On-line Load Balancing

Yossi Azar[1]

Dept. of Computer Science, Tel-Aviv University. *

Abstract. We survey on-line load balancing on various models.

1 Introduction

General: The machine load balancing problem is defined as follows: There are n parallel machines and a number of independent tasks (jobs); the tasks arrive at arbitrary times, where each task has an associated *load vector* and duration. A task has to be assigned immediately to exactly one of the machines, thereby increasing the *load* on this machine by the amount specified by the corresponding coordinate of the load vector for the duration of the task. All tasks must be assigned, i.e., no admission control is allowed. The goal is usually to minimize the maximum load, but we also consider other goal functions. We mainly consider non-preemptive load balancing, but in some cases we may allow preemption *i.e.,* reassignments of tasks. All the decisions are made by a centralized controller.

The online load balancing problem naturally arises in many applications involving allocation of resources. As a simple concrete example, consider the case where each "machine" represents a communication channel with bounded bandwidth. The problem is to assign each incoming request for bandwidth to one of the channels. Assigning a request to a certain communication channel increases the load on this channel, *i.e.,* increases the percentage of the used bandwidth. The load is increased for the duration associated with the request.

Load vs. Time: There are two independent parameters that characterize the tasks: the first is the duration and second is the type of the load vector. This leads to several load balancing problems. Note that the main difference between scheduling and load-balancing is that in scheduling there is only one axis (duration or load) in load balancing they are two independent axes(duration and load). Thus we may have tasks of high or low load with short or long durations. For example, in storing files on set of servers files that may have various sizes are created or deleted at arbitrary times which may be unrelated to their sizes.

Duration: We classify the duration of tasks as follows: we call tasks which start at arbitrary times but continue forever *permanent*, while tasks that begin and end are said to be *temporary*. The duration of each task may or may not

* E-Mail: azar@math.tau.ac.il. Research supported in part by Allon Fellowship and by the Israel Science Foundation administered by the Israel Academy of Sciences.

be known upon its arrival. Note that tasks arrive at specific times, depart at specific times and are active for their whole durations. Thus delaying tasks is not allowed. For example, in storing files on servers, once a file is created it must be stored immediately on a server until it is deleted.

Clearly, permanent tasks are an important special case of temporary ones, (departure time is ∞ or very large) and better results can be achieved for them. Also knowing the duration of (temporary) tasks may help in achieving better results compared with the unknown duration case. Note that permanent tasks may be viewed also in the scheduling framework; where "load" corresponds to "execution time" (the durations are ignored). Restating the problem in these terms, our goal is to decrease maximum execution time under the requirement that the arriving tasks are *scheduled immediately*.

Load vector: Formally, each arriving task j has an associated *load vector*, $\mathbf{p}(j) = (p_1(j), p_2(j), \ldots, p_n(j))$ where $p_i(j)$ defines the increase in the load of machine i if we were to assign task j to it. The load vector can be categorized in several classes: identical machines case, related machines case, restricted assignment case and unrelated machines case.

In the *identical* machines case, all the coordinates of a load vector are the same. In the *related* machines case, the ith coordinate of each load vector is equal to $w(j)/v_i$, where the "weight" $w(j)$ depends only on the task j and the "speed" v_i depends only on the machine i. In the *restricted assignment* case each task has a weight and can be assigned to one of a subset of machines. In terms of the load vector the coordinates are either $w(j)$ or ∞. The *unrelated* machines case is the most general case, i.e., $p_i(j)$ are arbitrary non-negative real numbers (∞ may be represented by large M). Clearly, related machines and the restricted assignment are not comparable, but are both special cases of the unrelated machines case. The identical machines case is a special case of the related machines where all the speeds v_i are the same. It is also a special case of restricted assignment where all the coordinates of tasks j are $w(j)$.

The measure: Since the arriving tasks have to be assigned without knowledge of the future tasks, it is natural to evaluate the performance in terms of the *competitive ratio* w.r.t. some performance measure e.g., the maximum load. In our case, the competitive ratio is the supremum, over all possible input sequences, of the ratio between the performance achieved by the on-line algorithm and the performance achieved by the optimal off-line algorithm. The competitive ratio may be constant or depends on the number of machines n (which is usually relatively small). It should not depend on the number of tasks that may be arbitrarily large.

The most popular performance measure is the maximum load, i.e, the maximum over machines and over time of the load. This measure focuses on the worst machine and is the equivalent of the makespan for scheduling problems. However, the maximum load is not the only reasonable measure. Measures that take into account how well all the machines are balanced are, for example, the L_2 norm or the L_p norm of the load vector.

To emphasize the difference between the maximum load and the L_2 norm we take an example where each task sees a delay in service that is proportional to the number (or total weight) of tasks that are assigned to its machine. Then the traditional load balancing which is to minimize the maximum load corresponds to minimize the *maximum* delay. Minimizing the sum of squares (equivalently, minimizing the L_2 norm) corresponds to minimizing the *average* delay of tasks in the system.

We note that all the definitions and theorems are stated for the maximum load performance measure except in the section where the L_p norm is considered.

Reassignments: We recall that each task must be assigned to some machine for its duration. It turns out that the performance of load balancing algorithms may be significantly improved in some cases if we allow limited amount of re-assignments. More specifically, the algorithm can reassign some of the existing tasks.

