has been completed. In the example, T6 completes at t_4 after which the remaining jobs T1 and T3 of the smaller gang are immediately continued.

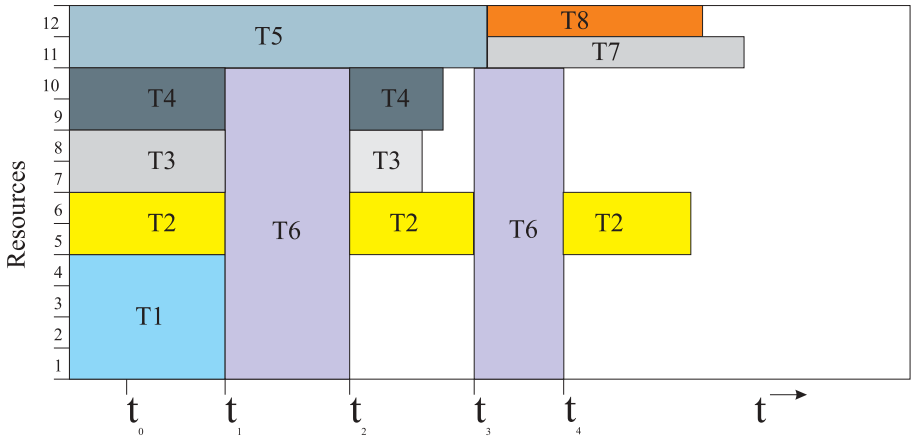

Fig. 2. Example for Gang Scheduling of a Wide Job (T6) without Migration

Intuitively, a schedule produced by rules outlined above can be described as the interleaving of two non preemptive FCFS schedules, where one of the schedules contains at most one wide job at any time instant. Note again that only a wide job can cause preemption and therefore increase the completion time of a previously submitted job. However, as the subset of a job is invariably determined at the starting time, it is also possible that the preemptive mode will actually increase the completion time of the wide job causing the preemption.

6 Simulation

6.1 Description of the Simulation

For obtaining information on the influence of the parameters and to determine good parameter values, various simulations have been done. For each strategy and each month we determined the makespan, the total flow time, and the total weighted flow time using our weight selection and compared these values with the results of a simple non-preemptive FCFS schedule. To compare the algorithm performance with a non-FCFS strategy, we made simulations with a backfilling scheduling policy.

Further, we examined variations on the specification of wide jobs. We also simulated with different gang lengths x and start delays δ before the start of the preemption.

The work traces of the Cornell Theory Center reflect a batch partition comprising 430 nodes. There are only very few jobs in these traces that use between 215 ($\frac{R}{2}$) and 430 nodes, as seen in Table 1. A reason is the tendency to use a number of processors which is equal to a power of 2. Taking into account that a noticeable gain by PFCFS can only be expected for a suitable number of wide

x Wide job spec.	n Maximum preemptions	Δ Start Delay	δ Gang Length
40%	1	60 sec	1 sec
45%	2	120 sec	60 sec
50%	3	240 sec	Δ
55%	10	600 sec	
60%	11	1800 sec	
		3600 sec	

Table 2. Different Parameter Settings for Simulation Experiments

jobs, we assume parallel computers with 128, respectively 256 resource. This is also reasonable as most installations only have a smaller number of nodes, and it can be assumed that there the percentage of wide jobs will be larger than for the CTC. This approach prevents the direct comparison of the simulation results with the original schedule data of the CTC.

In order to evaluate our scheduling strategy we assume a homogeneous MPP, that is a computer with identical nodes. As already mentioned, the batch partition of the CTC SP2 consists mainly of thin nodes with 128 or 256 MB memory. Therefore, our reference computer is almost homogeneous in this respect. Special hardware requests of the jobs are currently ignored. Although it is easy to consider those requests in the scheduler, it is not helpful in determining basic properties of the new strategy. Those requests can be taken into account in future studies.

As the CTC schedule itself is not suitable for comparison due to these deviations, we use a simple FCFS schedule for reference in this analysis.

Table 2 shows the parameter spectrum of our simulations.

