The Load Rebalancing Problem ⋆

Gagan Aggarwal [a,1] Rajeev Motwani [a,2] An Zhu [a,3]

[a] *Department of Computer Science, Stanford University, Stanford, CA 94305.*

Abstract

In the classical load balancing or multiprocessor scheduling problem, we are given a sequence of jobs of varying sizes and are asked to assign each job to one of the m empty processors. A typical objective is to minimize the makespan, which is the load on the heaviest loaded processor. Since in most real world scenarios, the load is a dynamic measure, the initial assignment may not remain optimal over time. Motivated by such considerations in a variety of systems, we formulate the problem of *load rebalancing* — given a possibly suboptimal assignment of jobs to processors, relocate a set of the jobs so as to decrease the makespan. Specifically, the goal is to achieve the best possible makespan under the constraint that no more than k jobs are relocated. We also consider a generalization of this problem where there is an arbitrary cost function associated with each job's relocation. Since the problem is clearly NP-hard, we focus on approximation algorithms. We construct a sophisticated algorithm which achieves a 1.5-approximation, with near linear running time. We also show that the problem has a PTAS, thereby resolving the complexity issue. Finally, we investigate the approximability of several extensions of the load rebalancing model.

1 Introduction

Consider a set of web servers, each with a set of (virtual) websites. As information is collected about the usage of each website on each web server, it

⋆ A preliminary version of this paper appeared in 15^{th} Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2003.

Email addresses: gagan@cs.stanford.edu (Gagan Aggarwal), rajeev@cs.stanford.edu (Rajeev Motwani), anzhu@cs.stanford.edu (An Zhu).

[1] Supported by a Stanford Graduate Fellowship.
[2] Supported in part by NSF Grant IIS-0118173, an Okawa Foundation Research Grant, an SNRC grant, and grants from Microsoft and Veritas.
[3] Supported by a GRPW fellowship from Bell Labs, Lucent Technologies.

might become apparent that the load is not uniformly distributed across the web servers. An obvious solution would be to reassign websites to web servers so as to minimize the maximum load on a server.

This is a typical instance of the multiprocessor scheduling or the load balancing problem. However, moving websites from one server to another could incur substantial cost so it would be desirable to balance the load while minimizing the movement cost. There are also a variety of other problems involving task migration to balance load. The systems community has shown considerable interest in the problem of process migration in the context of scheduling jobs on multiple processors. Some of the benefits derived from process migration are utilization of processing time on an idling CPU, migrating a small job to the site of a very large data file, and improved reliability. See [12] for a survey on process migration by Nuttall. Lazowska, Eager, and Zahorjan have argued that the performance benefits of process migration are limited to unrealistic CPU-bound workloads [9]. Meanwhile, Harchol-Balter and Downey have done trace-driven simulations to claim that process migration is useful in real-world situations [6]. The key issue is whether the benefit derived from utilization of an idle CPU justifies the cost of migrating a process. Rudolph, Slivkin-Allalouf, and Upfal have attempted to devise process migration schemes which migrate only a *few* processes, assuming a unit-size job model [13]. In this paper, we formalize the notion of *few processes* and eliminate the assumption of unit-sized processes, making the model more realistic. There has also been work on modeling the proximity of processors, under which processes can be migrated to *nearby* processors only. Hu, Blake, and Emerson provided an elegant diffusive technique for load balancing in this model [7]. Ghosh et. al. proved that a local load balancing algorithm (under a similar notion of proximity of processors) achieves a fairly good global balance, while using *few* moves (they actually use the notion of time rather than moves) [4]. In their study, all jobs are assumed to be of equal size.

Motivated by the above, we formulate the problem of *load rebalancing* – given a possibly suboptimal assignment of jobs to processors, reassign jobs to different processors so as to minimize the makespan, by moving as few jobs as possible. More specifically, we look at the problem of minimizing the makespan, while migrating no more than k jobs, or incurring cost no more than B where each job's migration has an arbitrary cost. The load rebalancing problem can be viewed as a special case of the generalized assignment problem, where the goal is to minimize makespan within a given cost budget for jobs with processor-dependent costs and processing times. For arbitrary number of processors, the generalized assignment problem is MAX-SNP-hard; further, it is known that this problem cannot be approximated better than a factor of 1.5 [10]. The best positive result known is the 2-approximation algorithm due to Shmoys and Tardos [14], via linear programming. When the number of processors is fixed, the generalized assignment problem admits a PTAS, as described in the work of Efraimidis and Spirakis [2], Jansen and Porkolab [8], and Fishkin, Jansen, and Mastrolilli [3].

In this paper, we devise algorithms that are specific to the load rebalancing problem and obtain better results. We show that a simple variant of Graham's greedy heuristic [5] yields a 2-approximation. We then construct a more sophisticated algorithm which achieves a better approximation factor of 1.5 and still has the same $O(n \log n)$ running time bound as the greedy algorithm. In fact, we also manage to devise a PTAS for our problem; though the 1.5-approximation algorithm is much faster and hence, more likely to be useful in practice. We would like to point out that Linder and Shah [11] have implemented our algorithms for load balancing of web servers via website migration and have achieved promising experimental results. We also show that the problem of minimizing the relocation cost to achieve a given load is hard to approximate. Finally, we investigate the approximability of several extensions of the rebalancing problem, and show that all such extensions are at least MAX-SNP-hard.

The rest of this paper is organized as follows. We first formally define the problem, establish its NP-hardness and describe a 2-approximate greedy algorithm in Section 2. Then in Section 3 we present the 1.5-approximation algorithm. In Section 4, we present a PTAS for this problem. Finally in Section 5, we investigate the approximability of several extensions of the rebalancing model.

2 Preliminaries

We begin with a formal definition of the problem. Suppose there is a set of n jobs of varying sizes. For convenience we assume jobs are listed in decreasing order of their job sizes, i.e., $s_1 \geq s_2 \geq \cdots \geq s_n$. Initially, these jobs are assigned to a set of m processors $P_1, \ldots, P_m$. The *load* on a processor is the sum of the sizes of the jobs assigned to it.

