Proc. 24'th ACM Symp. on Theoretical of Computer Science, May 1992, Vancouver

Competitive Distributed Job Scheduling

(Extended Abstract)

Baruch Awerbuch * Shay Kutten † David Peleg ‡

Abstract

This paper examines the problem of balancing the job load in a network of processors, and introduces an online algorithm for scheduling a sequence of jobs in a competitive manner. The algorithm is shown to be polylog(n)-competitive according to a strict definition that forces the online algorithm to be competitive even when considering any bounded area of the network and bounded period of time.

We also analyze the common greedy feedback-based approach, and provide matching lower and upper bounds (up to a polylogarithmic factor) for the tradeoff between the costs of searches and updates under this approach.

1 Introduction

1.1 Motivation

Distributed multi-processor computing environments are becoming increasingly more common with time. For example, in the National Research and Education Network (NERN) intended by NSF to succeed the Internet (ARPANET), it is intended that jobs will be initiated by users, possibly on their workstations, and often sent for remote execution on more powerful machines, e.g. supercomputers. Several testbeds for the proposed NERN are already being built, e.g. the AURORA network [ACG+90]. High bandwidth for transferring the jobs will be available [CGK88], however the amount of traffic is expected to be so high that it will be essential that basic control functions be carried out automatically by the system. In this paper we focus on the basic task of balancing the load of jobs in such a network of processors.

The setting can be modelled as follows. The processors are placed in various geographically dispersed sites, connected through an interconnection network. At any given time, some of the processors may be occupied in running various (system or user generated) jobs, while some other processors may be idle. Jobs arrive continuously, in a dynamic fashion, at various sites. Only the processor at that site is aware of the job arrival. For each job arriving at a particular site, we are faced with the decision of whether to execute the job locally or to transfer it to some other, possibly less loaded or stronger machine. The system may incorporate machines of different power. However, transferring a job from one machine to another incurs a cost based on the characteristics of the network, e.g., the distance between the sites, the speed of the links, and so on.

Our goal is to optimize the operation of the system, by providing an efficient global scheduling of all the jobs on all the machines, that minimizes completion time. Our performance measure relates to the overall time each job has spent in the system from arrival to completion. The problem is to provide an online scheduling algorithm that is competitive with the quality of the best offline algorithm.

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. E-mail: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM.

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: kutten@watson.ibm.com.

‡Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. E-mail: peleg@wisdom.weizmann.ac.il. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award.

1.2 The problem

Formulating the situation in these terms leads to a problem closely related to the well-known *parallel machine scheduling* problem from classical scheduling theory. One simple version of that problem involves m jobs $j_1, \ldots, j_m$, with job j_i requiring t_i time units to complete, and n uniform machines. At most one job can be executed on each machine at any time. It is required to schedule the jobs on the machines so as to minimize the total completion time.

Observe that optimal scheduling is not necessarily achieved by evenly balancing the jobs among the processors, which is a problem considered in [PU89] and [AHS91], respectively. This is because it may not be worthwhile to send a job to a free machine which is too far away, and one must strike a delicate balance between availability of processing power and geographical proximity.

How close are we to the desired goal of efficient distributed management of the job scheduling ? It turns out that scheduling theory supplies us with only partial answers. In particular, the parallel machine scheduling problem has received considerable attention in the centralized, sequential setting, but it abstracts away several key factors of the underlying "real" problem:

- The problem is *offline*, i.e., the input jobs are assumed to be known at the beginning.
- The problem ignores the costs of allocating jobs to machines and transferring jobs from one machine to another.
- The solutions are centralized, i.e., it is assumed that the allocation decisions are made by a central agent, in a sequential fashion.

All three points have been addressed before, although separately, in different settings. In particular, the problem of online arrivals was addressed in several papers [Gra66, SWW91], albeit in the centralized environment. The problem of allocation costs was taken into account in a recent paper by [FST91] that gives tight bounds, but with centralized decision making in a specific model allowing job splitting between processors, and for certain special network topologies. Finally, the problem of distributed information maintenance has received some attention before in the area of distributed computing [AP89, AP90], however, this did not include making decisions (e.g. which machine to assign which job).

1.3 Contributions of this paper

Problem definition. In this paper we introduce, for the first time, the distributed online version of the problem, in which jobs arrive at the nodes of an asynchronous network with arbitrary topology, and arbitrary delays on network links.

In order to evaluate the performance of our solution we employ the standard online framework of Sleator and Tarjan [ST85]. In this framework, a "competitive factor" of an online algorithm on a finite (or infinite) sequence of inputs (requests) is captured by looking at the corresponding sequence of *prefixes* of the execution, and comparing the execution cost (i.e., the average completion time of a job) of the algorithm at hand on each prefix against the offline algorithm.

