The Load Rebalancing Problem

Gagan Aggarwal [*]
gagan@cs.stanford.edu

Rajeev Motwani [†]
rajeev@cs.stanford.edu

An Zhu [‡]
anzhu@cs.stanford.edu

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 makespan, the load on the heaviest loaded processor. Since in most real world scenarios the load is a dynamic measure, the initial assignment may be not remain optimal with 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 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, resolving the complexity issue. Finally, we investigate the approximability of several extensions of the rebalancing model.

Categories and Subject Descriptors

F.2.2 [**Analysis Of Algorithms And Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*;
G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*;
C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

[*]Supported by a Stanford Graduate Fellowship.

[†]Supported in part by NSF Grant IIS-0118173, an Okawa Foundation Research Grant, an SNRC grant, and grants from Microsoft and Veritas.

[‡]Supported by a GRPW fellowship from Bell Labs, Lucent Technologies.

General Terms

Algorithms,Theory,Management

Keywords

Approximation Algorithms, Load Balancing, Scheduling

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 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. (Nuttall [12] gives a good survey of work in process migration.) Some have argued [9] that the performance benefits of process migration are limited to unrealistic CPU-bound workloads, while others [6] have done trace-driven simulations to claim that process migration is useful in real-world situations. The key issue is whether the benefit derived from utilization of an idle CPU justifies the cost of migrating a process. Rudolph et al [13] have attempted to devise process migration schemes which migrate only a *few* processes, assuming a unit-size job model. In this paper, we formalize the notion of *few processes,* while doing away with the assumption of unit-sized processes, thus 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 et al [7] provide an elegant diffusive technique for load balancing in this model. Ghosh et al [4] prove 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). In their model too, 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 machine dependent costs and processing times. For arbitrary number of machines, the generalized assignment problem is MAX-SNP-hard; further, it is known that this problem cannot be approximated better than 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 machines 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 et al [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 (Linder and Shah [11] have implemented our algorithms for load balancing of webservers via website migration and have achieved promising experimental results.) We then 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, 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, say $s_1 \geq s_2 \geq \cdots \geq s_n$. 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, and the constraint is that the total relocation cost be 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: Simply set $c_{ij} = 0$ (c_{ij} denotes the cost of assigning job i to machine j) if job i currently resides on machine j, and $c_{ij} = 1$ otherwise. By the results of Shmoys and Tardos [14], we obtain a 2-approximation algorithm for load rebalancing.

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 maximum-load processor, remove the largest job.
2. Consider the k jobs removed in an arbitrary order. Place each of these job in the current minimum-load 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. We claim that Step 1 is the optimal way of removing k jobs so as to minimize the maximum processor load. To see this, suppose there exists another set of removals R of jobs, after which the maximum load is smaller, say M_R. Without loss of generality, we can assume that R removes from each processor the largest possible jobs, i.e., R never leaves a larger job in the processor while removing a smaller job. In comparing, GREEDY with R, look at any particular processor b such that the greedy algorithm removes more jobs than R. Look at the first time GREEDY is about to remove a job from processor b which R does not. At that point, GREEDY has already removed all jobs removed by R from processor b, so if S_b is the load on processor b at that point, then $S_b \leq OPT$. Also since GREEDY is about to remove another job from processor b, it implies that b is the maximum-loaded processor at that time; hence, $G_1 \leq S_b$. Since $S_b \leq M_R$ and $M_R \leq OPT$, it follows that $G_1 \leq OPT$. □

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

PROOF. Consider the maximum-loaded processor after Step 2, say j, and note that this 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 the final load of processor j is G_2, and 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). We obtain that the total load on the processors is at least $m(G_2 - s) + s$ which is a lower bound on $m \times OPT$. Quite clearly, s itself is a lower bound on OPT, and so we obtain the two inequalities: $m \times (G_2 - p) + p \leq m \times OPT$ and $p \leq OPT$. Combining the inequalities implies 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, assume that $k \geq m - 1$ and consider the following example. The input instance 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 job of size m. If GREEDY considers the job of size m last in Step 2, it will reproduce the original configuration and be of value $2m - 1$. On the other hand, the relocation of only $m - 1$ jobs of size 1 from the first processor will give an assignment of value m.

