Heterogeneous Resource Allocation under Degree Constraints

Olivier Beaumont, Lionel Eyraud-Dubois, Hejer Rejeb, Christopher Thraves

Heterogeneous Resource Allocation under Degree Constraints

Olivier Beaumont, Lionel Eyraud-Dubois and Hejer Rejeb
INRIA Bordeaux – Sud-Ouest, University of Bordeaux, LaBRI , France
Christopher Thraves
LADyR, GSyC, Universidad Rey Juan Carlos, Madrid, Spain

***Abstract*—In this paper, we consider the problem of assigning a set of clients with demands to a set of servers with capacities and degree constraints. The goal is to find an allocation such that the number of clients assigned to a server is smaller than the server's degree and their overall demand is smaller than the server's capacity, while maximizing the overall throughput. This problem has several natural applications in the context of independent tasks scheduling or virtual machines allocation. We consider both the *offline* (when clients are known beforehand) and the *online* (when clients can join and leave the system at any time) versions of the problem. We first show that the degree constraint on the maximal number of clients that a server can handle is realistic in many contexts. Then, our main contribution is to prove that even if it makes the allocation problem more difficult (NP-Complete), a very small additive resource augmentation on the servers degree is enough to find in polynomial time a solution that achieves at least the optimal throughput. After a set of theoretical results on the complexity of the offline and online versions of the problem, we propose several other greedy heuristics to solve the online problem and we compare the *performance* (in terms of throughput) and the *cost* (in terms of disconnections and reconnections) of proposed algorithms through a set of extensive simulation results.**

I. Introduction

In a client-server computing platform, where servers have capacity and degree constraints and clients have demands, we consider the problem of finding an allocation of clients to servers such that each server's degree and capacity constraints are satisfied while fulfilled demand is maximized. For instance, this models the problem of scheduling a very large number of identical tasks on a server-client platform [19]. Initially, several servers hold or generate tasks that are transferred and processed by clients. The goal is to maximize the overall throughput achieved using this platform, *i.e.,* the (fractional) number of tasks that can be processed within one time unit. Since QoS mechanisms for bandwidth control have to be used in order to cope with the heterogeneity of the clients [11], [20], the degree constraint is related to the maximal number of TCP connections that a server can handle using QoS and the capacity of the server is defined as its overall outgoing bandwidth. This resource allocation problem also has applications in the context of Cloud Computing [31], [28], [12]. In this case, servers represent physical machines and clients represent services, which can be deployed on the servers by using one or more virtual machines (VMs). Each service comes with its demand and a physical machine can host at most a given number of virtual machines (see Section II). In this context, the resource allocation problem can be used to find the allocation that allows the maximal fraction (the same for all services) that can be processed on a set of physical machines.

The work of Olivier Beaumont, Lionel Eyraud-Dubois and Hejer Rejeb was partially supported by French ANR project Alpage. The work of Christopher Thraves was supported in part by Spanish MICINN grant Juan de la Cierva.

In the general setting, each server $\mathcal{S}_j$ is characterized by its *capacity* b_j (*i.e.,* the quantity of data that it can send, or the number of flops that it can process during one time-unit, depending on the context) and its *degree* d_j (*i.e.,* the maximal number of open TCP connections, or the number of virtual machines that it can handle simultaneously). On the other hand, each client $\mathcal{C}_i$ is characterized by its *demand* w_i (*i.e.,* the number of tasks that it can process during one time-unit, or its computational demand per time unit). Our goal is to build a bipartite graph between servers and clients, so that capacity, degree and demand constraints are satisfied.

Formally, let us denote by w_i^j the capacity allocated by server $\mathcal{S}_j$ to client $\mathcal{C}_i$. Then, a valid allocation must satisfy the following conditions

$$\forall j \qquad \sum_i w_i^j \leq b_j \qquad (1)$$
$$\forall j \qquad \text{Card}\{i:\ w_i^j > 0\} \leq d_j \qquad (2)$$
$$\forall i \qquad \sum_j w_i^j \leq w_i \qquad (3)$$

where Equation (1) refers to the *capacity constraint* at server $\mathcal{S}_j$, Equation (2) refers to the *degree constraint* at server $\mathcal{S}_j$ and Equation (3) refers to the *demand constraint* at client $\mathcal{C}_i$.

Therefore, as introduced in [6], *Maximize-Throughput-Bounded-Degree* (MTBD) problem is defined as follows:

$$\textit{Maximize} \sum_j \sum_i w_i^j \textit{ under constraints (1), (2) and (3).}$$

Due to the dynamic nature of the clients participating into a large scale volunteer computation or the virtual machines running on a Cloud, it is both interesting to study MTBD when the set of clients is known in advance or when clients join and leave the system at any moment, the *offline* [6] and *online* scenarios [7] respectively. In the online context, it makes sense to compare the algorithms according to their cost, the number of changes in the allocation induced by a client arrival or departure, and their performance, the achieved throughput, as discussed in Section III.

The rest of the paper is organized as follows. In Section II, we present the applications of MTBD to the scheduling of independent tasks in the context of large scale volunteer computing platforms and to the allocation of services to physical machines in the context of Cloud computing. In Section III, we justify the used model, we formalize the allocation problem and we visit the results of this paper. In Section IV, we prove that MTBD is NP-Complete in the strong sense but that a small additive resource augmentation (of 1) on the servers degrees is enough to find in polynomial time a solution that achieves at least the optimal throughput. Then, we consider in Section V the more realistic setting where the set of clients is not known in advance but clients rather join and leave the system at any time, *i.e.,* the online version of MTBD. We prove that no fully online algorithm (where only one change is allowed for each event) can achieve a constant approximation ratio, whatever the resource augmentation on servers degrees. Then, we prove that it is possible to maintain the optimal solution at the cost of at most 4 changes per server each time a new node joins or leaves the system. At last, we propose in Section VI several other greedy heuristics to solve the online problem and we compare performance in terms of throughput, and cost in terms of disconnections and reconnections, of proposed algorithms through a set of extensive simulation results based on realistic datasets. Concluding remarks are given in Section VII.

II. Applications and Related Work

A. Independent Tasks Scheduling on Large Scale Platforms

Scheduling computational tasks on a given set of processors is a key issue for high-performance computing, especially in the context of large scale computing platforms such as BOINC [2] or Folding@home [23]. These platforms are characterized by their large scale, their heterogeneity and the performance variations of the participating resources. These characteristics strongly influence the set of applications that can be executed on these platforms. First, the running time of the application has to be large enough to benefit from the platform scale, and to minimize the influence of start-up times due to sophisticated middleware. Second, the applications should consist of many small independent tasks in order to minimize the influence of variations in resource performances and to limit the impact of resource failures. From a scheduling point of view, the set of applications that can be efficiently executed is therefore restricted, and we can concentrate on "embarrassingly parallel" applications consisting in many independent tasks.

Even in the context of independent tasks on heterogeneous resources [18], makespan minimization, *i.e.,* minimizing the time to process a given number of tasks, is intractable. An idea to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective. Instead of aiming at the absolute minimization of the execution time, it is generally more efficient to consider asymptotic optimality only (when the number of tasks is large). The goal is then to optimize the throughput. *i.e.,* the fractional number of tasks that can be processed in one time-unit once steady-state has been reached. This approach has been pioneered by Bertsimas and Gamarnik [10] and has been extended to task scheduling [4] and collective communications [5]. Steady-state scheduling allows to relax the scheduling problem in many ways, and aims at characterizing the activity of each resource during each time-unit by deciding which (rational) fraction of time is spent sending and receiving tasks and to which client tasks are delegated, that is to focus on resource allocation rather than scheduling.

Independent task scheduling on large scale computing platforms can be modeled using MTBD problem. Following MTBD notation, each server $\mathcal{S}_j$ is characterized by its capacity b_j, the number of tasks it can send during one time-unit, and its maximal degree d_j, the number of open connections that it can handle simultaneously. On the other hand, each volunteer is considered as a client $\mathcal{C}_i$ and it's characterized by its demand w_i, the number of tasks it can handle during one time- unit. In this case, client's capacity w_i encompasses both its processing and communication capacities. More specifically, if comp_i denotes the number of tasks $\mathcal{C}_i$ can process during one time-unit, and comm_i denotes the number of tasks it can receive during one time-unit, then we set $w_i = \min(\mathrm{comp}_i, \mathrm{comm}_i)$.

To model contentions, we rely on the bounded multi-port model, that has already been advocated by Hong et al. [19] for independent tasks distribution on heterogeneous platforms. In this model, a server $\mathcal{S}_j$ can serve any number of clients simultaneously, each using a bandwidth $w'_i \leq w_i$ provided that its outgoing bandwidth is not exceeded, *i.e.,* $\sum_i w'_i \leq b_j$. This corresponds to modern network infrastructure, where each communication is associated to a TCP connection.

This model strongly differs from the traditional one-port model used in scheduling literature, where connections are made in exclusive mode: the server can communicate with a single client at any time-step. Previous results obtained in steady-state scheduling of independent tasks [4] have been obtained under this model, which is easier to implement. For instance, Saif and Parashar [25] report experimental evidence that achieving the performances of bounded multi-port model may be difficult, since asynchronous sends become serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular implementations of MPI, the message-passing standard: MPICH on Linux clusters and IBM MPI on the SP2. Nevertheless, in the context of large scale platforms, the networking heterogeneity ratio may be high, and it is unrealistic to assume that a 100MB/s server may be kept busy for 10 seconds while communicating a 1MB data file to a 100kB/s DSL node. Therefore, in our context, all connections must directly be handled at TCP level, without using high level communication libraries.

