Evaluation of Design Choices for Gang Scheduling using Distributed Hierarchical Control

Dror G. Feitelson* Larry Rudolph
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Gang scheduling — the scheduling of a number of related threads to execute simultaneously on distinct processors — appears to meet the requirements of interactive, multiuser, general-purpose parallel systems. *Distributed Hierarchical Control* (DHC) has been proposed as an efficient mechanism for coping with the dynamic processor partitioning necessary to support gang scheduling on massively parallel machines. In this paper, we compare and evaluate different algorithms that can be used within the DHC framework. Regrettably, gang scheduling can leave processors idle if the sizes of the gangs do not match the number of available processors. We show that in DHC this effect can be reduced by reclaiming the leftover processors when the gang size is smaller than the allocated block of processors, and by adjusting the scheduling time quantum to control the adverse effect of badly-matched gangs. Consequently the on-line mapping and scheduling algorithms developed for DHC are optimal in the sense that asymptotically they achieve performance commensurate with off-line algorithms.

Keywords: Distributed Hierarchical Control, fragmentation, gang scheduling, load balancing, mapping, processor utilization, variable partitioning.

Parts of this research have been presented at conferences [18, 19].
*Part of this work was done while at the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

1 Introduction

While there is great interest in multiuser, general-purpose parallel computing, it is not obvious how to partition the computing resources of the parallel computer among competing jobs. Space-slicing, where the parallel machine is physically partitioned among jobs, and time-slicing, in which processors run threads from a global queue, each have their drawbacks. In the early '80s, John Ousterhout proposed the thesis that related threads should be scheduled to execute together [32]. This idea, now known as *gang scheduling*, is becoming increasingly popular, and can be found in various forms on commercial machines such as the CM-5 from Thinking Machines [29], the Intel Paragon [25], the SGI multiprocessors running IRIX [1], the Meiko CS-2 [14], the Alliant FX/8 [41], and the MasPar and DAP SIMD arrays. Gang scheduling has also been used in a production system on a BBN Butterfly at LLNL [20], which is now being ported to a new Cray T3D machine, and several other experimental systems [13, 39, 4, 17, 6].

At first blush, it might appear that gang scheduling is a luxury that may not be worth the price. An optimal packing of gangs that gives minimal wasted processors is an NP-complete problem. Migration of gangs might be required to compensate for a poor initial mapping. The code to simultaneously schedule all the threads of each gang might be overly complex requiring elaborate bookkeeping and global system knowledge. Indeed, some of the existing implementations of gang scheduling are far from perfect. Symptoms include relying on a centralized controller, which might not scale well, and large overheads, that necessitate the use of long (and even non-interactive) scheduling time quanta.

The reason that gang scheduling prevails despite these difficulties is that it provides important benefits. For example, for applications with fine grained interactions, there are large performance improvements over uncoordinated scheduling [17]. Gang scheduling decouples the application from the system, providing an environment similar to that of a dedicated machine. It is useful with any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases, and fosters support for interactive response times. And the fact that interacting processes are guaranteed to execute simultaneously allows them to access hardware communication devices in user mode, without the overheads associated with operating system protection [29, 39, 23].

A distributed hierarchical control (DHC) scheme for supporting gang scheduling has been proposed previously [16]. DHC defines a control structure over the parallel machine and combines time slicing with a buddy-system partitioning scheme. Given the DHC framework, this paper investigates several design choices and their performance implications. Processor fragmentation could potentially negate all the benefits of gang scheduling. That is, because gang scheduling demands that no thread execute unless all other gang member threads execute, then some processors may be idle even when there are threads waiting to be run. Our results show that no more than 5 to 25% of the processing power is lost to fragmentation. This result means that neither optimal packing nor thread migration are needed, allowing for a scalable, efficient implementation, at least for the three widely differing job arrival and size distributions that are examined.

The design choices that are considered include different ways to map new gangs to processors and to allocate processing time to each gang. Prudent choices allow the system to reduce fragmentation and also to limit the resources lost to fragmentation. With the best design choices, our simulations demonstrate that an on-line scheduling algorithm can asymptotically achieve the same performance as an off-line algorithm. Our algorithms therefore leave little if any space for improvement, at least not within the DHC framework.

The next section reviews the concept of gang scheduling and outlines the DHC scheme for its implementation. Section 3 describes the proposed solution algorithms in detail, and presents simulation results that tabulate their performance characteristics. Section 4 analyzes the fragmentation caused by gang scheduling in general, and DHC in particular, and then combines this with the results of the previous section to show that DHC is optimal, in the sense that it approaches off-line performance. Section 5 presents the conclusions.

2 Background

The term *Gang scheduling* refers to the scheduling of a set of related threads[1] on a parallel machine so that all the threads in a gang execute *simultaneously* on a (sub)set of the processors. Thus, at any time, there is a one-to-one mapping between threads and processors. We emphasize that although the total number of threads in the system (and even in a single job) may be larger than the number of processors, no gang contains more threads than processors. For example, applications may accept the number of processors as a runtime parameter, and create gangs accordingly. Thus the gang becomes the schedulable entity, rather than the individual threads. The job affiliations of gangs are ignored, except in the context of accounting.

Gang scheduling combines the best features of both space slicing and time slicing. Space slicing provides simultaneous execution on a dedicated partition, which allows performance guarantees to be made — execution of a gang proceeds essentially the same as on a dedicated machine. Since not all parallel jobss require all the processors, the processors can be partitioned (or space sliced) so that more than one parallel job can execute at the same time. However, in pure space slicing, the multiprogramming level is limited by the maximum number of partitions.

Time slicing promotes efficient and flexible use of the resources by matching gangs with complementary requirements for processors. The preemption helps efficiency by allowing new and better matches to be found as the execution proceeds. It promotes flexible use of the resources by enabling applications that require all the PEs to co-exist with other applications. It is necessary in order to support swapping when the available physical memory is insufficient. and it enables the short response times needed for interactive work by recognizing interactive vs. batch jobs at runtime, without prior notification.

We assume a system consisting of a parallel processor that is to support a set of jobs in an interactive fashion. A job may be a parallel program consisting of one or more gangs or it may be a process with a single thread[2]. From the multiprocessor's point of view, some threads occasionally create new gangs of threads for execution. A job has no significance to the system, only gangs are important. Thus the workload is a sequence of requests to execute gangs, and the problem is to decide on-line when and where to schedule each gang. Note that this is not a batch system: single-thread gangs are also possible, and a fast response time is desired.

[1] Threads are the schedulable entity in parallel systems, analogous to processes in conventional (e.g. Unix) systems. This view is directly matched to the control-parallelism programming model, where threads are defined by language constructs. It is also applicable to the data-parallel and SPMD models, where there is one thread on each (virtual) processor, and they all execute the same code.

[2] One may find it helpful to consider the following specific scenario: Users access a shared multiprocessor through a local network, from graphical terminals (e.g. X terminals). We assume all programs, including shells, execute on the multiprocessor. The shell programs initiate parallel programs (i.e. gangs).

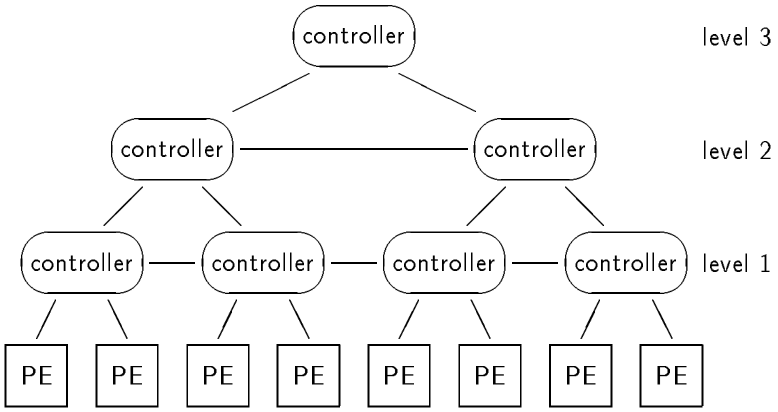

Figure 1: *Controllers and processors in the Distributed Hierarchical Control scheme.*

The input to the system is a stream of requests to schedule new gangs. Each request has a size (number of threads) and is characterized by the duration of its execution on a dedicated machine. Some gangs may have a very long execution time, and therefore do not require a fast response time, but this is *not* known in advance. We assume that all the threads within the same gang execute for the same amount of time, i.e. that the computational load is balanced between them. The request to schedule a gang may occur at any time. Thus the total workload may be characterized by three parameters: the distribution of gang sizes, the distribution of execution times, and the distribution of interarrival times.

Some degree of centralized control seems to be required to implement the coordinated multi-thread switching across processors required in order to implement gang scheduling. On the other hand, when the workload is composed of many small gangs, distributed control seems to avoid sequential bottlenecks. These two contradicting designs are reconciled by using a hierarchical control structure. Specifically, we consider *Distributed Hierarchical Control*, proposed previously [16]. This control structure is especially interesting because it is now being used in the operating system of the RWC-1 project in Japan [24] and in a prototype implementation of gang scheduling on an IBM SP2.

DHC is based on a hierarchy of controllers that organize the processors in a buddy-system fashion (Fig. 1) [34]. A distinct controller is associated with each block of processors. The size of each block is a power of two, and is the union of two blocks half the size. This also defines the hierarchy. A controller at level i coordinates activities involving more than half of the 2^i PEs spanned by its subtree. Controllers in low levels of the hierarchy provide for local control, while those in higher levels take care of global coordination. In addition, there are lateral connections among the controllers that are used for load balancing.

Note that the hierarchy describes the *logical* control structure used by the operating system, and only suggests but does not imply a hardware hierarchy. The possible advantages of a hardware hierarchy are that separate processors for operating system tasks can be optimized for their function, and that it would prevent the operating system from interfering with the execution of user threads. However, it is perfectly possible to implement the control structure in software on the same set of processors that are being scheduled.