To calculate 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 reinsert 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 on the approximation ratio within the same $O(n\log n)$ running time bound. We first consider the unit-cost version of the problem. We use OPT to denote the optimal value of the makespan.

DEFINITION 1. *Jobs of size* strictly *greater than $\frac{1}{2}OPT$ are referred to as* large, *and the rest are referred to as* small. *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. We say a processor is* large free *if currently it doesn't contain any large jobs.*

Clearly, the value of OPT needs to be known to enable us to classify jobs as being large or small. The following algorithm assumes that the exact value of OPT is provided. Later, in Section 3.1, we will do away with this assumption.

Algorithm PARTITION:

1. From each of the m_L processors which has a large job, remove all large jobs, except for the smallest-size large job therein.
2. Calculate for each processor i, the following values with respect to their current configuration:
   - a_i: 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_i: 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.
   - $c_i = a_i - b_i$
3. Select the L_T processors with the smallest values of c_i, breaking ties by giving preference to the processors containing large jobs. Remove the a_i *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_i jobs from them. Large jobs, if any, on these processors will be removed and need to be reassigned. Assign each of the removed large jobs (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-load 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.

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 OPTIMAL first removes L_E large jobs from processors containing multiple large jobs. (This is as in PARTITION's Step 1, although the two algorithms may remove different large jobs.) At this point OPTIMAL is in the same configuration as PARTITION after Step 2. We formalize this class of configuration 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. As always, the exact sizes of the large jobs remaining in the processors is ignored.*

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

DEFINITION 3. *A partial assignment of jobs to processors, excluding the L_E large jobs removed in Step 1, is called* half-optimal *if it satisfies all following 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 which have exactly one large job; in 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.*

It can be verified that PARTITION enters an extra-free configuration at the end of Step 1, and then a half-optimal configuration at the end of Step 4. While going from the initial to a final configuration, an algorithm must reassign all jobs that are removed at any point. However, since an intermediate configuration is only a partial assignment of jobs, we will encounter cases where an algorithm removes but does not reassign jobs in making a configuration change. Since the jobs which are reassigned are exactly the same as the jobs which are removed, we can keep track of the cost incurred by an algorithm by simply keeping track of cost of the removals only. Thus, while counting the number of moves made by an algorithm in going from one configuration to another, we consider only the removals of jobs, making reassignment of removed jobs a free operation.

The basic idea behind the rest of the analysis is to establish that both PARTITION and OPTIMAL go from an extra-free to a half-optimal configuration. We then establish that PARTITION uses the minimum possible number of moves to perform this configuration change, and therefore does not use more moves than OPTIMAL. After reaching a half-optimal configuration, PARTITION only reassigns jobs that already have been removed and we focus on the approximation ratio achieved in going from the half-optimal configuration to the final 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 $\mathcal{S}$ in 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_i value for that processor.

Phase 2: In the second phase, $\mathcal{S}$ removes jobs (small and large) from the remaining $m - L_T$ processors so as 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_i 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 amongst 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_i values for L_T processors, combined with the sum of b_i values for the remaining $m - L_T$ processors. This total is the same as the sum of c_i values for L_T processors, combined with the b_i values for all m processors. But PARTITION's cost is exactly the same, and it chooses the L_T processors with the minimum c_i values, which implies the desired result. $\square$

LEMMA 4. *The total number of moves performed by* PARTITION *is no more than the total number of moves performed by* OPTIMAL.

PROOF. We will reorder the sequence of moves 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 moves used by PARTITION in Phase 2 cannot exceed the number of moves for OPTIMAL. Phase 3 costs nothing since it does not involve any job removals. $\square$

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 $\frac{3}{2}OPT$.

