Packing Schemes for Gang Scheduling

Dror G. Feitelson

Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel
feit@cs.huji.ac.il — http://www.cs.huji.ac.il/~feit

Abstract. Jobs that do not require all processors in the system can be packed together for gang scheduling. We examine accounting traces from several parallel computers to show that indeed many jobs have small sizes and can be packed together. We then formulate a number of such packing algorithms, and evaluate their effectiveness using simulations based on our workload study. The results are that two algorithms are the best: either perform the mapping based on a buddy system of processors, or use migration to re-map the jobs more tightly whenever a job arrives or terminates. Other approaches, such as mapping to the least loaded PEs, proved to be counterproductive. The buddy system approach depends on the capability to gang-schedule jobs in multiple slots, if there is space. The migration algorithm is more robust, but is expected to suffer greatly due to the overhead of the migration itself. In either case fragmentation is not an issue, and utilization may top 90% with sufficiently high loads.

1 Introduction

Parallel supercomputers are increasingly being used in preference over the more traditional vector supercomputers. While some of these parallel supercomputers are dedicated to specific applications, a large number are also used as general purpose servers with large and diverse communities of users. As such they must provide convenient scheduling facilities that will handle the allocation of resources to different user jobs.

A large number of scheduling schemes have been proposed for parallel machines [7, 11]. One of these is gang scheduling, where all the threads of a parallel job are scheduled for simultaneous execution on distinct PEs [24]. If the total number of threads in all the jobs exceeds the number of PEs in the system, time slicing is used. However, the context switching is coordinated across the PEs, such that all the threads in a job are scheduled and de-scheduled at the same time. Gang scheduling is a prominent feature of the Connection Machine CM-5 system [28], and is available on the Intel Paragon [17], the Meiko CS-2, and multiprocessor SGI workstations [2]. It has also been used extensively in a home-grown system on a BBN Butterfly at Lawrence Livermore Labs [13], which has recently been ported to their new Cray T3D system.

The main drawback of using gang scheduling is the problem of fragmentation. Specifically, it may happen that a number of jobs are scheduled to run, and a few PEs are left over, but they are insufficient for any of the other queued jobs.

The severity of this problem depends to a large degree on the distribution of job sizes [12]. One solution, used in the CM-5, is to use all the PEs for each job, rather than allowing subsets to be used. In this paper, we investigate alternative solutions based on different schemes for packing the jobs together for scheduling. We show that it is possible to achieve significant improvements over a simple best-fit packing, using either a buddy system to control the mapping, or by migrating jobs so as to re-map them.

As noted above, the experienced fragmentation depends on the workload. Therefore an accurate workload model is essential in order to evaluate the effectiveness of the various packing schemes. To this end we have analyzed a number of accounting traces that include information about many thousands of jobs that have been executed on a number of parallel machines. It is felt that the resulting workload model is much more representative of real workloads than other models that have been used in the literature.

The rest of this paper is organized as follows. Section 2 describes the different packing schemes that we are proposing. Section 3 describes the workload analysis and model. Section 4 then describes the experimental results obtained when using the different packing schemes in conjunction with the workload model. The conclusions are presented in Section 5.

2 Packing Schemes

Our work is done within the framework of a gang scheduling system based on the matrix algorithm by Ousterhout [24]. This algorithm views scheduling space as a matrix, where rows represent time slots and columns represent PEs. Each job is allocated to a single row. If space permits, a number of jobs may be allocated to the same row. Gang scheduling is done by iteratively scheduling the jobs in one row after the other.

The question we wish to investigate is that of packing in this matrix. This includes three sub questions:

1. If multiple slots have enough capacity for a new job, which one should be chosen?
2. When should a new slot be opened?
3. If the chosen slot has more free PEs than required, which ones should be used?

The considerations involved are relatively simple. Relating to the second question, it is generally desirable to pack the jobs into the minimal number of slots possible, because the run fraction[1] for each job is equal to one over the number of used slots. We therefore only consider algorithms that do not open new slots unless there are no used slots with sufficient capacity (or, in one case, if the free processors are not organized as needed).

[1] The run fraction is defined as the fraction of wall-clock time that the job is actually running on the CPUs, as opposed to waiting in the run queue or elsewhere.

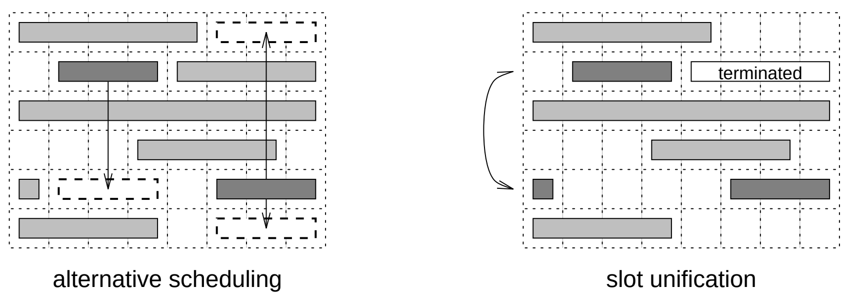


Fig. 1. *Packing should be done so as to promote alternative scheduling and facilitate slot unification.*

The other two questions can be tied together, by choosing the slot with the optimal choice of PEs. A judicious choice of PEs is important for two reasons (Fig. 1). First, choosing a set of PEs that are free in more than one slot may allow the job to be gang scheduled in multiple slots, thus increasing its run fraction and providing it with better service (this is called *alternative* scheduling). Second, if jobs are in general assigned to disjoint sets of PEs, then when a job terminates it may happen that the remaining jobs in its slot use PEs that are distinct from those used by the jobs in some other slot. This will make it possible to unite the two slots, thus reducing the number of used slots by one, and improving the run fraction of all jobs. Note that alternative scheduling and slot unification are features of the scheduler, and are independent of the packing scheme used. The point made is that better packing schemes will be able to make better use of these features.

A number of algorithms have been devised based on these considerations.

2.1 Capacity Based Algorithms

The first two algorithms just check the slot's capacity.

First Fit In this algorithm, the used slots are scanned in serial order. The first one with sufficient capacity is chosen. If no used slot has sufficient capacity, a new slot is opened. Within the chosen slot, free PEs are allocated in serial order.

Best Fit In this algorithm, the used slots are sorted according to their capacities. The one with the smallest capacity that is sufficient is chosen. If no used slot has sufficient capacity, a new slot is opened. Within the chosen slot, free PEs are allocated in serial order.

2.2 Left-Right Based Algorithms

The next two algorithms are modifications of the best fit algorithm, and modify the way that PEs are allocated within the chosen slot. The idea is to start from both sides, so as to reduce the overlap between sets of PEs assigned to different jobs.

Left-Right by Size In this algorithm, PEs are allocated either in serial order or in reverse serial order. The decision depends on the new job's size: for small jobs, the allocation is left-to-right, and for large jobs it is right-to-left. the threshold between small and large jobs should be near the median job size. In this study we assume 128 processors and use a threshold of 8, which reflects the fact that small jobs are much more common (see Section 3.1).

