Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling

David Talby Dror G. Feitelson
Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel

Abstract

Distributed memory parallel systems such as the IBM SP2 execute jobs using variable partitioning. Scheduling jobs in FCFS order leads to severe fragmentation and utilization loss, which lead to the development of backfilling scheudlers such as EASY. This paper presents a backfilling scheduler that improves EAST in two ways: It supports both user selected and administrative priorities, and guarantees a bounded wait time for all jobs. The scheduler gives each waiting job a slack, which determines how long it may have to wait before running: `important' and `heavy' jobs will have little slack in comparison with others. Experimental results show that the priority scheduler reduces the average wait time by about 15% relative to EASY in an equal priorities scenario, and is responsive to differential priorities as well.

1 Introduction

Most currently available distributed memory parallel supercomputers require users to request a specific number of processors for a job they wish to run. When the requested number of processors becomes available, the job is executed, and the processors are dedicated to it until it terminates or is killed. This scheme is called variable partitioning [3]. Allocating partitions on a FCFS basis results in severe fragmentation, and typical utilization of such systems is 50-80% [6, 7, 9, 12]. Two solutions that have proposed to this problem, dynamic partitioning [11, 1] and gang scheduling [4] are difficult to implement and do not enjoy much use.

A far simpler approach is to use a non-FCFS policy when allocating partitions, for example by allowing small jobs from the back of the queue to be executed while a large job is waiting for enough processors to be freed. Such an approach is called *backfilling* [8, 2]. The EASY scheduler [10], now part of IBM's LoadLeveler, uses an aggresive strategy that backfills a job if it does not delay the *first* job in the queue. We recently showed that a more conservative scheduler, which only backfills a jobs that doesn't delay *any* job in the queue, retains roughly the same performance [5]. Here we present a further improvement based on the notion of *slack*: each job could be backfilled if it does not delay any other job by more than that job's slack.

Another consideration is priorities. Supercomputers are typically used by several groups and projects at once. The administrators may wish to give each of them a different priority, or enforce a CPU quota on groups, projects or users. Different users within a system may also wish to prioritize themselves, for example when nearing a deadline. Finally, the scheduler itself may wish to adjust a job's priority, by raising it if the job initially had to wait longer than the system's average wait time, or lowering it if the job has been 'lucky'. However, priorities do not exist in a vacuum, and must be integrated with other considerations such as the wish to maximize utilization, maintain execution order, and ensure fairness. As these requirements may conflict with each other, we need to wieght their influence on the schedule.

The notation used in the following discussion is as follows. Jobs are denoted by $j_1, j_2,$ etc. A job may have the following attributes: n is the required number of processors, t is the user's estimate of the runtime, $up, pp,$ and sp are the job's user, political, and scheduler priority (each between 0 and 1), t_0 is the arrival time of the job to the queue, t_e is the time in which the job is scheduled to start executing (it may change several times before the job actually runs), p is the weighed priority of the job, s is the job's current slack, and s_0 is the job's initial slack. Slack will be measured in units of time. The weights of utilization, time, priorities and fairness are denoted $0 \leq \alpha_u, \alpha_t, \alpha_p, \alpha_f \leq 1$.

2 The Priority Scheduler

2.1 The Algorithm

The priority backfilling algorithm maintains a profile of scheduled jobs. When a new job is inserted, any other job may be rescheduled in order to optimize the overall utilization, subject to constraints of no preemption and execution guarantees. The algorithm gives a price to every possible new schedule, and chooses the cheapest one. The price of a schedule is the sum of prices of its jobs, and the price of each job is the product of its delay by the number of processors it uses. Formally, a scheduler is an event driven program that supports three events — insert a new job, remove a waiting job, or start or end jobs when reaching the end of a time slot:

insert(j): Loop over all possible schedules (conceptually) and give a price to every possible schedule. The price of scheduling j at $j.t_e$ and delaying $j_1, \ldots, j_k$ by $t_1, \ldots, t_k$

seconds is infinity if it causes one of the j_i to achieve a negative slack (if $j_i.s - t_i < 0$), and otherwise it is:

$$price = (j.t_e - CT)^{\alpha_t} j.n^{\alpha_u} + \sum_{i=1}^{k} cost(j_i, t_i, j)$$

The cost function determines the cost of delaying j_i by t_i seconds in favor of j, and will be discussed later. Cost will be negative if t_i is negative (i.e. if the job is being moved up rather than being delayed). Note that there is always at least one schedule of finite price: do what conservative backfilling would have done. Once we know what the cheapest schedule is, we adjust the slacks of the rescheduled jobs (for all $i = 1..k$, $j_i.s = j_i.s - t_i$), and start running jobs that should start now.