The theoretical analysis [19] suggests $x = 50\%$. We chose several values around 50% to determine the sensitivity of the schedule quality on this parameter. While good theoretical results require a potentially unlimited number of preemptions we wanted to determine whether in practice a restriction of this parameter is sufficient. The time period between two context switches Δ is selected to span a wide range. However, to prevent a significant affect of the preemption penalty a minimum value of 60 seconds was used. The theory also suggests to set $\delta = \Delta$. We additionally used two fixed values for δ.

Finally, we ignored any preemption penalty in our simulations although it can easily be included into the simulator.

Moreover, as migration is not needed and only local hard disks are used for saving of job data and job status, the strategy allows gang scheduling implementations with a relatively small preemption penalty. Note that a preemption penalty of 1 second as assumed in [21] will still be small compared to the minimal value of Δ.

Finally, our experiments were conducted for each month separately to determine the variance of our results. This is another reason why the CTC schedule

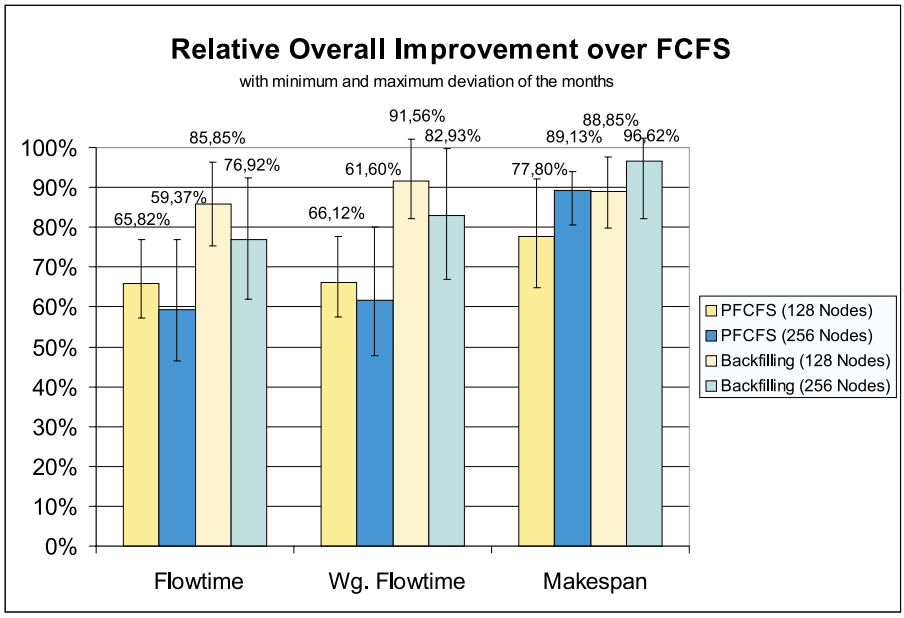

Fig. 3. Comparison of PFCFS with FCFS for Different Criteria

Resources	128	256
Wide job spec. x	40%	45%
Max. preemptions n	1	1
Start delay δ	60 sec	60 sec
Gang length Δ	not applicable	

Table 3. Parameter Setting for the Results Shown in Fig. 3

data could not be used for comparison as in the original schedule there are jobs which were submitted in one month and executed in another.

6.2 Analysis of the Results

Fig. 3 shows that the preemptive FCFS strategy was able to improve FCFS for all scheduling criteria if the best parameter setting was used, see Table 3. The preemptive FCFS strategy also outperformed the backfilling scheduling policy. This algorithm does not utilize preemption, but requires the user to provide a maximum execution time for a each job.

However, it can be noticed that the effect on the flow or weighted flow time is more pronounced for a 256 node parallel computer while the smaller 128 node machine achieves a better result for the makespan criteria. It can also be seen that

in every month a noticeable gain can be achieved with this set of parameters although the actual gains depend on the workload. Note that in the selected setting there is no time period between two context switches as with $n = 1$ a wide job will interrupt some small jobs and then run to completion. A similar approach was also subject of another publication ([1]) in a different context with dynamic jobs.

Also, there is only very little difference between the weighted flow time and the flow time results. This can be attributed to the large amount of short running jobs which do not require a large number of nodes. The stronger emphasis of larger highly parallel jobs in weighted flow time criterion does not have a significant impact on the evaluation of the strategy. However, this can be expected to change if the number of those jobs would increase.