Observe that if the number of reassignments per task is not limited, it is trivial to maintain optimum load, *i.e.* competitive ratio of 1 by reassigning optimally all the current tasks. However, reassignments are expensive process and should be limited. Thus, we measure the quality of an on-line algorithm by the competitive ratio achieved and the number of reassignments performed during the run.

Virtual circuit routing: Some of the algorithms for virtual circuit routing are extensions of the algorithms for load balancing. We consider the following idealized setting: We are given a network. Requests for virtual circuits arrive on line, where each request specifies source and destination points, and a load vector (the number of coordinate is the number edges of the network). The routing algorithm has to choose a path from the source to the destination, thereby increasing the load on each edge of the path by the corresponding coordinate of the load vector for the duration of the virtual circuit. The duration may or may not be specified in advance. The goal is to minimize the maximum over all edges of the load or some other function of the load. Reassignments which are called here reroutings may or may not be allowed.

The above problem is called *generalized* virtual circuit routing problem. It is easy to see that the load balancing problem is a special case of generalized virtual circuit routing. Load balancing can be reduced to a generalized virtual circuit routing problem on a 2 vertices network with multiple edges between them. Every edge corresponds to a machine and every arriving task is translated into a request between the two vertices s and t with the same load vector.

In the classical (in contrast to generalized) virtual circuit routing problem we assume that the load vector of request j on edge e is $r(j)/c(e)$ where $r(j)$ is the requested bandwidth and $c(e)$ is the capacity of the edge. Clearly the identical and related machines problems are special cases of virtual circuit routing on the 2 vertices network. There are various models for virtual circuit routing problems (e.g. allowing admission control and maximizing the throughput). For surveys on on-line virtual circuit routing refer to [21] and its references.

2 Definition and basic schemes

2.1 The model

The input sequence consists of task arrival events for permanent tasks and of task arrival and departure events for temporary tasks. Since the state of the system changes only as a result of one of these events, the event numbers can serve as time units, *i.e.* we can view time as being *discrete*. We say that time t corresponds to the tth event. Initially the time is 0, and time 1 is the time at which the first task arrives. Whenever we speak about the "state of the system at time t" we mean the state of the system *after* the tth event was already handled. In other words, the response to the tth event takes the system from the "state at $t-1$" to the "state at t".

A task j is represented by its "load vector" $\mathbf{p}(j) = (p_1(j), p_2(j), \ldots, p_n(j))$, where $p_i(j) \geq 0$. Let $\ell_i(t-1)$ denotes the load on machine i at time $t-1$. If at time t a task j is assigned to machine i, the load on this machine increases by $p_i(j)$. In other words $\ell_k(t)$ which denotes the load on machine k at time t, *i.e.*, after the tth event is defined as follows:

$$\ell_k(t) = \begin{cases} \ell_k(t-1) + p_k(j) & \text{if } k = i \\ \ell_k(t-1) & \text{otherwise} \end{cases}$$

Similarly, if at time t a task j departs from machine i, the load on this machine decreases by $p_i(j)$.

Let $\sigma = (\sigma_1, \sigma_2, \ldots)$ be a particular sequence of arrivals and departures of tasks. Denote by $\ell_k^*(t)$ the load of the optimal off-line algorithm on machine k at time t. The maximum load achievable by an optimum offline algorithm is denoted by $OPT(\sigma)$ which is the maximum of $\ell_k^*(t)$ over time and machines. If σ is clear from the context, we will use OPT for $OPT(\sigma)$.

Recall that for *identical* machines, $\forall i, j : \; p_i(j) = w(j)$. For *related* machines, $\forall i, j : \; p_i(j) = w(j)/v_i$ where v_i denotes the speed of machine i. For *restricted* assignment $p_i(j)$ is either $w(j)$ or ∞. For *unrelated* machines $p_i(j)$ are arbitrary non-negative real numbers.

2.2 Doubling

Several algorithms are designed as if the value of the optimal algorithm is known. This assumption can be easily eliminated by using simple doubling and losing a factor of 4 in the competitive ratio. More specifically, we define the notion of a *designed performance guarantee* β as follows: the algorithm $A(\Lambda)$ accepts a parameter Λ and never creates load that exceeds $\beta\Lambda$. $A(\Lambda)$ is allowed to return "fail" and to refuse to assign a task if $\Lambda < OPT$ otherwise it has to assign all of the tasks.

The algorithm A works in phases, the difference between phases is the value of Λ assumed by the algorithm. Within a phase the algorithm $A(\Lambda)$ is used to assign tasks, while ignoring all tasks assigned in previous phases. The first phase has $\Lambda = \min_i p_i(1) = \min_i p_i(1)$, which is the minimum possible load the first (non-zero) task may produce. At the beginning of every subsequent phase the value of Λ doubles. A new phase starts when the algorithm returns "fail". Thus, the last phase will never end.

It is easy to see that this approach set the competitive factor of A to be larger than the designed performance guarantee by a factor of 4 (a factor of 2 due to the load in all the rest of the phases except the last, and another factor of 2 due to imprecise approximation of OPT by Λ). Thus the competitive ratio is 4β. We note that the factor of 4 can be replaces by $e = 2.7..$ for restricted class of algorithms using randomization (see [27]).

