Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster

Arie Keren and Amnon Barak

Abstract—Computing Clusters (CC) consisting of several connected machines, could provide a high-performance, multiuser, time-sharing environment for executing parallel and sequential jobs. In order to achieve good performance in such an environment, it is necessary to assign processes to machines in a manner that ensures efficient allocation of resources among the jobs. This paper presents opportunity cost algorithms for online assignment of jobs to machines in a CC. These algorithms are designed to improve the overall CPU utilization of the cluster and to reduces the I/O and the Interprocess Communication (IPC) overhead. Our approach is based on known theoretical results on competitive algorithms. The main contribution of the paper is how to adapt this theory into working algorithms that can assign jobs to machines in a manner that guarantees near-optimal utilization of the CPU resource for jobs that perform I/O and IPC operations. The developed algorithms are easy to implement. We tested the algorithms by means of simulations and executions in a real system and show that they outperform existing methods for process allocation that are based on ad hoc heuristics.

Index Terms—Load balancing, competitive algorithms, cluster computing, I/O overhead, IPC overhead.

1 Introduction

Scalable computing clusters are becoming the primary platform for executing demanding scientific, engineering, and commercial applications [2]. A Computing Cluster (CC) usually consists of a collection of homogeneous or heterogeneous computers (machines) that are connected by a high-speed LAN. It can provide a cost-effective, high-performance environment for executing parallel and sequential jobs as well as high-availability.

A CC can either be used as dedicated system for a single user, or as a multiuser, time-sharing environment. In order to achieve good performance in the latter case, it is necessary to use a method for assignment of jobs to machines, to ensure efficient allocation of resources across all the machines. Such a method should take into account many parameters, e.g., CPU load, amount of available memory, I/O, communication, etc., in order to minimize the mean slowdown of the jobs.

It is well known that an optimal assignment of jobs to machines is NP-hard, even when jobs compete for a single resource, e.g., CPU, and the demands for this resource are known a priori [18]. In practice, the assignment problem is exaggerated by the unpredictable and dispersive nature of the resource requirements of arriving jobs, as well as by the diversity of factors that affect the performance of the jobs, e.g., machine architecture, paging, I/O, or interprocess communication.

• *The authors are with the Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel. E-mail: amnon@cs.huji.ac.il.*

Manuscript received 12 May 2000; revised 6 July 2001; accepted 25 Apr. 2002.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112133.

This paper presents an opportunity cost framework for online assignment of sequential and parallel jobs to machines. We develop algorithms to reduce the maximal utilization of the CPU resource for jobs that perform I/O and IPC operations. These algorithms are based on theoretical work on competitive algorithms for online load balancing and routing of virtual circuits [6]. The routing algorithms use the economical principle of commodity pricing in order to reduce the congestion of the network edges [29]. This method is applied in the current paper to assign jobs with CPU, I/O, and IPC requirements to machines in order to reduce the maximal usage of the CPU resource. We prove that this method yields a competitive assignment of jobs to machines for each possible combination of CPU, I/O, and IPC requirements. In our model, I/O and IPC requirements are expressed as CPU resources on several machines. The cost of the CPU resource in each machine is a nonlinear function of its utilization. The cost of an assignment is the sum of the CPU costs in all the involved machines. Arriving jobs are initially assigned to machines that yield a minimal marginal cost. Since an initial assignment can become nonoptimal, e.g., due to job terminations, the opportunity cost algorithm may decide to migrate (reassign) a job from one machine to another, to ensure a competitive assignment at all times.

The opportunity cost approach provides a *unified* algorithmic framework, which is based on theoretically proven competitive algorithms. This is different from other existing methods, which are based on ad hoc heuristics for one resource at a time. It was first used in [1] for management of CPU and memory resources of *sequential jobs*. This paper extends the results of [1] by presenting algorithms that assign both *sequential* and *parallel jobs* to machines in a manner that minimizes the CPU utilization for jobs with I/O and IPC overhead.

1.1 Contributions of this Paper

This paper has two main contributions. First, we show how to adapt theoretical results from the field of online competitive algorithms to better reflect a working cluster parameters. This is accomplished by developing opportunity cost algorithms for jobs with I/O and IPC overhead, and proving their competitiveness. These algorithms were designed to minimize the maximal utilization of the cluster's CPU resource.

Our second contribution is to demonstrate the performance of the opportunity cost algorithms by simulations and by real executions. While the theoretical model deal with maximal resource utilization and worst-case performance, we show that the resulting algorithms also improve the mean slowdown of the jobs. We conducted a series of simulations of a "typical" environment of parallel and sequential jobs, comparing the performance of the opportunity cost algorithms against a widely used, *greedy* based algorithms. The simulations show that the opportunity cost algorithms significantly outperforms the greedy algorithm in almost all cases. We note that these results were further validated in real executions, thus demonstrating that the transition from theory to practice is almost straight forward in this case.

1.2 Related Previous Work

The theory of online competitive algorithms offers many strong results on scheduling and load-balancing in a computing cluster [17]. In spite of the diversity of theoretical results and the availability of many abstract algorithms, it is rare to find real systems that employ such techniques. Since the goal of this paper is to bridge this gap, we give below a short overview of existing methods that are used in real computing clusters.

Computing Cluster (CC) can be used as a dedicated system for a specific application [5]. It is more difficult to use such systems as an efficient, multiuser time-sharing platform for running a mixture of sequential and parallel jobs. For example, current parallel programming environments for CC, such as PVM [19] and MPI [34], use a simple, static job assignment algorithm, completely ignoring the current state of the resource utilization of the CC. Another method for global resource management is by *space-sharing*, in which the cluster is partitioned to disjoint sets of machines, each executing one parallel job. In most systems that support this method, it is the job's responsibility to assign its processes to machines within its partition, e.g., as in Piranha [13]. Due to the obvious limitations of the *space-sharing* method, it is used mostly for batch systems [24]. To reduce the waiting time of queued jobs, a *dynamic* strategy can change the partitions at runtime, to reflect changes in system load or the job's requirements. For example, the dynamic equipartition policy [15], [32], which allocates machines evenly among jobs, outperforms other space-sharing strategies.

Time-sharing systems, which allow several jobs to share the cluster resources, can either use a *static*, a *dynamic*, or an *adaptive* job allocation policies [33]. A *Static* policy, e.g., as *round-robin* or a *random* allocation, uses no information about the system state during an assignment of a job. A *Dynamic* policy assigns a new job based on the current state of the system when the job arrives. For example, GLUNIX [20] assigns a newly created job to the least loaded machine. *Adaptive* (*preemptive*) policies can reassign processes of a job during its execution, to reflect changes in system state. For example, MOSIX [9], [11], [12] performs dynamic load-balancing by (preemptive) process migration.

Most existing methods for load-balancing consider only one resource at a time, e.g., CPU load, while ignoring other resources, e.g. memory. A method for incorporating inter process communication and CPU load-balancing is described in [28], [36]. In these cases, the tradeoff is between an even distribution of the processes to nodes and between grouping communicating processes in common nodes, to reduce the communication overhead. Another method, which includes both CPU-load and memory utilization is used by MOSIX [10]. In this case, when memory is nearly exhausted in one node, MOSIX overrides the load-balancing algorithm by migrating processes to machines with sufficient free memory.

1.3 Organization of the Paper

This paper is organized as follows: Section 2 presents the relevant theoretical background for the opportunity cost algorithms. Section 3 presents an opportunity cost algorithm for reducing the I/O-style communication overhead between sequential jobs and a fix set of machines. Section 4 presents an opportunity cost algorithm for reducing the inter process communication overhead between processes of a parallel job. Section 5 presents an empirical evaluation of our algorithms by means of simulations and real executions, comparing them to traditional, heuristic algorithms. The conclusions are given in Section 6.