Next, we examine the result for different limits on the number of preemptions. The results in Fig. 4 and Fig. 5 show that there are always improvements for odd values of n. Note that an odd n gives preference to the execution of the wide job once the maximum number of preemptions is reached. However, the gains change only little if the number of preemptions is increased. This indicates that the desired improvement of the scheduling costs can already be obtained by using a simple preemption for a wide job. On the other hand, even values of n result in significantly worse schedules in comparison to FCFS. This is due to the fact that the completion time of the wide job may actually be further increased over the FCFS schedule as its node allocation is fixed at the start time of the job and no job migration is allowed. This may then lead to a larger degree of fragmentation.

The schedule quality is not as sensitive on the other parameters. Theoretical studies [19] indicate that x=50% would be the best choice to separate wide jobs from small ones. Fig. 6 and Fig. 7 show that the schedule cost increases for all three criteria if x is selected to be larger than 50%. For real workloads a further slight improvement can be obtained by selected values for x which are less than 50%.

Δ is irrelevant if n is selected to be 1. For all other odd values of n the schedule cost varies only little with Δ. There, $\Delta = 120$ sec is frequently the best choice. If n is even, the results improve with a larger Δ as the chances to reach the preemption limit become smaller.

As the choice of Δ has little influence on the schedule quality, it is sufficient to just consider the cases δ= 1 sec and δ=60 sec. From theory we would expect a better makespan for δ= 1 sec as fragmentation will be reduced if wide jobs are executed as early as possible. On the other hand, δ= 60 sec will permit some short running small jobs to complete before they are interrupted by a wide job. However, in our simulation, switching from 1 sec to 60 sec changed the schedule costs by less than 0.01%. Therefore, δ can also be ignored.

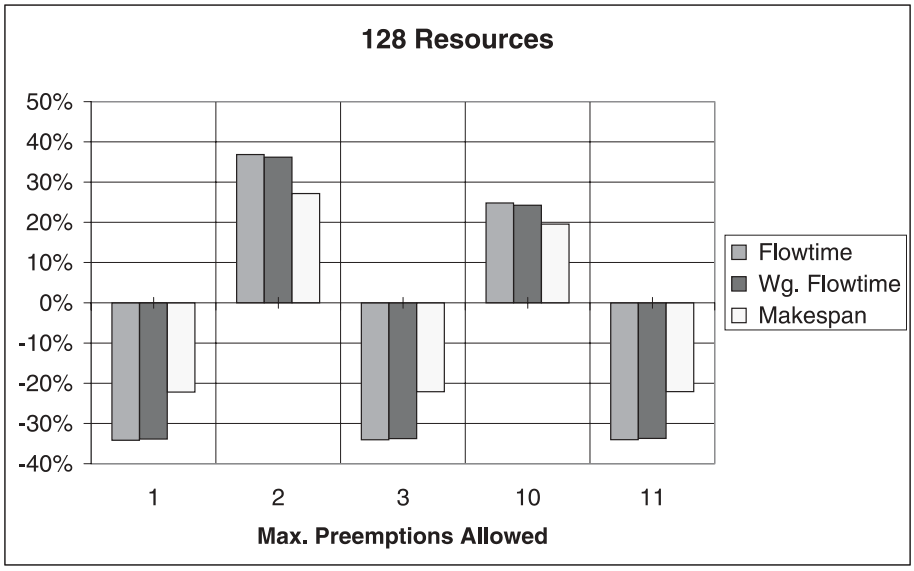

Fig. 4. Results for Different Limits on the Number of Preemptions for the 128 Node Machine

