Load Balancing for Response Time

Jeffery Westbrook*

July 11, 1995

Abstract

A centralized scheduler must assign tasks to servers, processing on-line a sequence of task arrivals and departures. Each task runs for an unknown length of time, but comes with a weight that measures resource utilization per unit time. The *response time* of a server is the sum of the weights of the tasks assigned to it. The goal is to minimize the maximum response time, *i.e.*, load, of any server. Previous papers on on-line load balancing have generally concentrated only on keeping the current maximum load bounded by some function of the maximum off-line load ever seen. Our goal is to keep the current maximum load on an on-line server bounded by a function of the current off-line load. Thus the loads are not permanently skewed by transient peaks, and the algorithm takes advantage of reductions in total weight. To achieve this, the scheduler must occasionally reassign tasks, in an attempt to decrease the maximum load. We study several variants of load balancing, including identical machines, related machines, restricted assignment tasks, and virtual circuit routing. In each case, only a limited amount of reassignment is used but the load is kept substantially lower than possible without reassignment.

1 Introduction

This paper examines on-line or dynamic load balancing problems that arise in heterogeneous distributed systems containing workstations, I/O devices, and variable-bandwidth communication channels. A typical instance consists of a fixed set V of n servers and a set of tasks U to be processed by those servers. Each task $u \in U$ is made up of a sequence of subtasks, or basic steps, and has a weight, w_u, which is in some way a measure of the amount of service needed to perform a basic step. The tasks arrive and depart on-line, that is, arrival time and the number of subtasks are unknown in advance.

For example, the servers may be communication channels between two sites, the tasks a video transmission made up of a stream of packets, and the weight of a task will be the time required to transmit one packet, generally a linear function of the packet size. The total number of packets to be sent is unknown, however, as is the start time of each task. As another example, the servers may be distributed database platforms, the tasks application programs performing a sequence of accesses to the database, and the weights the time for an individual access. Again, the start time of each application is unknown, as is the total number of queries per task.

The *load* on a server is the sum of the weights of the tasks assigned to it. The *response time* for a task is the time required to perform a basic unit of the task, such as transmit a single packet or

*Department of Computer Science, Yale University, New Haven, CT 06520.

answer a single database query. Assuming that a server timeshares fairly among the tasks assigned to it, the response time is bounded by the load. Response time is important when a task involves an interaction between a human user and the servers, such as audio or video transmission or database queries. Response time is also important when each task is a thread of a parallel computation, comprised of subtasks separated by synchronization barriers. Each task to should get to the synchronization barriers at roughly the same rate. The *load balancing problem* is the problem of keeping the tasks distributed over the servers so that no server has too high a load, *i.e.*, the maximum load is minimized.

More formally, a load-balancing problem consists of a game between an algorithm and an adversary. At the start of round t, there is a set of *active tasks*, $U(t)$. In a round, the adversary may add a new task to $U(t)$, or it may remove a task from $U(t)$. In response, the algorithm must assign each new task to some server or set of servers. It may also reassign tasks among servers. We refer to t as the *time*. An optimal assignment of the active tasks minimizes the maximum load. Let λ_t^* be the maximum load in an optimal assignment of the tasks active at time t. For an on-line load-balancing algorithm A, let $A(t)$ denote the maximum load on any server at time t if A is used to determine assignment of tasks to servers. Note that λ_t^* depends only $U(t)$, whereas $A(t)$ in general depends on the entire history of task arrivals and departures.

We say A is c-competitive against *peak load* if at for all t, $A(t) \leq \max_{t' \leq t} c\lambda_{t'}^*$. That is, the maximum load on an on-line server is bounded by c times the maximum optimal load ever seen. We say A is c-competitive against *current load* if for all t, $A(t) \leq c\lambda_t^*$. That is, the maximum load on an on-line server is no more than c times the maximum load in an optimal assignment of the currently active tasks.

Almost all previous papers on competitive on-line load balancing have dealt only with peak load. In [1, 3, 5], tasks never depart, in which case current and peak load are equivalent. In [2, 4], tasks may depart but the algorithms are competitive against peak load only. In this paper, we examine the design of algorithms that are competitive against current load.

The distinction between peak load and current load is quite significant. The peak load measurement is useful in network design, where it can be used, in conjunction with an estimate of the peak optimum load, to determine how fast a server must be to ensure a given upper bound on response time. The current load measurement is important when the system is actually running. Suppose a large number of video transmissions across a network start simultaneously. Viewers will see jerky images, since each individual frame in the image is slow to arrive through the network. Nothing can be done, since the network is heavily loaded. But suppose that most of transmissions are short and end quickly, leaving only a few long-term transmissions. The algorithm that assigns tasks to channels may have assigned all the long transmissions to the same channel, not knowing in advance which transmissions were long and which were short. Even if the load on this channel is competitive with the prior peak load, it is of little consolation to the viewers, who must continue to watch a jerky transmission even though the short-term jobs have left and there is enough capacity to send the long-term transmissions at a fast rate. The behavior of load-balancing algorithms that are competitive against current load cannot be skewed by an initial peak. The current load measure is also used in the application of load balancing to network flow described in [10]. Naturally, an algorithm that is c-competitive against current load is c-competitive against peak load.

The video transmission example suggests that to be competitive against current load, the on-line scheduler must sometimes reassign tasks to different servers. This should not be done too often, however. Usually, a certain amount of preprocessing is done in assigning a task, such as setting the switches in a virtual network circuit, or building search data structures for database queries. The cost of doing this work is called the *restart cost.* By constantly reassigning tasks, the algorithm can achieve the optimal maximum load, but all the server time will be taken up by restarts. Our results explore the tradeoff between restart cost and load balance.

We consider the following specific load balancing problems. The survey paper by Lawler *et al.* [9] lists many others.

