Local Optimization of Global Objectives: Competitive Distributed Deadlock Resolution and Resource Allocation

Baruch Awerbuch * **Yossi Azar**†

Abstract

The work is motivated by deadlock resolution and resource allocation problems, occurring in distributed server-client architectures. We consider a very general setting which includes, as special cases, distributed bandwidth management in communication networks, as well as variations of classical problems in distributed computing and communication networking such as deadlock resolution and "dining philosophers".

In the current paper, we exhibit first local solutions with globally-optimum performance guarantees. An application of our method is distributed bandwidth management in communication networks. In this setting, deadlock resolution (and maximum fractional independent set) corresponds to admission control maximizing network throughput. Job scheduling (and minimum fractional coloring) corresponds to route selection that minimizes load.

1 Introduction

1.1 Informal problem statement

Motivation. The work is motivated by deadlock resolution and resource allocation problems, in distributed server-client architectures. We consider a very general setting, which includes, as special cases, distributed bandwidth management in communication networks, as well as deadlock resolution [BT87, BBG83, BC89, AM86, AKP91], and "dining philosophers" [AS90, ACS94].

The goal of this paper is to develop *local* algorithms with *globally-optimum* performance guarantees. The problems considered are related to "fractional" versions of maximum independent set and minimum coloring in hyper-graphs. While integer versions of these problem appear to be hard to approximate, [BGLR93, FGL+91, AS92, ALM+92], the versions, that happen to be the ones that matter in practice, do *not* fall into this class. Thus, there is no excuse for substituting "local maximality" for "global maximum", since the gap between the two often grows linearly in the size of the problem. This is in fact the disadvantage of existing techniques in the field of distributed computing, such as algorithms for *maximal* independent sets, $\Delta + 1$ coloring, and dining philosophers [Lub86b, Lub86a, Lin87, GPS87, AGLP89, AS90].

This paper in fact achieves *globally-optimum* solutions by local asynchronous algorithms. To the best of our knowledge, this is the first example of a local (poly-logarithmic time) distributed algorithm for which no non-trivial (constant time) "checker" is known, i.e. we do not see immediate way to verify correctness by considering the *immediate* vicinities of individual nodes.

Essence of the problem. The nature of the problem can be illustrated on the classical example of philosophers dining at a round table, with only one fork on the table in between each two nearby philosophers. Each philosopher needs two forks in order to eat.

If each philosopher grabs the left fork, then, in fact, we reach a situation of "deadlock", since no philosopher can eat with only one fork. While philosophers cannot *all* simultaneously eat, the "maximum-throughput" resolution of such a deadlock would require, say, every other philosopher to drop its fork which allows half of the philosophers to eat.

In more general version of this problem, different

*Johns Hopkins University, Baltimore, MD 21218, and MIT Lab. for Computer Science. E-mail: baruch@blaze.cs.jhu.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM.

†Department of Computer Science, Tel-Aviv University. E-mail: azar@math.tau.ac.il. This research was partially supported by the Alon Fellowship and the Israel Science Foundation administered by the Israel Academy of Sciences.

philosophers may need different accessories, e.g. some philosophers prefer knifes to forks. In the *arbitrary-access* version of the problem, philosophers are not so stubborn; i.e. *either one* of the two forks would suffice, provided that there also is a knife. Generally speaking, one may request any monotone boolean function of the requested resources.

Observe that in the maximum-throughput version, philosophers do *not* wait for for each other; we only want to maximize the number of philosophers that eat *immediately*, since after that, the food already becomes cold and thus uneatable.

In the standard formulation of "dining philosophers" [AS90, ACS94] problem, philosophers are in fact ready to wait, and thus, instead of maximizing the number of philosophers who eat immediately ("throughput"), we are interested in *minimizing time* it will take to feed *all* philosophers. In case of the dining at a round table, this would involve *two* phases of concurrent eating.

The real motivation, of course, is to deal with general resource allocation in general client-server architectures; i.e., in the above simple example, philosophers correspond to clients, and forks are servers. In such setting, only local information is available. Clients can only communicate to the accessible servers the sizes of their jobs being submitted, and servers communicate back to their client the server's load at the time, i.e., the total volume of all the jobs previously enqueued in the server queue.

Example: distributed bandwidth management. An important example to which our model applies includes *distributed* bandwidth management algorithms in high-speed networks, that so far has only been considered in online centralized setting [AAF+93, AAP93, AAPW94]. In case in which number of route selections is polynomial, our methods yield poly-logarithmically competitive algorithms.

Bandwidth management is modeled by having server's resources be bandwidth of a certain communication link, and clients be connections entering the network. Each connection may need simultaneous access to all links on the communication path from the sender to receiver. The different variations of the bandwidth management problem are captured in our setting as follows.

- *admission control*, i.e. decision on whether to admit an incoming connection, so as to maximize the total throughput, without exceeding link capacities [GGK+93, LT94, ABFR93, AAP93], is captured by the *maximum-throughput* version of the problem. In particular,
  - *flow control* issue, i.e. decision on how much traffic to admit into the network given a *fixed* path from sender to receiver, is captured by the *full-access* version of the problem.
- *route selection* issue, i.e. decision on how to route traffic, so as to minimize maximum link *load*, [AAF+93, AKP+93, AAPW94] is captured in our setting by *maximum time* deadlock resolution.