The hierarchy of controllers is used to map threads to processors as follows: A request to map a new gang of size s originates from a thread executing on some processor. The request ascends the tree of controllers until it reaches the appropriate level for its size, and then moves across to some controller that will balance the overall load. This controller is thereafter said to *control the gang*. Finally, the gang is mapped to a set of processors under this controller. Once mapped, threads do not migrate to other processors and control of a gang is not passed between controllers. When a new gang is mapped or an existing one terminates, controllers exchange information about the change in load [16].

The scheduling proceeds in waves that propagate down the tree of controllers[3]. In regards to scheduling, each controller can be either idle, active, or disabled. At each instant, there is a front of active controllers across the tree. All the controllers above this front have already done their scheduling, so they are idle until the next wave. When a controller is active, it is involved in scheduling all the gangs it controls. The controller transmits a command to execute the threads of the scheduled gang on the processors by passing the execute command down to its subordinate controllers that, in turn, continue to pass it down until it reaches the appropriate processors. Each gang executes for a time quantum (the length of a time quantum is variable and will be investigated in a subsequent section). When a controller is active, all of its subordinate controllers are disabled. However, if the scheduled gang does not utilize all the processors, *selective disabling* can be used: some subordinate controllers are left active and may use the leftover processors to schedule smaller gangs. In either case, after the controller completes its scheduling of all its gangs, it becomes idle and its immediate subordinate controllers are activated, thus moving the active front downwards. When the bottommost controllers are done, a new wave is started from the root.

The use of a hierarchy of control is not a new idea: it exists in a restricted form in all master/slave configurations and in host/node configurations where the host processor controls the use of a back-end computation engine. The novelty of the DHC scheme lies in the fact that the master is also a parallel machine, tightly coupled to the target machine. To our knowledge, this design has so far been proposed only in the EGPA project from Erlangen and its followups [22], and in the NETRA image processing system [7]. However, we are unaware of any publications concerning operating system issues in these projects. Other systems that use a central controller, thus creating a two-level hierarchy, include PASM [42] and MIDAS [31]. A multilevel virtual hierarchy implemented in software appeared in the design of MICROS [43] and CHoPP [40]. All of these systems use the hierarchy and clustering to provide partitioning, but do not support interactive gang scheduling.

3 Evaluation of Distributed Hierarchical Control

In this section we detail the mapping and scheduling algorithms for DHC, and show how they support preemptive gang scheduling. Various stages in the algorithms may be performed in a number of alternative ways. These options are compared using simulations. We must first, however, define our performance metrics.

[3]It would be natural to call this a "wave scheduling" algorithm, but regrettably this name has already been used by others [43].

3.1 Performance Metrics and Simulation Parameters

The main performance metric is the *slowdown* suffered by the different jobs. The slowdown of a job is defined as the quotient of the response time of the job (execution time plus preempted time) divided by the actual execution time; this is similar to the "response ratio" criterion advocated by Brinch Hansen [3]. The time during which a job is preempted is used to run competing jobs, and so, assuming small system overheads, the slowdown actually measures the perceived load on the machine. The reason for using the slowdown rather than the more common response time metric is that it is a normalized value which is useful for cases where the mean execution time is not unity; when the mean is unity, the two measures are equivalent. The reciprocal of the slowdown is called the *run fraction* of the job. It gives the percentage of the time that the job is actually running. The slowdown can take on any value greater than one, whereas the run fraction is a number between 0 and 1.

In addition to the performance under a given load, it is also interesting to study the way in which the system reacts to various load conditions. Of course, the system will become saturated whenever the arrival rate of new jobs approaches the completion rate of jobs already in the system. As the completion rate depends on the efficiency of the system, saturation may occur even if the raw computing resources are more than sufficient. The average slowdown metric may be used to gauge the efficiency of the system: as the arrival rate increases and the system approaches saturation, the slowdown tends to infinity.

We use the functional dependence of the slowdown on the generated load as our main performance measure. The generated load is the fraction of available resources consumed and is defined explicitly in the Appendix. The slowdown starts at 1 for low loads, and increases as saturation is approached. The asymptote along which the slowdown shoots upwards indicates the maximal load that can be sustained by a certain system; the objective of the various algorithms is to push this threshold to higher values.

Recall that the workload is characterized by three parameters: the distribution of gang sizes, the distribution of execution times, and the distribution of interarrival times. We assume that there is no correlation between the different parameters. For example, a small gang may have a long execution time. In particular, this means that on average large gangs have a larger cumulative service requirement than small gangs, which reflects a hidden assumption that parallelism is used to solve larger problems [21, 15].

We shall make the common assumptions that the execution times and the interarrival times are exponentially distributed. The distribution of gang sizes is harder to characterize. It is often thought that parallel programs would use either all the available processors or only one, leading to a bimodal distribution. While this may be true in some cases, e.g. for programs written in the "agenda parallelism" style [5], it is not true in the general case, and especially not for massively parallel architectures [15]. For example, interdependencies among the program components may limit the degree on parallelism [41], and considerations involving communication cost may reduce the usable parallelism even further [30, 33]. Moreover, it is reasonable to assume that owners of expensive parallel supercomputers will charge jobs according to their level of parallelism in addition to the total number of CPU cycles consumed. This will prompt users to use the number of PEs that gives the best overall efficiency, rather than that which matches the maximal parallelism [12, 37].

In this paper, we shall use three specific distributions of gang sizes. The first is the *uniform* distribution. The second is the *harmonic* distribution, where the probability of size s is proportional to $1/s$. Consequently, there will be a larger fraction of small gang sizes. The third is *uniform over powers of two*, representing a conscious effort by users to be efficient, similar to the use of buffers with 512 bytes for I/O. All three are defined on the range of 1 to P, the number of processors in the system, which is assumed to be a power of two. $P = 32$ was used in all cases, except where otherwise noted. The simulation methodology is described in the appendix.

3.2 Mapping Algorithm

The mapping algorithm maps threads to processors. With the DHC scheme, this is done in three stages:

1. The request to map a new gang propagates up the tree of controllers until it reaches the level in which controllers have enough processors under their control to satisfy it.
2. The controllers at that level communicate among themselves to decide which will take responsibility for the new gang.
3. The gang's threads are mapped to processors (at the leaves) under the chosen controller. There is no thread migration.

The load balancing and mapping can be done in various ways. We now describe the options and compare them using simulation results.

Mapping gangs to controllers in a balanced way

Load balancing can be achieved by the initial placement in a balanced manner, i.e. newly created threads are mapped to those processors that have the least load, or by dynamic run-time migration of active threads from overloaded processors to underloaded ones. We restrict our attention to the first approach, thus saving the need to transfer state information from one processor to another.

Mapping new gangs to controllers to ensure a balanced load implies some knowledge about the loads under at least some of the controllers. A full knowledge scheme is out of the question, as it does not scale and delays the mapping too much. Therefore, we propose that controllers in each level be partially connected in some static pattern, with load information exchanged only between directly connected (i.e. neighboring) controllers. The performance of such a scheme may be expected to depend on the interconnection pattern of the controllers. We compare two possibilities:

- *Ring connections* — The controllers are connected to form a ring. The loads on the controller that originated the gang and its two nearest neighbors are compared; the new gang is mapped to the least loaded of the three.
- *Cube connections* — The controllers within each level are connected in a hypercube pattern; the higher the level, the lower the dimensionality of the cube.

To get an idea of the performance in absolute terms, three other (unrealistic) schemes are also checked:

- *No load balancing* — This is obviously the worst case. Each gang is mapped on the same subtree in which it originated.

Load balancing scheme	Expected load per PE
From root	2.57
Cube	2.61
Ring	2.88
Random	3.01
No balancing	3.30

Table 1: *The expected number of threads mapped to each processor for the different load-balancing schemes (requests generated by all threads, generated load of 0.645, harmonic distribution).*

- *Random* — Each new gang is mapped to a random controller at the appropriate level.
- *From root* — The mapping is done through the root of the tree, choosing the least loaded branch at each step until the appropriate level is reached. This is expected to be nearly optimal. It is an unrealistic scheme since the root would be a performance bottleneck.

In the first three cases the placement of a new gang will be under a controller that is directly connected to the one under which the request originated, so the resulting distribution depends on where requests originate. We simulated two models of request generation. In one, every thread currently in the system has an equal probability of generating the next request. This is realistic for parallel systems, especially if shell threads are executed on the same processors as application threads. In the second model, all requests are generated on processor #1. This represents an extreme case where requests are only generated by shells, and shells are restricted to run on a small subset of processors.

We compare load balancing schemes only for the case of a harmonic distribution of gang sizes. The case of uniform distributions is not of interest since most gangs will be mapped to the top levels of the tree where there are only a small number of controllers. With P processors, half of the gang sizes between 1 and P would be mapped to the root controller and a quarter of them to the controllers just below the root. Indeed, simulations showed that all the schemes have the same performance under these conditions.

It was thought that the differences between load balancing schemes would be more pronounced when there was a larger number of processors since in small systems the controller hierarchy is very shallow and the information available is nearly complete. We therefore performed simulations of the different load balancing schemes for both 32 and 128 processor configurations. As it turns out, our reservations were unfounded, and the results for 128 processors were almost identical to those for 32. The graphs shown here are from the results for 32 processors.

The results of the simulations are as follows. If we assume that any thread can generate the request to create the next gang, the distribution of loads turns out to be rather similar for all the schemes. Table 1) shows the means of the observed distributions. While there is an observable improvement over the case of no load balancing, the improvement is not dramatic. Similar results are observed when the slowdown curves of the different schemes are plotted (Fig. 2). It is possible to discern that the cube interconnection pattern and mapping through the root are best, while random placement performs poorly relative to explicit load balancing.

