# Moldable Parallel Job Scheduling Using Job Efficiency: An Iterative Approach

Gerald Sabin, Matthew Lang, and P Sadayappan

Dept. of Computer Science and Engineering, The Ohio State University, Columbus OH 43201, USA,
{sabin, langma, saday}@cse.ohio-state.edu

**Abstract.** Currently, job schedulers require "rigid" job submissions from users, who must specify a particular number of processors for each parallel job. Most parallel jobs can be run on different processor partition sizes, but there is often a trade-off between wait-time and run-time — asking for many processors reduces run-time but may require a protracted wait. With moldable scheduling, the choice of job partition size is determined by the scheduler, using information about job scalability characteristics. We explore the role of job efficiency in moldable scheduling, through the development of a scheduling scheme that utilizes job efficiency information. The algorithm is able to improve the average turnaround time, but requires tuning of parameters. Using this exploration as motivation, we then develop an iterative scheme that avoids the need for any parameter tuning. The iterative scheme performs an intelligent, heuristic based search for a schedule that minimizes average turnaround time. It is shown to perform better than other recently proposed moldable job scheduling schemes, with good response times for both the small and large jobs, when evaluated with different workloads.

## 1 Introduction

Parallel job scheduling in a space-shared environment[1–5] is a research topic that has received a large amount of attention. Traditional approaches to job scheduling operate under the principle that jobs are *rigid* — that they are submitted to run on a certain number of processors, and that number is inflexible. Previously considered rigid scheduling schemes range from an early and simple first-come-first-serve (FCFS) strategy, which suffers from severe fragmentation and leads to poor utilization, to current backfilling policies which attempt to reduce the number of wasted cycles. Backfilling creates reservations for $N$ jobs from a sorted queue (often based on arrival time, job size, or current wait time), and then allow jobs to start "out of order" provided that no reservations are violated. Variations of $N$, such as $N = 1$ (aggressive or EASY backfilling) or $N = \infty$ (conservative backfilling) exhibit different behaviors and have been studied in detail. The vast majority of this work assumes that the user provides the number of nodes the job must run on as well as the job's estimated runtime.

However, many jobs do not actually require a specific number of processors; they can run on a range of processors. This range may be limited by constraints

due to the nature of the job. For example, a job may require a minimum number of processors (possibly for memory or other hardware constraints), or it may not be able to effectively use a large number of processors. Thus, the user must balance these factors when determining the number of processors to request from the scheduler. In addition, in order to achieve a satisfactory wait time, the user must also consider the state of the job queue, the running jobs, and the scheduling policy in place.

In recent work, there has been interest in moldable scheduling, an alternative model to the traditional rigid scheme. In a moldable scheme, a job is submitted by the user accompanied by a range of processor choices and run times or the speedup characteristics and constraints of the job. In this way, the scheduler is given the ability to make the final decision regarding the size of the partition the job is given. In such a scheme, the increased flexibility the scheduler is afforded allows it to not only provide the user with a better response time than the rigid case but also be better suited to adapt to changes of job mix and load.

A fundamental issue in moldable job scheduling is the determination of the partition size for each job. Cirne [6, 7] proposed and evaluated a moldable scheduling strategy using a greedy submit-time determination of each job's partition size. Later studies [8] showed that under a number of circumstances, a greedy strategy was problematic. Improved schemes were proposed and evaluated [9], but a shortcoming of previously proposed approaches is that the scalability of jobs is not taken into consideration. Given two similarly sized jobs with different scalabilities that are submitted at the same time, clearly it would be desirable to preferentially allocate more processors to the more scalable job. However, job mixes typically contain jobs with very different sizes. This paper addresses the issue of incorporating consideration of job scalability into a moldable scheduling strategy and demonstrates that the the importance of efficiency varies with respect to the characteristics of the workload a scheduler encounters. With this knowledge in hand, an iterative scheduling scheme is introduced which eliminates the need for scheduler parameterization based on workload characteristics and implicitly considers efficiency.

The remainder of the paper is organized as follows: Section 2 discusses related moldable job scheduling work. Section 3 describes the event-based simulator as well as the workloads used. Section 4 discusses the effects of "overbooking" introduced in previous work in a moldable scheduling model. Section 5 explores a scheme which uses efficiency and overbooking to outperform schemes which ignore job scalability. Section 6 introduces an iterative scheduling strategy which eliminates the need for tunable parameters. Finally, section 7 concludes the paper.

## 2 Related Work

There has been extensive research on parallel job scheduling in a non-preemptive space shared environment [1, 2, 4, 10–12]. Much of the recent work focuses on scheduling "rigid" jobs, even though jobs may be able to run on a range of

partition sizes. Previous work that focuses on moldable job scheduling aims primarily to minimize makespan or is set in the context of offline scheduling. Further, the realistic workloads [13] available today were not available when previous research into moldable scheduling was undertaken. This paper focuses on minimizing average turnaround time in an online scenario using realistic workloads.

Du and Leung [14] introduce a "Parallel Task System" (PST) for moldable jobs. The system is comprised of $m$ processors, and $n$ moldable jobs, whose speedups are assumed to be non-decreasing functions. They show that finding the minimal completion time for a PST is NP-hard. Krishnamurti and Ma [15] develop an offline approximation algorithm that attempts to minimize the makespan of a set of moldable tasks. The number of tasks is defined to be less than the number of partitions and the number of partitions is bounded. They propose an algorithm that incrementally reduces the execution time of the longest job. Other work studied the problem of reducing the makespan in an offline, multi-resource context [16, 17] while others assumed processor subset constraints [18, 19].

Eager, Zahorjanm, and Lazowska [20] suggest using the average parallelism of each task as a basis for processor allocation. They do not propose detailed scheduling algorithms. Ghosal, Serazzi, and Tripathi [21] extend the Eager et. al. work by introducing the concept of the processor working set (PWS). The PWS maximizes the number of processors that a job can efficiently use. The scheduling algorithms developed increase the average "power" [20] of the schedule. They develop online algorithms based on PWS for a setting similar to that of this paper.

Kleinrock and Huang [22] determine the number of processors to allocate in a parallel system where only one job can be executing at any given time. Again, the goal is to maximize power. This system is clearly not ideal for minimizing average turnaround time, as jobs are run sequentially in an FCFS manner.

Mccann, Vaswani, and Zahorjan [23] present a policy for a multi-processor system where jobs which can be resized dynamically (malleable). The scheduling policy transfers processors between running jobs based on the current parallelism of a job.