3 Permanent tasks

Tasks which start at arbitrary times but continue forever are called *permanent*. The situation in which all tasks are permanent is classified as permanent tasks. Otherwise, it is classified as temporary tasks. (Permanent task is a special case of temporary one by assuming ∞ or high departure time.) In the permanent tasks case, task j is assigned at time j. Thus $\ell_k(j)$ corresponds to the load on machine k immediately after task j has been assigned.

We note that for the off-line problems there are polynomial approximation schemes for identical and related machines cases [26, 25] and 2 approximation for restricted assignment and unrelated machines cases [31].

3.1 Identical machines

In the *identical* machines case, all the coordinates of a load vector are the same. This case was first considered by Graham [23, 24] who considered the natural greedy which is assigning the next task to the machine with the current minimum load.

Theorem 1. [23] *The greedy algorithm has a competitive ratio of exactly* $2 - \frac{1}{n}$ *where* n *is the number machines.*

The fact that greedy is not better than $2 - \frac{1}{n}$ is shown by a simple example which is $n(n-1)$ unit size tasks followed by one task of size n. For $n = 2$ and $n = 3$ the competitive ratios are $3/2$ and $5/3$ (respectively) and are optimal. Somewhat better algorithms for small $n \geq 4$ appear in [22, 19]. It took some time until an algorithm whose competitive ratio is strictly below $c < 2$ (for all n) was found [13]. The competitive ratio of this algorithm, which does not always assign the next task to the lowest loaded machine, is $2 - \epsilon$ for a small constant ϵ. The algorithm was modified in [28] and its competitive ratio was improved to 1.945. Recently, Albers [1] designed 1.923 competitive algorithm and improved the lower bound for large number of machines to 1.852 (the previous lower bound for permanent tasks was 1.8370 [14]). We may also consider randomized algorithms. For example for two machines the competitive ratio (upper and lower bound) is 4/3 [13]. Somewhat better algorithms for small $n \geq 4$ appear in [34]. For large

n the best lower bound for randomized algorithms is 1.582 [18, 35] and the best randomized algorithm is just the deterministic one.

It is worthwhile to note that one may consider the case where the value of OPT is known. Then, for two machines the competitive ratio is exactly $4/3$ [30]. For any n a deterministic algorithm that is 1.625 competitive is given in [12]. The lower bound for this case is only $4/3$.

The identical machines case for permanent tasks is also considered in the on-line scheduling framework. It is also called jobs arriving one by one. A comprehensive survey on on-line scheduling appears in [36].

Open problem 3.1 *Determine the competitive ratio for deterministic and randomized load balancing algorithm for permanent tasks on identical machines.*

3.2 Related machines

Recall that in the *related* machines case, the ith coordinate of each load vector is equal to $w(j)/v_i$, where the "weight" $w(j)$ depends only on the task j and the "speed" v_i depends only on the machine i. This case was considered in [4] who showed an 8 competitive algorithm for the permanent tasks.

Note that the related machines case is a generalization of the identical machines case. However, assigning a task to the machine with minimum load (i.e greedy) results in an algorithm with competitive ratio that is at least the ratio of the fastest to slowest machines speed which maybe unbounded even for two machines. Nevertheless, one may consider the following natural post-greedy algorithm: each task j is assigned upon its arrival to a machine k that minimizes the resulting load *i.e.*, a machine k that minimizes $\ell_k(j-1)+p_k(j)$. It is easy to see that for the identical machines case greedy and post-greedy algorithms are the same. The lower bound of the next theorem was proved in [20] and the upper bound was proved in [4].

Theorem 2. **[20, 4]** *The post-greedy algorithm has a competitive ratio $\Theta(\log n)$ for related machines.*

A new algorithm is required to achieve constant competitive algorithm for the related machine case. We will use the doubling technique and thus may assume a given parameter Λ, such that $\Lambda \geq OPT$. Roughly speaking, algorithm ASSIGN-R will assign tasks to the slowest machine possible while making sure that the maximum load will not exceed twice Λ. More specifically, we assume that the machines are indexed according to increasing speed. The algorithm assign a task to the machine i of minimum index such that $\ell_i(j-1)+p_i(j) \leq 2\Lambda$ and returns fails if such an index does not exists. The above algorithm may be viewed as an adaptation of the scheduling algorithm [37] to the context of load balancing.

Theorem 3. [4] *If $OPT \leq \Lambda$, then algorithm* ASSIGN-R *never fails. Thus, the load on a machine never exceeds 2Λ. If OPT is unknown in advance the doubling technique for Λ implies that* ASSIGN-R *is 8 competitive.*

Using randomized doubling it is possible to replace the deterministic 8 upper bound by $2e \approx 5.436$ expected value [27]. Recently it was shown by [16] that replacing the doubling by a more refined method improves the deterministic competitive ratio to $3+\sqrt{8} \approx 5.828$ and the randomized variant to about 4.311. Also the lower bound is 2.438 for deterministic algorithms and 1.837 for randomized ones.

Open problem 3.2 *Determine the competitive ratio for load balancing of permanent tasks on related machines.*

3.3 Restricted assignment

Each task has a weight and can be assigned to one of a subset of admissible machines (which may depend on the task). This case was considered in [11], who described an optimal (up to an additive one) competitive algorithm for permanent tasks. The algorithm AW is just a natural greedy that assign a task to a machine with minimum load among the admissible machines breaking ties arbitrarily.