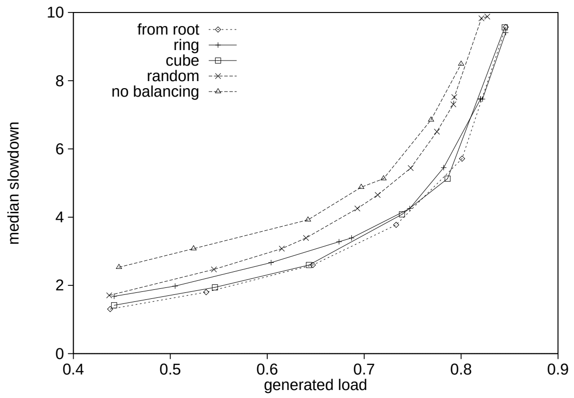

Figure 2: *If new gangs are generated by existing threads with a uniform distribution, all load balancing schemes are similar, and slightly better than no load balancing* (harmonic distribution).

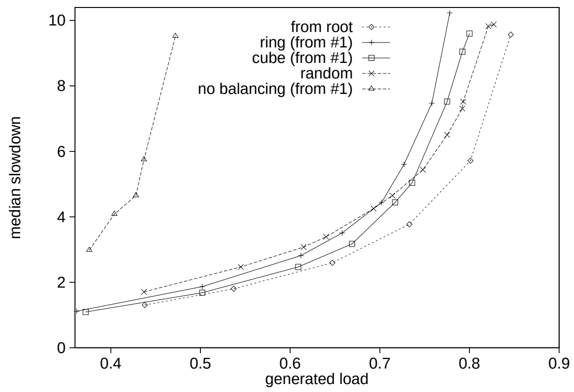

Figure 3: *If all new gangs are generated on processor #1, load balancing provides a dramatic improvement over no load balancing. In addition, the differences between different load balancing schemes are amplified.* (harmonic distribution).

The relatively small improvement gained by load balancing is in contrast to results from distributed systems, that indicate that even simple load balancing schemes are dramatically better than no load balancing (with a model of equal average arrival rates to all nodes) [11]. The reason is that gangs of threads, generated with different sizes, are always spread across a number of processors so as to support gang scheduling, thereby creating an implicit spreading of the load. In distributed systems all load balancing has to be done explicitly.

If we assume that all the requests originate from processor #1, there are marked differences between the different schemes (Fig. 3). If no load balancing is used, the system saturates at a very low load. Using either load balancing scheme improves things considerably, and while using cube connections is better than ring connections, the difference is small. Furthermore, both load balancing schemes are noticeably inferior to the ideal method of mapping through the root of the controller tree, and even inferior to random mapping at high loads. These results indicate that it is important to prevent situations in which requests originate predominately from a small set of processors. For example, restricting shells to execute on a small set of processors is a bad idea. Similar results have been reported in [26].

Mapping threads to PEs

Once a controller has been chosen, the threads need to be mapped to the underlying processors. To do so, the controller partitions the threads among its direct subordinates; they go on to partitions their respective portions among their own subordinates, and so on. Let s be the number of threads that a controller at level i is to map. Note that initially $2^{i-1} < s \leq 2^i$, where 2^i is the number of processors in the subtree of the controller, but in lower levels it may happen that $s \leq 2^{i-1}$. Denote the total loads on the controller's direct subordinates by t_1 and t_2. There are two basic approaches for dividing s into s_1 threads running under the left subordinate controller and s_2 under the right one:

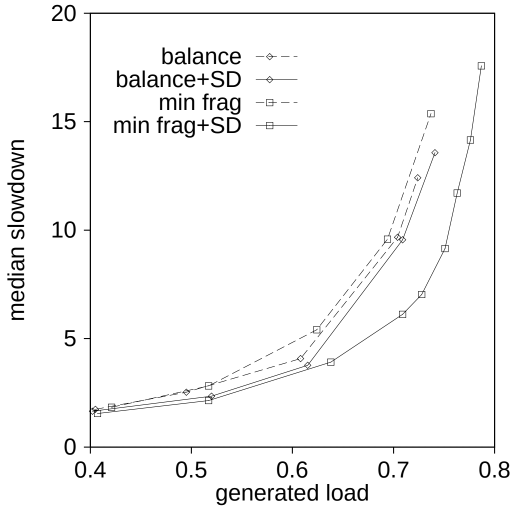

Figure 4: *Minimized fragmentation reduces the slowdown, provided selective disabling (SD) is used* (uniform distribution).

- *Strive for balance.* If $|t_1 - t_2| \leq s$, and also $|t_1 - t_2| \leq 2^{i-1}$, make the division so that $s_1 + t_1 = s_2 + t_2 \pm 1$. Otherwise, revert to the second approach:
- *Minimize the fragmentation.* Assuming w.l.o.g. that $t_1 < t_2$, assign $\min\{s, 2^{i-1}\}$ of the threads to the first child. If 2^{i-1} are mapped, this is the child's exact capacity, so there will be no fragmentation on that child. The other child gets the rest, if any.

The simulation results indicate that striving for balance is marginally better than minimizing the fragmentation. However, if selective disabling (SD) is used to improve the scheduling of different gangs side by side, then mapping to minimize the fragmentation is significantly better (Fig. 4). Selective disabling is discussed at length below.

3.3 Gang Scheduling Algorithm

The scheduling algorithm decides which gangs will be executed when, and for how long. The activity in the system is divided into *scheduling rounds*; during each round every gang is scheduled. A round begins at the root controller. First it takes care of scheduling gangs which it controls. For each such gang, the root controller instructs its subordinates to schedule threads of the chosen gang; the subordinates propagate these instructions down to their subordinates, and so on until the schedule command reaches the processors. As the tree is balanced, all the processors receive the instruction at approximately the same time. Hence when each processor switches to a thread from the designated gang, they are actually performing a multi-thread switch.

Let the total time allocated to a scheduling round be Q, and let Q_r be the time spent executing the gangs controlled by the root during this round. During this time only the root is active, and the other controllers are disabled[4]. After all gangs controlled by the root controller have executed for their respective time slices, the root becomes idle and its immediate subordinate controllers become active with a value of $Q - Q_r$ time left for this round. They repeat the process with their

[4]This is total disabling. Selective disabling is described below.

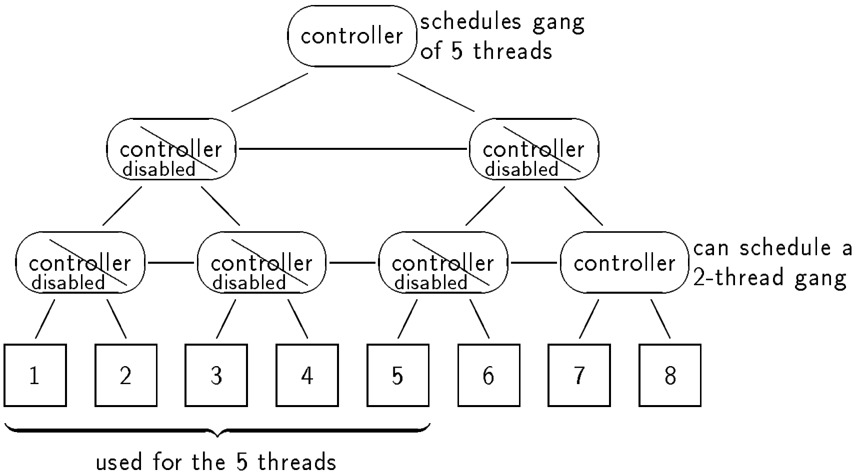

Figure 5: *Increased processor utilization due to selective disabling.*

gangs and their subordinates. This continues until all the controllers have scheduled their gangs, and then a new round starts from the root.

Selective disabling — increasing the controller activity

An obvious optimization is to disable only some of the subordinate controllers when a controller is scheduling a gang of threads using only part of the processors. This is referred to as *selective disabling* (SD). Subordinate controllers that control processors that are left over — i.e., do not have a thread in the gang — need not be disabled. These controllers may use the extra processors to schedule other smaller gangs of threads. For example, if the top controller in a three-level tree schedules a gang of 5 threads on processors 1–5 (Fig. 5), the controller of processors 7 and 8 should not be disabled; it can use these two processors to schedule a two-thread gang. Likewise, processor 6 can schedule a single independent thread locally. Note that the scheduling controller and the processors that are actually used for the current gang define a partial tree within the full binary tree rooted at this controller. Only controllers along branches of this partial tree should be disabled. This means that selective disabling is easy to implement: when the top controller issues an instruction to schedule a certain gang, controllers that receive this instruction check if this gang has any threads in their sub-tree. If so, they pass the instruction downwards and disable themselves. If not, they ignore the instruction and schedule one of their own gangs instead. Note that as the active front progresses down the tree, these lower controllers get another chance at scheduling their gangs, so small gangs tend to receive better service.