Consider now Step 6, where we greedily reassign the left-over small jobs. Consider the maximum-loaded processor at the end, say j. If j did not receive any new jobs during Step 6, then by the preceding comments the load on this processor is at most $\frac{3}{2}OPT$ and we are done. 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 of minimum load (otherwise the job would not have been assigned to it). We obtain that the total load on the processors is at least $m(l-s)+s$ which is a lower bound on $m \times OPT$, giving the inequality $m(l-s)+s \le m \times OPT$. Quite clearly, $s \le \frac{1}{2}OPT$ since in Step 6 we are only dealing with small jobs. Combining the two inequalities implies that $l \le (\frac{3}{2} - \frac{2}{m})OPT$ and again we are done. Overall, we can only guarantee a bound of $\frac{3}{2}$ (and not $\frac{3}{2} - \frac{2}{m}$) because it is possible that processor j already had load $\frac{3}{2}OPT$ at the end of Step 6 and did not receive any new job in Step 7.

To verify the tightness of this bound, consider the following instance: we have 2 processors, the first containing two jobs of size $\frac{1}{2}$ and 1, and the second containing 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 $\frac{3}{2}$. $\square$

3.1 Determining the Optimal Value

We now show it is possible to implement PARTITION without knowing the value of OPT. Observe that PARTITION does not directly take care of the constraint of making at most k moves; instead it assumes that OPTIMAL makes no more than k moves and guarantees that it makes no more moves than OPTIMAL. We will now show that there is a set of discrete threshold values, such that only when the value OPT changes across any of them does it affect the execution of PARTITION. And then, we will use the fact that OPT can make only k moves to infer where the true value of the OPT lies among those threshold values.

First, PARTITION needs to know the value of 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_j, does the value of L_T change. This gives us $O(n)$ threshold values.

We can extend this idea to incorporate the changes in a_i and b_i. More precisely, let n_i denote the number of jobs on processor i indexed in increasing order of size; also, let n_i^s denote the number of small jobs on processor i. Let p_j^i denote the processing time of the j^{th} job assigned to processor i. Then, for $1 \le l \le n_i$, the sums $B_l = \sum_{j=1}^{l} p_j^i$ are all the threshold values for OPT at which the b_i value can change. Specifically, $b_i = n_i - l$ for $OPT \in [B_l, B_{l+1})$, with $B_0 = 0$ and $B_{n_i+1} = \infty$. Similarly, for $1 \le l \le n_i^s$, the sums $A_l = 2\sum_{j=1}^{l} p_j^i$ are all the threshold values for OPT where the a_i values can change; in fact, $a_i = n_i^s - l$ for $OPT \in [A_l, A_{l+1})$, with $A_0 = 0$ and $A_{n_i^s+1} = \infty$. The set of all a_i and b_i threshold values over all processors i, together with $2p_j$ over all jobs j, constitute a set of discrete threshold values for OPT such that knowing OPT relative to these thresholds is sufficient to implement PARTITION.

LEMMA 5. *Enumerate in increasing order all threshold values for each processor i, with respect to L_T, a_i, and b_i. Then, L_T, a_i, b_i remain unchanged for OPT varying between any two consecutive threshold values.*

We can increase the value of OPT gradually, 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_i, b_i, and c_i 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_i, b_i, c_i, and $\widehat{k}$ values using PARTITION.

4. Return the result produced by the last execution of PARTITION.

LEMMA 6. *The maximum threshold value not exceeding OPT gives the same value for L_T, a_is and b_i as the exact value of OPT, and thus* M-PARTITION *proceeds exactly as* PARTITION *till Step 6. The reassignment of small jobs may differ but this does not affect the number of jobs moved. Hence, when* M-PARTITION *terminates, the final threshold value is at most OPT.*

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, using an analysis similar to the proof of Theorem 2, we can show that 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_i 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 to do the incremental changes to the value L_T, a_i, and b_i. Also, we note that for each run, at most one c_i value is changes by 1. Since the values of c_i are integral, we need constant time to update the sorted list of c_i and calculate $\widehat{k}$. $\square$