It is worth noting that at TCP level, several QoS mechanisms enable a prescribed sharing of the bandwidth [11], [20]. In particular, it is possible to handle simultaneously several connections and to fix the bandwidth allocated to each connection. In our context, these mechanisms are particularly useful since w_i encompasses both processing and communication capabilities of $\mathcal{C}_i$ and therefore, the bandwidth allocated to the connection between $\mathcal{S}_j$ and $\mathcal{C}_i$ may be lower than both b_j and w_i. Nevertheless, handling a large number of connections at server $\mathcal{S}_j$ with prescribed bandwidths consumes a lot of kernel resources, and it may therefore be difficult to reach b_j by aggregating a large number of connections. In order to circumvent this problem, we introduce another parameter d_j in the bounded multi-port model, that represents the maximal number of connections that can be simultaneously opened at server $\mathcal{S}_j$.

Therefore, the model we propose encompasses the benefits of both the bounded multi-port model (by setting $\forall i, d_i = +\infty$) and the one-port model (by setting $\forall i, d_i = 1$). It enables several communications to take place simultaneously, what is compulsory in the context of large scale distributed platforms, and practical implementation is achieved by using TCP QoS mechanisms and by bounding the maximal number of connections.

B. *Virtualization in Cloud Computing Platforms*

Cloud Computing [31], [3] has recently emerged as a new paradigm for service providing over the Internet. Among the challenges associated to Cloud Computing is the efficient use of virtualization technologies such as Xen [30], KVM [22] and VMware [29] and the migration of Virtual Machines (VMs) onto Physical Machines (PMs). Using virtualization, it is possible to run several Virtual Machines on top of a given Physical Machine. Since each VM hosts its complete software stack (Operating System, Middleware, Application), it is possible to migrate VMs from a PM to another. The ability to move virtual machines is crucial in order to achieve good load balancing [28], [12] in a dynamic context where VMs are added and removed from the system. It is also crucial for energy minimization [9], [8] in order to determine if some PM can be switched off.

The mapping problem of services having heterogeneous computing demands onto PM having heterogeneous capacities can be modeled using MTBD. In this context, each physical machine $\mathcal{S}_j$ is characterized by its computing capacity b_j (*i.e.,* the number of flops it can process during one time-unit) and its maximal degree d_j (*i.e.,* the number of different VMs that it can handle simultaneously, given that each VM comes with its complete software stack). On the other hand, each service $\mathcal{C}_i$ is characterized by its demand w_i (*i.e.,* its overall processing demand during one time-unit). Then, a valid solution of MTBD provides a valid mapping of services onto PMs. The online version of MTBD corresponds to the case where services are added to or removed from the Cloud, or to the case when their demands change over time. In this case, the property that we prove in Section V stating that the online algorithm we propose bounds the number of changes on any PM is crucial as it enables to bound the number of VM migrations.

C. Related Works

A closely related problem is Bin Packing with Splittable Items and Cardinality Constraints, where the goal is to pack a given set of items in as few bins as possible. The items may be split, but each bin may contain at most k items or pieces of items. This is very close to the problem we consider, with two main differences: in our case the number of servers (corresponding to bins) is fixed in advance, and the goal is to maximize the total used capacity of the servers (corresponding to the total packed size), whereas the goal in Bin Packing is to minimize the number of bins used to pack all the items (corresponding to the number of used servers). Furthermore, we consider heterogeneous servers (what would correspond to bins with heterogeneous capacities and heterogeneous cardinality constraints).

Bin Packing with splittable items and cardinality constraints was introduced in the context of memory allocation in parallel processors by Chung et al. [14], who considered the special case when $k = 2$. They showed that even in this simple case, this problem is NP-Complete, and they proposed a $3/2$-approximation algorithm. Epstein and van Stee [16] showed that Bin Packing with splittable items and cardinality constraints is NP-Hard for any fixed value of k, and that the simple NEXT-FIT algorithm achieves an approximation ratio of $2-1/k$. They also design a PTAS and a dual PTAS [15] for the general case where k is a constant.

Other related problems were introduced by Shachnai et al. [27]. They propose to model the size of an item as increasing when it is split and to ask for a global bound on the number of fragmentations. The authors prove that this problem does not admit a PTAS, and provide a dual PTAS and an asymptotic PTAS. In a multiprocessor scheduling context, another related problem is scheduling with allotment and parallelism constraints [26]. The goal is to schedule a certain number of tasks, where each task comes with a bound on the number of machines that can process it simultaneously and a bound on the overall number of machines that can participate in its execution. This problem can also be seen as a splittable packing problem, but this time with a bound k_i on the number of times an item can be split. In [26], an approximation algorithm of ratio $\max_i(1+1/k_i)$ is presented.

In a related context, resource augmentation techniques have already been successfully applied to online scheduling problems [21], [24], [13], [17] in order to prove optimality or good approximation ratio. More precisely, it has been established that several well-known online algorithms, that have poor performance from an absolute worst-case perspective, are optimal for these problems when allowed moderately more resources [24]. In this paper, we consider a slightly different context, since the off-line solution already requires resource augmentation on the servers degrees. We prove that it is possible in the on-line context to maintain at relatively low cost a solution that achieves the optimal throughput with the same resource augmentation as in the off-line context.

III. Model and Summary of Results

A. Platform Model and Maintenance Costs

Let us denote by b_j the capacity of server $\mathcal{S}_j$ and by d_j the maximal number of clients that it can handle simultaneously (its degree). The capacity of client $\mathcal{C}_i$ is denoted by w_i. All capacities are normalized and expressed in terms of (fractional) number per time-unit. Moreover, let us denote by w_i^j the allocated value by server $\mathcal{S}_j$ to client $\mathcal{C}_i$. Then, we have noticed in the introduction that MTBD can be expressed as a maximization problem under constraints (1), (2) and (3).

In the online version of MTBD, we introduce the notion of (virtual) rounds. A new round starts when a client joins or leaves the system, so that no duration is associated to a round. We denote by $\mathcal{LC}^t$ the set of clients present at round t (with their respective capacities). Client $\mathcal{C}$ *joins* (resp. *leaves*) the system at round t if $\mathcal{C} \in \mathcal{LC}^t \backslash \mathcal{LC}^{t-1}$ (resp. $\mathcal{C} \in \mathcal{LC}^{t-1} \backslash \mathcal{LC}^t$). The arrival or departure of a client can therefore only take place at the beginning of a round and $\forall t,\ |\mathcal{LC}^t \backslash \mathcal{LC}^{t-1}| + |\mathcal{LC}^{t-1} \backslash \mathcal{LC}^t| \leq 1$. Let us denote by $\mathcal{LS}$ the set of servers (with their respective capacity and degree constraints).

Solving the online version of MTBD comes into two flavors. First, one may want to maintain the optimal throughput at a minimal cost in terms of changes in existing connections between clients and servers. Second, one may want to achieve a minimal number of changes in existing connections at each server and to obtain the best possible throughput. In order to compare online solutions, we need to define precisely the cost of changing the existing allocation of clients to servers due to the arrival or departure of a new client.

Let us denote by $w_i^j(t)$ the allocated value by server $\mathcal{S}_j$ to client $\mathcal{C}_i$ at round t. We say that client $\mathcal{C}_i$ is *connected* to server $\mathcal{S}_j$ at round t if $w_i^j(t) > 0$. We say that the connection between server $\mathcal{S}_j$ and client $\mathcal{C}_i$ *changes* at round t if $w_i^j(t-1) \neq w_i^j(t)$, and we denote by $\mathcal{N}_j^t = |\{i, w_i^j(t-1) \neq w_i^j(t)\}|$ the number of changes occurring at server $\mathcal{S}_j$ at round t.

This notion of change covers three different situations. If $w_i^j(t-1) = 0$ and $w_i^j(t) > 0$, then this change corresponds to a *new connection* to the server. Symmetrically, if $w_i^j(t-1) > 0$ and $w_i^j(t) = 0$, then client $\mathcal{C}_i$ was *disconnected* from server $\mathcal{S}_j$. Finally, if both $w_i^j(t-1)$ and $w_i^j(t)$ are positive, this corresponds to a *change in the allocation* between client $\mathcal{C}_i$ and server $\mathcal{S}_j$.

In the context of independent tasks scheduling, since we rely on complex QoS mechanisms to achieve the prescribed bandwidth sharing between clients and servers, any change in bandwidth allocation induces some cost. If a new client connects to a server, a new TCP connection needs to be opened, what also induces some cost. On the other hand, all modifications in bandwidth connections made by the different server nodes can take place in parallel. Similarly, in the context of Virtualization, adding or removing a VM from a PM induces some cost, due to migration. On the other hand, the different migration operations can be done in parallel.

Therefore, we introduce the following definition to measure and compare algorithms that solve online MTBD.

Definition 3.1: Let $\mathcal{A}$ be an algorithm solving the online version of MTBD. $\mathcal{A}$ induces l *changes* in connections per round if

$$\max_t \max_{\mathcal{S}_j \in \mathcal{LS}} \mathcal{N}_j^t \quad = l.$$

B. Main Results

In the offline context, we first prove that MTBD is NP-Complete due to the degree constraint at the server nodes. On the other hand, we propose a sophisticated polynomial time algorithm, based on a slight resource augmentation to solve MTBD. More specifically, we prove that, if d_j denotes the degree constraint at node $\mathcal{S}_j$, then the throughput achieved using this algorithm and degree d_j+1 is at least the same as the optimal one with degree d_j (Theorem 4.4 in Section IV).

In the online context, the first result we present is that no online algorithm with cost less than 2 can achieve a constant approximation ratio, whatever the resource augmentation on the degree (Theorem 5.1 in Section V).