The simulations indicate that selective disabling is indeed an effective method, as demonstrated in Figs. 4 and 9. This can also be seen in the way that the distribution of run fractions (the fraction of the time that a gang actually executes, or 1/slowdown) changes when selective disabling is introduced. With selective disabling, low run fractions become less probable, and higher ones become more probable (Fig. 6). As may be expected, the improved performance is a result of the fact that small gangs get to run more. This does not come at the expense of large gangs; on the contrary, the run fraction of large gangs is also improved (Fig. 7). This happens because the small

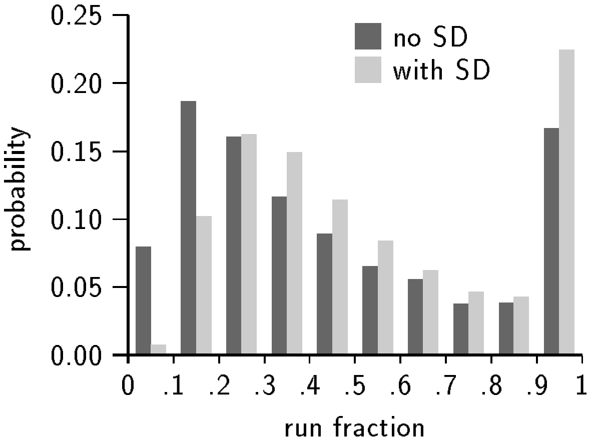

Figure 6: *With selective disabling higher run fractions are more probable (generated load of 0.517).*

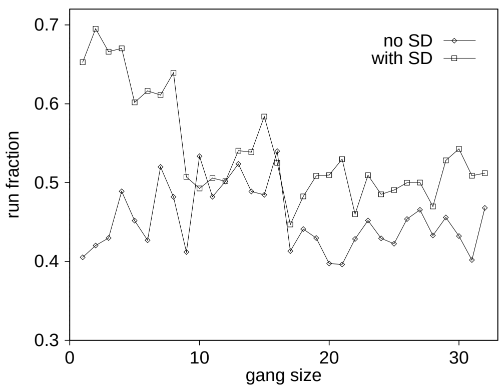

Figure 7: *Dependence of run fraction on gang size (generated load of 0.517).*

gangs terminate sooner, and therefore present less of a competition for the processors.

Determining time slices

We require that each scheduling round consume the same amount of time, Q. There are several ways in which this time can be divided among the executing gangs. We make two major divisions: (i) the ratio of time allocated between each controller and its subordinates, and (ii) the ratio of time allocated among the gangs assigned to a specific controller. The time allocated to a controller means the amount of time allocated to executing all the gangs assigned to the controller. Note that we are assuming that gangs may be allocated different lengths of time quantums.

We use the following notation to describe the allocation of time. We assume that there are n competing gangs, with sizes $s_1, \ldots, s_n$ (i.e. gang i has s_i threads). The total number of threads will be denoted $S = \sum_{i=1}^{n} s_i$. The time allocated to gang i is t_i. For simplicity, assume for now that the n gangs are all assigned to the same controller.

We define three approaches. The first emphasizes fairness to all gangs and the others penalize gangs that cause excessive waste of resources (and thereby deny other gangs the opportunity to execute). Fig. 8 illustrates the different schemes for a simple case of 4 gangs with staggered sizes.

- *Uniform time slices:* $t_i \propto \frac{1}{n}$. All gangs execute for the same amount of time, regardless of their size. At first glance this seems to be the fairest approach, but it might be inefficient (especially without selective disabling), as small gangs might leave a large number of processors idle during their whole time slice [16]. One might question the fairness of allowing a small gang to waste more processing power than a large one.

- *Divide according to weight:* $t_i \propto \frac{s_i}{S}$. The relative time allocated to a gang is determined by the number of its threads (its weight). Thus a gang with many threads (that leaves fewer extra processors) gets to run longer. This is desirable because given a certain number of processors, large gangs use more of them. At first glance it seems that this policy is subject to user countermeasures, where redundant threads will be generated just to increase the execution

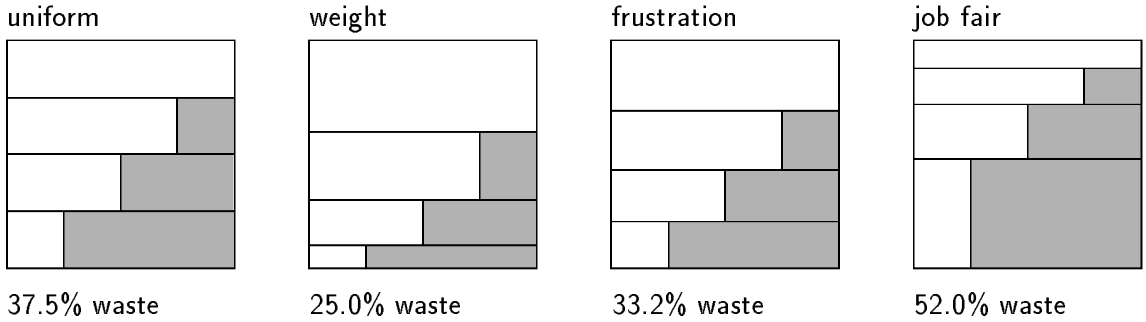

Figure 8: *Illustrative example of different schemes to set scheduling quanta, and the resulting loss of resources due to fragmentation. The workload is four gangs of sizes P, $\frac{3}{4}P$, $\frac{1}{2}P$, and $\frac{1}{4}P$. The horizontal dimension denotes processors, and the vertical shows the relative sizes of the scheduling slots within a scheduling round. Gray shading denotes loss to fragmentation.*

priority. The simulations show that such actions are futile. For example, the results of Fig. 7 use this policy, and bear witness to the fact that there is no advantage in creating extra-large gangs. In fact, this policy just partly compensates for the better service that selective disabling gives to small gangs.

- *Divide to reduce frustration:* $t_i \propto \frac{1}{S-s_i}$. The weight of a gang is not proportional to the number of its threads, but rather to the reciprocal of the number of threads in gangs that are blocked from running when it runs. This is justified as follows: scheduling a gang causes the others to be frustrated. The time should be set so that each scheduling decision adds a constant amount of frustration to the system (otherwise performance would be improved by reducing the time allocated to decisions that add more frustration). As the amount of frustration is the product of the number of frustrated threads by the time, the expression for t_i follows. The result is very similar to the previous scheme, but tends to have less variance in the run times.

It should be noted that we do not consider the option of setting $t_i \propto \frac{1}{s_i}$, i.e. inversely proportional to the gang size, since such a policy is counterproductive: it promotes fairness at the job level, at the expense of making parallelism unattractive. At the conceptual level, we believe that jobs that attempt to use the parallel resources should be encouraged, not penalized. The system must assume that if a job has many threads, it needs that many threads. Thus fairness at the thread level will translate into handing more resources to those jobs that have more threads, and parallelism becomes an effective means to obtain computing resources. If there is a real concern that users abuse the system, they should be curbed by suitable accounting practices. At the technical level, this scheme can lead to excessive wasted resources as illustrated in Fig. 8.

The three time slicing approaches can also be used to allocate time between the controllers. In the uniform case, one considers the longest chain in the hierarchy, where length is the number of gangs allocated to the controllers along the path. If the path length is m, then each gang along the path is allocated $1/m$ of the time. If the root has k gangs, it uses k/m of the time; then the two subtrees get to execute for the remaining $(m-k)/m$ time. Since they may have different numbers of gangs, their local allocations may be different. For division by weights, s_1 is taken to be the total

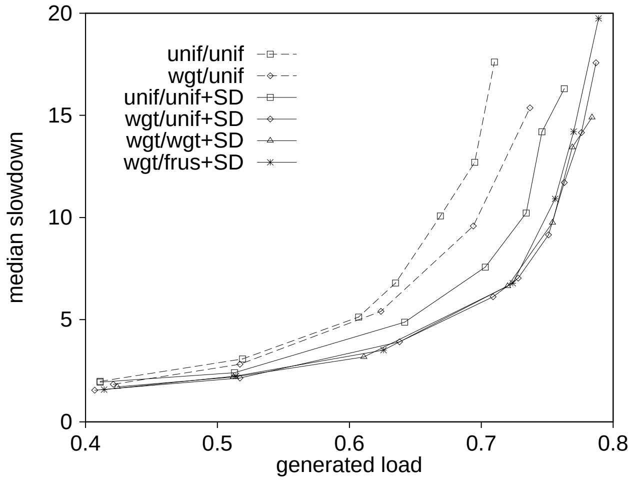

Figure 9: *Division by weights reduces the slowdown, both with and without selective disabling.*

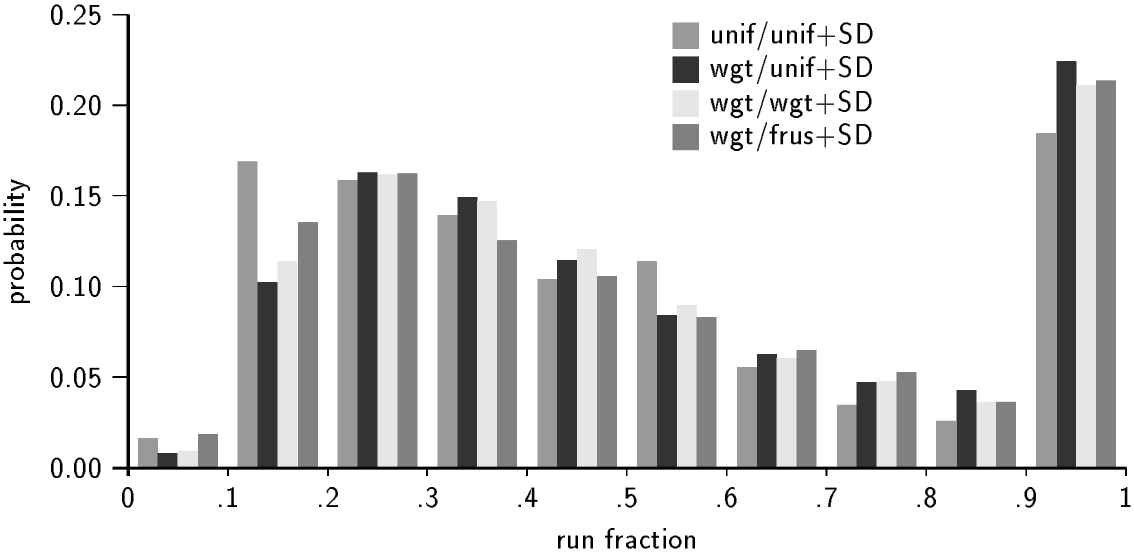

Figure 10: *Distributions of the run fractions created by the four scheduling schemes (generated load of approximately 0.515).*

number of threads in all the gangs controlled by a controller, and s_2 is the total number of threads in gangs controlled by all its subordinates, so $n = 2$. It turns out that with this formulation the weight and frustration approaches are the same.