Left-Right by Slots In this algorithm PEs are again allocated either from the left or from the right, but here the decision depends on the slot. Slots are alternately designated as being filled from the left or from the right. All jobs mapped to a certain slot will therefore be allocated PEs in the same order. When a new slot is opened, its direction is designated so as to make the numbers of slots with the two directions as nearly equal as possible.

2.3 Load Based Algorithms

All the previous algorithms were oblivious of the loads on the different PEs. The next two take this new parameter into account. Again, the motivation is to reduce the overlap between sets of PEs assigned to different jobs.

Minimal Maximum Load The PEs are sorted according to the load on them, measured in jobs that use each PE. For each slot with sufficient capacity, the PEs that are free in that slot are considered. When allocating PEs to a job of n threads, the nth PE in the load order thus defines the maximal load on any PE that will be used in that slot. The slot with the minimal maximal load is chosen. Within that slot, the n least loaded free PEs are then used.

Minimal Average Load This algorithm is similar to the previous one, except that instead of using the load on the nth PE to prioritize the slots, we use the average load on the n least loaded PEs.

2.4 Buddy Based Algorithm

This algorithm is different in the sense that PEs are assigned in groups rather than individually. These groups are organized as a buddy system, based on concepts that were originally developed for memory allocation [18, 25], and following the PE allocation mechanism in the Distributed Hierarchical Control scheme (DHC) [9, 10].

Specifically, the PEs are partitioned recursively into groups that are powers of two. Logically, each group has a controller, thus creating a hierarchy of such controllers. When a job of size n arrives, it is assigned to a controller of size

$2^{\lceil \lg n \rceil}$ (i.e. the smallest power of 2 that is larger than or equal to n). The choice is done by scanning all the used slots, and identifying groups of $2^{\lceil \lg n \rceil}$ contiguous free processors that belong to the same controller. Controllers whose groups of PEs are all free in some slot are candidates for mapping the newly arrived job. If no controller is completely free in any used slot, a new slot is opened[2].

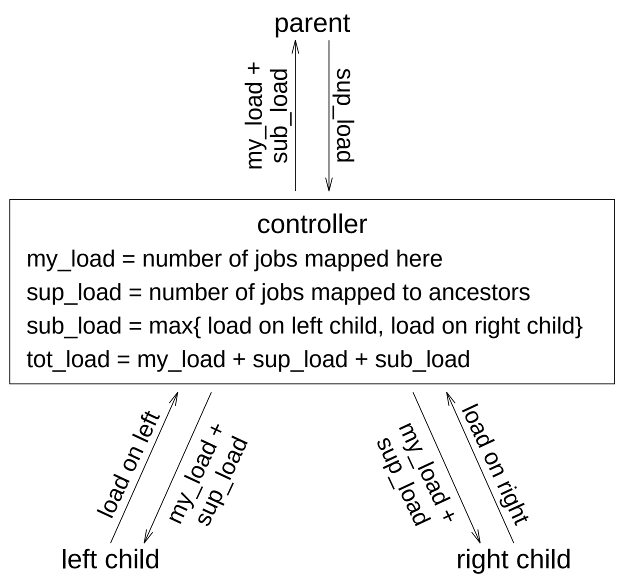

parent
my_load + sub_load
sup_load
controller
my_load = number of jobs mapped here
sup_load = number of jobs mapped to ancestors
sub_load = max{ load on left child, load on right child}
tot_load = my_load + sup_load + sub_load
load on left
my_load + sup_load
my_load + sup_load
load on right
left child
right child

Fig. 2. *Load calculation for controllers in the buddy system approach.*

Out of the free controllers, the one with the least load is chosen. Load on controllers is defined recursively according to the hierarchy as the sum of 3 terms: the number of jobs mapped to this controller, the maximum of the loads on the controller's two children, and the sum of the number of jobs mapped to the controllers ancestors. The data paths needed to compute loads using this scheme are illustrated in Fig. 2. A simplified scheme where load is simply the maximum of the loads on the PEs under the controller was also considered, but proved to be inferior.

If n is not a power of 2, only part of the PEs under the chosen controller are used. These are selected in groups that are powers of two, based on the loads on the controller's descendents. This scheme is equivalent to the "minimal fragmentation" scheme that was shown to be advantageous for DHC [9, 10]. The remaining PEs are not reserved or allocated in any sense, and can later be assigned to other (smaller) jobs. They can also be used to provide additional runtime to jobs that are mapped to other slots, via alternative scheduling.

[2] Note that in some cases this may lead to opening a new slot even if there are slots with n free PEs, because the n PEs are not all under the same controller.

2.5 Migration Based Algorithm

The final algorithm solves the problem of PE assignment in a completely different manner. Rather than seeking a good initial placement and then sticking to it, this algorithm migrates jobs from one set of PEs to another as needed in order to unite slots and improve run fractions. Specifically, our algorithm re-maps all jobs upon every job arrival and termination, using a first-fit-decreasing allocation to slots [4] (this algorithm is optimal if all job sizes divide each other, e.g. if they are powers of two [5]).

It is debatable whether this algorithm is realistic, because of the expected overhead, especially on distributed memory machines. It is true that systems that support migration have been implemented successfully [1, 6], but these systems do not attempt to perform migration at such a high rate. However, this algorithm is useful as a bound on the performance that is obtainable.

3 Workload Modeling

The most straightforward way to evaluate scheduling algorithms without a full scale implementation is through simulations. Naturally, the quality of the results depends on the quality of the inputs to the simulation. An important issue is the workload model. It has often been stated that there is no reliable information about workloads on parallel machine [23, 20, 19, 3]. However, this is in fact not true. Like uniprocessor systems, most parallel systems maintain administrative traces of all jobs run on the system. Analyzing these traces reveals a wealth of information about the workload.

For this study we used information derived from traces gathered on 6 different systems, all of which supported a real production workload. The traced systems are summarized in Table 1.

system	*trace*	*comments*
128-node iPSC/860 NASA Ames	42050 jobs, 4Q93 10821 parallel user jobs	Intel scheduler [16] analysis described in [8]
128-node IBM SP1 Argonne Natl Lab	19980 jobs, 12/94–6/95 15654 were parallel	home grown scheduler [21] submit trace, not run trace
400-node Paragon San-Diego SC	32500 jobs, 12/94–4/95 25867 were parallel	SDSC/Intel scheduler [29, 17]
126-node Butterfly LLNL	35848 jobs, 1991–1992 >30000 were parallel	home grown gang scheduler [14] no direct access to trace [13]
512-node IBM SP2 Cornell Theory Ctr	17947 jobs, 9/95–11/95 8598 were parallel	Scheduling by IBM LoadLeveler no direct access to trace [15]
96-node Paragon ETH Zürich	1723 jobs	Intel scheduler no direct access to trace [27]

Table 1. *Summary of systems and traces used in workload analysis.*

3.1 Distribution of Job Sizes

An important feature of the workload model is the distribution of job sizes, in terms of the number of nodes used by each job[3]. Histograms of the sizes observed in two of the traces are shown in Fig. 3. Examination of such histograms reveals three distinctive characteristics:

- Small jobs are more common than large ones.
- Some "interesting" sizes appear much more often than others, creating a distribution with pronounced discrete components.
- Practically all possible sizes up to about 100 nodes appear in practice, albeit a small number of times.