3.2 Extension to Arbitrary Cost Functions

We show how to modify the algorithm PARTITION to work for the more general arbitrary cost case. This algorithm assumes that we know that 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 $O(\log(\frac{\log m}{\log(1+\alpha)}))$ time for a guess to be between OPT and $(1+\alpha)OPT$.[1]

For a particular guessed value A, we need to estimate the 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_i, b_i, and c_i values in step 2. For unit cost, the greedy algorithms work and we can separate the two steps, which is no longer the case here. We merge steps 1 and 2, replacing them with the following:

- a_i: 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 be remain in 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_i: 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 straightforward knapsack routine for this calculation.
- $c_i = a_i - b_i$

[1]Clearly, the maximum load in the original instance is an upper bound on OPT. The average load is a lower bound on OPT, and is at most $\frac{1}{m}$ of the maximum load. This implies $\frac{\text{m}aximumload}{m} \leq OPT \leq \text{m}aximumload$.

If the maximum relocation cost or the jobs sizes are polynomially bounded, then we can solve the knapsack problems exactly. Otherwise, one can use a PTAS in the place of the knapsack routine to find the set of jobs with total size at most $(1+\varepsilon)\frac{1}{2}A$ (or $(1+\varepsilon)A$) and maximum relocation cost, the rest of the jobs will have a total relocation cost of $\overline{a_i}$ (or $\overline{b_i}$). Thus, instead of calculating the real a_i and b_i value, we substitute the lower bounds $\overline{a_i}$ and $\overline{b_i}$ for them. The technique here is similar to the PTAS we discuss in the next section; however, for the sake of completeness, we give the details here. We will show how to calculate $\overline{b_i}$; the $\overline{a_i}$'s can be calculated in a similar manner.

We redefine the notion of large and small jobs as follows - note that this definition applies only to the discussion on calculating $\overline{a_i}$, $\overline{b_i}$, and c_i, and *not* the algorithm PARTITION itself. Let δ be a constant, which we shall determine 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 size to the nearest value in the sequence $l_i = \delta(1+\delta)^i A$. 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. The minimum cost to achieve a makespan of $(1+\delta)A$ in the discretized instance is no more than the cost to achieve A in the original instance. Further, there are only constantly many distinct configurations with total job size less than $(1+\delta)A$. For each processor, we enumerate through all such possible configurations, and calculate the minimum cost to change to that configuration. For a given configuration, $(s+1)$-tuple, $(x_1, x_2, \ldots, x_s, V_S)$, we calculate the minimum cost as follows:

- For large jobs, we simply remove the most 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 largest cost to size ratio until the total size of the 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 one with the minimum cost of transformation. Thus, we find a configuration with total load $(1+2\delta)A$, such that the cost of transformation from the processor's current configuration to that configuration is no more than the cheapest way to transform to any configuration with total load at most A. Thus, if we let $\delta = \varepsilon/2$, the minimum cost found by this procedure is a lower bound on b_i. Also if we remove the jobs corresponding to this cost, we are left with a processor load of at most $(1+\varepsilon)A$. We can calculate a_i in a similar manner.

Having calculated a_i,b_i, and c_i, the rest of the algorithm proceeds the same as Steps 3–6 in PARTITION. Via arguments similar to Lemma 3, we have the following.

LEMMA 7. *The total cost of removal needed to go from the initial configuration to a half-optimal configuration is at least as many 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 $\mathcal{S}$ in three phases:

Phase 1: In the ending 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 number of removals from each of these processors is at least the a_i 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 and that none of them contain a large job. For each of the $m - L_T$ processors, $\mathcal{S}$ must remove at least b_i 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 amongst the L_T processors. We simply ignore these extra moves and do not include their cost.

