Offline file assignments for online load balancing

Paul Dütting a, Monika Henzinger b, Ingmar Weber c,∗

a Ecole Polytechnique Fédérale de Lausanne (EPFL), Station 14, CH-1015 Lausanne, Switzerland
b University of Vienna, Faculty of Computer Science, Universitätsstraße 10, A-1090 Wien, Austria
c Yahoo! Research Barcelona, Av. Diagonal 177, E-08003 Barcelona, Spain

1. Introduction

In the online load balancing with restricted assignment problem a set of m machines has to execute a sequence of requests, arriving one by one. Every request has a load and must be placed on exactly one of a subset of machines. The assignment of a request to a machine increases the load on this machine by the load of the request. The goal is to minimize the makespan, i.e., the maximum machine load placed on any machine. When neither the load of each request nor the set of machines a request can be assigned to are under the control of the algorithm, then no online load balancing algorithm can achieve a competitive ratio better than \(\lceil \log_2 m \rceil\), see [1].

In this paper we study the following variant of this problem which consists of two phases. In the offline phase, n files need to be assigned to m identical machines with the possibility, given space, to copy some or all of the files. In the online phase, a sequence of requests arrives. Each request \(t\) asks for one file \(f_j\) and has to be placed on one of the machines \(m_i\) to which (a copy of) this file was assigned. That machine’s load \(ML_i\) is then increased by \(l(t)\). \(FL_j\) denotes the sum of the loads of all the requests for file \(f_j\). The goal is still to minimize the makespan, i.e., the maximum machine load \(ML_\ast = \max_i ML_i\).

In this model the position of the algorithm is strengthened: by placing the files “intelligently” it can influence the set of machines a request can be assigned to. However, it still does not control the load of each request nor the file the request asks for. Also note that the makespan will generally depend on the number of file copies made by the algorithm. In our model, each machine has \(s\) “slots” which can be used to store (copies of) files. We require \(s \geq \lceil n/m \rceil\) because if \(s < \lceil n/m \rceil\) it is impossible to store all of the \(n\) files on the \(m\) machines.

We call an algorithm for both the offline and the online phase a dual-phase algorithm. To analyze the quality of a dual-phase algorithm there are two sensible points of reference. First, one could compare \(ML_\ast\) to the makespan \(OPT\) of the optimal offline1 algorithm for the same parameter \(s\). Such an analysis emphasizes the optimality gap while caring less about how good the optimal solution

1 By an “offline” algorithm we mean an algorithm which is given the sequence of requests \(t\) before assigning files to machines.
actual is. Second, one could compare ML\textsuperscript{α} to the average makespan \( \text{AVG} := \frac{1}{n} \sum_{t \in [1,n]} t_l \) which corresponds to a perfect load balance. In this approach the emphasis is less on the optimality gap, as even the optimal offline algorithm might not be able to achieve AVG.

To see this, consider the following two examples for the case \( s = \lceil n/m \rceil\): (1) For \( n = 2 \) and \( m = 3 \) \( \text{OPT}_{\lceil n/m \rceil} \) can only copy one of the two files and cannot achieve a makespan of less than \( L := \min_{j} FL_j \), compared to \( \text{AVG} = 2L/3 \) for equal file loads. (2) For \( n = 5 \) and \( m = 3 \) \( \text{OPT}_{\lceil n/m \rceil} \) can again only copy one of the files and one machine has only unshared\footnote{A file \( f_j \) is called unshared if it only resides on a single machine \( m_i \).} files leading to a makespan of at least \( 2L \), compared to \( \text{AVG} = 5L/3 \) for equal file loads.

Of course, for any \( s \) we always have \( \text{OPT}\_{s} \geq \text{AVG} \) and so competitive ratios with respect to AVG lead to competitive ratios with respect to \( \text{OPT}\_{s} \). However, as far as lower bounds are concerned we still keep AVG as a reference point as we ultimately care more about the quality of the solution (\( \approx \) “How far from perfect load balance is it?”) than about the quality of the algorithm (\( \approx \) “How far from the optimal solution for the given \( s \) is it?”).

