Metrics and Benchmarking for Parallel Job Scheduling

Dror G. Feitelson[1] and Larry Rudolph[2]

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

[2] Laboratory for Computer Science
MIT
Cambridge, MA 02139
http://www.csg.lcs.mit.edu:8001/Users/rudolph/

Abstract. The evaluation of parallel job schedulers hinges on two things: the use of appropriate metrics, and the use of appropriate workloads on which the scheduler can operate. We argue that the focus should be on on-line open systems, and propose that a standard workload should be used as a benchmark for schedulers. This benchmark will specify distributions of parallelism and runtime, as found by analyzing accounting traces, and also internal structures that create different speedup and synchronization characteristics. As for metrics, we present some problems with slowdown and bounded slowdown that have been proposed recently.

1 Introduction

Since the performance of a computer system depends on the workload which it is processing [3, 18], we argue that a workload benchmark suite is needed in order to evaluate and compare the many features of job schedulers for parallel supercomputers. But unlike standard benchmarks suites that consist of a set of "representative jobs" executed in isolation, a workload benchmark specifies the submission of jobs into the system and characterizes the types of jobs. Part of this characterization may include a description of the internal structure of the jobs themselves. This additional specification allows one to exercise various scheduler features.

A workload benchmark is likely to be useful in quantitative comparisons of two different job schedulers, perhaps even if they are executing on two different machine types. The only requirement is that the same type of generic workload will be relevant to both machines. Moreover, it is useful in evaluating the impact of various scheduler features. For example, it will be possible to evaluate the benefit of a scheduler sensitive to the mass-storage needs of its workload over one that ignores them. It is by far preferable to demonstrate the usefulness of some scheduler feature on a workload that is representative of what may occur on real systems, rather than generating one's own workload tailored to demonstrating the superiority of one's new feature.

There are other approaches to scheduler evaluation. One common method is to use traces of real workloads directly. The problem with this approach is that such traces are not necessarily representative, and that they only provide a single data point. In order to be able to assess the importance of different characteristics of the workload, it is better to use a synthetic benchmark suite. Analytical methods are another common approach. Although this works fine in certain limited domains, most realistic scenarios are too complex. These methods make assumptions about the workload of the scheduler. It is the claim of this paper, that the workload assumptions should be standardized. It does not matter if the synthetic workload is input to analysis, simulation, emulation, or real execution.

We start by explaining the different types of system dynamics that may be assumed, and justifying our focus on on-line, open systems (Section 2). As it seems premature to fully specify a benchmark, we discuss the specifications that are needed and identify topics that require additional research (Section 3). A discussion of metrics then follows since it is the goal of a scheduler to optimize one or more metrics (Section 4). Then we discuss implementation concerns (Section 5) and finally present our conclusions (Section 6).

2 Types of Queueing Systems

A computer system is essentially a *queueing* system: jobs arrive, may wait for some time, receive the required service, and depart. Such systems can be classified as on-line vs. off-line, with the on-line branch being further classified as open or closed (Fig. 1). All of these classes have been used in the analysis of computer systems.

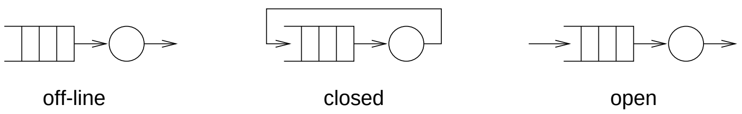

Fig. 1. *The three generic types of queueing systems.*

Off-line analysis assumes all the jobs — and maybe also their resource requirements — are available from the outset. There are no additional arrivals later. The scheduler can then pack the jobs together in order to minimize the total processing time. Such a model is often suitable for space slicing, batch job schedulers and their performance can often be predicted using analytical methods [7]. It is also convenient for measuring the execution of real applications as scheduled by real schedulers.

Alternatively, one can assume an *on-line* model where jobs arrive over a period of time. In this case the scheduler must handle new jobs "in real time," without the benefit of prior knowledge about future arrivals. A *closed* on-line system assumes that there is a fixed set of jobs to be handled. Thus arrivals are in effect linked with departures of previous jobs, and there is a bound on the maximum number of jobs in the system at any one time. Although more difficult, such a workload model is still amenable to analytic analysis.

The approach followed in this paper can be characterized as an *open*, on-line system in which there is an endless stream of jobs arriving for service. This most closely models the challenges of real job schedulers, where arrivals are independent of departures and indeed of the current load conditions. However, this type of model is more complex, because the arrival process has to be modeled as well.

Using an open, on-line model implies that the scheduler must be able to handle extreme situations, since in an open system, the tail of a distribution can and will occur. In fact, part of the analysis is to see when the scheduler breaks down because it can no longer handle the incoming load (this always happens when the load approaches the system capacity). Such an analysis is not possible with off-line or closed models.

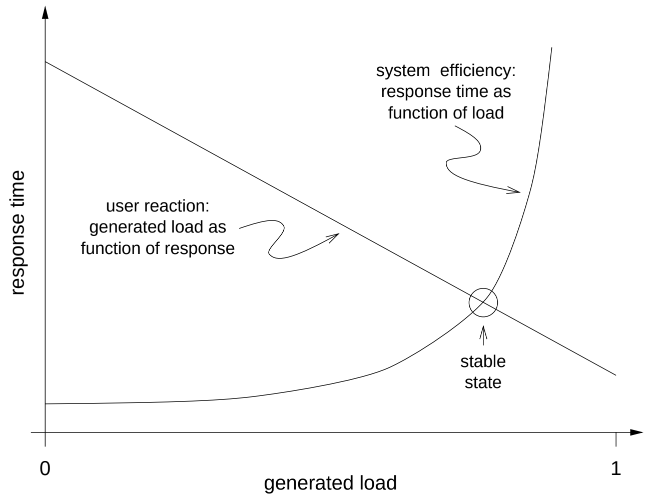

Fig. 2. *User reaction to system performance may be described by supply and demand curves. Only the "system efficiency" curve need be characterized in order to evaluate the system.*

At first blush, it appears that the modeling process is complicated by the fact that users may react to certain "features" (or bugs :-) of the job scheduler. Sophisticated users will learn to exploit imbalances, and cause a bias in the workload. Static workloads, such as those assumed by off-line or closed on-line analysis, cannot capture such user's reactions. Dynamic workloads — as in on-line open systems — allow for a better characterization of the system, but more importantly, they leave the modeling of the user as a separate issue (Fig. 2).

Finally, we note that the use of dynamic workloads captures some second-order effects, whereby feedback due to system characteristics may modify the workload. Examples include jobs that receive preferential treatment, and therefore stay in the system less time, whereas jobs that receive degraded service stay in the system longer than may be expected. As a result the observed mix has more "bad" jobs than the original mix. While it is not clear whether such effects are really significant, it is prudent to be conservative and use a model that does not preclude them.