Definition 1 [The Load Rebalancing Problem] *Given an assignment of the n jobs to m processors, and a positive integer k, relocate no more than k jobs so as to minimize the maximum load on a processor. More generally, we are given a cost function c_i which is the cost of relocating job i, with the constraint that the total relocation cost is bounded by a specified budget B.*

The problem is NP-complete via a reduction from multiprocessor scheduling (just set $k = n$). There is a simple reduction from the load rebalancing problem to the generalized assignment problem. Let c_{ij} denote the cost of assigning job

i to processor j in the generalized assignment setting. Then

$$c_{ij} = \begin{cases} 0 & \text{If job } i \text{ currently resides on processor } j, \\ c_j & \text{Otherwise.} \end{cases}$$

By the results of Shmoys and Tardos [14], we obtain a 2-approximation algorithm for the load rebalancing problem.

We first observe that for this specific problem, a much simpler algorithm achieves the same 2-approximation. Consider the following GREEDY algorithm, which is a simple variant of Graham's greedy algorithm for makespan.

Algorithm GREEDY**:**

(1) Repeat k times: From the current maximum-loaded processor, remove the largest job.
(2) Consider the k jobs removed in an arbitrary order. Place each of these jobs on the current minimum-loaded processor.

Let OPT denote the value of the optimal solution to the load rebalancing problem. Further, let G_1 and G_2 denote the maximum processor load after the first and second step, respectively.

Lemma 1 *$G_1 \leq OPT$.*

Proof: In each iteration of Step 1, GREEDY removes jobs from only those processors with a current load greater than G_1. Let the number of jobs removed by GREEDY from processor j be N_j. Now suppose, in contradiction to the claim above, $G_1 > OPT$. Since GREEDY removes the largest jobs from a processor, in order to reduce the maximum load to $OPT < G_1$, the optimal algorithm must have removed at least N_j jobs from processor j. In addition, consider the maximum-loaded processor j with load G_1 after Step 1 of GREEDY. In order to reduce the load to OPT or below on this processor, the optimal algorithm has to remove at least $N_j + 1$ jobs. Thus, the total number of removals made by the optimal algorithm is at least one more than the number of removals made by GREEDY. This contradicts the constraint that the optimal algorithm is allowed to make only k moves. □

Lemma 2 *$G_2 \leq (2 - \frac{1}{m})OPT$.*

Proof: Consider the maximum-loaded processor after Step 2, say j. Note that its load is G_2. If j did not receive any new jobs during Step 2, then $G_2 = G_1 \leq OPT$. Otherwise, let s denote the size of the last job added to processor j. Note that prior to the addition of the job of size s, processor j

was the processor of minimum load (otherwise the job would not have been assigned to it). Thus, the total load of all the jobs is at least $m(G_2 - s) + s$. Since the average load is a lower bound on OPT, we get $m \times OPT \geq m(G_2 - s) + s$. Quite clearly, s itself is a lower bound on OPT, giving $OPT \geq s$. Combining the two inequalities shows that $G_2 \leq (2 - \frac{1}{m})OPT$. □

Theorem 1 GREEDY *runs in $O(n \log n)$ time and has a* tight *approximation ratio of $2 - \frac{1}{m}$.*

Proof: Lemma 2 provides an upper bound for the algorithm. To see that the approximation bound is tight, consider the following instance with $k = m$. The set of jobs consists of one job of size m, and $m^2 - m$ jobs of size 1. In the initial configuration, each processor has $m - 1$ jobs of size 1, and the first processor also contains the additional job of size m. GREEDY at the end of Step 2 will reproduce the original configuration with makespan value $2m - 1$. On the other hand, the relocation of the $m - 1$ jobs of size 1 from the first processor to each of the remaining $m - 1$ processors will give a configuration with makespan value m.

For the running time, note that in Step 1, sorting takes $O(n \log n)$ time. In Step 2, we need $O(k \log m)$ to re-insert the removed jobs. Since the interesting instances have $k, m \leq n$, we obtain a running time of $O(n \log n)$. □

3 The 1.5-Approximation

We now present a more sophisticated algorithm which improves the approximation ratio within the same $O(n \log n)$ running time bound. We first consider the unit-cost version of the problem.

Definition 1 *Jobs of size* strictly *greater than $\frac{1}{2}OPT$ are referred to as* large *jobs, and the rest are referred to as* small *jobs. Let L_T denote the* total number *of large jobs. Let m_L be the number of processors with at least one large job; then, $L_E = L_T - m_L$ denotes the number of* extra *large jobs on this set of processors. At any instant, we say a processor is* large-free *if it doesn't contain any large jobs at that instant.*

Clearly, we need to know the value of OPT in order to classify the jobs according to their sizes. The following algorithm assumes that the value of OPT is known. Later, in Subsection 3.1, we will show how to run the algorithm without the knowledge of OPT.

Algorithm PARTITION:

(1) From each of the m_L processors which have at least one large job, remove all large jobs, except for the smallest-sized large job therein.
(2) For each processor j, calculate the following values with respect to their current configuration:
   - a_j: the minimum number of small jobs to be removed so that the total size of the remaining small jobs is at most $\frac{1}{2}OPT$.
   - b_j: the minimum number of jobs (including any large job) to be removed so that the total size of the remaining jobs (including any large job) is at most OPT.
   - $c_j = a_j - b_j$
(3) Select the L_T processors with the smallest values of c_j, breaking ties by giving preference to the processors containing large jobs. Remove the a_j *small* jobs from the selected processors, thereby ensuring that the total size of the remaining small jobs on these processors is at most $\frac{1}{2}OPT$.
(4) From the remaining $m - L_T$ processors, remove the b_j jobs from them. Large jobs removed from these processors need to be reassigned. Assign each of them arbitrarily to distinct *large-free* processors created in Step 3.
(5) Arbitrarily assign the large jobs removed in Step 1 to the remaining large-free processors.
(6) For the small jobs removed in Steps 3 and 4, assign them one-by-one to the current minimum-loaded processor.

Henceforth, Optimal will denote an algorithm which solves the problem optimally. We approach the analysis through the notion of a "configuration." A configuration is a (possibly partial) assignment of jobs to processors, where we ignore the exact sizes of large jobs. Specifically, the *initial* configuration denotes precisely the initial assignment of jobs to processors and is the same for both Optimal and Partition. A *final* configuration is the assignment produced by an algorithm and could be different for the two algorithms. While going from the initial to a final configuration, we might encounter intermediate configurations in which some of the jobs are not assigned to any processor.