Even with the sequence of requests given in advance it is tricky to find an optimal assignment as (i) the loads for the different files can differ widely, leading to a “packing” problem to level out these differences, and (ii) individual requests can be large, making it difficult to spread one file’s load across several machines. To factor out these two difficulties, we introduce the following parameters:

- We define \( \alpha \) to be the smallest value such that \( \forall j, k: FL_j \leq (1+\alpha)FL_k \), or equivalently, such that \( \forall j: L \leq FL_j \leq (1+\alpha)L \) where \( L := \min_{j} FL_j \).
- We define \( \beta := \max_{i} L(t) \) to be the maximum request load.

As a consequence we state the bounds on the makespan in terms of \( n, m, \alpha, \beta \). To summarize, the differences between our new model and the previous models are: (1) The online phase is preceded by an offline phase, where a file assignment is computed. This assignment determines where requests can be placed in the online phase. (2) The loads are parametrized. Individual requests have load at most \( \beta \) and the load of all requests for two distinct files differs by a factor of at most \( 1+\alpha \).

The structure of the remaining paper is as follows. Section 1.1 discusses an application of the studied problem and Section 1.2 briefly discusses results concerning “classical” load balancing. In Section 2, we present some observations related to the inherent difficulty of the problem. The following three sections, Sections 3–5, discuss algorithms for different values of \( s \), going from the minimal \( s = \lceil n/m \rceil \) to a tunable value. Finally, in Section 6 we present two lower bounds for deterministic dual-phase algorithms.

1.1. Application to web search engines

One application for our problem arises in web search engines. Every web search engine builds a data structure, called (inverted) index \([2,3]\). In current web search engines the size of this data structure is in the order of hundreds of TeraBytes. The index is broken into a large number \( n \) of equally-sized files \( f_j \) which are distributed over multiple machines. This split is usually done using a document-based partitioning where each file contains the information for a subset of documents \([3]\). With this partitioning, each query has to access all the files. Thus one query is broken into \( n \) requests, each request having to access one specific file. The time it takes to answer the request for a given file depends on the query terms in the request, but also on the file itself as some files might contain many matching documents. However, search engines usually time-out requests that cannot be answered in a fixed time limit, e.g. a second. Thus, the maximum load \( \beta \) of an individual request for file \( f_j \) is much smaller than the total file load \( FL_j \) of \( f_j \). In addition, when a search engine splits the index into files it tries to build the files in a way so that over a long enough sequence of requests the different file loads differ only by a small constant factor \( 1 + \alpha \) with \( \alpha < 1 \).

We also computed \( \alpha \) experimentally using the 05/2007 crawl of the Stanford WebBase project with about 20 million pages. From this crawl we constructed an index for Lucene\footnote{http://lucene.apache.org/}. of about 500 GB and divided it into 20, 40 and 80 equally-sized indices according to a random document-based partitioning as described above. Afterwards, we ran two query logs, one from AllTheWeb and one from AltaVista, with 0.5 million queries against each of the indices using a Quad-Core AMD Sempron 280 with 2.4 GHz and 8 GB RAM. In average it took about 1230 s, 689 s, and 419 s to run all queries of one query log against a single index. This time varied by at most a factor of 1.08, 1.12, and 1.15 between two distinct indices and individual queries took at most 12 s, 7 s, and 5 s. Thus, \( \alpha \) was about 0.08, 0.12, and 0.15 while \( \beta \) was about 1% of the average file load.