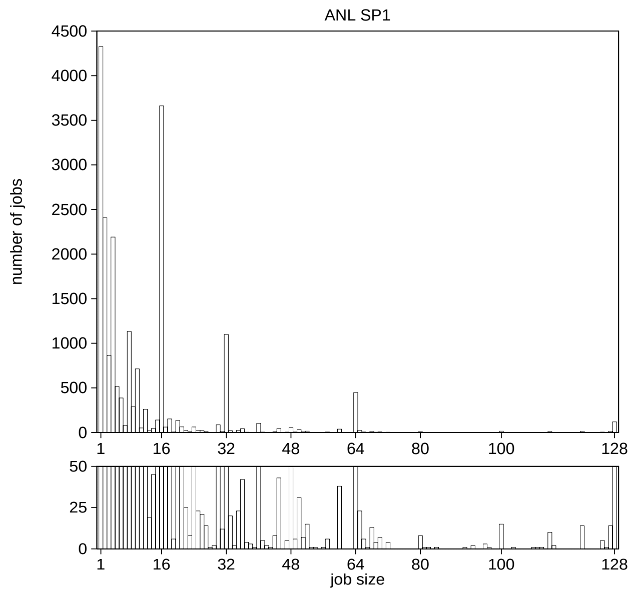
system	powers of 2	squares	multiples of 10	full system	sizes plus 1
NASA Ames iPSC/860	yes	(some)	no	(yes)	no
ANL/IBM SP1	yes	some	some	yes	no
SDSC Paragon	yes	some	no	no	no
LLNL Butterfly	yes	yes	some	yes	yes
Cornell SP2	yes	some	no	no	no
ETH Paragon	yes	no	no	yes	no

Table 2. *"Interesting" sizes in the different traces.*

The special sizes that appeared in the different traces are summarized in table 2. The most common one is jobs that use power-of-two nodes — this was a pronounced feature of all the traces. The reasons for using such sizes in preference over others are varied, and include algorithmic suitability (e.g. when using a divide-and-conquer paradigm) and system considerations (e.g. system administrators tend to create batch queues for power-of-two nodes, and system size is often a power of two).

A special case is jobs that require the full machine. In some cases, this is a power of two, but using the full machine was also popular in cases where this is not a power of two. For example, 12.2% of the jobs on the ETH Paragon used all 96 nodes, and these jobs used up 63.8% of the total resources (measured in CPU-seconds). On the LLNL Butterfly, using the whole machine was represented by jobs that used 112 or 113 nodes, which were typical sizes of the parallel cluster (the rest of the nodes were used for login, and did not participate in running parallel jobs). Notable exceptions are the SDSC Paragon and Cornell SP2. For example, the SDSC Paragon had only three 400-node jobs. This is because the system is heterogeneous: 256 nodes have 32MB of memory, and the rest only

[3] We assume that the number of nodes does not change during execution, as is the case in many systems that support an SPMD programming model.



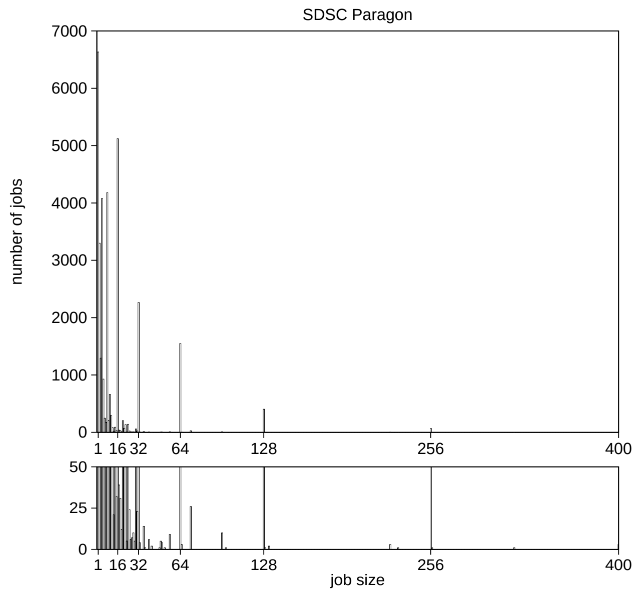


Fig. 3. *Histograms of job sizes on the ANL SP1 and SDSC Paragon.*

16MB, so for many jobs the effective maximum is 256. Also, 32 nodes were typically reserved for interactive use, so using all 400 nodes required turning off interactive use. The Cornell SP2 is also heterogeneous [15], and was never configured for using all 512 nodes.

Other popular sizes are squares (25, 49, 64, 81, 100), used when the algorithm is naturally expressed on a square array of processors (even if the architecture is not a mesh), and multiples of 10 (20, 50, 100), probably used mainly for aesthetic reasons when no other size was specifically warranted. Jobs using a master-workers paradigm sometimes created sizes that are larger by one than another popular size, e.g. $26 = (5 \times 5) + 1$. Of course, several sizes appear multiple times in these lists, and it is hard to know what interpretation to attach to them: 16 and 64 are both squares and powers of two, 50 is a multiple of 10 and one more than the square 49, and 100 is a square, a multiple of 10, and a nice round number.

Finally, in some cases arbitrary numbers seem to appear for no obvious reason. It is possible that this is a result of a specific preference by a single user, that uses a certain size for many repeated executions. Such behavior is discussed in Section 3.3 below.

3.2 Correlation of Runtime with Size

It is largely accepted that the runtimes of jobs in a computer system have a wide distribution, with many jobs that have a short runtime but a few jobs that have very long runtimes. This is typically modeled by a hyperexponential distribution. However, a-priori it was not clear whether or not there is a correlation between the runtime and the size (number of nodes) in parallel systems.

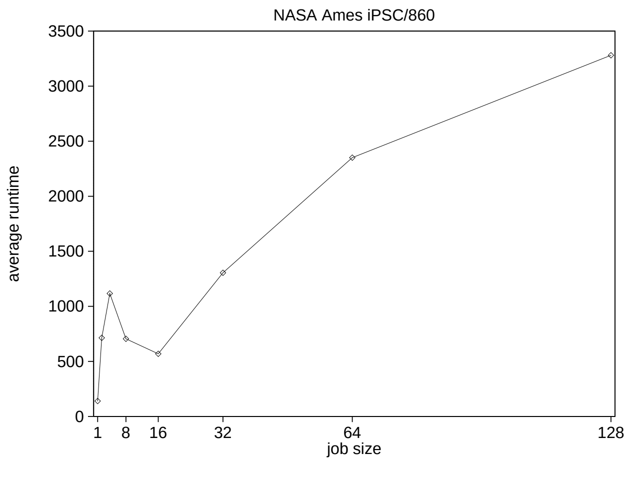


Fig. 4. *Runtimes vs. job sizes on the NASA Ames iPSC/860.*

Plotting the average runtime as a function of the size for the NASA Ames iPSC/860 trace produces the results shown in Fig. 4. There is an obvious correlation, with larger jobs running longer than smaller jobs.

