Scheduling Parallel Jobs on Partitionable Parallel Computers

Allen B. Downey
downey@cs.berkeley.edu

Qualifying Exam Proposal

Computer Science Division
Department of Electrical Engineering
and Computer Science
University of California at Berkeley

Abstract

Scheduling jobs on parallel computers has turned out to be a more difficult problem than uniprocessor scheduling. In this paper I will explain some of the reasons for this difficulty, and examine an approach to parallel scheduling that is likely to be an essential part of parallel systems — space-sharing.

I will outline the design space of space-sharing schemes, and suggest several research projects intended to explore this space. The thesis of this work is that the most important goal of a parallel scheduler should be to balance the local goal of users — to minimize the response times of their jobs — with global goals that insure good performance, fairness, and good resource utilization. I will propose a formulation of this problem and an approach to its solution.

1.0 Introduction

A universal problem in resource allocation is the conflict between individual users trying to maximize their use of a resource, and global limitations on the availability of that resource. In the context of processor allocation — where the resource is a shared pool of processing elements — this conflict leads to a “tragedy of the commons,” in which users competing for a shared resource overexploit it and cause a global degradation in service.

In this paper, I will discuss one form of resource sharing — space-sharing on parallel computers — and suggest that finding a solution to the "tragedy of the commons" will be necessary to insure the good performance and long-term usefulness of space-sharing systems. I will also propose a research project intended to find such a solution.

Since every word of my title is overloaded (except "on"), I will start by explaining what I mean by each of them:

1.1 Scheduling ...

Research in parallel scheduling is divided along two axes: the job-arrival axis and the prior-information axis [Sevcik94]. Two common ways to formulate arrivals are

- batches: the scheduler is given a set of jobs to assign to processors, usually with the goal of minimizing the *makespan* of the set (the time from the beginning of the first job to the completion of the last).
- intermittent: jobs arrive from outside the system at unpredictable intervals

Common assumptions about *a priori* information are:

- none: the system knows nothing about the runtime or behavior of a job until it begins execution
- stochastic: based on the name of a job, user-provided information, or some other source, the system knows something about the expected behavior of a job
- deterministic: the system has accurate prior knowledge about jobs; for example, their runtimes and dependencies.

The most densely-populated corner of this space is probably batch scheduling of jobs with deterministic information. It is often formulated as a bin-packing problem, and for interesting formulations, it is NP-complete. This paper does not address this problem; [NormanThanisch93] does.

This paper is limited to scheduling techniques that are more directly applicable to real systems, which usually implies intermittent arrivals and little, if any, *a priori* knowledge about jobs.

Although the goal of such schedulers is usually to optimize the overall performance of the system, they are seldom based on a formal optimization problem because (1) there are often multiple metrics of system performance that cannot be optimized simultaneously, (2) the overhead of making optimal decisions is prohibitive — often it is better to make a fast decision than an optimal decision, and (3) the information necessary to make optimal decisions is usually not available.

1.2 … Parallel …

I will be considering parallel applications that are based on a SPMD programming model and targeted for distributed-memory architectures. Scheduling issues for these systems are different from those for shared-memory architectures using a workpile-of-chores programming model.

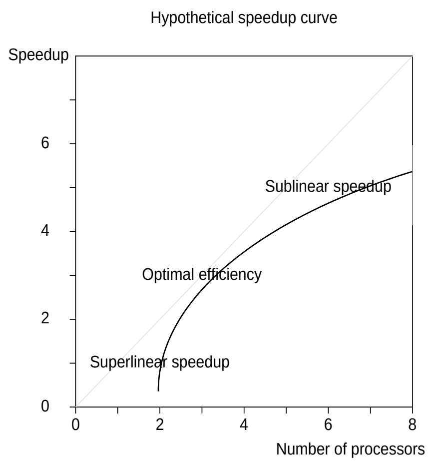

Figure 1: Hypothetical speedup curve for parallel jobs.