2 Theoretical Background

This section presents the relevant theoretical background for the opportunity cost algorithms. First, we present the online load-balancing problem, then the online routing of virtual circuit problem. These two problems form the basis for the presentation of the opportunity cost algorithms, which minimizes the maximal utilization of multiple resources in a computing cluster. Comprehensive surveys of load-balancing and routing problems are given in [8] and [16], respectively.

2.1 Competitive Analysis

A *competitive analysis* [35] is a common technique for measuring the effectiveness of online algorithms. In this technique, the performance of an online algorithm, for any input sequence, is compared to the optimal, offline algorithm which has an a priori, complete knowledge about the input sequence. More formally, let $ALG(I)$ be the performance of an online algorithm ALG for an input sequence I and let $OPT(I)$ be the performance of the optimal offline algorithm for the same input sequence. Then, algorithm ALG is c-competitive if, for any input sequence I, $ALG(I) \leq cOPT(I) + \alpha$, where α is a constant.

2.2 The Load Balancing Problem

Online load-balancing algorithms of a single resource in a computing cluster assign arriving jobs to machines (nodes) in order to minimize the maximal utilization of that resource. This section presents the load-balancing problem and two online competitive algorithms for this problem.

Formally, the load-balancing problem is defined as follows: given n machines and a sequence of independent jobs that arrive at arbitrary times. A job j is defined by its *demand vector*, $p(j) = (p_1(j), p_2(j), \cdots, p_n(j))$, where $p_i(j)$ is the *resource demand* of job j in machine i. When a job arrives it is assigned online to one of the machines, using an online algorithm, thereby increasing the *utilization* of this machine by the corresponding value of its demand vector. When the job departs from a machine, the utilization of this machine is decreased, respectively. The goal of an online algorithm is to minimize the maximal resource utilization among all the machines.

There are three kinds of demand vectors, according to the machine types:

- *Identical machines.* $\forall i, j : p_i(j) = w(j)$, where $w(j) \geq 0$ is the *resource demand* of job j.
- *Related machines.* $\forall i, j : p_i(j) = w(j)/v_i$, where $w(j)$ is as before and $v_i \geq 0$ is the *resource capacity* of machine i. For example, this type can model a cluster with machines of different speeds.
- *Unrelated machines.* $\forall i, j : p_i(j)$ is an arbitrary, nonnegative resource demand, i.e., the resource demand of a job in one machine is unrelated to its demand in any other machine. For example, this type can model a cluster of machines with diverse characteristics, e.g., due to I/O or communication speeds, or SMP architecture constraints.

Jobs can be classified into two types: *permanent* jobs, which run indefinitely; and *temporary* jobs, with a finite execution time that is unknown at the job arrival time.

2.2.1 The Greedy Algorithm

Greedy, a popular online algorithm for load-balancing, assigns a new job to a machine in order to minimize the resulting resource utilization. For this algorithm it was proved that:

Theorem 1 ([21]). *For identical machines, the greedy algorithm has a competitive ratio of* $2 - 1/n$.

Theorem 2 ([6]). *For related machines, the greedy algorithm has a competitive ratio of* $O(\log n)$.

Theorem 3 ([6]). *For unrelated machines, the greedy algorithm has a competitive ratio of* n.

2.2.2 The Marginal Cost Algorithm

An online algorithm that improves the competitive ratio of *Greedy* for unrelated machines, called ASSIGN-U, is presented in [6]. This algorithm defines a nonlinear cost function for the assignment of jobs to machines, such that a new job is assigned in order to minimize its marginal added cost. More formally, let $l_i(j)$ be the utilization of machine i before assigning job j, let $p_i(j)$ be the coordinate i of the demand vector of job j, and let $1 < a < 2$ be a constant. Then, algorithm ASSIGN-U computes the marginal cost of the assignment of job j to machine $i, i \in [1, n]$,

$$H_i(j) = a^{l_i(j)+p_i(j)} - a^{l_i(j)},$$

and chooses the assignment that yields a minimal marginal cost.

Theorem 4 ([6]). *Algorithm* ASSIGN-U *is* $O(\log n)$ *competitive for unrelated machines and permanent jobs.*

In order to maintain an $O(\log n)$ competitive ratio for temporary jobs, the ASSIGN-U algorithm was modified to allow job reassignments between machines during the execution. Reassignments are performed only when a job is terminated. At such time, the algorithm checks if it is possible to reassign some running job j from machine i to machine i'. The following "stability" condition is used:

$$a^{h_i(j)+p_i(j)} - a^{h_i(j)} \leq 2(a^{l_{i'}(j)+p_{i'}(j)} - a^{l_{i'}(j)}), \tag{1}$$

where $h_i(j)$ is the utilization of machine i, just before j was last assigned to i. If the stability condition is not satisfied by some job j, the algorithm reassigns j to a machine i' that minimizes $H_{i'}(j)$.

Theorem 5 ([7]). *Algorithm* ASSIGN-U *is* $O(\log n)$ *competitive. It reassign each job at most* $O(\log n)$ *times.*

The above algorithms assume that the maximum utilization of the optimal offline algorithm OPT is known and that the values $l_i(j)$, $p_i(j)$ (for all i, j) are scaled down, i.e., divided by OPT. When the value of OPT is unknown, it can be approximated using a doubling technique, as shown in [6]. Briefly, the algorithm works in phases, assuming a different value of OPT in each phase. The first phase assumes a very small OPT, e.g., the minimal possible utilization that the first job produces. The value of OPT is doubled at the beginning of each subsequent phase. Within a phase, the algorithm assigns the arriving jobs, ignoring the jobs that were assigned in the previous phases. A new phase is started when a job j arrives and cannot be assigned to any machine without causing its resulting load to exceed $C * OPT$, where C is the exact competitive ratio of the algorithm. This technique increases the competitive ratio by a factor of 4 [6].

2.3 Online Routing of Virtual Circuits

This section presents the problem of online routing of virtual circuits. The algorithm for this problem is based on enhancements of the ASSIGN-U algorithm, presented in [6].

The routing of a virtual circuits problem can be described as follows: given a (directed or undirected) graph $G(V, E)$ and a sequence of independent requests that arrive in arbitrary times. Each edge $e \in E$ has a capacity $u(e)$. Each request j is represented by a tuple $(s_j, t_j, p(j))$, where s_j and t_j are the source and destination nodes, and $p(j)$ is the required bandwidth. A request that is assigned to some path P from a source to a destination increases the utilization of each edge $e \in P$ by the amount $p_e(j) = p(j)/u(e)$. The goal is to assign the requests to paths so that to minimize the maximum utilization over all of the edges.

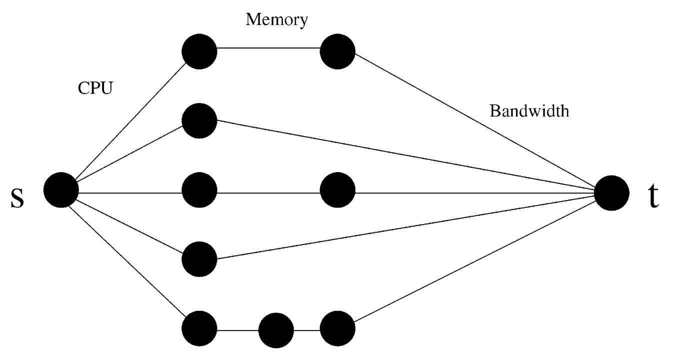

Fig. 1. Mapping graph from multiple resources to virtual circuits.

The following algorithm, called ASSIGN-ROUTE, is based on the economical pricing scheme presented in the previous section. Upon an arrival of request j, the algorithm computes the marginal cost for each possible path P from s_j to t_j as follows:

$$H_P(j) = \sum_{e \in P} a^{l_e(j)+p_e(j)} - a^{l_e(j)},$$

then it assigns request j to a path P that yields a minimal marginal cost.