We stress that the "serially-competitive" routing algorithms, say in [AAF+93, AAP93] do not work in the concurrent setting. These algorithms operate by selecting the *shortest* weighted path for an incoming connection, where links weights grow exponentially with traffic admitted so far into the system. The coordination between routing decisions is expressed in that the load introduced by the previous connection must be incorporated into the routing decision made by the subsequent connection. While the algorithms works regardless of the order in which connections come in, their competitiveness is crucially dependent on proper coordination with respect to *some* order, making these algorithms infeasible for concurrent decision-making.

1.2 Our results versus existing work

Performance evaluation: "concurrent" competitiveness. As in this paper we would like to consider the problem in distributed concurrent setting, we first need to define the appropriate complexity measures. These definitions, informally outlined below, and further elaborated in Section 2.1, constitute one of the innovations of this paper.

In maximum-throughput version of the problem, there are two performance measures: *throughput competitiveness*, i.e. how many philosophers do we manage to feed compared to offline optimum, as well as *time* it takes our distributed algorithm to figure that out. Time performance of a distributed algorithm is measured in a standard way by assuming that the time it takes any client to communicate with any of the servers it is attached to is *exactly* (*at most*) one time unit in the synchronous (resp., asynchronous) network.

In minimum-time version of the problem, there is only one performance criteria - total completion time, which consists of actual execution time plus the number of rounds needed to compute the schedule. Even though our results hold for most general case, to simplify our initial discussion and to develop intuition, we will be restricting ourselves to the special case in which job run-time is one unit, (as in [AS90, AKP92]), e.g. run-time equals communication delay between servers and clients.

As in [AKP92], the efficiency is measured by the competitive ratio in total running time of the distributed algorithm, including both the time to distributively *construct* and *execute* the schedule. In contrast, offline algorithm does not waste any time to distributively construct the schedule.

Unlike in online centralized version of the somewhat simpler problem, where, as pointed out by [SWW91], we can always achieve a factor of 2 in completion time by essentially reducing the problem to an offline problem, this option does not exist in the distributed version of the problems considered in this paper.

Related work *Centralized* algorithms approximating the maximum fractional independent set and maximum fractional coloring, can be easily obtained by linear programming or incorporating the techniques in [PST91]. Our problem can be viewed as dual of positive linear programs considered by Luby and Nisan [LN93] who also provided a parallel approximation algorithm, which, however, lacks the desired locality properties.

The deadlock resolution and job scheduling problems are analogous to fractional versions of maximum independent set and coloring problems in hypergraphs. Combining techniques in [RT85, Rag86] with methods in [PST91] yields centralized approximations.

Online *centralized* scheduling and load balancing algorithms were considered various of papers such as [SWW91, ANR92, ABK92, BFKV92, AAF+93, PSW94].

Unfortunately, there exist no "competitive" distributed deadlock resolution strategies, in the sense that all known techniques for distributed symmetry breaking and deadlock resolution [CM83, BT87, BBG83, BC89, AM86, AKP91, AS90, ACS94], even though ensure eventual progress, have competitive ratio that may grow linearly in the number of processors involved.

Results and techniques of this paper. In contrast, in this work, we provide first competitive distributed solutions, that have logarithmic or polylogarithmic overhead.

- *maximum-throughput:* our algorithm in Section 2 computes the schedule in $O(\log n)$ time in either synchronous or asynchronous setting, and achieves $O(\log n)$ throughput-competitiveness.
- *minimum-time:* our algorithm in Section 3 computes the schedule in $O(\log^3 n)$ time in either synchronous or asynchronous setting, and achieves $O(\log n)$ time-competitiveness.

We comment that the asynchronous version of our algorithm poses another attractive feature, which is is *wait-freedom:* undetectable failure of one client will not slow scheduling for another client provided that the servers are reliable.

In the new algorithms, we build on techniques used in context of online resource allocation [AAF+93, AKP+93, AAP93, AAPW94] as well as on techniques used in field of distributed computing. Our algorithm is similar to the Luby-Nisan algorithm [LN93].

Structure of this extended abstract. Maximum-throughput problem is handled in Section 2 and Minimum-time (load) problem is handled in Section 3. We prove the minimum-time fractional algorithm in Section 4. In the in the final version we show how to achieve the integer solution via randomization and rounding techniques, and provide the proofs for the max-throughput case.

2 Maximum-throughput

In this section, we deal with maximum-throughput deadlock resolution. We start with the simplest "full access" case, in which job requests access to a specific set of resources (that may may depend on the job). We then generalize it to more general case, in which choice is possible.

2.1 Full-access maximum-throughput problem

Generally speaking, we have a collection of clients ("philosophers") $\mathcal{X}$, with a job of *demand* d^s, associated with each client $s \in \mathcal{X}$. Also, we have a collection of servers ("resources" or "forks") E, each server $e \in E$ having a *capacity* $c(e)$. During its execution a job $s \in \mathcal{X}$ needs access to subset $P(s) \subset E$ of servers consuming d^s resources from each of these servers.