Sevick [24] provides a generic scheduling algorithm designed to reduce the average turnaround time in a wide range of environments (e.g., preemptive, non-preemptive, online, offline). The algorithm, based on Least Work First, determines a number of tasks to start simultaneously and then uses heuristics to assign each of the chosen tasks a set of processors.

Rosti et. al. [25] perform an analysis of non-work conserving scheduling algorithms. The analysis highlights the importance of realistic workload models when evaluating moldable schedulers. The non-work conserving algorithms are effective when there is large variance in the workload trace (as seen in real workloads) and with varying job types (as seen in real workloads). Non-work conserving algorithms outperform work conserving algorithms for the realistic workloads considered.

Downey [26, 27] presents a careful analysis of job characteristics and mix in real traces; this analysis [26] is used to create predictors for the queue time of jobs in synthetic workloads. Downey describes a moldable scheduling scheme which aims to optimize the performance of each job by determining a partition size $n$ such that the run time on $n$ processors plus the predicted queue time on $n$ processors is minimized. However, jobs are scheduled in a strict first-come-first-serve order which, again, hinders the ability of the system in improve average user metrics. Also, the greedy selection of partition size for individual jobs may harm the performance of other jobs in the system.

Downey [27] examines the performance of existing algorithms [28, 29] under his workload model. He defines two variations of moldable schemes—those that make greedy decisions for individual jobs, resulting in smaller partition sizes, and those that schedule jobs on only the "ideal" number of processors that each algorithm chooses. Both variations suffers from the issue described above and from the strict first-come-first-serve order imposed on the scheduler.

Cirne et. al. [6, 7] proposed a submit-time-based algorithm for moldable scheduling, where the desired processor allocation is decided upon submission to the scheduler in order to minimize response time. Once the desired allocation is determined the scheduler functions essentially the same as in the rigid case. As such, the scheduler is not able to take into account the inherently dynamic information about jobs and new job arrivals. Also, each job makes a greedy decision, which may not be a wise global decision [8]. However, using simulations and moldable traces based on real rigid traces, Cirne et. al. were able to show that their moldable scheduler can outperform a standard rigid parallel job scheduler.

Srinivasan et. al. [9, 8] use lazy processor allocation, delaying this allocation decision until schedule time. This allows the scheduler to obtain more information regarding job runtimes and job arrivals before finalizing the number of processors a job will run on. In this context, an unbounded greedy choice will not lead to a good schedule. Therefore, techniques to limit the number of processors a job can take are developed. The authors are able to show that their new methods can improve the schedule for many moldable workloads.

## 3 Simulation Setup

This work uses an event based simulator in which we are able to evaluate proposed scheduling policies using varying workload characteristics. The simulator uses workload traces in the Standard Workload Format [13], which can be obtained from Dror Feitelson's publicly available Parallel Workload Archive [13]. This allows us to perform multiple simulations on identical workloads in order to achieve comparable results across proposed scheduling policies.

### 3.1 Workload Generation

The simulations were run with workloads based on a trace from a 512-node IBM SP2 system at the Cornell Theory Center (CTC) obtained from Feitelson's workload archive. The trace, supplied in the Standard Workload Format, contains the

submit time, number of processors, actual runtime, and user estimated runtime of each job. To generate different offered workloads we multiply both the user supplied runtime estimate and the actual runtime by a suitable factor to achieve the desired offered load. As an example, assume that the original trace had a utilization of 65%. To achieve an offered utilization of 90%, the actual runtime and the estimated runtime are multiplied by a factor of 0.9/0.65. We use this method in lieu of shrinking the inter-arrival time between jobs to keep the duration of the trace consistent. In all simulations, the scheduler uses the runtime estimates provided by the user for scheduling purposes.

The data presented in the paper shows effective load, which is the load after adjusting for the scalability of the jobs. For instance, assume a job originally ran for 1000 seconds on 5 processors and had an efficiency of 50% (using our scalability model). Then the job contributes 2500 processor seconds to the effective load. In other words, the effective load represents the load for all jobs assuming the scheduler is able to run the jobs with ideal efficiency.

The trace used, as well as every other trace that we are aware of, does not contain any information regarding the scalability of the jobs. Therefore, we use the Downey model [30] of speedup for parallel programs and assign speedup characteristics to a job either by using fixed values or a random distribution.

### 3.2 The Downey Model

Downey's work [30] describes a model of speedup for parallel jobs. Speedup is defined as the ratio of the job's runtime on a single processor to the job's runtime on $n$ processors. If $L$ is the sequential runtime of the job and $T(n)$ is the runtime of the job on $n$ processors, then $S(n) = L/T(n)$ where $S(n)$ is the speedup of the job. Downey's model is a non-linear function of two parameters:

- $A$ denotes the average parallelism of a job and is a measure of the maximum speedup that the job can achieve.
- $\sigma$ (*sigma*) is an approximation of the coefficient of variance in parallelism within a job. It determines how close to linear the speedup is. A value of 0 indicates linear speedup and higher values indicate greater deviation from the linear case. Previous work has shown that a sigma between 0 and 2 can be expected for many workloads [27].

Downey's speedup function is defined as follows:

For low variance ($\sigma \leq 1$)

$$S(n) = \begin{cases} \frac{An}{A + \sigma(n-1)/2} & 1 \leq n \leq A \\ \frac{An}{\sigma(A-1/2) + n(1-\sigma/2)} & A \leq n \leq 2A-1 \\ A & n \geq 2A-1 \end{cases}$$

and for high variance ($\sigma \geq 1$)

$$S(n) = \begin{cases} \frac{nA(\sigma+1)}{\sigma(n+A-1)+A} & 1 \leq n \leq A + A\sigma - \sigma \\ A & n \geq A + A\sigma - \sigma \end{cases}$$

## 4 Fair-share Allocation and Overbooking

In this section, we review the fair-share strategy proposed in [8] along with an examination of the effect of varying the "weight factor" used in the fair-share schemes and how it affects jobs with different speedup characteristics.

### 4.1 Fair-share Based Allocation

The fundamental problem with using an unrestricted greedy approach to choose partition sizes for jobs is that most jobs tend to choose very large partition sizes. In the extreme case, this degenerates to a scenario where each job chooses a partition size equal to the number of processors in the system, with jobs being run in FIFO order. In order to rectify this problem, fair-share-based limits were introduced [8]. Fair-share-based schemes impose an upper bound on a job's allocation based on its fractional weight (resource requirement in processor-seconds) in the mix of jobs. The partition size for each job is then chosen to optimize its turnaround time, subject to its fair-share upper bound. A proportional-share limit was first evaluated [8], where the upper-bound for a job's partition size was set in direct proportion to the job's weight. A later study [9] showed that better turnaround times were achieved by using a "square-root" based fair-share limit, where the bound was set in proportion to the square root of job's weight: $Weight\ fraction\ of\ job\ i = \frac{\sqrt{Weight\ of\ job\ i}}{\sum_{j \in jobs} \sqrt{Weight\ of\ job\ j}}$.