The number of processors used by a SPMD application is usually determined at compile time (hardcoded) or during an initialization phase when the program runs (adaptable). Although it is conceivable that the system might change the number of processors allocated to a job during the run, it would complicate the programming model and introduce considerable overhead (moving process state among processing elements, redistributing data, etc.). As far as I know, no such systems exist.

Some work has been done on scheduling independent (sequential) tasks on parallel computers [Polychronopoulos89, NarahariKrishnamurti93], but I am interested in cooperating processes that synchronize and communicate. In general, these jobs have sublinear speedup as the number of processors increases, which implies that their efficiency decreases.

In addition, at least some of these jobs require large amounts of memory — larger than the memory available on a single processor. Thus, there is a lower limit on the number of processors a job can run on before memory thrashing degrades its performance unreasonably.

1.3 … Jobs …

On both parallel and sequential machines, the scheduler faces a conflict between performance objectives for batch jobs and for interactive jobs. Batch processing systems are usually tuned to minimize makespan, the elapsed time required to complete a given set of jobs. Interactive systems try to minimize response time, usually by some form of preemption that gives priority to short jobs. On sequential machines, these conflicting goals can be reconciled with time-sharing systems using multi-level queues.

On parallel machines, the conflict between interactive and batch jobs is complicated by the possibility of adaptable jobs. For example, suppose two jobs with the speedup curve shown in Figure 1 are submitted to a system with 8 processors. Each job in isolation would choose to run on all 8 processors, in order to maximize speedup (minimize run time).

An interactive system might run both jobs on 8 processors and time-slice in round robin fashion. The advantages of this approach would be that (1) the mean response time for the two jobs would be near optimal, and (2) the idle time of one job (e.g. waiting for I/O) could be filled with useful work on the other job. The disadvantages of this approach would be the overhead of time-slicing (which is far greater on a distributed machine than on a uniprocessor) and the efficiency penalty the jobs incur by running on a larger number of processors.

A batch system would prefer to run each job on four processors, in order to maximize the utilization of the system (both jobs run more efficiently with 4 processors). For computation-bound jobs, this schedule would minimize the makespan of the two jobs.

1.4 … Partitionable …

Many existing parallel computers can be partitioned into (usually) disjoint sets of processors so that more than one job can run on the machine simultaneously, each in its own partition. Section 2 presents a taxonomy of partitioning schemes.

1.5 … Parallel Computers

As mentioned above, I will be considering distributed-memory parallel computers, including networks of workstations. Scheduling for shared-memory and SIMD machines is sufficiently different that it falls outside the scope of this paper.

1.6 Organization of this paper

The next section presents a taxonomy of partitioning schemes. Section 3 presents two research topics I propose to undertake.

2.0 Taxonomy of partitionable computers

I will distinguish between two ways of allocating the processing elements of a parallel computer — partitioning and space sharing. Partitioning is a long-term division of a parallel computer into disjoint sets of processors, often with distinct roles assigned to each partition. Space-sharing is the short-term creation of clusters of processors and the allocation of these processors to jobs.

TABLE 1.

Partitioning	Space-sharing
partitions persist across many job executions	space-sharing is done on a per process basis and does not persist
partitions are created explicitly by either a system administrator or a user who allocates specific processors	space-sharing is done implicitly when a job is created; the job does not need to know which processors are allocated
partitions often have different roles and characteristics: e.g. a login and a computation partition or a batch and an interactive partition	allocations within a partition — clusters — are homogeneous
partitions are almost always disjoint	clusters need not be disjoint; conflicts can be resolved by preemption with priorities or by timesharing
maintaining independence among partitions is comparatively easy and often done in hardware	maintaining independence among clusters is harder and usually requires a software mapping between virtual and physical PE numbers

Some existing systems provide only partitioning, some provide only space-sharing, and some provide both. The distinction between them is not clear in the literature, and the nomenclature is not consistent. For example, on the Cray T3D what I would call a partition is called a "pool," and clusters within the pool are called "partitions." In much of the academic literature, partitioning and space-sharing are used interchangeably.