Based on the suggested approaches, four scheduling algorithms were formulated. These algorithms use different combinations of approaches at the two levels. One is uniform/uniform: this divides the time uniformly among the controller and its subordinates according to the number of gangs they each have, and then divides the time uniformly among the gangs mapped to the controller itself. The other combinations used in the simulations are weight/uniform, weight/weight, and weight/frustration.

The results are as follows: Fig. 9 shows a comparison of the two schemes to divide the time between the controllers, and indicates that division by weights is more effective. It also emphasizes

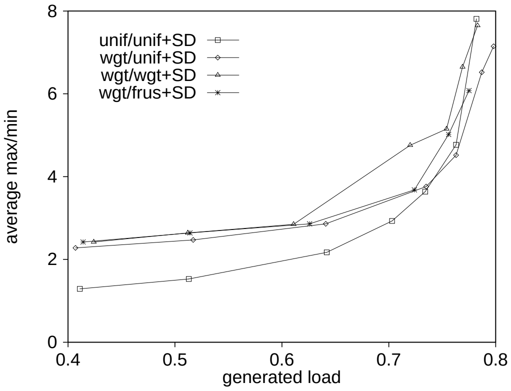

Figure 11: *Average ratio of the maximal to minimal run fractions, over those scheduling times in which two or more gangs are running. Uniform/uniform is fairer for low loads.*

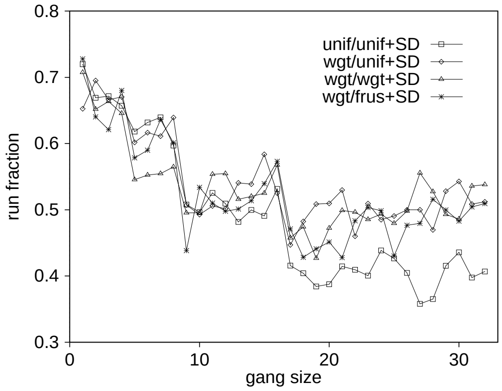

Figure 12: *Run fractions as a function of gang size. Scheduling by weight provides some compensation for the better service that small gangs get due to selective disabling (generated load of about 0.515).*

the importance of selective disabling: starting with uniform/uniform scheduling, the improvement due to selective disabling is twice as large as the improvement gained by switching to weight/uniform scheduling. In addition, the graph shows that the way in which the time is divided between the gangs mapped to the same controller is not very important: the three schemes that use division by weight between controllers and different strategies within a controller all give similar results.

The difference between the schemes is evident also when the distribution of run fractions that they generate is examined (Fig. 10): uniform/uniform scheduling has a higher probability to generate low run fractions. It is interesting to note that uniform/uniform also has a relatively high probability for generating a run fraction of exactly 0.5 (this is tabulated in the .5–.6 interval of the chart). This is evidence to the fact that it actually creates run fractions that are Egyptian fractions, or other fractions with small integers in the numerator and denominator.

The distributions shown in Fig. 10 can also be used to get an idea about fairness. Ideally, the distribution should be a delta function, indicating that all the gangs get the same run fraction. This is of course impossible in a stochastic system, because the load changes in a random manner; gangs that happen to execute in an empty system thus enjoy a run fraction of 1, while gangs that share the resources necessarily get a lower run fraction. A better indication of fairness is therefore the average ratio of the maximal to minimal run fractions that are observed when two or more gangs coexist in the system. This is shown in Fig. 11. Obviously, uniform/uniform division of the time results in better fairness, at least for low loads, because its ratio is lower and closer to a value of 1.

Another aspect of fairness is the question of which characteristics may influence the service that a gang receives. It turns out that small gangs always get a better run fraction than large gangs (Fig. 12); this is a result of using selective disabling. Note that selective disabling more than offsets the advantage that scheduling by weights gives to large gangs, and with uniform scheduling it even causes large gangs to receive a disproportionally low run fraction.

The execution time of a gang, on the other hand, does not influence its run fraction; this means that the response time is linearly proportional to the execution time. This may be expected considering that the simulations are based on the assumption that the scheduling time quantum is significantly smaller than the intervals between consecutive gang creation and gang termination events. If this assumption is removed, we may expect very short jobs to suffer from a low run fraction, because the time they have to wait is long relative to their execution time. Note, however, that the absolute waiting time may still be quite short, being limited to a single scheduling round.

4 Wasted Processing Power Due to Gang Scheduling

The main drawback of gang scheduling is that sometimes there is no set of gangs that can execute side by side and utilize all of the processors. Thus adhering to a strict gang scheduling policy causes processing power to be wasted explicitly by the system[5]. Real implementations can avoid this by using alternative scheduling, that disregards gang affiliation. If the scheduled threads make progress, then no resources are wasted. However, if the application requires gang scheduling in order to make progress, due to interactions among the threads, it will not benefit from such partial scheduling. By analyzing the waste under strict gang scheduling, we find how much resources applications *may* not be able to utilize. The rest is *guaranteed* to be utilized just as if the applications were running on a dedicated machine.

The DHC scheme can only partition the machine into powers of two that correspond to the structure of the tree of controllers. This restriction might increase the waste. In this section we evaluate the expected waste. We first examine optimal, off-line algorithms operating under ideal conditions, but restricted to the same mapping of gangs onto groups of PEs whose number is a power of two as in DHC. This gives an upper bound on the performance that can be expected from the DHC algorithms (or equivalently, a lower bound on the waste). Next we investigate an off-line unrestricted best-fit mechanism and show that the waste in the restricted case can indeed be attributed to the restrictive partitioning of DHC. However, a naive on-line version of the unrestricted best-fit mechanism suffers much more waste than DHC, indicating that restrictive partitioning is a reasonable compromise in the quest for a realistic on-line algorithm.

Gang scheduling is reminiscent of dynamic memory allocation, in the sense that a set of processors must be found to execute a gang, much as a block of memory must be found to satisfy a request for a new segment [32, 16]. The DHC scheme in particular is related to the buddy system method for memory allocation [34]. It is therefore possible to use results about fragmentation in buddy systems to evaluate the waste generated by DHC. However, there is no analogy to selective disabling in the literature.

4.1 Off-Line Algorithms Restricted to DHC-like Partitioning

This subsection derives bounds on the performance of off-line partitioning algorithms, i.e. the algorithm has full knowledge about the sizes of all the gangs in the workload in advance and can organize them at will. Naturally, the off-line algorithm is not a candidate for implementation in a real system, but any algorithm that *is* implemented will suffer from at least as much fragmentation as the off-line algorithm. Thus we obtain a lower-bound on the waste caused by fragmentation under ideal conditions.

[5]We note in passing that it has recently been shown that leaving processors idle is sometimes beneficial also in the context of partitioning with no time slicing [35].

Recall that we assume the gang size is not correlated with the duration of execution. Consequently, the distribution of execution times may be hidden in the distribution of sizes by allowing gangs to be preempted: a gang of size s that executes for t time units is equivalent to t gangs of size s that execute for one time-unit each[6]. It should be emphasized that the workload is given in advance (i.e. all gangs are present at the outset), thus relieving the algorithm from any dependencies on stochastic arrivals. Hence the evaluated waste depends only on the distribution of gang sizes. This is in line with the results of this subsection being an upper bound on the performance that may be expected in a real system, where dependencies on stochastic processes cannot be avoided.

To simplify the analysis, we assume that the distribution of gang sizes is given in the form of a table specifying the number of gangs of each size (thus the table actually represents the probability mass function). For example, a uniform distribution will be specified by a table in which there are equal numbers of gangs of all sizes. We further assume that the numbers in the table are large enough so that end effects may be ignored.

A general bound is obtained for any algorithm that maps each gang to a group of processors, where a group contains a power of two number of processors. Since we assume that the total number of processors is also a power of two, the groups of processors can always be fitted together to allow gangs to execute in parallel. Using bin-packing terminology, the waste is therefore due to internal fragmentation alone, where not all the allocated processors are used.

Uniform distribution

Under the uniform distribution, a large fraction of the gangs have large sizes, generating large wastes. Selective disabling cannot improve the situation by much, as there are not enough small gangs to fill the empty spaces.

Claim 1 *The optimal performance of DHC-like partitioning on a uniform distribution of gang sizes is 25% waste with total disabling and 20% waste with selective disabling.*

Proof sketch The proof is based on a geometrical representation. A gang of size s is represented by a rectangle of height s and unit width (Fig. 13 (a)). A set of gangs with a uniform distribution of sizes can then be represented by the arrangement shown in Fig. 13 (b), which for large systems may be approximated by the triangle of Fig. 13 (c). The area of the triangle represents the processing resources required by all the gangs.

The case of total disabling is shown in Fig. 14. This shows an enumeration of gangs ranging in size uniformly from P (at left) down to 1. When each gang is scheduled, a block of processors that is a power of two has to be allocated. Assuming that the only waste is due to the difference between the gang sizes and these blocks, the waste is represented by the shaded area in the figure. As exactly a quarter of the total area of the blocks is shaded, this implies a 25% waste. This result agrees with the derivations of Peterson and Norman [34] and Russell [36] for internal fragmentation in the binary buddy system memory allocation[7].

[6] This equivalence is strictly valid only if (i) gangs may migrate at run time, and (ii) a gang may spread out on $k \cdot s$ processors and finish in time t/k. While this is an unreasonable requirement, it just makes the bound stronger, so we allow it in this subsection.

[7] They also show how the fragmentation changes between 25% and 33% if the maximal request size is not a power of two.