3 Workload Specifications

A workload description for an on-line, open system can be viewed as consisting of two major components: job arrival and job structure. Each job arrives at a specific time and requires a specific amount of processing time, which we refer to as *work*. Thus, there is a model for the distribution of the arrival process and a separate model for the distribution of each particular job's work requirement. Fortunately, trace data accumulated at various supercomputer computation centers enables realistic models.

The first component describes how jobs are submitted to the system over a period of time. This can be somewhat involved, as a distinction has to be made between short interactive jobs and long batch jobs. In addition, there are daily and weekly cycles in the arrival process, due to the working patterns of the human users of the system.

The second component is that of modeling the work requirements of each job. This can be done in a monolithic manner, or else the internal structure of each job can be specified. As additional internal job structure is modeled, more sophisticated scheduler features can be evaluated, presumably resulting in a more efficient system. Unfortunately, there is not much hard data that has been measured about typical internal structural distributions, but there are common scenarios. The most common and clearly identifiable structures are the computational structure (parallelism and barrier synchronizations), interprocess communication, memory requirements, and I/O needs. The discussion will be limited to only the computational structure because of two reasons: it is the one about which we have some knowledge about typical patterns and it illustrates the specification choices.

The rest of this section addresses possible choices for arrival and computational structure distributions.

3.1 Modeling Job Arrivals

Two broad classes of arrival scenarios can be identified. In one, the arrival process is a memory-less, continuous process. In the other, it is cyclic. The latter case more closely represents observed arrival patterns, where the number of jobs

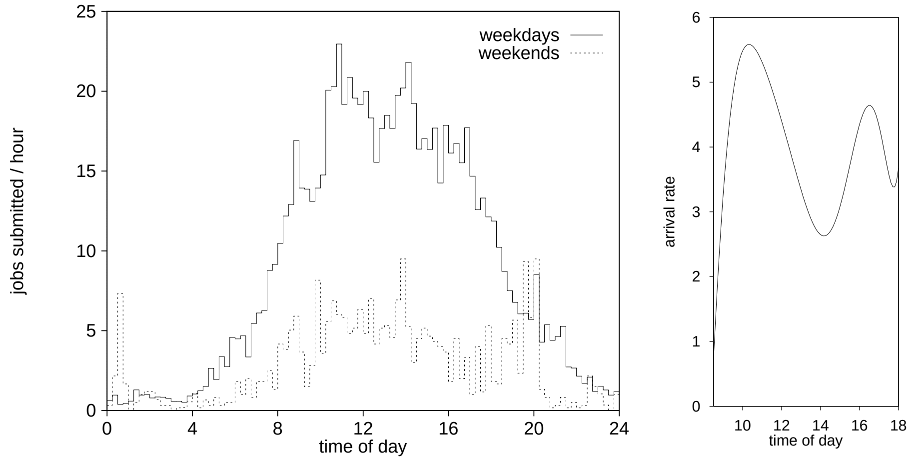

Fig. 3. *Left: cyclic job arrival pattern in the NASA Ames iPSC/860 (from [10]). Right: model of Calzarossa and Serazzi for workload at the University of Pavia [2].*

submitted strongly correlates with the time of day and the day of the week (Fig. 3). However, the simpler continuous model is the one that system performance evaluation almost always uses in practice. This has the unfortunate effect of excluding the evaluation of scheduler optimizations that increase the priority of interactive jobs during the day, at the expense of computational (batch) jobs that are delayed to when more resources are available at night.

It should be noted that a good model of the arrival process is also necessary in order to create various load conditions for the evaluation. If the cyclic structure of the arrival process is acknowledged, it is no longer possible to increase the load by uniformly reducing the interarrival times, because such a practice will also shrink the cycle length. Regrettably, very little work has been done on the derivation of realistic models.

The only detailed model we know of was proposed by Calzarossa and Serazzi [2]. This model uses a polynomial of degree 8 to model the changing arrival rate of interactive work (Fig. 3). The proposed polynomial for "normal" days is

$$\lambda(t) = 3.1 - 8.5t + 24.7t^2 + 130.8t^3 + 107.7t^4 - 804.2t^5 - 2038.5t^6 + 1856.8t^7 + 4618.6t^8$$

where $\lambda(t)$ is the arrival rate at time t, and t is in the range $[-0.5..0.5]$, and should be scaled to the range from 8:30 AM to 6:00 PM. This expression represents the centroid for a set of polynomials that were obtained by fitting measured results for different days. Slightly different polynomials were discovered for abnormal days, in which the administrative office closed early, or were the first day after a weekend or a holiday. While the authors warn against using this data without additional verifications, we propose it as an initial model until more suitable ones are derived.

Much additional work is required in order to better characterize the arrival process of parallel jobs. Specific research questions include

– The possible differentiation between arrival models for batch and interactive jobs. Do both types of jobs arrive according to the same patterns?
– The possible correlation of arrival time with work requirement. Are jobs that arrive at different times of the day and night statistically equivalent, or do they tend to have different structures? For example, do users submit smaller jobs during the morning and larger ones in the afternoon, in anticipation of the resources that will be freed up at night?

3.2 Modeling Rigid Jobs

At the least specific level, no internal computational structure is specified and a job consists of just an amount of work. This work can be processed sequentially or in parallel, with no loss of efficiency. However, such a model is usually too simple minded to be useful.

The simplest useful model is rigid jobs, in which both the work and the degree of parallelism are specified. This is an "external" model, with no details of the internal structure of jobs. It is useful because many parallel supercomputers provide schedulers for this type of jobs, and because this type of workload model can be derived from accounting logs. Indeed, a number of such models have already been derived and used in the evaluation of schedulers for parallel systems [8, 15, 18].

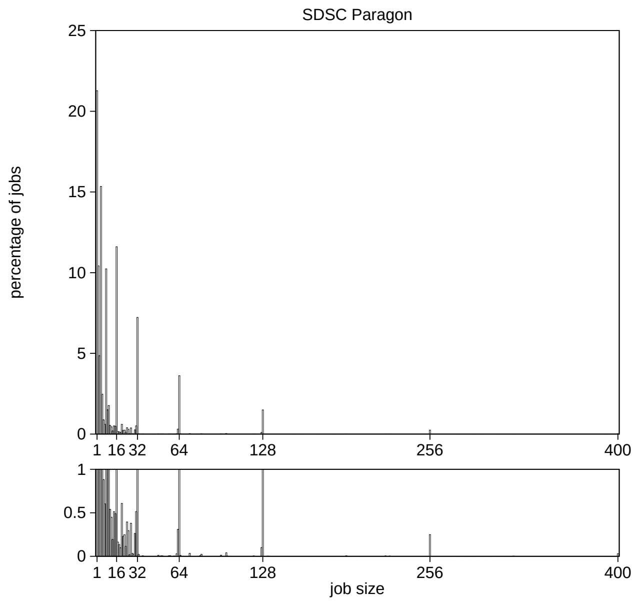

Fig. 4. *Representative distribution of job sizes, showing dominance of small jobs and jobs using a power-of-two processors. (Data from SDSC Paragon.)*

Interestingly, the accounting logs from many diverse systems show several common characteristics, most of which were not anticipated in advance. One such characteristic concerns the distribution of job sizes, i.e. the number of processors that are used. It turns out that even in very large machines, small jobs using only a handful of processors dominate the workload (in terms of number of jobs, though not in terms of runtime). In fact, in machines having more than about 100 processors, there are usually only very few jobs that use the whole system. In addition, there is a strong tendency to use a power-of-two number of processors, even if this is not warranted by the architecture. A representative distribution of job sizes, taken from the 400-node Paragon machine at San-Diego Supercomputer Center, is shown in Fig. 4.