Theorem 4. [11] *Algorithm AW achieves a competitive ratio of $\lceil \log_2 n \rceil + 1$. The competitive ratio of any on-line assignment algorithm is at least $\lceil \log_2 n \rceil$.*

It is interesting to realize that randomized algorithms may improve the performance but only by a constant factor. More specifically:

Theorem 5. [11] *The competitive ratio of any randomized on-line assignment algorithm is at least $\ln(n)$*

Using the on-line randomized matching algorithm of [29] the lower bound can be matched for the unit tasks case by an algorithm AR defined as follows. Choose a sequence of random permutations $\pi(k)$ of 1 to n for $k \geq 1$. Assign tasks greedy. If the minimum load among the admissible machines for a given task is k then breaks ties by priorities given by $\pi(k)$. The result is summarized in the next theorem.

Theorem 6. [11] *For unit size tasks the (expected) competitive ratio of Algorithm AR is at most $\ln(n) + 1$ assuming that the optimal load is 1.*

Open problem 3.3 *Design a randomized algorithm for arbitrary sized tasks for the permanent restricted assignment case that achieves $\ln(n) + O(1)$ competitive ratio or show an appropriate lower bound.*

3.4 Unrelated machines

The *general* unrelated machines case for permanent tasks was considered in [4] who described an $O(\log n)$-competitive algorithm.

We note that natural greedy approaches are far from optimal for this case. More specifically consider the post greedy algorithm in which a task is assigned to a machine that minimizes the resulting load or an algorithm in which a task is assigned to a machine whose increase in load is minimum.

Lemma 7. [4] *The competitive ratios of the above greedy algorithms are exactly n for the unrelated machines case.*

For describing the $O(\log n)$-competitive algorithm we first consider the case where we are given a parameter Λ, such that $\Lambda \geq OPT$. As before, an appropriate value of Λ can be "guessed" using a simple doubling approach, increasing the competitive ratio by at most a factor of 4. We use tilde to denote normalization by Λ, *i.e.* $\tilde{x} = x/\Lambda$.

We use again the notion of *designed performance guarantee.* Let $1 < a < 2$ be any constant and β the designed performance guarantee. The basic step of the algorithm called ASSIGN-U is to assign task j to a machine s such that after the assignment $\sum_{i=1}^{n} a^{\tilde{\ell}_i(j)}$ is as small as possible. More precisely, we compute

$$\text{Increase}_i(j) = a^{\tilde{\ell}_i(j-1)+\tilde{p}_i(j)} - a^{\tilde{\ell}_i(j-1)}$$

and assign the task to a machine s with minimum increase unless $\ell_s(j-1) + p_s(j) > \beta\Lambda$ which results in returning "fail".

Theorem 8. [4] *There exists $\beta = O(\log n)$ such that if $OPT \leq \Lambda$ then algorithm* ASSIGN-U *never fails. Thus, the load on a machine never exceeds $\beta\Lambda$.*

The lower bound for the restricted assignment case implies that the algorithm is optimal (up to a constant factor). Also it is interesting to note that for the restricted assignment case the algorithm becomes the AW greedy algorithm described in the previous subsection.

3.5 Virtual circuit routing

Surprisingly, the algorithm for the unrelated machines case can be extendible to the more complex case of virtual circuit routing. We are given a (directed or undirected) graph and requests for virtual paths arrive on-line. The jth request is $(s_j, t_j, r(j))$ where s_j, t_j are source and destination points and $r(j)$ is a required bandwidth. If the path assigned to request j uses an edge e then the load $\ell(e)$ is increased by $p_e(j) = r(j)/c(e)$ where $c(e)$ is the capacity of the edge. The goal is to minimize the maximum load over the edges.

As usual we assume that we are given a parameter $\Lambda \geq OPT$. The algorithm assigns a route such that $\sum_{e \in E} a^{\tilde{\ell}_e(j)}$ is as small as possible. More precisely, we compute

$$\text{Increase}_e(j) = a^{\tilde{\ell}_e(j-1)+\tilde{p}_e(j)} - a^{\tilde{\ell}_e(j-1)}$$

and assign the request to the shortest path from s_j to t_j unless some load exceeds $\beta\Lambda$ which results in returning "fail".

Theorem 9. [4] *If $OPT \leq \Lambda$, then there exists $\beta = O(\log n)$ such that the routing algorithm never fails. Thus, the load on a link never exceeds $\beta\Lambda$.*

It is possible to translate the $\Omega(\log n)$ lower bound for restricted assignment to the virtual circuit routing problem on directed graphs. Recently [15] showed that the lower bounds hold also for undirected graphs.

	Competitive ratio
Identical	$2-\epsilon$
Related	$\Theta(1)$
Restricted	$\Theta(\log n)$
Unrelated	$\Theta(\log n)$
Routing	$\Theta(\log n)$

Fig. 1. Summary of competitive ratio for permanent tasks

4 Temporary tasks unknown duration

Tasks that arrive and may also depart are called temporary tasks. Recall that permanent tasks is a special case of temporary tasks. We refer to the case that the duration of a task is unknown at its arrival (in fact until it actually departs) as unknown duration.