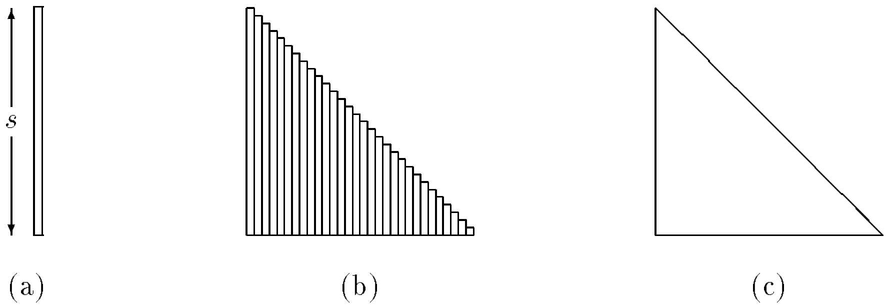

Figure 13: *Geometrical representation of gangs: (a) A gang of size s is represented by a rectangle of height s and unit width. (b) A set of gangs with a uniform distribution of sizes. (c) For large systems a uniform distribution may be approximated by a triangle.*

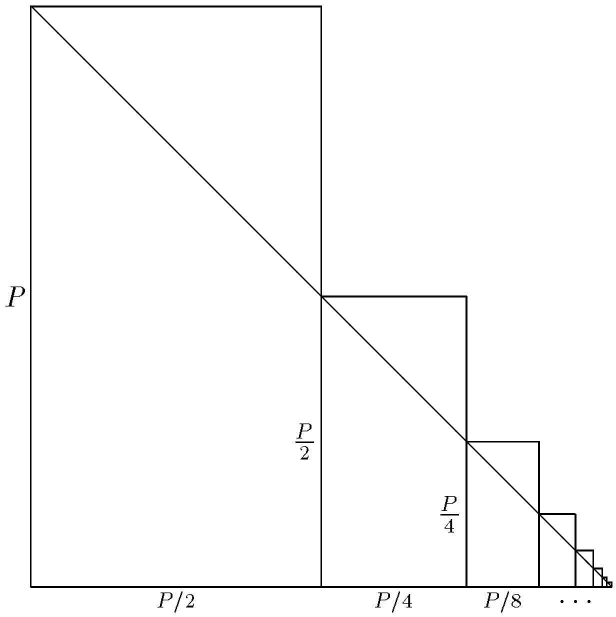

Figure 14: *To aid proof of 25% waste for DHC with total disabling under a uniform distribution.*

The proof for the selective disabling case is similar. The only difference is that small gangs can be executed on processors left over by larger gangs, provided that the difference in sizes is more than a factor of two. An arrangement using this option is shown in Fig. 15. The shaded area in this case is 1/5 of the total, implying 20% waste. ∎

Harmonic distribution

Claim 2 *The optimal performance of DHC on a harmonic distribution of gang sizes is 27.9% waste with total disabling and 10% waste with selective disabling.*

Proof sketch The proof follows the same principle as before. When gangs from a harmonic distribution are enumerated according to size, they may be approximated by the curve $e^{-\alpha n}$ (Fig. 16). To see this, note that when N gangs whose sizes have a cumulative distribution function

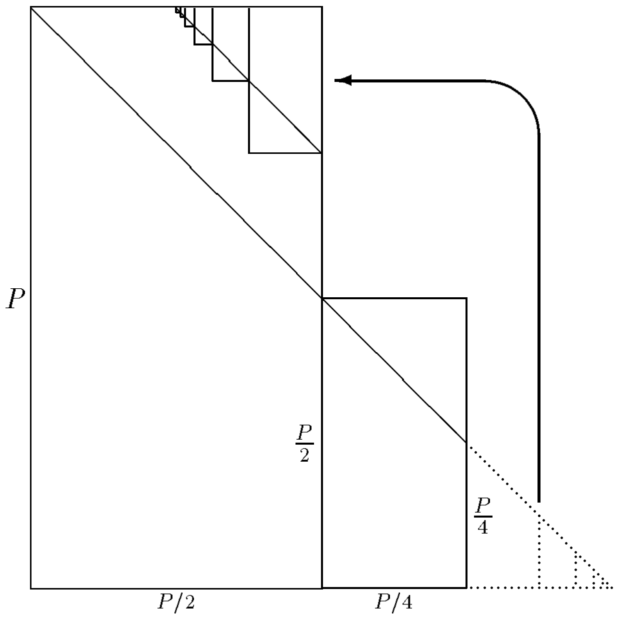

Figure 15: *To aid proof of 20% waste for DHC with selective disabling under a uniform distribution.*

$F(x)$ are sorted from the largest to the smallest, the serial number of a gang with size s is $n(s) = N(1 - F(s))$. For the harmonic distribution, $F(s) = \sigma \ln s$ where $\sigma = 1/\ln P$ is a normalization factor. Inverting the expression, we find that the size of the nth gang is $s(n) = \exp\{(1 - n/N)/\sigma\}$. The approximation is better for the large sizes, which are the more important because of their larger contribution to the waste. Note that when the gangs are grouped by sizes that fall between successive powers of two, there are equal numbers of gangs in each group. Therefore completing each gang to the nearest power of two yields blocks with equal widths. For total disabling, the area above the curve in each block is wasted (shaded in the figure). Evaluating the integral gives $\frac{2\ln 2 - 1}{2\ln 2} \approx 27.86\%$. This result agrees with the derivation by Russell [36] for the binary buddy system with a harmonic distribution; it is also very close to the 28% reported by Shen and Peterson [38] and to the 28–30% reported by Peterson and Norman [34], both for simulations of buddy systems using a truncated exponential distribution.

With selective disabling we have to construct an explicit arrangement as we did for the uniform distribution. This arrangement must abide by the restrictions imposed by DHC, i.e. it must use certain partitions into powers of two. We begin by noting that the third block (with gangs ranging in size from $P/4$ to $P/8$) cannot be accommodated fully in the wasted area of the first block — a width of $x = \frac{2\ln 2 - \ln 3}{\alpha}$ is left over (Fig. 17). Therefore the minimal allocation of resources needed is proportional to the area of the first two blocks plus x of the third (heavy outline in the figure). As we show below, all the smaller blocks can be accommodated in the wasted part (above the curve) of this area, so this allocation is indeed sufficient. The area that is wasted in the end is shaded in the figure. As the requirements are proportional to the total area under the curve, we can now calculate the resulting waste. It turns out to be $\frac{8\ln 2 - \ln 3 - 4}{8\ln 2 - \ln 3} \approx 10.04\%$.

The way to pack the smaller blocks into the wasted parts of the first blocks is shown in Fig. 17. The procedure is recursive, starting from the smallest. Each block is divided into three parts, which are fitted into the wasted area in larger blocks. First, the part of the block starting at x and extending till the end of the block is fitted into the block after next (i.e. the block that contains

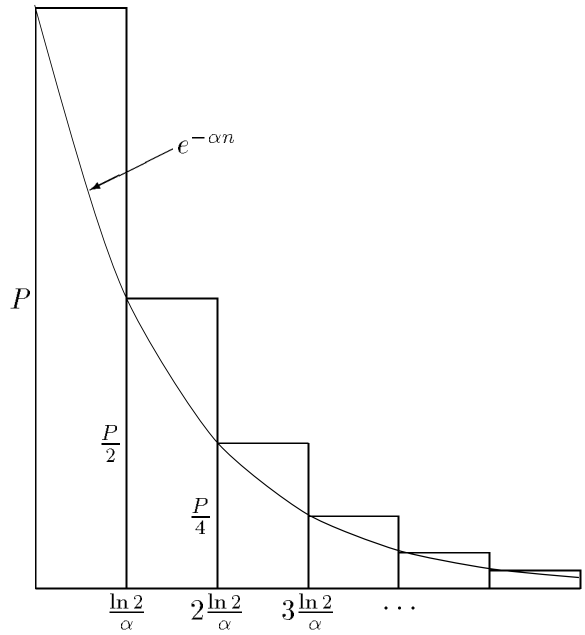

Figure 16: *To aid proof of 27.9% waste for DHC with total disabling under a harmonic distribution.*

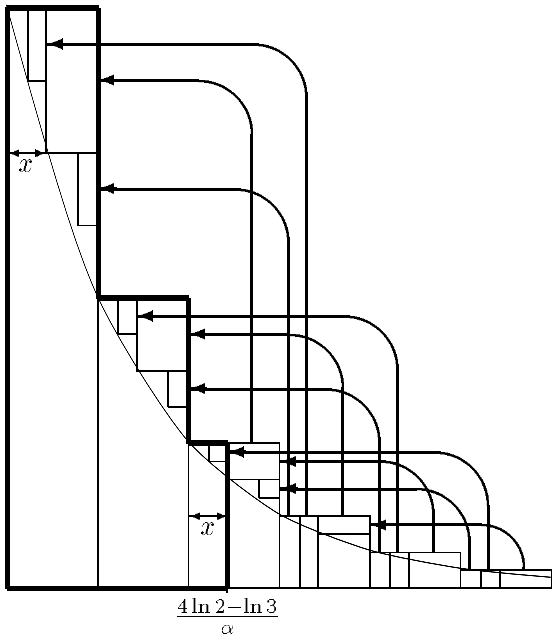

Figure 17: *Packing of smaller blocks into wasted parts of larger ones, generating 10% waste.*

gangs that are four times larger than the gangs in the current block). The x that is left is divided into two, and fitted into the next larger block (with gangs eight times larger than in the current block). As all blocks have the same width and all are divided according to the same proportions, parts of small blocks can be moved like this a number of times, until they end up in the first blocks (i.e. the area marked by the heavy outline). ∎

Uniform distribution over powers of two

In the case of a uniform distribution over powers of two the DHC scheme does not impose any restrictions; the gang sizes are directly mapped onto groups of power-of-two processors. Therefore there is no waste, implying a utilization of 100%. This is a special case of packing items with sizes that divide each other [10].