- Identical Machines: Each task u has an associated weight w_u and can be served by any one of the servers. All servers run at the same speed. The load on server v is the sum of the weights of tasks assigned to it.
- Related Machines: The same as the previous problem, except that each server v runs at a different speed, or has a capacity, cap_v. The load on v is the sum of the assigned weights divided by the capacity.
- Restricted Assignment: Each task has a unit weight but can only be served by one of some subset of the machines. The load on a machine is the number of assigned tasks.
- Virtual Circuit Routing: The servers are the edges of an undirected graph. Each edge has an associated capacity cap_e. A task u is a request for a connection between two nodes s_u, t_u. The set of servers assigned to the task must form a path between s_u and t_u. Each task has an associated weight w_u. The load on an edge is the sum of the weights of the connections using that edge divided by its capacity.

For all these problems, an algorithm that never reassigns tasks is at best n-competitive against current load. Each problem contains the identical machines problem as a restriction. In the identical machines problem, an adversary may generate n^2 unit cost tasks, after which some server v must have load at least n. The adversary then deletes all tasks except for those on v. On the other hand, as noted in [4], Graham's [7] list processing heuristic for identical machines, which simply assigns each new task to the least loaded server, and never reassigns a task, is 2-competitive against peak load.

To measure the cost of reassignment, we assume that each task u has an associated restart cost r_u. The restart cost is incurred every time u is assigned or reassigned to some server or connection path. We assume that r_u depends only on u. That is, r_u does not vary with the time of assignment or the servers to which u is assigned. In addition, in all but the last section of this paper we assume that $r_u = c \cdot w_u$ for a fixed constant c. In the final section we give a general method to remove this last restriction, at the price of an increase in the competitive ratio. Dealing with restart costs that depend upon time and server is an intriguing open problem. The algorithms in this paper keep the total cost of reassignments bounded by some small function, usually linear, of the sum $S = \sum_{u \in U} r_u$. Since each task incurs its restart cost at least once, when it is first assigned, any algorithm must incur a total restart cost of at least S.

We first present a simple algorithm for the case of identical machines that is 6-competitive against current load, with total restart cost either $2S$ or $3S$, depending on whether the restart costs are unit or equal to a constant times the weight of a task, respectively. Section 3 gives a general *witness-based* strategy for constructing algorithms competitive against current load, and uses it to derive an

$(8+\epsilon)$-competitive algorithm for related machines, ϵ an arbitrarily small constant, with total restart cost $O(S)$. Previously, Azar *et al.* gave an algorithm for related machines that is 20-competitive against peak load [3], but not competitive against current load. Their algorithm never reassigns a task.

In Section 4 we study the restricted assignment problem, and give a rebalancing scheme that is parameterizable to trade off competitive ratio against restart cost; its best competitive ratio is $O(1)$ at a restart cost of $O(S \log n)$. Previously, Phillips and Westbrook [10] give a preemptive algorithm that is $O((\log n)/\rho)$-competitive against current load while incurring restart cost ρS, where $0 < \rho \leq 1$ is a user-specified parameter. Their algorithm works for arbitrary weights. Azar *et al.*[2] give an eager algorithm for the case of unit weights that is $O(1)$ competitive against peak load, but only if the optimum peak load is $\Omega(\log n)$. This algorithm reassigns each task $O(\log n)$ times, but is no better than n-competitive against current load.

In Section 5 we give an algorithm for virtual circuit routing that is $O(\log n)$-competitive against current load, with restart cost $O(S \log n \log(C/\text{cap}_{min}))$, where C is the sum of edge capacities and cap_{min} is the minimum edge capacity. This algorithm is based on an algorithm of Azar *et al.*[2] that is $O(\log n)$ competitive against peak load, not competitive against current load, and reassigns each task $O(\log n)$ times. Finally, in Section 6 we give a general method to handle restart cost functions of the form $r : U \rightarrow R^+$.

2 Identical Machines

In this section we give an algorithm that is 6-competitive against current load. It performs $2S$ reassignments in the case that the restart cost $r_u = 1$ for all $u \in U$, and $3S$ reassignments in the case that $r_u = cw_u$, c a constant, for all $u \in U$. There are several previous algorithms that are competitive against peak load with a ratio $2 - \epsilon$ for a small constant ϵ [6, 7, 8]. None of these reassign tasks, and none are better than n-competitive against current load.

First, consider a particularly simple case: $w_u = \omega$ for all $u \in U$, ω a constant. It is easy to show that $\lambda_t^* = \omega \lceil |U(t)|/n \rceil$. Furthermore, it is easy to maintain on-line an assignment of the active tasks that achieves this optimum load, and incurs total reassignment cost $2S$. When a task arrives, it is placed on a server with minimum load. When a task departs from server v_j, a task is moved from some server with maximum load to server v_j, unless v_j itself has maximum load. The restart cost of the reassignment is charged to the departing task.

Now suppose that the tasks weights are not all equal. At time t, let $w_{\min} = \min_{u \in U(t)} w_u$ and let $w_{\max} = \max_{u \in U(t)} w_u$. Let $a = \lfloor \log w_{\min} \rfloor$ and $b = \lfloor \log w_{\max} \rfloor$. Partition the jobs by size into $b - a + 1$ classes, so that class i contains all jobs $u \in U(t)$ such that $2^i \leq w_u < 2^{i+1}$, $a \leq i \leq b$.

We treat each class independently. Within class i, we treat each job as if it had size 2^{i+1}, and run the unit-weight algorithm. If a new task arrives, it is assigned into the appropriate class, adjusting the partition upwards or downwards as necessary if the new task changes the bound a or b. When a task departs, the appropriate reassignment is done within its class, and the partition is again adjusted if the task departure changes a or b. Let U_t^i denote the set of tasks in class i.