Modeling the distribution of degrees of parallelism is relevant for schedulers that handle rigid jobs. We propose the following approach: first, model the overall distribution as linear in the logarithm of the parallelism, as suggested by Downey [5]. This means that the probability of using fewer than n processors is roughly proportional to $\log n$. Then, modify the distribution by creating steps at powers of two. The size of the steps is determined by a parameter describing the workload, which specifies what percentage of the jobs use power-of-two nodes. The value for the SDSC Paragon cited above is about 81% (including 21% that were serial). More work is required to derive better models, including answers to the following questions:

- What is a good representative value for the fraction of jobs that use power-of-two processors?
- Is the use of powers of two a real feature of workloads, or only an artifact resulting from old habits and common interfaces to batch queueing systems?
- Are all powers of two equally likely?

Another potentially important characteristic concerns the correlation between the degree of parallelism and the runtime. In the past, it has been speculated that highly parallel jobs should be shorter, because parallelism is used to achieve speedup. In fact, workload traces indicate that highly parallel jobs run longer (Fig. 5). This has two possible interpretations: either the smaller jobs are development while the larger ones are production runs, or parallelism is used to solve larger problems rather than to achieve speedup on given problems.

A possible model for such a correlation has been proposed by Feitelson [8]. The basis for the model is the observation that job runtimes have a very large variability, manifested by a coefficient of variation that is larger than 1. A plausible model for runtimes is therefore a hyperexponential distribution. For example, a two-stage hyperexponential can be used; intuitively, this means that we first choose at random from two exponential distributions according to a probability p, and then sample the chosen distribution. The correlation with parallelism is achieved by making the probability, p, a function of the parallelism, n. Specifically, Feitelson used

$$p(n) = 0.95 - 0.2(n/N)$$

where N is the system size; thus for small n we get that p is near 0.95, and for large n it goes down to 0.75. Given p, sample an exponential distribution with

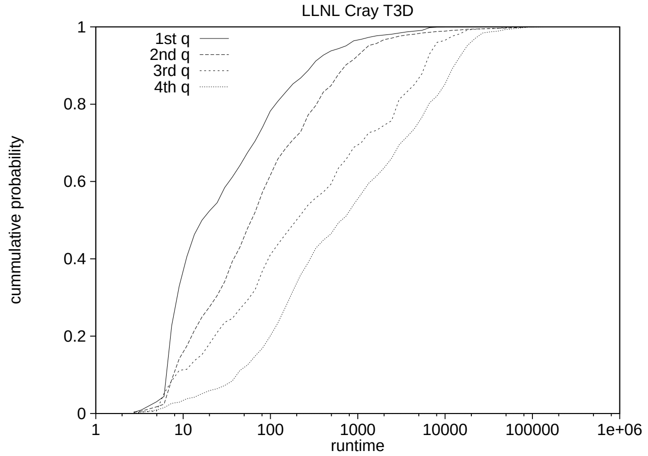

Fig. 5. *Correlation of runtime with parallelism is evident when the distribution of runtimes is plotted for 4 sets of jobs independently, where each set contains jobs with a different degree of parallelism. The weight of the distribution for the set with the smallest jobs is at low values, while jobs with high parallelism tended to have higher runtimes as well. (Data from LLNL Cray T3D.)*

mean 1 with probability p, or a distribution with mean 7 with probability $1-p$. We are now working on a better model, that will be based on better statistical analysis of workload traces.

Simpler models for job runtime have also been proposed. For interactive jobs (e.g. in a Unix environment), it has been suggested that job runtimes have a cumulative distribution of $F(t) = t^k$, with $k \approx -1$, provided the jobs are longer than a second or so [13]. This means that

$$p(\text{runtime} = t \mid \text{age} = 1sec) = 1/t^2$$

For batch jobs, it has been proposed that the logarithms of the runtimes are uniformly distributed, so their cumulative distribution is linear, $F(t) = a \ln t + b$, with $a \approx 0.1$ [6]. This leads to

$$p(\text{runtime} = t) = 1/10t$$

3.3 Modeling Internal Job Structure

Jobs come in many different shapes, sizes, and styles, and it is important to model much of this internal job structure since models of rigid jobs do not allow for the evaluation of many innovative schedulers. For example, schedulers may wish to change dynamically the degree of parallelism provided to a job, in order to account for various load conditions. The resulting performance depends on the speedup curves of the application [21,19]. Thus, for each job, it must be possible to compute runtime as a function of partition size. Another example is that a scheduler may want to modify the "gangedness" of an application, that is the degree to which all the processes execute simultaneously on distinct processors. Again, the resulting performance depends on the characteristics of the application [17], and the workload model must specify runtime as a function of skew.

There are two general methods for modelling the "internal" job structure. One is based on equations describing job behavior. This approach has been used for analyzing specific situations, such as how runtime changes with degree of parallelism for adaptive partitioning [20, 4]. However these equations are typically expressed as speedup functions, and imply some assumption about the scheduling, e.g. that all the threads execute simultaneously without interference [19]. A more general approach is to specify the internal structure so that simulation or detailed analytic methods can be used to calculate the runtime from the structure.

A hierarchical model that includes the internal structure of the workload has been proposed by Calzarossa et al [1]. Their model includes the levels of applications, algorithms, and routines, and thus is suitable for the modeling of real applications. We prefer a synthetic workload that only includes certain abstract structures.