Theorem 6 ([6]). *Algorithm* ASSIGN-ROUTE *is* $O(\log n)$ *competitive.*

We note that, in [6], it is proven that this algorithm is applicable for the more general routing problem, in which the requested bandwidth is not uniform along the route, i.e., $p_e(j)$ are arbitrary nonnegative values. Also, note that, as in the case of load-balancing of unrelated machines, algorithm ASSIGN-ROUTE can be extended to handle *temporary* requests [7]. In this setting, the algorithm reassigns each request at most $O(\log n)$ times using a stability condition similar to (1). It achieves a competitive ratio of $O(\log n)$ with respect to the maximal utilization.

2.4 Management of Multiple Cluster Resources

The problem of online management of multiple resources in a computing cluster can be defined as follows: given n machines and a sequence of jobs that arrive in arbitrary times. Each machine has multiple resources, e.g., CPU, Memory, I/O, etc. Each job specifies its demand for each resource. An arriving job is assigned online to one of the machines, thereby increasing the utilization of the machine's resources by the corresponding demand. The goal is to minimize the maximal utilization of all the resources.

The *opportunity cost* framework is based on the observation that the problem of management of multiple resources can be reduced to the problem of routing of virtual circuits. This reduction can be obtained by constructing a graph G, see Fig. 1, with a *source* node s and a *target* node t, such that there are n paths between nodes s and t. Each path represents one machine, with the edges representing its resources. A job is represented by a request for a virtual circuit from s to t, with a resource demand for each edge.

The opportunity cost method applies the ASSIGN-ROUTE algorithm to the management of multiple resources problem by the above reduction. Thus, we get:

Theorem 7. *The opportunity cost method is* $O(\log n)$ *competitive with respect to the maximal utilization of each of the resources.*

In a previous work [1], the opportunity cost approach was used to manage the allocation of CPU and memory resources of sequential jobs. In this paper, we develop two new type of opportunity cost algorithms for communication and I/O operations of parallel and sequential jobs.

3 Jobs with I/O-Style Communication Overhead

I/O-style communication is a special form of communication between a job and a fixed set of machines, e.g., access to the job's (local and/or remote) files. This section presents an opportunity cost algorithm for reducing the I/O-style communication overhead. The general case of interprocess communication overhead is presented in the next section.

3.1 The Communication Model

In general, communication overhead can be represented by three components: CPU overhead, network bandwidth, and network latency [14]. The first component is due to the CPU processing time for sending or receiving messages. In this section, we develop an opportunity cost algorithm for minimizing the CPU communication overhead. For completeness, we also analyze the performance of the greedy algorithm for this case.

The algorithms that we develop do not deal with the network latency since it can be reduced using machine scheduling techniques, such as overlapping communication with computation [3], [31], [37]. We note that the bandwidth component can also be handled by the opportunity cost approach as an additional resource, as explained in Section 2.4.

3.2 Statement of the Problem

Consider a computational model in which a job is characterized by its computational load, called the *CPU weight*, and by its CPU *communication overhead vector*. In this vector, each element corresponds to the CPU communication overhead to other jobs. The *overall machine load* is the sum of the CPU weights and communication overheads of all its jobs. For simplicity, we assume that internal communication, within the same machine is negligible.

The problem of online assignment of I/O jobs to related machines is defined as follows: given a cluster $\mathcal{M}$ with n machines and a sequence of independent jobs that arrive at arbitrary times. A machine i is defined by its *CPU speed* $v(i)$. A job j is defined by its *CPU weight* $w(j)$, and *communication overhead vector* $\vec{o}(j) = (o_1(j), o_2(j), \cdots, o_n(j))$, where $o_i(j)$ is the CPU overhead of communication between job j and machine i. When a job j arrives, it is assigned online to some machine i. This increases the *overall CPU load* of machine $k, k \in \mathcal{M}$, by $p^i_k(j)$, defined as follows:

$$p^i_k(j) = \begin{cases} w(j)/v(i) + \sum_{i' \neq i} o_{i'}(j)/v(i) & \text{if } i = k, \\ o_k(j)/v(k) & \text{otherwise.} \end{cases}$$

Similarly, when a job terminates, the loads of all machines are decreased accordingly. The goal of the algorithm is to minimize the maximal *CPU load* among all the machines.

Note that the algorithms assume that the maximum CPU load of the optimal offline algorithm OPT is known and that the values of the *CPU weight* $w(j)$ and *communication overhead vector* $\vec{o}(j)$ are scaled down, i.e., divided by OPT. When the value of OPT is unknown, it can be approximated using a doubling technique, as explained in Theorem 5.

3.3 An Opportunity Cost Algorithm for I/O Jobs

This section presents an opportunity cost algorithm for permanent jobs and an extend algorithm for temporary jobs.

3.3.1 Permanent I/O Jobs

When an I/O job is assigned to one machine, it increases the CPU load in a set of other machines. The method used by the opportunity cost algorithm is to assign an arriving job in order to minimize the sum of the marginal costs of all the affected machines.

More formally, assume that the weights are scaled such that the maximal overall CPU load of the optimal offline algorithm OPT is 1. Define $a = 1 + \gamma/2$ for some constant $\gamma, 0 < \gamma < 1$. Then, the marginal cost of assigning a job j to machine i is:

$$H_i(j) = \sum_{k \in \mathcal{M}} a^{l_k(j)+p_k^i(j)} - a^{l_k(j)}, \tag{2}$$

where $l_k(j)$ is the overall CPU load of machine k just before the assignment of job j.

Algorithm COST-IO. Assign an arriving job j to machine i that minimizes the marginal cost $H_i(j)$.

Theorem 8. *Algorithm* COST-IO *is* $O(\log n)$ *competitive for permanent I/O jobs.*

Proof. The proof is based on that of the virtual circuits routing algorithm [7]. First, we define a *stability condition* and then show that as long as the stability condition is satisfied, the maximal load is below $O(\log n)$. □

Definition 1. *Denote by* $h_k(j)$ *the overall CPU load on machine* k*, just before job* j *was last assigned to some machine* i*, and by* l_k *the current overall CPU load on machine* k*. Then, the algorithm is* stable *if for any job* j *and any machine* $i', i' \neq i$*, we have:*

$$\sum_{k \in \mathcal{M}} a^{h_k(j)+p_k^i(j)} - a^{h_k(j)} \leq 2 \sum_{k \in \mathcal{M}} a^{l_k+p_k^{i'}(j)} - a^{l_k}. \tag{3}$$

Lemma 1. *For permanent I/O jobs, algorithm* COST-IO *is always stable.*

Proof. The algorithm starts in a stable state. When a new job j arrives, it is assigned to machine i with minimal $H_i(j)$. Thus, this assignment satisfies the stability condition for job j. This assignment also does not impair the stability condition of previous jobs. Since these jobs were assigned before job j, the lefthand side of their stability condition is not affected, while the righthand side can only be increased. Thus, the algorithm remains stable after an arrival of a new job.

Lemma 2. *If algorithm* COST-IO *is stable, then*

$$\sum_{i \in \mathcal{M}} a^{l_i} \leq n/(1-\gamma).$$

Proof. Consider job j that is currently assigned to machine i. Assume that the offline algorithm assigns this job to machine i^*. From the stability condition, it follows that:

$$\begin{aligned}\sum_{k \in \mathcal{M}} a^{h_k(j)+p_k^i(j)} - a^{h_k(j)} &\leq 2 \sum_{k \in \mathcal{M}} a^{l_k+p_k^{i^*}(j)} - a^{l_k} \\ &= 2 \sum_{k \in \mathcal{M}} a^{l_k}(a^{p_k^{i^*}(j)} - 1).\end{aligned}$$

The fact that the load of the offline algorithm does not exceed 1 implies that

$$\forall j, k : 0 \leq p_k^{i^*}(j) \leq 1.$$

Since $a = 1 + \gamma/2$, we have:

$$\forall x \in [0,1] : 2(a^x - 1) \leq \gamma x.$$

Hence,

$$\sum_{k \in \mathcal{M}} a^{h_k(j)+p_k^i(j)} - a^{h_k(j)} \leq \gamma \sum_{k \in \mathcal{M}} a^{l_k} p_k^{i^*}(j).$$