1.2. Related work

There is a large body of work on online load balancing. See \([4–6]\) for excellent surveys. For space reasons we mention here only the results that are directly related to our problem. If the assignment of each request is unrestricted, i.e., each request can be assigned to every machine, then there exist constant upper and lower bounds. The earliest result is due to Graham \([7,8]\) who showed that the Greedy algorithm (which always places the request on the least loaded machine) has a competitive ratio of exactly \( 2 - \frac{\alpha}{\beta} \). The best algorithm known today is 1.92-competitive \([9]\). The lower bound currently stands at 1.88 \([10]\). If the assignment of each request is restricted to a subset of the machines, then the Greedy algorithm achieves a competitive ratio of \( \lceil \log_2 m \rceil + 1 \) and no online load balancing algorithm can be better than \( \lceil \log_2 m \rceil \)-competitive \([1]\).
2. Preliminaries

Generally, the reason that an online load balancing algorithm cannot achieve AVG is two-fold. The first reason is an “integrality gap” which rules out the possibility of leveling out the discrete loads across a small set of slots. This problem also exists for OPTs (see previous examples). However, assuming that β is small enough, this integrality gap disappears already for OPT\[n/m\] (see the offline result of Theorem 2) and it always vanishes asymptotically as n and m (and n/m) become large as then β/AVG → 0.

The second reason is a “knowledge gap” which summarizes the fact that an online algorithm will not know in advance which files are big (load equal to \((1 + \alpha)L\)) and which are small (load equal to \(L\)). Therefore it could end up with all big file loads on one machine and that all of these files are unshared. This problem can be avoided by OPT which knows the whole input sequence before distributing the files. This problem also vanishes for a randomized algorithm with an oblivious adversary, see Theorem 1. As far as upper bounds are concerned, we make the following observation.

**Observation.** Any reasonable two-phase load balancing algorithm achieves

\[
ML_n \leq \left[\frac{n}{m}\right] \cdot (1 + \alpha) \cdot \frac{n}{m} \cdot AVG
\]

\[
\leq \left(1 + \frac{m}{n}\right) \cdot (1 + \alpha) \cdot AVG.
\]

Even with the minimal space per machine of \(s = \left[\frac{n}{m}\right]\) and without making any copies at all we obtain the bound above which corresponds to the case where all files on one machine have the maximal load \((1 + \alpha)\). This observation shows that upper bounds on the makespan of an algorithm are essentially only interesting if (ii) they are of the form \((1 + o(1)) \cdot AVG\) when \(n\) and \(m\) grow large or if (ii) they are of the form \((1 + \alpha) \cdot AVG + O(1)\) without requiring \(n\) and \(m\) to be large.

Our result in Theorem 1 is of the first form, whereas Theorems 2 and 3 give results of the second form.

3. Minimal space: \(s = [n/m]\)

To store all \(n\) files on the \(m\) machines a minimum space of \(s = [n/m]\) per machine is required. When \(m\) does not divide \(n\) then there are still \(m \cdot [n/m] - n\) spare slots which could be used to store copies of the files. If \(n \geq m\) then at least \(2n - m \cdot [n/m]\) files are unshared and when \(m \cdot [n/m] - n < m/2\), i.e., there are less than \(m/2\) free slots, then there will always be one machine without any shared files. If all files on this machine have a load of \((1 + \alpha)L\) while all other files have \(L\), then no deterministic algorithm can be substantially better than \((1 + \alpha)L\), see Theorem 4. However, with an oblivious adversary and a randomized algorithm the case of minimal space can still be solved asymptotically optimally by the following algorithm called “Randomized”.

**Randomized.** Randomized sorts the \(n\) files randomly s.t. each permutation has equal probability. It then distributes the files in a round-robin manner to the machines. Requests are assigned to the unique machine storing the corresponding file.

**Theorem 1.** Let \(k \in \mathbb{N}^+\) be such that \(n/k = m\) and \(k \geq m^2\ln(m)\). Then “Randomized” uses exactly \(s = n/m = k\) slots per machines and assigns requests to machines such that with probability at least \(1 - \frac{1}{m}\)

\[
ML_n \leq \left(1 + \frac{1}{m}\right) \cdot AVG.
\]

The proof of Theorem 1 makes use of the following lemma [11].