4.2 Unrestricted Best-Fit Algorithms

To analyze the waste due to the partitioning into powers of two, consider an unrestricted best-fit algorithm that does not abide by such a restriction. We first examine the optimal off-line case, and then a strict on-line case, that services gangs on a first-come first-serve (FCFS) basis. The performance of a realistic algorithm is expected to fall between these two extremes.

Off-line case

It is easy to see that an optimal algorithm producing no waste exists for the uniform distribution: simply match each gang of size s with a gang of size $P-s$. Even if random behavior is allowed, i.e. the gang sizes are chosen according to a uniform distribution but the number of gangs from each size is not necessarily identical, the relative waste can be shown to tend to zero as the number of gangs increases [27]. This is so because the larger the pool of gangs, the easier it is to fit gangs together.

The no-waste result can be generalized for other distributions (including the harmonic) as follows.

Claim 3 *If, for every size s, where $s > 1$, the number of gangs of size s is less than or equal to the number of gangs of size $s-1$, i.e. the probability mass function of the gang sizes is nonincreasing, then an optimal matching with no waste exists.*

Proof sketch Recall that we assume that P is a power of two. The idea is that there are always enough small gangs to pad the holes left by the bigger gangs. More formally, the optimal algorithm proceeds as follows: divide the table of gang sizes into two halves. Match each gang of size s from the top half with a smaller gang of size $2^{\lceil \lg s \rceil} - s$ from the lower half, thus creating a full block which is a power of two (initially it is P). After this step there are no more gangs of sizes larger than $P/2$, and the numbers of those with sizes smaller than $P/2$ are again nonincreasing. Therefore it is possible to repeat this step recursively on the lower half. The sizes of the blocks that are generated are nonincreasing powers of two, so they can be fitted together to create blocks of size P with no waste. The last block may be only partially full, but we assumed the numbers are big enough for this to be insignificant. ∎

Obviously, any distribution over powers of two can also be scheduled with no waste. Thus, for the three distributions we are investigating, unrestricted off-line scheduling achieves full utilization of the resources. We may therefore conclude that the results on waste in Section 4.1 are a direct consequence of restricting the partitioning to powers of two, like in DHC.

Błażewicz et. al. [2] present a linear programming formulation for off-line gang scheduling, thus indicating a polynomial complexity. Hence we actually have a tractable optimal algorithm for any distribution. Note, however, that this algorithm is centralized and off-line, so it is not useful for real systems.

FCFS on-line case

The above analysis of the optimal performance assumed an off-line algorithm. We now turn to the other extreme, i.e. an on-line algorithm that schedules gangs in the order they arrive. Moreover, there is no preemption or migrating gangs from one set of processors to another. As an arriving gang may be larger than any block of available processors, this may create external fragmentation in addition to the internal fragmentation considered before.

The requirement that gangs be scheduled in the order of arrival provides for an independence from the details of the distribution of arrival times. However, it is necessary to account for the distribution of execution times. We consider two cases: unit execution times and execution times that are exponentially distributed. With unit execution times gangs are accumulated until the next

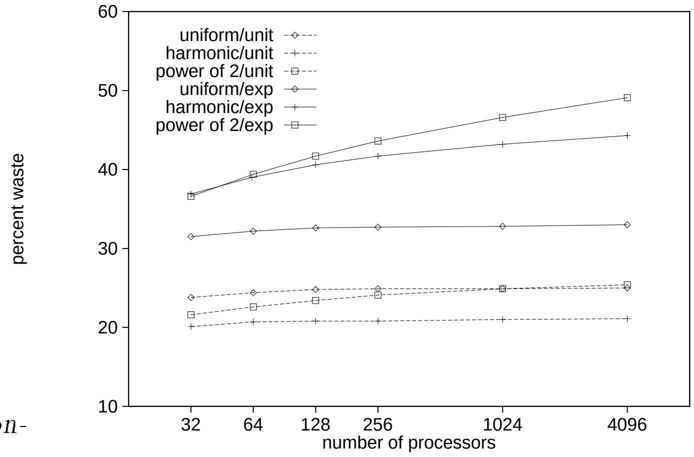

Figure 18: *Results of waste with on-line FCFS best-fit algorithms.*

gang cannot be satisfied, and then a new scheduling round in started. The situation in one round has no effect whatsoever on subsequent rounds. This is equivalent to the "next-fit" bin-packing algorithm [9]. This case is easy to analyze, and provides a lower bound on the fragmentation under FCFS on-line scheduling. When variable execution times are allowed, gangs may retain their hold on a block of processors from one scheduling round to the next. Gangs that terminate may leave a free block of processors that cannot be unified with other blocks, creating a checkerboard effect [8]. This exacerbates the problem of wasted space. We used an exponential distribution with a mean of five time-units as one specific example of this effect, showing that the fragmentation can be considerably worse than for unit execution times.

The analysis is based on simulations: a sequence of gangs with sizes drawn at random from the desired distribution was generated, and these gangs were then accumulated as long as their combined sizes plus waste did not exceed P. When the next gang could not be satisfied, the waste was tabulated and a new round started. For the unit execution times, all one has to do is to sum up the gang sizes, until the next one would cause the total to exceed P. The difference between P and the accumulated sum is the waste, and each new round is a fresh start. The simulation of exponentially distributed execution times simply keeps track of the allocation of blocks to gangs, and maintains a list of free blocks. Free blocks are merged whenever possible. Each new round already has all the gangs from the previous round that have not terminated. The average waste over a large sequence was used to gauge the performance.

The simulation results are shown in Fig. 18. The waste ranges from 20% to 25% for unit execution times. With our example of exponentially distributed execution times, the range is 31% to 37% for 32 processors, and goes up higher than that for certain distributions when more processors are used. For uniform over powers of two, the waste reaches 49% at 4096 processors. This matches results by Krueger et. al., who report a maximum utilization of about 50% for FCFS scheduling of subcubes from a hypercube, when all sizes are equally likely [28].

Comparing this with the result of no waste for the off-line case shows that on-line scheduling is a real challenge. This is especially true in the more realistic case of nonuniform execution times. The checkerboard effect caused by the increased fragmentation as new gangs are fitted into the space left by previous ones results in a severe degradation in performance. This means that nonpreemptive

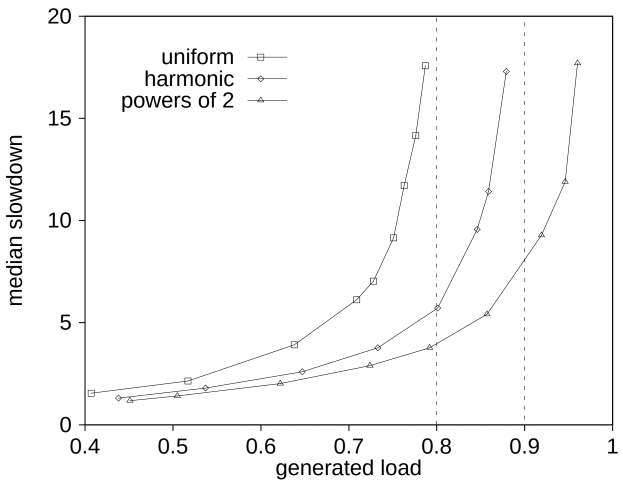

Figure 19: *The effect of the workload statistics on performance. The vertical lines show the maximal utilizations possible for the three distributions (20% waste for the uniform distribution, 10% for the harmonic, and no waste for uniform over powers of 2).*

algorithms would have to perform some sort of compaction and load balancing to be acceptable.

4.3 The optimality of DHC

Most of the DHC simulation results reported in Section 3 were for a uniform distribution of gang sizes. We can compare this with the two other distributions that we have used, namely harmonic and uniform over powers of two. The results are as follows (Fig. 19): a uniform distribution gives the worst performance among the distributions that were checked. This indicates that the results presented in Section 3 should be taken as conservative estimates. The harmonic distribution gives better performance, which is understandable considering that it has a larger percentage of small gangs and it is easier to fit small gangs together. The uniform distribution over powers of two gets much better performance then the other two, because it increases the probability that gangs will fit together.

When compared with the bounds derived in Section 4.1, these results are found to be optimal in the sense that the DHC algorithms approach the performance of optimal off-line algorithms that use the same partitioning into powers of two (Fig. 19). This is especially impressive considering that the DHC algorithms operate in the face of stochastic arrivals, and might waste processing resources unavoidably if there is a gap in the stream of arrivals. The off-line algorithms do not have to contend with such cases. In effect, the DHC algorithms eliminate nearly all the external fragmentation under loaded system conditions, leaving only the internal fragmentation that is inherent for each distribution of gang sizes.

Moreover, the DHC algorithms outperform the on-line FCFS best-fit scheduling with unrestricted partitioning. This result may be traced to the use of preemption with time slices determined by weights, thus giving gangs that generate less waste a higher priority. In other words, clever modifications of time quantums allow us to avoid the problematic migration of tasks that would otherwise be necessary to achieve acceptable performance in the face of fragmentation.

5 Conclusions

General-purpose, multiuser, interactive systems are a promising direction for the further development of parallel systems. A basic issue in such systems is the partitioning of the machine between the users; both time slicing (as in conventional uniprocessors) and space slicing (i.e. the allocation of distinct processors to different users) can be used. We showed how both methods can be combined efficiently using a Distributed Hierarchical Control scheme, which is scalable to very large machines. This provides support for gang scheduling, which in turn provides a convenient interactive execution environment to applications.