The second result presented in the online context shows that there exists a polynomial time online algorithm whose cost is at most 4 (see Theorem 5.5), with a resource augmentation of 1 (Lemma 5.2), and that maintains the optimal throughput at any round. Indeed, we know that Algorithm SEQ (Algorithm 1 in Section IV) provides at least the optimal throughput allowing the smallest possible additive resource augmentation $\alpha = 1$. Hence, we transform algorithm SEQ into an online algorithm, and we use it to solve the online version of MTBD. The online version of Algorithm SEQ is called OSEQ (see Algorithm 2 in section V).

Therefore, in our context, maintaining the optimal throughput (with resource augmentation) is not more expensive, in terms of online cost and up to a constant ratio smaller than 2, than maintaining a constant approximation ratio of the optimal throughput.

IV. OFFLINE CASE ANALYSIS

We start the study of MTBD with the analysis of its complexity. Let us consider the corresponding decision problem, *Throughput-Bounded-Degree-Dec (*TBD-DEC*)*, where the goal is to decide whether a throughput K can be achieved given a set of servers and a set of clients.

Lemma 4.1: TBD-DEC is NP-Complete in the strong sense.

Proof: To prove this result, we use a reduction to the 3-Partition problem [18]. Indeed, let us consider an instance of 3-Partition consisting of $3m$ items a_i such that $\sum a_i = mB$ and $\forall i,\ \frac{B}{4} < a_i < \frac{B}{2}$ and let us set $\forall j,\ \ d_j = 3,\ \ b_j = B$, $n = 3m$, $\forall i,\ \ w_i = a_i$ and $K = mB$. Since the overall out degree of the servers is at most $3m$ and since all $3m$ clients must be used in order to reach throughput mB, each server must be connected to exactly 3 clients and no client should be connected to more than one server. Since the overall capacity of the servers is $m \times B$, each server must be connected to 3 clients whose aggregated capacity is exactly B, what achieves the NP-Completeness proof. ■

A. A Resource Augmentation Based Algorithm

Let us now present Algorithm SEQ, that relies on resource augmentation to provide a solution to MTBD problem. Due to the mentioned resource augmentation, SEQ outputs a *non-valid* solution in the sense that the number of clients allocated to a server $\mathcal{S}_j$ may be d_j+1 instead of d_j as stated in constraint (2). SEQ is described precisely in Algorithm 1.

In the following, we will consider lists of clients sorted by increasing capacities, and if $\mathcal{LC} = \{\mathcal{C}_i\}$ denotes such a list, we will denote by $\mathcal{LC}(l,k) = \sum_{i=l}^{k} w_i$ the sum of the capacities of the clients between $\mathcal{C}_l$ and $\mathcal{C}_k$, both of them included.

Throughout a whole computation, Algorithm SEQ maintains an ordered list of remaining clients. At each step, it picks up a server $\mathcal{S}_j$ arbitrarily and goes through the list to find a suitable set of clients for this server. A suitable set of clients is a set of d_j+1 consecutive clients in the ordered list, called an *interval* of length d_j+1, with total capacity at least b_j, and such that the sum of the capacities of the first d_j clients is less than the capacity b_j of the server. These constraints ensure that the whole capacity and the maximum out-degree of the server are used. If such an interval $[l, l+d_j]$ exists (there may be several, but any of them does the trick), Algorithm SEQ selects the rightmost one, *i.e.,* the interval $[l, l+d_j]$ such that $\mathcal{LC}(l, l+d_j-1) < b_j$ and $\mathcal{LC}(l+1, l+d_j) \geq b_j$.

This choice ensures that clients $\mathcal{C}_l, \mathcal{C}_{l+1}, \ldots, \mathcal{C}_{l+d_j-1}$ are served completely by server $\mathcal{S}_j$ by setting $w_i^j = w_i$ for all $i \in \{l, l+1, \ldots, l+d_j-1\}$. If the total capacity of the interval exceeds b_j, the last client can only be partially served. In that case, client $\mathcal{C}_{l+d_j}$ is served with capacity $w_{l+d_j}^j = b_j - \mathcal{LC}(l, l+d_j-1)$ and then reinserted in the list of remaining clients with new capacity w'_{l+d_j} equal to $\mathcal{LC}(l, l+d_j) - b_j$. In that case, client $\mathcal{C}_{l+d_j}$ will be connected to more than one server in the final solution. The list of clients is then updated and the algorithm goes on with the next server.

With respect to the ordering of the updated list of clients, let us point out that the choice of the rightmost interval ensures an ordering property. That is: *the position of the modified client $\mathcal{C}_{l+d_j}$ in the sorted list remains the same*. Indeed, $\mathcal{C}_{l+d_j}$'s new capacity is equal to $w'_{l+d_j} = \mathcal{LC}(l, l+d_j) - b_j = w_l + \mathcal{LC}(l+1, l+d_j) - b_j$, and then the constraint $\mathcal{LC}(l+1, l+d_j) \geq b_j$ ensures that $w'_{l+d_j} \geq w_l$. Hence, $w_{l-1} \leq w'_{l+d_j} \leq w_{l+d_j+1}$, and thus the updated list of clients is already ordered. This property will be crucial in Section V. Indeed, among all possible valid intervals that can be allocated to $\mathcal{S}_j$, only the rightmost one produces an allocation that does not require many changes when a client joins or leaves the system (see Section VI).

It may happen that there exists no suitable interval for two reasons. The first one is that any set of d_j+1 clients has not enough capacity to use all the bandwidth b_j (*i.e.,* the overall capacity of the d_j+1 largest clients is not big enough). In this case, SEQ allocates to server $\mathcal{S}_j$ the d_j largest clients (the last d_j clients in the ordered list). Note that in that case, SEQ would be allowed to allocate one more client to server $\mathcal{S}_j$. But no valid solution could allocate more bandwidth to this server, and the extra connection may actually be useful later on.

On the other hand, if any set of d_j clients has overall capacity larger than b_j (*i.e.,* the overall capacity of the d_j smallest clients is already too large), then the algorithm simply allocates the k smallest clients, where k is the smallest index such that $\mathcal{LC}(1, k) \geq b_j$. In this case also, the last client may be split, and its remaining capacity will be $\mathcal{LC}(1, k) - b_j$ (clearly, the new client is the smallest one in the list and is reinserted at the same place in this case also).

Algorithm 1 Algorithm SEQ

Set $\mathcal{S} = \{\mathcal{S}_j\}_{j=1}^m$ and $\mathcal{LC} = \text{sort}(\{\mathcal{C}_i\}_{i=1}^n)$;
Set $\mathcal{A} = \{\mathcal{A}_j = \{\emptyset\}\}_{j=1}^m$ and $j = 1$;
for $j = 1$ to m **do**
  if $\exists l$ such that $\mathcal{LC}(l, l+d_j-1) < b_j$ and $\mathcal{LC}(l, l+d_j) \geq b_j$ **then**
    Pick l s.t. $\mathcal{LC}(l, l+d-1) < b$ and $\mathcal{LC}(l+1, l+d) \geq b$
    Split $\mathcal{C}_{l+d_j}$ in $\mathcal{C}'_{l+d_j}$ and $\mathcal{C}''_{l+d_j}$ with $w_{l+d_j} = w'_{l+d_j} + w''_{l+d_j}$ and $w''_{l+d_j} = b_j - \mathcal{LC}(l, l+d_j-1)$
    Set $\mathcal{A}_j = \{\mathcal{C}_l, \mathcal{C}_{l+1}, \ldots, \mathcal{C}_{l+d_j-1}, \mathcal{C}''_{l+d_j}\}$
    Remove $\mathcal{C}_l, \mathcal{C}_{l+1}, \ldots, \mathcal{C}_{l+d_j}$ and insert $\mathcal{C}'_{l+d_j}$ in $\mathcal{LC}$
  end if
  if $\mathcal{LC}(1, d_j) \geq b_j$ **then**
    Search for the smallest k such that $\mathcal{LC}(1, k) \geq b_j$
    Split $\mathcal{C}_k$ in $\mathcal{C}'_k$ and $\mathcal{C}''_k$ with $w_k = w'_k + w''_k$ and $w''_k = b_j - \mathcal{LC}(1, k-1)$
    Set $\mathcal{A}_j = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{k-1}, \mathcal{C}''_k\}$
    Remove $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ and insert $\mathcal{C}'_k$ in $\mathcal{LC}$
  end if
  if $\mathcal{LC}(n-d_j, n) < b_j$ **then**
    Set $\mathcal{A}_j = \{\mathcal{C}_{n-d_j+1}, \mathcal{C}_{n-d_j+2}, \ldots, \mathcal{C}_n\}$
    Remove $\mathcal{C}_{n-d_j+1}, \mathcal{C}_{n-d_j+2}, \ldots, \mathcal{C}_n$ from $\mathcal{LC}$
  end if
end for
RETURN $\mathcal{A} = \{\mathcal{A}_j\}_{j=1}^m$

B. Approximation Results

Let us now prove that the throughput allocated by Algorithm SEQ is at least as much as the throughput provided by any valid solution. For the sake of simplicity, we consider that the length of the list of clients remains n during the execution of the algorithm. Without loss of generality, we assume that removed clients will thus be considered as 0-capacity clients and reinserted at the beginning of the list. To prove the result, we need to introduce an order $\preceq$ between two lists of clients. Intuitively, if two lists of clients $\mathcal{LC}$ and $\mathcal{R}$ satisfy $\mathcal{LC} \preceq \mathcal{R}$, then, whatever the remaining servers, list $\mathcal{LC}$ will be easier to allocate than list $\mathcal{R}$.