**Lemma 1** (Hoeffding’s inequality). Let \(X_1, \ldots, X_k\) be independent random variables such that \(X_i \in [a_i, b_i]\) for \(1 \leq i \leq k\). Let \(S = \sum_i X_i\). Then, for any positive value \(t\),

\[
\Pr(S - E[S] \geq kt) \leq \exp\left(-\frac{2k^2t^2}{\sum_i(b_i - a_i)^2}\right).
\]

**Proof of Theorem 1.** Instead of placing \(k = n/m\) randomly chosen files on each machine, we could conceptually first split the files randomly into \(k\) groups of size \(m\) and then randomly place one file of each group on each machine. Let \(X_{ij}\) be the random variable that represents the file load of the \(j\)-th file (not necessarily \(j\)) that is placed on machine \(m_i\). For a given machine \(m_i\) the \(X_{ij}\) have different distributions for different values of \(j\), depending on the random split into \(k\) groups of size \(m\). However, once the distributions have been fixed (by the random split) \(X_{i1}, \ldots, X_{ik}\) are independent. Note that \(X_{ij} \in [1, (1 + \alpha)L]\) for all \(i\) and \(j\). Let \(S_i = \sum_j X_{ij}\). We choose \(t = \alpha \frac{1}{m} E[S_i]\), i.e., \(kt = \alpha \frac{1}{m} E[S_i]\), and apply Lemma 1 to get

\[
\Pr\left(S_i - E[S_i] \geq \alpha \frac{1}{m} E[S_i]\right) \leq \exp\left(-\frac{2k^2\alpha^2 E[S_i]^2}{(k\alpha L)^2}\right).
\]

Since \(E[S_i] \geq kL\) and \(k \geq m^2\ln(m)\) we get \(\Pr(S_i - E[S_i] \geq \alpha \frac{1}{m} E[S_i]) \leq \exp(-2k^3 \frac{1}{m^2}) \leq \exp(-2k^3 \frac{1}{m^2}).\) Now let \(E_i\) be the event that \(S_i - E[S_i] \geq \alpha \frac{1}{m} E[S_i]\) and let \(E = \bigcup E_i\). We get \(\Pr(E) = \Pr(\bigcup E_i) \leq \sum_i \Pr(E_i) \leq \frac{1}{m}\). Hence \(S_i - E[S_i] \leq \alpha \frac{1}{m} E[S_i]\) for all machines \(m_i\) with probability at least \(1 - \frac{1}{m}\). In that case \(ML_n = \max_i ML_i = \max_i S_i \leq \max_i (1 + \alpha)E[S_i].\) Since \(OPT \geq \frac{1}{m} \sum_i S_i = \frac{1}{m} \sum_i E[S_i]\) and \(E[S_i] = E[S_i]\) for all \(i\) and \(j\), we have that \(OPT \geq E[S_i]\) for all \(i\). It follows that with probability at least \(1 - \frac{1}{m}\)

\[
ML_n \leq (1 - \frac{1}{m})OPT.
\]

If \(m\) does not divide \(n\) then we can introduce up to \(m - 1\) dummy files with a load of \(L\). This artificially increases AVG to \(AVG' \leq AVG + L \leq AVG \cdot (1 + m/n)\) and one obtains the following corollary.

**Corollary 1.** Let \(k \in \mathbb{N}^+\) be such that \([n/k] = m\) and \(k \geq m^2\ln(m)\). Then “Randomized” uses at most \(s = [n/m]\) slots per
machines and assigns requests to machines such that with probability at least 1 − \( \frac{1}{m} \)

\[
ML_u \leq \left( 1 + \frac{1}{m} \alpha \right) \cdot \left( 1 + \frac{1}{m \ln(m)} \right) \cdot AVG.
\]

4. Slightly more space: \( s = \lceil n/m \rceil + 1 \)

Even with only one additional slot per machine the problem already becomes easier and the following “Fractional” algorithm can be applied.