In our case, we will be comparing a distributed online algorithm running on asynchronous network with an offline (optimal) sequential scheduler. The former algorithm is handicapped not only because it does not know the future inputs, but also because it lacks the global information about the *current* state of the network, a difficulty that does not arise in the setting of sequential online algorithms of [ST85]. Consequently, in the online distributed setting it is possible to distinguish between two major aspects of control algorithms. The first aspect, which exists also in the centralized setting, is *decision making*. The second aspect is that of *knowledge maintenance*. By this we mean that it is necessary to constantly collect, spread and update information on the current global distribution state of the system in order to enable the decision making. While in the centralized setting this aspect is rather trivial, in the distributed setting it poses challenging problems, and adds complications to the first aspect as well.

For specific network topologies whose diameter is polylogarithmic in the number of nodes, such as expanders or Batcher networks, and assuming equal-size jobs, polylogarithmic competitive factor can be achieved by using methods of [PU89] and [AHS91], respectively. Using either these results or alternative ones based on the ideas of [Ran87], we argue in Section 4.1 that a $O(D \cdot \mathrm{polylog}(n))$-competitive solution can be achieved for any network with n nodes and diameter D. This approach is appealing for local area networks or shared-memory multiprocessors [AHS91], where the diameter is small.

However, linear dependence on the diameter is unacceptable in wide-area networks, where the diameter can be rather large. This is particularly true in heterogeneous network architectures, where delays of different network links varies arbitrarily.

Such variations are captured by assigning different weights to the network links, and considering weighted distances and weighted diameter. In this case, the weighted diameter D may significantly exceed the number of network nodes n, regardless of the network topology.

Competitive online algorithm. We eliminate the linear dependency on the diameter by providing an online solution to the distributed job scheduling problem, whose competitive factor is polylogarithmic in the number of network nodes, namely $O(\text{polylog}(n))$, for all possible network topologies, all possible link weights and all possible input (job arrival) sequences. Even though response time is the main complexity measure considered, the communication cost of our protocol is polylog-competitive, as well.

The main novelty of our solution is that it handles both information maintenance and decision making simultaneously, within the realistic distributed setting described above.

Let us say a few words on the general technique used for the algorithm and its competitiveness analysis. One of the tools we use is a simple variant of the "network decomposition" data structure of [AP90], which is a hierarchical family of overlapping clusters, covering all of the network nodes. These clusters have an internal part (the "nucleus"), consisting of the vertices that are "well inside" the cluster. This separation property between the nucleus and the exterior of the cluster is used for providing lower bounds on the performance of an offline algorithm on a given sequence of jobs for scheduling. One of the ideas of our solution is that the machines in a cluster must handle not all the jobs in the cluster, but rather only the jobs in the nucleus. This means that the separation estimates the distance from the jobs in the "nucleus" of the cluster to any other cluster that may help in performing them. Thus the separation enables us to lower-bound the "penalty" the offline algorithm is forced to "pay" for shipping jobs outside the periphery of a given cluster. The analysis of competitiveness is now based on the idea that in any given cluster, one of the following must happen: either (1) our online algorithm will succeed in redistributing the jobs in the nucleus *within* the cluster so that there is enough processing power (internally in the cluster) to complete the computation within some specified time bound, or (2) any competing algorithm will fail in the same task as well (up to a certain competitiveness factor). This allows us to look for the lowest-level cluster in which our algorithm will succeed, since distances of the same magnitude must be used by the optimal algorithm.

Novel measures for online performance evaluation. Another interesting issue that arises from this work is the question of how to define competitiveness in a distributed setting, assuming a very long (or even non-terminating) execution. The standard approach [ST85] allows the online algorithm to grossly misbehave *locally* in certain "epochs" of the execution's history: the fact that computers in the Thirties were extremely slow could be used as an excuse for the slow responses in the system last week.

While our algorithm *is* competitive according to this traditional definition as well, we feel that a more sensitive method of performance analysis is to make *localized* comparisons of performance, where locality is taken with respect to both *time* and *space*: The only possible excuse for a certain job j running slow on site v at time t, is that the *local conditions* necessitated this, i.e., any alternative scheduling algorithm would necessarily have done just as badly, if not on j itself then on some other job j' of *similar size*, at a *nearby site* v', and at roughly *the same time period* t'. (One cannot concentrate on j only, since the adversary could, in this case, improve the service to j only, at the expense of the service to the other jobs.) This type of local comparisons leads to a definition of competitiveness based on a *mapping* between executions of algorithms, matching individual job executions of the online algorithm at hand with those of other algorithms. We give a formal definition to this measure of competitiveness, and show that our protocol is competitive according to this (stronger) measure, as well.

Feedback-based policies. In the last part of the paper we address the problem from a different angle. We consider the common class of greedy, feedback-based policies for the job balancing problem. At any given moment, each processor may be either *overloaded*, meaning that its jobs queue is currently full in excess of a certain threshold value, or free to accept new jobs for execution. A greedy (or *locality-sensitive*) policy will attempt to assign each newly-arriving job individually to some *nearby* processor that is currently free, attempting to minimize the costs of relocation, output return, and any communication with the generating site that might become necessary during execution. These assignments are not coordinated in any way, and global control is achieved only through feedback. Namely, load changes cause processors to change their status from free to overloaded and vice versa. This general type of feedback-based policies is commonly used in communication networks for a variety of control tasks, e.g., for determining the best routes for packet routing [MRR80].