Since the relocation of a job can be viewed as a removal followed by a reassignment, we can count the cost of moves made by an algorithm by counting the cost of removing jobs, making the reassignment of jobs a free operation.

Fact 1 *In Optimal's final configuration, each processor has at most one large job.*

Observe that for any algorithm, we may freely reorder the sequence of job removals and reassignments (while ensuring that a job's removal precedes its reassignment) without affecting the final result. By fact 1, Optimal must relocate at least L_E large jobs. We assume, without loss of generality, that Op-

TIMAL first removes L_E large jobs from processors containing multiple large jobs. (This is similar to the first step of PARTITION, although the two algorithms may remove different large jobs.) At this point, OPTIMAL is in a configuration similar to PARTITION. We formalize this class of configurations as follows.

Definition 2 *An* extra-free *configuration is any configuration obtained from the initial configuration by removing all but one large job from processors containing multiple large jobs. The size difference of the large job remaining on a processors is ignored.*

We claim, without loss of generality, that both OPTIMAL and PARTITION must go from an *extra-free* configuration to a special class of configurations defined below.

Definition 3 *A (possibly partial) assignment of jobs to processors is called* half-optimal *if it satisfies the following four conditions:*

- *There are exactly L_E large jobs which are* not *assigned to any processor.*
- *An arbitrary subset of the small jobs are* not *assigned to any processor.*
- *There are exactly $L_T - L_E$ processors with exactly one large job; on these processors, the small jobs must have total size not exceeding $\frac{1}{2}OPT$.*
- *The remaining processors do not have any large jobs. Of these, at least L_E processors have total load at most $\frac{1}{2}OPT$, and the remaining $m - L_T$ processors have total load at most OPT.*

The basic idea behind the rest of the analysis is to establish that both PARTITION and OPTIMAL go from an extra-free configuration to a half-optimal configuration. We then show that PARTITION uses the minimum number of moves to perform this configuration change, and therefore does not use any more moves than OPTIMAL. After reaching a half-optimal configuration, PARTITION only reassigns jobs that already have been removed, thus incurring no further cost. We then focus on the approximation ratio achieved in going from the half-optimal configuration to the final configuration.

It can be verified that PARTITION enters a half-optimal configuration at the end of Step 4. As for OPTIMAL, we view its sequence of job relocations as follows. First it relocates L_E large jobs to enter an extra-free configuration. Then, OPTIMAL ignores the L_E large jobs removed and continues to relocate the rest of the jobs, until the remaining jobs have been moved to their final locations. We can see that, at this point, OPTIMAL must be in a half-optimal configuration.

Lemma 3 *The number of removals needed to go from an extra-free configuration to a half-optimal configuration is at least as many as the number of removals performed by* PARTITION *in Steps 2 through 4 in going from its own* *extra-free configuration to its own half-optimal configuration.*

Proof: Consider any sequence of moves $\mathcal{S}$ which starts with an extra-free configuration and reaches a half-optimal configuration. We will reorder the moves of $\mathcal{S}$, and divide them into three phases:

Phase 1: In the target half-optimal configuration, there must be L_T processors such that the total size of the small jobs on each of them is no more than $\frac{1}{2}OPT$; further, exactly $L_T - L_E$ of these processors will also contain a large job (the outstanding L_E large jobs will be placed into the large-free processors at a later stage). We assume, without loss of generality, that $\mathcal{S}$ first completes the removal of small jobs from these L_T processors. We know that the total number of removals from each of these processors is at least the a_j value for that processor.

Phase 2: In the second phase, $\mathcal{S}$ removes jobs (small and large) from the remaining $m - L_T$ processors to ensure that each of them has load no more than OPT and that none of them contain a large job. For each of the $m-L_T$ processors, $\mathcal{S}$ must remove at least b_j jobs.

Phase 3: This phase will contain all the remaining moves of $\mathcal{S}$, including any moves it makes to shuffle the large jobs around among the L_T processors.

Thus, for $\mathcal{S}$ to achieve a half-optimal configuration, the total number of removals is at least the sum of some a_j values for L_T processors, combined with the sum of b_j values for the remaining $m - L_T$ processors. This total is the same as the sum of c_j values for L_T processors, combined with the b_j values for all m processors. PARTITION chooses the L_T processors with the minimum c_j values, implying the desired result. □

Lemma 4 *The total number of removals performed by* PARTITION *is no more than that of* OPTIMAL.

Proof: We will reorder the sequence of job removals for both algorithms into three phases:

(1) removal of L_E extra large jobs to achieve an extra-free configuration;
(2) removal of small jobs and relocation of remaining large jobs to achieve a half-optimal configuration;
(3) reassignment of the L_E extra large jobs from Phase 1 and the small jobs from Phase 2.

In Phase 1, both algorithms remove the same number of jobs. By Lemma 3, the number of removals by PARTITION in Phase 2 cannot exceed that of OPTIMAL. Phase 3 costs nothing since it does not involve any job removals. □

Theorem 2 *Algorithm* PARTITION *achieves an approximation ratio of $\frac{3}{2}$, and this bound is tight.*

Proof: First, we claim that after Step 5, the maximum processor load for PARTITION is no more than $\frac{3}{2}OPT$. To see this, observe that at the end of Step 4, PARTITION is in an half-optimal configuration. In this configuration, there are $m - L_T$ processors with load at most OPT. Of the rest, there are $L_T - L_E$ processors with a single large job and total size of small jobs not exceeding $\frac{1}{2}OPT$, and therefore their total load is at most $\frac{3}{2}OPT$. The remaining L_E processors have no large jobs and total load at most $\frac{1}{2}OPT$. The latter processors are assigned a single large job each in Step 5, and therefore end up with load not exceeding $\frac{3}{2}OPT$.

Now consider Step 6, where the remaining small jobs are reassigned greedily. Consider the maximum-loaded processor j at the end of Step 6. If j did not receive any new jobs during Step 6, then by the preceding argument, the load on this processor is at most $\frac{3}{2}OPT$. Otherwise, let s denote the size of the last job added to processor j. Let the final load of processor j be l. Note that prior to the addition of the job of size s, processor j was the processor with minimum load (otherwise the job would not have been assigned to it). Thus, the total load on the processors is at least $m(l-s)+s$. Since the average load is a lower bound on OPT, we get $m \times OPT \geq m(l-s)+s$. Since in Step 6, we are only reassigning small jobs, $s \leq \frac{1}{2}OPT$. The two inequalities imply $l \leq (\frac{3}{2} - \frac{2}{m})OPT$.