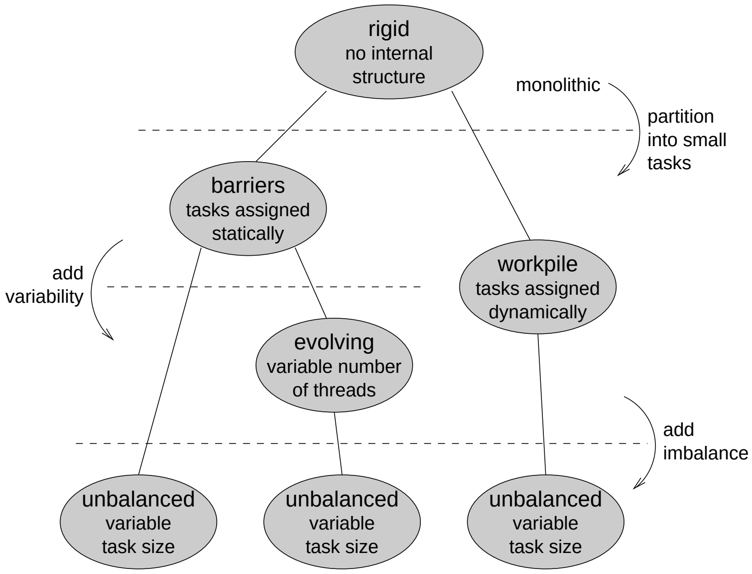

Fig. 6. *The proposed hierarchical workload benchmark suite, in which lower levels add more detail to the internal structure of jobs.*

Our proposal is to capture the "popular" alternative programming styles and scheduler features and is outlined in Fig. 6. The space is deliberately sparse; to keep the set of alternatives manageable, it is necessary to exclude many combinations. At the top level are external models as described above. Lower levels inherit the distributions of total work in the different jobs, and add internal structure. Two basic internal structures are proposed: one in which the computation is organized as communicating threads that synchronize with barriers, and the other in which the computation is organized as an unordered workpile. The barrier structure has a variant in which the number of threads changes from barrier to barrier — this is essentially the fork/join model that represents a sequence of parallel loops. All models are parameterized by their granularity: for barriers, this is the amount of computation each thread does between barriers; for workpile, this is the typical task size. Finally, in all models we may add imbalance by specifying a distribution of task sizes. Workload parameters used to define these workloads are described in Section 5.

One anticipated use of the workload models is that one can choose the model that is most suitable to exercise the scheduler being evaluated. Another important use is to check how the scheduler handles jobs with other characteristics, that are not specifically dealt with in the design of the scheduler. For example, how does a gang scheduler handle a job with evolving parallelism? And how does a two-level scheduler with dynamic partitioning handle a strictly SPMD code with barriers?

While standardization of the job structures that are used to benchmark parallel job schedulers is important, it does not cover the whole workload modeling question. The missing part is creating a job mix from these structures. One must always be careful when evaluating a scheduler with a set of jobs that all have the same structure, because then the likelihood of correlations between the jobs grows. Regrettably, there is no information about typical and realistic job mixes. The definition of good mixes is left as a question for future research.

4 Performance Metrics

As noted in Section 2, computer systems can be modeled in several ways. For each type of system, a different metric is commonly used (Fig. 7). In this section we investigate metrics related to the response time, which is the most suitable for open on-line systems, and explain why we do not use other metrics such as utilization and throughput.

4.1 Metrics and System Types

One problem with selecting a performance metric is that in a real system different metrics may be relevant for different jobs. For example, response time may be the most important metric for interactive jobs, while system utilization is more important for batch jobs. But in an open, on-line system, utilization is largely determined by the arrival process and the requirements of the jobs, not by the scheduler. This leaves response time as the main metric.

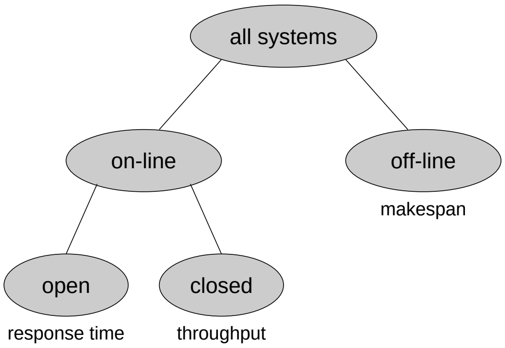

Fig. 7. *Classification of system types and common metric used for each.*

The way to use response time as a metric is to find its functional dependence on system load, as in Fig. 2. This means that many different load conditions should be checked. With rigid jobs, load and utilization are completely determined by the arrival rate, so it is easy and justifiable to present the results as a function of utilization [11]. But with adaptive or dynamic partitioning schemes, changing the partition size may change the efficiency of job execution and thus change the utilization. The correct load variable is therefore the arrival rate, not the resulting utilization. This has the unfortunate consequence that it becomes harder to compare different schemes, because one needs to understand the workload details to correlate the arrival rates. A standard workload benchmark will solve this problem.

Throughput is a good metric for closed systems, because there the arrival process depends on system performance: each job is re-submitted immediately each time it terminates. The question is then how fast the jobs repeat, i.e. how many times they are executed per unit time. Utilization is also a good metric in this case, because faster job turnaround increases utilization. The power metric is an intriguing variant [16]. It is defined as the throughput divided by the response time, so it goes up when either the throughput goes up or the response time goes down. However, if the throughput is determined by the arrival process, power provides the same information as response time.

Makespan is the metric of choice for off-line scheduling. It can be thought of as an off-line version of response time: it is the time that the whole workload terminates, rather than the (average of the) time that each job terminates. In the off-line scenario, it is directly linked with utilization and throughput, and each can be derived from the others (given information about average requirements of jobs). The same is not true of response time, which depends on the scheduling order.

4.2 Response Time, Slowdown, and Bounded Slowdown

The average response time is a widely accepted metric for open, on-line systems. However, it seems that this metric places greater emphasis on long jobs, as opposed to short jobs, which are much more common. For example, the average response time of 100 1-hour jobs and one 3-week job is 6 hours. A possible solution to this problem is to normalize the reported values by using slowdown rather than raw response time (slowdown is defined as runtime on a loaded system divided by runtime on a dedicated system). Thus all jobs are reduced to the same scale, with 1 indicating good performance, and higher values measuring the degree of degradation. The problem with slowdown is that extremely short jobs with reasonable delays lead to excessive slowdown values. For example, a 1 second job that is delayed for 20 minutes suffers a slowdown of 1200. The proposed solution to this problem is to apply a lower bound on job runtimes, e.g. 10 seconds [12]. Shorter jobs are treated as if their duration is this lower bound; in the above example, the bounded slowdown value is then 120 rather than 1200.

The above "handwaving" arguments indicate that using bounded slowdown should lead to measurements with less variance (and thus quick convergence) that take fair account of all jobs. Regrettably, actual measurements seem to indicate that this is in fact not always the case. The following results are from a simulation of variable partitioning with backfilling, using a realistic model of rigid jobs (similar to the proposal in Section 3.2), and assuming a system of 128 nodes. This is one of the simulations reported in [9], which has been instrumented to collect more data.

Fig. 8 shows the behavior of the three metrics (response time, slowdown, and bounded slowdown) for the first 5000 jobs in the simulation run. The individual value for each job is plotted, as well as a running average. It shows that while response times vary much more than slowdowns, both types of slowdown suffer from bursts of very high values. As a result, the running average of the slowdown converges more *slowly* than that of the response time. Bounded slowdown is somewhat better.