**Fractional.** Fractional associates with each file \( f_j \) a projected load \( PFL_j \) and tries to balance this projected load across all machines. When used as an offline algorithm it sets the projected load \( PFL_j \) of each file \( f_j \) to \( FL_j \). When used as an online algorithm it simply sets \( PFL_j = 1 \) for all files \( f_j \). Afterward it assigns the files and their projected load to the machines such that (1) no machine is assigned more than \( \lceil n/m \rceil + 1 \) files, (2) the projected load of every file is assigned completely, and (3) every machine is assigned the same amount of projected load \( \sum_j PFL_j/m \). If we use \( fr_t(j) \in [0,1] \) to denote the fraction of \( f_j \)’s projected load \( PFL_j \) that is assigned to \( m_i \), then this can be formalized as follows: (i) \( \sum_i fr_t(j) = 1 \) for all files \( f_j \) and (ii) \( \sum_j fr_t(j) \cdot PFL_j = \frac{1}{m} \sum_j PFL_j \) for all machines \( m_i \). In the online phase, Fractional assigns the requests in a way such that the total load \( ML_u(j) \) placed on \( m_i \) by requests for \( f_j \) is roughly equal to \( fr_t(j) \cdot FL_j \).

**Details of the offline phase:** Assign the files in ascending order of \( FL_j \) in a round-robin manner to the machines. At the end, some machines will have \( \lceil \frac{n}{m} \rceil \) files and some will have \( \lfloor \frac{n}{m} \rfloor \) files. Compute the projected load for each file where, initially, the projected load of a machine equals the total load of its assigned files. Split the machines into three groups. First, machines whose assigned projected load equals \( AVG \). Call these machines *closed*. Second, machines whose assigned projected load surpassed \( AVG \). Call these machines *overloaded*. And third, machines whose assigned projected load is smaller than \( AVG \). Call these machines *underloaded*. While there are any overloaded machines pick an arbitrary one \( m_i \). Copy its largest file, i.e. the file that was assigned to it last, to an underloaded machine \( m_{i′} \). Here, pick an underloaded machine \( m_{i′} \) which was previously overloaded itself, if one exists, or otherwise pick an arbitrary one. Then assign just enough of \( m_{i′} \)’s largest file to \( m_{i′} \) such that \( m_{i′} \) reaches \( AVG \) and becomes closed.

**Details of the online phase:** Fractional keeps track of the load \( FL_{j−1} \) of \( f_j \) generated during the first \( t − 1 \) requests and the part \( ML_{j−1} \) of \( FL_{j−1} \) that is placed on \( m_i \). The \( t \)-th request for \( f_j \) is assigned to a machine \( m_i \) storing \( f_j \) s.t. \( (i) \) \( It(t) \leq fr_t(j) \cdot ML_{j−1}^{−1} \) or if a machine exists, else \( (ii) \) \( fr_t(j) \cdot ML_{j−1}^{−1} \) and \( m_i \) stores no other shared files, or else \( (iii) \) \( fr_t(j) – FL_{j−1}^{−1} \) is largest.

**Theorem 2.** “Fractional” uses \( s = \lceil \frac{n}{m} \rceil + 1 \) slots per machine and assigns requests to machines such that

1. \( ML_u \leq AVG + \beta \) in offline mode, and
2. \( ML_u \leq (1 + \alpha) \cdot AVG + \beta \) in online mode.

**Proof.** We will prove the result for the offline mode where \( PFL_j = FL_j \). The result for the online mode then follows easily by considering the worst case where all files on one machine have a load of \( (1 + \alpha) \cdot L \) and all other files have a load of \( L \).

Without loss of generality let us assume that files are already sorted by their load, i.e., \( FL_1 \leq FL_2 \leq \cdots \leq FL_n \). Let \( FL_{v,w} \) be the (complete) load of the \( w \)-th file placed on machine \( m_v \). From the round-robin assignment we know that \( FL_{v,w} = FL_{(w−1)m+ν} \). Let \( n_v \) be the number of files assigned to machine \( m_v \) in the first part of the offline phase, before files are copied. We know that \( |n_1 − n_v| \leq 1 \) and that \( t < i′ \Rightarrow n_i \geq n_v \).

**Claim 1.** At any time during the offline phase any overloaded machine \( m_i \) can “close” any underloaded machine \( m_{i′} \).

By “close” we mean that the remaining assigned load \( fr_t(j) \) of \( m_i \)’s biggest/last file \( f_j \) is bigger than \( AVG - \sum_{w=1}^{ν} FL_{v,w} \). This claim then implies that any machine has at most two shared files as an initially overloaded machine can only become underloaded once. Claim 1 follows from the following claim.