To verify the tightness of this bound, consider the following instance with two processors. The first processor contains two jobs of size $\frac{1}{2}$ and 1, and the second processor contains only a job of size $\frac{1}{2}$. We are allowed $k = 1$ moves, so $OPT = 1$. Given this value of OPT, we will obtain $L_T = 1$, $L_E = 0$, $a_1 = 0$, $a_2 = 0$, $b_1 = 1$, and $b_2 = 0$. It is easy to see that PARTITION will not make any moves whatsoever and will have performance ratio exactly $\frac{3}{2}$. □

3.1 Running PARTITION *Without the Optimal Value*

We now show that it is possible to implement PARTITION without knowing the value of OPT. We produce a small set of discrete threshold values, such that the execution of PARTITION is affected only when the value of OPT changes across these threshold values. And then, we will use the fact that OPTIMAL makes only k moves to infer where the true value of OPT lies among those threshold values.

First, PARTITION needs to know L_T, the number of large jobs. The key observation is that as the value of OPT varies, only when $\frac{1}{2}OPT$ crosses some job's processing time p_i, does the value of L_T change. This gives us n threshold values for OPT, i.e., $2p_i$ for $1 \leq i \leq n$.

We can extend this idea to incorporate the changes in the values of a_j and b_j.

We index the jobs on each processor in increasing order of their sizes. Let n_j denote the number of jobs on processor j and n_j^s denote the number of small jobs on processor i. Let p_j^k denote the processing time of the k^{th} job assigned to processor j. Then, for $1 \leq l \leq n_j$, the sums $B_j^l = \sum_{k=1}^{l} p_j^k$ are all the threshold values for OPT where the b_j value can change. Specifically, $b_j = n_j - l$ for $OPT \in [B_j^l, B_j^{l+1})$, with $B_j^0 = 0$ and $B_j^{n_i+1} = \infty$. Similarly, for $1 \leq l \leq n_j^s$, the sums $A_j^l = 2\sum_{k=1}^{l} p_j^k$ are all the threshold values for OPT where the a_j values can change. In fact, $a_j = n_j^s - l$ for $OPT \in [A_j^l, A_j^{l+1})$, with $A_0 = 0$ and $A_{n_j^s+1} = \infty$. The values A_j^l and B_j^l over all processors j, plus $2p_i$ over all jobs i, constitute a set $\mathcal{T}$ of discrete threshold values for OPT such that knowing OPT relative to these thresholds is sufficient to implement Partition. Note that $|\mathcal{T}| = O(n)$.

Lemma 5 *Enumerate in increasing order the threshold values in the set $\mathcal{T}$. Then, L_T, a_j, and b_j remain unchanged for OPT varying between any two consecutive threshold values. Hence the execution path of* Partition *remains unchanged as well.*

We can guess the value of OPT incrementally, by trying the threshold values in increasing order. The modified Partition algorithm is then as follows:

Algorithm M-Partition**:**

(1) Use the average load as the starting guess for OPT.
(2) Calculate the corresponding L_T, L_E, a_j, b_j, and c_j values using Partition. Let $\widehat{k}$ be the total number of moves needed by this algorithm.
(3) **while** $\widehat{k} > k$ **do**
  - Increase the guessed value of OPT to be the next higher threshold value.
  - Recalculate the L_T, L_E, a_j, b_j, c_j, and $\widehat{k}$ values using Partition.

(4) Return the result produced by the last execution of Partition.

Observe that the lower the threshold value, the higher the number of moves needed by Partition. By the previous lemma, when M-Partition terminates, the final threshold value is the smallest possible value of OPT that could have resulted in k moves or less for Partition. We know that using the exact value of OPT results in k moves or less for Partition. Hence, the exact value of OPT is at least as much as the value of the final threshold when M-Partition terminates.

Theorem 3 *Algorithm* M-Partition *gives an approximation ratio of* 1.5 *in time* $O(n \log n)$.

Proof: Since the final threshold used is no more than OPT, by Theorem 2, the approximation ratio achieved by M-PARTITION is 1.5. Now we shall calculate the running time. For the first run of PARTITION in Step 3 of M-PARTITION, we have to sort the jobs by their sizes and also sort the c_j values, which requires time $O(n \log n)$. For each of the subsequent runs corresponding to the $O(n)$ possible distinct threshold values, we need only constant time per run to take care of the incremental changes to the values of L_T, a_j, and b_j. Also, we note that for each run, at most one c_j value changes by one. Since the values of c_j are integral, we need only constant time to update the sorted list of c_j and calculate $\widehat{k}$. □

3.2 Extension to Arbitrary Cost Functions

In this section, we show how to modify the algorithm PARTITION to work for the more general arbitrary cost case. This algorithm assumes that we know the exact value of the makespan in the optimal solution; we can guess this value via binary search while incurring a small error. In particular, we could spend $\lceil \log \log_{1+\alpha} m \rceil$ time for a guess to be between OPT and $(1+\alpha)OPT$.[4]

For a particular guessed value A, we need to estimate the minimum cost C_A required to achieve a makespan of A. We modify PARTITION to find a solution with makespan at most $1.5A$ incurring a cost C_A^P, which is at most C_A. If $C_A^P \geq B$, then it is an indication that the guess A is too low, and vice versa. So we revise our guess accordingly and run modified PARTITION with the new guess.

The difficulty for the arbitrary cost function lies in determining which large jobs to remove in step 1 and also calculating the a_j, b_j, and c_j values in step 2. For unit cost, the greedy strategy works and we can separate the two steps, which is no longer the case here. We merge steps 1 and 2, replacing a_j, b_j, and c_j with the following definition:

- a_j: the minimum cost to remove all but one large job (if any) and a set of small jobs so that the total size of the remaining small jobs is at most $\frac{1}{2}A$. This is equivalent to the following knapsack problem: find the set of small jobs to remain on the processor such that the total size is no more than $\frac{1}{2}A$, and the total relocation cost of these jobs is as high as possible. As for the large jobs, simply remove all but the most costly one.
- b_j: the minimum cost to remove jobs so that the total size of the remaining jobs (including any large job) is at most A. Similarly, one can use a

[4] Clearly, the maximum load $\mathcal{M}$ in the original configuration is an upper bound on OPT. The average load is a lower bound on OPT, and is at least $\frac{\mathcal{M}}{m}$. This implies $\frac{\mathcal{M}}{m} \leq OPT \leq \mathcal{M}$.

straightforward knapsack routine for this calculation.