The same effect can be seen in Fig. 9, in which the average values are plotted for a very long simulation. remarkably, the plot for the bounded slowdown is nearly identical to that of the response time, whereas the one for slowdown is much more erratic. Nevertheless, even the "better behaved" response time and bounded slowdown continue to vary even after more than 200000 jobs have been simulated. This is extremely long, considering that typical large supercomputers execute less than 100000 jobs in a whole year.

Some insights can be obtained from Fig. 10, which shows a scatter plots of slowdown and bounded slowdown vs. response time. Two clusters stand out in these plots. In one cluster, jobs have a high response time coupled with a low slowdown. This means that these are long jobs, and the high response time actually reflects their computational demands. In the other cluster the slowdown is proportional to the response time, with much weight concentrated where both slowdown and response time are high. This cluster includes jobs that are actually

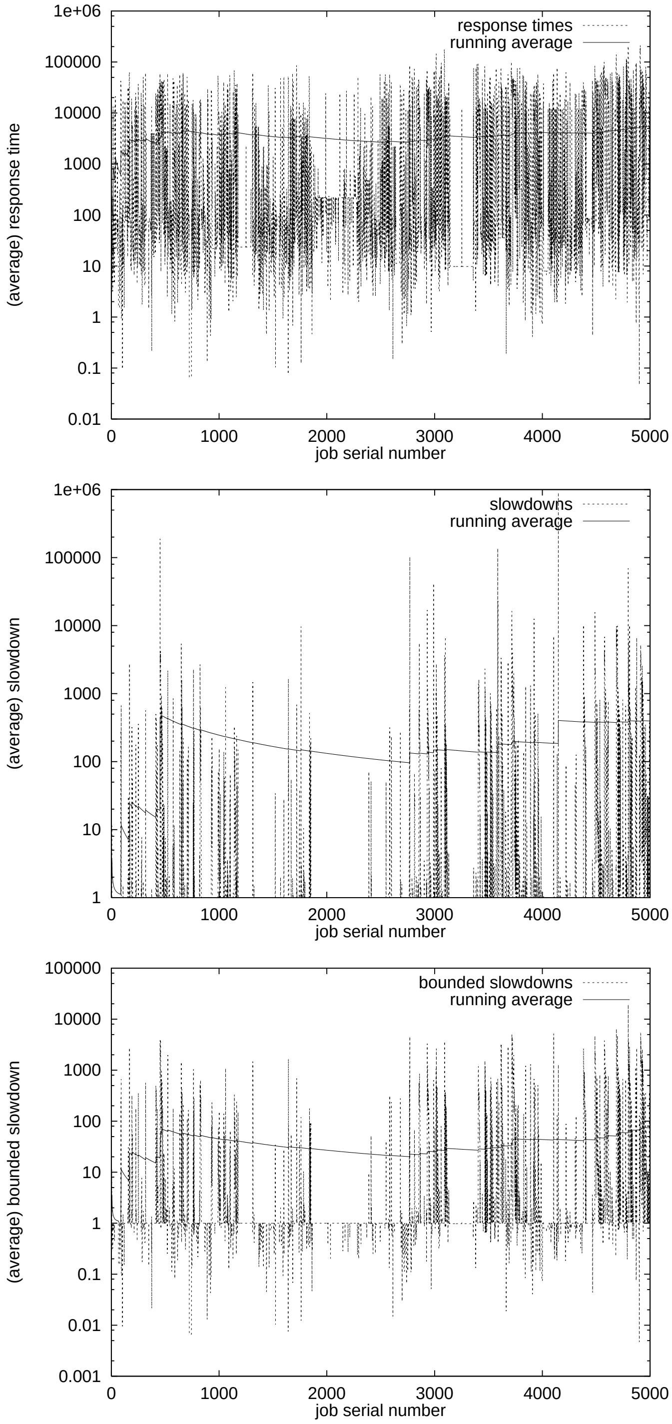

Fig. 8. *Pointwise and running average of metrics for first 5000 jobs in simulation.*

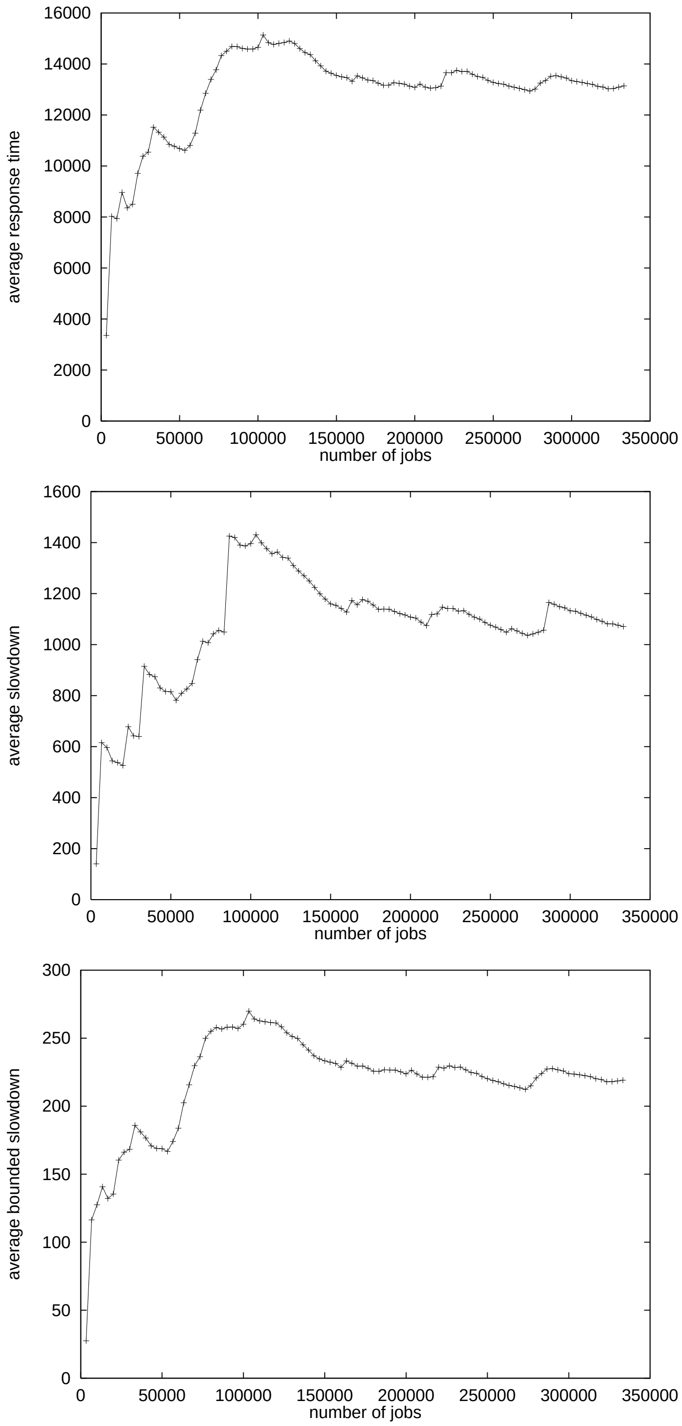

Fig. 9. *Running average of metrics for a very long simulation. Each data point represents an additional 3333 job terminations.*