This approach introduces a tradeoff between the costs of searches for a near-by free processor and the costs of status updates. We identify the parameters of the problems that govern this trade-off, establish a lower bound on this tradeoff, and present an algorithm that is optimal up to polylogarithmic factors.

2 Preliminaries

2.1 Problem statement

Model. The computation model consists of an undirected graph $G = (V, E)$, $|V| = n$, where V is the set of nodes, representing the *machines*, and E is a set of links connecting them. Associated with G is a weight function $w : E \rightarrow \mathcal{R}^+$, assigning an arbitrary positive weight $w(e)$ to each edge $e \in E$, which is the upper bound on the *propagation delay* of the link e, in terms of some global clock.

In the standard asynchronous network model, which is the model considered in this paper, this global clock is not accessible by the machines, and is only used for the purpose of performance evaluation. It is worth stressing, however, that we do not know how to take advantage of synchronicity, namely availability of the global clock, and our improvements apply to the synchronous case, as well.

The distribution of jobs in the system at a given moment is represented by a (time-dependent) *job-distribution* mapping μ_J, that maps a set of jobs $\mu_J(v)$ into each node v in the network, representing its local queue at that time. (For simplicity, a job in transit from node u to v will be assigned to v.) Whenever no confusion arises we let $\mu_J(v)$ represent also the number of jobs maped to v. For subsets of nodes $S \subset V$, we define $\mu_J(S) = \sum_{v \in S} \mu_J(v)$.

We also denote by $\mu_P(v)$ the amount of processing power of v, and again extend the notation to sets of vertices in a similar way. For convenience, we define one unit of processing power as one unit on the slowest machine. For most of the sequel we assume the uniform machines model, i.e., set $\mu_P(v) = 1$ for every vertex v. However, our results extend also to the case of *uniformly-related* machines, where $\mu_P(v)$ is any constant number in $[1, \infty)$.

Both the set of jobs and the job-distribution mapping μ_J change in time in a manner that is described next. However, no global clock exists, and the times of such changes are not known to the nodes, neither in advance, nor after they occurred. Each node is assumed to "know" which jobs are mapped to it (*reside* in it) at any given time.

The input to the system consists of an online stream of *job arrivals*, $(J_i)_{i \geq 1}$. $J_i = (j_i, s_i, v_i, t_i)$ means that job j_i arrived at node v_i at time t_i, and was added to its local queue $\mu_J(v_i)$. The *size* of the job, s_i, represents the duration of its execution. When the job is executed on a node v starting at some time τ, it will complete by time $\tau + s_i/\mu_P(v)$. No two jobs can execute on the same node at the same time, although interleaved executions are allowed.

Given a job j residing at a node v, the node has several options. First, it can decide to execute the job locally. At the end of the execution, the job is removed from the queue of waiting jobs, $\mu_J(v)$. Another option is to send a job (or a batch of jobs) to some other node u in the network. If the sending operation was performed at time τ, and the weighted length of the path along which the job was sent from v to u is T, then node u receives the job at time $\tau + T$. The processing of a message requires one unit of time; if many messages arrive simultaneously at the same node, they are queued and subsequently processed at the rate of one message per unit of time per neighbor.

Complexity measures. In this paper, our main complexity measure is the amortized *time complexity*, which is the average response time of a job. For a given job, the response time of that job is the amount of time (in terms of the global clock above) that elapsed since the job is generated until this jobs if finally executed. This standard measure is somewhat strengthened in Section 5.2 so that averaging is limited to some small area in time and space, rather than to all the jobs.

The other measure considered is *communication complexity*, which is the sum of the communication costs of all message transmissions. The *communication cost* of transmitting a single ($O(\log n)$-bit) message length over edge e is the weight $w(e)$ of that edge. Note that communication complexity cost includes the cost of shipping the jobs between machines, as well as the cost of sending control messages of the protocol.

In addition to sending jobs, nodes can perform other kinds of local computations, and send messages of arbitrary contents to other nodes. The costs associated with these additional local computations (but not the messages) are ignored in this paper, as they are assumed to be negligible compared to the execution of the jobs and message reception.

2.2 Graph notation

Next let us define some basic graph notation. The *distance* between two vertices u, w in G, denoted $dist(u, w)$, is the weight of the cheapest path connecting them. The *distance* between two sets of vertices A, B is defined as the minimum distance between a pair of vertices in these sets. The τ-*neighborhood* of a vertex $v \in V$ is defined as $\Gamma_\tau(v) = \{w \mid dist(w, v) \leq \tau\}$. Given a subset of vertices $R \subseteq V$, denote $\Gamma_m(R) = \{\Gamma_m(v) \mid v \in R\}$. Let $D(G)$ denote the *diameter* of the network, i.e., $\max_{u,v \in V}(dist(u, v))$. Let $Rad(G)$ denote the *radius* of the network, i.e., $\min_{v \in V} \max_{w \in V}\{dist_G(v, w)\}$. A *center* of G is any vertex v realizing the radius of G (i.e., such that $Rad(v, G) = Rad(G)$).