I think that it is worth making the distinction for the following reasons:

- Because partitions are usually implemented in hardware and created explicitly by a system administrator, the scheduler has no control over them and no need for complex allocation algorithms. In contrast, space-sharing is done implicitly by the operating system, and raises many opportunities for the scheduler to improve (or hinder) system performance.

- The design decisions that affect partitioning are different from those that affect space-sharing. The next section discusses these decisions in more detail.

2.1 Partitioning

Partitioning schemes differ regarding (1) when partitions can be created, (2) who can create them, (3) what roles the partitions play, and (4) what restrictions there are on the shape or size of partitions.

Machines with *fixed partitioning* can only be partitioned in hardware; in order to be reconfigured, they have to be rebooted (e.g. the CM-5 [Leiserson92]). Machines with *flexible* or *software partitioning* can be reconfigured without restarting the machine, either by a system administrator or by users. Two examples of flexible partitioning are:

- On the Cray T3D [Oed95], the system administrator typically reconfigures the machine twice daily in order to dedicate more processors to batch processing at night and interactive processing during the day.
- On HPC/VORX [KatseffGaglianelloRobinson90] users can allocate partitions for indefinite lengths of time in order to guarantee repeatable performance over multiple runs.

On most machines, partitions have different characteristics and different roles. For example,

- The BBN TC2000 [GordaBrooks91] is divided into a login partition, which handles interactive, sequential jobs, and a compute partition, which handles parallel jobs (and long-running, sequential jobs).
- The Cray T3D [Oed93] and the Intel Paragon [Intel95] provide partitions with different scheduling behavior tailored for batch jobs, interactive jobs with little communication/synchronization, and interactive jobs that require gang scheduling.

Some machines impose restrictions on the ways the machine can be partitioned. These restrictions are usually related to the topology of the interconnect network.

- The Meiko CS-2 [Feitelson95] and the Intel Paragon [Intel95] allow arbitrary partitions. Non-contiguous partitions, however, can cause messages in one partition to interfere with messages from another.
- The Connection Machine CM-5 [Leiserson92] and the Cray T3D allow contiguous partitions that contain a number of processors that is a power of two.

- Hypercubes (e.g. the Intel iPSC/2 [Pierce88]) can generally be divided recursively into subcubes.

2.2 Space-sharing

The three flavors of space-sharing are *static*, *adaptive* and *dynamic*:

- static: incoming jobs request a certain number of processors. If enough processors are available, they are allocated to the job; otherwise, either the job is queued, or it will share processors with another job.
- adaptive: incoming jobs can run on a range of cluster sizes; the number of processes assigned depends on the state of the system (e.g. current load), but does not vary while the job is running.
- dynamic: the number of processes assigned to a job can change over the lifetime of the job.

Adaptive space-sharing requires that at least some of the jobs be able to configure themselves for various numbers of processors. Although this requirement complicates the programming model and imposes some run-time costs, it is not infeasible. The requirement for dynamic space-sharing, that jobs be able to reconfigure themselves at run time, is probably not feasible within a SPMD programming model. This paper does not address dynamic space-sharing.

The advantages of adaptive over static space-sharing are

- improved efficiency, because jobs will tend to run on fewer processors
- reduced external fragmentation
- reduced queue times

The advantages of static space-sharing are a simpler programming model and the optimization opportunities created by hardcoded parallelism.

In any of these schemes, the number of processors may be unrestricted or it may be limited to particular values, for example powers of two. The advantages of arbitrary cluster sizes are

- the elimination of internal fragmentation
- better match between problem topology and cluster topology

The advantages of restricted cluster sizes are

- accommodating the topology of the interconnect simplifies the network hardware and the communication protocols

- contiguity restrictions make it easier to prevent activity in one cluster from affecting others
- power-of-two restrictions simplify the programming model and allow optimizations like power reduction

2.3 Multi-level systems

Some machines provide both partitioning and space-sharing. For example, the Cray T3D has partitions intended to separate batch and interactive jobs, and does static space-sharing within partitions.