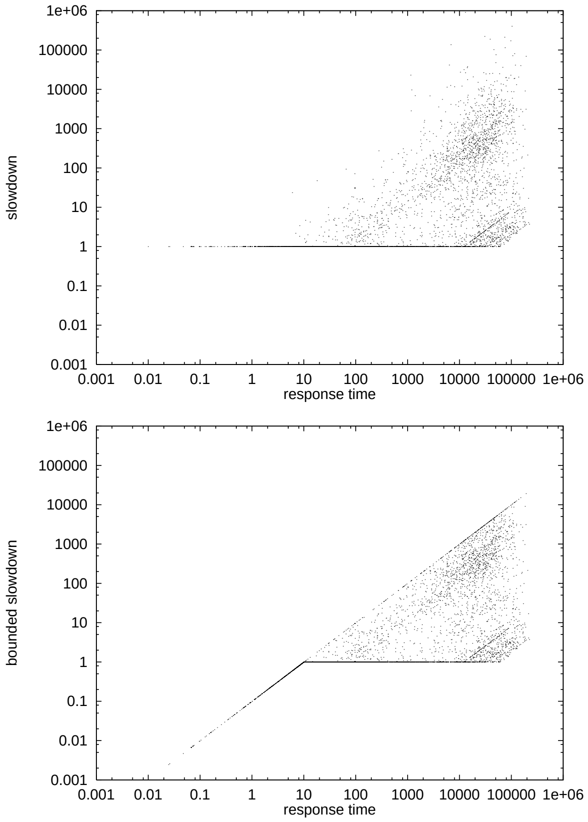

Fig. 10. *Scatter plots of slowdown vs. response time.*

quite short, and their high response time reflects time stuck in the queue waiting for some long job to terminate. Using bounded slowdown trims the most extreme values in this case (bottom plot), where the jobs that are delayed are *very* short. In this cases the denominator in taken as a constant, rather than being the job's runtime, leading to values that are linearly related to the response time — hence the similarity of bounded slowdown and response time.

Nevertheless, this manipulation can't make the problem go away. Due to the high variability of runtimes, at rare intervals a long job comes along and jerks the response time; it also causes multiple short jobs to wait, and thus jerks the slowdown and bounded slowdown. However, this is actually an artifact of the scheduler used in these simulations, which is based on variable partitioning and causes short jobs to be delayed. The behavior of slowdown and bounded slowdown with mechanisms that do not delay jobs, such as gang scheduling or dynamic partitioning, is expected to be different.

In summary, the question of what makes a good metric is still open. More work is required in order to refine our understanding of the relations between response time, slowdown, and bounded slowdown, and maybe additional metrics should also be investigated.

4.3 Workload-Dependent Metrics

Our proposed benchmark suite (Section 3) contains families of programs whose behavior depends on a parameter that specifies their granularity. The results will naturally depend on the value chosen for this parameter. One approach is to report performance results for a range of granularity values. Another is to make such detailed measurements, but report only the following:

1. The best performance that is obtained, at either very high or very low granularity, and
2. The granularity at which half this performance is obtained.

This approach is inspired by the $r_\infty, n_{1/2}$ metrics proposed for vector processors [14].

5 Implementation Issues

This section addresses some practical issues for the implementation of a workload benchmark generator. There is a small set of parameters that need to be specified in order to generate the workloads discussed in Section 3. Some parameters deal with the external job structure, while the rest deal with the internal job structure. The basic idea is to have one common set of parameters through which the relevant internal job structure features can be specified. Then, a single synthetic job skeleton is required for either a simulated or real execution environment.

Of all the parameters, there is sufficient trace data to realistically model the arrival time of a job and its parallelism. There is little or no data concerning the internal job structure. It is therefore hard to assess representative distributions of the parameter values. Unfortunately, when executing jobs in a parallel environment, the actual time of the execution of individual pieces of code is of crucial importance. In particular, if the time between barriers is too short, the implementation of the barrier may dominate the performance.

5.1 Workload Parameter Space

We define small set of parameters that can be used to capture all the workloads defined in Section 3. This is based on the observation that they are all expressible by various combinations of barrier synchronizations and workpile semantics. By workpile semantics we mean that processors do not stay idle if there is an atomic unit of work to execute; thus if the number of work units is no more than the number of processors, they get mapped one each to the processors. If there are more work units than processors, they are executed in an undefined order by the processors as they become available.

W	Total number of work units in the job
P_l, P_u	lower and upper bounds on the number of processors
B_l, B_u	lower and upper bounds on the number of barriers
$\bar{w}_i, w_{std}$	mean number of work units per barrier, standard deviation of the work units per berrier
$\bar{u}_i, u_{std}$	mean compute time of a work unit, standard deviation of work unit time

Fig. 11. A proposed set of parameters to specify the internal structure of a workload.

The table in Fig. 11 lists the parameters. The idea is that the work done by the job is the sum of many atomic work units, W, which are each computed by a single processor. Precedence constraints between these work units, if any, are expressed by the number B of barrier synchronizations. The number of work units between barriers w_i therefore represents the degree of parallelism in that phase. The mean compute time of a work unit u is used to calibrate the workload across different machines. The variability of u, expressed as its standard variation u_{std}, can be used to add variability among work units.

The next paragraphs explain how to set these parameters to specify the different workloads.

Rigid Jobs Rigid jobs are usually represented by two parameters: the number of processors P and the execution time T. It is assumed that the job is gang scheduled, since if it is not, then there is not enough internal structure to understand the runtime.

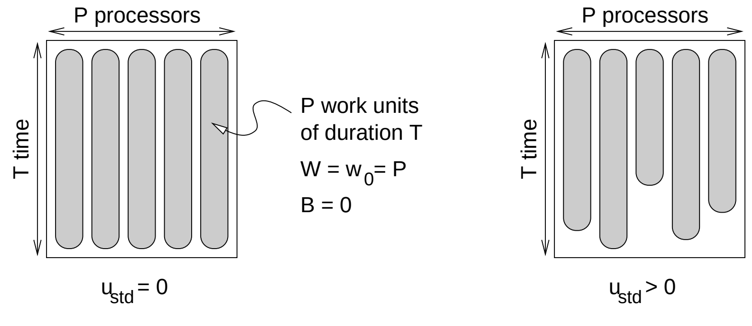

Fig. 12. *Expressing a rigid job structure with the parameters.*

This formulation is expressed using our parameters by creating P work units that execute on P processors for T time, with no additional structure (Fig. 12). The definitions of P and W are then $P_l = P_u = P$ and $W = P$. Since there is no internal structure, the rest of the parameters are easily set too: $B_l = B_u = 0$,

$w_0 = P$, $w_{std} = 0$. The mean compute time $\bar{u}$ links with the parameter T normally used to express the duration of rigid jobs: $\bar{u} = T$, $u_{std} = 0$. In fact, we can say that the total work in work units is actually $W = PT/\bar{u}$, but because there is only one work unit on each processor, T and $\bar{u}$ cancel out. The variability of u_{std} can be used to express imbalance among the processors.