**Claim 2.** At any time during the offline phase the socket \( c_i \) of any underloaded machine \( m_i \) is no lower than the socket \( c_i \) of any overloaded machine \( m_{i′} \).

Here the “socket” of an underloaded machine equals its total assigned file load \( c_i = \sum_j fr_t(j) \cdot FL_j \) and the socket of an overloaded machine is defined as \( c_i = \sum_{w=1}^{ν_1} FL_{v,w} \). This claim then proves Claim 1 as whenever the load of the last file on \( m_i \) was big enough to extend from \( c_i \) past \( AVG \), then this load will certainly suffice to extend from \( c_i \) to \( AVG \) to close this machine.

**Proof of Claim 2.** We will first show that the claim holds initially before any overloaded machine becomes underloaded. Until this happens we always have \( c_i = \sum_{v=1}^{ν_1} FL_{v,w} \) for all underloaded machines \( m_{i′} \) and the relevant \( fr_t(j) \) are all 0 or 1. If \( n_i = n_{i′} \) and \( m_i \) is overloaded, then we must have \( i′ < i \) and \( c_i = \sum_{w=1}^{ν_1} FL_{v,w} > \sum_{w=1}^{ν_1} FL_{v,w} > \sum_{w=1}^{ν_1} FL_{v,w} = c_{i′} \), using \( FL_{v,w+1} \geq FL_{v,w} \). If \( n_i = n_{i′} + 1 \), then we have \( c_i = \sum_{w=1}^{ν_1} FL_{v,w} ≥ \sum_{w=1}^{ν_1} FL_{v,w} = c_{i′} \), using \( FL_{v,w} ≥ FL_{v,w} \) as \( i′ < i \) (otherwise \( n_i = n_{i′} \)). Now when an overloaded machine \( m_{i′} \) becomes underloaded itself the computation of \( c_i \) changes and we have \( c_i = \sum_j fr_t(j) \cdot FL_j \geq \sum_{w=1}^{ν_1} FL_{v,w} \). In fact, if \( m_{i′} \) is initially overloaded and \( m_{i′} \) is underloaded we know that \( \sum_{w=1}^{ν_1} FL_{v,w} ≤ \sum_{w=1}^{ν_1} FL_{v,w} = c_{i′} \). But as \( m_{i′} \) has enough additional load \( x \) remaining on top of this s.t. \( x + \sum_{w=1}^{ν_1} FL_{v,w} > AVG \) we obtain \( c_i = \sum_{w=1}^{ν_1} FL_{v,w} + x < c_{i′} \). Therefore, \( c_i \) is no lower than \( c_{i′} \).
(AVG − C_f) > C_f. Here X − (AVG − C_f) is exactly the additional remaining load of the top file on m_i after m_f is closed. Thus the claim follows for m_i since it held for m_f before the update. □

Claim 3. Any shared file is stored on at most two machines storing another shared file.

Proof of Claim 3. Observe that each time an underloaded machine gets closed it gets closed in one chunk. As we preferentially close underloaded machines which were previously overloaded, at any time there is at most one underloaded machine which was previously overloaded. This ensures that any shared file is stored on at most two machines storing other shared files. One of these two machines can be the machine which, after the round-robin assignment, stores the single copy of the file and the second machine can be the machine that the initially overloaded machine closes first. □

Claim 4.

(1) If machine m_i only stores a single shared file f_j then for all t ML_i(j) ≤ fr_i(j) + β.

(2) If machine m_i stores f_j as well as another shared file f_j then for all t ML_i(j) ≤ fr_i(j) + β/2.

(3) For unshared files we have for all t ML_i(j) ≤ fr_i(j), where equality holds at the end of the online phase.