Let us now recall some definitions concerning covers. A *cluster* is a subset of vertices $S \subseteq V$ whose induced subgraph is connected. We denote by $Rad(S)$ the radius of this induced subgraph. A *cover* is a collection of clusters $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $\bigcup_i S_i = V$. Given a collection of clusters $\mathcal{S}$, let $Rad(\mathcal{S}) = \max_i Rad(S_i)$. For every vertex $v \in V$, let $\deg_{\mathcal{S}}(v)$ denote the number of occurrences of v in clusters $S \in \mathcal{S}$, i.e., the degree of v in the hypergraph $(V, \mathcal{S})$. The *maximum degree* of a cover $\mathcal{S}$ is defined as $\Delta(\mathcal{S}) = \max_{v \in V} \deg_{\mathcal{S}}(v)$. Given two covers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and $\mathcal{T} = \{T_1, \ldots, T_k\}$, we say that $\mathcal{T}$ *coarsens* $\mathcal{S}$ if for every $S_i \in \mathcal{S}$ there exists a $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

3 Outline

Let us first comment that the job balancing problem has a simple centralized offline approximation algorithm, based on flow methods, even with job allocation costs taken into account. Since our main interest is in online and distributed solutions, we will not go into the details of this algorithm in this abstract.

It is convenient to partition the jobs into a logarithmic number of classes according to their sizes (with class i containing jobs whose sizes are between 2^{i-1} and 2^i) and handle each class separately. To recombine the separate schedules we use the standard multiprogramming technique, where each processor dedicates an equal fraction of its time to each class. This slows the computation by at most a logarithmic factor and consequently results in an increase of the competitiveness ratio by a logarithmic multiplicative factor, but it allows us to simplify the discussion by focusing our attention on a single class, in which all jobs are of unit size. (Since we aim for a polylogarithmic competitive ratio, we ignore more sophisticated approaches that deal with all the jobs together in an attempt to save this extra log factor.)

As a stage towards our final online algorithm, we present an *offline distributed* solution to the simplified *static* problem where all the jobs arrive initially, at the same time. This algorithm serves both to illustrate the ideas of our main online algorithm, and provide the basic mechanism to be used later on.

The algorithm relies on a hierarchy of clusterings that cover the network, and schedules the jobs within clusters in different levels of the clustering hierarchy, starting with the lowest level and going upwards. Intuitively, the scheduling policy we employ is based on handling each job at the lowest possible level in which this is feasible, that is, in the smallest cluster that is *balanced*, in the sense that it has enough processing power to handle all the jobs it has (up to a polylogarithmic factor). In every balanced cluster, we employ a distribution procedure for assigning jobs to processors as evenly as possible.

Our online algorithm for the dynamic problem is based on similar operations, combined so as to handle online arrivals efficiently. The jobs on each level are scheduled separately, in a periodic manner. A job that fails to be scheduled in a certain level attempts to be scheduled within a higher level cluster. Informally, failure to balance a cluster at a certain level signifies temporary congestion in a certain region of the network, and therefore it can be overcome by balancing the load in a larger cluster. Of course, the frequency of balancing phases on the different levels is decreasing exponentially with their height in the hierarchy.

4 Distributed offline scheduling for the static problem

4.1 Intra-cluster redistribution

In this section we handle the subproblem of distributing a collection of unit size jobs within a single cluster. We are given a collection X of unit size jobs in a cluster C. Our task is to process these jobs in time proportional to the sum of the cluster radius and the "load" factor, which is the ratio $|X|/\mu_P(C)$. This task can be achieved in several different ways. For instance, one may use a counting network of Aspnes, Herlihy and Shavit [AHS91], or a simpler nearly-balancing network based on the butterfly, which can be shown to balance its inputs up to an additive factor of $\log n$, which is suitable for our application. Another alternative is to use the token distribution method of Peleg and Upfal [PU89]. Both approaches require us to embed some network (a counting network or an expander) in the network at hand, and deal with pipelining problems in order to achieve efficient distribution.