Definition 4.2: Let $\mathcal{LC}$ and $\mathcal{R}$ be two lists of clients with the same length n and ordered by increasing capacities. We say that $\mathcal{LC}$ is *easier* than $\mathcal{R}$ (denoted by $\mathcal{LC} \preceq \mathcal{R}$), if

$$\forall k \leq n, \quad \mathcal{LC}(1, k) \leq \mathcal{R}(1, k)$$

Let us now consider a given step of the algorithm SEQ in which the considered server has capacity b and degree d. Let $\mathcal{LC}$ and $\mathcal{R}$ be two lists of clients. The application of this step of algorithm SEQ to the list $\mathcal{LC}$ leads to a remaining list $\mathcal{LC}'$. Similarly, a valid allocation[1] of this server to the list $\mathcal{R}$ yields a list of remaining clients $\mathcal{R}'$. The following lemma states that this atomic operation preserves the order $\preceq$.

Lemma 4.3: If $\mathcal{LC} \preceq \mathcal{R}$, and $\mathcal{LC}'$ and $\mathcal{R}'$ are obtained from $\mathcal{LC}$ and $\mathcal{R}$ as described above, then $\mathcal{LC}' \preceq \mathcal{R}'$.

Proof: We begin by proving two lower bounds for $\mathcal{R}'(1,k)$. Since $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a valid allocation, there exists a set $C \subseteq [1,n]$ of chosen clients, and assigned values v_i for $i \in C$ such that: Card$(C) \leq d$, $\forall i, v_i \leq \mathcal{R}_i$ (where $\mathcal{R}_i$ denotes the capacity of the i^{th} client in $\mathcal{R}$), and $\sum v_i \leq b$. There also exists a sorting permutation σ such that $\mathcal{R}'_{\sigma(i)} = \mathcal{R}_i$ if $i \notin C$, and $\mathcal{R}'_{\sigma(i)} = \mathcal{R}_i - v_i$ if $i \in C$. We can then write $\mathcal{R}'(1,k)$ in two different ways,

$$\begin{aligned}\mathcal{R}'(1,k) &= \sum_{i:i\notin C \wedge \sigma(i)\leq k} \mathcal{R}_i + \sum_{i:i\in C \wedge \sigma(i)\leq k} \mathcal{R}_i - v_i \\ &= \sum_{i:\sigma(i)\leq k} \mathcal{R}_i - \sum_{i:i\in C \wedge \sigma(i)\leq k} v_i\end{aligned}$$

For $k > d$, since there are at least $k-d$ indexes i such that $i \notin c \wedge \sigma(i) \leq k$, and since $\mathcal{R}(1,k-d)$ is the sum of the $k-d$ smallest $\mathcal{R}_i$ values, then $\sum_{i:i\notin c\wedge\sigma(i)\leq k} \mathcal{R}_i \geq \mathcal{R}(1,k-d)$. Together with $\mathcal{R}_i - v_i \geq 0$, we obtain the first upper bound

$$\mathcal{R}'(1,k) \geq \mathcal{R}(1,k-d) \qquad \forall k > d. \tag{4}$$

Similarly, since there are k indexes i such that $\sigma(i) \leq k$, then $\sum_{i:\sigma(i)\leq k} \mathcal{R}_i \geq \mathcal{R}(1,k)$. Together with $\sum_{i\in C} v_i \leq b$, we obtain the second upper bound

$$\mathcal{R}'(1,k) \geq \mathcal{R}(1,k) - b. \tag{5}$$

To complete the proof, we need to evaluate $\mathcal{LC}'(1,k)$. Since we identified three main situations when adding a server, we evaluate $\mathcal{LC}'(1,k)$ in each possible situation.

Case 1 $\exists l$ such that $\mathcal{LC}(l, l+d-1) < b$ and $\mathcal{LC}(l, l+d) \geq b$: In this case (see lines 4 to 8 in Algorithm 1) the algorithm allocates completely clients $\mathcal{C}_l, \mathcal{C}_{l+1}, \ldots, \mathcal{C}_{l+d-1}$ to $\mathcal{S}$ and only partially client $\mathcal{C}_{l+d}$, whose remaining capacity is w'_{l+d}. The first d clients of the list $\mathcal{LC}'$ will thus have zero capacity, and $\mathcal{C}'_{l+d}$ will be reinserted at the same position as pointed out earlier. Then, the updated list $\mathcal{LC}'$ is equal to

$$\{\mathcal{C}'_l = 0, \ldots, \mathcal{C}'_{l+d-1} = 0, \mathcal{C}_1, \ldots, \mathcal{C}_{l-1}, \\ \mathcal{C}'_{l+d} = \mathcal{LC}(l, l+d) - b, \mathcal{C}_{l+d+1}, \ldots, \mathcal{C}_n\}.$$

[1]Remember that the number of clients allocated to the server may be as high as $d+1$ with SEQ, whereas it is limited to d in the valid solution.

Then, for $k \leq d$, $\mathcal{LC}'(1,k)$ is a sum over the completely allocated reinserted clients, and thus $\mathcal{LC}'(1,k) = 0$. For the second interval $d < k \leq l-1+d$, $\mathcal{LC}'(1,k)$ is a sum of the first $k-d$ capacities in $\mathcal{LC}$, since they were shifted by d positions (due to the insertion of d clients at the beginning of the list), and so $\mathcal{LC}'(1,k) = \mathcal{LC}(1,k-d)$. If the interval includes one more client, *i.e.,* $d < k \leq l+d$, the sum is the same than in the previous interval, but the last element in the sum is replaced by the size of the split client that has been inserted, $\mathcal{LC}'(1,k) = \mathcal{LC}(1,k-d-1) + w'_{l+d}$. Finally when $l+d < k$, the sum is equal to the sum in the original list, decreased by the total capacity allocated to $\mathcal{S}$, $\mathcal{LC}'(1,k) = \mathcal{LC}(1,k) - b$.

Now, using Equations (4) and (5), and the fact that $\mathcal{LC} \preceq \mathcal{R}$, we have:

$$\begin{aligned}\mathcal{LC}'(1,k) &= 0 \leq \mathcal{R}'(1,k) \quad \text{ for } \quad k \leq d \\ \mathcal{LC}'(1,k) &= \mathcal{LC}(1,k-d-1) + w'_{l+d} \\ &\leq \mathcal{LC}(1,k-d) \leq \mathcal{R}(1,k-d) \\ &\leq \mathcal{R}'(1,k) \quad \text{ for } \quad d < k \leq l+d \\ \mathcal{LC}'(1,k) &= \mathcal{LC}(1,k) - b \leq \mathcal{R}(1,k) - b \\ &\leq \mathcal{R}'(1,k) \quad \text{ for } \quad l+d < k.\end{aligned}$$

Case 2 $A(1,d) \geq b$: In this case (see lines 9 to 14 in Algorithm 1), since SEQ uses the first $l \leq d$ clients, there is no reordering of the list. The new list $\mathcal{LC}'$ can therefore be written as $\{\mathcal{C}'_1, \ldots, \mathcal{C}'_{l-1}, \mathcal{C}'_l, \mathcal{C}_{l+1}, \ldots, \mathcal{C}_n\}$, where $\mathcal{C}'_i$ has zero capacity for $i < l$. Moreover, since the overall allocated capacity is equal to b, then $\mathcal{LC}'(1,k) = 0$ when $k \leq l-1$ and $\mathcal{LC}'(1,k) = \mathcal{LC}(1,k) - b$ for $k > l-1$. Hence, Equation (5) combined to $\mathcal{LC} \preceq \mathcal{R}$ leads to $\mathcal{LC}'(1,k) \leq \mathcal{R}'(1,k)$.

Case 3 $A(n-d,n) < b$: In this case (see lines 15 to 18 in Algorithm 1), SEQ allocates completely the d last clients to $\mathcal{S}$, and therefore all reinserted clients $\mathcal{C}'_i$ will have zero capacity and will be reinserted at the beginning of the list. The new list $\mathcal{LC}'$ can therefore be written as $\{\mathcal{C}'_{n-d+1}, \ldots, \mathcal{C}'_n, \mathcal{C}_1, \ldots, \mathcal{C}_{n-d}\}$. Therefore, $\mathcal{LC}'(1,k) = 0$ when $k \leq d$ and $\mathcal{LC}'(1,k) = \mathcal{LC}(1,k-(d+1))$ for $k > d$. Once again, Equation (4) combined with $\mathcal{LC} \preceq \mathcal{R}$ leads to $\mathcal{LC}'(1,k) \leq \mathcal{R}'(1,k)$. ∎

We can now state and prove the main result of this section.

Theorem 4.4: Let $\mathcal{A}$ be any valid solution of an instance I, and SEQ(I) be the solution given by algorithm SEQ. Then the throughput of SEQ(I) is at least as much as the throughput of $\mathcal{A}$.

Proof: Using the ordering $\preceq$ and Lemma 4.3, the proof of Theorem 4.4 becomes straightforward. Indeed, let us start with the initial list of clients $\mathcal{LC}_0 = \mathcal{LR}_0 = \mathcal{L}$

and let us denote by $\mathcal{LC}_j$ (resp., $\mathcal{LR}_j$) the list of remaining clients after the j-th first steps of Algorithm SEQ (resp., not fully allocated to servers $\mathcal{S}_1, \dots, \mathcal{S}_j$ in the valid allocation $\mathcal{A}$).