Remove(j): Remove j from the profile, and then loop over all possible schedules and find the cheapest one, exactly as in insert (remove the job, and insert a dummy timeless job). Here we expect a negative price — a profit.

Tick(): This is the same as in EASY and conservative backfilling: simply kill jobs that were supposed to terminate but didn't, and start executing jobs whose scheduled execution time is now.

The following two sections describe how the priority and slack of a new job are determined, and the $cost(j_i, t_i, j)$ function that prices reschedules of jobs. Section 2.4 addresses the complexity problem: since there is an exponential number of possible schedules of k jobs (which is $k!$), it is not practical to check each of them in a naive manner. The last section presents several alternative solutions to this problem, thereby completing the algorithm's specification.

2.2 Calculating Priority and Slack

The priority $j.p$ of a job is composed of its user, political and scheduler priorities. The user and political priorities are given when the job is submitted, but the scheduler priority is not. The scheduler priority is a number in the range [0,1], and we wish it to be higher when the job's initial wait time is longer.

At first, we assign $SP = \frac{1}{2}$ for all jobs. This means that we want all jobs to wait exactly the average wait time. Then, we calculate the job's initial priority:

$$j.p = \frac{j.UP + j.PP + j.SP}{3}$$

Afterwards, we calculate the job's initial slack:

$$j.s_0 = \begin{cases} (1 - j.p)\, SF \cdot AWT & \text{if } j.p > -\infty \\ \infty & \text{otherwise} \end{cases}$$

The constant SF is the slack factor of the system — it's another parameter of the algorithm. In section 2.1 we saw that giving jobs slack can improve utilization; however, giving jobs too much slack makes the upper bound on delays meaningless. The slack factor gives a way to express an opinion about this tradeoff. The case in which $j.p = -\infty$ occurs when the user exceeds his or her quota for one of the system's resources. The administrator then submits j with $j.PP = -\infty$ which causes the job to have infinite slack. This means that this job can suffer an unbounded delay, and will only run when it's not disturbing any other job. Once we have a priority and an initial slack for j, We can compute the price of each possible schedule. After deciding where it's best to execute the new job, its start time $j.t_e$ will be defined. Then, we recalculate:

$$j.SP = \min\left\{ \frac{j.t_e - NOW}{2AWT}, 1 \right\}$$

Note that $j.SP$ is zero if j doesn't wait at all, $\frac{1}{2}$ if it waits the average wait time, and 1 if it waits twice the AWT or more.

Once we have the new $j.SP$, we recalculate $j.p$ and $j.s_0$ according to this new $j.SP$ value, and use the new priority and slack values from now on. This recalculation takes place only once — we do not reschedule j after recalculating its priority and initial slack. These new values will only have an effect in case of future backfilling attempts.

2.3 Calculating Cost

The cost of moving j_i by t_i seconds in favor of j depends on the utilization gain or loss that the move causes, the relative priorities of the two jobs, and the percentage of j_i's slack that was already used. The preference of earlier jobs is contained in the fact that when two schedules are equally priced, we'll choose the schedule with the least number of moved jobs. Since delaying the new jobs doesn't `count' as a delay, older jobs have an advantage over the new one. All the above considerations are weighed by α_u, α_t, α_p, and α_f. To conclude we get:

$$cost(j_i, t_i, j) = \begin{cases} j_i.n^{\alpha_u} t_i^{\alpha_t} \left(\frac{j_i.p}{j.p}\right)^{\alpha_p} \left(\frac{j_i.s_0}{j_i.s}\right)^{\alpha_p \alpha_f} & \text{if } t_i \le j_i.s \\ \infty & \text{otherwise} \end{cases}$$

Note that the cost of delaying a job is greater when it uses more processors, the delay is longer, the job has a higher priority than the new job, or has already lost most of its initial slack. The global weights $\alpha_u, \alpha_t, \alpha_p, \alpha_f$ govern the relative importance of each factor.