Since class b contains at least one job, $\lambda_t^* \geq 2^b$. Also, $\lambda_t^* \geq \sum_{i=a}^{b} 2^i |U_t^i|/n$; this sum is a lower bound on the average total weight per processor. Conversely, the maximum load on any on-line server is at most the sum of the maximum loads in each class. Hence

$$\begin{aligned} A(t) &\leq \sum_{i=a}^{b} 2^{i+1}\lceil |U(t)^i|/n\rceil \leq \sum_{i=a}^{b} 2^{i+1}(|U(t)^i|/n+1) \\ &\leq 2\left(\sum_{i=a}^{b} 2^i |U(t)^i|/n\right) + 4\cdot 2^b. \end{aligned}$$

Theorem 1 *The algorithm maintains load within* 6 *times the current optimum. Over all assignments and reassignments, the total cost of reassignments is order* $2S$*, where* $S = \sum_{u\in U} r_u$*, if* $r_u = 1\,\forall u \in U$*, or* $3S$*, if* $r_u = cw_u\,\forall u\in U$*,* c *a constant.*

Proof: The competitive ratio is maximized when the two lower bounds on λ_t^* are equal, which gives a ratio of 6. Each departing job is charged for at most one reassignment. In the case that $r_u = cw_u$, however, the reassigned job may be as much as twice as large as the departing job, in which the departing job is charged twice its own restart cost. □

3 Related Machines

In the related machines problem, each machine v_j, $1 \leq j \leq n$, has a particular capacity or throughput, cap_j. Without loss of generality, let $\text{cap}_j \geq \text{cap}_{j+1}$ for all $1 \leq j < n$. Let $W_j(t)$ denote the sum of the weights of tasks assigned to server v_j at time t. The load on server j at time t is given by $W_j(t)/\text{cap}_j$. An algorithm for the special case that tasks never depart is given in [1].

3.1 Witness-based partitions.

The paradigm used here and in the remaining sections of this paper is as follows. Assume the existence of a load-balancing algorithm, A, parameterized by λ, with the following properties. When a new task u is added to the active set U_t, A either assigns u so that the maximum load is at most $c(n)\lambda$, or reports (correctly) that in any assignment of $U_t \cup \{u\}$, the maximum load is greater than λ. In the first case A *accepts* u, and in the second case A *rejects* u. Algorithm A may perform rebalancing after task arrivals or departures, but the restart cost incurred will be limited. Algorithm A is called the *one-level* algorithm and is used as a subroutine.

We maintain a partition of the active tasks, $U(t)$, into "levels," $U_i(t)$, $a \leq i \leq q$, where $q \leq \lceil \log \lambda_t^* \rceil$. The lower bound a depends on the particular application, but in general is chosen so that the restart cost can be bounded.

The partition changes as jobs arrive and depart, but will always satisfy the following condition: the load on a server due to jobs in $U_i(t)$ is at most $c(n)\lambda_i$, where $c(n)$ is some function of n and $\lambda_i = 2^i$. A partition that satisfies this conditions is called a $c(n)$*-witness partition.*

Lemma 2 *An on-line algorithm that maintains a* $c(n)$*-witness partition is* $4c(n)$*-competitive against current load.*

Proof: The on-line load is at most $\sum_{i=a}^{q} 2^i c(n) \leq \sum_{i=-\infty}^{q} 2^i c(n) \leq 2^{q+1}c(n) \leq c(n)2^{2+\log\lambda^t}$. $\square$

Each level in the partition is regarded as a separate instance of the load-balancing problem, and managed using the one-level algorithm. To assign a new task u, we proceed as follows. The value of a is adjusted as necessary. For related machines, we will set $a = min_{u \in U(t)} \lfloor \log w_u/(c(n)\text{cap}_1) \rfloor$, so a can only decrease. Then, starting at level $i = a$, we attempt to put u into level i, using the one-level algorithm. If the one-level algorithm accepts u, we are done. If it rejects u, we increment i by 1 and repeat. Once $i = \lceil \log \lambda_t^* \rceil$, the insertion must be accepted by definition of the one-level algorithm. Finally, we set $q = \max\{q, i\}$.

To delete task u, we remove it from whatever level it belongs to, using the one-level algorithm. This may cause local rebalancing within that level. The value of a is adjusted upward, if necessary. Then a global rebalancing phase occurs. Each task u in level-q is selected in turn, and an attempt is made to insert u into any level $i < q$. If u is accepted by a lower level, then the next task in level q is selected and rebalancing continues. If level q is emptied of tasks, then q is decremented and global rebalancing resumes on the new maximum level. If u is rejected by all lower levels, and by level $q-1$ in particular, then u is left in level q and global rebalancing terminates. By the definition of the one-level algorithm, a rejection implies $\lambda_t^* > 2^{q-1}$, which implies $q \leq \lceil \log \lambda_t^* \rceil \}$. Task u is called a *witness for load* 2^q. By using witnesses, the algorithm can avoid compute the exact value of λ_t^* (although it could be computed by brute force simulation, if necessary).

The partition limits the restart cost incurred during global rebalancing in the following way. Suppose task u is reassigned from q down to q'. When u first arrived, it must have been rejected by level q' to end up in q. Since q' now accepts u, it must be that some number of tasks have departed from level q' in the meantime. The restart cost of moving u can be amortized against the restart costs of the departed jobs. The precise details of the amortization argument depend on the application.

3.2 Algorithm ONE-LEVEL.

In this section we give a one-level algorithm for related machines, called ONE-LEVEL. Given a parameter λ, ONE-LEVEL keeps the load bounded by $(1+\beta)\lambda$ where $\beta > 1$ is any constant. If the total weight of all tasks ever accepted is W, then ONE-LEVEL reassigns a total weight of $W(1 + 1/(\beta - 1))$. This scheme is an adaptation of the algorithm of [1].

Recall $W_j(t)$ is the total weight assigned to processor v_j at time t. We also define a quantity $M_j(t)$, which will roughly be the maximum weight ever on processor v_j since it was last rebalanced. For convenience, we will often drop the time parameter, when the time is fixed, and simply refer to W_j and M_j.

Insertion. Let u be a new task, with weight w_u. Let j be the maximum index such that $(W_j + w_u)/\text{cap}_j \leq (1+\beta)\lambda$. If there is no such j, then reject task u. Otherwise, assign task u to processor j, increase W_j by w_u, and set $M_j = \max\{W_j, M_j\}$.

Deletion. Let u be the job that is departing, say from processor j. Decrease the value of W_j by w_t. Then apply the following rebalancing procedure:

1. Let $\ell = \max\{k \mid \sum_{j=1}^{k} M_j - \beta W_j \geq 0\}$
2. If there is no such ℓ, then stop.
3. Otherwise, for all $j \leq \ell$, set $M_j = 0$, $W_j = 0$. Preempt all jobs assigned to servers $v_1, ..., v_\ell$ and reassign them by re-inserting them one by one, using the above insertion algorithm.