Then, a trivial induction, based of successive applications of Lemma 4.3 proves that $\mathcal{LC}_m \preceq \mathcal{LR}_m$. This means that $\forall k \leq n, \mathcal{LC}_m(1,k) \leq \mathcal{LR}_m(1,k)$, and in particular $\mathcal{LC}_m(1,n) \leq \mathcal{LR}_m(1,n)$, where $\mathcal{LC}_m(1,n)$ and $\mathcal{LR}_m(1,n)$ respectively denote the overall unused capacity of the clients in the solution computed respectively by SEQ and $\mathcal{A}$. Hence, the throughput obtained using Algorithm SEQ is larger than the throughput obtained in solution $\mathcal{A}$, what achieves the proof of Theorem 4.4. ∎

C. Approximation algorithms

SEQ can easily be turned into a valid approximation algorithm with ratio $\rho = \frac{d_{\min}}{d_{\min}+1}$, where $d_{\min}$ is the smallest degree of all servers. At the end of algorithm SEQ, we can disconnect one client from each server whose out-degree has been exceeded. Removing the smallest connected client cannot decrease the average quantity of resource allocated per connection. Thus, if we denote by w^j the average quantity of resource allocated per connection of server $\mathcal{S}_j$ at the end of SEQ, and by w'^j the average quantity of resource allocated per connection after the modification, we have $\frac{w'^j}{d_j} \geq \frac{w^j}{d_j+1}$. Hence $w'^j \geq \frac{d_j}{d_j+1} w^j \geq \rho w^j$. Since the overall throughput T is equal to the sum of all w^j (and therefore is larger than the optimal throughput T^*), we obtain $T' \geq \rho T^*$.

This resource augmentation result can also be seen as an approximation result for the problem MDGT (Minimize Degree for a Given Throughput). Indeed, if we are given a bound $T \leq \min(\sum_j b_j, \sum_i w_i)$ on the throughput, a simple dichotomic search finds the minimum value α_{SEQ} of α such that the throughput of SEQ$(I(\alpha))$ is at least T on the modified instance $I(\alpha)$ in which server $\mathcal{S}_j$ has degree $d_j + \alpha$. Theorem 4.4 states that if there exists a solution $\mathcal{A}$ of throughput T for instance $I(\alpha - 1)$, then SEQ$(I(\alpha-1))$ provides a valid solution for instance $I(\alpha)$ whose throughput is at least T.

Therefore, $\alpha_{\text{SEQ}} \leq \alpha^* + 1$, where α^* is the optimal (integer) value of the problem MDGT for instance I. Since MDGT is NP-complete, this is the best possible approximation result.

V. Online Case Analysis

In this section, we consider more specifically the online case, where the set of clients is not known in advance, but clients can join and leave the system at any time.

Let us start the analysis of the online case by proving that no online algorithm whose cost is less than 2 (see Definition 3.1) can achieve a constant approximation ratio for the online MTBD problem. This result holds true even if we allow any constant resource augmentation ratio on the degree of the servers, what strongly differs from the offline setting, where a constant additive resource augmentation of 1 is enough to achieve optimal throughput. The proof is by counter-example.

An algorithm $\mathcal{A}_\alpha$ uses $\alpha \geq 1$ *resource augmentation ratio* when the maximal degree used by a server $\mathcal{S}_j$ is $d_j + \alpha$, while its original degree is d_j. Moreover, let us denote by $OPT(I)$ the optimal throughput on instance I, and by $\mathcal{A}_\alpha(I)$ the throughput provided by Algorithm $\mathcal{A}_\alpha$ on instance I.

Theorem 5.1: Given a resource augmentation ratio α and a constant k, there exists an instance I of online MTBD, such that for any algorithm $\mathcal{A}_\alpha$ with cost less than 2,

$$\mathcal{A}_\alpha(I) < \frac{1}{k} OPT(I).$$

Proof: The proof is by exhibiting an instance I on which any online algorithm with cost less than 2 will fail to achieve the required approximation ratio.

This platform consists in only one server $\mathcal{S}$ with bandwidth $b = (2k)^{\alpha+1}$ and degree constraint $d = 1$. On the other hand, let us consider a set of clients $\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_{\alpha+1}$ whose capacities are $1, 2k, (2k)^2, \dots, (2k)^{\alpha+1}$. In the online instance I, clients arrive one after the other, by increasing capacities. More precisely, at round j, for $0 \leq j \leq \alpha+1$, client $\mathcal{C}_j$ with capacity $(2k)^j$ is added. Clearly, since the degree of the server is 1, only 1 client can be attached to the server and, since clients arrive by increasing capacity, the optimal solution consists in attaching $\mathcal{C}_j$ to the server at round j. Note that maintaining this optimal solution at any time step has cost 2, since at each round, client $\mathcal{C}_j$ is connected to the server and client $\mathcal{C}_{j-1}$ is disconnected.

In fact, any online algorithm that achieves an approximation ratio of at most k must attach $\mathcal{C}_j$ to the server at round j. Indeed, the capacity of $\mathcal{C}_j$ is larger than $\frac{3}{2}k$ times the overall capacity of all previous clients, since $\sum_0^{j-1}(2k)^i < (\frac{3}{2}k)(2k)^j$. Therefore, any online algorithm whose approximation ratio is at most k needs to connect a new client at each round. Therefore, if its cost is strictly less than 2, it cannot disconnect clients, so that after round $\alpha+1$, the degree of the server would be $\alpha+2$, thus violating the maximal resource augmentation on the degree of the server node. ∎

A. OSEQ *Algorithm*

Let us now present OSEQ Algorithm, the online version of Algorithm SEQ. OSEQ Algorithm retains the performance guarantee of SEQ by achieving the optimal throughput with only one extra connection per server. Moreover, OSEQ guarantees that each time a client joins or leaves the platform it produces at most 4 changes at each server, *i.e.,* the cost of OSEQ Algorithm is 4.

OSEQ Algorithm can at first be seen as a *pseudo-online* algorithm in the sense that it produces the same solution as if SEQ was computed from the start at each round. In fact, even if it is easier to present and analyze OSEQ in this way, we will show in Section V-C how to re-use some of the computations to lower the complexity. A global view of the naive version of OSEQ is presented in Algorithm 2.

Algorithm 2 Algorithm OSEQ (naive version)

```
UPON a new round starts;
    SET LS the list of servers;
    SET LC = sort(LC) the ordered available clients at the current round;
    APPLY algorithm SEQ to the instance (LS, LC);
RETURN SEQ(LS, LC), the allocation at the current round;
```

Corollary 5.2: The throughput provided by algorithm OSEQ at every round is at least as much as the optimal throughput when the degree constraint is satisfied.

Above corollary follows directly from Theorem 4.4

B. *Guarantee on the number of changes*

We proceed now by proving that the solution provided by OSEQ Algorithm at every round (*i.e.,* when a client joins or leaves the platform) produces at most 4 changes per server. To this end, we will keep track of the differences between the lists of remaining clients throughout the execution of OSEQ.

Definition 5.3: Let $\mathcal{LC}$ and $\mathcal{R}$ be two ordered lists of clients. We will say that $\mathcal{R}$ is an *augmented* version of $\mathcal{C}$ if it is obtained from $\mathcal{LC}$ by the insertion of a new client and possibly the increase of the capacity of the next client. Formally, $\mathcal{LC}$ is *augmented* to $\mathcal{R}$ if there exists an integer $p \leq n$, a new client $\mathcal{X}$ and a value $y \geq 0$ such that $\mathcal{R} = \{\mathcal{C}_1, \ldots, \mathcal{C}_{p-1}, \mathcal{X}, \mathcal{C}'_p, \mathcal{C}_{p+1}, \ldots, \mathcal{C}_n\}$, where the capacity of $\mathcal{X}$ is smaller or equal to w'_p (the new capacity of client $\mathcal{C}_p$) and $w'_p = w_p + y \leq w_{p+1}$.

The following lemma shows that a list of clients and any augmented version of it, when allocated to the same server, produces almost the same allocation. Let I be an instance consisting on one server with capacity b and degree d. Let $\mathcal{LC}'$ be the updated list of clients after OSEQ is applied to instance I with list of clients $\mathcal{LC}$, and let us denote with $\mathcal{A}$ the output of OSEQ. Similarly, let $\mathcal{R}'$ denote the updated list of clients and $\mathcal{B}$ be the output of OSEQ when applied to instance I with a different list of clients $\mathcal{R}$.

Lemma 5.4: If $\mathcal{R}$ is an augmented version of $\mathcal{LC}$, then $\mathcal{R}'$ is an augmented version of $\mathcal{LC}'$, and the allocations $\mathcal{A}$ and $\mathcal{B}$ differ by at most 4 changes.

Proof: Let $\mathcal{R}$ consist of $\{\mathcal{C}_1, \ldots, \mathcal{C}_{p-1}, \mathcal{X}, \mathcal{C}'_p, \mathcal{C}_{p+1}, \ldots, \mathcal{C}_n\}$, where $w'_p = w_p + y$ and the capacity of $\mathcal{X}$ ($w_\mathcal{X}$) is equal to x. The first step of the proof consists in computing the partial sums $\mathcal{R}(u, v)$ for any $u \leq v \leq n$. A quick case study shows that

$$\mathcal{R}(u,v) = \begin{cases} \mathcal{LC}(u-1, v-1) & \text{if } p < u-1, \\ \mathcal{LC}(u-1, v-1) + y & \text{if } p = u-1, \\ \mathcal{LC}(u, v-1) + x + y & \text{if } u \leq p < v, \\ \mathcal{LC}(u, v-1) + x & \text{if } p = v, \\ \mathcal{LC}(u, v) & \text{if } p > v. \end{cases}$$

In particular, since by hypothesis $x \leq w_p$ and $x + y \leq w_{p+1}$, then in all cases, $\mathcal{R}(u, v) \leq \mathcal{LC}(u, v)$. Furthermore, since $x \geq w_{p-1}$, $\mathcal{R}(u, v) \geq \mathcal{LC}(u-1, v-1)$ also holds in all cases.

Let us now consider the application of OSEQ(d, b) to $\mathcal{LC}$. Without loss of generality, we consider that a suitable interval $[l, l+d]$ has been found, *i.e.,* that $\mathcal{LC}(l, l+d-1) < b$ and $\mathcal{LC}(l+1, l+d) \geq b$. In that case, allocation $\mathcal{A}$ is $(\mathcal{C}_l, \ldots, \mathcal{C}_{l+d-1}, \mathcal{C}^{(a)}_{l+d})$, and the updated list $\mathcal{LC}'$ is $(\mathcal{C}_1, \ldots, \mathcal{C}_{l-1}, \mathcal{C}^{(b)}_{l+d}, \ldots, \mathcal{C}_n)$, where $\mathcal{C}^{(a)}_{l+d}$ and $\mathcal{C}^{(b)}_{l+d}$ are the two parts of the split client $\mathcal{C}_{l+d}$.