2.4 Complexity Reduction Heuristics

Basically, the problem at hand is a scheduling problem of jobs of variable duration with no preemption, variable deadlines, a resource constraint on processors, and variable costs for delaying a job. Not surprisingly, this is a NP-hard problem.

The following exponential algorithm finds the optimal (cheapest) schedule for inserting a job j into a profile: for each time slot ts in the profile starting with NOW, delay all waiting jobs from ts to the profile's end by $j.t$, and insert j as the only job scheduled for time ts. Then, compress the schedule in every possible way, and remember the cheapest schedule. Note that when trying to assign the new job to a time slot we don't have to try to change jobs that are scheduled for earlier slots, since we assume that the schedule was optimal before the new job was submitted.

If k jobs were delayed in this manner, there may be $k!$ schedules to check, because every permutation of the delayed jobs defines an order in which the delayed jobs can be compressed, and each such permutation can create a different schedule with a different price. However, in many cases only few of these permutations will be worth checking. Consider, for example, a scenario in which ten jobs, all requiring all the processors in the system, are in the queue, and a new such job arrives. Clearly the best schedule is the one in which the more expensive jobs are executed first — that is, jobs are scheduled according to descending $cost(j_i, 1, Jnew)$. The suggested algorithm will still try $10!$ schedules, and as profiles in practice can be as long as a hundred jobs, it is impractical.

Several heuristics can be offered for choosing which permutations are checked. These heuristics choose one permutation to consider in each iteration — this approach still requires $O(n^3)$ time to insert or remove a job.

Ascending Scheduled Time (AST) — Sort the delayed jobs by their scheduled time (before the delay), and check this permutation only. This heuristic tries the most to preserve the current ordering of the delayed jobs.

Ascending Arrival Time (AAT) — Sort the delayed jobs by their $j.t_0$, and check this permutation only. This benefits jobs that are already waiting longer.

Descending Utilization (DU) — Check the permutation that results from sorting the delayed jobs by descending $j_i.n \cdot j_i.t$. This heuristic acknowledges that jobs with higher utilization are likely to be delayed more because of their size, and therefore they should be scheduled first.

Descending Cost (DC) — Sort the delayed jobs by descending cost to delay the jobs for one second, and check this permutation only. This way `expensive' jobs will be delayed less, and priorities and fair share issues will also be considered.

Descending Priority (DP) — Reschedule first the jobs whose priorities are highest. For the common case of equal priorities, a secondary sort by ascending arrival time was also used. This heuristic is expected to be more responsive to priorities.

	Average Wait Time		
Sept	440.6	440.6	0.0%
Oct	3062.7	2504.3	18.2%
Nov	4887.0	3774.6	22.8%
Dec	2157.2	1904.2	11.7%
Jan '97	3302.3	2531.9	23.3%
Feb	2624.3	2373.9	9.5%
March	2644.8	2196.9	16.9%
April	2220.3	1905.2	14.2%
May	2093.6	1779.2	15.0%
June	1443.2	1282.0	11.2%
July	399.9	354.1	11.5%
August	1941.5	1670.8	13.9%
Average	**2401.44**	**2004.46**	**16.5%**

Table 1. *Simulation results of priority backfilling compared to conservative backfilling.*

3 Experimental Results

3.1 The Simulator

A simulator for testing the conservative and priority schedulers was written in C++, implementing all aspects of both algorithms. Jobs that exceeded their declared runtime were immediately killed.

The logs used for the simulations were the Swedish Royal Institute of Technology (KTH) files from September 1996 to August 1997. The average number of jobs per month is 2357 and the average wait time of the system with 128 processors, under the conservative scheduler (which is nearly the same as the average wait time obtained by EASY scheduling) is 2401 seconds. The logs contain, for each job, the number of processors and both the estimated and actual runtimes of the job. Hence the tests do not require a model of the human ability to estimate runtimes, and use actual numbers.