The insertion and deletion routines preserve the following two properties:

Prop. 1. $M_j(t) \geq W_j(t)\ \forall t, j$.

Prop. 2. $\sum_{j=1}^{k}(M_i - 2W_i) < 0$, for all $1 \leq k \leq n$.

Lemma 3 *Let u be a task assigned to processor v_k. For all $j > k$, $(M_j + w_u)/cap_j > (1+\beta)\lambda$.*

Proof: Let t be the current time and $t' < t$ be the time at which ONE-LEVEL assigned u to v_k. Then $M_j(t) \geq M_j(t')$ for all $j > k$. Otherwise, M_j must have been decreased at some time $t'' > t'$. But M_j can only be decreased in a rebalancing operation, at which time all jobs on servers numbered lower than j are reassigned, contradicting the assertion that u was assigned to v_k at time $t' < t''$. Since u was not placed on server v_j, it must be that $(M_j(t') + w_u)/\text{cap}_u \geq (W_j(t') + w_u)/\text{cap}_u > (1+\beta)\lambda$. $\square$

Lemma 4 *Suppose task u is rejected upon attempted insertion at time t. Let λ^* be the maximum load in the optimum assignment of the active tasks, $U(t)$, including u. Then $\lambda^* > \lambda$.*

Proof: Let ℓ be minimal such that $M_\ell/\text{cap}_\ell < \beta\lambda$. If there is no such ℓ, define $\ell = n+1$.

Suppose $\ell = 1$. Since u is rejected it must be the case that $w_u)/\text{cap}_1 > (1+\beta)\lambda - (W_1/\text{cap}_1$. Since $\lambda^* \geq w_u/\text{cap}_1$ and $W_1/\text{cap}_1 \leq M_1/\text{cap}_1 < \beta\lambda$, it follows that $\lambda^* > \lambda$.

Suppose $\ell > 1$. For all servers $j < \ell$, $M_j/\text{cap}_j \geq \beta\lambda$. Let X be the set of tasks currently assigned by ONE-LEVEL to servers $v_1, \ldots, v_{\ell-1}$. Given some optimal assignment of tasks to servers, consider where the tasks in X are located.

Suppose that in the optimal assignment, all tasks in X are also assigned to servers $v_1, \ldots, v_{\ell-1}$. Let W_j^* be the weight on v_j in the optimal assignment. We have $\lambda^* \geq W_j^*/\text{cap}_j$ for all $j < n$. Hence $\lambda^* \sum_{i=1}^{\ell-1} \text{cap}_i \geq \sum_{i=1}^{\ell-1} W_i^* \geq \sum_{i=1}^{\ell-1} W_i$. By Prop. 3.2, $\sum_{i=1}^{\ell-1} W_i > \sum_{i=1}^{\ell-1} M_i/\beta \geq \lambda \sum_{i=1}^{\ell-1} \text{cap}_i$. Thus $\lambda^* > \lambda$.

Now suppose that in the optimal assignment there is some task $x \in X$ that is assigned to some server v_k, $k \geq \ell$. Suppose ONE-LEVEL has assigned x to $v_{k'}$, $k' < \ell$. By Lemma 3, $(M_\ell + w_x)/\text{cap}_\ell > (1+\beta)\lambda$. Since $M_\ell/\text{cap}_\ell < \beta\lambda$, it follows that $\lambda^* \geq w_x/\text{cap}_k \geq w_x/\text{cap}_\ell > \lambda$. $\square$

Lemma 5 *Over a series of m task insertions and departures, if the total weight of all tasks is W, the total weight of assigned and reassigned tasks is $(1 + \frac{1}{\beta-1})W$.*

Proof: We perform an amortized analysis. Let $M(t) = \sum_{j=1}^{n} M_j(t)$ and let

$$\Psi = \frac{M(t)}{\beta - 1}$$

When task u is assigned to processor v_j, the amortized assigned weight is . $w_u(1 + \frac{1}{\beta - 1})$. When u departs, Ψ does not change and the amortized weight reassigned is zero. Suppose a rebalancing step reassigns all tasks on servers $v_1, \ldots, v_k$. Let $\Delta W = \sum_{j=1}^{k} W_j$ and $\Delta M = \sum_{j=1}^{k} M_j$. The actual weight reassigned is ΔW. The potential initially decreases by $\Delta M/(\beta - 1)$ as M_j's are set to zero, and then increases by at most $\Delta W/(\beta - 1)$ as the tasks are reassigned. By definition, $\Delta M - \beta \Delta W \geq 0$, which implies $\Delta W - \Delta M f \leq -(\beta - 1)\Delta W$. The net potential change is $(\Delta W - \Delta M)/(\beta - 1) \leq -\Delta W$. Thus the amortized reassigned weight is zero. □

3.3 A witness-based algorithm for related machines.

Algorithm ONE-LEVEL is used to maintain a witness-based partition for related machines. To derive a good amortized bound on the restart cost, several modifications are made to the basic definitions of a witness based strategy.

First, call a task u *new* if it has never been assigned to a processor, otherwise *old*. Algorithm ONE-LEVEL is allowed to accept a new task into level i if it can be placed on some processor v_j without increasing the load beyond $(1 + \beta + \alpha)\lambda_i$, where $\alpha > 0$ is any constant. As before, ONE-LEVEL rejects an old task unless it can be placed without increasing the load beyond $(1 + \beta)\lambda_i$. Recall that the partition has levels $a \leq i \leq q$. The lower bound a is set to $min_{u \in U(t)} \lfloor \log w_u / ((1 + \beta + \alpha)\text{cap}_1) \rfloor$.

Second, in global rebalancing, determine q, the maximum occupied level, and v_j, the minimum numbered processor holding a task in level q. Select any task u' assigned to j at level q. Attempt to insert u into level $q - 1$ using algorithm ONE-LEVEL. If u is rejected, terminate rebalancing. If u is accepted, and reassigned from processor v_j in level q to processor v_k in level $q - 1$, then W_{qj} and M_{qj} are decreased by w_u.