We restrict our discussion of the fair-share moldable scheduling schemes to the schedule-time aggressive scheme, where the backfilling policy allows for $N = 1$ reservations from the queue and the decision of partition size is delayed until reservation time.

Srinivasan et. al. [8, 9] use an additional system-wide "weight factor" which is multiplied with the weight fraction to raise the limit on the number of processors allocated for all jobs. Rajan [31] further examined the use of a system-wide weight factor. We will call this the *overbooking factor* and it will be the focus of our examination. Specifically, we describe how changes in the overbooking factor can benefit or harm jobs with different speedup characteristics and weight fractions.

### 4.2 Perfect Scalability

The "overbooking factor" (ObF) is a multiplicative factor used to scale up the weight-fraction of a job in determining the upper bound on partition size. With an overbooking factor of one (i.e., no overbooking), the sum of fair-share based partition limits of all jobs add up to the total number of available processors. With an overbooking factor of two, the sum of upper bounds add up to twice the

number of processors, etc. As ObF increases, average turnaround time improves at low load, but worsens at high load. An increase in ObF has several effects:

- It tends to increase the average number of waiting jobs in the queue; since each job's maximum partition size is increased, the number of jobs that can concurrently run decreases. This causes the average turnaround time of light jobs to increase, since turn-around of these jobs is dominated by queue time.
- The average run-time of heavy jobs tends to decrease, causing the average response time to also decrease, since it is dominated by the run-time and not queue time.
- When several similarly sized jobs are present, where as with ObF of one, they could all run concurrently, with higher ObF their execution gets serialized, but lowers average response time. For example, with two identically sized jobs, with ObF of one, they both could run concurrently using half the processors each. With ObF of two, each job would run using all the processors for one half the time, giving an average response time that is (T/2 + T)/2, i.e., 75% of that with ObF=1.

As the system approached saturation, the queue size increases rapidly with high ObF, causing the deterioration of performance of light jobs to overshadow the benefits of high ObF for the heavy jobs.

### 4.3 Non-ideal Job Scalability

The effect of the overbooking factor on performance changes under non-ideal scalability conditions [31]. Unlike the case where all jobs share a value of $\sigma = 0$ (perfect scalability), when $\sigma$ is higher (poorer job scalability), it can be seen that increasing ObF causes an increase in average response time, even at low loads. This is because a higher ObF causes jobs to receive wider partition choices, and therefore uses more processor cycles for job execution than narrower partition choices. The detrimental effect of increasing ObF is more pronounced at high loads, where the waste of processor cycles by inefficient wide jobs causes an increase in queuing delays. This points to a need to take job scalability into consideration when performing moldable job scheduling.

## 5 Efficiency Considerations

In the previous section, we considered how overbooking, by itself, can either be helpful or harmful to the average response time of jobs within the fair-share scheme and that a job's efficiency needs to be taken into consideration when computing its processor allocation. In this section, we describe a scheduling policy that corrects for this oversight by optimizing for efficiency.

We must be careful when discussing "optimal efficiency," though. A schedule that is optimally efficient for the whole would be a schedule where every job is simply allocated a single processor. This schedule, while maximizing efficiency

and throughput, obviously falls short of providing users with adequate response times.

Therefore we choose to maximize the "instantaneous" effective utilization. This is the sum of the number of processors a job runs on $N_i$ multiplied by the efficiency of that job on that number of processors $e(N_i)$ for all jobs. We can see that maximizing the effective utilization is then the same as maximizing the speedup $s(N_i)$ of all jobs ($\sum[N_i * e(N_i)] = \sum[N_i * \frac{s(N_i)}{N_i}] = \sum[s(N_i)]$). In situations where there are less jobs than processors, each job's partition size will be computed such that processors are being used in a locally optimal manner.

### 5.1 Incorporating Efficiency into Fairshare

An optimally efficient schedule is one that makes the most efficient use of available cycles. However, response time is an important metric, so we still need to incorporate job size. Thus the thrust of this scheme is to close the gap between the weight-based allocation of the fair-share scheme, where jobs receive a proportion of the system ignoring how well they scale to fit their allocation, and an efficiency-based allocation, where the relative sizes of the jobs are ignored and the effective utilization is optimized.

In order to maintain this balance, we define a system-wide *efficiency factor* (EF). The efficiency factor limits how much a job's maximum allocation can change from its fair-share limit:
$\max(1, FairshareLimit * (1 + EF)) \leq$
$EfficiencyLimit \leq min(SystemSize, FairshareLimit * (1 - EF))$

In order to maximize the "instantaneous" effective utilization, or the sum of the speedups of all jobs, we take processors away from the fair share limit of the job with the smallest slope of its speedup curve for its current allocated limit and give processors to the job with the highest slope of its speedup curve, this leads towards equivalent derivatives of the speedup.

The algorithm for determining a job's maximum processor allocation is shown in Figure 1.

By including a job's speedup characteristics in its allocation we are able to take advantage of the benefits of overbooking for jobs that scale well enough to efficiently use additional processors without wasting processors on jobs that cannot efficiently use them.

### 5.2 Experimental Results

We evaluated our algorithm over a set of input traces, varying the efficiency and overbooking parameters of the scheduler. Traces were modified to contain speedup characteristics of jobs subject to the Downey model. For the sake of brevity we limit our discussion to overbooking factors of 1 and 4 and efficiency factors of 0, 0.5, and 1. We show two sets of results — one which assumes that each job can scale to the size of the system ($A$ = system size) and another that assigns each job a random value of $A$ from a random uniform distribution

```
void selectMaxProcessorLimit(){
     OrderedList jobs;
     /** All jobs start with the
     original fair share limit **/
     foreach j in jobs{
        j.nodeLimit = getFairshare(j);
        j.maxNodesLimit =
           min(SYS_SIZE, (1+EF)*j.nodeLimit);
        j.minNodeLimit =
           max(1,(1-EF)*j.nodeLimit);
     }

     /**Transfer processors from jobs with a small
        speedup slope to jobs with a high speedup
        slope, to optimize instantaneous effective
        utilization **/
     while(!complete){
        complete=true;
        sortBySlope(jobs);
        while(!canMove(sJob=jobs.getFirst()))
                jobs.removeFirst();
        while(!canMove(lJob=jobs.getLast()))
                jobs.removeLast();
        if(sJob.getSlope()<lJob.getSlope()){
                sJob.nodeLimit--;
                lJob.nodeLimit++;
                sortBySlope(jobs);
                complete=false;
        }
     }
}

/** Each job's original limit is between the max and
    the min.  During each call to selectMaxProcessorLimit
    each job will either gain or lose processor (not both). **/
boolean canMove(Job j){
  if(j.nodeLimit >= j.maxNodeLimit ||
        j.nodeLimit <=j.minNodeLimit){
      return false;
  }
  return true;
}
```