Summing over all currently active jobs and exchanging the order of summation, we get:

$$\sum_j \sum_{k \in \mathcal{M}} a^{h_k(j)+p_k^i(j)} - a^{h_k(j)} \leq \gamma \sum_j \sum_{k \in \mathcal{M}} a^{l_k} p_k^{i^*}(j),$$

and

$$\sum_{k \in \mathcal{M}} \sum_j a^{h_k(j)+p_k^i(j)} - a^{h_k(j)} \leq \gamma \sum_{k \in \mathcal{M}} a^{l_k} \sum_j p_k^{i^*}(j).$$

The lefthand side is a telescopic sum for each machine k. The righthand side contains the term $\sum_j p_k^{i^*}(j)$, which is the load of the offline algorithm in machine k. Thus,

$$\sum_{k \in \mathcal{M}} (a^{l_k} - 1) \leq \gamma \sum_{k \in \mathcal{M}} a^{l_k} \cdot 1.$$

Since $\gamma < 1$, we get

$$\sum_{k \in \mathcal{M}} a^{l_k} \leq n/(1-\gamma),$$

as stated in Lemma 2. □

From Lemma 1 and Lemma 2 it follows that:

$$\max_{i \in \mathcal{M}} l_i \leq \log_a(n/(1-\gamma)) = O(\log n),$$

which proves the theorem. □

3.3.2 Temporary I/O Jobs

Algorithm COST-IO can be extended for temporary I/O jobs as follows: when some job terminates, the algorithm checks if the stability condition, defined in (3), holds. Each job j that does not satisfy this condition is reassigned to machine i' with minimal $H_{i'}(j)$, as defined in (2).

Theorem 9. *Algorithm* COST-IO *is $O(\log n)$ competitive for temporary I/O jobs. It reassigns each job at most $O(\log n)$ times.*

Proof. First, we show that if the stability condition holds when a job arrives, then the number of reassignments of that job is bounded. Then, we show that the stability condition always holds when a job arrives.

Lemma 3. *Assume that job j arrive when the algorithm is stable. Then, j is reassigned at most $O(\log n)$ times.*

Proof. Define

$$Z(j) = \min_{i \in \mathcal{M}} \sum_{k \in \mathcal{M}} (a^{p_k^i(j)} - 1).$$

Let z be the machine which yields $Z(j)$. Note that $Z(j)$ is a lower bound of the marginal cost associated with job j at any time.

By the assumption of the lemma, j arrive when the algorithm is stable. Since j is assigned to machine i with minimal $H_i(j)$, the marginal cost of j satisfies:

$$\begin{aligned} H_i(j) &= \sum_{k \in \mathcal{M}} a^{l_k(j)+p_k^i(j)} - a^{l_k(j)} \le \sum_{k \in \mathcal{M}} a^{l_k(j)+p_k^z(j)} - a^{l_k(j)} \\ &= \sum_{k \in \mathcal{M}} a^{l_k(j)}(a^{p_k^z(j)} - 1) \le n/(1-\gamma) Z(j). \end{aligned}$$

Each reassignment of j decreases its marginal cost by at least a factor of 2. Since the marginal cost is bounded from below by $Z(j)$, it follows that the number of reassignments of job j is bounded by $O(\log n)$. □

Lemma 4. *For temporary I/O jobs, algorithm* COST-IO *is always stable.*

Proof. The algorithm starts in a stable state. When a new job j arrives, it is assigned to machine i with minimal $H_i(j)$. Clearly, this assignment satisfies the stability condition for job j and does not impair the stability condition of the running jobs. When a job j terminates, it may cause several reassignments. By induction, the jobs that are reassigned arrived when the algorithm was stable. Lemma 3 implies that the number of reassignments is finite. Thus, a stable state is reached again. □

We conclude the proof of the theorem by noting that from Lemma 4 and Lemma 2:

$$\max_{i \in \mathcal{M}} l_i(t) \le \log_a(n/(1-\gamma)) = O(\log n).$$

From Lemma 3 and Lemma 4 it follows that the number of reassignments of each job is at most $O(\log n)$. □

3.4 A Greedy Algorithm for I/O Jobs

This section presents an analysis of a greedy algorithm for assigning I/O jobs. Informally, this algorithm assigns an arriving job to a machine which yields the minimal resulting load. As before, the load is composed of both the CPU weight and the CPU communication overhead.

Let j be an arriving job that is assigned to machine i. Let $l_k(j)$ be the load of machine k, just before the arrival of job j. Let $\mathcal{C}^i(j)$ denote the set of machines that job j communicate with, i.e., $\mathcal{C}^i(j) = \{k \in \mathcal{M} \mid p_k^i(j) > 0\}$.

Algorithm GREEDY-IO. Assign an arriving job j to machine i that minimizes:

$$\max_{k \in \mathcal{C}^i(j)} l_k(j) + p_k^i(j).$$

Note that, in case of a tie, i.e., there are several assignments with the same maximum resulting load, the algorithm compares the second largest resulting load, then the third values, etc. If all the values are equal, then the tie is broken randomly.

Theorem 10. *Algorithm* GREEDY-IO *is $O(\log n)$ competitive for I/O jobs and related machines.*

Proof. The key point of the proof is that, for any job j, the overall increase of the load, i.e., in all the machines, due to the assignment of job j by the greedy algorithm, is greater by at most a constant factor than that of the offline algorithm.

Lemma 5. *Let $W^i(j)$ be the overall increase of the load incurred by the assignment of job j to machine i:*

$$W^i(j) = \sum_{k \in \mathcal{M}} p_k^i(j) v(k).$$

Then, for any assignment of job j by the greedy algorithm to machine g, we have:

$$W^g(j) \le 3 W^f(j),$$

where the offline algorithm assigns job j to machine f.

Proof. Denote by $O_c(j)$ the communication overhead of job j to all the machines except g and f, i.e., $O_c(j) = \sum_{k \in \mathcal{M} \setminus \{g,f\}} o_k(j)$. Observe that:

$$W^g(j) = w(j) + 2O_c(j) + 2o_f(j)$$

and that:

$$W^f(j) = w(j) + 2O_c(j) + 2o_g(j).$$

Case 1. $o_f(j) = 0$. Then:

$$W^g(j) = w(j) + 2O_c(j) \le w(j) + 2O_c(j) + 2o_g(j) = W^f(j).$$

Case 2. $o_f(j) > 0$. The fact that algorithm GREEDY-IO assigned job j to machine g implies that:

$$\max_{k \in \mathcal{C}^g(j)} l_k(j) + p_k^g(j) \le \max_{k \in \mathcal{C}^f(j)} l_k(j) + p_k^f(j),$$

with appropriate tie break. If the resulting overall CPU utilization on all the machines except g and f is the same, either j is assigned to g or to f. Therefore:

$$\max_{k \in \{g,f\}} l_k(j) + p_k^g(j) \le \max_{k \in \{g,f\}} l_k(j) + p_k^f(j). \tag{4}$$

From (4), we deduce that either:

$$l_g(j) + p_g^g(j) \le l_g(j) + p_g^f(j), \tag{5}$$

or

$$l_f(j) + p_f^g(j) \le l_f(j) + p_f^f(j). \tag{6}$$

Assuming that (5) holds we get $w(j) + O_c(j) + o_f(j) \leq o_g(j)$, thus:

$$W^g(j) = w(j) + 2O_c(j) + 2o_f(j) \\ \leq 2o_g(j) - w(j) \leq w(j) + 2O_c(j) + 2o_g(j)) = W^f(j).$$

Otherwise, (6) holds and we get $o_f(j) \leq w(j) + O_c(j) + o_g(j)$, thus:

$$W^g(j) = w(j) + 2O_c(j) + 2o_f(j) \\ \leq 3w(j) + 4O_c(j) + 2o_f(j) \leq 3W^f(j).$$

□

The rest of the proof of the theorem is a refinement of the proof in [6] on the competitive ratio of the GREEDY algorithm for load balancing of related machines. First, it can be shown that if the load on all the machines with speed $\geq v$ is at least l, then the load on all the machines with speed $\geq v/2$ is at least $l - 6L^*$, where L^* is the maximum load of the offline algorithm.