Similarly the Meiko CS-2 has four types of top-level partition, with static space-sharing within partitions. The Intel Paragon has long-term partitions created by the system administrator, short-term subpartitions created by users, and static space-sharing within both partitions and subpartitions.

3.0 Research proposal

The next two sections discuss the research projects I would like to undertake. The first is a short-term project which is already underway and which I want to finish this summer. The second is a somewhat grander effort which is still a bit sketchy. I think, however, that it has the promise of leading to an interesting and important thesis.

3.1 Performance implications of partitioning restrictions

Various network configurations (or software designs) impose restrictions on the ways a parallel computer can be partitioned. The diversity of these restrictions in existing systems implies that there are unanswered questions about their implications:

- What performance advantage do flexible space-sharing schemes provide compared with restricted space-sharing schemes?
- What are the performance costs of providing flexible space-sharing; e.g. hardware costs, more complex communication/synchronization protocols?

Flexible space-sharing provides better support for a programming model with fixed (hardcoded) parallelism; restricted space-sharing schemes suffer from internal fragmentation unless at least some of the jobs have adaptable parallelism.

Thus, a necessary step in assessing the implications of restricted space-sharing will be to characterize the distribution of processor requests in real workloads. One of the designers of the Cray T3D reports [Oed95]:

> [On the Cray T3D] a partition must be a power of 2. We were planning to implement "software partitions" to alleviate that restriction, however [we] have abandoned this since it would impose too much overhead (any communication would require a system call) and our customers do not see a restriction in power of 2 partitions (it seems even users tend to think in powers of 2).

This claim is contradicted by data from the BBN TC2000 at LLNL, which provides arbitrary cluster sizes. Figure 2 shows[1] that although many processor requests are for powers of 2, many others are not. For example, there are numerous requests for perfect squares (9, 25, 36, 49, 81) and powers of two plus one (where the extra processor is used as a control node). These latter requests would suffer horribly in the hands of a power-of-two allocation scheme.

These data suggest that the cost of space-sharing restrictions might be large. On the other hand, they also suggest an additional avenue of research …

3.1.1 Tailoring a scheduling policy to the request distribution

Figure 2 is quite spiky, indicating that there are only a few common request sizes, with few jobs requesting other sizes. The scheduler might do well to take advantage of this distribution by restricting cluster sizes accordingly.

For example, on some machines that allow arbitrary cluster sizes it is either required or preferable to allocate contiguous clusters. In this case, it may not be possible to satisfy a request even if there are enough idle processors, due to external fragmentation. It might be possible to reduce this fragmentation by limiting clusters to a few common sizes.

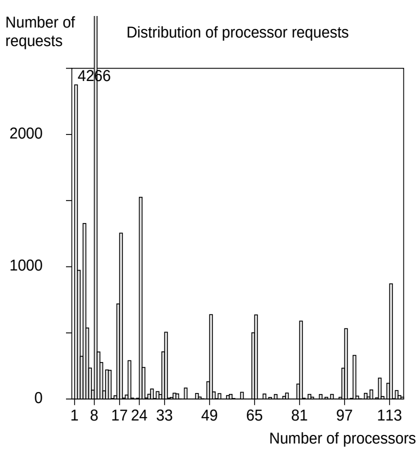

Figure 2: Distribution of ~20000 processor requests on a BBN TC2000.

For example, on a 64-processor machine, the most common requests might be:

> 64, 50, 49, 32, 26, 25, 16, 9, 8, 4, 2

It might be wise to reduce the number of legal partitions by rounding requests for 8, 25, or 49 processors up by one:

> 64, 50, 32, 26, 16, 9, 4, 2

This a set of legal cluster sizes suggests an algorithm for breaking a pool of processors into clusters and allocating processors to jobs. I am currently working out the details of such an algorithm and implementing it in a simulator.

1. This figure is really small because I only got this data by promising that I wouldn't give it away, and I'm not sure that publishing it is really OK.