The essence of full-access deadlock resolution is to find an approximately "maximum" weighted subset $\mathcal{I} \subset \mathcal{X}$ of clients (philosophers), that can be concurrently scheduled without exceeding capacity constraints at the servers. Formally, we need to maximize the "throughput" $\sum_{s\in\mathcal{I}} d^s$, subject to the "capacity constraints" $\sum_{s\in\mathcal{I}|e\in P(s)} d^s \le c(e)$, for all server e.

Let $n = \max\{|\mathcal{X}|, |E|, \rho\}$ where $\rho = \max_e c(e)/min_e c(e)$.

The fractional version of the problem. Instead of making an "integer decision" about admitting jobs (yes or no) it is much easier to make a "fractional decision", i.e., determine values $0 \le p^s \le 1$ indicating the *fraction* of each job $s \in \mathcal{X}$ to be executed. We define $f^s = d^s p^s$ which is the absolute size of the fraction of the job s. The capacity constraints for this version of the problem are modified as $\sum_{s\in\mathcal{I}|e\in P(s)} f^s \le c(e)$ where the goal is maximize "fractional throughput" $\sum_{s\in\mathcal{X}} f^s$. This is the version of the problem for which our algorithms will be designed.

To transfer the fractional solution to an integer one we view $p(s)$ as values which are proportional to the *probability* that the jobs s is executed (not aborted). (The formalisms are elaborate in the final version.) For this transformation to work, we need to make a (quite realistic) assumption (at least, in case of virtual circuit routing), that capacity of each resource exceeds size of each job by a logarithmic factor, i.e.

$$\min_{e\in P} c(e) = \Omega(\log n) \cdot \max_{s\in\mathcal{X}} d^s \tag{1}$$

We comment that the general "integer" problem, without making an assumption of such type, is provably un-approximable [BGLR93, FGL+91, AS92, ALM+92], unless P = NP. Indeed, the *maximum-throughput* (*minimum-time*) problems, are in fact, generalization of *maximum* independent set (*minimum coloring*, resp.) problem on hyper-graphs.

2.2 Max-throughput full-access algorithm

The algorithm executed by each job s (see Figure 1) works as follows. It starts by calling Procedure INIT, which initializes the assignment to some small of its demand d^s. Then, inside the inner loop, this fraction is successively doubled, using procedure PUMP, until either total assignment reaches the value of demand d^s, or the local "weight variable" $weight^s_P$ exceeds 1. The latter weight variable is updated by procedure UPDATE.

The procedures used by the main algorithm in Figure 1 are described in Figure 2.

Procedure INIT defines the "bottleneck capacity" c_P as the minimum server capacity in P, and then sets the initial assignment to be $1/n^2$ fraction of the minimum between demand and the bottleneck capacity.

Procedure UPDATE computes the non-linear function of the loads at the servers used by this job. Specifically, this is the sum, over all the servers, of $\Psi(h^s_{P,e})$, where $\Psi(h) = ((3n)^{2h} - 1)/n$. Estimates $h^s_{P,e}$, which are the load on the servers with respect to this source, are determined according to the messages CURRENT_LOAD$^s_P(h)$ received from servers. We should note that the weight is a measure for the usage of the whole subset (e.g. [SM90]), rather than for a specific server. We may stop increasing the load on set well before any single server over-utilized.

Procedure PUMP$^s(P,\epsilon)$ is used to increase the assignment of job by ϵ factor; in this case we choose $\epsilon = 1$. The need for the graduate growth in the assignment value is to prevent the effect of extreme changes for the load on any server. This procedure is also in charge of communicating the new load ADD_LOAD$^s_P(\Delta f)$ to e. It will subsequently wait for the reply CURRENT_LOAD$^s_P(h)$ from e, contains the

Call Procedure INIT$(P = P(s), \Lambda = 1)$
repeat
  Call Remote Procedure PUMP$^s(P = P(s), \epsilon = 1)$,
  Call Local Procedure UPDATE$^s(P = P(s), \Psi(h))$
  where $\Psi(h) = ((3n)^{2h} - 1)/n)$,
until $f^s_P \ge d^s$ **or** $\sum_{e\in P} weight^s_P \ge 1$

Figure 1: Full-access Maximim-thruput algorithm w.r.t. client s. Uses procedures in Figure 2.

Define Remote Procedure PUMP$^s(P,\epsilon)$
  $\Delta f^s_P \leftarrow \epsilon f^s_P$
  $f^s_P \leftarrow f^s_P + \Delta f^s_P$
  $\forall e \in P(s)$
    send message ADD_LOAD$^s_P(\Delta f)$ to e
    await CURRENT_LOAD$^s_P(h)$ from e
    $h^s_{P,e} \leftarrow h$
Define Local Procedure UPDATE$^s(P,\Psi)$
  $weight^s_P \leftarrow \sum_{e\in P} \Psi(h^s_{P,e})$
Define Procedure INIT(P,Λ)
  $c_P \leftarrow \min_{e\in P} c(e)$
  $f^s_P \leftarrow 1/n^2 \cdot \min\{d^s, \Lambda \cdot c_P\}$
  $weight^s_P \leftarrow 0$

Figure 2: Procedures used for Deadlock Resolution Algorithms.

current load on e, and update load estimate $h^s_{P,e}$ accordingly.

The algorithm executed by each server (see Figure 3) is straightforward: simply keep track of its load, and after the load is being changed by some job, the server reports back the new load.