- $c_j = a_j - b_j$.

If the maximum relocation cost or the job sizes are polynomially bounded, then we can solve the knapsack problems exactly. Otherwise, one can use a PTAS in the place of the exact knapsack routine to find the set of jobs with total size at most $(1+\epsilon)\frac{1}{2}A$ (or $(1+\epsilon)A$) and maximum relocation cost, the rest of the jobs will have a total relocation cost of $\overline{a_j}$ (or $\overline{b_j}$). Thus, instead of calculating the real a_j and b_j value, we substitute the lower bounds $\overline{a_j}$ and $\overline{b_j}$ for them.

Since the techniques used in the PTAS for the knapsack problem are useful in describing the PTAS discussed in the next section, we show how to calculate $\overline{b_j}$ to illustrate the PTAS (the $\overline{a_j}$'s can be calculated in a similar manner). Recall that we know the optimal assignment $\mathcal{S}_A$ incurs a cost b_j while achieves a final load no more than A on processor j. And the goal is to find a final assignment with load no more than $(1+\epsilon)A$, and cost no more than b_j.

We redefine the notion of large and small jobs as follows – note that these new definitions apply only to the discussion on calculating $\overline{a_j}$, $\overline{b_j}$, and c_j, and *not* the algorithm PARTITION itself. Let δ be a constant, which will be specified later.

Definition 4 *Large jobs are defined to be jobs of size* strictly *greater than* $\delta \times A$. *The remaining jobs are said to be small.*

We discretize the large jobs by *rounding up* their sizes to the nearest values in the sequence $l_i = \delta(1+\delta)^i A$. This restricts the large jobs' sizes to consist of only $s = \lceil \frac{1}{\delta} \log \frac{1}{\delta} \rceil$ distinct values. Let $\overline{A}$ denote the final load after discretizing the large jobs in the optimal assignment $\mathcal{S}_A$.

Lemma 6 $A \leq \overline{A} \leq (1+\delta)A$.

We describe the configuration of a processor by an $(s+1)$-tuple:

$$(x_1, x_2, \ldots, x_s, V_S)$$

where x_i denotes the number of large jobs of (discretized) size $l_i = \delta(1+\delta)^i A$ and V_S is the total size of the small jobs. We will discretize V_S too, by *rounding up* to the nearest integral multiple of $\delta \times A$. Thus, V_S can take on $O(\frac{1}{\delta})$ distinct values. From now on the word *configuration* refers exclusively to the discretized assignment of large and small jobs.

Definition 5 *The configuration of a processor* $(x_1, \ldots, x_s, V_S)$ *is called* W-feasible *if* $V_T = V_S + \sum_{i=1}^{s} x_i l_i \leq W$. *Similarly, an assignment of jobs to a processor is* W-feasible *if the total load is no more than* W. *An assignment of jobs to processors is* W-feasible *if each processor is* W-feasible.

Lemma 7 *There are only a constant number, $O(\frac{1}{\delta^{s+1}}(log\frac{1}{\delta})^s)$, of W-feasible configurations for a single processor, when W is of the order of A.*

Lemma 8 *Consider the optimal assignment $\mathcal{S}_A$. The corresponding discretized configuration is $(\overline{A} + \delta A)$-feasible.*

Note that $\overline{A}+\delta A \leq A+2\delta A$ by Lemma 6. The above lemma implies that if the optimal assignment is A-feasible with cost b_j, then there exists a $(A + 2\delta A)$-feasible configuration with cost no more than b_j. We enumerate through all $(A+2\delta A)$-feasible configurations, and calculate the minimum cost to transform to each configuration. By Lemma 7, there are only a constant number of such configurations. For a given $(s + 1)$-tuple, $(x_1, x_2, \ldots, x_s, V_S)$, corresponding to a configuration, we calculate the minimum cost as follows:

- For large jobs, we simply remove the least expensive jobs in each fixed size class so there are exactly x_i remaining.
- For small jobs, we allow further approximation to the total load. We greedily remove small jobs with the least cost-to-size ratio until the total size of remaining small jobs is between V_S and $V_S + \delta A$. This is possible because the size of a small job is no more than δA.

Out of all these possible configurations, we pick the configuration C with the minimum cost of transformation T_C. Clearly, C is $(A + 3\delta A)$-feasible, and T_C is the minimum relocation cost of transforming to any $(A + 2\delta A)$-feasible configuration. As noted earlier, the configuration corresponding to the optimal assignment is $(A + 2\delta A)$-feasible. Thus, $T_C \leq b_j$. By setting $\delta = \epsilon/3$, we have found a $(A + \epsilon A)$-feasible configuration C with cost $T_C \leq b_j$. We set $\overline{b_j} = T_C$. This completes the PTAS for the knapsack problem used to calculate $\overline{b_j}$.

Having calculated a_j and b_j (or their lower bounds), the calculation of c_j and the rest of the algorithm proceeds along the same lines as Steps 3–6 in PARTITION. Via arguments similar to Lemma 3, we have the following lemma.

Lemma 9 *The total cost of removals needed to go from the initial configuration to a half-optimal configuration is at least as much as the cost incurred by* PARTITION.

Proof: Consider any sequence of moves $\mathcal{S}$ which starts with the initial configuration and reaches a half-optimal configuration. We will reorder the moves of $\mathcal{S}$ and divide them into three phases:

Phase 1: In the resulting half-optimal configuration, there must be L_T processors such that the total size of the small jobs on each of them is no more than $\frac{1}{2}A$; further, each of these processors contains at most one large job. We assume, without loss of generality, that $\mathcal{S}$ first completes the removal of small and large jobs from these L_T processors. We know that the total cost of removals from each of these processors is at least the a_j value for that processor.