If the change from $\mathcal{LC}$ to $\mathcal{R}$ lies outside of the interval $[l, l+d]$ (*i.e.,* $p < l$ or $p > l+d$), then this change does not affect to the execution of Algorithm OSEQ, and allocations $\mathcal{A}$ and $\mathcal{B}$ are the same. In that case, thus, the result holds.

Otherwise, the resulting allocation $\mathcal{B}$ depends on the value of $\mathcal{R}(l+1, l+d)$. Indeed, our previous bounds for $\mathcal{R}(u, v)$ shows that $\mathcal{R}(l, l+d-1) < b$, and $\mathcal{R}(l+2, l+d+1) \geq b$. Thus, either $[l, l+d]$ or $[l+1, l+d+1]$ is the suitable interval for the application of OSEQ to $\mathcal{R}$.

Let us suppose that $\mathcal{R}(l+1, l+d) \geq b$*:* In this case, the resulting allocation $\mathcal{B}$ is $(\mathcal{C}_l, \ldots, \mathcal{X}, \mathcal{C}'_p, \ldots, \mathcal{C}^{(a)}_{l+d-1})$. It differs from $\mathcal{A}$ by 4 changes: the addition of $\mathcal{X}$, the removal of $\mathcal{C}^{(a)}_{l+d}$, and the modification of both $\mathcal{C}_p$ and $\mathcal{C}_{l+d-1}$. The updated list of clients $\mathcal{R}'$ is $(\mathcal{C}_1, \ldots, \mathcal{C}_{l-1}, \mathcal{C}^{(b)}_{l+d-1}, \mathcal{C}_{l+d}, \ldots, \mathcal{C}_n)$. It is thus an augmented version of $\mathcal{LC}'$, since $\mathcal{C}^{(b)}_{l+d-1}$ is inserted between

$\mathcal{C}_{l-1}$ and the capacity of $\mathcal{C}_{l+d}^{(b)}$ is increased to w_{l+d}. Indeed, by definition of the splitting process, $w(\mathcal{C}_{l+d-1}^{(b)}) = \mathcal{R}(l, l+d) - b$ and $w(\mathcal{C}_{l+d}^{(b)}) = \mathcal{LC}(l, l+d) - b$, which implies $w(\mathcal{C}_{l+d-1}^{(b)}) \leq w(\mathcal{C}_{l+d}^{(b)})$.

If $p = l + d$, then $\mathcal{X}$ is the split client. This case actually results in only two changes in the allocations: the addition of (one part of) $\mathcal{X}$, and the removal of $\mathcal{C}_{l+d}^{(a)}$. The updated list $\mathcal{R}'$ is also an augmented version of $\mathcal{LC}'$, with the remaining part of $\mathcal{X}$ inserted and the capacity of $\mathcal{C}_{l+d}^{(b)}$ increased to $w_{l+d} + y$.

Let us now suppose that $\mathcal{R}(l+1, l+d) < b$*:* In this case, the suitable interval is $[l+1, l+d+1]$ and thus the resulting allocation $\mathcal{B}$ is $(\mathcal{C}_{l+1}, \ldots, \mathcal{X}, \mathcal{C}'_p, \ldots, \mathcal{C}_{l+d}^{(a')})$. Once again, it differs from $\mathcal{A}$ by 4 changes: the addition of $\mathcal{X}$, the removal of $\mathcal{C}_l$, and the modification of both $\mathcal{C}_p$ and $\mathcal{C}_{l+d}^{(a)}$. The updated list of clients $\mathcal{R}'$ is $(\mathcal{C}_1, \ldots, \mathcal{C}_l, \mathcal{C}_{l+d}^{(b')}, \ldots, \mathcal{C}_n)$. It is therefore an augmented version of $\mathcal{LC}'$, since $\mathcal{C}_l$ is inserted in right after $\mathcal{C}_{l-1}$ and the capacity of $\mathcal{C}_{l+d}^{(b)}$ is increased to $w(\mathcal{C}_{l+d}^{(b')})$. In this case, $w_l \leq w(\mathcal{C}_{l+d}^{(b)})$ comes from the ordering property of SEQ, see Section V-A.

If $p = l$, then $\mathcal{X}$ is actually not included in $\mathcal{B}$. Thus, this case results in two changes only: the modification of the capacity of $\mathcal{C}_p$, and the fact that $\mathcal{C}_{l+d}$ is split differently. The updated list $\mathcal{R}'$ is also an augmented version of $\mathcal{LC}'$, with $\mathcal{X}$ inserted and the capacity of $\mathcal{C}_{l+d}^{(b)}$ increased.

■

Theorem 5.5: The cost of Algorithm OSEQ' is at most 4.

Proof: Let us prove that if two lists of clients $\mathcal{LC}$ and $\mathcal{R}$ differ by the addition of a new client, then the resulting allocations to each server computed by OSEQ differ by at most 4 changes.

Denote by $\mathcal{LC}^j$ the current list of clients after the first j rounds of OSEQ starting from $\mathcal{LC}^0 = \mathcal{LC}$, and similarly for $\mathcal{R}$. It is clear that $\mathcal{R}$ is an augmented version of $\mathcal{LC}$, Lemma 5.4 shows that if $\mathcal{R}^j$ is an augmented version of $\mathcal{LC}^j$, then $\mathcal{R}^{j+1}$ is an augmented version of $\mathcal{LC}^{j+1}$. Then, a trivial induction based on the application of Lemma 5.4 proves that resulting allocations differ by at most 4 changes.

In the case of the removal of a client, we can simply swap the role of $\mathcal{LC}$ and $\mathcal{R}$ in the previous statements. ■

C. Efficient Implementation Issues

We have first presented the naive version of OSEQ, as an algorithm that recomputes from scratch the whole solution at each round. However, the proof of Lemma 5.4 shows that it is possible to compute only the changes between the previous allocation and the new one. The implementation is made more complex by the analysis of many cases depending on the values of p, l, $l + d$ and so on. However, it is also more efficient, since for each server, we only have to decide whether the suitable interval is $[l, l+d_j]$ or $[l+1, l+d_j+1]$, instead of going through the whole list of clients. It is thus possible to do it in constant time, what leads to a global complexity of $\theta(m)$ for Algorithm OSEQ.

VI. EXPERIMENTAL EVALUATION

A. Heuristics for comparison

As already mentioned in Section II, related work has mostly been done in the context of *Bin Packing*, where there is an infinite amount of identical bins, and the goal is to pack all items in as few bins as possible[2]. Interestingly, in this setting, the NEXT-FIT algorithm has a worst-case approximation ratio of $2 - 1/k$ [16], but it can easily be observed that it does not exhibit a constant approximation ratio for the total packed size when the number of bins is fixed. Moreover, most of existing algorithms in this context are approximation schemes, with prohibitive running times. To provide a basis of comparison, we thus introduce online versions of the natural greedy heuristics that performed best in the offline setting.

- **LCLS (Largest Client Largest Server)** At each step, the client with the largest w_i is associated with the server with the largest available capacity $b'_j = b_j - \sum_i w_i^j$. The client is split if necessary, in which case the remaining $w'_i = w_i - b'_j$ is inserted in the ordered list.
- **LCBC (Largest Client Best Connection)** In this heuristic, we also consider the largest client first, but servers are ordered according to their remaining *capacity per connection*, which is defined as the ratio between the remaining capacity b'_j and the remaining available degree d'_j. The server with the largest capacity per connection is selected. Here also, the client is split if necessary.
- We also define an online version of this heuristic: **Online Best Connection (OBC)**. Servers are still ordered by their remaining *capacity per connection*. When a new client arrives, it is connected to the server whose capacity per connection is closest to the client's capacity. If no server is available, OBC goes through all servers which have some bandwidth remaining but no degree left, and swaps the

[2] In our context, servers are bins and clients are items

newly arrived client with a smaller one, selecting the server which yields the largest gain in total throughput.

When a client $\mathcal{X}$ leaves, OBC tries to use the newly available bandwidth to reduce the indegree of other clients. Assume that $\mathcal{X}$ was connected to server $\mathcal{S}$, and that client $\mathcal{Y}$ is connected to both $\mathcal{S}$ and $\mathcal{S}'$. When $\mathcal{X}$ leaves, $\mathcal{S}$ can reallocate the corresponding bandwidth to client $\mathcal{Y}$, what can be of interest if this allows to disconnect $\mathcal{Y}$ from $\mathcal{S}'$, since this lowers the outdegree of $\mathcal{S}'$. OBC selects as many such *incident* connections as possible, starting from the smallest ones. If there are some unconnected clients remaining, OBC then acts as if they had just arrived and tries to connect them with the procedure described earlier.

We analyze here a worst-case instance on which LCBC achieves a throughput significantly lower than SEQ, for the same resource augmentation on the degree. For fixed even m and B, let us consider an instance with m servers, each of capacity B and degree $d = \frac{B}{2}$, $\frac{mB}{2}$ *small* clients of size 1, with an additional *big* client of size $\frac{mB}{2}$.

On this instance, the solution of SEQ is to assign to each server d small clients and a part of size $\frac{B}{2}$ of the big client. This solution achieves a throughput of mB.