However, it turns out that for the particular problem at hand, these methods can be replaced by a considerably simpler and more effective one, somewhat reminiscent of the combining method of Ranade [Ran87]. The idea is to fix a root node as a centralized manager of cluster C for the process of assigning jobs to processors. Jobs awaiting allocation issue requests, which are satisfied using a two-phase process. In the first phase, the requests are gathered upwards along a BFS spanning tree rooted at the center node in a "convergecast" fashion. The requests need not specify the current location of the job, and therefore individual requests can be combined together, passing upwards only a count of the number of jobs awaiting assignment. Let r_i be the number of requests passed by child i. Assumming the processors are properly numbered within each cluster, the root then issues a contiguous interval of processors (in round-robin, in case there are more requests than processors) to each of its children in the tree, such that the first r_1 processors are issued to the first child, and so forth. These intervals are split further as they are distributed down the tree, according to the number of requests that arrived from each branch. In the second phase, once each job is assigned a processor, the jobs are shipped to their intended location of execution using any efficient routing mechanism defined for the cluster.

Let $Load(C)$ be ratio between the number of jobs in C and its processing power $\mu_P(C)$.

Lemma 4.1 The jobs in cluster C can be executed in time $Load(C) + Rad(C)$. ∎

4.2 Well-separated clusterings

For the purpose of the scheduling algorithm we make use of a cover referred to as a *well-separated* clustering. By a clustering in this section we mean a collection of pairs $\mathcal{U} = \{(N_i, U_i) \mid 1 \le i \le q\}$, namely, we associate with each of our clusters a *nucleus*, which is a subset $N \subseteq U$ of the vertices in the cluster, intuitively thought of as the internal portion of the cluster. The nuclei are required to form a disjoint partition of V, i.e., $\bigcup N_i = V$ and $N_i \cap N_j = \emptyset$ for $i \ne j$.

Definition 4.2 A clustering as above is a *τ-well-separated clustering* of $G(V,E)$ if it has the following properties.

1. the *radius bound*: $Rad(\mathcal{U}) = O(\tau \log n)$
2. the *degree bound*: $\Delta(\mathcal{U}) = O(\log n)$
3. the *well-separation property*: $dist(N_i, V - U_i) \ge \tau$ for every $1 \le i \le q$,

Lemma 4.3 A well-separated clustering is constructible for every graph G and integer $\tau \ge 1$.

Proof Sketch: A τ-clustering can be constructed for any positive τ as follows. First construct the collection of τ-neighborhoods $\{\Gamma_\tau(v) \mid v \in V\}$, and then apply the maximum degree cover algorithm of [AP90], to get a collection of clusters $\mathcal{U} = \{U_i \mid 1 \le i \le q\}$ satisfying the radius and degree bounds of the above definition, and *coarsening* the collection of τ-neighborhoods, namely, satisfying that for every v, there exists a cluster $U_i \in \mathcal{U}$ such that $\Gamma_\tau(v) \subseteq U_i$. Next, for each node v, assign a `home` cluster, denoted $\texttt{home}(v)$, to be an arbitrary cluster $U_i \in \mathcal{U}$ containing $\Gamma_\tau(v)$. Finally, define N_i for each $U_i \in \mathcal{U}$ as its subset containing all vertices whose home is U, namely, $N_i = \{v \mid \texttt{home}(v) = U_i\}$. Now the first two properties are met trivially, and the well-separation property is satisfied by the fact that for every $v \in N_i$, the entire τ-neighborhood of v, $\Gamma_\tau(v)$, is contained in U_i. ∎

4.3 The offline scheduling algorithm

In this subsection we discuss the simpler static problem, in which all the jobs are given in advance, at the same time. We give a distributed offline algorithm for the problem, in which a processor is not aware of the inputs of other processors.

As a preliminary stage, we construct in the network a collection of clusterings $\mathcal{U}_i$, for $1 \le i \le D$, as in Section 4.2.

Definition 4.4 We define *balance factor* of a cluster $C = (N, U)$ given a job-distribution μ_J, denoted $\beta(C, \mu_J)$, as $\beta(C, \mu_J) = \mu_J(N)/\mu_P(U)$. We call a cluster U *τ-balanced* under job-distribution μ_J if its balance factor is bounded from above by τ. A *balanced* τ-clustering $\mathcal{U}$ under job-distribution μ_J is a τ-clustering in which every cluster is τ-balanced under job-distribution μ_J.

The algorithm proceeds in (at most) $\log D(G)$ phases, such that phase i handles the jobs remaining unprocessed in previous phases. Specifically, phase i operates as follows. The input to the phase consists of the (modified) node-weight functions μ_P and μ_J; the modification captures the fact that some of the jobs have already been processed in previous phases. During phase i, we use the (modified) weight function μ_J. Inside each cluster, we run the intra-cluster scheduling algorithm for the number of steps equal to the radius of the cluster. (This radius is upper-bounded by $O(2^i \cdot \log n)$.) We schedule only the jobs residing in N, and execute them over the entire cluster U.

Lemma 4.5 Consider the ith phase of the algorithm. There exists a scheduling of the jobs that completes every job j originated at vertex v within time $2^i \log n$, for the smallest i for which the cluster $\mathtt{home}(v) = (N, U)$ is balanced.