Let V be the speed of the fastest machines. Using Lemma 5, it can be shown that the load on the fastest machines is at least $L - 3L^*$, where L is the maximum load of the greedy algorithm. By iteration, all the machines with speed at least $V/2^i$ have load at least $L - 3L^* - 6iL^*$. Thus, every machine with speed at least V/n has load at least $L - (3 + 6\lceil\log n\rceil)L^*$. Using a similar argument to [6], it can be shown that $L - (3 + 6\lceil\log n\rceil)L^* \leq 6L^*$, which implies $L = O(\log n)L^*$. □

4 Jobs with Interprocess Communication Overhead

This section presents an opportunity cost algorithm for reducing the Interprocess Communication (IPC) overhead. In our computing model, a parallel job consists of several communicating processes, with an arbitrary communication topology. Processes are assigned online to machines in order to minimize the maximal load among the machines. The details of the communication model are as described in Section 3.1.

4.1 Statement of the Problem

Consider a parallel job J, represented by a graph $G_J = (V_J, E_J)$, whose vertices V_J represents the job's processes, and its edges E_J, represent pairs of communicating processes. Denote by $w_J : V_J \to \mathcal{R}^+$ the *CPU weight* of the processes of job J, and by $o_J : E_J \to \mathcal{R}^+$ the *CPU communication overhead* of all its pairs of communicating processes. Let $\mathcal{A}_J : V_J \to \mathcal{M}$ be an *assignment* of job J to cluster $\mathcal{M}$ with n machines. In what follows, the index J is omitted when it is obvious.

The problem of online assignment of a parallel job to related machines is defined as follows: given a cluster $\mathcal{M}$, where machine i has *CPU speed* $v(i)$. Assume that a sequence of parallel jobs arrive at arbitrary times and are assigned online. Each assignment $\mathcal{A}_J$ increases the load of machine $k, k \in \mathcal{M}$, by $p_k^{\mathcal{A}}(J)$, as follows:

$$p_k^{\mathcal{A}}(J) = \sum_{j \in V | \mathcal{A}(j)=k} w(j)/v(k) + \sum_{e \in E | e=(s,t), \mathcal{A}(s)=k, \mathcal{A}(t) \neq k} o(e)/v(k).$$

Similarly, when the job terminates, the loads of all the machines are decreased accordingly. The goal of the algorithm is to minimize the maximal *CPU load* among the machines.

Note that the algorithms assume that the maximum CPU load of the optimal offline algorithm OPT is known and that the values of the *CPU weight* w_J and *communication overhead* o_J are scaled down, i.e., divided by OPT. When the value of OPT is unknown, it can be approximated using a doubling technique, as explained in Theorem 5.

4.2 An Opportunity Cost Algorithm for IPC Overhead

This section presents an opportunity cost algorithm for permanent parallel jobs and an extended algorithm for temporary parallel jobs. The section begins by a proof that any online algorithm that assigns one process at a time, e.g., algorithm COST-IO described in Section 3.3, does not perform well in this case.

Theorem 11. *The competitive ratio of any online algorithm that assigns communicating processes sequentially without reassignments is at least n.*

Proof. Let the input sequence contain n^2 processes with weight 1. If the online algorithm assigns these processes to a single machine then the resulting maximal load is n^2. The offline algorithm assigns in this case n processes to each machine, thus its maximal load is n. Assume that the online algorithm does not assign the processes to a single machine. Assume that an additional process arrives, where this process communicates with two processes, which are assigned to different machines, with a communication overhead of n^2 to each process. After the assignment of the additional process, the load of the machine to which it is assigned is at least n^2 since it performs remote communication with at least one of the above two processes. For this case, the offline algorithm allocates the two processes to one machine, then assign the additional process to that machine too. Thus, its resulting maximal load is $n + 1$. □

4.2.1 *Permanent Parallel Jobs*

An assignment of a parallel job increases the load of a set of machines to which the job's processes are assigned. Like algorithm COST-IO, the marginal cost of such an assignment is the sum of the marginal costs of all the affected machines.

More formally, assume that the weights are scaled such that the maximal overall CPU load of the optimal offline algorithm OPT is 1. Define $a = 1 + \gamma/2$ for some constant $\gamma, 0 < \gamma < 1$. Then, the marginal cost of an assignment $\mathcal{A}$ of job J is:

$$H_{\mathcal{A}}(J) = \sum_{k \in \mathcal{M}} a^{l_k(J)+p_k^{\mathcal{A}}(J)} - a^{l_k(J)}, \tag{7}$$

where $l_k(J)$ is the load of machine k, just before the assignment of job J.

Algorithm COST-PJ. Assign an arriving job J to minimize $H_{\mathcal{A}}(J)$.

Theorem 12. *Algorithm* COST-PJ *is* $O(\log n)$ *competitive for permanent parallel jobs.*

Proof. The proof is similar to the proof of algorithm COST-IO, except that instead of assigning an I/O jobs to machine i, algorithm COST-PJ assigns a parallel job according an assignment $\mathcal{A}$. □

4.2.2 Temporary Parallel Jobs

For temporary parallel jobs, algorithm COST-PJ is extended such that after each job termination, the algorithm checks if the a stability condition similar to (3) holds. Each job J that does not satisfy this stability condition is reassigned according to assignment $\mathcal{A}'$ with minimal $H_{\mathcal{A}'}(J)$.

Theorem 13. *Algorithm* COST-PJ *is* $O(\log n)$ *competitive for temporary parallel jobs. It reassign each job at most* $O(\log n)$ *times.*

Proof. The proof is similar to the proof of algorithm COST-IO for temporary I/O jobs, except that instead of assigning an I/O job to a machine i, algorithm COST-PJ assigns a parallel job according to an assignment $\mathcal{A}$. □

4.3 A Greedy Algorithm for Parallel Jobs

A greedy algorithm assigns a newly arriving parallel job in order to minimize the maximal load among all the machines. More formally, let J be an arriving job, and let $l_k(J)$ be the load of machine k, just before this arrival. Denote by $\mathcal{C}^{\mathcal{A}}(J)$ the set of machines that are affected by an assignment $\mathcal{A}$ of job J, i.e.:

$$\mathcal{C}^{\mathcal{A}}(J) = \{k \in \mathcal{M} \mid p_k^{\mathcal{A}}(J) > 0\}.$$

Algorithm GREEDY-PJ. Assign an arriving job J to minimize:

$$\max_{k \in \mathcal{C}^{\mathcal{A}}(J)} l_k(J) + p_k^{\mathcal{A}}(J). \tag{8}$$

Theorem 14. *Algorithm* GREEDY-PJ *is at least* $O(n)$ *competitive for parallel jobs and related machines.*

Proof. Let the input sequence contain n parallel jobs, each with n processes of weight $1+\epsilon$. The communication graph of each job is a clique with a communication overhead 1. The offline algorithm assigns each job to one machine, yielding a maximal load of $n(1+\epsilon)$. The online algorithm distributes the processes of each job evenly, among all the machines, yielding a load of $n+\epsilon$ for each job. Thus, its maximal load, after the assignment of all the jobs is $n(n+\epsilon)$. □

4.4 The Minimal Cost Assignment

In this section, we explain how algorithm COST-PJ and algorithm GREEDY-PJ were implemented. These algorithms are difficult to implement because they require a method for solving the following minimization problem:

MIN-COST-ASSIGNMENT. Given a weighted graph, find an assignment $\mathcal{A}$ of its vertices to n machines, to minimize a cost function $C(\mathcal{A})$, where $C(\mathcal{A})$ is defined either by (7) for Algorithm COST-PJ, or by (8) for Algorithm GREEDY-PJ.