Workpile A workpile job has no real internal structure, rather there is a pile of work to be processed. The more processors there are, the faster the work can be processed. However, there is often a minimal number of processors required to meet resource constraints (e.g. to have enough memory). There is also a maximum number of processors that can be assigned to a job. This is trivially bounded by the total number of processors in the system. Therefore the two relevant parameters typically used to describe a workpile are W and P_{min}. It is usually assumed that there is a linear speedup for processing the job.

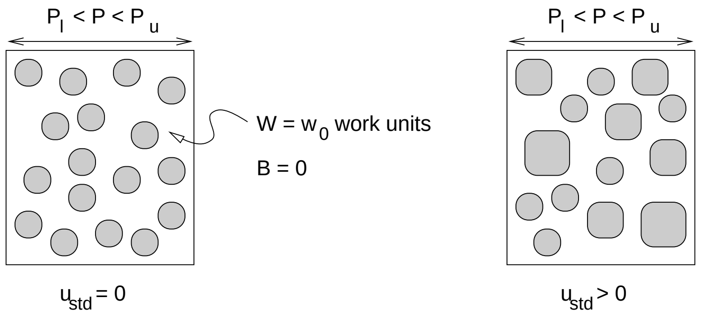

Fig. 13. *Expressing a workpile job structure with the parameters. With $W > P$ there is no internal structure.*

These parameters are easily converted to our job parameters as follows. W is simply the number of work units in the pile. $P_l = P_{min}$ based on resource requirements, and $P_u = P_{max}$. There are no barriers, so $B_l = B_u = 0$, $w_0 = W$, and $w_{std} = 0$. Assuming $W > P_{max}$, there are more work units then processors, and they can therefore be computed on any processor in any order. $\bar{u}$ can be set as desired for calibration, and u_{std} is used to add variability in work unit sizes.

Barriers The main internal job structure feature in this type of job is the number of barriers. The crucial parameter in terms of scheduling and performance is the granularity of each barrier. A secondary issue is how the barrier is implemented: with busy wait, with yielding, with operating system help, or with some combination of these; this affects overhead.

Let us start with a simple case in which each processor performs one unit of work at each barrier (Fig. 14). This is expressed by $w_i = P, 0 \leq i \leq B$, where B is the number of barriers in the job. It then follows that $B = W/P$ (assuming

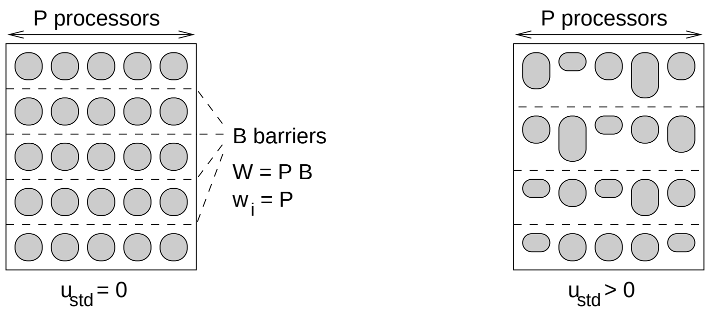

Fig. 14. *Expressing a simple barrier job structure with the parameters.*

that the number of processors is fixed: $P_l = P_u = P$). Alternatively, it is possible to select B from the range $B_l \le B \le B_u$, and then set $W = BP$. This is useful to create a workload of non-identical jobs, representing runs that took a different number of iterations to converge. The granularity of the barriers is expressed by $\bar{u}_i$.

Alternatively, the granularity may be expressed using a "workpiles between barriers" structure. This structure represents a sequence of parallel loops, where the iterations are independent of each other and can be done in any order. In this case $w_i > P$ for all i, so all the processors share all the work units between barriers in a workpile manner. The granularity can then be expressed in work units, rather than in time, as $\bar{w}_i/P$: this is the average number of work units per processor per barrier. The number of barriers is implicitly defined by the formula $B = W/\bar{w}_i$.

Fork-Join There is a large class of jobs in which the amount of parallelism varies during the course of the execution. For example, there may be a sequence of parallel loops with different degrees of parallelism, or separated by sequential phases. It is possible to express such structures by using different values for w_i, the number of work units associated with barrier i (Fig. 15).

If $w_i \le P$ for all i, then some processors will be idle in some phases, because there are less work units than processors. If $w_i > P$ in some phases, the work units are computed as a workpile in this phase; if it is in all phases, this is the situation discussed above. The number of processors P can be fixed, or else it can vary according to resource requirements and availability, leading to a continuum of possibilities between these two end points.

Creating a Job Mix We note that although it is possible to allow all the parameters to vary, it is doubtful that much meaningful information can be gathered from such cases.

There are two choices for specifying the mean values and their standard deviations: they can be identical for all the jobs (but each job has its own unique seed to the random number generator), or they can be generated from some

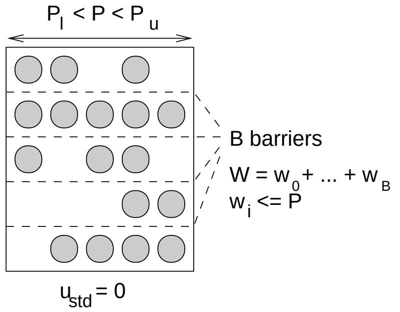

Fig. 15. *Expressing a variable-parallelism job structure with the parameters.*

other distribution during job creation. Without any hard evidence from actual workloads, it is impossible to say which is preferable. The mechanisms outlined above easily permit either approach.

5.2 Granularity Issues

Although we leave the actual distribution of the parameters as a subject for future research, there are some interesting points to be noted about the granularity of barriers, or in other words, the length of time of a work unit. It seems to be very important to get this value correct: if it is too big, there is little benefit for gang scheduling and if too small, jobs may never complete.

When there is a large value for a work unit, nearly as long as a time quantum, there is almost no difference between workpile or barrier. Moreover, there is little difference whether or not the system is gang scheduled or not. Similarly, when the work unit is very small the job terminates within the first time quantum, assuming that the processors all begin at the same time.

The number of barriers is also a sensitive issue. If there are too many barriers, the work-unit is short, and the system is not gang-scheduled, many jobs may never terminate. On the other hand, if the are only a few barriers, and the work-unit is short, then jobs may terminate within the first time quantum. Further experimentation is necessary to resolve these issues.