Specifically, let L_e be the current load on server e normalized by its capacity $c(e)$. Whenever a server receives a message $\text{ADD\_LOAD}^s_P(\Delta f)$ from job s, it means that that this job increased its demand by Δf, and thus the server's load is increased by Δf (normalized by the $c(e)$) and the reply $\text{CURRENT\_LOAD}^s_P(L_e)$ is sent back, carrying the normalized load on that server.

for message $\text{ADD\_LOAD}^s_P(\Delta f)$ from s:
  $L_e \leftarrow L_e + \Delta f / c(e)$
  send $\text{CURRENT\_LOAD}^s_P(L_e)$ to s

Figure 3: Algorithm execution by each server e.

Theorem 2.1 The algorithm in Figure 1 achieves $O(\log n)$ throughput-competitiveness, and converges in $O(\log n)$ distributed time, either in synchronous or asynchronous distributed computation model.

Proof: Omitted. ▮

2.3 Maximum-throughput arbitrary access problem

In a different version of the problem, job $s \in \mathcal{X}$ may request access to only *one* of the resources in $\mathcal{Y}(s)$. This versions of the problem will be referred to as "OR" version, in contrast to previously discussed "AND" version in which access to all resources is required.

More generally, a client s, instead of requesting access to a single set P, requests access to at least one of sets of a a collection $\mathcal{P}(s) = \{P_1(s), P_2(s) \ldots P_k(s)\}$. Each set $P \in \mathcal{P}(s)$ consists of a number of servers, $P = \{e^1_i(s), e^2_i(s), \ldots e^l_i(s)\}$.

This captures an arbitrary monotone boolean function (written as a DNF formula), e.g., if client s needs either resource e_1 or both resources e_2, e_3, this corresponds to setting $P_1(s) = \{e_1\}$ and $P_2(s) = \{e_2, e_3\}$.

As for the full-access problem, we can define the fractional version of the problem. Here instead of one variable f^s for each job s we have many variables $\{f^s_P | P \in \mathcal{P}(s)\}$, for the different feasible subsets each job s. Here each job may split faction of its demand among the possible feasible subsets.

If we want to select of a given collection of communication paths in order to minimize the load, this can be captured by presenting the function as sum of minterms, each minterm representing another communication path.

2.4 Max-throughput arbitrary access algorithm

The algorithm, presented in Figure 4, is just as before, with the difference that the job maintains a list of "active" feasible subsets, with a single feasible subset P being active if its weight is still less than 1, and increases its assignment only on active feasible subsets. All the procedures and the server algorithm remain as before, i.e. as in Figures 3 and 2, respectively.

for all $P \in \mathcal{P}(s)$, **in parallel**
  Call Procedure $\text{INIT}(P, \Lambda = 1)$
repeat
  for all $P \in \mathcal{P}(s)$ s.t. $weight^s_P < 1$
    Call Remote Procedure $\text{PUMP}^s(P, \epsilon = 1)$
    Call Local Procedure $\text{UPDATE}^s(P, \Psi(h))$
until $\sum_{P \in \mathcal{P}(s)} f^s_P \geq d^s$ **or** $\forall P \in \mathcal{P}(s), weight^s_P \geq 1$

Figure 4: Deadlock Resolution Algorithm for the general case w.r.t. client s that needs access to one the sets of servers $P \in \mathcal{P}(s)$. Uses procedures in Figure 2.

Theorem 2.2 The algorithm in Figure 4 achieves $O(\log n)$ throughput-competitiveness, and converges in $O(\log n)$ distributed time, either in synchronous or asynchronous distributed computation model.

3 Minimum time/load

3.1 Problem statement

The input for the Minimum-time deadlock resolution is the same as for the the arbitrary access throughput problem. However, the goal is to schedule *all* the jobs in non-conflicting way. More specifically we need to find a feasible subset $P(s) \in \mathcal{P}(s)$ and a time, (*color*) $T(s)$ such that the capacity constraints are satisfied at each step i.e.

$$\forall e, t \quad \sum_{s | T(s) = t, e \in P(s)} d^s \leq c(e).$$

The goal is to minimize the maximum T. We can define somewhat relaxed version of the problem in which

```
for all P ∈ 𝒫(s), in parallel
    Call Procedure Init(P(s), Λ = Λ*)
a = 1 + γ/8,   β = log_a |E|/(1 − γ) = O(log n)
stage ← 0
repeat
    stage = stage + 1
    repeat
        for all P ∈ 𝒫(s), s.t. weight^s_P < 2^stage,
            Call Procedure Pump^s(P, ε = β^{-1})
            Call Local Procedure Update^s(P, Ψ_e(h))
            where Ψ_e(h)) = a^{h/Λ}/c(e)))
    until ∀P ∈ 𝒫, weight^s_P > 2^stage
until Σ_{P∈𝒫} f^s_P ≥ d^s or 2^stage > Λ · (β + 2)
```

Figure 5: The *load* or *minimum-time* deadlock resolution for job s. Uses procedures in Figure 2.