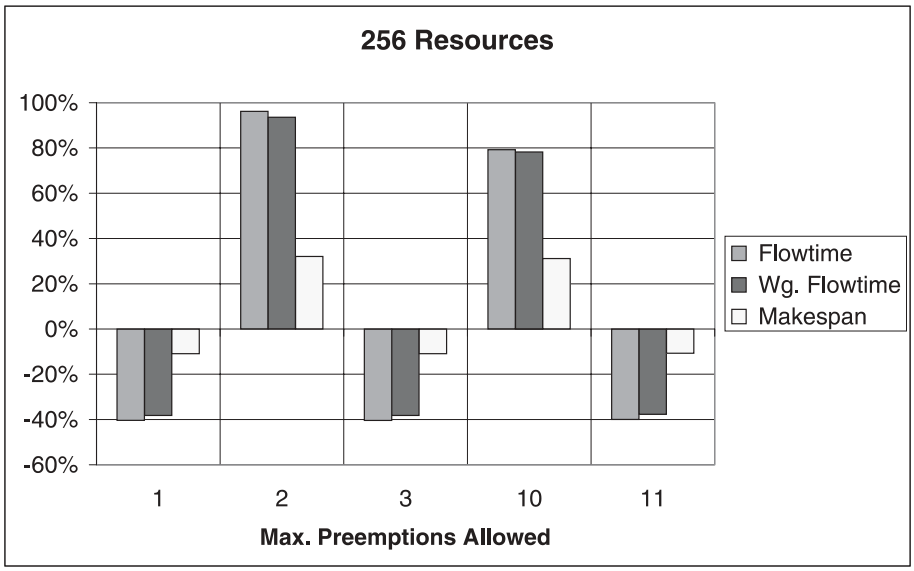

Fig. 5. Results for Different Limits on the Number of Preemptions for the 256 Node Machine

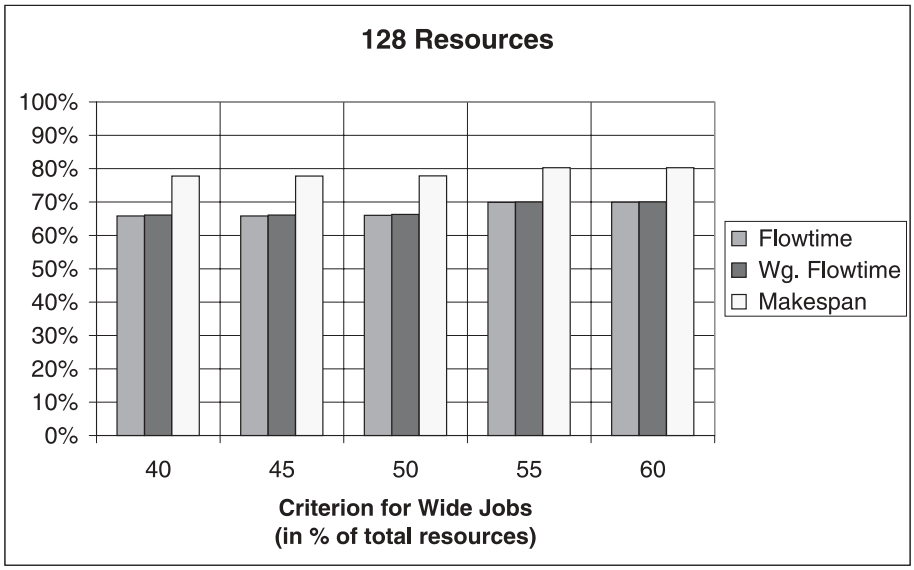

Fig. 6. Results for Different Characterizations of Wide Jobs for the 128 Node Machine