On the other hand, LCBC assigns first the big client to as few servers as possible ($\frac{m}{2}$ of them), with parts of size B. Once this is done, $\frac{m}{2}$ servers remain unused, and each can only accommodate $d+1$ small clients because of the degree constraint. The total throughput of this solution is thus $B\frac{m}{2} + (d+1)\frac{m}{2} = mB\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{2B}\right)$.

Hence, when B grows, the ratio between the throughput of SEQ and LCBC tends to $\frac{3}{4}$.

B. Random Instance Generation

We generate instances randomly, trying to focus at the same time on realistic scenarios and difficult instances. Instances are more difficult to solve when the sum of server capacities is roughly equal to the sum of client capacities. Indeed, the minimum of both is a trivial upper bound on the total achievable throughput, and a large difference between them provides a lot of freedom on the largest component to reach this upper bound. Based on the same idea, we generate instances where the sum of the server degrees $\sum_j d_j$ is roughly equal to the number n of clients.

In order to get a realistic distribution of server and client capacities, we have used information available from the volunteer computing project GIMPS [1] that provides the average computing power of all its participants. A simple statistical study shows that the computational power (based on the 7,000 largest participants) follows a power-law distribution with exponent $\hat{\alpha} \approx 2.09$. We have thus used this distribution and this exponent to generate the capacities of both clients and servers. The resulting values are then scaled so that their sums ($\sum_i w_i$ and $\sum_j b_j$) are roughly equal. Furthermore, the degree d_j of server $\mathcal{S}_j$ is chosen proportional to its capacity b_j (it seems reasonable to assume that a server with a larger capacity can accommodate more clients), with a Gaussian multiplicative factor of mean 1 and variance 0.1. We generate instances with m servers and $n = pm$ clients, where p is chosen as 10 or 50, and m varies between 10 and 160.

To generate online instances, we start from a complete instance. Two kinds of random events are then generated: departure of a client (picked uniformly at random), or arrival of a newly generated client. We generate 300 such events, each kind having probability $1/2$. The time intervals between two successive events are generated as a Poisson process.

C. Results

We ran simulations for different instance sizes by varying the number m of servers. For each value of m, 250 instances were generated, and we plot on the figures the average, median, and the first and last decile over these 250 instances. For each algorithm, the line connects the average values, the upper error bar shows the last decile (which means that on 10% of the instances, the value was higher), the lower error bar shows the first decile (the value was lower on 10% of the instances), and the lonely mark in between is the median (half of the instances had lower values).

1) Offline simulations: In the first set of experiments, we have measured the throughput of the solutions proposed by each algorithm. All values are normalized against the previously mentioned upper bound $\min(\sum_j b_j, \sum_i w_i)$. Figure 1 shows the average results on 250 instances when the number of servers varies from 10 to 160. We can already make some remarks:

- For these instances, algorithm SEQ performs consistently better than the others. In fact, it almost always reaches the upper bound.
- The performance of algorithm LCBC is around 4% worse, and LCLS is around 10-12% worse than SEQ.
- The value of p has little influence on the results, except that variability of SEQ decreases with higher values of p.

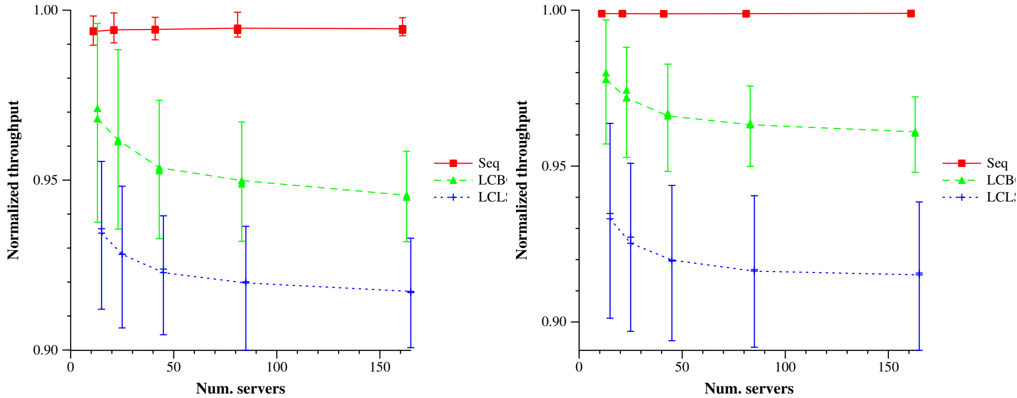

Figure 1. Offline simulations: Average normalized throughput for $p = 10$ and $p = 50$.

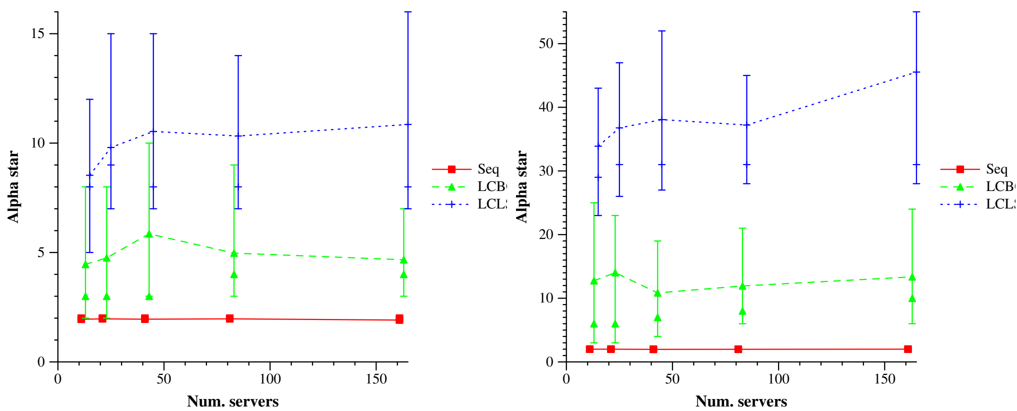

Figure 2. Offline simulations: Average α^* for 250 instances for $p = 10$ and $p = 50$.

In a second set of experiments, we have computed for each algorithm $\mathcal{A}$ the minimum value α^* that needs to be added to the degree of each server so that algorithm $\mathcal{A}$ reaches the upper bound $B = \min(\sum_j b_j, \sum_i w_i)$. Note that the results of Section IV do not imply that $\alpha^* \leq 1$ for algorithm SEQ, since it may well be the case that the upper bound cannot be reached with the original degree sequence. Average results for all algorithms and for varying m are depicted in Figure 2.

We can see that, as expected, algorithm SEQ makes very good use of the additional degree, and can almost always reach the upper bound with an increase of 1 or 2. As expected also, the ranking of algorithms observed for the total throughput is still the same when considering α^*. We see that with LCBC, one needs about 5 more connections to reach the bound for $p = 10$, and between 10 and 15 when $p = 50$ (notice that since the sum of the server degrees $\sum_j d_j$ is roughly equal to the number n of clients, p represents the average degree of the servers).

A more precise look at the results for $m = 160$ is shown on Figure 3, where the value of α^* for each instance is plotted against the dispersion of the clients capacities, measured by the relative mean difference of these values[3]. We can see that most of the values for LCBC are between 2 and 5 for $p = 10$, and between 6 and 12 for $p = 50$. However, it can be as high as $2p$ for instances with very large dispersion in client capacities, and these high values tend to increase the average. The results for algorithm LCLS exhibit the same kind of behavior, with larger values of α^* for the most heterogeneous instances, and this explains larger average values. Therefore, for these difficult heterogeneous instances, we can see the benefit of the guarantee proved in Section IV for algorithm SEQ. Indeed, in these simulations the mean value of α is p, so that a value of α^* of order more than 5 is expected to degrade significantly the networking

[3]The *mean difference* of values $\{y_i\}$ is the average absolute difference of all couples of values. The *relative mean difference* is the *mean difference* divided by the arithmetic mean.

performances of the servers. Thus, greedy algorithms fail to use the whole capacity of the platform in strongly heterogeneous cases, whereas 1 or 2 extra connections are enough using SEQ.

2) Online simulations: In the online simulations, we compare Online SEQ with OBC, an online version of LCBC. LCBC consistently outperforms LCLS in offline simulations, and this ranking still holds about their online version, and thus the online version of LCLS is not analyzed here. However, we also consider another version of SEQ, named SEQLEFT, which selects the leftmost suitable set of clients (instead of the rightmost one for SEQ). SEQ andSEQLEFT have very similar performance in the offline case, but their cost in online situations is quite different.

On Figure 4, we plot the total number of computed tasks throughout the instance, which is simply the integral over time of the instantaneous throughput (assuming for simplicity that changes from one solution to the other take no time). The value obtained is then normalized against the upper bound $\min(\sum_j b_j, \sum_i w_i)$ (so that an average over 250 instances make sense). We can see that the offline results can be observed in this situation as well: the performance of OBC is about 5% worse than that of SEQ, which is always very close to the upper bound. Furthermore, higher values of p lower the variability of the results.

Figure 5 shows the cost of the algorithms. We can see that the cost of SEQ is always 4, while the cost of OBC is between 10 and 20 on average. However, it is once again very variable and reaches 35 on roughly 10% of the instances. Remember that the average outdegree of the servers is p (instances with m servers contain pm clients), so this result means that it is quite likely that, using OBC, at some point in the execution, one server has to change more than half of the clients it is connected to. This figure also shows the importance of the locality obtained by selecting the rightmost suitable interval in SEQ: the cost of SEQLEFT is not bounded by 4, and can get as high as the cost of OBC.

On the other hand, Figure 6 shows the average of the costs over the 300 events. We can see that the average cost of an event for SEQ is between 3 and 4, while it is around 1.5 (varying between 1.3 and 2) for OBC. This shows that events that incur many changes for OBC are relatively rare, and are compensated by many events that generate no or very few changes. However, we feel that this cost for maintaining the guarantees are justified by the higher performance and use of computing resources, and by the stability of SEQ.