we need to choose for each s, $P(s) \in \mathcal{P}(s)$ where the goal is to minimize the maximum load which is

$$L_e = \sum_{s|e\in P(s)} d^s/c(e).$$

It is clear that the maximum load is a lower bound for the minimum-time deadlock resolution since we do not require that all the resources for some job will be scheduled simultaneously. Nevertheless, if we assume that $\min_e c(e) = \Omega(\log n) \cdot \max d^s$ then techniques of [LMR88] can be used to achieve a randomized algorithm for the minimum-time deadlock resolution. This increases the competitive ratio only by a constant factor. Specifically, a job chooses a random slot uniformly among the target number of time slots. The value of this target number is some constant time the number of slot achieved by the min load algorithm. Furthermore, we can define the fractional version of the load problem as for the throughput problem. Again using randomization one can translate a solution for the fraction problem to a solution to integer problem by increasing the competitive ratio by only a constant factor as described in final version. The above allow us to concentrate from now on on the fractional load problem since its solution yields a randomized solution for minimum-time deadlock resolution.

3.2 Minimum load algorithm

The algorithm in Figure 5 deals with the fractional load version problem. The algorithm, for each job, proceeds in stages, with the goal of each stage *stage* being to maximally utilize "active feasible subsets". These are defined as feasible subsets of "weight" less or equal to 2^{stage}. The weight of a feasible subset is the sum of the weights of individual servers, which grow exponentially with the utilization of these servers.

Throughout a stage, the job will gradually increase the volume of the jobs sent over the active feasible subsets, until all active feasible subsets become "saturated" and thus cease being active, or until the assigned fraction satisfies the demand.

Specifically, each stage will consist of a number of phases, each phase increasing the assignment over each active feasible subset by certain fraction $1/\beta$ with some appropriate initial assignment. Increasing the assignment on a feasible subset P during a phase is done by calling the procedure $\textsc{Pump}^s(P, \beta^{-1})$.

We assume that the algorithm is given a value Λ^* which is larger or equal to the load of the optimal algorithm (Λ^* can be found by doubling). Here the function $\Psi_e(h)$ is define for some constant $a > 1$ as

$$\Psi_e(h) = a^{h/\Lambda^*}/c(e)$$

which results in setting

$$weight_P = \sum_{e\in P} a^{L_e/\Lambda^*}/c(e).$$

The values a and β above defines as follows. Let $a = 1 + \gamma/8$ for some arbitrary constant $0 < \gamma < 1$ and $\beta = \log_a |E|/(1-\gamma) = O(\log n)$. Also let $c_P = \min_{e\in P} c(e)$. Without loss of generality we normalize the capacity such that $\max_e c(e) = 1$.

Theorem 3.1 The algorithm in Figure 5 (given Λ^*) achieves $O(\log^3 n)$ time-competitiveness either in synchronous or asynchronous distributed computation model.

Comment: In fact, the algorithm $O(\log n)$ time-competitive in centralized model, in which communication between servers and clients takes negligible time. It converges in $O(\log^3 n)$ phases.

4 Load Analysis

In this Section, we provide the proof of Theorem 3.1. Since we describe the load version of the problem, jobs are only scheduled but not executed until the end of the scheduling phase. Denote by $f^i_P(k)$ the value of f^i_P at the end of phase $k = k_i$ of job i. Clearly $f^i_P(k)$ is 0 for $k = 0$ and is a monotone non-decreasing function of k. The *incremental volume* of a feasible subset P at phase k of job i is denoted as $\Delta f^i_P(k)$. By definition

$$\Delta f^i_P(k) = f^i_P(k) - f^i_P(k-1) \geq 0$$

To simplify the formulas, we will use $\tilde{L}_e(t) = L_e(t)/\Lambda^*$ to denote the normalized load of the distributed algorithm (i.e. load divided by the offline load), and will use the ˜ notation consistently.

For the purpose of analysis we define a mapping from the global time t to the index of the phase number of each server. Let $k_i(t)$ be the index of the phase of job i at time t. Also let $k^i_P(t)$ be the index of the last phase of job i before time t that feasible subset P was active and at a later phase (but still before time t) it was active again.

Denote by t^* the time immediately after the incremental volume of job i subscribed to server e of feasible subset $P \in \mathcal{P}^i$ at phase k. Let

$$h^i_{P,e}(k) = L_e(t^*)$$

If P is not active at phase k of job i then its height on each server is the same as in the previous phase (initially it is 0).

We define the *relative load* of job i on e due to feasible subset$P$ at the end of phase k as

$$f^i_{P,e}(k) = f^i_P(k)/c(e)$$

provided that $e \in P$ (otherwise it is 0) and similarly define $\Delta f^i_{P,e}(k)$.