MIN-COST-ASSIGNMENT is a variant of the *k*-PARTITION problem. In the latter, the goal is to partition the vertices of a graph into k equally weighted sets, to minimize the sum of the weights of the edges crossing between the sets. This problem is NP-hard, even for approximation [4]. Therefore, its solutions are based on heuristic methods, such as Kerninghan-Lin (KL) [27], spectral bisectioning [38], multilevel schemes [25], or genetic algorithms [30]. These methods are used extensively in the fields of parallel computing [39].

We implemented a heuristic for solving the MIN-COST-ASSIGNMENT problem. We note that the communication graph is usually small because the number of processes is roughly equal to the number of machines. Thus, it can be solved efficiently by using a two-phase algorithm, where the first phase constructs a coarse initial assignment, which is then refined, e.g., by means of a KL-style heuristic [23].

The first phase of the algorithm assigns the processes of a parallel job sequentially, minimizing the cost to previously assigned processes. This is sufficient to produce a coarse initial assignment.

The second phase of the algorithm consists of successively choosing pairs of machines, and applying a two-way refinement algorithm on these pairs, until stabilization occurs. The two-way refinement algorithm is intended to reduce the cost function for a pair of machines by moving processes between these machines in a manner similar to the KL heuristic.

5 Validation by Simulations and Real Executions

This section presents the results of simulations and real executions of several job streams in a "typical" computing cluster, in order to evaluate the performance of the opportunity cost algorithms and also to compare it to other schemes for online job assignments.

5.1 Simulations

We developed a simulator of a computing cluster consisting of machines with different speeds and memory sizes. We simulated time-sharing environments, in which each machine can execute several jobs concurrently.

In each simulation, the input consists of a sequence of arriving jobs, each with a randomly chosen start time, amount of work, number of processes, memory, and communication overhead. In particular, we assume that, in the simulation, all the processes in a jobs are always ready to run.

The simulations were performed for four different assignment policies using an identical input sequence of jobs. The four policies were:

1. **COST.** The opportunity cost policy with no reassignments.

2. **COST-P.** The opportunity cost policy with reassignments
3. **GREEDY.** A greedy policy with no reassignments.
4. **GREEDY-P.** A greedy policy with reassignments.

The algorithm that we used in all these policies were a straightforward derivation of the theoretical algorithms described in the previous sections.

To evaluate the effectiveness of the policies, we used the mean job *slowdown* (of all the jobs in a sequence) as a performance measure. The slowdown of a job is defined as the ratio between the job's actual execution time and the job's optimal execution time without any resource limitations, i.e., unlimited number of machines, unlimited memory, overhead-less communication [28].

We note that we also checked the performance of the Round-Robin policy and found that it performed much poorer than all the other policies. For example, in some cases, the slowdown of Round-Robin was almost two orders of magnitude larger than that of the other policies. Due to this fact our simulations do not include the Round-Robin policy.

5.1.1 Simulation of Sequential Jobs

In these simulations the input sequence consists of $50n$ single-process jobs, where n is the number of machines in the cluster.

The following assumptions are made: first, we assume that a job requires $1/r$ seconds of CPU time units, where r is a random number chosen uniformly between 0 and 1. We note that this distribution is based on traces of real workloads, as reported in [22]. The interarrival times of the jobs are exponentially distributed with a mean value of $4/n$ seconds.

We also assume that each job performs I/O to one machine, chosen at random with uniform distribution. The CPU overhead due to this I/O is chosen randomly using an exponential distribution. For example, if the I/O overhead is 0.1, then for each unit of CPU work the job spends 0.1 CPU units for I/O-style communication. Note that there is no overhead associated with I/O operations in a local machine.

The memory usage of a job is set to m/r of the installed machine's memory, where r is a random number chosen uniformly between 0 and 1, and $0 \leq m \leq 0.1$. The speed and memory of machines in the cluster are randomly chosen with uniform distribution between 1 and 2, and 1 and 3, respectively.

The first set of simulations shows the effect of the I/O overhead on the performance of the four different policies. The cluster consists of $n = 32$ machines. The job's mean memory usage is 0.01, while its I/O overhead is gradually increased from 0, i.e., no I/O, to 0.5. We note that the above choice of a small mean memory usage is made in order to avoid paging during the executions.

Fig. 2 presents the mean slowdown versus I/O overhead, where each measurement is averaged over 200 different simulations. Observe that, for CPU-bound jobs with low I/O overhead, the four policies have almost identical performances, with slightly better results for the policies which perform reassignments. For increasing values of

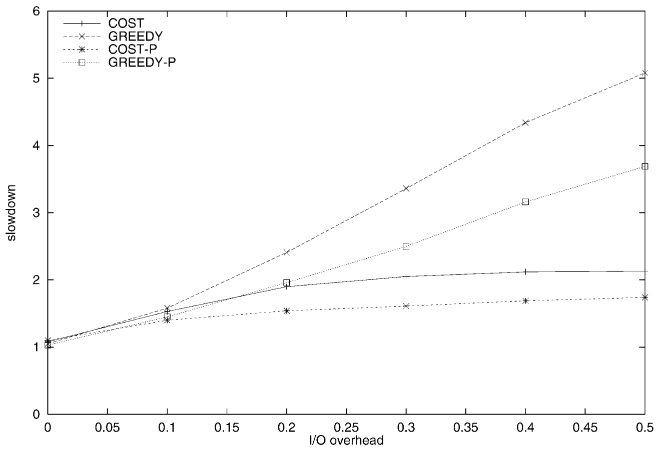

Fig. 2. I/O jobs: mean slowdown vs. I/O overhead.

I/O overhead, i.e., higher than 0.2, it can be seen that both of the opportunity cost policies outperform the greedy policies, with almost 100 percent improvement for an I/O overhead of 0.4 or higher. Also, by comparing **COST-P** to **COST** and **GREEDY-P** to **GREEDY**, it is interesting to note that reassignments improve the mean slowdown by as much as 25 percent. This last result indicates that preemptive process migration could substantially improve the performance [11], [22].

5.1.2 Simulation of Parallel Jobs

In the simulations of parallel jobs, the input sequence contains $10n$ parallel jobs, where n is the number of machines in the cluster. Each parallel job contains a random number of processes chosen uniformly between 1 to n. The processes communicated along a ring or a torus topology, with equal probability for each topology. The communication overhead between two communicating processes is exponentially distributed, with mean varying from 0 to 0.5, for different cases. For simplicity, we assume the same communication overhead between each pair of communicating processes. The memory usage of all the processes of each parallel job is equal, chosen as in the case of the sequential jobs. The amount of work of each process is $(1/r)$ CPU time units, where r is a random number chosen uniformly between 0 and 1. The jobs arrival times are exponentially distributed with mean 10 seconds.

Fig. 3 presents the slowdown as a function of the communication overhead, gradually increased from 0 (no IPC) to 0.5 (heavy IPC). We assumed 16 machines in the cluster and mean memory usage of 0.01. The simulation results are averaged over 200 executions.

From the figure, it can be seen that, as the communication overhead is increased, the opportunity cost policy with reassignments consistently outperforms the other three policies. For example, for an overhead of 0.5, **COST-P** is almost 20 percent better than **GREEDY** and 10 percent better than **COST**.