4.1 Identical machines

It turns out that the analysis of Graham [23, 24] of the greedy algorithm for permanent tasks also holds for temporary tasks. It is shown in [9] that no algorithm can achieve a better competitive ratio. Thus, the optimal algorithm is greedy which is $(2-\frac{1}{n})$-competitive. Recall that for permanent tasks the competitive ratio is below 1.923 which implies that the temporary tasks case is a strictly harder than the permanent tasks case. However, for $n=2,3$ the competitive ratio is the same as for permanent tasks. It turns out that randomization does not help much. Specifically, randomized algorithms cannot achieve a competitive ratio which is better than $2-2/(n+1)$ [9]. If the sequence is limited to a polynomial size in n then the lower bound is $2-O(\log\log n/\log n)=2-o(1)$. Also, the tight lower bound for randomized algorithms for $n=2$ is $3/2$. This is contrast to the $4/3$ randomized competitive ratio for permanent tasks. The above facts separate between permanent to temporary tasks also for randomized algorithms.

As we have seen randomization cannot help much for temporary tasks and the obvious question is whether it can help at all here.

Open problem 4.1 *Can we get below the $2-1/n$ using randomized algorithms for temporary tasks of unknown durations on identical machines ?*

4.2 Related machines

This case was considered in [10] who showed a 20 competitive algorithm for temporary tasks. Recall that for permanent tasks there is a 5.828 (improvement over the 8) competitive algorithm.

We use again the notion of *designed performance guarantee* and give an algorithm SLOW-FIT which guarantees a load of 5Λ given a parameter $\Lambda \geq OPT$. By the simple doubling approach this results in a 20-competitive algorithm. Assume that the machines are indexed according to increasing speed. For a task j that arrives at time t we say j is *assignable* to machine i if $w(j)/v_i \leq \Lambda$ and $\ell_i(t-1) + w(j)/v_i \leq c \cdot \Lambda$. SLOW-FIT assign task j to the *assignable* machine of minimum index.

Theorem 10. **[10]** *Provided $c \geq 5$ and $\Lambda \geq OPT$, the* SLOW-FIT *guarantees that every task is assignable. Thus, if OPT is unknown in advance the doubling technique for Λ implies that* SLOW-FIT *algorithm is 20-competitive.*

The best lower bound is the following:

Theorem 11. **[10]** *The competitive factor c of any on-line algorithm for the related machines case satisfies $c \geq 3 - o(1)$.*

Note that the lower bound is proved even when the value of OPT is known to the algorithm (the upper bound is 5 if OPT is known).

Open problem 4.2 *Determine the competitive ratio for load balancing of tasks with unknown durations on related machines.*

4.3 Restricted assignment

Recall that in the restricted assignment case for permanent tasks the competitive ratio of the greedy algorithm is at most $\log n + 2$ and that no algorithm can do better (up to an additive one).

In contrast for the case of temporary tasks with *unknown duration* it is shown in [8, 10] that there is an algorithm with competitive ratio $\Theta(\sqrt{n})$ and that no algorithm can do better. More precisely, the following theorems are proved:

Theorem 12. **[8]** *The competitive ratio of the greedy on-line assignment algorithm is exactly $\frac{(3n)^{2/3}}{2}(1 + o(1))$.*

Theorem 13. **[8]** *The competitive ratio of any deterministic on-line assignment algorithm is at least $\lfloor\sqrt{2n}\rfloor$. For randomized algorithms the lower bound is $\Omega(n^{1/2})$.*

The lower bound is proved using exponential size sequence of requests. It is shown in [33] that the lower bound can also be achieved (up to some constant factor) even on polynomial length sequence.

Next we describe an $O(\sqrt{n})$ competitive algorithm called Robin-Hood. Again we first design an algorithm for the case that we are given a parameter $\Lambda \geq OPT$. A machine g is said to be *rich* at some point in time t if $\ell_g(t) \geq \sqrt{n}\Lambda$, and is said to be *poor* otherwise. A machine may alternate between being rich and poor over time.

If g is rich at t, its *windfall time* at t is the last moment in time it became rich. More precisely, g has windfall time t_0 at t if g is poor at time $t_0 - 1$, and is rich for all times t' $t_0 \leq t' \leq t$.

Algorithm Robin-Hood assigns a new task to some poor machine if possible. Otherwise, it assigns to the machine with the most recent windfall time.

Theorem 14. [10] *The competitive ratio of Algorithm* Robin-Hood *is at most* $2\sqrt{n}+1$.

We can apply doubling to overcome the problem that OPT is unknown in advance. This would result in increasing the competitive ratio by a factor of 4. However, it turns out that we do not need to lose this factor of 4. We do so by maintaining an estimate $L(t)$ and use it instead of Λ. Instead of doubling $L(t)$ it is updated after the arrival of a new task j at time t by setting

$$L(t) = \max\{L(t-1), w_j, \frac{1}{n}(w_j + \sum_g \ell_g(t-1))\} .$$

4.4 Unrelated machines

The only facts that are known for tasks of unknown duration on unrelated machines is that the competitive ratio is at least $\Omega(\sqrt{n})$ (by the restricted assignment lower bound) and at most n (many versions of greedy). The main open problem here is the following:

Open problem 4.3 *Determine the competitive ratio for load balancing of tasks with unknown durations on unrelated machines.*

5 Known duration

It is not known if knowing the durations of tasks help in the identical and related machines cases. Recall that for identical machines, if the duration are not known then the deterministic and randomized competitive ratios are $2-o(1)$. An interesting open problem is the following:

Open problem 5.1 *Can we get a competitive ratio which is below 2 for identical machines knowing the duration (deterministic or randomized) ?*

Knowing the durations certainly helps in the restricted assignment and unrelated machines cases. Denote by T is the ratio of the maximum to minimum duration (the minimum possible task duration is known in advance). Recall the $\Omega(\sqrt{n})$ lower bound on the competitive ratio for the restricted assignment case when the duration of a task is not known upon its arrival [8]. In contrast if the duration is known we have the following:

Theorem 15. [10] *There is an online load balancing algorithm for unrelated machines with known tasks duration which is* $O(\log nT)$*-competitive.*

It is unclear if the $\log T$ is really necessary when the durations are known. Of course, we can ignore the durations and get $O(\sqrt{n})$ competitive algorithm for the restricted assignment (which is better for huge T). The obvious question is whether we can do better.