As an example, we experimented with a job with 10,000 barriers executed on an SMP IBM RS/6000 workstation with four PowerPC 604 Processors. The upper table in Fig. 16 shows the execution times on an idle system, as a function of the duration of the work units between barriers and the number of processors. Barriers were implemented by the MPI function. Notice that it is not until the work unit is 10^5 instructions long, that a linear speedup occurs. In the bottom table, the same set of experiments were performed while another 4 node job was executing. Here the four node jobs execute relatively worse than the others until the granularity is even larger.

Another example shows a case of shared memory jobs executed on the same idle system while varying the number of processors from 1 to 4. The barrier

Empty Machine

	granularity of work between barriers						
P	10^0	10^1	10^2	10^3	10^4	10^5	10^6
1	0.013	0.026	0.160	3.494	16.836	150.293	1486.261
2	4.022	4.028	6.720	6.977	12.071	80.657	748.408
3		5.279	7.315	7.736		54.859	500.494
4	5.302	5.122	5.110	7.360	9.211	42.705	382.139

With One Other Job

	granularity of work between barriers						
P	10^0	10^1	10^2	10^3	10^4	10^5	10^6
1	0.013	0.026	0.160	1.697	22.233	228.080	2300.667
2	8.673	6.662	8.967	10.390	21.454	144.723	1418.324
3	10.080	10.015	12.114	11.206	12.696	112.375	1136.834
4	11.462	13.975	11.112	12.166	19.621	104.883	1004.811

Fig. 16. Run-time of an MPI program, with 10,000 barriers, and various values for the granularity between the barriers.

Empty Machine (Shared Memory Program)

	granularity of work between barriers						
P	10^0	10^1	10^2	10^3	10^4	10^5	10^6
1	0.007	0.012	0.057	0.510	5.034	50.279	502.761
2	0.022	0.024	0.047	0.227	2.822	27.188	251.614
3	0.028	0.033	0.047	0.196	1.712	18.815	168.068
4	2.056	0.092	0.064	0.178	1.315	14.703	128.787

Fig. 17. Run-time of a shared-memory program, with 10,000 barriers, and various values for the granularity between the barriers. The barriers were implemented using busy-waiting on a global variable protected by a system mutex lock.

synchronization was executed as a busy-wait by increasing a counter protected by a lock, and there were 10,000 barrier synchronizations. The performance results are shown in Fig. 17. The granularity has a large effect on how the program performs: with a shared memory implementation of barriers, linear speedup occurs for a granularity as small as 10^3.

5.3 Execution Issues

Portability One goal of the benchmarks is that they be executable on a large number of platforms – both hardware and software. But, to be useful, the port to a new system should be as smooth as possible. The problem is that different systems support different features. For example, not many systems provide support for dynamically allocating and deallocating processors to jobs. It has been claimed that this ability dramatically improves overall performance.

The easiest option is for the system to simply ignore the extra specification of the job. If the amount of parallelism per barrier is variable, a system that does not support this feature can simply choose the maximum parallelism.

Length of Execution How long should the benchmark be executed? Given the probabilistic nature, given enough time, anything is likely to happen. That is, the system may get into a saturated state from which it never exits. We also do not have unbounded time in which to execute benchmarks. Experience will have to be teacher.

But we can say that the scheduler will behave differently during warm up and cool down. For this reason, we suggest that measurements only be taken during the steady-state behavior of the system. It is crucially important to ensure that the system does not "dry out" towards the end. For example, if we want to measure performance characteristics of 10000 jobs, we need to keep the arrival process going until all 10000 terminate. When the system is close to saturation, this means that we may have to generate much much more than 10000 jobs.

6 Conclusions

There is still much to be done before a comprehensive workload benchmark can be built. There are many aspects of a job's internal structure for which there is no experimental evidence concerning their actual distributions. It is our hope that this be rectified.

Thus, this paper has only begun the quest for a workload benchmark and for widely accepted and suitable metrics. Although a large design space has been outlined, it is likely that only a small portion of the space is needed to make progress. This portion should be identified and subjected to a focused research effort.

References

1. M. Calzarossa, G. Haring, G. Kotsis, A. Merlo, and D. Tessera, "*A hierarchical approach to workload characterization for parallel systems*". In *High-Performance Computing and Networking*, pp. 102–109, Springer-Verlag, May 1995. Lect. Notes Comput. Sci. vol. 919.
2. M. Calzarossa and G. Serazzi, "*A characterization of the variation in time of workload arrival patterns*". *IEEE Trans. Comput.* **C-34(2)**, pp. 156–162, Feb 1985.
3. M. Calzarossa and G. Serazzi, "*Workload characterization: a survey*". *Proc. IEEE* **81(8)**, pp. 1136–1150, Aug 1993.
4. S-H. Chiang and M. K. Vernon, "*Dynamic vs. static quantum-based parallel processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 200–223, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

5. A. B. Downey, "*A parallel workload model and its implications for processor allocation*". In 6th *Intl. Symp. High Performance Distributed Comput.*, Aug 1997.
6. A. B. Downey, "*Predicting queue times on space-sharing parallel computers*". In 11th *Intl. Parallel Processing Symp.*, pp. 209–218, Apr 1997.
7. M. Drozdowski, "*Scheduling multiprocessor tasks — an overview*". *European J. Operational Research* **94**, pp. 215–230, 1996.
8. D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.
9. D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
10. D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
11. D. G. Feitelson and L. Rudolph, "*Toward convergence in job schedulers for parallel supercomputers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–26, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.
12. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–34, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
13. M. Harchol-Balter and A. B. Downey, "*Exploiting process lifetime distributions for dynamic load balancing*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 13–24, May 1996.
14. R. W. Hockney, "*Performance of parallel computers*". In *High-Speed Computation*, J. S. Kowalik (ed.), pp. 159–175, Springer-Verlag, 1984. NATO ASI Series Vol. F7.
15. J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "*Modeling of workload in MPPs*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 95–116, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
16. L. Kleinrock, "*Power and deterministic rules of thumb for probabilistic problems in computer communications*". In *Intl. Conf. Communications*, vol. 3, pp. 43.1.1–43.1.10, Jun 1979.
17. W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "*Implications of I/O for gang scheduled workloads*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 215–237, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.
18. V. Lo, J. Mache, and K. Windisch, "*A comparative study of real workload traces and synthetic workload models for parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 25–47, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.
19. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Parallel application characterization for multiprocessor scheduling policy design*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 175–199, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

20. E. W. Parsons and K. C. Sevcik, "*Multiprocessor scheduling for high-variability service time distributions*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 127–145, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
21. K. C. Sevcik, "*Application scheduling and processor allocation in multiprogrammed parallel processing systems*". *Performance Evaluation* **19(2-3)**, pp. 107–140, Mar 1994.