Using the set of legal cluster sizes and the distribution of requests, it is easy to calculate the mean internal fragmentation for all jobs. The question, then, is whether a scheme like this would reduce external fragmentation enough to justify the internal fragmentation.

3.1.2 Research proposal

Using a simulator and the distribution of requests in Figure 2, I would like to compare the system performance of different restrictions on cluster sizes:

- arbitrary sizes
- powers of two
- a set of sizes derived from the distribution

Some of the parameters of the simulation are:

- arrival process of incoming jobs
- distribution of run times, including correlation with request size
- whether or not clusters must be contiguous

Performance metrics include:

- average internal fragmentation (system average and per job average)
- external fragmentation (system average)
- system utilization

I have started building this simulator; I hope to have some results before my qualifying exam.

3.1.3 Thesis

My research will address the following questions

- what are the costs and benefits of restrictions on space-sharing?
- how can parallel schedulers accommodate the demands of parallel jobs given the limitations of an underlying interconnect (both the network hardware and the communication software)?

The aspects of this research that are novel are

- the use of empirical workload characteristics to evaluate scheduling strategies
- the attempt to improve system performance by tailoring the scheduling strategy to the workload

3.1.4 Related work

Little work has been done on empirical descriptions of parallel workloads, especially for distributed-memory architectures (in fact, few distributed-memory machines are sufficiently well-established that they are supporting workloads that could be considered representative). A forthcoming paper that sounds useful is, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860" ([Feitelson-Nitzberg95]).

The analogy between processor allocation and memory allocation has been around since the beginning of time [Ousterhout82], as have buddy systems for memory allocation [PetersonNorman77]. Some proposed systems have combined these ideas [TilborgWittie84, FeitelsonRudolph90, among others], but none have suggested tailoring the allocation strategy to the distribution of requests.

3.2 Selfish users and degenerate space-sharing

In the previous section, I assumed that the distribution of cluster sizes was a given; i.e. that the cluster size for each application was fixed and that the distribution only depended on the job mix. In reality, the distribution of cluster sizes is influenced by the characteristics and performance of the system. A trivial example of this effect is that machines with restricted cluster sizes see a very different distribution of cluster sizes than machines with arbitrary clusters (see Figure 2).

In addition, there are more interesting mechanisms at work, which I call adaptation and selection:

- adaptation: users will modify their programs to match the characteristics of the system. For example, if the programming model supports applications with parametric parallelism, users will adapt the size of their jobs in response to system load and scheduling policy.
- selection: programs that cannot adapt to the system, and thus receive poor service, will either be abandoned, refashioned, or ported to a more appropriate environment. For example, large, long-running jobs running on a busy, interactive machine may be starved; thus there are likely to be few such jobs on that machine.

These feedback effects should be a source of concern for designers of parallel schedulers. I believe that for many systems, these effects will eventually drive the system toward a degenerate allocation of processors, and degrade overall system performance.

For example, many existing space-sharing strategies implicitly favor jobs with small cluster sizes, since it is more likely at a given time that there will be a small number of free processors. As system load increases, users may find that requests for more than a few processors face unacceptable queueing delays (starvation). Thus, the distribution of requests might become skewed toward small clusters.

The problems with this outcome are

- it defeats the purpose of parallel programming, which is to escape the ultimate limitations of sequential computing
- it excludes a class of parallel jobs that, because of memory requirements, cannot run in a small cluster

Correspondence with one of the designers of the Meiko CS-2 confirms that they are experiencing this problem ([Roweth95]). The solutions they are considering are (1) limiting the range of cluster sizes for a given partition, thus controlling the distribution of cluster sizes by fiat, and (2) increasing the priority of jobs according to their queue time, thus eliminating starvation.

I believe that this second approach is likely to lead to an equally pathological outcome — a tragedy of the commons. If the scheduler goes too far in alleviating the disincentive for large jobs, users trying to minimize run times will be inclined to submit requests for large clusters, most likely the entire machine. This scenario would be doubly unpleasant, since:

- queue times would be excessive, because every job would allocate the entire machine
- system utilization would be low, because jobs would execute with minimal efficiency

Game theory enthusiasts will note that this pessimal outcome is stable; i.e. although all players suffer, none have any incentive to change their behavior unilaterally.