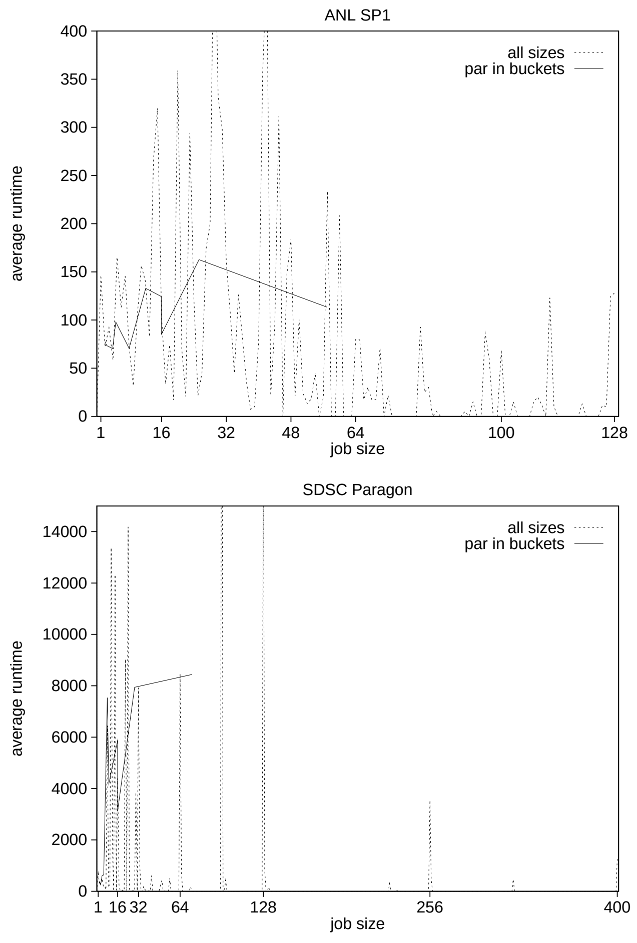


Fig. 5. *Runtimes vs. job sizes on the ANL SP1 and SDSC Paragon.*

For the other systems, plotting the average runtime as a function of the job size using data from the traces produces graphs with wildly varying shapes (see Fig. 5 for the ANL SP1 and SDSC Paragon). However, this is misleading, because different data points represent different numbers of jobs, and therefore should be give different weights. A more meaningful representation is obtained by dividing the jobs into 10 buckets according to size, and plotting the average runtime for each bucket. That is, each bucket contains a tenth of the total jobs, with the first one containing the smallest jobs, the next bucket containing the next larger jobs, and so on until the last bucket that contains the jobs using the largest number of PEs. In the plot, the representative size for each bucket is calculated as the average of the sizes of the jobs in the bucket. As there are many more small jobs than large jobs, the plots end at rather small sizes relative to the maximal size possible.

The results, using only parallel jobs, are also shown in Fig. 5. They indicate a weak tendency for larger jobs to have a higher runtime. However, it should be remembered that this is only a general trend, and the runtimes of specific jobs are widely distributed. Also, other studies have noticed differences between the distributions of jobs with "interesting" sizes and jobs that have other sizes, or between interactive and batch jobs. We intend to study such correlations further in the future.

3.3 User Modeling

An important issue in workload modeling is the question of whether jobs are independent of each other. The answer is that very often they are not. Specifically, users tend to submit sequences of similar jobs, one after the other.

In a preliminary effort to study this effect, the runlengths of such sequences were measured. In this context, a sequence is defined as the same user submitting the same job and using the same number of nodes. Results for the NASA Ames trace and the ANL SP1 trace (the two available traces that included user and job information) are shown in Fig. 6. It is seen that some sequences are extremely long (the maximum observed is 402 runs on the ANL SP1). The fact that the slope is a straight line in these log-log plots indicates a generalized Zipf distribution (i.e. $p(n) \propto 1/n^{\theta}$) [30, 26]. Using linear regression, the harmonic order (θ in the equation for the probability distribution) is around 2.2 for both cases, after deleting outliers that appear only a small number of times. Similar results were obtained for the Cornell trace [15].

3.4 Job Classes

In many cases jobs in a system can be classified into a number of classes, and such classification is often an explicit goal of workload analysis. In multiuser parallel systems an obvious classification is the distinction between interactive and batch jobs, as this distinction is supported directly by many systems: interactive jobs are those that are submitted directly and run immediately, while batch jobs are queued for later execution (often using NQS).

The significance of the class distinction is twofold. First, batch jobs tend to run longer than interactive ones. Second, batch queues are often enabled for execution only during the night, thus creating a daily cycle of completely different workloads at prime time and non-prime time. This effect is very pronounced in the NASA Ames trace [8], and can also be seen in the SDSC trace.

The reason to delay batch jobs to non-prime time is that in systems that use space slicing without preemption, the decision to run a batch job might block future requests to run interactive jobs. This consideration is eliminated in time

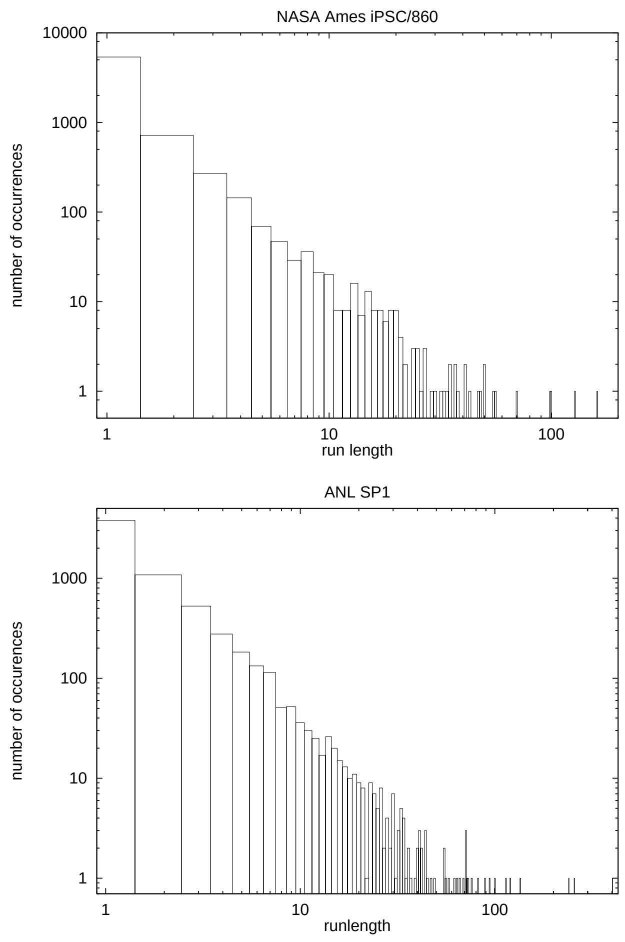


Fig. 6. *Distribution of runlengths of repeated executions on the NASA Ames iPSC/860 and the ANL SP1.*

slicing systems, because a batch job can share the processors with interactive jobs that come later. Moreover, interactive jobs can be demoted automatically to batch status if they run for too long. For example, this is done in the LLNL Butterfly (regrettably, there is no data on how often this actually happened). While such options are interesting, we leave them for future work, and ignore the distinction between batch and interactive jobs in the context of the current gang scheduling study.