The *committed relative load* on server e at time t is defined by:

$$\ell_e(t) = \sum_{i,P:e\in E} f^i_{P,e}(k^i_P(t))$$

We call the load which is not yet committed *pending committed relative load.* Clearly at any time the load on an server is the sum of the committed load and the pending load. Clearly the the pending load consists of the union of the incremental demand of the current or the last positive incremental demand of each feasible subset. We use the notation of ˆ for the parameters of the off-line centralized algorithm. In particular, $\hat{f}^i_P$ is the part of the demand that is assign by the centralized algorithm to the subset P by job i and $\hat{f}^i_{P,e} = \hat{f}^i_P/c(e)$

We first prove the following lemmas

Lemma 4.1 For any i,

$$\sum_P \sum_k \Delta f^i_P(k) = \sum_P f^i_P \le 2d_i = 2\sum_P \hat{f}^i_P$$

Proof: The equalities follows from the definitions. We prove the inequality. As long as the job is still unsatisfied then by definition $\sum_P f^i_P(k) \le d_i$. At the last phase k' (i.e when the job becomes satisfied) the value of each $f^i_P(k'-1)$ may increase by a factor of $(1+1/\beta)$ for $f^i_P(k'-1) > 0$. Also the value of at most n feasible subsets may increase from 0 to at most $d_i/(n^2)$. Thus,

$$\begin{aligned}\sum_P f^i_P &= \sum_P f^i_P(k') \\ &\le (1+1/\beta)\sum_P f^i_P(k'-1) + n\cdot d_i/(n^2) \\ &\le (1+1/\beta)d_i + d_i/n \le 2d_i\end{aligned}$$

∎

The algorithm maintain the following

Definition 4.2 The Main Induction Hypothesis at time t (or up to time t) denote by MIH(t) is as follows: for any job i, any feasible subsets $P, P' \in \mathcal{P}^i$ in and $k \le k^i_P(t)$

$$\sum_{e\in P} a^{\tilde{h}^i_{P,e}(k)}/c(e) \le 4\sum_{e\in P'} a^{\tilde{\ell}_e(t)}/c(e).$$

Lemma 4.3 MIH(t) implies $\sum_{e\in E} a^{\tilde{\ell}_e(t)} \le |E|/(1-\gamma)$ and thus $\tilde{\ell}_e(t) \le \beta = O(\log n)$

Proof: If $\sum_{e\in E} a^{\tilde{\ell}_e(t)} \le |E|/(1-\gamma)$ then clearly $\tilde{\ell}_e(t) \le \log_a(|E|/(1-\gamma)) = \beta$

MIH(t) with Lemma 4.1 imply that for any i

$$\begin{aligned}&\sum_P \sum_{k\le k^i_P(t)} \Delta f^i_P(k) \sum_{e\in P} a^{\tilde{h}^i_{P,e}(k)}/c(e) \\ &\le 8\sum_P \hat{f}^i_P \sum_{e\in P} a^{\tilde{\ell}_e(t)}/c(e)\end{aligned}$$

or

$$\begin{aligned}&\frac{\gamma}{8}\sum_P \sum_{k\le k^i_P(t)} \sum_{e\in E} a^{\tilde{h}^i_{P,e}(k)} \Delta\tilde{f}^i_{P,e}(k) \\ &\le \gamma \sum_P \sum_{e\in P} a^{\tilde{\ell}_e(t)} \tilde{\hat{f}^i}_{P,e}\end{aligned}$$

Note that since $a = 1+\gamma/8$ we have that $\forall x \ge 0:\ (1-a^{-x}) \le (\gamma/8)x$. Applying the inequality for $x = \tilde{f}^i_{P,e}$ and using the inequality above yield:

$$\begin{aligned}&\sum_P \sum_{k\le k^i_P(t)} \sum_{e\in E} (a^{\tilde{h}^i_{P,e}} - a^{\tilde{h}^i_{P,e}(k) - \Delta\tilde{f}^i_{P,e}(k)}) \\ &= \sum_P \sum_{k\le k^i_P(t)} \sum_{e\in P} a^{\tilde{h}^i_{P,e}}(1 - a^{-\Delta\tilde{f}^i_{P,e}(k)}) \\ &\le \frac{\gamma}{8}\sum_P \sum_{k\le k^i_P(t)} \sum_{e\in E} a^{\tilde{h}^i_{P,e}} \Delta\tilde{f}^i_{P,e}(k) \\ &\le \gamma \sum_P \sum_{e\in E} a^{\tilde{\ell}_e} \tilde{\hat{f}^i}_{P,e}\end{aligned}$$

Summing over all currently active jobs, we get:

$$\sum_{i,P} \sum_{k \le k_P^i(t)} \sum_{e \in E} (a^{\tilde{h}^i_{P,e}(k)} - a^{\tilde{h}^i_{P,e}(k) - \Delta \tilde{f}^i_{P,e}(k)}) \le \gamma \sum_{i,P} \sum_{e \in E} a^{\tilde{\ell}_e(t)} \tilde{\hat{f}}^i{}_{P,e}$$

Exchanging the order of summation yields

$$\sum_{e \in P} \sum_{i,P|e \in E} \sum_{k \le k_P^i(t)} (a^{\tilde{h}^i_{P,e}(k)} - a^{\tilde{h}^i_{P,e}(k) - \Delta \tilde{f}^i_{P,e}(k)}) \le \gamma \sum_{e \in E} a^{\tilde{\ell}_e(t)} \sum_{i,P|e \in E} \tilde{\hat{f}}^i{}_{P,e}$$