Phase 2: In the second phase, the moves of $\mathcal{S}$ are those which remove jobs (small and large) from the remaining $m - L_T$ processors so as to ensure that each of them has load no more than A. For each of the $m - L_T$ processors, $\mathcal{S}$ must incur a cost of at least b_j to remove jobs.

Phase 3: This phase will contain all the remaining moves of $\mathcal{S}$, including any moves it makes to shuffle the large jobs around among the L_T processors.

Thus, for $\mathcal{S}$ to achieve a half-optimal configuration, the total cost of removals is at least the sum of some a_j values for L_T processors, combined with the sum of b_j values for the remaining $m - L_T$ processors. When we substitute $\overline{a_j} \leq a_j$ and $\overline{b_j} \leq b_j$, we can only lower this sum. The sum can also be expressed as the sum of c_j values for L_T processors, combined with the b_j (or $\overline{b_j}$) values for all m processors. Partition chooses the L_T processors with the minimum c_j values, thus implying that it incurs a cost no more $\mathcal{S}$. □

Overall, one achieves an approximation ratio of $(1.5 + \epsilon + \alpha)$, where $\epsilon > 0$ and $\alpha > 0$ are constants. Due to the high running time required for solving the knapsack problem, we can no longer guarantee an overall low running time.

4 Approximation Scheme for Arbitrary Cost Functions

We now present a PTAS for the load rebalancing problem. The PTAS applies to the general version where the cost of relocation of a job i is c_i and the constraint is to keep the total relocation cost below a specified budget B.

We first give an overview of the PTAS. As in the case of any PTAS for packing or scheduling problems, we will need to employ the standard techniques of discretization and dynamic programming; however, we need to be careful in handling the small jobs which are usually not an issue. In some other problems such as the multiple knapsack problem [1], small (high-profit) items need to be handled separately, but the choice of a bin for a small item does not have a dramatic effect on the final profit. In our case, since small jobs may have a huge relocation cost, they cannot be arbitrarily relocated to any processor. At a high level, our idea is to bundle the small jobs together and only consider their total size on any single processor, while managing the roundoff errors.

We again guess the optimal value. For each guessed value A, let C_A denote the minimum relocation cost to achieve a makespan of value A. We attempt to find an $(A + 4\delta A)$-feasible assignment such that the associated relocation cost D_A is no more than C_A. If $D_A > B$, then $C_A \geq D_A > B$ and the guessed value A is too low, and vice versa. As in Subsection 3.2, we could spend $\lceil \log \log_{1+\alpha} m \rceil$

time refining the guessed value A to be between OPT and $(1+\alpha)OPT$. We define large and small jobs as in Subsection 3.2.

Definition 6 *Large jobs are defined to be jobs of size* strictly *greater than* $\delta \times A$. *The remaining jobs are said to be small.*

We discretize the sizes of large jobs, and also the total size of small jobs on a processor, as in Subsection 3.2. Here we use the phase *total discretized size of small jobs* to refer to the sum of discretized size of small jobs on each processor (which is calculated by adding up the value of V_S in each processor's configuration tuple). Note that this sum is different from the sum of the actual size of all small jobs, which we refer to as *total size of small jobs*.

Here $\mathcal{S}_A$ refers to an optimal assignment of jobs, such that each processor is A-feasible. Let C_A be the minimum relocation cost needed to achieve this configuration. According to Lemma 8, we know the following.

Lemma 10 *After discretizing* $\mathcal{S}_A$*, each processor's configuration is* $(A+2\delta A)$*-feasible.*

We will use dynamic programming to construct such a solution. We assume an ordering on the processors. Consider the following $(s+2)$-tuple:

$$(n_1, n_2, \ldots, n_s, M, V).$$

Here M denotes the number of processors being considered, n_i denotes the number of large jobs of size l_i, and V denotes the total size of small jobs. The function $\mathcal{F}(n_1, n_2, \ldots, n_s, M, V)$ denotes the minimum removal cost needed to change the first M processors to the following configuration:

(1) Each of the first M processors is $(A+3\delta A)$-feasible.
(2) The first M processors contain a total of n_i large jobs of (discretized) size $l_i = \delta(1+\delta)^i A$.
(3) The total discretized size of small jobs on the first M processors is V.

Let $\mathcal{C}(n_1, n_2, \ldots, n_s, M, V)$ denote the configuration (of the first M processors) that achieves this minimum.

Ideally, we would like to compute an $(A+2\delta A)$-feasible configuration that requires the minimum relocation cost. The reason we consider $(A+3\delta A)$-feasible configurations instead of $(A+2\delta A)$-feasible ones is as follows. Let N_i denote the total number of large jobs of size l_i on all processors. Also, let V_A be the total discretized size of small jobs, when we discretize the optimal final assignment $\mathcal{S}_A$ (Note that V_A is an integral multiples of δA). We can calculate N_i from the input, however, we cannot calculate the precise value of V_A unless we know the optimal assignment. Instead, we have to use an upper bound on V_A. Let V_R be the total size of small jobs, rounded up to the next higher integral multiple of δA. Clearly, $V_R \leq V_A$. Observe that after discretizing $\mathcal{S}_A$, the total discretized size of small jobs on a processor is no more than δA plus the actual size of small jobs on the processor. With m processors, we get that $V_R \leq V_A \leq V_R + \delta m A$. Now, we can use $V_R + \delta m A$ in the computation of $\mathcal{F}$. However, we cannot guarantee the existence of an $(A + 2\delta A)$-feasible configuration which has a total discretized size of small jobs equal to $V_R + m\delta A$. If we discretize the optimal assignment S_A, we get $(A+2\delta A)$-feasible processors with a total discretized load of V_A, which is some integral multiples of δA away from $V_R + \delta m A$. In order to increase this total discretized load up to the upper bound $V_R + \delta m A$, we might have to add some δA small jobs on some of the processors, thus increasing the load on a processor to at most $A + 3\delta A$. Since the optimal assignment S_A modified in this manner is $(A + 3\delta A)$-feasible, we get the following lemma.

Lemma 11 $\mathcal{F}(N_1, N_2, \ldots, N_s, m, V_R + \delta m A) \leq C_A$.