Proof Sketch: A cluster $C_i = (N_i, U_i)$ will be assigned for executing the jobs originated by the vertices in its nucleus N_i. Note that:

(a) each job gets assigned to some cluster,

(b) the amount of work assigned to a node in the cluster U_i w.r.t. the jobs in N_i assuming equal distribution is bounded by the balancing factor $\beta(C_i, \mu_J)$, and

(c) each node participates in at most $O(\log n)$ different clusters.

The result now follows from Lemma 4.1. ∎

Lemma 4.6 If the level i cluster $C = (N, U)$ is not balanced, then in every scheduling of the jobs, at least half of the jobs in it complete by time 2^{i-1} or later.

Proof Sketch: Consider an imbalanced cluster $C = (N, U)$, i.e., such that $\beta(C, \mu_J) = \frac{\mu_J(N)}{\mu_P(U)} > 2^i$. Note that a vertex v in the nucleus N is at distance 2^i or more from $V - U$, since its entire 2^i-neighborhood is contained in U. Hence a job J residing in the nucleus whose execution has been completed within time 2^{i-1}, must have been executed inside U, since leaving U would cost it 2^i steps. However, in 2^{i-1} time units, the nodes of the cluster U can process at most $2^{i-1} \cdot \mu_P(U) \le \mu_J(N)/2$ job units. Hence at least $\mu_J(N)/2$ units remain unprocessed at time 2^{i-1}. ∎

Corollary 4.7

1. If the 2^i-clustering $\mathcal{U}$ constructed in the i'th phase is balanced, then there exists a scheduling of the jobs, that runs in time $O(Rad(\mathcal{U}) \cdot \text{polylog}(n))$.
2. If the 2^i-clustering $\mathcal{U}$ is not balanced, then every scheduling of the jobs requires at least 2^{i-1} time.

∎

5 Online scheduling

We now consider the online dynamic case, where the jobs arrive dynamically at the vertices. In this abstract we ignore the degenerate case of system overload (i.e., too many jobs or too long jobs), especially since these cases are the less interesting ones, as when they occur, all processors work in full throughput. (It is clear that when the arrival rate is greater than the system's computational capacity, queues may increase without bound.) In fact, we make a stronger assumption for the abstract. Recall that we break the jobs into classes by size. We will assume that each class in itself is not overloaded, compared to its relative share in processing power. This will again allow us to treat a single class of identical-size jobs, and them merge the schedules appropriately by giving each class an equal share of the processing power. The case where some classes are relatively overloaded while other are relatively lightly loaded requires a more careful merging policy, and is left for the full paper. We now describe the algorithm, and subsequently analyze it using our strong definition of competitiveness.

5.1 The algorithm

Just as we did with the job classes, it is convenient to break the scheduling process itself into a logarithmic number of subtasks, where intuitively the ith task involves distributing jobs in the ith level of the hierarchy, based on 2^i-well-separated clusterings. As argued before, this results in an increase of the competitiveness ratio by a multiplicative factor of $\log D$.

The processes run in the network correspond to the levels described in the solution of the static problem. Specifically, process i repeatedly runs a balancing scheme on the clustering $\mathcal{U}_i$. That is, a new execution starts whenever the previous one ends, specifically every $O(2^i \log n)$ steps.

The jobs participate in these processes according to the following rule. A newly arriving job j enters the process on level 1, and attempts to be scheduled for execution in the next iteration of the balancing process on that level. If that iteration of the process succeeds for that job, i.e., its home cluster is balanced and therefore the jobs in its nucleus are completed successfully, then the job j is done. Otherwise, it is passed (along with all the unprocessed jobs from its cluster) to the 2nd level process. It awaits the beginning of the next iteration on that level, and participates in that iteration. The job proceeds going up the levels in this way, until it is executed.

5.2 Measuring competitiveness for online scheduling

We adopt the following approach to measuring competitiveness in this setting. Consider the sequence of job arrivals $J_i = (j_i, s_i, v_i, t_i)$, $i \geq 1$, ordered by arrival times t_i. For a scheduling σ, let $\mathcal{T}_i^\sigma$ denote the time elapsing from t_i until j_i is completed by σ. Clearly, a job-by-job comparison between two schedules is meaningless, since a particular schedule might always expedite the execution of some jobs at the expense of others. Therefore we use the following definition. We say that schedule σ is *α-competitive* with respect to schedule σ' if there is a mapping $\phi : N \mapsto N$ s.t.

1. only a constant number of jobs is mapped to any given job, i.e., $|\phi^{-1}(i)| \leq c$ for every i, for some fixed constant c,
2. the mapping is restricted to a local area, i.e., $dist(v_i, v_{\phi(i)}) = O(\mathcal{T}_i^\sigma)$, and
3. the mapping is restricted to a local time, i.e., $|t_i, t_{\phi(i)}| = O(\mathcal{T}_i^\sigma)$,
4. jobs are mapped to jobs of similar size, i.e., $\frac{1}{2}s_i \leq s_{\phi(i)} \leq 2s_i$, and
5. $\mathcal{T}_i^\sigma = \alpha \cdot \mathcal{T}_{\phi(i)}^{\sigma'}$.