Open problem 5.2 *Can we get below the* $\Theta(\sqrt{n})$ *bound for restricted assignment assuming known durations and can we prove lower bounds ?*

Open problem 5.3 *Can we get below the* $\Theta(n)$ *for unrelated machine case knowing the durations and can we prove lower bounds ?*

	Unknown durations	**Known durations**	**Permanent**
Identical	$2 - o(1)$	?	$2 - \epsilon$
Related	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Restricted	$\Theta(n^{1/2})$	$O(\log nT)$	$\Theta(\log n)$
Unrelated	?	$O(\log nT)$	$\Theta(\log n)$
Routing	?	$O(\log nT)$	$\Theta(\log n)$

Fig. 2. Summary of competitive ratio for the various models

6 Reassignments

In order to overcome the above large lower bounds for load balancing of tasks with unknown durations, one may allow *task reassignments.*

6.1 Restricted assignment

For the unit size task i.e. all coordinates of the load vector are either ∞ or 1, [32] presented an algorithm that achieves $O(\log n)$ competitive ratio with respect to load while making at most a constant amortized number of reassignments per task. Their algorithm belongs to the class of algorithm that does not perform reassignments unless tasks departs. Hence the $\Omega(\log n)$ lower bound for permanent tasks holds for this type of algorithms.

Later, [7] considered the unit size task case. They gave an algorithm that achieves a *constant* competitive ratio while making $O(\log n)$ amortized reassignments per task. However, they required that the optimum load achieved by the off-line algorithm is at least $\log n$. The algorithm has been extended in [38] and that assumption has been removed.

We first describe the algorithm that maintain constant competitive ratio and $O(\log n)$ reassignment per unit size task with the assumption that $OPT \geq \log n$.

As usual we assume that the algorithm has a knowledge of $\Lambda \geq OPT$. The algorithm will maintain the following *stability condition*:

Definition 16. Let j be some task which is currently assigned to machine i. Consider a machine i' which is an eligible assignment for task j (*i.e.* machine i with $p_{i'j} = 1$). We say that the algorithm is in a *stable state* if for any such i and i', we have:

$$\ell_i - \ell_{i'} \leq 2\Lambda/\log n$$

The main idea of the algorithm is to make sure that the above stability condition is satisfied. More precisely the algorithm is described as follows: A new task j is assigned to any eligible machine and when a task departs, it is removed from the machine on which it is currently assigned. If at any moment the stability condition is not satisfied by some task j that is currently assigned to machine i, the algorithm reassigns j to a least loaded machine among the machines that are eligible with respect to j.

Theorem 17. [7] *The assignment algorithm maintains load of at most* 4Λ *with* $O(\log n)$ *reassignments per arrival or departure of a task assuming* $\Lambda \geq OPT$.

As before we eliminate the need to know the optimal load in advance by the doubling technique. This increases the competitive ratio by at most a factor of 4 to be 16.

Observe that, as opposed to the previous algorithm, this algorithm reassigns tasks both as a result of task arrival and departure. As mentioned this is necessary to achieve constant competitive ratio, since the lower bound of [11] implies that an algorithm that does not reassign tasks as a result of task arrivals can not achieve better than $\Omega(\log n)$ competitive ratio.

Next, we describe how to get rid of the assumption that $OPT \geq \log n$ [38]. Regard the problem as a game on dynamic bipartite graph. On one side the machines V and on the other side the tasks U. An edge (u, v) indicate that u can be assigned to v. Edge (u, v) is *matching* if u is assigned to v.

Let $\ell(v)$ denote the load on $v \in V$, i.e., the number of matching edges incident on v. A balancing path is an even-length path sequence of alternating matched and unmatched edges $(v_1, u_1), (u_1, v_2), \ldots, (u_{m-1}, v_m)$ with the property that $\ell(v_i) < \ell(v_1)$ for $1 \leq i \leq m-1$ and $\ell(v_m) < \ell(v_1) - OPT$. 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 machines are r-balanced if there is no balancing path of length r or less.

The algorithm is described as follows: A new task j is assigned to any eligible machine and when a task departs, it is removed from the machine on which it is currently assigned. If at any moment there is a balancing path of length r or less then re-balance using this path. For $r = \log n$ we have

Theorem 18. [38] *The assignment algorithm above is constant competitive with $O(\log n)$ reassignments per arrival or departure of a task.*

Open problem 6.1 *Is it possible to achieve constant ratio and constant number of reassignments per task.*

6.2 Unrelated machines

The general case *i.e.* load balancing of unknown duration tasks with no restrictions on the load vectors was considered in [7]. They designed a new algorithm that makes $O(\log n)$ reassignments per task and achieves $O(\log n)$ competitive ratio with respect to the load.