Thus, for $\mathcal{S}$ to achieve a half-optimal configuration, the total number of removals is at least the sum of some a_i values for L_T processors, combined with the sum of b_i values for the remaining $m - L_T$ processors. This total is the same as the sum of c_i values for L_T processors, combined with the b_i values for all m processors. But PARTITION's cost is exactly the same, and it chooses the L_T processors with minimum c_i value, which implies the desired result. As for the case where we substitute $\overline{a_i} \le a_i$ and $\overline{b_i} \le b_i$, we argue the optimal solution involving $\overline{a_i}$ and $\overline{b_i}$ is no more than that of a_i and b_i. $\square$

Overall, one achieves an approximation ratio of $(1.5 + \varepsilon + \alpha)$, where $\varepsilon > 0$ and $\alpha > 0$ are constants.

4. APPROXIMATION SCHEME FOR ARBITRARY COST FUNCTIONS

We now present a PTAS for the load rebalancing problem. In fact, the PTAS applies to the more general version of the load rebalancing problem 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 machine. At a high level, our idea is to bundle the small jobs together and only consider their total size on any single machine, while managing the roundoff errors.

As before, we begin by assuming that the exact value of OPT is known. Let $\delta \in (0,1]$ be a parameter to be specified later. We redefine the notion of large and small jobs as follows.

DEFINITION 5. *Large jobs are defined to be jobs of size* strictly *greater than* $\delta \times OPT$. *The remaining jobs are said to be small.*

We use V_T to denote the total size of the jobs on a processor, and V_S to denote the total size of *small* jobs on a processor. We discretize the large jobs by *rounding up* their size to the nearest value in the sequence $l_i = \delta(1+\delta)^i OPT$. This restricts the large jobs' sizes to consist of only $s = \lceil \frac{1}{\delta} \log \frac{1}{\delta} \rceil$ distinct values. Let $\overline{OPT}$ denote the optimal load value for the *discretized* instance.

LEMMA 8. $OPT \le \overline{OPT} \le (1+\delta)OPT$

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 $l_i = \delta(1+\delta)^i OPT$ 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 $\delta \times OPT$. Thus, V_S can take on $O(\frac{1}{\delta})$ distinct values.

DEFINITION 6. *The configuration of a processor* $(x_1, \ldots, x_s, V_S)$ *is called* W-feasible *if* $V_T = V_S + \sum_{i=1}^{s} x_i l_i \le W$.

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

LEMMA 9. *Consider a job assignment which achieves* $\overline{OPT}$ *for the discretized instance. If we round up the total size of small jobs on each processor to an integral multiple of* δOPT, *then the resulting configuration is* $(\overline{OPT} + \delta OPT)$*-feasible.*

We will use dynamic programming to construct a solution for the discretized instance. We assume an ordering on the processors. The goal is to construct a table, each entry of which represents a solution for an instance of the problem and is indexed by an $(s+2)$-tuple $(n_1, n_2, \ldots, n_s, M, V)$. Here M denotes the total number of processors in the instance, n_i denotes the total over all M processors of the number of large jobs of size l_i, and V denotes an upper bound on the total (rounded-up) load of the small jobs on the M processors. Let V_R be the total load of all small jobs rounded up to the next higher multiple of $\delta \times OPT$

While specifying the initial input instance, ideally we want to make V the total rounded up load of small jobs in Lemma 9. But we don't know the exactly value, instead, we use a good upper bound on V. We know in Lemma 9, the total rounded up load of small jobs is no more than the total load of small jobs, plus at most δOPT per processor. Let V_R be the total load of all small jobs rounded up to the next higher multiple of $\delta \times OPT$. Thus, $V = V_R + \delta m OPT$, where m is the total number of processors.