5.1.3 Summary of the Simulation Results

The slowdown results of the simulations of I/O jobs and parallel jobs are summarized in Tables 1 and 2, respectively. Each table presents 2 cases, pure CPU jobs and CPU jobs

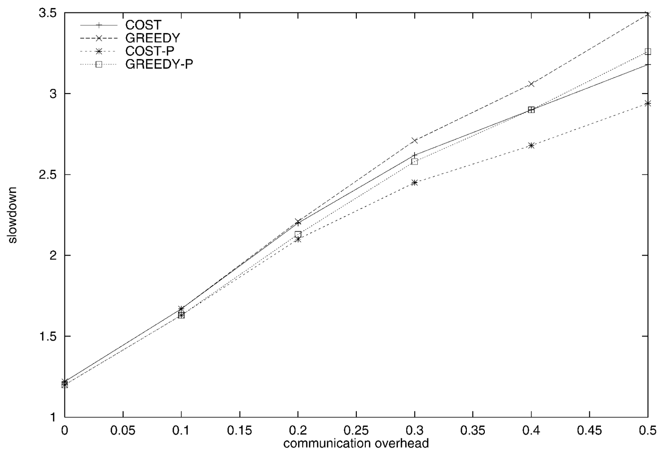

Fig. 3. Parallel jobs: mean slowdown vs. communication overhead.

with 0.5 I/O or communication overhead. Each value in the tables corresponds to the measured results shown in Fig. 2 and Fig. 3, respectively. All the presented values are scaled such that the slowdown of **COST-P** is 1.0.

The results presented in the tables shows that all four policies perform almost equally for pure CPU jobs, except that **GREEDY-P** is 6 percent better for sequential jobs. With the addition of I/O or communication overhead, the opportunity cost policies almost always outperform the greedy policies, with some cases by as much as 100-200 percent. The tables also show the superiority of the preemptive polices over the nonpreemptive polices.

5.2 Executions on a Real System

This section presents the performance of **COST-P**, **GREEDY-P**, and the MOSIX policy, using a real computing cluster. Due to the complexity of the implementation we tested only sequential jobs.

The execution cluster consists of six machines, two Pentium-Pro 200MHz, two Pentium-II 300MHz, and two Pentium-II 400MHz, with memory sizes ranging from 64MB to 256MB. The machines where connected by a Fast-Ethernet switch. The executions were performed using a modified version of MOSIX [9], [12], a cluster computing enhancement of UNIX that supports preemptive process migration.

5.2.1 COST-P vs. GREEDY-P

In this test, we replaced the resource sharing policy of MOSIX with the **COST-P** and the **GREEDY-P** policies respectively. We note that both algorithms sum the cost of CPU utilization and I/O overhead of the process. In the case of the **COST-P**, the algorithm migrates the process to a machine which minimizes the process marginal cost, as explained in Section 3.3. The **GREEDY-P** algorithm migrates the process to a machine which yields the minimal resulting load, as explained in Section 3.4.

We conducted a set of tests for jobs with I/O overhead. Each test consists of arriving jobs, with interarrival and CPU times exponentially distributed with mean 2 and 7 seconds, respectively. The tests were repeated 50 times, with different sequences of jobs.

Fig. 4 presents the execution results of jobs with I/O overhead. In these executions, the I/O overhead is exponentially distributed with mean ranging from 0 to 1.05.

From the figure, it can be seen that, with increased values of I/O overhead, the opportunity cost policy outperforms the greedy policy. For example, for an I/O overhead of 0.7, i.e., for every second of CPU the job waits 0.7 seconds for I/O, the average slowdown of **COST-P** is about 20 percent lower than that of **GREEDY-P**. Note that, for pure CPU bound jobs, **GREEDY-P** is slightly better, i.e., about 9 percent, than **COST-P**. This result confirms the simulations results presented in Table 1.

5.2.2 The MOSIX Policy

In this test, we compared the opportunity cost policies to the fine-tuned, heuristic policy of MOSIX. For this set of tests, we executed the same sets of sequential jobs as in the previous section using the original MOSIX policy.

Briefly, the MOSIX resource sharing policy continuously attempts to reduce the CPU load differences between pairs of machines, by migrating processes from over-loaded machines to less loaded machines. MOSIX also uses a

TABLE 1
Relative Slowdown of Sequential Jobs

	Type of Jobs	
Algorithm	CPU	0.5 I/O
COST-P	1.00	1.00
COST	0.98	1.22
GREEDY-P	0.94	2.12
GREEDY	0.96	2.92

TABLE 2
Relative Slowdown of Parallel Jobs

	Type of Jobs	
Algorithm	CPU	0.5 IPC
COST-P	1.00	1.00
COST	1.02	1.08
GREEDY-P	1.00	1.11
GREEDY	1.02	1.19

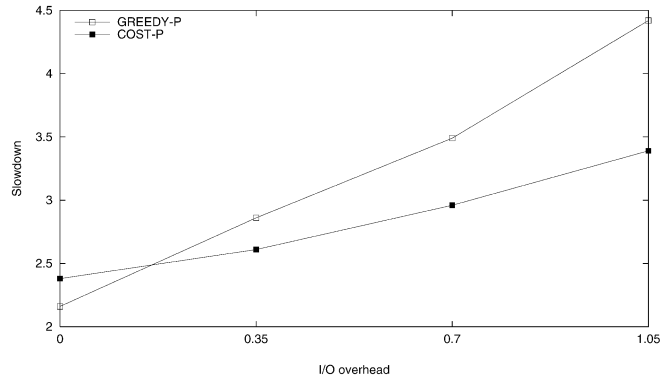

Fig. 4. I/O jobs: mean slowdown vs. I/O overhead.

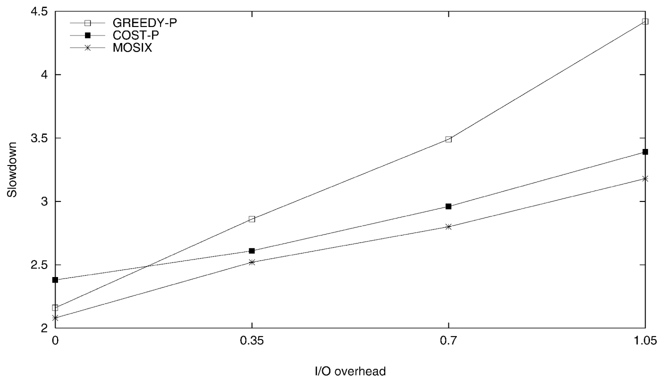

Fig. 5. I/O jobs: mean slowdown vs. I/O overhead.

memory ushering algorithm, which migrates processes from machines that has exhausted their main memory to machines that has free memory. Another heuristics performed by MOSIX is that it chooses the best target machine for each migrating process. This decision is based on gathered information by the system, in order to minimizes the expected remaining process run time. For example, MOSIX takes into account the migration time based on the process size, past CPU time, amount of I/O performed, etc. These heuristics were verified by studies of process and system behavior [10], [22].

Fig. 5 presents the slowdown for jobs with I/O overhead using the **COST-P** and the **GREEDY-P** policies, previously shown in Fig. 4, and the MOSIX policy. From the figure, it can be seen that the MOSIX policy is consistently better than the other two policies with an average of 6 percent improvement over **COST-P**.

To explain this last result, we note that the MOSIX policy converts all migration decisions into time units, assuming a past-repeat estimate on the expected remaining execution time of the job. This assumption was verified empirically in [22]. This method enables MOSIX to include in its considerations variables that can only be expressed in time units, e.g., migration cost, network latency, etc. The load-balancing theory, on the other hand, does not deal with such time-dependent variables.

6 Conclusions

The opportunity cost approach provides a unifying framework for the management of multiple heterogeneous resources in a computing cluster by the assignment and reassignment of jobs. The algorithms of this approach apply a pricing scheme as a way to reduce congestion on different resources. Each resource is given a cost, which is an exponential function of its utilization. Each job is assigned to a machine in a cluster to minimize the sum of the marginal costs of its resources.

In this paper, we developed opportunity cost algorithms for online assignment of sequential and parallel jobs to machines. These algorithms are geared to reduce the maximal utilization of the CPU resource for jobs that perform IPC and I/O operations. We have proven that the resulting performance is guaranteed to be within $O(\log n)$ of the performance of the optimal, offline algorithm. We showed by means of simulations and executions in a real system that the opportunity cost algorithms provide better performance than the widely used greedy methods.

The method presented in this paper is general and could be used to manage other resources, such as memory, communication bandwidth, system tables, buffers, channels, semaphores, etc. It can also be used in clusters of heterogeneous machines or symmetrical multiprocessors.