We now show how to compute the value $\mathcal{F}(N_1, N_2, \ldots, N_s, m, V_R + \delta m A)$. We create a table storing the values $\mathcal{F}(n_1, n_2, \ldots, n_s, M, V)$, for all possible n_i's, M, and V. We know n_i takes on integers 0 through n, M takes on integers 0 through m, and V takes on multiples of δA from 0 to $V_R + \delta m A$. Thus, the table size is bounded by $O(mn^{s+1})$, which is polynomial in the input size. The base case for the dynamic programming is $\mathcal{F}(n_1 = 0, n_2 = 0, \ldots, n_s = 0, M = 0, V = 0) = 0$. During the computation, n_i and V stay non-negative. To calculate $\mathcal{F}(n_1, n_2, \ldots, n_s, M, V)$, we look at the M^{th} processor (the last among the M processors in consideration). Let $C = (x_1, x_2, \ldots, x_s, v)$ denote the current configuration of this processor. Try all possible $(A+3\delta A)$-feasible configurations $C' = (x'_1, x'_2, \ldots, x'_s, v')$. For each feasible configuration, calculate the minimum removal cost $COST(C, C')$ needed to transform from configuration C to C' as follows (we only count the total cost of jobs to be removed from that processor to achieve the configuration):

(1) Initialize $COST(C, C') = 0$;
(2) **For** $i = 1$ to s **do**
  if $x_i > x'_i$ **then**
    Remove the $x_i - x'_i$ jobs of size l_i with minimum cost.
    Increment $COST(C, C')$ with the sum of those costs.
(3) **If** $v > v'$ **then**
  Greedily remove small jobs in increasing order of cost-to-size ratio until the total size of small jobs on the processor does not exceed $v' + \delta OPT$.
  Increment $COST(C, C')$ with the costs of removed jobs.

Let $n_i^{C'} = n_i - x'_i$ and $V^{C'} = V - v'$. Then

$$\mathcal{F}(n_1, n_2, \ldots, n_s, M, V) = \min_{\text{feasible } C'} COST(C, C') + \mathcal{F}(n_1^{C'}, \ldots, n_s^{C'}, M-1, V^{C'}).$$

The configuration C' that achieves the minimum $COST(C, C')$ appended to $\mathcal{C}(n_1^{C'}, \ldots, n_s^{C'}, M-1, V^{C'})$ is stored as $\mathcal{C}(n_1, n_2, \ldots, n_s, M, V)$.

It is clear that the cost incurred in removing large jobs is the minimum possible. Now consider the small jobs. Since their sizes are at most δA, as we perform greedy removals, there will come a point when the total size of small jobs falls in $(v', v' + \delta OPT]$. The total cost incurred till this point is no more than the cost needed to achieve a configuration in which the total size of small jobs is at most v'. Thus, the total removal cost incurred by the algorithm is no more than C_A. However, some of the processors will only be $(A+4\delta A)$-feasible.

Now let us focus on the reassignment of removed jobs, according to the processor configurations computed in $\mathcal{C}(N_1, N_2, \ldots, N_s, m, V_R+\delta mA)$, henceforth called the *target configuration*. A large job of type i can be assigned freely to any processor which has available space for jobs of type i according to the target configuration $\mathcal{C}(N_1, N_2, \ldots, N_s, m, V_R+\delta mA)$. By definition, all processors will be assigned precisely the same number of large jobs of type i as specified by the target configuration. Next, we reassign the remaining small jobs. Let v_j be the total discretized size of small jobs to be assigned to processor j according the the target configuration. We consider the small jobs one-by-one in an arbitrary order and assign each of them to one of the processors j whose current total size of small jobs is below the bound v_j. This is because the total discretized size of small jobs in the target configuration is $V_R + \delta mA > V_R$, an upper bound on the total size of small jobs on all the processors. Thus, as long as we have unassigned small jobs, there exists a processor j whose actual size of small jobs is below its target bound v_j. Recall that the bound v_j together with the large jobs on processor j results in an $(A + 3\delta A)$-feasible configuration. Since we are adding at most one extra small job to processor j, and the size of any small job is at most δA, the final load on processor j does not exceed $A + 4\delta A$. So overall, the final load on each processor is no more than $(1+4\delta)A = (1+\alpha)(1+4\delta)OPT$. Setting $\delta = \epsilon/5$ and $\alpha = 1-(1+\epsilon)/(1+0.8\epsilon)$, we obtain:

Theorem 4 *Our algorithm rebalances the jobs among processors with cost at most B, such that the maximum processor load is at most $(1+\epsilon)OPT$.*

5 Limits to Approximations

Consider the *move minimization* problem: Given a bound L on maximum processor load, minimize the number of moves required to achieve this load (if the bound L is not achievable, the algorithm should report ∞). The following theorem is a result of a reduction from the well-known Partition problem.

Theorem 5 *There is no approximation algorithm with polynomial running time for the move minimization problem, unless $P = NP$.*

One question left open is whether the relocation cost is hard to approximate even when the target load L is known to be strictly above the minimum load achievable.

We now present results which show that certain variants of the problem do not admit polynomial-time approximation algorithms unless P=NP. We study the problem under the broader setting of the generalized assignment problem, where one seeks to minimize the makespan, incurring cost no more than a given budget. We restrict our attention to instances where the processing time of a job is independent of the processor on which it is processed, while the cost of a job is processor-dependent.[5] Note that if the cost function for a particular job i is independent of the processor, i.e., $c_{ij} = c_i$ for all processors j, then the problem is equivalent to multiprocessor scheduling, for which an FPTAS exists. The load rebalancing problem is a special case where the cost of a job i is a constant $p = 0$ on a particular processor j, and a constant $q = c_i$ on all other processors. Our analysis shows that such a problem admits a PTAS.[6] In a more general setting, when each job can have only one of the two distinct costs for any processor, the problem is MAX-SNP-hard.

Theorem 6 *The makespan minimization problem with costs $c_{ij} \in \{p, q\}$ $(p < q)$ does not have a polynomial ρ-approximation algorithm, for any $\rho < \frac{3}{2}$, unless P=NP.*

Proof: The proof is similar to the $\frac{3}{2}$ hardness proof in [10]. We reduce the 3-dimensional matching problem to our problem.