Intuitively, whenever schedule σ is tardy in scheduling job j_i, it can "charge" this to a similar tardiness (up to factor α) of schedule σ' with respect to some other job $\phi(i)$ in approximately the same area of the network and at approximately the same time, and this "charging process" does not charge any individual job more than a constant times.

A schedule σ is *α-competitive* if it is α-competitive w.r.t. every other schedule σ'. A scheduling *algorithm* is *α-competitive* if for every graph and given any sequence of job arrivals, the dynamic algorithm produces an α-competitive schedule.

5.3 Analysis outline

Theorem 5.1 The dynamic algorithm is polylog(n)-competitive.

Proof Sketch: Let σ_A denote the schedule determined by our algorithm. In order to establish that our algorithm is *polylog*(n)-competitive, we need to exhibit an appropriate mapping w.r.t. every other possible schedule σ. Consider a job j_i, and suppose that it was executed by our algorithm on process level l. If $l = 1$, then let ϕ simply map i to itself. No other job will be mapped to J_i. It is also clear that our scheduling is competitive with every other scheduling, as far as job j_i is concerned. Therefore, let us suppose from now on that $l > 1$.

Prior to its execution on level l, the job j_i attempted participating in some iteration p of process $l-1$, and failed. Let (N, U) be the level $l-1$ cluster in which j_i was considered (i.e., $v_i \in N$), which was evidently unbalanced at that iteration. We then refer to the triple $(U, l-1, p)$ as the *critical block* of j, and denote the collection of jobs assigned to this triple as the block $\mathcal{B}(U, l-1, p)$. Note that each job j_i belongs to exactly one critical block, since its originator v_i has exactly one home cluster.

The mapping ϕ we shall construct (with respect to a given schedule σ) has the property that it maps each block into itself, i.e., the jobs of each block $\mathcal{B}(U, l-1, p)$, for every U, $l-1$, p, are mapped to jobs within the block. Thus the mapping obeys the requirements that j and $\phi(j)$ are close in time in space. Specifically, consider an arbitrary schedule σ, and consider the block $\mathcal{B}(U, l-1, p)$. By Lemma 4.6, since the level $l-1$ cluster (N, U) is not balanced in iteration p, at least half of the jobs in N are "tardy," in the sense that they require at least 2^{l-2} time units to complete under schedule σ. Let X be the subset of tardy jobs in the block $\mathcal{B}(U, l-1, p)$, and let the set Y contain the rest of the jobs ($|Y| \leq |X|$). Then ϕ will map the jobs of X to themselves, and the jobs in Y to X, so that at most two jobs get mapped to each job in X. Note that since each job belongs to exactly one critical block, this means that for every i, $\phi^{-1}(i)$ contains at most two elements.

The defined mapping is clearly total, and obeys the bound on the sizes of the inverse images. It remains to show that it satisfies the tardiness bound requirement. Let σ_A be the schedule produced by our algorithm. Since each job j_i in X requires time $\mathcal{T}_i^\sigma \geq 2^{l-2}$, it remains to bound $\mathcal{T}_l^{\sigma_A}$ from above for any job $J_l \in \mathcal{B}(U, l-1, p)$. These jobs where all performed in the next available iteration of process level l. The time job j_i may spend in the process corresponding to level l is $O(2^l \log n)$, by Lemma 4.7. Since this dominates the time spent by j_i in processes corresponding to lower levels, $\mathcal{T}_l^{\sigma_A} = O(2^j \cdot \log n)$. and the theorem follows. ∎

In the full paper we shall also discuss several generalizations of the problem. In particular, we will show that a theorem similar to Thm. 5.1 holds in the case of uniformly related machines and unknown job sizes.

6 A greedy feedback-based approach

We conclude by analyzing a distributed greedy feedback-based scheduling policy which is an alternative to the global scheduling discussed so far. At any given moment, each processor may be either `Free` to accept new jobs for execution, or *overloaded*, meaning that its jobs queue exceeds a certain threshold. An overloaded processor refuses to receive any new jobs. When its queue is emptied below the threshold, the processor will accept jobs once again. A new job arriving the system has to be allocated to some processor for execution. The policy is greedy (or *locality-sensitive*) in the sense that it will attempt to dispatch each job to a near-by processor, aiming at minimizing the costs of relocation, output return, and any communication with the generating site that might become necessary during execution.

The problem of locality-sensitive distributed greedy assignments has not been studied directly in the literature, to the best of our knowledge. In particular, the solutions of [AAPS87, ADD82] do not take the locality issue into account. The problem can be dealt with in a straightforward way via the distributed match-making policies of [MV88], but these policies employ a global view of the system, and hence do not achieve optimal performance.