LEMMA 10. *For* $V = V_R + \delta m OPT$ *and* n_i*'s as in the discretized instance, there exists an* $(\overline{OPT} + 2\delta OPT)$*-feasible configuration. (We call this configuration* C_{round}*.)*

PROOF. To verify this lemma, observe that in the rounded-up configuration used in Lemma 9, the sum of the total rounded-up load of each processor may add up to less than V, but is definitely at least V_R. So, we need to add at most δOPT load to each processor to match V. After this, the load on any processor is no more than $\overline{OPT} + 2\delta OPT$. $\square$

The value indexed by an $(s+2)$-tuple $(n_1, n_2, \ldots, n_s, M, V)$ is the minimum relocation cost needed to get the first M processors to an

$\overline{OPT} + 2\delta OPT$-feasible configuration in which they have n_i jobs of size l_i (for $1 \le i \le s$), and the total size of the small jobs is V. Since $0 \le n_i \le n$, $0 \le M \le m$, and V is a multiple of δOPT, the table size is bounded by $O(mn^{s+1})$, which is polynomial in the input size.

For an entry indexed by $(n_1, n_2, \ldots, n_s, M, V)$, we look at the last of the M processors. Let the current configuration of this processor be $C = (x_1, x_2, \ldots, x_s, v)$. Try all possible $(\overline{OPT} + 2\delta OPT)$-feasible configurations $C' = (x'_1, x'_2, \ldots, x'_s, v')$. For each feasible configuration, calculate the minimum relocation cost $COST(C, C')$ needed to get from C to C' configuration 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 $x_i - x'_i$ jobs of size $l_i$ of
      minimum cost, and 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 $V_S \le v' + \delta OPT$, and increment
      $COST(C,C')$ with the sum of those costs.
```

For configuration C', let $n_i^{C'} = n_i - x'_i$ and $V^{C'} = V - v'$. Then, the entry for index $(n_1, n_2, \ldots, n_s, M, V)$ is computed by

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

The base case for the dynamic programming is $(n_1 = 0, n_2 = 0, \ldots, n_s = 0, M = 0, V = 0) = 0$. During the computation, n_i and V stay non-negative.

It is easy to show that the cost incurred in removing large jobs is minimum. As for small jobs, since their sizes are all at most δOPT, as we perform greedy removals, there will come a point when $v' < V_S \le v' + \delta OPT$. The total cost incurred till this point is no more than the cost that would be incurred to achieve a configuration in which the total load of the small jobs is no more than v'. Thus, the total removal cost incurred by the algorithm is no more than an optimal algorithm would use to achieve the configuration C_{round}.

Now lets focus on reassignment of removed jobs. A large job of type i can be assigned freely to any processor which has available space for jobs of type i. Now consider the load on each processor before we reassign any of the small jobs.

LEMMA 11. *If no small jobs were removed from a processor, then its load is no more than its load in the C_{round} configuration, which is $(\overline{OPT} + 2\delta OPT)$. For the other processors from which small jobs were removed, the load exceeds the load in the C_{round} configuration by at most δOPT. Thus, the load on each of those processors is no more than $(\overline{OPT} + 3\delta OPT)$.*

It can be seen that the small jobs may be assigned to any processor i whose total load of small jobs is below the calculated V_S bound, i.e., the processors that are $(\overline{OPT} + 2\delta OPT)$-feasible. Since the small jobs are of size at most δOPT, the final load does not exceed $\overline{OPT} + 3\delta OPT$ on these processors that received more small jobs. So overall, the final load on each processor is no more than $\overline{OPT} + 3\delta OPT \le (1 + 4\delta)OPT$.

It remains to remove the assumption that the value of OPT is known in advance. We can determine the value of OPT to within a factor of $1 + \delta$ via binary search, using the total relocation cost produced by the algorithm for a given guess for OPT to find the smallest value of OPT (within a precision of δ) for which we get relocation cost at most B. The algorithm finds a solution with relocation cost at most B, in time $O(mn^{s+1})$, such that the maximum processor load is at most $(1 + 5\delta)OPT$. Setting $\delta = \varepsilon/5$, we obtain:

THEOREM 4. *Our algorithm rebalances the jobs amongst processors at cost at most B, such that the maximum processor load is at most $(1+\varepsilon)OPT$, and with running time $m(\frac{5n}{\varepsilon})^{\frac{5}{\varepsilon}\log\frac{5}{\varepsilon}}$.*

5. LIMITS TO APPROXIMATION

Consider the *move minimization* problem: Given a bound B on maximum processor load, minimize the number of moves required to achieve this. (If the bound B 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 is strictly above the minimum load achievable.