Instance: Disjoint sets $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$, and a family $F = \{T_1, \ldots, T_m\}$ of triples with $|T_j \cap A| = |T_j \cap B| = |T_j \cap C| = 1$ for $j = 1, \ldots, m$.
Question: Does F contain a matching, i.e., a subfamily F' for which $|F'| = n$ and $\cup_{T_j \in F'} T_j = A \cup B \cup C$?

Given an instance of this problem, we create an instance of the two-cost version of the generalized assignment problem as follows. We create m processors, with processor j corresponding to the triple T_j, for $j = 1, \ldots, m$. We call the triples that contain a_i of type i. Let t_i be the number of triples of type i for $i = 1, \ldots, n$. We have $2n$ element jobs of unit size, each job corresponding to an element in $B \cup C$. For each type i, we also create $t_i - 1$ dummy jobs of size 2 each (so there are a total of $m - n$ dummy jobs). Consider a processor

[5] Lenstra et al [10] considered jobs with processor-dependent processing times, but without cost constraints.
[6] Our proof can be extended to arbitrary p and q $(q > p)$.

j corresponds to a triple of type i, say $T_j = (a_i, b_k, c_l)$. The element jobs corresponding to b_k and c_l and the type i dummy jobs incur a cost of p on processor j, while the rest of the jobs incur a cost of q. The maximum cost allowed is $(m+n)\times p$. This implies that any feasible solution must assign each job to a processor with cost p.

Suppose there is a matching, then all jobs can be scheduled with cost exactly $(m+n)\times p$ so that the makespan is 2 for each processor. Conversely, a schedule of makespan 2 within cost $(m+n)\times p$ implies a matching of size n. Since the next possible value for makespan is 3, it is NP-hard to approximate the makespan better than a factor of $\frac{3}{2}$. □

We consider two natural extensions of the load rebalancing problem, and present negative results in both cases. The *Constrained Load Rebalancing* problem has the additional constraint that each job can be reassigned to a specified subset of processors only.

Corollary 1 *The Constrained Load Rebalancing problem does not have a polynomial ρ-approximation algorithm, for any $\rho < \frac{3}{2}$, unless P=NP.*

Proof: We are going to use the same reduction introduced in the previous theorem. We introduce one extra processor P, and one extra job J with size 2. Initially, all jobs reside on processor P. Job J cannot be relocated to any of the remaining m processors. A job i (other than J) can only be relocated to one of the remaining m processors iff i has a processing cost of p on that processor in our previous reduction. Let the relocation budget be at least $(m+n)\times p$. So there is a matching iff all jobs except J can be relocated to the remaining m processors for a makespan of value 2. Otherwise, the smallest makespan value one can hope for is 3. □

The best upper bound known for the Constraint Load Rebalancing problem is the 2-approximation by Shmoys and Tardos [14]. However, whether there exists a $\frac{3}{2}$ approximation algorithm for the Constrained Load Rebalancing problem is an interesting open question.

We define another variant called the *Conflict Scheduling* problem with a different additional constraint that some specified pairs of jobs have conflicts and cannot be assigned to the same processor. We show that the Conflict Scheduling problem is very hard to approximate.

Theorem 7 *There is no approximation algorithm with polynomial running time for the Conflict Scheduling problem, unless P=NP.*

Proof: We perform a reduction from the 3-dimensional matching problem as in Theorem 6. There are m processors, each corresponds to a triple in F. There are m jobs, each corresponds to a triple in F (call them triple jobs). No two triple jobs can be assigned to the same processor, so a feasible solution would have spread these m jobs one per processor. There are also $3n$ jobs, each corresponds to an element in $A \cup B \cup C$ (call them element jobs). Each element job can only be assigned together with the corresponding triple job, i.e., an element job u can be assigned together with a triple job T_i, iff $u \in T_i$. And finally, there are $m-n$ dummy jobs. No two dummy jobs can be assigned together. And none of the element jobs can be assigned with any of the dummy jobs. Disregarding job costs and sizes, if there is a feasible assignment of jobs to processors that satisfy the conflicts, then the following must hold:

- There is one triple job per processor.
- There are $m-n$ processors, each of which has one dummy job.
- All the $3n$ element jobs are assigned to the remaining n processors. Each processor is assigned exactly the three element jobs corresponding to a_i, b_k, c_l, and the triple job on that processor is $T_j = (a_i, b_k, c_l)$.

So a feasible assignment of the jobs implies a matching of size n and vice versa. Since any approximation algorithm will have to return a feasible solution whenever a feasible assignment exists, we get the desired result. □

6 Acknowledgments

We thank S. Keshav of Ensim Corporation for posing the load rebalancing problem.

References

[1] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001, pages 213-222.

[2] P. Efraimidis and P. Spirakis. Randomized Approximation Schemes for Scheduling Unrelated Parallel Machines. *Electronic Colloquium on Computational Complexity (ECCC)*, Technical Report TR00-007, 2000.

[3] A. Fishkin, K. Jansen, and M. Mastrolilli. Grouping techniques for scheduling problems: simpler and faster. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA)*, 2001, pages 206–217.

[4] B. Ghosh, F.T. Leighton, B.M. Maggs, S. Muthukrishnan, C.G. Plaxton, R. Rajaraman, A.W. Richa, R.E. Tarjan, and D. Zuckerman. Tight analyses of two local load balancing algorithms. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, 1995, pages 548–558.

[5] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45 (1966):1563–1581.

[6] M. Harchol-Balter and A.B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(1997):253–285.

[7] Y.F. Hu, R.J. Blake, and D.R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10(1998):467–483.

[8] K. Jansen and L. Porkolab. Improved Approximation Schemes for Scheduling Unrelated Parallel Machines. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC)*, 1999, pages 408–417.

[9] E.D. Lazowska, D.L. Eager, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. *ACM Performance Evaluation Review*, 16(1998):63–72.

[10] J.K. Lenstra, D. Shmoys, and E. Tardos. Approximation Algorithms for Scheduling Unrelated Parallel Machines. *Mathematical Programming*, 46(1990):259–271.

[11] P.B. Linder and A. Shah. Website Migration Load Balancing of Web Servers. *Manuscript*.

[12] M. Nuttall. A brief summary of systems providing process or object migration facilities. *Operating Systems Review*, 28(1994):64–80.

[13] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991, pages 237–245.

[14] D. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1993):461–474.