**Fig. 1.** The efficiency based moldable scheduling algorithm

between 1 and 2 times the number processors the job requests in the unmodified trace. In both sets of results, we chose the value of $\sigma$ for each job from a uniform distribution between 0 and 2.

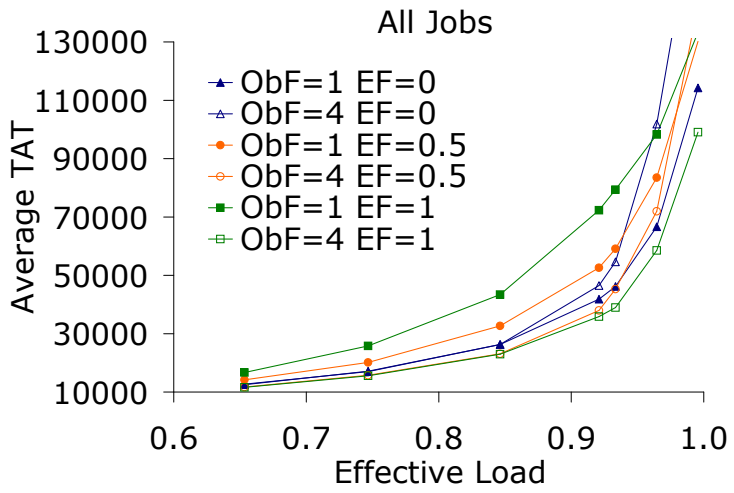


**Fig. 2.** With an increased overbooking factor, increasing the efficiency factor improves average turnaround time

Figure 2 shows that in the first scalability scenario ($A$ = system size), a high overbooking factor and an efficiency sensitive strategy (EF=1.0) outperforms other scheduling strategies; the overall average turnaround time (TAT) is better than when using the fair-share alone (EF=0). We also note that increasing the efficiency factor in low overbooking hurts the average turnaround time due to poor utilization and a negative effect on large jobs (shown below).

In Figure 3 we examine the effects of overbooking on various job sizes. In general, we see that small jobs (200-3,200 processor seconds and 3,200 to 100,000 processor seconds) benefit from a low overbooking factor. When overbooking is low, large jobs (greater than 2,000,000 processor seconds) have limited partition sizes and processors remain free for small jobs. As the turnaround time of small jobs is dominated largely by time spent waiting in the queue, any increase in their runtime due to a smaller maximum partition size is negligible. We also see that a high efficiency factor boosts the performance of small jobs; they are able to gain processors at the expense of larger and more inefficient jobs.

As a job's size grows, its turnaround time becomes less dominated by the time it spends waiting in the queue and more dominated by its run time. The medium sized jobs illustrate the point where this transition begins to occur; the effect of a low overbooking factor and high efficiency factor becomes less pronounced. Allowing large jobs to claim more processors is the dominating factor in their

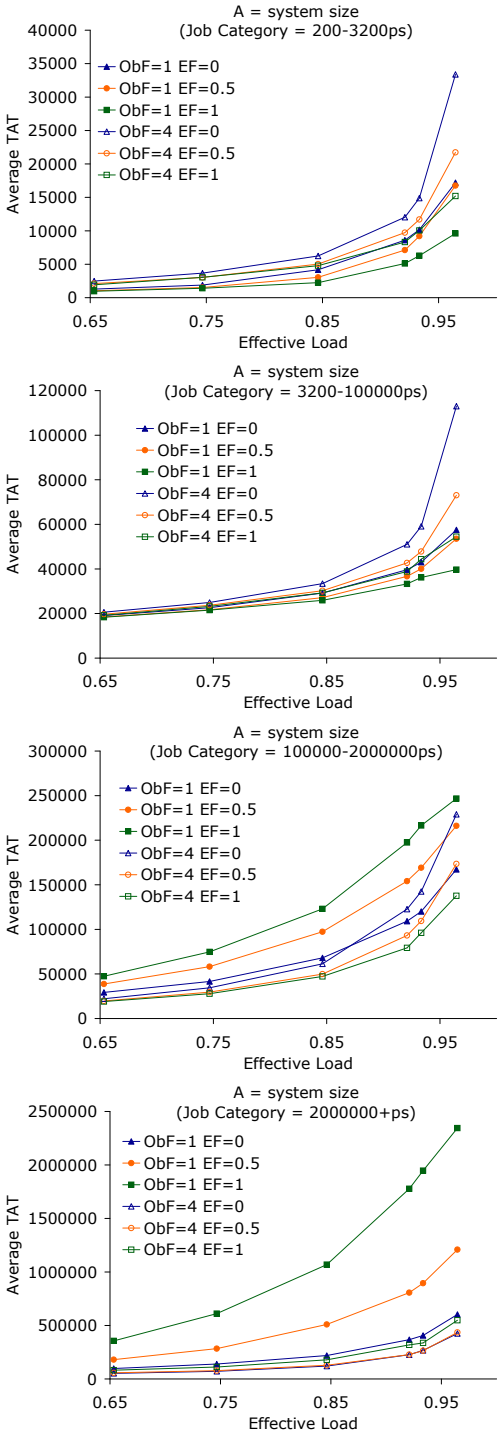


**Fig. 3.** Small jobs benefit from low overbooking and higher efficiency consideration, as their turnaround time is dominated by wait time. As job size grows, the benefit of efficiency consideration is diminished and eventually becomes detrimental to large jobs. However, in order for the larger jobs to perform well, a large overbooking factor is required.

turnaround time, as they can afford to wait in the queue to reduce runtime. A high overbooking factor plays the biggest role with these large jobs and the efficiency factor has little effect on their performance. However, when a *low* overbooking factor is used, a high efficiency factor becomes detrimental to large jobs — precisely for the same reason this scenario was beneficial for small jobs.