We first assume that the algorithm has a knowledge of $\Lambda \geq OPT$. Let $1 < a < 2$ be a constant. At every instance t, each active task j is assigned to some machine i. Define the *height* $h_i^j(t)$ of task j that is assigned to machine i at time t as follows. It is the sum of $p_i(j)$ for all tasks j' that are currently assigned to i and were last reassigned to machine i before j was last assigned to i. The weight of task j is

$$W_i^j(t) = a^{\tilde{h}_i^j(t)+\tilde{p}_i(j)} - a^{\tilde{h}_i^j(t)}$$

Note that the weight of task j immediately after it is assigned to machine i is:

$$a^{\tilde{\ell}_i(t)} - a^{\tilde{\ell}_i(t)-\tilde{p}_i(j)}$$

where t is the time immediately after the assignment.

From now on we will omit the parameter t. The algorithm maintains the following *stability condition*:

Definition 19. We say that the algorithm is in a *stable state* if for any machine i' we have:

$$W_i^j = a^{\tilde{h}_i^j+\tilde{p}_i(j)} - a^{\tilde{h}_i^j} \leq 2\left(a^{\tilde{\ell}_{i'}+\tilde{p}_{i'}(j)} - a^{\tilde{\ell}_{i'}}\right) .$$

Intuitively, the main idea of the algorithm is to make sure that the above stability condition is satisfied. More precisely the algorithm is described as follows:

A task is assigned upon its arrival to a machine i which minimizes W_i^j. When a task departs it is removed from the machine on which it is currently assigned. If at any moment the stability condition is not satisfied by some task j that is currently assigned to some machine i, the algorithm reassigns j on machine i' that minimizes $W_{i'}^j$.

Observe that the algorithm will never reassign as a result of an arrival of a new task. The stability condition is strong enough to maintain the competitive ratio and weak enough to cause many reassignments.

Theorem 20. [7] *For the unrelated machines problem where the duration of tasks is a-priori unknown, the above assignment algorithm makes $O(\log n)$ reassignments per task and achieves $O(\log n)$ competitive ratio with respect to the load.*

Recall that for restricted assignment (and therefore for unrelated machines) an $\Omega(\log n)$ lower bound was proved on the competitive ratio for the load balancing case where tasks never depart. Observe that the algorithm reassigns tasks only as a result of task departures, and hence can not achieve better than $O(\log n)$ competitive ratio with respect to load.

A natural extension of the algorithm also works for the virtual circuit routing problems [7]. By making $O(\log n)$ reroutings per path achieves $O(\log n)$ competitive ratio with respect to the load.

6.3 Current load

We conclude this subsection by an alternative definition of competitive ratio which requires reassignments to get reasonable results. In the standard definition compare the maximum on-line load to the the maximum off-line load. It was suggested in [38] to compare the current load against the current off-line load. It is easy to see that for permanent tasks the standard definition and the new definition are the same (since the sequence may stop at any time and the on-line and off-line loads are monotonically non-decreasing). However, for temporary tasks it is immediate to show that if no reroutings are allowed then the lower bound is n even on identical machines. Specifically, n^2 unit tasks appear, after which some machine k must have load at least n. Then, all tasks depart except for those on k. Thus, the current on-line load is n while the current optimal off-line load is 1. Thus, one must allow reassignments to achieve significant results for this model. Algorithms for this purpose appear in [32, 38, 2, 3].

7 L_p norm

In all the previous sections we evaluated the performance of the algorithm by the maximum load. In section we consider the L_p norm measure ($p \geq 1$).

Recall that $\ell_k(t)$ denotes the load on machine k at time t and $\ell^*_k(t)$ denotes the load on machine k at time t of the optimal off-line algorithm.

For a given vector $X = (x_1, x_2, \ldots, x_n)$ the L_p norm and L_∞ norm of X are

$$|X|_p = \left(\sum_{1 \leq i \leq n} |x_i|^p \right)^{1/p} \quad \text{and} \quad |X|_\infty = \max_{1 \leq i \leq n} \{|x_i|\}.$$

The L_2 norm is the Euclidean norm, which measures the length of the vector X in Euclidean space. The maximum load measure of an algorithm is the maximum over time of $|\ell(t)|_\infty$. The L_p norm measure for an algorithm A on a sequence

σ denote by $A(\sigma)$ is the maximum over time of $|\ell(t)|_p$. The performance of an algorithm A is the supremum over all sequences of $A(\sigma)/OPT(\sigma)$.

We first consider permanent tasks. It is not hard to show that for identical machines the greedy algorithm, (i.e assign to the minimum loaded machine)is 2 competitive. In fact, the competitive ratio of greedy is determined in [5]. In particular, for the L_2 norm it is $\sqrt{4/3}$ and no algorithm can achieve a better competitive ratio. Surprisingly, the asymptotic competitive ratio is below $\sqrt{4/3}-\epsilon$.

Next we consider unrelated machine. A natural post greedy type algorithm for minimizing the L_p norm is to assign a task on a machine to minimize $\sum_i \ell_i^p(j)$. More precisely, when task j arrives we compute weights to the machines,

$$\text{Increase}_i(j) = (\ell_i(j-1) + r_i(j))^p - \ell_i^p(j-1)$$

and assign the task to a machine with minimum increase. Note that for the identical machines and the restricted assignment cases the algorithm is equivalent to the greedy that assigns a task to a least loaded machine.