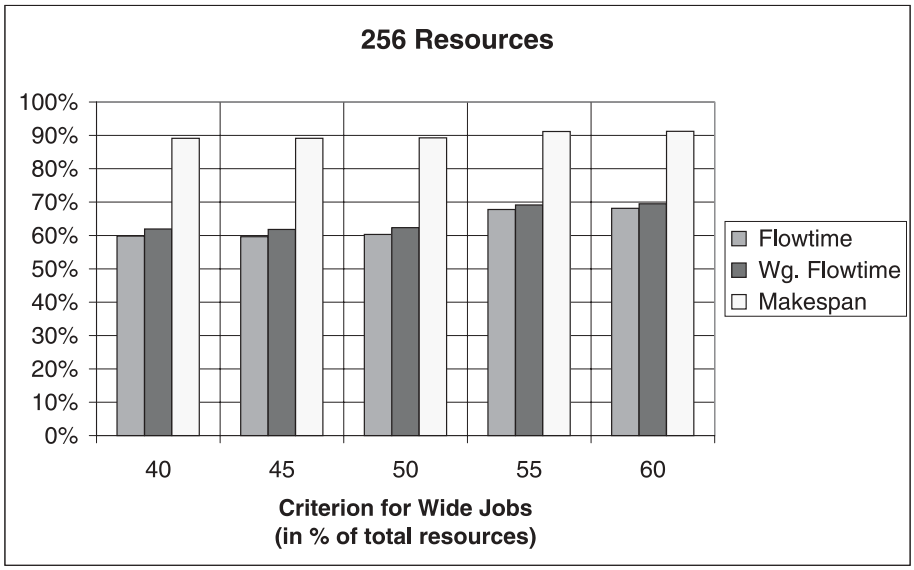

Fig. 7. Results for Different Characterizations of Wide Jobs for the 256 Node Machine

7 Conclusion

In this paper we presented a new scheduling method called PFCFS for batch jobs on MPPs. This strategy uses gang scheduling and is able to produce significant improvements of up to 40% in flow time and 22% in makespan for real workload data over FCFS.

It has been shown that the scheduling algorithm is sensitive to some of the scheduling parameters and should be carefully adapted for the specific workload. Nevertheless, it is possible to apply default settings that seem to increase schedule quality compared to FCFS. For instance, it may be sufficient to start execution of a wide job by preempting a suitable node set and let it run to completion. Based on our simulation this is sufficient to significantly improve the flow time and increase the machine utilization. For better results, we suggest that the parameters are determined on-line in an adaptive fashion from the workload results of previous months.

The implementation of the scheduling algorithm can also be improved, e.g. by allowing the start of new small jobs during gang scheduling. It may also be beneficial to check that the preemption cycle is only initiated, if more resources are utilized after the start of the wide job than before. Both modifications are not used in the presented simulations.

The PFCFS strategy is especially interesting since it has only limited requirements on the preemption support of the MPP. It uses no migration and at any time instance at most 2 jobs are resident on any node. Therefore, it can be assumed that the algorithm can be implemented on most systems with preemption capabilities. Also, there is no need for users to provide additional job information like an estimate of the execution time.

Acknowledgment The authors are especially grateful to Steve Hotovy for providing them with the workload data of the CTC SP2.

References

1. S.-H. Chiang, R.K. Masharamani, and M.K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of ACM SIGMETRICS Conference on Measurement of Computer Systems*, pages 33–44, 1994.
2. D.G. Feitelson. Packing schemes for gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.
3. D.G. Feitelson and M.A. Jette. Improved utilization and responsiveness with gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.
4. D.G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 337–360. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

5. D.G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain parallelization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
6. D.G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.
7. D.G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35:18–34, 1996.
8. D.G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.
9. D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.
10. A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130:49–72, 1994.
11. H. Franke, P. Pattnaik, and L. Rudolph. Gang scheduling for highly efficient distributed multiprocessor systems. In *Proceedings of the 6^{th} Symp. on the Frontiers of Massively Parallel Computation*, pages 1–9, 1996.
12. M. Garey and R.L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, June 1975.
13. S. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.
14. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. Modeling of workload in MPPs. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 94–116. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.
15. T. Kawaguchi and S. Kyan. Worst case bound of an LRF schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15(4):1119–1129, November 1986.
16. S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29^{th} ACM Symposium on the Theory of Computing*, pages 110–119, May 1997.
17. D.A. Lifka. The ANL/IBM SP scheduling system. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.
18. U. Schwiegelshohn. Preemptive weighted completion time scheduling of parallel jobs. In *Proceedings of the 4^{th} Annual European Symposium on Algorithms (ESA96)*, pages 39–51. Springer–Verlag Lecture Notes in Computer Science LNCS 1136, September 1996.
19. Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of First-Come-First-Serve Parallel Job Scheduling. In *Proceedings of the 9^{th} SIAM Symposium on Discrete Algorithms*, pages 629–638, January 1998.

20. F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M.S. Squillante. A gang scheduling design for multiprogrammed parallel computing environments. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 111–125. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.
21. F. Wang, M. Papaefthymiou, and M.S. Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 277–298. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.