The extreme end of the spectrum illustrates this clearly; efficiency plays almost no role when combined with a high overbooking factor for the largest jobs in the system.

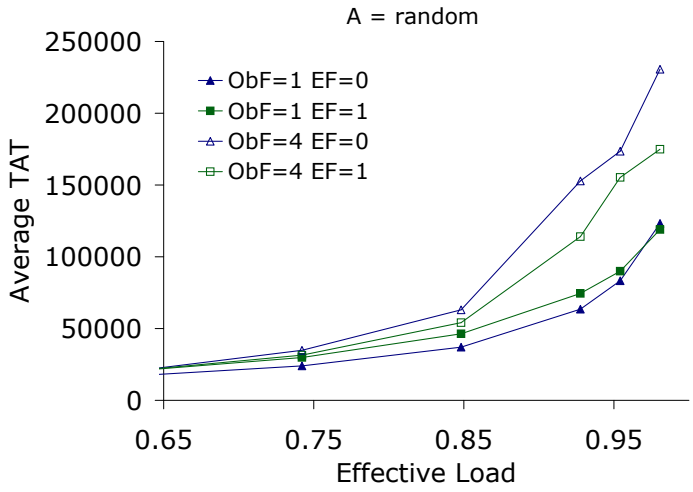


**Fig. 4.** With more variably scaling jobs, neither efficiency nor overbooking achieve better performance

To provide further contrast from the scenario presented in Figure 2, Figure 4 presents the situation where jobs do not all share a uniform maximum partition size. In this perhaps more realistic situation, each job's value of $A$ is chosen randomly between 1 and 2 times the partition size requested in the original trace. Now that jobs do not all scale to the size of the system, we notice that the scheme which performs the best uses the plain fair-share scheme with no overbooking (ObF=1) and doesn't take efficiency into consideration at all (EF=0)! With overbooking, jobs can no longer effectively use the once-helpful large partitions given to them in the fair-share scheme and essentially waste machine cycles. The effects of this wastage become even more pronounced in higher load. Taking efficiency into account can reduce the detrimental effect of high overbooking, but at the cost of severely reducing the allocation of all but the most scalable jobs.

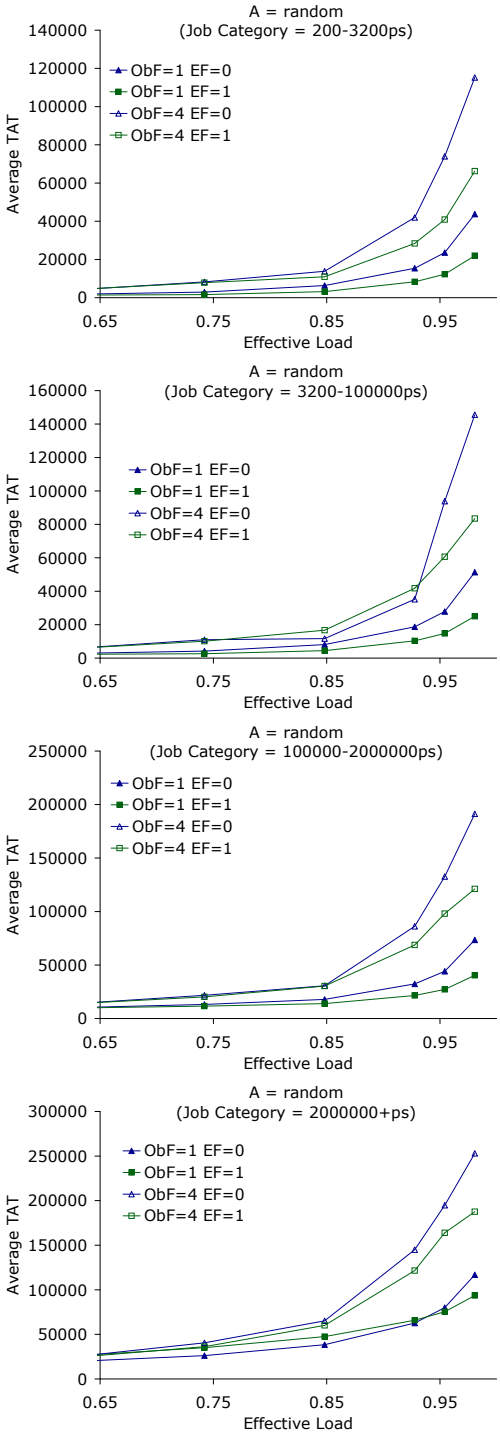


**Fig. 5.** With more variability in the speedup characteristics of jobs, overbooking is no longer helpful. Using the efficiency-based scheme is helpful for most jobs

Figure 5 shows category based results for this scalability scenario. Here it can be seen that taking efficiency into account is helpful for all job sizes. Due to the poor scalability, overbooking is detrimental to all job categories.

The results here make it clear that the choice of an effective overbooking factor and efficiency factor not only depend on the relative size of jobs in the system, but also their relative scalabilities and overall system load. With good overall scalability, using a high overbooking factor in combination with the efficiency based scheme provides the best results. However, with poorer scalability, a higher overbooking factor is very detrimental.

## 6 An Iterative Approach for Moldable Scheduling

The efficiency-sensitive moldable scheduling approach presented in the previous section was seen to provide benefits over the base fair-share strategy (EF=0). However, a difficulty with the approach is the need to choose appropriate parameters — the choice of the best overbooking factor and efficiency factor are dependent on the overall scalability characteristics of the job mix. If a job mix were to contain jobs of relatively uniform weight and similar maximum partition size, a high overbooking factor produces the best response times. However, if jobs differ considerably in their maximum partition size, a high overbooking factor leads to poor performance. It is equally problematic to choose the efficiency factor. Small jobs benefit from efficiency-based schemes but large jobs suffer under the same schemes.

A desirable moldable scheduling strategy would inherently take into account the efficiency, job size, system load and job mix without the need to "tune" parameters. In this section, we develop an iterative backfilling approach that does so.

Before describing the algorithm, we first provide a high level contrast of this approach with the previous section's strategy. The previous section's moldable scheduler associates a maximum allowable partition size with each job and uses a greedy scheduling strategy to choose an actual partition size (subject to a job's upper limit) in order to minimize response time. A job's size limit was determined using a fair-share proportion adjusted via the overbooking and efficiency factors. Although the idea of incorporating efficiency was effective, the problem with the approach was that the best choice for the overbooking factor and efficiency factor was dependent on the job mix. In order to avoid this problem, we consider a completely different approach to moldable scheduling — instead of simply setting an upper bound on job partition sizes, generate schedules incrementally and iteratively using global information.