Theorem 6 *The witness-based algorithm is $4(1 + \beta + \alpha)$-competitive against current load. If the total weight of all tasks is W, then the total weight that is reassigned is $W(1 + (\beta^2 + \beta + \alpha)/\alpha(\beta - 1))$.*

Proof: It is not hard to verify that Lemma 4 still holds for the modified algorithm. Propositions 3.2 and 3.2 remain true when M_{qj} and W_{qj} are reduced during global rebalancing. Lemma 3 also remains true: as long as task u is assigned to v_k in level q, no M_{qj}, $j > k$, can be reduced except in a ONE-LEVEL rebalancing. (M_{qj} can be reduced in global rebalancing only if there are no tasks on processors v_k, $k < j$.) Finally, if ONE-LEVEL rejects a new task u from level i, it is certainly true that for all j, $(W_{ij} + w_u)/\text{cap}_j > (1 + \beta)\lambda_i$. Hence the modified algorithm correctly maintains a witness partition, with $c(n) = (1 + \beta + \alpha)$. The competitive ratio follows from Lemma 2. To show the bound on total weight reassigned, we extend the amortized analysis of Lemma 5. Define a new value, $\hat{M}_{ij}$, which will roughly be the maximum total weight on processor v_j due to tasks in level i, since i was last the top level. This value is used only for purposes of analysis, and is not maintained by the algorithm. Upon any assignment of a task u to processor v_j in level i at time t, set $\hat{M}_{ij} = \max\{\hat{M}_{ij}, W_{ij}(t)\}$. During global rebalancing, if task u is reassigned from processor v_j in level q to some processor in level $q - 1$, set $\hat{M}_{qj} = \hat{M}_{qj} - w_u$.

Let $M = \sum_{i=0}^{q} \sum_{j=1}^{n} M_{ij}$ and $\hat{M} = \sum_{i=0}^{q} \sum_{j=1}^{n} \hat{M}_{ij}$. Define

$$\Phi_{ij} = \frac{1+\beta}{\alpha}\hat{M} + \frac{1}{\beta-1}\left(1+\frac{1+\beta}{\alpha}\right)M.$$

The amortized weight assigned in inserting new task u is $w_u(1+(\beta^2+\beta+\alpha)/\alpha(\beta-1))$. The amortized weight assigned in a deletion is zero. Consider a ONE-LEVEL rebalancing, with reference to the proof of Lemma 5. First M decreases by some amount ΔM, and then M and $\hat{M}$ increase by at most ΔW, where ΔW is the total weight reassigned in the rebalancing. The change in Φ is therefore

$$\frac{-\Delta M}{\beta-1}\left(1+\frac{1+\beta}{\alpha}\right) + \frac{1+\beta}{\alpha}\Delta W + \frac{\Delta W}{\beta-1}\left(1+\frac{1+\beta}{\alpha}\right).$$

Since $\Delta W - \Delta M \leq -(\beta-1)\Delta W$ (see Lemma 5) the net change in potential is $-\Delta W$, which pays for the actual weight reassigned.

Finally, consider a global rebalancing step, in which task u is moved from processor v_j in the top level, q, processor v_k in level $q-1$. The value of M is non-increasing, since M_{qj} decreases by w_u while $M_{(q-1)k}$ increases by at most w_u. The value of $\hat{M}_{qj}$ decreases by w_u, while $\hat{M}_{(q-1)k}$ increases by $w_u - s$, where s, the *slack*, is the difference between $\hat{M}_{(q-1)k}$ and $W_{(q-1)k}$ prior to reassigning u.

We claim that $s > \alpha\lambda_{q-1}\text{cap}_j$, for the following reason. If t is the current time, consider the time $t' < t$ at which u was new. The choice of a guarantees that level $q-1$ existed when u was new. Since u was rejected by level $q-1$, $(W_{(q-1)k}(t')+w_u)/\text{cap}_j > (1+\beta+\alpha)\lambda_{q-1}$. Since level $q-1$ has not been the top level since time t', $\hat{M}_{(q-1)k}$ has not decreased since time t'. Hence $\hat{M}_{(q-1)k} \geq W_{(q-1)k}(t')$. On the other hand, since u can fit on v_j, $(W_{(q-1)k}(t) + w_u)/\text{cap}_j \leq (1+\beta)\lambda_{q-1}$. Combining these observations yields $s > \alpha\lambda_{q-1}\text{cap}_j$.

We have that $w_u \leq (1+\beta)\lambda_{q-1}\text{cap}_j \leq s\frac{1+\beta}{\alpha}$. Therefore the potential decrease by at least w_u, and the amortized weight reassigned in a global rebalancing step is 0. $\square$

Corollary 7 *For any $\epsilon > 0$ there is a load-balancing algorithm for the related machines problem with competitive ratio $8+\epsilon$ that incurs $O(S)$ startup cost.*

The corollary follows from Theorem 6 by choosing α and β sufficiently small, and using the restriction that $S = cW$ for some constant c.

4 Eager Load Balancing for Restricted Assignment

In the restricted assignment problem, each task u has an associated subset of servers on which it can be executed. It must be assigned to one server in that subset. If no tasks ever depart and tasks cannot be preempted, the best possible competitive ratio for both randomized and deterministic algorithms is $\Omega(\log n)$; this ratio is achievable with a simple greedy strategy [5]. Azar *et al.*[4] showed that when tasks both arrive and depart, no non-preemptive algorithm can be better than $O(\sqrt{n})$ competitive against peak load. An non-preemptive algorithm that is $O(\sqrt{n})$ competitive against peak load is given in [3].

In this section we give an *eager* algorithm for the case of unit weights. It is eager because tasks may be reassigned after both arrival and departure of other tasks. The previous *lazy* algorithms reassigned tasks only in response to other tasks being deleted from the system. In the restricted assignment problem, no lazy algorithm can improve on the $\Omega(\log n)$ bound for permanent tasks without reassignment. The competitive ratio of the eager algorithm is parameterized by a value $1 < q < \log n$, which determines both the ratio and amount of reassignments. It may be as good as $O(1)$-competitive against current load. The algorithm maintains a witness-based partition using the following one-level algorithm parameterized by λ.