3.2.1 Perversity and greed

In other words, such scheduling strategies create perverse incentives: users acting in their (narrow) best interest will degrade not only global performance, but their own performance as well. I will define the following property of parallel schedulers:

A scheduling strategy is perverse *(for a given load) if in the presence of omniscient, greedy users the system reaches a non-optimal, degenerate equilibrium.*

By "degenerate equilibrium", I mean that either all jobs run on one processor or all jobs allocate the entire machine. By "omniscient", I mean that users know the speedup curves of their applications (see Figure 1) and have some information (maybe stochastic) about queueing times. That is, users know $R(p)$, the run time of a job as a function of the number of processors, and $Q(p)$, the queue time until n processors are free. By "greedy", I mean that for a given job and system state, users will choose the number of processors that minimizes $Q(p) + R(p)$.

A degenerate equilibrium is not necessarily suboptimal. Trivially, if the workload consists entirely of sequential jobs, allocating one processor per job is optimal. Also, if the rate of arrivals is so low that $Q(p)$ is always zero, all jobs *should* allocate the entire machine. It is, therefore, only meaningful to say that a scheduling strategy is perverse if it reaches a degenerate equilibrium that is non-optimal.

Of course, from the point of view of designing a space-sharing strategy, workloads with optimal degenerate solutions are of no interest. If a workload is best served by allocating all the processors to each job, then we shouldn't be doing space-sharing in the first place. If a workload is best served by allocating one processor per job, then we shouldn't be doing parallel programming. Hereafter, I will assume that for any relevant workload, the optimal space-sharing schedule is non-degenerate.

So far, I have not defined what I mean by optimal. There are at least two likely choices: maximum system utilization and minimum mean normalized response time. In this context, utilization is the average over processors and time of $e_{j(i,\,t)}(p)$ where $j(i, t)$ is the job running on processor i at time t, and $e_j(p)$ is the efficiency of job j running on p processors. Normalized response time is

$$\frac{R(p) + Q}{R(p^*)} \qquad \text{(EQ 1)}$$

where p is the number of processors allocated to the job, Q is the time the job spends in queue, and p^* is the some number of processors chosen as a reference point (usually 1 or, for processors with superlinear speedup, the number of processors that maximizes efficiency).

3.2.2 Research proposal

Many current parallel workloads are degenerate, and would derive no benefit from space-sharing. Future workloads are more likely to be non-degenerate because

- workloads will be high enough that queue times are significant, and
- at least some jobs will have memory requirements that exceed the capacity of a single processor, and thus super-linear speedups for small cluster sizes.

In this environment, systems without some form of multiprogramming (time- or space-sharing) will not be competitive. But for non-degenerate workloads, it is difficult to design a scheduler that does not create perverse incentives for speedup-maximizing users. So, what can you do if (for your workload) your scheduler is perverse?

- time-sharing: either instead of or in addition to space-sharing

- central planning: if all jobs are adaptable, the scheduler can allocate processors according to some global objective.
- scheduler tuning: it should be possible to tune the parameters of the scheduler until it is non-perverse for that workload, or for a class of workloads.
- greed-friendly scheduling: it may be possible to implement a scheduler which generates an optimal schedule (or near-optimal, in some sense) in the presence of greedy users. Such a scheduler is non-perverse for all workloads.

Time-sharing parallel machines is, of course, an active research topic. I am choosing to do research on space-sharing instead, because I believe that space-sharing is a more promising approach:

- the overhead of time-sharing is much greater than the overhead of space-sharing. Context-switching is not only expensive in terms of run-time overhead; it also requires significant hardware, system, and network support.
- one of the benefits of time-sharing in a sequential environment — the ability to overlap one job's I/O with computation on another job — is likely to be less significant for parallel workloads.
- time-sharing is perverse, in the sense that it encourages jobs to allocate more processors than they can use efficiently.