### 6.1 The Iterative Algorithm

Our iterative algorithm begins by giving each job an initial minimal partition of one processor. A conservative backfilling schedule is generated; this schedule is then iteratively modified by giving a processor to the "most worthy" job — the

job that, if given an additional processor, has the greatest decrease in runtime. If the addition of a processor to the most worthy job decreased the *average* response time of the schedule, the addition is accepted, otherwise not. Note that a job given an additional processor may have a start time *later* than previously reserved if its "waiting" allows it to improve the average turnaround time of the schedule.

```
1.   void iterativeNodeAssignment(OrderedList reservedJobs){
2.      unmark all jobs in the reservedJobs list and
           set partition sizes to 1
3.      while(unmarked jobs exist)
4.         find unmarked candidate job j (see line 15)
5.         add one to partition size of job j
6.         create a conservative schedule for all jobs
7.         if(average turnaround time did not improve)
8.            mark job j
9.            decrement partition size of candidate job j
10.           create a conservative schedule for all jobs
11.        end if
12.     end while
13.  }
14.
15.  Job findUnmarkedCandidate(OrderedList reservedJobs){
16.     set bestImprovement to zero
17.     for each unmarked job j in the reserved job list
18.        let n be the current node assignment of job j
19.        let i be the expected runtime on n processors
20.        let i' be the expected runtime on n+1 processors
21.        if(i - i' > bestImprovement)
22.           set bestImprovement to i - i'
23.           set bestJob to j
24.     end for
25.     return bestJob
26.   }
```

**Fig. 6.** The iterative moldable scheduling algorithm

Fig. 6 shows pseudocode for the iterative algorithm. Initially, each job is assigned one node. This allocation results in optimal per job efficiency, but may result in poor average turnaround and/or system utilization.

The next step (lines 3 to 12) searches for a schedule with an improved average turnaround time. Step 4 chooses the job which will benefit the most from receiving an extra processor. This job is a "good" candidate to try increasing its processor allocation. Steps 5 to 11 determine if the increased allocation results in a better schedule. If the increase produces a worse schedule, the job is marked as a "bad" choice and the remaining jobs are considered.

This approach takes all the aspects discussed previously into account: load, scalability, job size, and utilization. If a job is small, the improvement from adding a processor will be minimal, and thus it will be less likely to receive an increased allocation. Likewise, if a job scales poorly, it will benefit less from receiving more processors, and will be less likely to be chosen as the candidate. If the load is low, "wider" jobs will result in a better average turnaround time, and wider allocations will be given. If the load is high, increasing the allocation of poorly scalable jobs will increase average turnaround time, and such jobs will be left "narrow". Finally, the system achieves good utilization, as processors will not be wasted unless there is no work to be done or using the processor reduces the average turnaround time.

Using turnaround time as the scheduling metric, selecting the job with the best absolute improvement in expected runtime, and iteratively searching and marking jobs provides a flexible, adaptable algorithm that is able to handle a diverse set of job scalability characteristics. This flexibility and adaptivity present here is not achievable with other algorithms without the addition of a complicated and dynamic tuning system, which while plausible, would not have the elegance of the simple iterative scheme.

### 6.2 Results

In this section we compare the iterative algorithm described in Figure 6 to schemes which have been shown to be effective in certain contexts. The results show that the iterative algorithm is indeed able to perform very well in a variety of contexts and is competitive with the best previous algorithm (which varies when the scalability varies).

Figure 7 is the case where jobs scale to the size of the system ($A$ = system size). As discussed previously, overbooking alone is not helpful as its generous allocation of processors leads to "wastage" of resources. Explicitly taking efficiency into account when choosing job widths allows more scalable jobs to receive more processors than non-scalable jobs, proving a better average turnaround time. In contrast, the iterative scheme is able to choose the "correct" job sizes, implicitly considering job size and scalability, and is better than even with the best of the previous scheduling scheme (ObF = 4, EF = 1). This search does come at a small cost: the scheduling time increases to a few hundred milliseconds. However, this cost is much lower than the time between useful scheduling events.

Figure 8 shows the iterative scheme's performance within job size categories. We can see that small jobs actually receive better performance in the iterative scheme than in other schemes, which was a major issues with earlier moldable scheduling strategies. Further, this improvement does not coincide with a deterioration in performance for the largest jobs. This is because the larger jobs are more likely to receive more processors — but this allocation is limited by a large job's effect on the other jobs in the queue. The iterative scheme is able to balance the needs of both small and large jobs.

Finally, Figure 9 shows the situation where each job's $A$ value varies, as previously described. Recall that in this situation, the poor scalability becomes a prob-

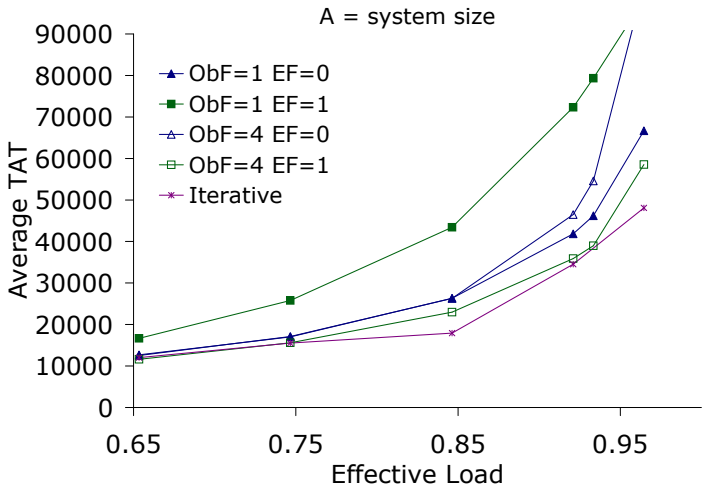


**Fig. 7.** In the situation where jobs scale to the size of the system, the iterative scheme outperforms even the best previous scheme

lem for the schemes discussed. Increasing the overbooking factor was not helpful, nor was explicitly considering efficiency. Therefore, it was beneficial to use an efficiency factor of 0 or 1 and no overbooking. However, the iterative scheme outperforms all schemes previously considered — without having to "tune" any parameters. Figure 10 illustrates that the improvement in performance carries across all job size categories as well.