Theorem 21. **[6]** *The above algorithm is* $1+\sqrt{2}$ *competitive with respect to the* L_2 *norm.*

Theorem 22. **[6]** *For any constant* $p \geq 1$ *the above load balancing algorithm is* $O(p)$*-competitive in the* L_p *norm. Moreover, any deterministic algorithm must be* $\Omega(p)$*-competitive even for the restricted assignment case.*

Theorem 23. **[17]** *For the restricted assignment case with unit jobs the greedy algorithm is approximately* 2.01 *competitive with respect to the* L_2 *norm.*

Open problem 7.1 *Design an algorithm for related machine case (permanent tasks) whose competitive ratio in the* L_p *norm is constant (independent of* p*).*

It is not quite clear how to define the performance measure for temporary tasks. One possible definition is the maximum over the duration of the L_p norm of the load vector. For the case of known duration one may use a different definition which is the L_p norm of the nT vector of the n machines over the sequence of total length T. For this definition one can achieve a competitive ratio of $O(p)$ (known durations). Not much is known for the unknown duration case.

Open problem 7.2 *Determine the competitive ratio in the* L_p *norm for tasks with unknown duration for related machines, unrelated machines and for restricted assignment with and without reassignments.*

References

1. S. Albers. Better bounds for on-line scheduling. In *Proc. 29th ACM Symp. on Theory of Computing*, pages 130–139, 1997.
2. M. Andrews. Constant factor bounds for on-line load balancing on related machines. Manuscript.

3. M. Andrews, M. Goemans, and L. Zhang. Improved bounds for on-line load balancing. In *COCOON96*, 1996.
4. 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 Symposium on the Theory of Computing*, pages 623–631, 1993.
5. A. Avidor, Y. Azar, and J. Sgall. Ancient and new algorithms for load balancing in the l_p norm. In *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, 1998. To appear.
6. B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter. Load balancing in the l_p norm. In *Proc. 36th IEEE Symp. on Found. of Comp. Science*, pages 383–391, 1995.
7. B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 321–327, 1994.
8. Y. Azar, A. Broder, and A. Karlin. On-line load balancing. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 218–225, 1992. Also in *Theoretical Compute Science* 130 (1994) pp. 73-84.
9. Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. In *5th Israeli Symposium on Theory of Computing and Systems*, pages 119–125, 1997.
10. Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. Online load balancing of temporary tasks. In *Workshop on Algorithms and Data Structures*, pages 119–130, 1993.
11. Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.
12. Y. Azar and O. Regev. On-line bin stretching. Manuscript.
13. Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24th ACM Symposium on Theory of Algorithms*, pages 51–58, 1992. To appear in *Journal of Computer and System Sciences.*
14. Y. Bartal, H. Karloff, and Y. Rabani. A better lower bound for on-line scheduling. *Information Processing Letters*, 50:113–116, 1994.
15. Y. Bartal and S. Leonardi. On-line routing in all-optical networks. In *Proc. 24rd International Colloquium on Automata, Languages, and Programming*, 1997.
16. P. Berman, M. Charikar, and M. Karpinski. A note on on-line load balancing for related machines. In *5th annual Workshop on Algorithms and Data Structures*, 1997.
17. R. Boppana and A. Floratos. Load balancing in the euclidean norm. Manuscript, 1997.
18. B. Chen, A. van Vliet, and G. J. Woeginger. Lower bounds for randomized online scheduling. *Information Processing Letters*, 51:219–222, 1994.
19. B. Chen, A. van Vliet, and G. J. Woeginger. New lower and upper bounds for on-line scheduling. *Operations Research Letters*, 16:221–230, 1994.
20. Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9:91–103, 1988.
21. A. Fiat and S. Leonardi. On-line routing. this volume, 1997.
22. G. Galambos and G. J. Woeginger. An on-line scheduling heuristic with better worst case ratio than graham's list scheduling. *SIAM J. Computing*, 22:349–355, 1993.
23. R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

24. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math*, 17:263–269, 1969.
25. D. Hochbaum and D. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17, 1988.
26. Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *J. of the ACM*, 34(1):144–162, January 1987.
27. P. Indyk. Personal communication.
28. D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 132–140, 1994.
29. R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, Baltimore, Maryland, May 1990.
30. H. Kellerera, V. Kotov, M. G. Speranza, and Zs. Tuza. Semi on-line algorithms for the partition problem. *Operations Research Letters.* To appear.
31. J.K. Lenstra, D.B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Prog.*, 46:259–271, 1990.
32. S. Phillips and J. Westbrook. On-line load balancing and network flow. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 402–411, 1993.
33. S. Plotkin and Y. Ma. An improved lower bound for load balancing of tasks with unknown duration. Manuscript.
34. S. Seiden. Randomized algorithms for that ancient scheduling problem. In *5th annual Workshop on Algorithms and Data Structures*, 1997.
35. J. Sgall. On randomized on-line multiprocessor scheduling. To appear in Information Processing Letters.
36. J. Sgall. On-line scheduling. this volume, 1997.
37. David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. In Lyle A. McGeoch and Daniel D. Sleator, editors, *On-line Algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–166. AMS/ACM, February 1991.
38. J. Westbrook. Load balancing for response time. In *3rd Annual European Symposium on Algorithms*, 1995.