3.5 The Model

Based on the above, we model the workload as follows. The distribution of sizes is based on a harmonic distribution, which is then hand tailored to emphasize

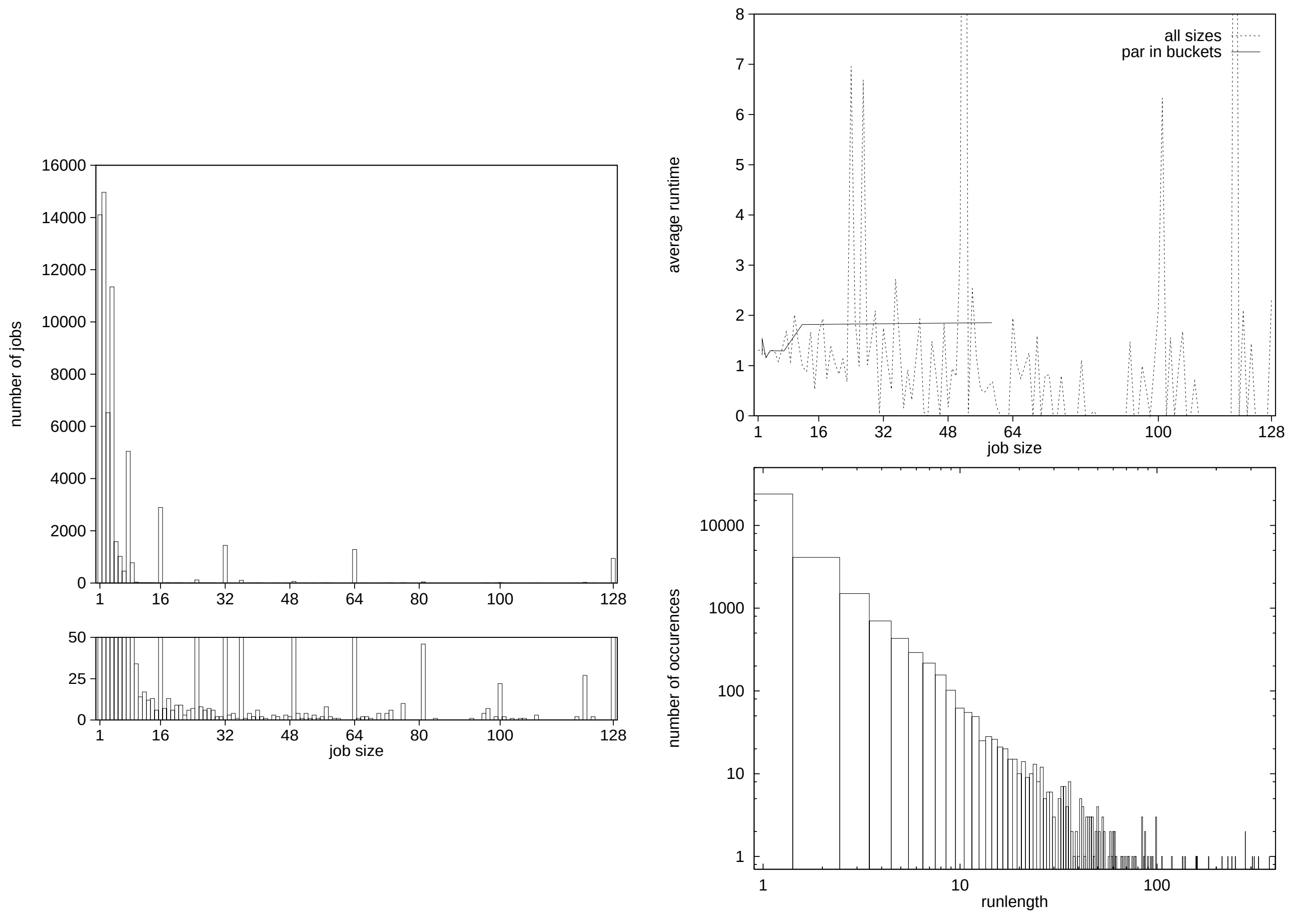


Fig. 7. *Histogram of job sizes, correlation of runtimes with job size, and histogram of run lengths for the workload model.*

small jobs and interesting sizes. This gives a qualitative approximation to the types of distribution observed in practice, as witnessed by the histogram shown in Fig. 7. The runtimes are distributed according to a two-stage hyperexponential, with a linear relation between the job size and the probability of using the distribution with the higher mean (so there is actually a different distribution for each job size, and the mean for larger jobs is at a higher value). Again, this provides a qualitatively good approximation. The runlengths are from a generalized Zipf distribution with a harmonic order of 2.5. The interarrival times are exponentially distributed.

In the future we plan to conduct a more thorough and quantitative analysis of the workload traces, taking more statistical properties into account. This will be used to create a more accurate workload model.

4 Experimental Results

The packing schemes described in Section 2 were compared by simulation, in which they were exercised by a workload model as described in Section 3.

4.1 Methodology

A single sequence of job arrivals was generated according to the model. This sequence was re-used for all data points and for all packing schemes, thus assuring that the comparison is fair in the sense that they all contend with the same workload.

Each data point represents the average of 30 experiments, each including 1000 job terminations. An additional initial experiment was discarded in order to account for simulation warmup. 95% confidence intervals were computed using the batch means approach [22].

The simulation itself is event-driven, where events are job arrival and termination. The average interarrival time is changed to simulate different load conditions. Between consecutive events, jobs are assigned constant run fractions according to the number of slots in which they can run. Overheads for context switching and for computing the packing are ignored.

The main performance metric is the slowdown experienced by jobs, and more specifically, the functional dependence of the slowdown on the system load. Slowdown is just the normalized response time, where the response time of each job in the loaded system is divided by its response time in an empty system (i.e. its actual computation time). Alternatively, it can be regarded as the reciprocal of the run fraction.

4.2 Comparison of Packing Schemes

The simulation results are shown in Fig. 8. They can be summarized by the following points:

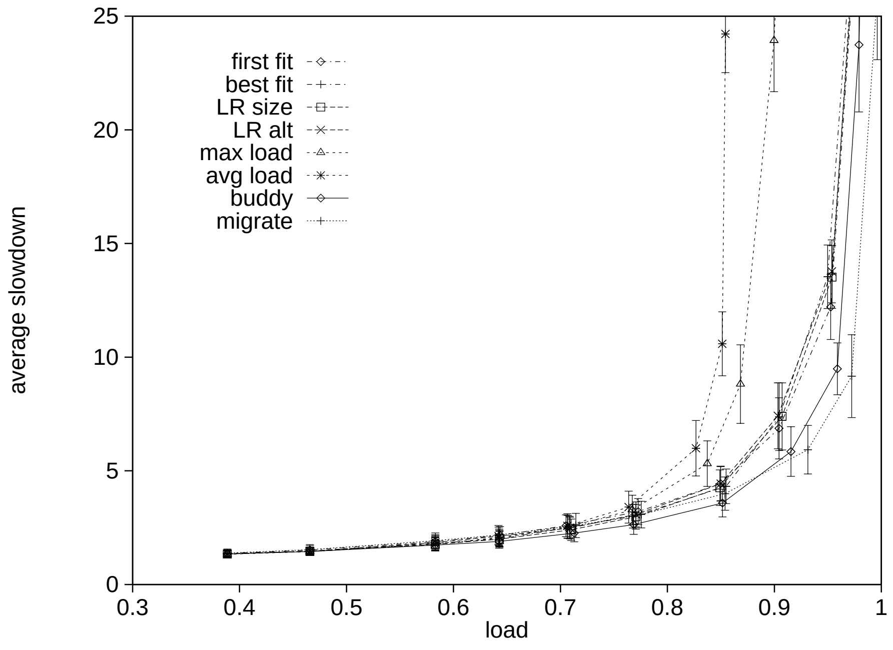


Fig. 8. *Average slowdown as a function of load for the different packing schemes.*