### 6.3 Discussion

The iterative approach describe is a flexible moldable scheduler which is able to adapt to a variety of job scalabilities without the need to "fine tune" parameters. The scheme performs well across various loads and small jobs do not receive poor performance in order to improve the performance of large jobs, as was a problem in previous work. The overall performance of the iterative algorithm competes with, and in many cases outperforms, both new and prior schemes that require tuning.

## 7 Conclusions

Current schedulers require users to examine the set of queued and running jobs when deciding upon a partition size for a job. It is left up to them to decide whether to request few resources and reduce the wait time of their job or request more resources and reduce the job's run time. A moldable scheduler shifts this responsibility from the user to the scheduler.

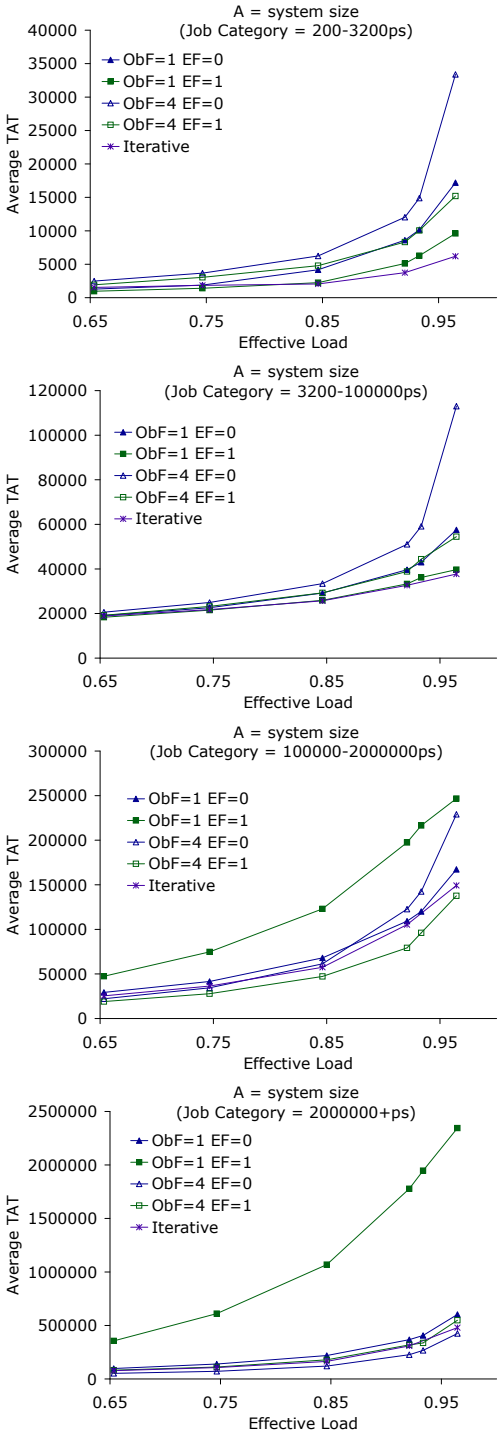


**Fig. 8.** The iterative scheme is able to mirror the performance of the best overbooking efficiency choices for different job categories

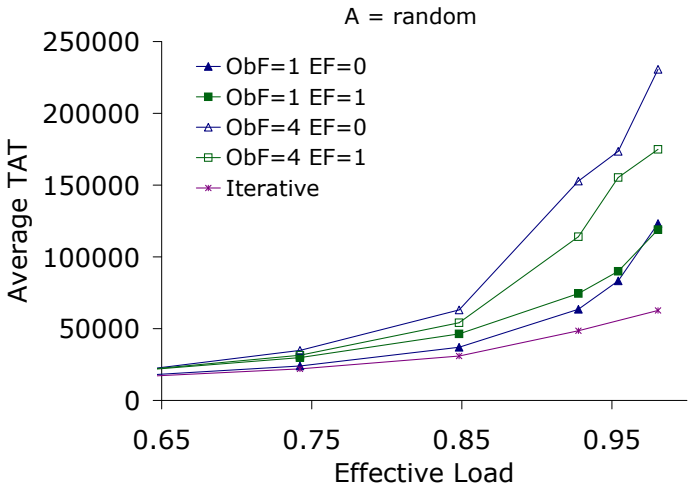


**Fig. 9.** When jobs vary more widely in scalability, the iterative scheme performs better than all previous schemes, especially as load increases

The work presented in this paper examines the effects of the overbooking factor introduced in previous work and demonstrates that overbooking in workloads consisting of jobs which scale well is beneficial, while overbooking can have a negative affect in workloads consisting of jobs of varying scalability. Additionally, we explore the role efficiency can play in the selection of partition size and how the explicit consideration of job scalability can either reduce or increase the response time of a system, depending on job mix and scalability. Additionally, the "best" scheme for a particular job depends on the job's size. The results show that in order to achieve good performance, parameters must be heavily tuned according to expected job characteristics.

We introduce an iterative scheme to eliminate the need for fine grained performance tuning. The approach provides a flexible, robust moldable scheduling policy that provides good performance in all situation studied. Without tuning, the scheme mirrors and, in some cases, improves upon the response time of the best of the efficiency/overbooking schemes across drastically differing scalability scenarios.

## References

1. Feitelson, D.: Workshops on job scheduling strategies for parallel processing. (www.cs.huji.ac.il/ feit/parsched/)
2. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In Feitelson, D.G., Rudolph, L., eds.: JSSPP. Volume 1291 of Lecture Notes in Computer Science., Springer (1997) 1–34