The work presented in the paper could be extended in several directions. First, we used a centralized method for job assignment and reassignment. It is interesting to explore a method that uses a decentralized scheme, preferable by using only partial information about the system state. Such a scheme is more suitable for large clusters. A preliminary study of such a method is presented in [26], where it is shown that a distributed scheme achieved similar performance to the centralized scheme, with a slight increase in the number of reassignments.

Another extension is to include time-dependent variables, e.g., migration cost or network latency, as already done by the MOSIX policy.

Acknowledgments

The authors would like to thank Y. Azar and A. Shiloh for many helpful discussions, R. Lavi for conducting the executions on the MOSIX system, and R. Chermoni for his help. This research was supported in part by the Ministry of Defense, the Ministry of Science, and by a grant from Dr. and Mrs. Silverston, Cambridge, UK.

References

[1] Y. Amir, B. Awerbuch, A. Barak, R. Borgstrom, and A. Keren, "An Opportunity Cost Approach for Job Assignment and Reassignment in a Scalable Computing Cluster," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 7, pp. 760–768, July 2000.

[2] T.E. Anderson, D.E. Culler, and D.A. Patterson, and the NOW team, "A Case for NOW," *IEEE Micro,* vol. 15, no. 1, pp. 54–64, Feb. 1995.

[3] M. Andrews, T. Leighton, P.T. Metaxas, and L. Zhang, "Automatic Methods for Hiding Latency in High Bandwidth Networks," *Proc. 28th ACM Symp. Theory Of Computing (STOC'96),* pp. 257–265, 1996.

[4] S. Arora and C. Lund, "Hardness of Approximations," *Approximation Algorithms for NP-hard Problems,* D.S. Hochbaum, ed., pp. 399–446, 1997.

[5] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, D.E. Culler, J.M. Hellerstein, and D.A. Patterson, "High-Performance Sorting on Networks of Workstations," *Proc. 1997 ACM SIGMOD Int'l Conf. Management of Data,* pp. 243–254, 1997.

[6] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts, "On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling," *J. the ACM,* vol. 44, no. 3, pp. 486–504, 1997.

[7] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts, "Competitive Routing of Virtual Circuits with Unknown Duration," *Proc. Fifth ACM-SIAM Symp. Discrete Algorithms (SODA '94),* pp. 321–327, 1994.

[8] Y. Azar, "On-Line Load Balancing," *Online Algorithms: The State of Art,* A. Fiat and G. Woeginger, eds., 1998.

[9] A. Barak, "MOSIX—Scalable Cluster Computing for Unix," http://www.mosix.org, 2002.

[10] A. Barak and A. Braverman, "Memory Ushering in a Scalable Computing Cluster," *Microprocessors and Microsystems,* vol. 22, nos. 3–4, pp. 175–182, Aug. 1998.

[11] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *J. Future Generation Computer Systems,* vol. 13, nos. 4–5, pp. 361–372, 1998.

[12] A. Barak, O. La'adan, and A. Shiloh, "Scalable Cluster Computing with MOSIX for LINUX," *Proc. Fifth Ann. Linux Expo,* pp. 95–100, May 1999.

[13] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive Parallelism and Piranha," *Computer,* vol. 28, no. 1, pp. 40–49, 1995.

[14] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken, "LogP: A Practical Model of Parallel Computation," *Comm. ACM,* vol. 39, no. 11, pp. 78–85, Nov. 1996.

[15] X. Deng, N. Gu, T. Brecht, and K. Lu, "Preemptive Scheduling of Parallel Jobs on Multiprocessors," *Proc. Seventh ACM-SIAM Symp. Discrete Algorithms (SODA '96),* pp. 159–167, 1996.

[16] A. Fiat and S. Leonardi, "On-Line Routing," *Online Algorithms: The State of Art,* A. Fiat and G. Woeginger, eds., 1998.

[17] *Online Algorithms: The State of Art,* A. Fiat and G. Woeginger, eds., 1998.

[18] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[20] D.P. Ghormley, D. Petrou, S.H. Rodrigues, A.M. Vahdat, and T.E. Anderson, "GLUnix: a Global Layer Unix for a Network of Workstations," *Software—Practice & Experience,* vol. 28, no. 9, pp. 929–961, 1998.

[21] R. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Applied Mathematics,* vol. 17, pp. 263–269, 1969.

[22] M. Harchol-Balter and A. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Trans. Computer Systems,* vol. 15, no. 3, pp. 253–285, 1997.

[23] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proc. Supercomputing '95,* 1995.

[24] IBM Corp, *LoadLeveler—Efficient Job Scheduling and Management.* 2001.

[25] G. Karypis and V. Kumar, "Multilevel *k*-Way Partitioning Scheme for Irregular Graphs," *J. Parallel and Distributed Computing,* vol. 48, no. 1, pp. 96–129, 1998.

[26] A. Keren, "On-Line Assignment of Processes in a Scalable Computing Cluster," PhD thesis, The Hebrew Univ. of Jerusalem, Israel, 1998.

[27] B. Kerninghan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical J.,* vol. 49, no. 2, pp. 291–307, 1970.

[28] O. La'adan and A. Barak, "Inter Process Communication Optimization in a Scalable Computing Cluster," *Ann. Rev. of Scalable Computing,* pp. 121–180, 1999.

[29] J. MacKie-Mason and H. Varian, "Pricing Congestible Network Resources," *IEEE J. Selected Areas in Comm.,* vol. 13, no. 7, pp. 1141–1149, Sept. 1995.

[30] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka, "Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning," *Proc. Supercomputing '94,* pp. 449–457, 1994.

[31] R.P. Martin, A.M. Vahdat, D.E. Culler, and T.E. Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* 1997.

[32] V.K. Naik, S.K. Setia, and M.S. Squillante, "Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems," *J. Parallel and Distributed Computing,* vol. 46, pp. 28–47, 1997.

[33] M. Nuttall, "Survey of Systems Providing Process or Object Migration," *Operating System Rev.,* vol. 28, no. 4, pp. 64–79, 1994.

[34] P. Pacheco, *Parallel Programming with MPI,* Morgan Kaufmann, 1996.

[35] D. Sleator and R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Comm. ACM,* vol. 28, no. 2, pp. 202–208, 1985.

[36] A.D. Stoyenko, J. Bosch, M. Aksit, and T.J. Marlowe, "Load Balanced Mapping of Distributed Objects to Minimize Network Communication," *J. Parallel and Distributed Computing,* vol. 34, pp. 117–135, 1996.

[37] V. Strumpen, "Software-Based Communication Latency Hiding for Commodity Workstation Networks," *Proc. 1996 Int'l Conf. Parallel Processing (ICPP '96),* 1996.

[38] R. Van Driessche and D. Roose, "An Improved Spectral Bisection Algorithm and its Application to Dynamic Load Balancing," *Parallel Computing,* vol. 21, pp. 29–48, 1995.

[39] C. Walshaw, M. Cross, M.G. Everett, S. Johnson, and K. McManus, "Partitioning & Mapping of Unstructured Meshes to Parallel Machine Topologies," *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems,* pp. 121–126, 1995.

Arie Keren received the BSc degree in computer engineering from the Technion, Heifa in 1983, the MSc degree in computer science from Tel-Aviv University in 1991, and the PhD degree in computer science from the Hebrew University of Jerusalem in 1998. His research interests include parallel and distributed systems, software engineering, and algorithms.

Amnon Barak received the BS degree in mathematics from the Technion, and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign. He is the S&W Strauss Professor of Computer Science and the director of the Laboratory for Distributed Computing in the Institute of Computer Science at The Hebrew University of Jerusalem. He is the developer of the MOSIX load-balancing cluster computing system. His current research interests include parallel and distributed systems, operating systems for scalable clusters, dynamic resource allocation, parallel I/O, high availability, and competitive algorithms for efficient resource utilization in clusters.