Regard the problem as a game on a dynamic bipartite graph. On one side are the servers, V, called *columns* in this section; on the other side are the tasks, U. An edge $\langle u, v\rangle$ indicates that u can be assigned to v. Edge $\langle u, v\rangle$ is *matching* if u is assigned to v. Given $X \subseteq U$, let $Y(X) = \{v \in V \mid \exists\langle u, v\rangle \in E, u \in X\}$. In other words, Y is the set of columns v such that some element in X has an edge to v. By the pigeonhole principle, $\lambda \geq \max_{X\subseteq U} |X|/|Y(X)|$.

Let *load*(v) denote the load on $v \in V$, *i.e.*, the number of matching edges incident on v. In this section we will also use the term *height* to mean the load. A *balancing path* is an even-length sequence of alternating matched and unmatched edges $\{v_1, u_1\}, \{u_1, v_2\}, \{v_2, u_2\}, \ldots, \{u_{m-1}, v_m\}$ with the property that $load(v_i) < load(v_1)$ for $1 \leq i \leq m-1$ and $load(v_m) < load(v_1) - \lambda$. A balancing path can be used to reduce the maximum load on $v_1, v_2, \ldots, v_m$ by reassigning u_i to v_{i+1} for $1 \leq i \leq m-1$. The servers are *r-balanced* if there is no balancing path of length r or less.

Lemma 8 *If the n servers are $2q$-balanced, $1 \leq q \leq \ln n$, then the maximum on-line load is $c\lambda$, where c satisfies the inequality*

$$\ln n/q \geq c(\ln c - 1) \tag{1}$$

Proof: Let h be the maximum j such that there is an on-line server of height at least $j\lambda$. For all $j \geq 0$, let Y_i^j be the set of columns $v \in V$ that are reachable by an alternating path of length at most $2i$, $1 \leq i \leq q$, starting from a column of height at least $j\lambda$. Thus Y_0^j is the set of columns of height at least $j\lambda$. Let $y_i^j = |Y_i^j|$.

No column in Y_i^j has height less than $(j-1)\lambda$, since otherwise there would be a balancing path of length $2i \leq 2q$. Let X_i be the set of items on columns of height at least λi. We have $|X_i| \geq i\lambda y_0^i$.

For any j and $i \leq (q-1)$, the set Y_{i+1}^j contains all servers adjacent to some item that is currently placed on a server in Y_i^j. There are at least $\lambda(j-1)y_i^j$ such items, and hence by the pigeonhole principle

$$y_i^j(j-1)\lambda/y_{i+1}^j \leq \lambda. \tag{2}$$

We also have (2) $y_0^j \geq y_q^{j+1}$. This follows from the observation that no server of load less than $j\lambda$ is reachable in $2q$ or fewer steps from a server of load $(j+1)\lambda$, or else there must be a $2q$-balancing path. Combining equations 2 and (2) we derive the following recurrence:

$$y_0^j \quad \geq \quad (j)^q y_0^{j+1} \qquad y_0^0 = n$$

Solving the recurrence, we find $n \geq y^h(h!)^q$. Since y^h is at least 1, this implies $n \geq (h!)^q$. Applying the natural logarithm and Stirling's approximation yields $\ln n/q \geq h(\ln h - 1)$. $\Box$

Lemma 9 *For any q, all servers can be kept $2q$-balanced using $O(qh)$ reassignments per insertion or deletion.*

Proof: To insert an item u, place it on any server v to which it is adjacent. This increases the height of v and may create a balancing path starting at v. It cannot, however, create a rebalancing path originating at any other node. Rebalance along any balancing path starting at v and terminating at v_1. At the conclusion of rebalancing, v is returned to its initial height, server v_1 has increased in height by 1, and all other servers have unchanged height. Recursively apply the same procedure starting at v_1. By the definition of a rebalancing path, $\mathit{load}(v_1) = \mathit{load}(v) - \lambda$ after rebalancing. Hence the procedure can only be applied $O(h)$ times before reaching a column of height 1. Each call performs $O(q)$ reassignments.

The case of deletion is similar, except that deleting an item from v may create a rebalancing path terminating at v, and recursive calls occur at servers that are increasing in height by λ. □

We use the eager rebalancing algorithm as a one-level algorithm in a witness-based algorithm. As in the related machines problem, a series of levels are maintained, with load parameter $\lambda_i = 2^i$, $a \leq i \leq b$, with $a = 0$. After a deletions, rebalancing is performed within the affected level, and then repeatedly a task from the most heavily loaded column in level q is chosen, and an attempt is made to insert it into a lower column. As before, if the insertion fails, the element is a witness for load 2^q. Observe that removing a task from the most heavily loaded column cannot create a balancing path. Let h be the competitiveness constant selected for the eager algorithm. An insertion fails if the task cannot be placed without increasing the load on some column above $h\lambda_i$.

Theorem 10 *The eager rebalancing witness-based algorithm is $O(h)$ competitive against current load and performs $O(qh)$ rebalances per item, where h is the value of c satisfying equation 1 for parameter q.*

Proof: For the purpose of analysis, define M_{ij} for each column $1 \leq j \leq n$ and each level $0 \leq i \leq q$. When a task is inserted onto column j, set $M_{ij} = \max\{M_{ij}, W_{ij}\}$. When a task is moved from column j in level q to column k in level $q-1$, decrease M_{qj} by 1.

Define
$$\Phi = qh \sum_{i=0}^{q} \sum_{j=1}^{n} M_{ij}.$$
We make an additional definition, also solely for the purpose of analysis. Each task, u, is given a bit, $b(u)$. The potential associated with a task is $\Psi(u) = qh \cdot b(u)$. When a task u is new, $b(u) = 1$. The bit may be set to 0 at various times during rebalancing steps, to be described below. If u is inserted into a lower level, its bit is set to 0.

The potential increase when an item is new is $O(qh)$, and so the amortized weight assigned is $O(qh)$. The amortized weight of a task departing is zero, since the potential can only decrease.