Note that the whole discussion does not mention the architecture of the parallel computer itself, nor the model that the user sees. This results from the fact that the relevant operating system functions are orthogonal to the architecture, and are relevant for any parallel computer. In particular, it should be stressed that the hierarchical structure is in the control, not in the processors. Thus we achieve the flexibility and scalability of cluster machines without forcing a nonuniform model on the user. On the other hand, if the architecture is indeed nonuniform, this does not invalidate our algorithms. To the contrary, the tree structure of the controllers can guarantee the best possible locality properties, by matching it to the architecture. Thus in a clustered architecture subtrees would correspond to clusters, and in a hypercube each level of the tree would correspond to a dimension.

The main problem in supporting gang scheduling is the coordination of the multi-thread-switching. These should be synchronized across the relevant processors, but should not cause redundant dependencies when small gangs are involved. Our solution is based on a hierarchy of controllers, that provide the desired degree fo control for different sized gangs. The price is that we use a predefined partitioning into powers of two. This was shown to result in a loss of 5–25% of the resources to fragmentation, depending on the distribution of gang sizes. An optimal off-line algorithm that is not restricted to use such a partitioning can find a schedule with no waste.

However, real systems must be based on realistic, on-line algorithms. The simple approach to gang scheduling is to use a non-preemptive FCFS scheduler, like those used in systems that provide pure space slicing. In one example we checked, such an algorithm suffered from 31–49% loss to fragmentation, depending on the gang-size distribution and on the system size. Our preemptive algorithm is much more efficient. Moreover, we have shown that asymptotically it achieves the same utilization as an off-line algorithm that is restricted to use the same partitioning into powers of two. Consequently, within the framework of such a partitioning, our algorithms are optimal and leave no space for improvement. In particular, the use of preemption eliminates the need to implement runtime migration of threads as a means to counter external fragmentation.

Appendix: Simulation Methodology

The following list of assumptions underlies the algorithms and simulations presented in Section 3. The simulation assumes that the scheduling time quantum is significantly smaller than the interval between gang creation and termination events, so that all the gangs get to run in the manner specified by the scheduling policy. This assumption simplifies the simulation and removes the scheduling time quantum from the list of parameters.

The execution times of the gangs are drawn from an exponential distribution with a mean of 1. The interarrival times are exponentially distributed as well, and their mean is changed so as to create various loading conditions. The number of processors in the simulations was set at 32; simulations for higher numbers required too much time. The only exception were a few runs with 128 processors, used to validate the results of the load balancing experiments. The system performance is measured as a function of the generated load, which is given as the fraction of the available resources consumed. This is also the average non-idle time of each processor; it is calculated by the expression

$$\frac{(\text{average gang size}) \times (\text{average execution time})}{(\text{number of processors}) \times (\text{average interarrival time})}$$

Each combination of load balancing, mapping, and scheduling schemes was simulated for several load conditions, which were generated by changing the average interarrival time. In each simulation, the following information was collected: histogram of gang sizes; distribution of loads; distribution of run fractions; run fractions as a function of gang size; and run fractions as a function of execution time. The slowdown is the reciprocal of the run fraction, so information about it is readily obtained. We use the median slowdown in the presentation of the results, rather than the average. The reason is that the median slowdown and run fraction correspond to each other, while the average slowdown and run fraction do not. Thus using averages tends to distort the picture, painting it in rosy colors when run fractions are used, but in gray when slowdown is used.

The length of the simulation runs was long enough so that the differences between the average slowdown on independent runs was less than one percent for low loads. The difference may be substantially larger for high loads, but this does not effect the quality of the results because the slowdown increases sharply in that region. A single long run for each configuration was preferred over a number of independent short runs, because full distributions and not just averages were collected.

It should be noted that the simulations presented here are different from those presented in a previous paper [16]. Those simulations provided an average over multiple *static* configurations with exactly the same load, whereas the current simulations model the dynamic changes in system configuration that occur when new jobs are submitted and old ones terminate.

References

[1] 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.

[2] J. Błażewicz, M. Drabowski, and J. Węglarz, "*Scheduling multiprocessor tasks to minimize schedule length*". *IEEE Trans. Comput.* **C-35(5)**, pp. 389–393, May 1986.

[3] P. Brinch Hansen, "*An analysis of response ratio scheduling*". In *IFIP Congress, Ljubljana*, pp. TA–3 150–154, Aug 1971.

[4] R. H. Campbell, N. Islam, and P. Madany, "*Choices, frameworks and refinement*". *Computing Systems* **5(5)**, pp. 217–257, Summer 1992.

[5] N. Carriero and D. Gelernter, "*How to write parallel programs: a guide to the perplexed*". *ACM Comput. Surv.* **21(3)**, pp. 323–357, Sep 1989.

[6] E. M. Chaves Filho and V. C. Barbosa, "*Time sharing in hypercube multiprocessors*". In 4th *IEEE Symp. Parallel & Distributed Processing*, pp. 354–359, Dec 1992.

[7] A. N. Choudhary, J. H. Patel, and N. Ahuja, "*NETRA: a hierarchical and partitionable architecture for computer vision systems*". *IEEE Trans. Parallel & Distributed Syst.* **4(10)**, pp. 1092–1104, Oct 1993.

[8] E. G. Coffman, Jr., "*An introduction to combinatorial models of dynamic storage allocation*". *SIAM Rev.* **25(3)**, pp. 311–325, Jul 1983.

[9] 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.

[10] 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.

[11] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "*Adaptive load sharing in homogeneous distributed systems*". *IEEE Trans. Softw. Eng.* **SE-12(5)**, pp. 662–675, May 1986.

[12] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "*Speedup versus efficiency in parallel systems*". *IEEE Trans. Comput.* **38(3)**, pp. 408–423, Mar 1989.

[13] P. A. Emrath, M. S. Anderson, R. R. Barton, and R. E. McGrath, "*The Xylem operating system*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 67–70, Aug 1991.

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

[15] 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.

[16] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[17] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[18] D. G. Feitelson and L. Rudolph, "*Mapping and scheduling in a shared parallel environment using distributed hierarchical control*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 1–8, Aug 1990.

[19] 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.

[20] B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, Aug 1995.

[21] J. L. Gustafson, "*Reevaluating Amdahl's law*". *Comm. ACM* **31(5)**, pp. 532–533, May 1988.

[22] W. Händler, A. Bode, G. Tritsch, W. Henning, and J. Volkert, "*A tightly coupled and hierarchical multiprocessor architecture*". *Computer Physics Communications* **37(1–3)**, pp. 87–93, Jul 1985.

[23] D. S. Henry and C. F. Joerg, "*A tightly coupled processor-network interface*". In 5th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 111–122, Sep 1992.

[24] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo, "*A scalable time-sharing scheduling for partitionable, distributed memory parallel machines*". In 28th *Hawaii Intl. Conf. System Sciences*, vol. II, pp. 173–182, Jan 1995.

[25] Intel Supercomputer Systems Division, *Paragon User's Guide.* Order number 312489-003, Jun 1994.

[26] B. S. Joshi, S. H. Hosseini, and K. Vairavan, "*A methodology for evaluating load balancing algorithms*". In 2nd *Intl. Symp. High Performance Distributed Computing*, pp. 216–223, Jul 1993.

[27] R. M. Karp, M. Luby, and A. Marchetti-Spaccamela, "*A probabilistic analysis of multidimensional bin packing problems*". In 16th *Ann. Symp. Theory of Computing*, pp. 289–298, 1984.

[28] 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.

[29] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S-W. Yang, and R. Zak, "*The network architecture of the Connection Machine CM-5*". In 4th *Symp. Parallel Algorithms & Architectures*, pp. 272–285, Jun 1992.

[30] V. M. Lo, "*Heuristic algorithms for task assignment in distributed systems*". *IEEE Trans. Comput.* **37(11)**, pp. 1384–1397, Nov 1988.

[31] C. Maples, W. Rathbun, D. Weaver, and J. Meng, "*The design of MIDAS — a modular interactive data analysis system*". *IEEE Trans. Nuclear Sciences* **NS-28(5)**, pp. 3746–3753, Oct 1981.

[32] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[33] C. H. Papadimitriou and M. Yannakakis, "*Towards an architecture-independent analysis of parallel algorithms*". *SIAM J. Comput.* **19(2)**, pp. 322–328, Apr 1990.

[34] J. L. Peterson and T. A. Norman, "*Buddy systems*". *Comm. ACM* **20(6)**, pp. 421–431, Jun 1977.

[35] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy, "*Analysis of non-work-conserving processor partitioning policies*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 165–181, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[36] D. L. Russell, "*Internal fragmentation in a class of buddy systems*". *SIAM J. Comput.* **6(4)**, pp. 607–621, Dec 1977.

[37] K. C. Sevcik, "*Characterization of parallelism in applications and their use in scheduling*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 171–180, May 1989.

[38] K. K. Shen and J. L. Peterson, "*A weighted buddy method for dynamic storage allocation*". *Comm. ACM* **17(10)**, pp. 558–562, Oct 1974.

[39] P. Steiner, "*Extending multiprogramming to a DMPP*". *Future Generation Comput. Syst.* **8(1-3)**, pp. 93–109, Jul 1992.

[40] H. Sullivan, T. R. Bashkow, and D. Klappholz, "*A large scale, homogeneous, fully distributed parallel machine, II*". In 4th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 118–124, Mar 1977.

[41] J. A. Test, "*Multi-processor management in the Concentrix operating system*". In *Proc. Winter USENIX Technical Conf.*, pp. 173–182, Jan 1986.

[42] D. L. Tuomenoksa and H. J. Siegel, "*Task scheduling on the PASM parallel processing system*". *IEEE Trans. Softw. Eng.* **SE-11(2)**, pp. 145–157, Feb 1985.

[43] A. M. van Tilborg and L. D. Wittie, "*Wave scheduling — decentralized scheduling of task forces in multicomputers*". *IEEE Trans. Comput.* **C-33(9)**, pp. 835–844, Sep 1984.