Observe that the fact that the normalized load of the offline algorithm never exceeds 1 implies that $\sum_{i,P|e\in P} \tilde{\hat{f}}^i{}_{P,e} \le 1$. Also, for each server e the left hand-side is a telescopic sum without the differences created by the pending incremental demand on that server. However, reducing the height of the each committed incremental demand by the accumulative size of the pending demand on that serverwith of lower heights may only reduce the value of the expression. That results in a telescopic sum which is just $a^{\tilde{\ell}_e(t)} - a^0$. Thus we conclude that

$$\sum_{e \in E} (a^{\tilde{\ell}_e(t)} - 1) \le \gamma \sum_{e \in E} a^{\tilde{\ell}_e(t)}$$

Using the fact that $\gamma < 1$, we get

$$\sum_{e \in E} a^{\tilde{\ell}_e(t)} \le |E|/(1-\gamma).$$

▮

Theorem 4.4 The algorithm maintain MIH(t)

Proof: Clearly MIH holds initially. New inequalities are added to MIH only when some $k_P^i(t)$ is increased. That occurs after some job i_0 starts a new phase which causes some previous pending incremental demand of feasible subsets in $\mathcal{P}^{i_0}$ to become committed. Let t^+ be a time immediately after committing and t^- a time just before committing. Both times are at the same phase k_{i_0}. We assume by induction that the MIH(t^-) and show show that it is maintained also at t^+.

By assumption the inequalities hold at t^- for any feasible subsets $P, P' \in \mathcal{P}^i$. Since the right hand-side is monotonic non-decreasing function of t they also hold at time t^+ for all i, P and $k \le k_P^i(t^-)$. Thus, we need to proof the inequalities for any committed $P \in \mathcal{P}^{i_0}$ and arbitrary $P' \in \mathcal{P}^{i_0}$ where $k^* = k_P^{i_0}(t^+)$. We first proof the following:

Lemma 4.5 For any $t \le t^-$, $\tilde{L}_e(t) \le \tilde{\ell}_e(t) + 2$

Proof: There is at most one pending incremental demand for each f_P^i for any unsatisfied job. The value of this incremental demand is at most f_P^i/β if the committed flow of job i on feasible subset$P$ is positive and it is at most $\Lambda^* c(e)/n^2$ otherwise. Since the load is just the sum of the flow we have

$$\tilde{L}_e(t) - \tilde{\ell}_e(t) \le \tilde{\ell}_e(t)\beta + n \cdot (\Lambda^* c(e)/n^2)/(\Lambda^* c(e)) = \tilde{\ell}_e(t)\beta + 1/n \le 2$$

where the last inequality follows from the inductive hypothesis and Lemma 4.3 ▮

Lemma 4.5 implies that

$$\sum_{e \in P'} a^{\tilde{L}_e(t^-)}/c(e) \le a^2 \sum_{e \in P'} a^{\tilde{\ell}_e(t^-)}/c(e).$$

Let $stage(k_{i_0})$ denote the value of the stage of the current phase k_{i_0} of for job i. By the definition of a stage

$$2^{stage(k_{i_0})-1} \le \sum_{e \in P'} a^{\tilde{L}_e(t^-)}/c(e)$$

On the other hand since P is currently active

$$weight_P(k^*) \le 2^{stage(k_{i_0})}.$$

Moreover, by the monotonicity of the load and the fact that testing the weight of a feasible subsetis computed after adding to its flow

$$\sum_{e \in P} a^{\tilde{h}^i_{P,e}(k^*)}/c(e) \le weight_P(k^*).$$

Combining the above inequalities yields that

$$\sum_{e \in P} a^{\tilde{h}^i_{P,e}(k^*)}/c(e) \le 4 \sum_{e \in P'} a^{\tilde{\ell}_e(t^+)}/c(e).$$

▮

We conclude

Corollary 4.6 For all t and e,

- $\sum_{e \in E} a^{\tilde{\ell}_e(t)} \le |E|/(1-\gamma)$
- $\tilde{\ell}_e(t) \le \beta$
- $\tilde{L}_e(t) \le \tilde{\ell}_e(t) + 2$

Proof of Theorem 3.1: By corollary 4.6, $\tilde{L}_e(t)$ is bounded by $O(\log n)$. Thus, the algorithm is $O(\log n)$ competitive. Next we bound the number of steps until it converges. Consider job i. We claim that after $c\beta \log n$ phases the value of *stage* increases by 1 or the job is already satisfied for an appropriate constant c. Otherwise there exists a feasible subset P whose weight is below 2^{stage} at all these phases and it is active at all of them. Clearly after the first phase $f_P^i \geq \min\{d_i, \Lambda^* c_P\}/n^2$. Moreover, the flow is increased by a factor of $1+1/\beta$ at each phase and hence it increases by factor of $(1+1/\beta)^{c\beta \log n} \geq 4n^2\beta$. Thus after the $c\beta \log n$ phases the value of the flow is at least $4\beta \cdot \min\{d_i, \Lambda^* c_P\}$. If the minimum in the above expression is d_i then job i is already over-satisfied and therefore its assignment was completed. On the other hand if the minimum is $\Lambda^* c_P$ then the server whose capacity is c_P has relative load $\tilde{L}_e \geq 4\beta \geq \beta+2$ which contradicts corollary 4.6.

Thus, unless the job it satisfied *stage* increases by 1 every $c\beta \log n$ phases. The minimum possible weight of a feasible subset is 1 since we normalized $\max_e c(e) = 1$. By corollary 4.6, the maximum weight of a feasible subset is $|E|/(1-\gamma)/\min_e c(e)$. Thus by the claim above the total number of rounds is at most

$$O(\beta \log n(\beta + \log \rho)) = O(\log^3 n)$$

∎

References

[AAF+93] Jim Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 623–631, May 1993.