Finally we consider the amortized cost of reassigning a task u from v_j in level q to v_k in level $q-1$. The value of M_{qj} decreases by qh. It remains to show that there is no increase in the potential due to the insertion of u into level $q-1$. This proof is rather intricate. We say a column j *refuses* the insertion of u in level i if u cannot be placed on v_j because $\mathit{load}(v_j) = h\lambda_i$ and there is no balancing path. A column j is called *completed* in level i if it has ever refused an insertion in that level. A task u is called *free* in level i if, for any column j to which u can be matched, $M_{ij} > (h-1)\lambda_i$. (It is then free to be moved anywhere as part of rebalancing without increasing M_{ij}, since no task can be rematched to a column with load greater than $(h-1)\lambda_i$, by the definition of the rebalancing algorithm.)

Given these definitions, several observations can be made.

i. A task that has been assigned to level q is free in all lower levels, since by the insertion procedure, it must have been refused by all eligible columns in lower levels.

ii. Suppose column j refuses the insertion of task u. Let u' be any task already matched to column j at the time of the refusal. Then u' is free in level i. Otherwise, u' is adjacent to some column k such that $M_{ik} \leq (h-1)\lambda_i$. But then there is a balancing path from j to k, and the insertion would not be refused.

The bit values of tasks are maintained so that the following two properties are invariant.

A. If u is assigned to a completed column j, but u is not free, then $b(u) = 1$.

B. If task u is moved from v_j to v_k, then at the time it is moved either u is free or $b(u) = 1$.

We give rules for maintaining bit values, and show that, if the invariants hold at the start of a single move during a rebalancing step, they hold at the end of a single move. Suppose u is moved from column j to k. One possibility is that rebalancing terminates at k. If v_k is complete, then M_{ik} cannot increase. Invariant A remains true at k, because either u is free or $b(u) = 1$, by Invariant B. If v_k is not complete, then u cannot be free. Hence $b(u) = 1$, by Invariant B. If M_{ik} increases, bit $b(u)$ is set to zero. Invariant A does not apply.

A second possibility is that rebalancing continues, and an item u' is chosen to move off k. In this case, M_{ik} does not increase. If k is complete, then by Invariant A, either u' is free or $b(u') = 1$. In either case, Invariant B is satisfied as u' moves. Invariant A holds at k, since by Invariant B either u is free or $b(u) = 1$. If k is not complete, then u cannot be free, and so by Invariant B, $b(u) = 1$. Invariant A is not applicable. If $b(u') = 0$, then set $b(u') = 1$ and $b(u) = 0$. Otherwise, no bits are changed. This ensures that Invariant B holds.

When a column j first becomes complete, Invariant A holds, because all items on j are free. (Refer to Observation (i)).

In the one case above that M_{ij} increases, Ψ decreases by a matching amount. Therefore, the change in total potential due to moving item down pays for the cost of the insertion. □

Corollary 11 *For any $c < 1$, there is an algorithm that performs $O(\log n/\log\log n)$ reassignments and keeps the on-line load within a factor of $O((\log n)^c/\log\log n)$ of the current load.*

Corollary 12 *There is an algorithm that performs $O(\log n)$ reassignments and keeps the on-line load within a factor of $O(1)$ of the current load.*

Corollary 11 follows from setting $q = (\log n)^{1-c}$, and Corollary 12 from setting $q = \log n$. This improves on the results of [2] in two ways: it works for all values of the optimum load and it is competitive against current rather than peak load.

5 Virtual Circuit Routing

In the virtual circuit routing problem one is given a communication network modeled by an undirected graph. Each edge $e \in E$ has an associated capacity cap_e. A task u is a triple (w_u, s_u, t_u), indicating the need for a weight w_u connection between nodes s_u and t_u. An on-line algorithm must choose a path in the graph between s_u and t_u to serve as the virtual connection. The current weight on each edge, W_e, is increased by w_u. The load on an edge is W_e/cap_e. Recall we assume that the restart cost, $r_u = c \cdot w_u$. This restriction can be eased using the method of Section 6.

Azar *et al.* [2] give a one-level algorithm, parameterized by λ, that is $O(\log n)$ competitive and reassigns each task $O(\log n)$ times. We use this in a witness-based algorithm that is competitive against current load.

Let $C = \sum_{e\in E} \text{cap}_e$ and let $W(t) = \sum_{u \in U(t)} w_u$. Let λ_t^* be the optimum maximum load. We have

$$\begin{aligned} \lambda_t^* &\geq W_e/\text{cap}_e \quad \forall e \in E \\ \lambda_t^* \sum_{e\in E} c(e) &\geq \sum_{e \in E} W_e \geq W(t) \\ \lambda_t^* &\geq W/C \end{aligned}$$

On the other hand, $\lambda_t^* < W(t)/\text{cap}_{min}$, where $\text{cap}_{min} = \min_{e\in E}\{\text{cap}_e\}$.

In the witness partition, the lowest level, a, is approximately $\lfloor \log(W(t)/C \rfloor$. The value of a is only adjusted, however, if the value of $W(t)$ has doubled or halved since a was last set. In particular, a value $\bar{W}$ is maintained. Initially, $\bar{W} = 0$, and at at all times, $a \geq \lfloor \log(\bar{W}/C) \rfloor - 1$. If ever $\bar{W} > 2W(t)$ or $W(t) > 2\bar{W}$, the algorithm resets $\bar{W} = W(t)$, and recalculates a. The value of a changes by plus or minus one. If a increases, all jobs assigned to the former bottom level are reassigned using the standard reassignment algorithm.

If the new value of a increased The highest level, q, is at most $\lceil \log W(t)/\text{cap}_{min} \rceil$. During global rebalancing, select tasks to move down from level q in any order. Deleting a task from q and inserting it into $j < q$ may cause rebalancing within levels q and j.