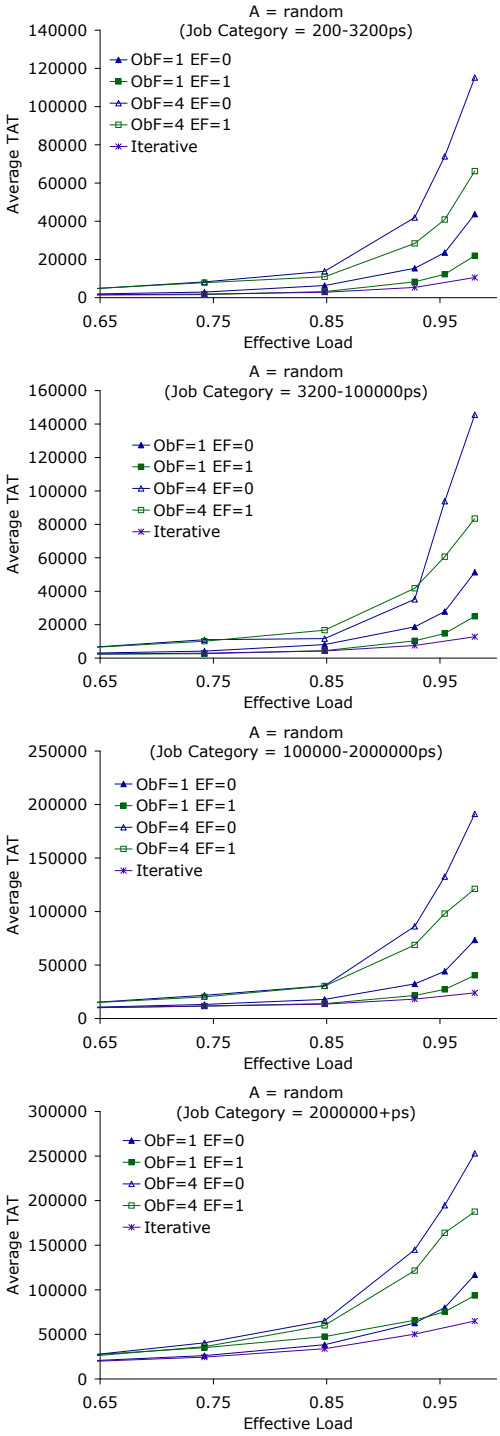


**Fig. 10.** Even when the scalability of jobs has more variety ($A$ random), the iterative scheme is able to match the performance of the best overbooking and efficiency choices

3. Weil, A.M., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6) (2001) 529–543
4. Skovira, J., Chan, W., Zhou, H., Lifka, D.A.: The easy - loadleveler api project. In Feitelson, D.G., Rudolph, L., eds.: JSSPP. Volume 1162 of Lecture Notes in Computer Science., Springer (1996) 41–47
5. Frachtenberg, E., Feitelson, D.G., Petrini, F., Fernandez, J.: Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In: IPDPS. Number 17 (2003)
6. Cirne, W., Berman, F.: Adaptive selection of partition size for supercomputer requests. In Feitelson, D.G., Rudolph, L., eds.: JSSPP. Volume 1911 of Lecture Notes in Computer Science., Springer (2000) 187–208
7. Cirne, W., Berman, F.: Using moldability to improve the performance of supercomputer jobs. J. Parallel Distrib. Comput. **62**(10) (2002) 1571–1601
8. Srinivasan, S., Subramani, V., Kettimuthu, R., Holenarsipur, P., Sadayappan, P.: Effective selection of partition sizes for moldable scheduling of parallel jobs. In Sahni, S., Prasanna, V.K., Shukla, U., eds.: HiPC. Volume 2552 of Lecture Notes in Computer Science., Springer (2002) 174–183
9. Srinivasan, S., Krishnamoorthy, S., Sadayappan, P.: A robust scheduling strategy for moldable scheduling of parallel jobs. In: CLUSTER, IEEE Computer Society (2003) 92–99
10. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: ICPP Workshops, IEEE Computer Society (2002) 514–522
11. Frachtenberg, E., Feitelson, D.G.: Pitfalls in parallel job scheduling evaluation. In Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U., eds.: JSSPP. Volume 3834 of Lecture Notes in Computer Science., Springer (2005) 257–282
12. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling - a status report. In Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., eds.: JSSPP. Volume 3277 of Lecture Notes in Computer Science., Springer (2004) 1–16
13. Feitelson, D.G.: Logs of real parallel workloads from production systems. (URL: http://www.cs.huji.ac.il/labs/parallel/workload/)
14. Du, J., Leung, J.Y.T.: Complexity of scheduling parallel task systems. SIAM J. Discret. Math. **2**(4) (1989) 473–487
15. Krishnamurti, R., Ma, E.: An approximation algorithm for scheduling tasks on varying partition sizes in partitionable multiprocessor systems. IEEE Transactions on Computers **41**(12) (1992) 1572–1579
16. Garey, M.R., Graham, R.L.: Bounds for multiprocessor scheduling with resource constraints. SIAM J. Comput. **4**(2) (1975) 187–200
17. Garey, M.R., Johnson, D.S.: Complexity results for multiprocessor scheduling under resource constraints. SIAM J. Comput. **4**(4) (1975) 397–411
18. Li, K., Cheng, K.H.: Job scheduling in partitionable mesh connected systems. In: ICPP (2). (1989) 65–72
19. Tuomenoksa, D.L., Siegel, H.J.: Task scheduling on the pasm parallel processing system. IEEE Trans. Software Eng. **11**(2) (1985) 145–157
20. Eager, D.L., Zahorjan, J., Lazowska, E.D.: Speedup versus efficiency in parallel systems. (1995) 76–91
21. Ghosal, D., Serazzi, G., Tripathi, S.K.: The processor working set and its use in scheduling multiprocessor systems. IEEE Trans. Softw. Eng. **17**(5) (1991) 443–453

22. Kleinrock, L., Huang, J.H.: On parallel processing systems: Amdahl's law generalized and some results on optimal design. IEEE Trans. Softw. Eng. **18**(5) (1992) 434–447
23. McCann, C., Vaswani, R., Zahorjan, J.: A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. ACM Trans. Comput. Syst. **11**(2) (1993) 146–178
24. Sevcik, K.C.: Application scheduling and processor allocation in multiprogrammed parallel processing systems. Technical Report CSRI-282, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1 (1993)
25. Rosti, E., Smirni, E., Serazzi, G., Dowdy, L.W.: Analysis of non-work-conserving processor partitioning policies. In Feitelson, D.G., Rudolph, L., eds.: JSSPP. Volume 949 of Lecture Notes in Computer Science., Springer (1995) 165–181
26. Downey, A.B.: Using queue time predictions for processor allocation. In Feitelson, D.G., Rudolph, L., eds.: Job Scheduling Strategies for Parallel Processing. Springer Verlag (1997) 35–57
27. Downey, A.B.: A parallel workload model and its implications for processor allocation. Cluster Computing **1**(1) (1998) 133–145
28. Sevcik, K.C.: Characterizations of parallelism in applications and their use in scheduling. SIGMETRICS Perform. Eval. Rev. **17**(1) (1989) 171–180
29. Chiang, S.H., Mansharamani, R.K., Vernon, M.K.: Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In: SIGMETRICS. (1994) 33–44
30. Downey, A.B.: A model for speedup of parallel programs. Technical Report CSD-97-933 (1997)
31. Rajan, A.: Evaluation of scheduling strategies for moldable parallel jobs. Master's thesis, The Ohio State University (2004)