- The first four packing schemes (first fit, best fit, left-right by size, and alternating left-right) produce essentially identical performance.
- The two load-based schemes (minimal maximum load and minimal average load) are significantly worse than the previous four schemes. This is surprising since they take additional pertinent information into account.
- The buddy scheme and the migration scheme are similar to each other, and perform better than all other schemes. Migration has a slight advantage at the highest loads, while buddy has a slight advantage at medium loads.
- When looking at absolute values, rather than just comparing the different schemes, it is apparent that all schemes except the load-based ones can sustain loads leading to over 90% system utilization. The buddy and migration schemes can sustain loads leading to over 95% utilization. This implies that fragmentation is less of a problem than sometimes thought.

Why Load-Based Schemes Are Bad The load based packing schemes were expected to out-perform the oblivious schemes, because they judiciously choose the least loaded PEs to run new jobs. Such a choice was expected to make it easier to unify slots and to run the jobs in alternate slots. However, the simulation results show that choosing lightly loaded PEs leads to poor performance!

The reason for this situation seems to be that choosing PEs individually based on their loads leads to excessive fragmentation. As a result, it actually becomes harder to unite slots, as can be seen in the low unification counts for these two schemes in Fig. 10. Also, it is relatively difficult to schedule jobs to run in additional slots, beyond those to which they are mapped. This can be seen in the low slot counts for these two schemes in Fig. 11, which only improve at the highest sustained loads.

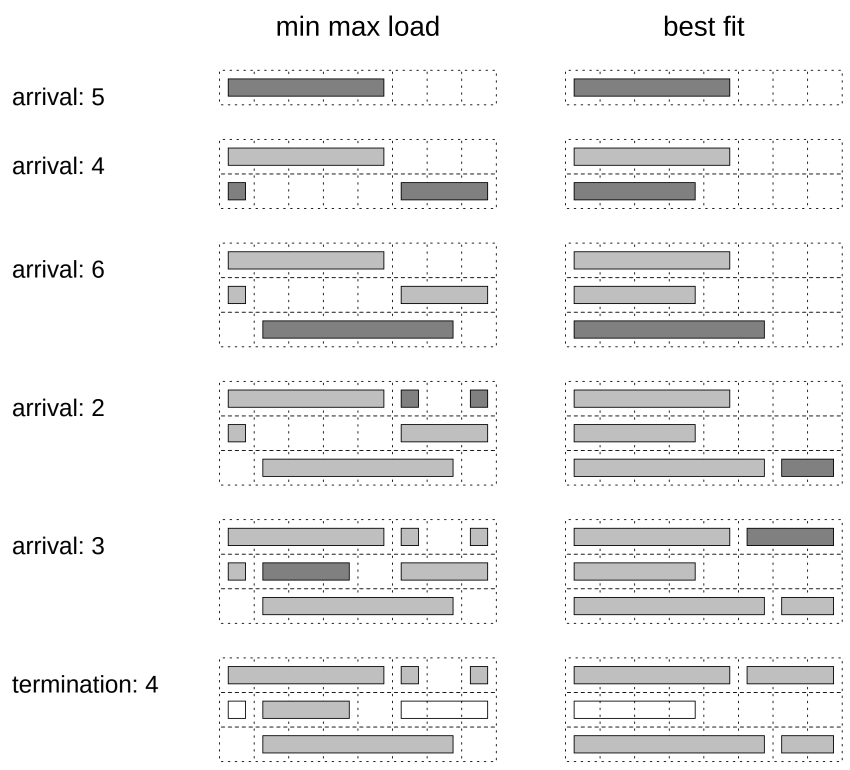


Fig. 9. *Example explaining the poor performance of load-based packing schemes.*

The following example shows how such harmful fragmentation can come about. Consider a sequence of job arrivals with sizes of 5, 4, 6, 2, and 3, in a system with 8 PEs. Fig. 9 shows how these jobs will be mapped by the minimal maximal load scheme and by the best fit scheme. Note that after the third job arrives, it seems that mapping to less loaded PEs leads to good balancing, as no PE has a load of more than two threads, whereas under best-fit some PEs have a load of 3 and some are completely idle. However, the mapping is fragmented, and becomes more so when the two additional jobs arrive. If now the job with size 4 terminates, the best fit scheme will end up with two fully-allocated slots, whereas the minimal maximum load scheme will have three lightly populated slots, and only the 2-PE job will be able to run in an additional slot.

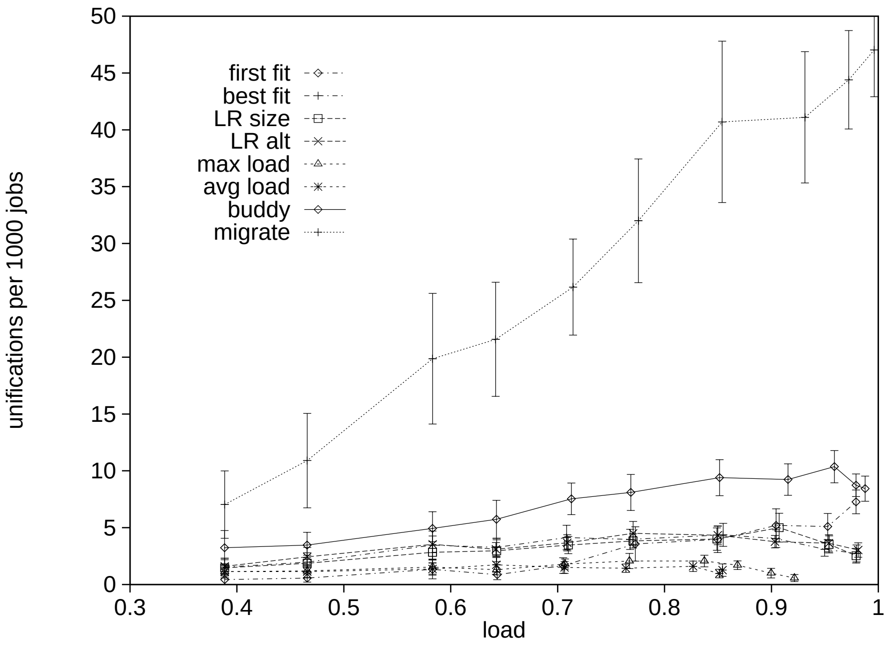


Fig. 10. *Number of slot unification performed with the different packing schemes.*

How Buddy Packing and Migration Achieve Their Good Performance The simulation results singled out the buddy packing scheme and the migration scheme as those that provide the best performance. It is interesting to note that while the final outcome of these two schemes is very similar, the underlying mechanisms are very different. To show this, we tabulate the number of slot unifications performed by the various schemes (Fig. 10) and the average number of slots in which a job may run under the various schemes (Fig. 11).