A possible solution may be based on the distance dependent directories of [Pel91]. Using these directories for match-making as discussed above goes half way towards a satisfactory solution of the problem, in the sense that it can guarantee close to optimal "find" operations, as well as reasonable memory requirements. However, these directories are geared for static data, and the cost of updates is rather high. Even modifying these directories by basing them on the improved cover algorithm of [AP91], the update costs are still higher than $D(G)$.

It is clear that a search for a `Free` processor may be performed for the job in optimal time, by standard probing techniques. However, the communication requirements of such a policy may be prohibitively large. We are therefore led to the idea of maintaining a *distributed data structure* containing information on the current status of the different processors. That is, similar to the case of sequential algorithms, investing in an "update" operation, one can reduce the cost of a "find" operation. One solution is to employ the "full information" approach of [ACK90], keeping at each processor an accurate picture of processor status (`Free` or overloaded) in the entire network, rendering the cost of "find" operations optimal. However, the cost of the "update" operation will be high. We are interested in the inherent tradeoffs between the costs of updating the data structure and those of performing the search.

Informally speaking, our approach enables us to keep the overhead of "find" nearly optimal and still significantly reduce the "update" overheads, by allowing requests to be matched with resources whose location is "almost" the closest. Specifically, let $D_{\text{near}}(v) = \min\{dist(v,w) \mid \texttt{Free}(w) = 1,\ w \in V\}$. That is, $D_{\text{near}}(v)$ denotes the distance from v to the nearest free processor. An α-local policy (for parameter α) is a policy guaranteeing that whenever the instruction $\texttt{Find}(v)$ is issued from a vertex v, the free processor assigned to execute v's job by the dispatcher is at distance at most $\alpha \cdot D_{\text{near}}(v)$ from v. We prove close upper and lower bounds on the trade-offs between the costs of updates and the accuracy of the matching (measured by α).

Our bounds are given in terms of $D_{\text{near}}(v)$ and a second natural parameter, denoted $D_{\text{usable}}(v)$, capturing the maximal distance from which a job may be sent to v, given the current distribution of `Free` processors. (More formal definitions are deferred to the full paper.)

Our lower bound shows that if one wishes to keep the complexity of "find" operations optimal ($O(D_{\text{near}})$), then one must pay $\Omega(D_{\text{usable}})$ for updates. More generally, the product of the complexities of "find" and "update" is $D_{\text{near}} \cdot D_{\text{usable}}$.

In order to discuss the novel elements of our algorithm, recall that in Sections 2, 3 and 4.2, clusters have leaders, and changes in a node v (e.g., job arrivals) are reported to leaders that are as close as possible.

The main novelty in the feedback-based algorithm is that it is not necessary to report to leaders who are "much" farther away than D_{usable}, and still keep the complexity of "find" near-optimal (up to a polylogarithmic factor). This (in addition to the lower bound) shows that D_{usable} is indeed a basic parameter of this problem. The second novelty is again a specific clustering method, which constructs a hierarchy of covers with a new desirable property, which is required by our dispatcher. Namely, each cluster on a *higher level* of the hierarchy (a "larger" cluster) consists of the *union* of clusters of the level below it. Thus the status of a cluster of a higher level (i.e., information of the form "there exists a `Free` processor") can be computed by OR-ing the status information of the clusters at the level below, which is essential for our dispatching algorithm.

References

[AAPS87] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael Saks. Local management of a global resource in a communication network. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[ACG+90] Baruch Awerbuch, Israel Cidon, Inder Gopal, Marc Kaplan, and Shay Kutten. Distributed control for paris. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 145–160, 1990.

[ACK90] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of replicated information. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[ADD82] G.R. Andrews, D.P. Dobkin, and P.J. Downey. Distributed allocation with pools of servers. In *Proc. 1st ACM Symp. on Principles of Distributed Computing*, pages 73–83. ACM, August 1982.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, 1991.

[AP89] Baruch Awerbuch and David Peleg. Online tracking of mobile users. Technical Memo TM-410, MIT, Lab. for Computer Science, August 1989.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[AP91] Baruch Awerbuch and David Peleg. Routing with polynomial communication - space trade-off. *SIAM J. on Discr. Math.*, 1991. To appear. Also in Technical Memo TM-411, MIT, Sept. 1989.

[CGK88] Israel Cidon, Inder Gopal, and Shay Kutten. New models and algorithms for future networks. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 74–89. ACM, August 1988.

[FST91] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 111–120, 1991.

[Gra66] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.

[MV88] S.J. Mullender and P.M.B. Vitányi. Distributed match-making. *Algorithmica*, 3:367–391, 1988.

[Pel91] D. Peleg. Distance-dependent distributed directories. *Info. and Computation*, 1991. To appear. Also in Tech. Report CS89-10, The Weizmann Institute, May 89.

[PU89] D. Peleg and E. Upfal. The token distribution problem. *SIAM J. on Comput.*, 18:229–243, 1989.

[Ran87] A.G. Ranade. How to emulate a shared memory. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 185–194, 1987.

[ST85] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[SWW91] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 131–140, 1991.