Furthermore, I do not think that combining time- and space-sharing is promising, because with moderately high loads, jobs are properly encouraged to run on the number of processors they can use most efficiently. For jobs with superlinear speedups, the point of maximum efficiency is near the point where the job fills the physical memory of the processors. In this environment, time-sharing would be ineffective.

Although central planning would be an interesting research topic, I do not think that in real systems it would be either practical or effective.

I would, therefore, like to explore the possibilities of scheduler-tuning and greed-friendly scheduling. I am confident that the first is possible. I am not sure about the second, but if it is possible, it would be a very good thing.

For a given (non-degenerate) workload, there is reason to believe that it is possible to tune a scheduler to produce a non-degenerate (if not optimal) schedule. The justification of this claim is

- it is easy to implement a simple scheduler that leads to the degenerate solution where all jobs get one processor

- by increasing the priority of jobs with large cluster size, or jobs that have long queue times, it is possible to reach the other degenerate solution, where all jobs allocate the entire machine
- somewhere in between, there should be a scheduler that is at least non-perverse, and probably close to optimal

The goal of my research will be to find this middle ground, and study its behavior for a variety of real workloads. I will also try either to find a greedy-friendly scheduler or to show that it is not possible.

3.2.3 Thesis

It is my thesis that a space-sharing strategy cannot be effective unless there is a scheduler that is greed-friendly for some broad class of workloads. If there is not, then I do not believe that space-sharing (as defined here) will be a viable option for future systems.

3.2.4 Related Work

In the context of networks of workstations, [Atallah et al. 92] present a problem formulation similar to what I have proposed: minimizing response time for a parallel job, given the speedup of the job as a function of the number of processors it is allocated, and the expected queue time as a function of cluster size. They propose an operating system structure which includes application agents that operate on behalf of a submitted job and choose the optimal cluster size. They mention the issues of fairness and starvation, but do not address them; also, they mention

> A more general issue is how local concerns, e.g. the desire to obtain optimal effective speedups for individual parallel applications, could be reconciled with the goal of minimizing the number of idle cycles of all workstations.

This is exactly the issue I would like to address.

[Rosti et al. 94] present several adaptive space-sharing policies and compare their performance for a synthetic workload running on an Intel iPSC/2. They also propose a metric of "robustness," that quantifies the ability of an allocation policy to provide good performance for a variety of workloads. They assume that all jobs are malleable and that the system chooses the cluster size for each job; thus, the issue of perverse incentives does not arise. Nevertheless, this paper provides a useful catalog of allocation strategies.

[NaikSetiaSquillante93] presents a simulation study of simple static, adaptive and dynamic space-sharing policies. Based on the poor performance of static space-sharing, they conclude that

> …the system scheduling policy must reduce the number of processors allocated to each job with increasing system load … [thus], it is essential for applications to be written for a variable number of processors.

For shared-memory multiprocessors, [ZahorjanMcCann90] compares the performance of several adaptive and dynamic space-sharing policies. For distributed memory parallel computers, [McCannZahorjan94] compares two dynamic space-sharing policies for distributed memory parallel computers. In my opinion, the second paper is a flying pig study; nevertheless, it defines a useful concept for discussing space-sharing policies:

> Efficiency preservation is the degree to which the decisions of the processor allocator degrade the processor efficiencies experienced by individual applications relative to their efficiencies when run alone.

[Stanovic85] discusses the stability and robustness of distributed scheduling algorithms for independent sequential jobs on networks of workstations. Although his formulation of stability is not applicable to the problem I am considering, he makes some interesting comments on the limited ability of analysis and simulation to address stability issues.

Also in the context of independent sequential jobs, [KruegerLivny87] propose a number of fairness principles that suggest alternate metrics for system performance (besides mean service time). These include repeatability, predictability, and non-discrimination. Not surprisingly, they observe that these goals are often mutually contradictory.