The plots show that while buddy packing achieves many more unifications than most other schemes, they are still a rare event: less than one percent of job terminations lead to a unification. With migration[4], this jumps to nearly 5%. Thus by using migration the system may keep the number of used slots close to the minimum necessary, at the price of re-mapping jobs frequently.

Another result of keeping the number of slots down to the minimum is that there is very little free space in each slot, and therefore there is little chance for a job to run in any other slot except the one to which it is mapped. therefore with migration the average number of slots available to each job is lower than in any other scheme. With buddy packing, the average number of available slots

[4] The definition of unification under migration is that the number of slots is reduced as a result of a job termination, excluding cases where the terminated job was the only one mapped to the slot.

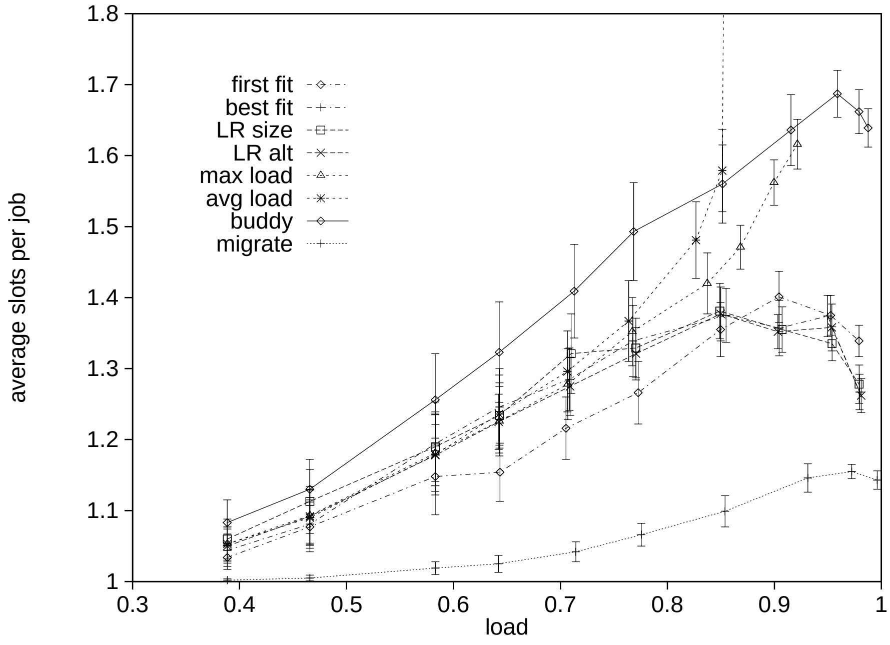


Fig. 11. *Average number of slots in which each job runs under the different packing schemes.*

is higher than in other schemes, because the buddy packing chooses groups of lightly loaded PEs for new jobs. Thus buddy packing achieves its performance not by minimizing the number of slots but rather by using alternative scheduling to allow each job to run in more slots.

The Importance of Unification and Using Alternate Slots As we saw, the performance of buddy packing is achieved by packing jobs so that they have a better chance to run in multiple slots. It is then interesting to check how important it is for the system to support this feature. Also, it is interesting to see how important it is to support slot unification.

The results are plotted in Figs. 12 and 13. Fig. 12, where the system does not support slot unification, is essentially identical to Fig. 8. We can therefore conclude that slot unification is not such an important feature[5] In Fig. 13, where the system does not support the execution of jobs in alternative slots, all the plots show somewhat reduced performance relative to Fig. 8. The most extreme degradation occurs with buddy packing. In fact, when jobs are not allowed to

[5] Not allowing unifications at all contradicts the definition of the migration scheme, so it is not plotted. However, note that migration does its unifications itself, and does not rely on the scheduler to do it.

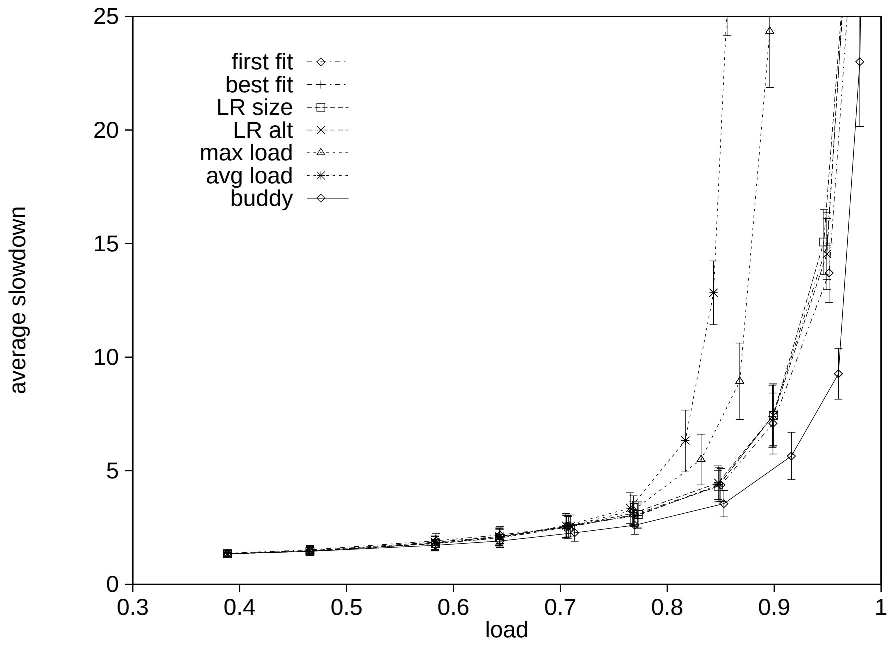


Fig. 12. *Average slowdown as a function of load for the different packing schemes, when no slot unifications are done.*

run in alternative slots, buddy packing performs quite poorly. Thus we see that this is an essential feature if buddy packing is to be used.

5 Conclusions

The current literature does not include any reference to the question of how to pack jobs for efficient gang scheduling. We have developed a number of packing algorithms, and evaluated them using simulations based on a realistic workload model. The results are that two approaches can lead to significant performance improvements over simple best-fit like algorithms: either use mapping based on a buddy system, or use migration to re-map jobs upon each job arrival and termination. Other approaches, such as mapping to the least loaded PEs, proved to be counterproductive.

The relatively good performance of the migration approach is a result of the fact that re-mapping leads to using the minimal number of scheduling slots possible. However, an implementation must then contend with the overhead of the migration process itself, which may be onerous. Therefore it is doubtful whether using migration is a realistic option, but it is still useful as a bound on the performance achieved by a strong on-line algorithm. Mapping based on a buddy system is much simpler and is not expected to involve considerable overhead.