Theorem 13 *The witness-based virtual circuit algorithm is is $O(\log n)$-competitive against current load. If the total weight of all tasks is W, the total reassigned weight is $O(W \log n \log C/\mathit{cap}_{min})$.*

Proof: We give an amortized bound on the total weight reassigned. For task $u \in U(t)$, let $\ell(u)$ be the level containing u and $a(u,i)$ be the number of times u has been reassigned by the one-level algorithm for level i. Define

$$\Psi_u = [(c_1 \log n + 1)(\ell(u) - a + 1) - a(u,i)]w_u,$$

where $c_1 \log n$ is the maximum number of reassignments within a level, due to the one-level algorithm. Define a potential function

$$\Phi(t) = 2(1 + c_1 \log n \log C/\text{cap}_{min})|W(t) - \bar{W}| + \sum_{u\in U(t)} \Psi_u.$$

When u is newly arrived,

$$\ell(u) - a = O(\log W(t)/\text{cap}_{min} - \log(W(t)/C)) = O(\log n \log C/\text{cap}_{min}).$$

Hence the amortized weight assigned on a new insertion is $O(w_u \log n \log(C/\text{cap}_{min}))$. The amortized weight reassigned when u is reassigned within a level or across levels is 0, since the decrease in Φ pays for the reinsertion.

If a increases by 1, then at most $W(t)$ weight is reassigned. This weight increases in level by at most $\log C/\text{cap}_{min}$. Hence the increase in potential due to increases in Ψ terms is at most $(c_1 \log n \log C/\text{cap}_{min})W(t)$. Since $2\bar{W} < W(t)$, the decrease in potential after $\bar{W}$ is reset is at least

$$2(1 + c_1 \log n \log(C/\text{cap}_{min}))W(t)/2.$$

Hence the decrease pays for the increase in Ψ as well as the reassignment of all tasks, for a net amortized cost of 0.

If a decreases by 1, then $\bar{W} > 2W(t)$. No weight is reassigned, but the value of $\ell(u) - a$ increases by 1. The potential increase due to Ψ terms is therefore $(c_1 \log n + 1)(Wt)$. This is paid for by the decrease of $2(1 + c_1 \log n \log C/\text{cap}_{min})W(t)$ in the first term. $\square$

Under the assumption that $r_u = c \cdot w_u$ the total reassignment cost is $O(S \log n \log C/\text{cap}_{min})$.

6 Unrelated Weights and Startup Costs

In the previous sections we gave algorithms that provide competitive bounds while guaranteeing that the total weight that is assigned and reassigned is bounded by a small function f of the total weight of the input tasks. Under the assumption that $w_u = c \cdot r_u$, this implies that the total assignment cost is bounded by the same function f on the total startup cost. In this section we show how to extend these algorithms to handle any restart cost function of the form $r : u \rightarrow R^+$ (but still not independent of t and the servers involved in the restart.) Tasks are partitioned by startup cost per unit weight, r_u/w_u. Let a be the minimum value of this ratio over all input tasks and b the maximum value.

Choose a parameter $1 < \delta \leq (b/a)$. Partition the input tasks into $O(\log_\delta(b/a))$ classes such that task u is in class i if

$$a\delta^i \leq \frac{r_u}{w_u} < a\delta^{i+1}.$$

Within each class run the load-balancing algorithm appropriate to the problem being solved.

Lemma 14 *Let A be an algorithm that achieves a competitive ratio of c while the total weight of reassigned items is bounded by $g(W)$, where $W = \sum_{u \in U} w_u$ and $g(x) = \Omega(x)$. Then there exists an algorithm A_δ that achieves a competitive ratio of $c \log_\delta(b/a)$ and incurs a total reassignment cost $a\delta^{i+1} g(S/a\delta^i)$, where $S = \sum_{u \in U} r_u$.*

Proof: Within each of the classes, the algorithm is c competitive with the optimum load for tasks within that class. Let λ^* be the maximum over classes of the optimum load within that class. This is a lower bound on the true optimum load of all tasks. Since within each class no server has load greater than $c\lambda^*$, the maximum on-line load is $O(c \log_\delta(b/a)$. Within class i, the reassignment cost per unit weight is at most $a\delta^{i+1}$, hence the total cost of reassignments is $a\delta^{i+1}g(W)$. On the other hand, $W \geq S/(a\delta^i)$. $\square$

Using Lemma 14 we have the following:

- Algorithms for identical and related machines that are $O(\log_\delta(b/a))$ competitive against current load and incur total assignment cost $O(\delta S)$.
- An algorithm for the restricted machines problem that is $O(\log_\delta(b/a))$ competitive against current load and incurs reassignment cost $O(\delta \log n)$ when all weights are unit, and an algorithm for the restricted machines problem that is $O(\frac{1}{\rho} \log n \log_\delta(b/a)$ competitive against current load and incurs a reassignment cost $O(\rho\delta S)$ for arbitrary weights, where ρ is any parameter between 0 and 1.
- An algorithm for virtual circuit routing that is $O(\log n \log_\delta b/a)$ competitive against peak load and incurs a reassignment cost $O(\delta S \log n \log(C/\text{cap}_{min})$.

All results follow by applying the lemma to algorithms presented herein, with the exception of the algorithm for general weights in the restricted subset case, which follows by applying the lemma to the algorithm in [10]

While the bounds of this section are not ideal, in that they depend on the value of a ratio between input costs, they are independent of the number of tasks. An interesting open problem is to find a scheme with competitive ratio solely a function of the number of machines, n.

References

[1] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 623–631, 1993.

[2] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In *Proc. ACM/SIAM Symp. on Discrete Algorithms*, pages 321–330, 1994.

[3] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Proc. 1993 Workshop on Algorithms and Data Structures (WADS 93), Lecture Notes in Computer Science 709.* Springer-Verlag, Aug. 1993.

[4] Y. Azar, A. Karlin, and A. Broder. On-line load balancing. In *Proc. 33nd Symp. of Foundations of Computer Science*, pages 218–225, 1992.

[5] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 203–210, 1992.

[6] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24nd ACM Symp. on Theory of Computing*, 1992.

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

[8] D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. 1994 ACM/SIAM Symp. on Discrete Algorithms*, 1994. To appear.

[9] E. L. Lawler, J. K. Lenstra, A. H. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. Rinnooy Kan, and P. Zipkin, editors, *Handbook of Operations Research and Management Science, Volume IV: Production Planning and Inventory*, pages 445–522. North-Holland, 1993.

[10] S. Phillips and J. Westbrook. On-line load balancing and network flow. In *Proc. 1993 Symp. on Theory of Computing*, pages 402–411, Apr. 1993.