Proof of Claim 4. During the online phase, we first assign a request to a machine where it still fits. When we go above the projected file load we first pick machines without any other shared files. Here, in case (1) above, we can exceed the projected file load up to a factor β. If we exceed the projected file load for a machine which has two shared files then we only do this as a “last resort”, i.e. all other machines storing no other shared files are already full and the only remaining “holes” are, according to Claim 3, on the two machines with other shared files. Note that all holes combined are always big enough to accommodate the remaining file load which is still to come and so the bigger of the two holes will always be big enough to accommodate l(t)/2. This ensures that in case (2) above whenever there is a request t with load l(t) for file f_j, there is a machine m_i s.t. ML_i(j) + l(t)/2 ≤ fr_i(j). □

5. Variable space: \([n/m] \leq s \leq n\)

If the amount of space s per machine is not limited then by copying all files onto all machines one can obtain AVG + β by assigning each request to the least loaded machine. More generally, the following algorithm called “Cluster” makes a certain space-performance tradeoff possible, see Theorem 3.

Cluster. In addition to the regular input, Cluster is given q ∈ \(\mathbb{N}^+\) such that q|m and q|n. In the offline phase, Cluster divides the set of files and the set of machines into q equally-sized clusters \(F_1, \ldots, F_q\) and \(M_1, \ldots, M_q\). Afterwards, it assigns each of the \(n/q\) files in \(F_k\) to each of the \(m/q\) machines in \(M_k\). In the online phase, it assigns each request to the least-loaded permissible machine.

Theorem 3. If there exists q ∈ \(\mathbb{N}^+\) such that q|m and q|n, then the deterministic dual-phase algorithm “Cluster” uses exactly s = \(n/q\) slots per machine and assigns requests to machines such that

\[
ML = \left[1 + \left(1 - \frac{1 + \alpha}{q + \alpha}\right)\right] \cdot AVG + \beta.
\]

Note that within each cluster the assignment is unrestricted and so AVG + β can be achieved by a greedy approach. Here AVG denotes the average load of machines in the cluster. The worst case corresponds to the setting where one cluster has only files of size \((1 + \alpha)L\) whereas all other clusters have files of size L. An analysis of this case gives the bound above.

If we assume that the desired q is constant then asymptotically as n → ∞, m → ∞ and n/m → ∞ the condition that q divides both m and n can be dropped in a similar fashion to Corollary 1. Concretely, we can (i) use dummy files with load \(L\) to increase the number of files by at most q to ensure q|m and (ii) choose not to use at most q machines to ensure that q|m. This then leads to a new average satisfying \(AVG' = \frac{mnL + qL}{m+q} \leq \frac{mnL + qnL}{m+q} \leq (1+q/n) \cdot AVG = (1 + q/m) \cdot AVG = (1 + o(1)) \cdot AVG.

Corollary 2. Given any constant q ∈ \(\mathbb{N}^+\), “Cluster” uses \(s = [n/q]\) slots per machine and, for large n and m, has a makespan of

\[
ML ≤ \left[1 + \left(1 - \frac{1 + \alpha}{q + \alpha}\right)\right] \cdot (1 + o(1)) \cdot AVG + \beta.
\]

6. Lower bounds

Next we give lower bounds for any deterministic dual-phase algorithm. Theorem 3 (with q = 1) showed that without any unshared files we can obtain ML ≤ AVG + β and so the bounds depend on the number of unshared files.

Theorem 4. Let A be a deterministic dual-phase algorithm that produces a file assignment where at least one machine stores \(\lceil n/m \rceil\) unshared files. Then, for all n and m there exists a sequence of requests such that

\[
ML ≥ \left[1 + \left(1 - \frac{1 + \alpha}{m + \alpha}\right)\right] \cdot AVG.
\]

The proof follows in a straightforward way from an analysis of the case where the \(\lceil n/m \rceil\) unshared files on one machine have a file load of \((1 + \alpha)\)L, whereas all the other files have L. In this case ML = \(\lceil n/m \rceil\) · \((1 + \alpha)\) · L and AVG = \(\lceil n/m \rceil\) · \((1 + \alpha)\) · L. Dividing these two expressions and using the bounds \(\frac{n}{m} \leq \lceil n/m \rceil \leq \frac{n}{m} + 1\) and \(\frac{m}{n+q} \leq m \leq \) leads to the stated result. For settings where there are even more unshared files, we can get lower bounds of the form \((1 + \Omega(\frac{m}{n})) \cdot AVG\) for the typical case of \(\alpha < n\).
Theorem 5. Let A be a deterministic dual-phase algorithm that has at least \( cn \) unshared files, where \( 0 < \epsilon \leq 1 \). For all \( n \geq m \geq 3 \) and \( \alpha \geq 0 \) there exists a sequence of requests such that