3.2 Equal Priorities

The following simulations tested the performance of the priority scheduler against those of the conservative scheduler, assuming that all jobs have equal user and political priorities and an unlimited quota. The following parameters were used for the priority scheduler: $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$, SF = 3, AWT = 2401, Heuristic = AST.

The results in Table 1 show an average reduction of 16.5% in wait time over conservative scheduling (which has almost identical performance compared to that of the EASY scheduler). Except for September '96 in which only 85 jobs were executed, the algorithm performs well on a variety of workloads. Logs of other months included between 1924 and 4060 jobs. This implies that priority scheduling is useful as is, even without using the option to assign priorities. The overall average was calculated correctly — not as the

SF	1	3	5	7	9	11
AWT	2156	2004	2027	1986	1939	1956

Table 2. *Dependence of AWT on the slack factor.*

average of the rows above it, but as the quotient of the total wait time by the total number of jobs throughout the year.

The scheduler can be adjusted by several parameters: the slack factor, the average wait time, the heuristic used, and the weights given to each priority requirement. The effect of the first three is considered next, and the fourth is deffered until the next section.

The same simulation using larger slack factors modestly improved the results (19.25% average improvement for SF=9, see Table 2). However, giving too much slack to jobs increases the extent to which jobs can be starved — for example, a slack factor of nine means that a job can be delayed up to nine times the average wait time (see section 2.2). Therefore a small slack factor, which still gives significant improvements, seems like the best tradeoff.

The average wait time used for all months was the same — the yearly average. A good estimation of the AWT is important: using a very small value causes the algorithm to collapse to conservative scheduling (too little slack implies inability to delay jobs), and using a very large value increases the risk of (bounded) starvation. The yearly AWT is usually known to administrators or can be derived from logs, and presents no practical problem. Our attempts to improve and automate the AWT estimation were in vain: using a monthly instead of the yearly AWT separately for each month produced slightly worse results, and using a running average method to update the AWT dynamically caused significantly worse performance. Finding a better prophet of changing workload characteristic is a desirable future research direction.

The ascending scheduled time heuristic used to obtain the above results is the one that gives the best performance. Ascending arrival time gave an average improvement of 13% over conservative scheduling, descending priority gave an average improvement of 11.7%, descending cost gave 9.2% and descending utilization gave 8.1% (see table 5). Notably all heuristics improve conservative scheduling (and hence, EASY as well).

3.3 Differential Priorities

To test how well the algorithm schedules jobs with a higher priority then others, the following mechanism was used. For each of the twelve monthly logs, each fifth job was given a user priority of 1.0 and a political priority of 1.0, and all other jobs were given zero for both the user and political priorities. This simulates a scenario in which a user or a group have a considerably higher priority than others, and they submit jobs uniformly. It was implicitly assumed that the `fifth jobs' distribute identically compared

Month	Equal Pri	Unequal Priorities		
		All Jobs	Hi-Pri	Lo-Pri
Sept	440.6	440.6	0.0	550.8
Oct	2504.3	2783.7	2295.6	2906.0
Nov	3774.6	4797.7	4460.2	1882.3
Dec	1904.2	2026.0	1814.8	2078.8
Jan '97	2531.9	2777.4	2603.8	2820.8
Feb	2373.9	2728.7	2369.2	2818.7
March	2196.9	2220.9	1687.2	2354.6
April	1905.2	2056.2	1797.4	2120.9
May	1779.2	1889.3	1491.9	1988.7
June	1282.0	1374.7	1323.3	1387.6
July	354.1	372.6	320.2	385.8
August	1670.8	1847.1	1817.5	1854.5
Average	**2004.46**	**2226.17**	**1955.28**	**2294**

Table 3. *Simulation results of priority backfilling with differential priorities.*

gap	0	0.5	1	1.5	2
AWT	2004.46	2020.40	2044.39	2017.82	2226.17

Table 4. *Dependence of the overall AWT on the priority gap.*

to other jobs regarding wanted time, used time, processors and so on. The results, summed in table 3, were received using the same parameters as in the previous section: $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$, SF = 3, AWT = 2401, Heuristic = AST

The results show that the priority change decreased the average wait time of the preferred jobs by 2.5% but increased the wait time of the other jobs by 3.9%, compared to the best results achieved with equal priorities. In total, assigning different priorities had, as expected, a cost — the average wait time of the entire system rose by 11.1%, from 2004.5 to 2226.2 seconds. This is still an improvement over conservative scheduling and EASY, and the decision of whether to support priorities or maximize system performance is in the administrators' hands. In any case, the priority scheduler outperforms conservative scheduling and EASY.