We now present results which show that certain polynomial approximation algorithms cannot exist unless P=NP. We study the problem under the broader setting of the generalized assignment problem. We restrict our attention to instances where the processing time of a job is independent of the processor on which it is processed.[2] 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 the special case where the cost of a job i is a constant $l_i = 0$ on a particular processor j, and a constant $h_i \ge l_i$ on all other processors. Our analysis shows that such a problem admits a PTAS.[3] In a more general setting, when each job can have only one of two distinct costs for any processor, the problem is MAX-SNP-hard.

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

PROOF. The proof is along the same line as the $\frac{3}{2}$ hardness proof in [10]. We start from the 3-dimensional matching 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 $|T_i \cap A| = |T_i \cap B| = |T_i \cap C| = 1$ for $i = 1, \ldots, m$.

Question: Does F contain a matching, i.e., a subfamily F' for which $|F'| = n$ and $\cup_{T_i \in F'} T_i = A \cup B \cup C$?

We create an instance of m machines. Machines i corresponds to triple T_i, for $i = 1, \ldots, m$. We call the triples that contain a_j of type j. Let t_j be the number of triples of type j for $j = 1, \ldots, n$. We have 2n element jobs of unit size, each job corresponds to an element in $B \cup C$. For each type j we also create $t_j - 1$ dummy jobs of size 2 each (so there are a total of $m - n$ dummy jobs). Machine i corresponding to a triple of type j, say $T_i = (a_j, b_k, c_l)$, incurs a cost of p processing each of the element jobs corresponds to b_k and c_l and the type j dummy jobs. And the rest of the jobs have cost of q on machine i. The maximin cost allowed is $(m + n) \times p$. This implies each feasible solution must assignment each job to a machine 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 machine. Conversely, a schedule of makespan 2 within cost $(m + n) \times$

[2] Lenstra et al [10] considered jobs with machine dependent processing times, but without cost constraints.

[3] Our proof can be extended to arbitrary l_i and $h_i \ge l_i$.

p implies a matching of size n. Since the next possible value for makespan is 3, this implies a $\frac{3}{2}$ hardness bound. $\square$

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 machines only.

COROLLARY 1. *Unless P=NP, the Constrained Load Rebalancing problem cannot be approximated below* 1.5 *in polynomial time.*

The best upper bound known 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 the 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 polynomial algorithm that approximates the makespan of the Conflict Scheduling problem within any ratio, unless P=NP.*

PROOF. We perform a reduction from 3-dimensional matching. We start with a description of the 3-dimensional matching 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 $|T_i \cap A| = |T_i \cap B| = |T_i \cap C| = 1$ for $i = 1, \ldots, m$.

Question: Does F contain a matching, i.e., a subfamily F' for which $|F'| = n$ and $\cup_{T_i \in F'} T_i = A \cup B \cup C$?

There are m machines, 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 machine, so a feasible solution would have spread these m jobs one per machines. 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., element job u can be assigned together with 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 them either. Disregard job costs and sizes, if there is a feasible assignment of jobs to machines that satisfy the conflicts, then the following holds:

- There is one triple job per machine.
- There are $m - n$ machines, each has one dummy job.
- All the $3n$ element jobs are assigned among the rest of the n machines. Each machine is assigned exactly 3 element jobs corresponds to a_j, b_k, c_l, and the triple job on that machine is $T_i = (a_j, b_k, c_l)$.

So a feasible assignment of the jobs implies a matching of size n and vice versa. Notice that any approximation algorithm will give an answer that is feasible if and only if a feasible assignment exists, hence the result. $\square$

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. *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. *Proceedings of the 9th Annual European Symposium on Algorithms*, 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 ACM Symposium on Theory of Computing*, 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. *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 Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

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