VII. CONCLUSIONS

In this work, we have considered a resource allocation problem that models both independent tasks scheduling and virtual machines allocation problems. With respect to existing literature, our main contribution is to introduce a degree constraint, that is crucial for realism in both contexts. We prove that even if this additional constraint makes the resource allocation problem NP-Complete, only a very small resource augmentation on the degree is sufficient to achieve optimality. We also analyze the online setting, where the resources can change during the execution, as expected in mentioned applications. In the online context, we prove that maintaining optimality is not more expensive (up to a ratio of 2) than achieving a constant approximation ratio. Finally, we provide an extensive set of simulation results based on realistic data.

REFERENCES

[1] The great internet mersenne prime search (gimps). http://www.mersenne.org/.

[2] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 365–372, 2004.

[3] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[4] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, pages 319–330, 2004.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Pipelining Broadcasts on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, pages 300–313, 2005.

[6] Olivier Beaumont, Lionel Eyraud-Dubois, Hejer Rejeb, and Christopher Thraves. Allocation of clients to multiple servers on large scale heterogeneous platforms. In *IEEE 15th International Conference on Parallel and Distributed Systems, ICPADS*, pages 142–149, 2009.

[7] Olivier Beaumont, Lionel Eyraud-Dubois, Hejer Rejeb, and Christopher Thraves. On-line allocation of clients to multiple servers on large scale heterogeneous systems. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP*, pages 3–10, 2010.

[8] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 577–578. IEEE, 2010.

[9] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M.Q. Dang, and K. Pentikousis. Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045, 2010.

[10] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[11] Martin A. Brown. Traffic Control HOWTO. Chapter 6. Classless Queuing Disciplines. *http://tldp.org/HOWTO/Traffic-Control-HOWTO/classless-qdiscs.html*, 2006.

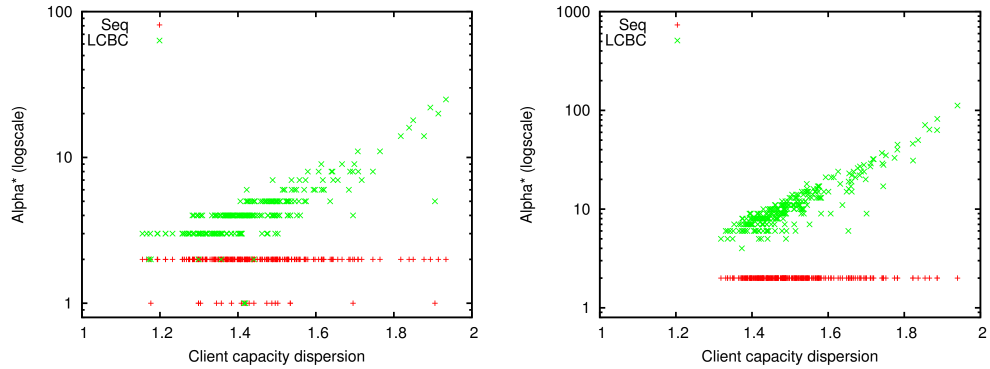

Figure 3. Offline simulations: α^* against client dispersion for $m = 160$, $p = 10$ and $p = 50$.

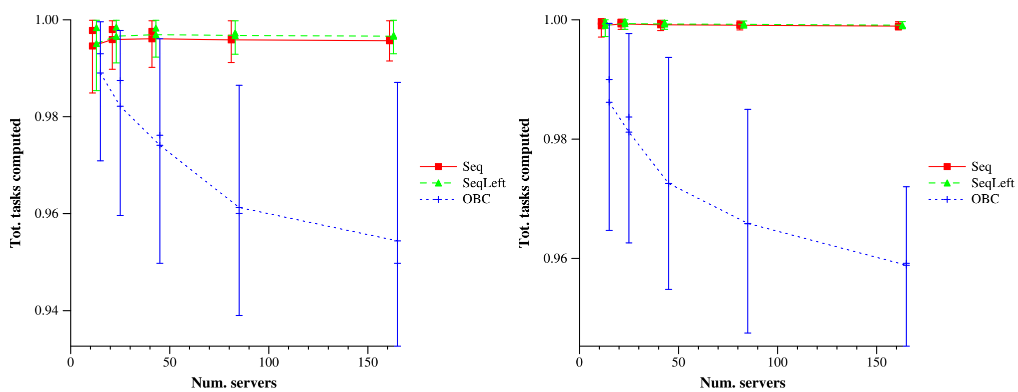

Figure 4. Online simulations: Average normalized tasks computed for $p = 10$ and $p = 50$.

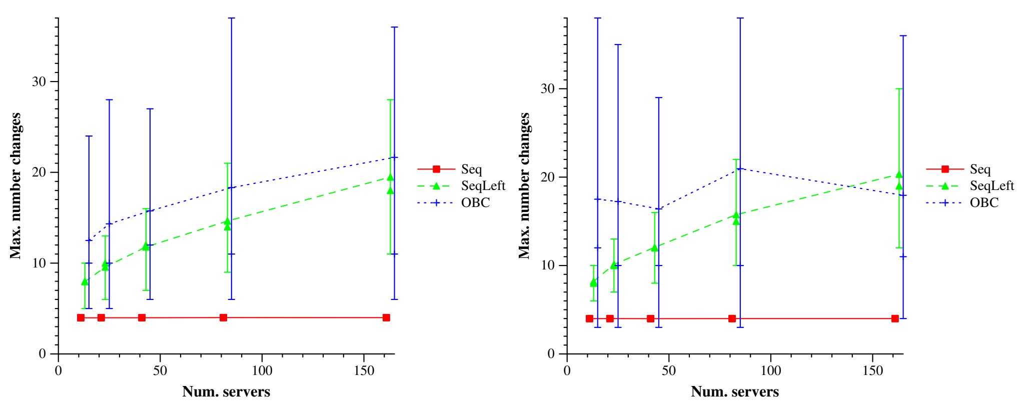

Figure 5. Online simulations: Maximum cost for $p = 10$ and $p = 50$.

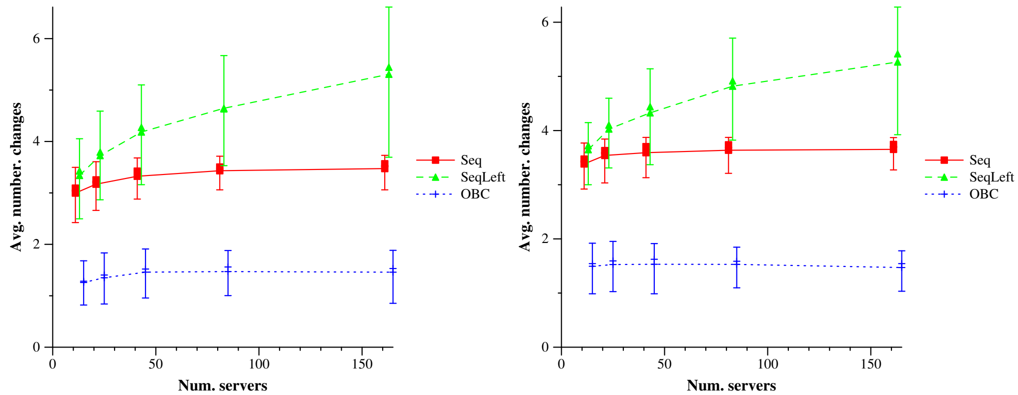

Figure 6. Online simulations: Average cost over 300 events for $p = 10$ and $p = 50$.

[12] R.N. Calheiros, R. Buyya, and C.A.F. De Rose. A heuristic for mapping virtual machines and links in emulation testbeds. In *2009 International Conference on Parallel Processing*, pages 518–525. IEEE, 2009.

[13] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 363–372, New York, NY, USA, 2004. ACM.

[14] F. Chung, R. Graham, J. Mao, and G. Varghese. Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines. *Theory of Computing Systems*, 39(6):829–849, 2006.

[15] L. Epstein and R. van Stee. Approximation Schemes for Packing Splittable Items with Cardinality Constraints. *Lecture Notes in Computer Science*, 4927:232, 2008.

[16] Leah Epstein and Rob van Stee. Improved results for a memory allocation problem. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 362–373. Springer, 2007.

[17] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:183, 2001.

[18] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman San Francisco, 1979.

[19] B. Hong and V.K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. *International Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[20] B. Hubert et al. Linux Advanced Routing & Traffic Control. Chapter 9. Queueing Disciplines for Bandwidth Management. *http://lartc.org/lartc.pdf*, 2002.

[21] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[22] Kernel based virtual machine. http://www.linux-kvm.org/page/Main_Page.

[23] S.M. Larson, C.D. Snow, M. Shirts, and V.S. Pande. Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[24] CA Phillips. Optimal Time-Critical Scheduling via Resource Augmentation. *Algorithmica*, 32(2):163–200, 2008.

[25] T. Saif and M. Parashar. Understanding the Behavior and Performance of Non-blocking Communications in MPI. *Lecture Notes in Computer Science*, pages 173–182, 2004.

[26] H. Shachnai and T. Tamir. Multiprocessor Scheduling with Machine Allotment and Parallelism Constraints. *Algorithmica*, 32(4):651–678, 2002.

[27] Hadas Shachnai, Tami Tamir, and Omer Yehezkely. Approximation schemes for packing with item fragmentation. *Theory Comput. Syst.*, 43(1):81–98, 2008.

[28] H.N. Van, F.D. Tran, and J.M. Menaud. SLA-aware virtual resource management for cloud infrastructures. In *IEEE Ninth International Conference on Computer and Information Technology*, pages 357–362. IEEE, 2009.

[29] Vmware. http://www.vmware.com/virtualization/.

[30] Xen. http://www.xen.org/.

[31] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.