The above simulations, as mentioned above, gave a random group of 20% of the jobs a user plus political priority of two against zero to the other jobs. Other simulations using a smaller gap exhibited a smaller, but still positive, difference in the average wait time between the groups. Table 4 shows the average wait time of the entire system as a function of the gap. With the exception of the maximal gap, the overall AWT was very close to the optimum (which is obtained with zero gap, e.g. in equal priorities). This means that in most cases the prioritization does not lead to a general degradation of service.

	Ascend Sched	Ascend Arrival	Descend Cost	Descend Util	Descend Pri
Equal Pri	2004.5	2088.5	2179.9	2206.0	2120.0
Unequal Pri	2226.2	2223.3	2250.1	2279.8	2228.5
Perf Loss[1]	11.1%	10.9%	12.3%	13.7%	11.2%
Hi-Pri Jobs	1955.3	1952.6	1988.1	2048.9	1962.3
Lo-Pri Jobs	2294.0	2291.1	2315.7	2337.6	2295.1
Gap[2]	338.7	338.5	327.6	288.7	332.7

[1] The ratio of increase in the total AWT due to the differential priorities, relative to the best result with equal priorities which is with AST.

[2] The average difference between the low priority group and the high priority group AWT.

Table 5. *Simulation results comparing various complexity reduction heuristics.*

Other simulations tested the effect of increasing the importance of priority in contrast with time, utilization and fairness, for example by assigning $\alpha_p = 1$, $\alpha_u = \alpha_t = \alpha_f = 0.2$. These tests indicated a slight increase in the wait time gap between the preferred and the regular groups, but also exhibited a considerable degradation of the average wait time of both groups. In several cases, the average wait time of the preferred group was worse than that of the $\alpha_u = \alpha_t = \alpha_p = \alpha_f = 1$ setting. It seems that a high α_u and α_t are crucial to the effectiveness of the scheduler.

The results that were achieved using the descending scheduled time heuristic were actually only the second best. Descending arrival time performed marginally better. Descending priority is also very close, hinting that this performance level is probably the best that can be expected. The results of all five heuristics are summarized in Table 5. They were all tested using the full set of logs. All numbers are yearly averages, in seconds.

4 Conclusions

The many production installations of EASY around the world prove that backfilling is advantageous over FCFS allocation of processors to jobs. The ability to backfill increases the overall system performance by being more responsive to short jobs, while preventing the starvation of long batch jobs. We have presented an algorithm that significantly outperforms EASY and conservative scheduling in simulations, which is based on the notion of slack. The priority scheduler also supports assigning differential priorities to jobs and is responsive to such requests, although a small penalty for prefering priorities over utilization is inevitable. The algorithm also includes a set of parameters to control its behavior, whose effects have been analyzed as well. Although backfilling was originally developed for the SP2, and was so far tested using workload traces from SP2 sites only, it is applicable to any other system using variable partitioning. This includes most distributed memory parallel systems in the market today.

Acknowledgements

This research was supported by the Ministry of Science and Technology. Thanks to Lars Malinowsky of KTH for his help with the workload traces. This trace is now available from the parallel workloads archive at URL http://www.cs.huji.ac.il/labs/parallel/workload/.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "*Scheduler activations: effective kernel support for the user-level management of parallelism*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 53–79, Feb 1992.

[2] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[3] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[4] D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[5] D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.

[6] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[7] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[8] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual*. Order number 312261-002, May 1992.

[9] P. Krueger, T-H. Lai, and V. A. Dixit-Radiya, "*Job scheduling is more important than processor allocation for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **5(5)**, pp. 488–497, May 1994.

[10] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[11] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[12] P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.