1. \( ML^* \geq \frac{n}{2} \cdot \text{AVG} \) if \( \alpha \geq n \),

and

2. \( ML^* \geq \min(1 + \frac{m}{2n} \alpha, 1 + \frac{\epsilon}{1 - \epsilon} \frac{m}{2n} \alpha) \cdot \text{AVG} \) if \( \alpha < n \).

Proof. In this proof let \( k = n/m \geq 1 \), which does not need to be an integer.

Case 1. \( \alpha \geq n \). Since A uses less than \( 2n \) slots there exists an unshared file. The adversary gives a file load of \( kL \) to this file and \( L \) to every other file. Then \( ML^* \geq (1 + \alpha)L \) and \( \text{AVG} = kL + \alpha L/m \). Thus, \( ML^*/\text{AVG} \geq (1 + \alpha)/(k + \alpha/m) \), which is at least \( m/2 \) for \( \alpha \geq n \).

Case 2. \( \alpha < n \). We consider two cases depending on the file assignment.

Case 2.1. There exists a machine with \([k]\) unshared files. By Theorem 4, \( ML^* \geq [1 + (1 - 1/4 \alpha) \cdot \text{AVG} \) is at least \( 1 + \alpha/(4k) \) for \( \alpha < n \) and \( m \geq 3 \).

Case 2.2. There does not exist a machine with \([k]\) unshared files. It follows that every machine has at most \([k] - 1\) many unshared files. Thus, at least \( \epsilon n/(|k| - 1) \) machines have at least one unshared file. The adversary first sends a sequence of requests that gives equal load of \( L \) to all files. Depending on the machine loads after these requests the adversary proceeds as follows.

Case 2.2.1. There exists a machine \( m_\ast \) with an unshared file \( f_\ast \) and total load of at least \( kl - \alpha L/2 \). In this case the adversary sends another sequence of requests, each requesting \( f_\ast \) and having a total load of \( \alpha L \). As a consequence \( m_\ast \) has final load at least \( kL + \alpha L/2 \) and \( \text{AVG} = kL + \alpha L/m \). Thus \( ML^*/\text{AVG} \geq (k + \alpha/2)/(k + \alpha/m) \) is at least \( \alpha(m - 2)/(2n + 2\alpha) \), which is at least \( 1 + \alpha/(12k) \) for \( m \geq 3 \) and \( \alpha < n \).

Case 2.2.2. Every machine with an unshared file has at this point a total load of at least \( kl - \alpha L/2 \). In this case the adversary sends no further requests. Thus, \( \text{AVG} = kL \). Let \( S \) be the set of machines with at least one unshared file. Then the total load on the machines in \( S \) is at most \( |S|(kL - \alpha L/2) \). Thus on the remaining \( m - |S| \) machines there is at least a load of \( nL - |S|(kL - \alpha L/2) = (m - |S|)kL + |S|\alpha L/2 \). Thus there exists a machine \( m_\ast \) not in \( S \) with load at least \( kL + |S|\alpha L/2(m - |S|) \). Since \( |S| \geq \epsilon n/(|k| - 1) \) and \( |S|/(m - |S|) \) increases monotonically in \( |S| \), it follows that \( m_\ast \) has a load of at least \( kL + \epsilon\alpha L/(2(1 - \epsilon)) \). Thus, \( ML^*/\text{AVG} \geq 1 + \alpha/12 \epsilon^{-1} \). \( \square \)

References


---

4 When \(|k| = 1\), then \( n = m \) and Case 2.1 arises. Thus we can assume that \(|k| \geq 2\).