[AAP93] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput competitive on-line routing. In *Proc. 34rd IEEE Symp. on Found. of Comp. Science*, pages 32–40. IEEE, November 1993.

[AAPW94] Baruch Awerbuch, Yossi Azar, Serge Plotkin, and Orli Waarts. Competitive routing of virtual circuits with unknown duration. In *Proc. 5'th ACM-SIAM Symp. on Discrete Algorithms*, 1994. to appear.

[ABFR93] Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rosén. Competitive non-preemptive call control. In *Proc. 5'th ACM-SIAM Symp. on Discrete Algorithms*, 1993. to appear.

[ABK92] Yossi Azar, Andrei Broder, and Anna Karlin. On-line load balancing. In *Proc. 33rd IEEE Symp. on Found. of Comp. Science*, pages 218–225, October 1992.

[ACS94] Baruch Awerbuch, Lenore Cowen, and Mark Smith. Efficient asynchronous distributed symmetry breaking. In *Proc. 26th ACM Symp. on Theory of Computing*, May 1994.

[AGLP89] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Found. of Comp. Science*, May 1989.

[AKP91] Baruch Awerbuch, Shay Kutten, and David Peleg. Deadlock-free routing. In *Proc. 10th ACM Symp. on Principles of Distrib.Computing*, 1991.

[AKP92] Baruch Awerbuch, Shay Kutten, and David Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 571–580, 1992.

[AKP+93] Yossi Azar, B. Kalyanasundaram, Serge Plotkin, K. Pruhs, and Orli Waarts. On-line load balancing of temporary tasks. In *Proc. Workshop on Algorithms and Data Structures*, pages 119–130, August 1993.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verifiction and hardness of approximation problems. In *Proc. 33rd IEEE Symp. on Found. of Comp. Science*, pages 14–23, October 1992.

[AM86] Baruch Awerbuch and Silvio Micali. Dynamic deadlock resolution protocols. In *Proc. 27th IEEE Symp. on Found. of Comp. Science*, October 1986.

[ANR92] Yossi Azar, Joseph Naor, and Raphael Rom. The competitiveness of on-line assignment. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 203–210, 1992.

[AS90] Baruch Awerbuch and Mike Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Found. of Comp. Science*, 1990.

[AS92] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs. In *Proc. 33rd IEEE Symp. on Found. of Comp. Science*, pages 2–13, October 1992.

[BBG83] J. Blazewick, J. Brzezinski, and G. Gambosi. Time-stamp approach to store-and forward deadlock prevention. In *IEEE Transactions on Communications*, volume 35, pages 490–495, May 1983. number 5.

[BC89] Barry J. Brachman and Samuel T. Chanson. A hierarchical solution for application level store-and-forward deadlock prevention. In *Proc. of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 25–32. ACM SIGCOMM, ACM, September 1989.

[BFKV92] Yair Bartal, Amos Fiat, Howard Karloff, and R. Vorha. New algorithms for an ancient scheduling problem. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[BGLR93] M. Bellare, S. Goldwasser, C. Lund, and A. Russel. Efficient probabilistically checkable proofs and applications to approximation. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 294–303, May 1993.

[BT87] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[CM83] K.M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 157–164, 1983.

[FGL$^+$91] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. In *Proc. 32nd IEEE Symp. on Found. of Comp. Science*, October 1991.

[GGK$^+$93] Juan Garay, Inder Gopal, Shay Kutten, Yishay Mansour, and Moti Yung. Efficient on-line call control algorithms. In *Proceedings of 2'nd Annual Israel Conference on Theory of Computing and Systems*, 1993.

[GPS87] A. V. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. In *Proc. 19th ACM Symp. on Theory of Computing*. ACM SIGACT, ACM, May 1987.

[Lin87] Nathan Linial. Locality as an obstacle to distributed computing. In 27^{th} *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[LMR88] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proc. 29th IEEE Symp. on Found. of Comp. Science*, pages 256–271. IEEE, October 1988.

[LN93] Michael Luby and Noam Nissan. A parallel approximation algorithm for positive linear programming. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 448–457, May 1993.

[LT94] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 302–311, Arlington, VA, January 1994.

[Lub86a] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Comput.*, 15(4):1036–1053, November 1986.

[Lub86b] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Comput.*, 15(4):1036–1053, November 1986.

[PST91] S. Plotkin, D. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proc. 32nd IEEE Symp. on Found. of Comp. Science*, 1991.

[PSW94] Cindy Phillips, Cliff Stein, and Joel Wein. Task scheduling in networks, center for advanced technology in telecommunications. Report 94-71, Polytechnic Unviversity, Brooklyn, NY, February 1994.

[Rag86] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *Proc. 27th IEEE Symp. on Found. of Comp. Science*, pages 10–18, May 1986.

[RT85] P. Raghavan and C.D. Thompson. Provably good routing in graphs: Regular arrays. In *Proc. 17th ACM Symp. on Theory of Computing*, May 1985.

[SM90] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *J. of the ACM*, 37:318 – 334, 1990.

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