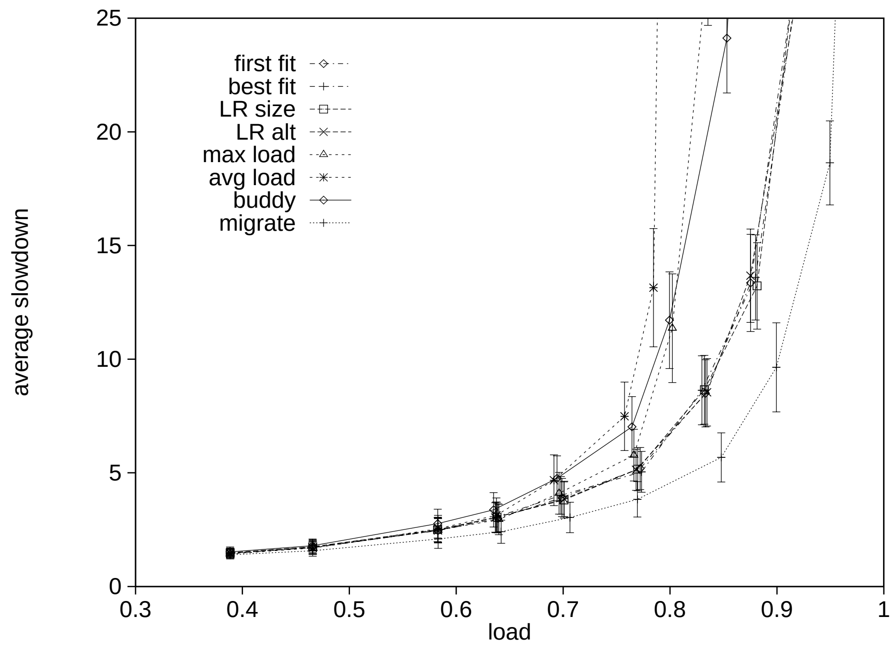


Fig. 13. *Average slowdown as a function of load for the different packing schemes, when jobs only run with their designated slot and do not use free space in other slots.*

However, it requires the system to support gang scheduling of jobs in multiple slots in order to achieve performance benefits. Without such a capability, the performance degrades sharply. Luckily, such support can be provided rather easily by mapping each job to multiple slots to begin with, where one mapping is the "real" one, and the others are tentative and can be deleted when some other job needs the space.

It should be noted that when a good mapping scheme is used, very high system utilization is possible. In our simulations of the buddy and migration schemes, the system only saturated when the utilization was higher than 95%, which is significantly higher than the 50-80% range reported for production systems using static partitioning [8, 29, 15]. This means that fragmentation is less of a concern than is sometimes thought. The high utilization can be attributed to two factors: first, when using time slicing, bad scheduling decisions are less harmful than when using static partitioning, because they only affect one scheduling slot. Other slots that suffer less fragmentation dilute the bad effect, and lead to lower average fragmentation. Second, our workload study indicated that there are many small jobs and many jobs that are powers of two. Both these classes are easier to pack than large jobs with strange sizes.

In the future, we would like to extend this work in the following directions:

- Improve the workload model even further, by making a more quantitative analysis and by studying the arrival process in more detail. For example, is the common assumption of a constant arrival rate a good model, or are there large fluctuations?
- Check what features of the workload model are the most significant ones, in terms of their effect on the results. for example, is user modeling important or can it be ignored?
- Add batch processing both to the workload model and to the gang scheduling algorithm. This is a significant issue because batch jobs, unlike interactive jobs, can be buffered by the system and used to fill in holes that would otherwise be lost to fragmentation.
- Add memory considerations to the mapping schemes. This is hampered at present by the lack of any real data about memory requirements of parallel jobs.

References

1. A. Barak and A. Shiloh, "*A distributed load-balancing policy for a multicomputer*". *Software — Pract. & Exp.* **15(9)**, pp. 901–913, Sep 1985.
2. J. M. Barton and N. Bitar, "*A scalable multi-discipline, multiple-processor scheduling framework for IRIX*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 45–69, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.
3. S-H. Chiang, R. K. Mansharamani, and M. K. Vernon, "*Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 33–44, May 1994.
4. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.
5. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Bin packing with divisible item sizes*". *J. Complex.* **3(4)**, pp. 406–428, Dec 1987.
6. F. Douglis and J. Ousterhout, "*Process migration in the Sprite operating system*". In 7th *Intl. Conf. Distributed Comput. Syst.*, pp. 18–25, Sep 1987.
7. D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
8. 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. Lecture Notes in Computer Science Vol. 949.
9. D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.
10. D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.*, 1996. to appear.
11. D. G. Feitelson and L. Rudolph, "*Parallel job scheduling: issues and approaches*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson

and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

12. D. G. Feitelson and L. Rudolph, "*Wasted resources in gang scheduling*". In 5th *Jerusalem Conf. Information Technology*, pp. 127–136, IEEE Computer Society Press, Oct 1990.
13. B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, Aug 1995.
14. B. C. Gorda and E. D. Brooks III, *Gang Scheduling a Parallel Machine.* Technical Report UCRL-JC-107020, Lawrence Livermore National Laboratory, Dec 1991.
15. S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
16. Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.
17. Intel Supercomputer Systems Division, *Paragon User's Guide.* Order number 312489-003, Jun 1994.
18. K. C. Knowlton, "*A fast storage allocator*". *Comm. ACM* **8(10)**, pp. 623–625, Oct 1965.
19. P. Krueger, T-H. Lai, and V. A. Radiya, "*Processor allocation vs. job scheduling on hypercube computers*". In 11th *Intl. Conf. Distributed Comput. Syst.*, pp. 394–401, May 1991.
20. S. T. Leutenegger and M. K. Vernon, "*The performance of multiprogrammed multiprocessor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 226–236, May 1990.
21. 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. Lecture Notes in Computer Science Vol. 949.
22. M. H. MacDougall, *Simulating Computer Systems: Techniques and Tools.* MIT Press, 1987.
23. S. Majumdar, D. L. Eager, and R. B. Bunt, "*Scheduling in multiprogrammed parallel systems*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 104–113, May 1988.
24. J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.
25. J. L. Peterson and T. A. Norman, "*Buddy systems*". *Comm. ACM* **20(6)**, pp. 421–431, Jun 1977.
26. D. L. Russell, "*Internal fragmentation in a class of buddy systems*". *SIAM J. Comput.* **6(4)**, pp. 607–621, Dec 1977.
27. T. Suzuoka, J. Subhlok, and T. Gross, *Evaluating Job Scheduling Techniques for Highly Parallel Computers.* Technical Report CMU-CS-95-149, School of Computer Science, Carnegie Mellon University, 1995.
28. Thinking Machines Corp., *Connection Machine CM-5 Technical Summary.* Nov 1992.
29. M. Wan, R. Moore, G. Kremenek, and K. Steube, "*A batch scheduler for the Intel Paragon MPP system with a non-contiguous node allocation algorithm*". In *Job Scheduling Strategies for Parallel Processing II*, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Computer Science.
30. G. K. Zipf, *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.